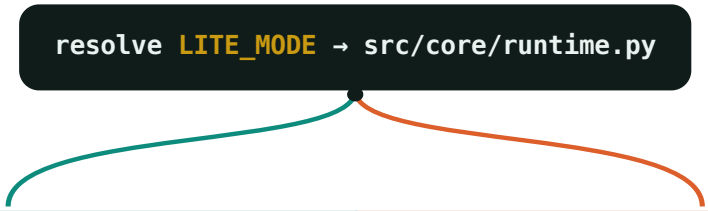
● SELF-HOSTED · V0.1.0

One flag.
Two runtimes.

Fagoon is a self-hostable platform for AI agents and visual workflows. It reads a single environment variable at startup and resolves into one of two topologies — a zero-dependency single-user install, or a Redis-backed distributed deployment. The application code never knows which one it's running in.

FastAPI · Python 3.12+ | PostgreSQL + pgvector | Celery / Inline async | SSE · WebSockets



● Lite

Default package install · one user · zero env required

Rate limiter	in-memory
Cache	in-memory TTL
Task queue	inline asyncio
Workers	forced to 1
Redis / Celery	none

● Full

Contributors & production · concurrent · multi-worker

Rate limiter	Redis
Cache	Redis
Task queue	Celery + eventlet
Workers	concurrent
Broker	Redis

— QUICK START

Two ways to run it

Package users get an install that behaves like a desktop app — one command, no configuration. Contributors clone the repo and run the full stack with Redis and a Celery worker.

Package users — one command

Install the CLI, then bring the stack up. Lite mode needs no Redis, no Celery, and no environment file. Secrets are generated on first boot and the database URL and config live in `~/.fagoon`.

bash

```
$ pipx install fagoon
$ fagoon up                 # lite mode: no Redis, no Celery, zero env
$ fagoon up --ollama        # add the local Gemma fallback LLM
```

Open http://localhost:8000 and complete the setup wizard — an LLM key and the features you want enabled.

Contributors — clone and run

Copy the example environment file, fill in the minimal values, and bring up the full Compose stack. This builds from source and runs Redis plus a Celery worker alongside the app and database.

bash

```
$ git clone https://github.com/<owner>/fagoon
$ cp .env.example .env      # fill minimal values
$ docker compose -f deploy/docker-compose.full.yml up --build
```

pgvector means Postgres is mandatory in both modes. There is no SQLite path — the vector store for RAG lives inside Postgres via the pgvector extension, so the bundled or external database is always present.

TECH STACK

What it's built on

A layered FastAPI backend orchestrating AI providers, with PostgreSQL as both the relational store and the vector database.

CONCERN	CHOICE
Framework	`FastAPI · Python 3.12+`
Package management	`uv · pyproject.toml`
Primary database	`PostgreSQL · SQLAlchemy AsyncSession · asyncpg`
Vector database	`PostgreSQL via pgvector`
Migrations	`Alembic`
Background jobs	`Celery + Redis + eventlet · or inline asyncio`
Real-time	`Server-Sent Events · WebSockets`
LLM & AI integrations	`OpenAI · Anthropic · Google GenAI (Gemini/Veo) · Groq · Hugging Face · Fal AI · ElevenLabs · Kokoro`
Architecture pattern	`Routers → Services → Providers → Data/Storage`

CORE CONCEPT

The dual-mode runtime

Everything else in Fagoon follows from one design move: put an interface between the application and its infrastructure, then pick the backend once at startup. The flag is read a single time in `src/core/runtime.py`, which builds the right backends and attaches them to `app.state`.

Lite vs Full at a glance

CAPABILITY	LITE MODE	FULL MODE
Audience	Single self-hosted user	Contributors · production
Launch	`fagoon up`	`docker compose -f …full.yml up`
Rate limiter	`MemoryRateLimiter`	`RedisRateLimiter`
Cache	`MemoryCache`	`RedisCache`
Task queue	`InlineTaskQueue`	`CeleryTaskQueue`
Concurrency	Single process (WEB_CONCURRENCY=1)	Multi-worker
External deps	None beyond Postgres	Redis broker + Celery worker
Env required	Zero — secrets auto-generate	REDIS_URL required, fails loud if missing
Job durability	Job dies on restart	Survives via broker

Job status always flows through Postgres, regardless of backend. That's why the WebSocket status broadcaster is identical in both modes — only *where the work runs* changes, not how its progress is tracked.

THE ONE RULE

Business code never touches Redis or Celery

This is the contract the whole design depends on. Application code talks only to three abstractions on `app.state`. Only the backend modules, `runtime.py`, and `celery_app.py` may import the real packages.

`app.state.limiter`
`.allow(key, limit, window)`

Every rate-limit check routes here. Sliding window in a dict (lite) or a Redis sorted set (full).

`app.state.queue`
`.enqueue(fn, *args)`

Every background job is dispatched here. Inline asyncio task (lite) or a registered Celery task (full).

`app.state.cache`
`.get(key) / .set(key, v)`

Every cache operation routes here. In-process TTL dict (lite) or JSON-encoded Redis values (full).

⚠ **Enforced in CI, not by convention.** A direct `import redis` or `import celery` in business code would crash a lite install that has neither package. `import-linter` fails the build, a grep guard script catches string-level cases, and a lite-boot test installs *without* the full extra to prove the contract at the dependency level.

ARCHITECTURE

Four layers, strict boundaries

A clean separation that physically isolates routing, business logic, third-party clients, and bootstrapping. Each layer may only call inward.

API
`src/api/`

Request parsing, Pydantic validation, HTTP statuses, endpoint definitions, dependency injection. No business logic — routers delegate all heavy work to services. Domains: agents, authentication, external, misc, upgrade, video_gen, workflow, workspace.

Services
`src/services/`

Core business logic — orchestrating LLM steps, chat flows, task processing, database lookups. Stays mode-agnostic by talking to runtime abstractions. Houses tool_handlers, rag, google_workspace, channel_adapter, limiter, cache, taskqueue, and the database switch service.

Providers
`src/providers/` · `src/llms/`

Wrappers around external SDKs (OpenAI, Anthropic, Google GenAI, Groq, ElevenLabs). They normalize payloads, handle retries, and translate responses into standard internal objects.

Core `src/core/`	App bootstrapping, layered settings resolution, the runtime factory, the PostgresManager connection pool, and lifespan management. The only place mode branching happens.

Lifespan singletons

Heavy clients are initialized once during the FastAPI lifespan and attached to `app.state` — never instantiated per request: `postgres_manager`, `httpx_client`, `vector_store`, `agent_manager`, and `chat_orchestrator`.

REPOSITORY MAP

Folder structure

The tree below mirrors the layered architecture — routing, services, providers, and models live in physically separate places.

agents-workflow/

```
.github/workflows/     # ci.yml (tests + guards) · release.yml (multi-arch + PyPI)
alembic/               # database migrations & schema history
deploy/                # Dockerfile, entrypoint.sh, both compose files
fagoon_cli/            # Typer control CLI + bundled compose files
scripts/               # setup & import-guard helper scripts
src/
├── api/v1/            # routers/ (domain routers) + setup_api.py
│   └── custom_middleware.py   # Auth & CORS
├── core/              # bootstrap · globals · runtime · settings
├── models/sql/        # SQLAlchemy models → PostgreSQL
├── providers/         # OpenAI, Gemini, … client wrappers
├── schemas/           # Pydantic request/response validation
├── services/          # cache · database · limiter · rag · taskqueue …
│   ├── google_workspace/      # Gmail, Docs, Drive
│   └── tool_handlers/         # search, mermaid, deep research
├── launch_server.py  # entry point · lifespan · teardown registry
└── constants.py
tests/                 # unit · integration · guardrails
pyproject.toml          # packaging, scripts, deps
GEMINI.md               # AI mandate & current-state doc
```

PLATFORM FEATURES

What you can build

Fagoon bridges visual workflow construction, a live code sandbox, conversational channels, and secure external API endpoints — all from one dashboard.

Visual Workflow Canvas & Orchestrator

A drag-and-drop Directed Acyclic Graph builder where you chain multi-modal nodes into automated pipelines. Reach it from the **Workflows** sidebar icon or `/workflow`.

Node types

- LLMs: OpenAI, Gemini, Anthropic, Groq, Perplexity
- Multimedia: Veo Video, Imagen, ElevenLabs, Kokoro
- Tools: Deep Research, Web Search, Code Interpreters, Mermaid
- Google Workspace: Gmail, Docs, Drive connectors

Premium developer features

- **Pin Output** — cache an expensive node's result so downstream nodes get it instantly on re-runs
- Toggle the orange 📌 **Pinned** badge to disable
- **Telemetry terminal** — real-time colored execution logs with auto-scroll

PLATFORM FEATURES

Publish a workflow as an API

Any saved workflow can become a fully-fledged REST endpoint, triggerable from your own sites or servers. Save in the canvas, then click 🔑 **API Integration** on the header bar.

Controls

- **Dynamic slugs** — human-friendly URL mapped to your workflow
- **One-time secret key** — shown exactly once on creation
- **Revocation** — instantly destroy credentials if compromised

Call it from anywhere

POST with an `X-API-Key` header to the webhook endpoint formatted with your slug.

curl

```
$ curl -X POST "https://fagoon.tech/api/v1/workflow-api/{slug}/execute" \
    -H "Content-Type: application/json" \
    -H "X-API-Key: YOUR_API_KEY" \
    -d '{"input": "Describe a spaceship"}'
```

PLATFORM FEATURES

Enterprise Vibe Coder

A visual sandbox that compiles, renders, and previews interactive HTML5 and React single-page apps live, using high-speed AI streaming. Reach it from the **Vibe Coder** sidebar icon or `/coder`.

Double-viewport design

- Code view with formatted syntax and one-click **Copy Code**
- Sandboxed `iframe` rendering compiled HTML5 live
- Contextual model selector with provider logos and pollution guards that default to the first valid model

Streaming & memory

- Token-streaming SSE appends code as the AI types, stripping Markdown backticks live
- Conversation history memory enables incremental directives — "make the score red", "add a restart button" — with full recall

---

PLATFORM FEATURES

Standard Chat Portal

A versatile multi-model conversational interface with retrieval-augmented generation. Open the home route `/chat` or a saved conversation at `/c/{historyId}`.

Multi-model & knowledge base

- Switch models mid-conversation — GPT-4o, Llama-3, Gemini, Anthropic, DeepSeek
- File uploads with a strict 10MB limit, buffered to private GCS buckets

Resilient streaming

- Incomplete-chunk buffering via a trailing `lines.pop()` assembles sliced packets without JSON parse errors
- Floating **Powered by Fagoon AI** indicator

PLATFORM FEATURES

Autonomous AI Agents Hub

A specialized environment for goal-directed, pre-configured agents that perform enterprise tasks in loops. Open **Agents** in the sidebar or `/agents`.

Multi-step parsing

Handles autonomous execution line-by-line: `TOOL_SELECTION`, `STATUS`, `LLM_RESPONSE`, `IMAGE`, `ERROR`.

Collapsible logs

Live-scrolling terminal screens nested inside messages show exact tool selections and background tasks.

Rich media sandbox

Displays generated images and embedded video players, including clips returned from long-running background video tasks.

PLATFORM FEATURES

Model Configuration Portal

A secure admin console to add and modify model setups and coordinate permissions across features. Open **Manage Models** or `/manage-models`.

FIELD	PURPOSE
Configuration name	Friendly display label, e.g. `GPT-4o Production`
Provider selector	OpenAI, Gemini, Anthropic, Perplexity — with a custom **Base URL** override for OpenRouter-style setups
Model ID	Exact system key, e.g. `gpt-4o-2024-05-13`
API key	Secure password field, masked as dots during retrieval
Feature access toggles	Grant or restrict per surface: Chat · Workflow · Coder

PLATFORM FEATURES

User Settings & Preferences

Tailor persona, behavior, and appearance. Open your profile avatar → **Profile**, or `/profile`.

Response tone

Switch the conversational voice between casual, professional, formal, or friendly presets.

System prompt override

Inject global constraints — "always output TypeScript", "summarize in bullets" — across all models.

Display theme

Toggle dark, light, or system themes seamlessly.

REQUEST LIFECYCLES

RAG-enabled agent chat

Retrieval-augmented generation streamed over Server-Sent Events. Entry point: `POST /api/v1/agent/chat`.

REQUEST → STREAM

Auth & validation

`AuthMiddleware` validates the JWT, then the router validates `AgentChatRequest` and retrieves the `ChatOrchestrator` from `app.state`.

Embed the query

The orchestrator calls the embedding model to turn the user query into a vector.

Retrieve chunks

Queries Postgres via `pgvector` asynchronously for relevant `document_chunks`.

Optional BM25 reranking is offloaded to a background thread. `app.state.vector_store`

Tool handling

If the LLM requests a tool, execution routes to `src/services/tool_handlers/` — deep web search, Mermaid charts, and more.

Stream the response

Yields SSE chunks back to the client in event-stream format, using the custom user API key resolved for the logged-in context.

Persist history

Saves chat history to PostgreSQL asynchronously via `AgentChatService`.

REQUEST LIFECYCLES

Background video generation

Long-running rendering, safe in both modes. The HTTP request returns immediately with a `job_id`; the work happens in a Celery worker (full) or an inline asyncio task (lite). Entry point: `/api/v1/video-generation/…`.

DISPATCH → RENDER → NOTIFY

Create the job

A `VideoJob` row is written to PostgreSQL with status `PENDING`.

Enqueue & return

Dispatched through `app.state.queue.enqueue(...)`. The request returns a `202` with the `job_id` right away. `Celery · or inline asyncio`

Worker picks it up

Updates status to `PROCESSING`, then dispatches to `VeoVideoGenerator` or a text-to-video handler.

Store the asset

The result is saved locally or uploaded to Google Cloud Storage, and status is marked `COMPLETED` or `FAILED`.

Broadcast status

A background `periodic_status_broadcaster` reads job statuses from the DB and pushes real-time updates over WebSockets — identical in both modes.

REQUEST LIFECYCLES

Google Workspace operations

An OAuth authorization flow followed by token-delegated requests to Gmail, Docs, and Drive.

OAUTH → DELEGATED CALL

Authorize

The user hits `/api/v1/google-auth/login`, which builds the Google authorization URI.

Exchange & store

On callback to `/api/v1/google-auth/callback`, the temporary code is exchanged for access and refresh tokens, stored in the `google_tokens` table.

Dependency check

A workspace request (e.g. `/api/v1/gmail/messages`) runs through the `get_google_credentials` dependency, verifying the token exists and is valid.

Delegated service call

Service classes in `src/services/google_workspace/` use the `google-api-python-client` to act on the user's behalf. LLM-driven Google AI ops resolve the custom Groq/Gemini key via `resolve_api_key()`.

REQUEST LIFECYCLES

Authentication flow

`AuthMiddleware` runs on all non-public routes, bypassing paths in `ALLOWED_URL_PATH_WITHOUT_AUTHORIZATION`.

EXTRACT → VALIDATE → INJECT

Extract the token

Reads the `Authorization` header, falling back to the `jwt` cookie.

Validate

Decodes the JWT with `JWT_SECRET` and verifies the user is active.

Auto-refresh

If the access token is expired but a valid `refresh_token` cookie exists, a new pair is generated and appended to the response as cookies.

Inject the user

`request.state.user` is injected for router dependencies. Routes are managed in `auth_router.py`.

REQUEST LIFECYCLES

Database switching & live migration

Users can move from the bundled local Postgres to an external, user-supplied database. The process runs sequentially for zero-downtime safety — the local DB is left intact as a fallback. Handled in `src/services/database/switch_service.py`.

VALIDATE → MIGRATE → RESTART

Validate connection

Establishes a connection to the target with the supplied user roles.

pgvector preflight

Verifies the `pgvector` extension is available and can be enabled. Fails fast with a clear message if missing.

Privilege preflight

Confirms the role can create schemas and drop tables.

Schema migration

Runs `alembic upgrade head` against the target database.

Optional data copy

Uses `pg_dump` (data only) from local and restores via `pg_restore`.

Persist & restart

Writes the new URL to `config.json` and sets a restart flag — pools can't hot-swap, so the server restarts to initialize the new pool.

Status polling is file-based. Every phase logs to `db_switch_status.json` so progress stays observable even when the database connection is interrupted.

OPERATE

The fagoon CLI

A thin Typer wrapper over Docker Compose, giving package users an n8n-like one-command experience. It ships its own bundled compose files, so nothing has to be downloaded by hand.

COMMAND	WHAT IT DOES
`fagoon up`	Start the stack — lite mode by default
`fagoon up --full`	Start with Redis + Celery worker
`fagoon up --ollama`	Also start the Ollama fallback LLM
`fagoon down`	Stop the stack
`fagoon logs [-f]`	Tail logs
`fagoon config set KEY=VALUE`	Write to `~/.fagoon/config.json`
`fagoon config show`	Print config with secrets redacted
`fagoon db set-url URL`	Repoint the database; migrates on next restart
`fagoon update`	Pull the newest pinned image and restart

OPERATE

Configuration & secrets

A layered settings resolver lets one codebase serve both audiences: a contributor's environment wins, while a package user who sets nothing still boots on the file and default layers.

Resolution precedence (highest first)

1. **Constructor kwargs**
   Used internally.
2. **Environment variables**
   How contributors configure via shell or CI.
3. **.env file**
   Local development values.

4 **config.json**

Where package mode persists generated values, in `<DATA_DIR>/config.json`.

5 **Field defaults**

Hardcoded fallbacks on the settings model.

First-boot bootstrap

On first run, `ensure_bootstrap` generates the `JWT_SECRET` and a Fernet `encryption_key`, persisting them to `config.json` with `0o600` permissions. This is what lets a package user install with zero environment variables. User-supplied integration secrets (LLM keys, Google OAuth) are entered via the UI and stored encrypted in the DB `credentials` table.

Key environment variables

VARIABLE	ROLE
`LITE_MODE`	The fork. Unset/false → full mode; true → lite mode
`DATABASE_URL`	Explicit DB connection (full mode)
`DATABASE_URL_DEFAULT`	Bundled DSN injected by Compose; used when nothing explicit is set (lite mode)
`REDIS_URL`	Required in full mode — startup raises if missing
`CELERY_BROKER_URL`	Celery broker (full mode)
`WEB_CONCURRENCY`	Worker count; pinned to 1 and enforced in lite mode
`FEATURES`	Comma list: `chat,agents,workflow,vibecoder`
`JWT_SECRET / ENCRYPTION_KEY`	Auto-generated in package mode; set explicitly for dev
`OPENAI / GEMINI / GROQ_API_KEY`	Integration keys — optional, also settable in the UI
`GOOGLE_CLIENT_ID / _SECRET`	Workspace OAuth credentials

Full mode fails loud. If `LITE_MODE` is false and no `REDIS_URL` is set, the settings resolver raises rather than silently degrading. Lite mode ignores `redis_url` even if present — an explicit contract.

DEVELOP

Development & quality

Tests run under pytest with lite mode as the default test environment. Three checks gate every change.

workflow

```
# Run the core test suite
$ uv run pytest tests

# Architecture boundary — forbid stray redis/celery imports
$ uv run lint-imports
$ bash scripts/guard_imports.sh

# Code style
$ uv run ruff check .

# Database migrations
$ alembic upgrade head
```

Async everywhere

DB lookups use `await session.execute()`; blocking work is offloaded with `asyncio.to_thread`. Never lock connections with sync code.

Strict validation

Request payloads and DB-bound structures always pass through Pydantic. Never bypass or suppress validation errors.

SHIP SAFELY

Guardrails, CI & release

The answer to "I can't review every PR." Three automated gates make it physically impossible to merge a change that breaks the lite-mode contract.

import-linter

Forbidden-import contracts fail the build if business code imports redis or celery.

guard script

A blunt grep backup that catches string-level cases outside the approved backend modules.

lite-boot test

Installs *without* the full extra and boots with no env, no Redis — and asserts lite mode refuses multiple workers.

Pipelines

`ci.yml` runs tests, the lite-boot proof, and architecture checks on every PR and on main — publishing nothing. `release.yml` fires only on `v*` tags: it builds a multi-arch image to GHCR, publishes the CLI to PyPI via trusted publishing, and cuts a GitHub release. `CODEOWNERS` forces maintainer review on the mode-machinery files.

> **The tag is the boundary.** Contributor changes never reach package users until a release is tagged. Users pin a version; `main` is for people running from source.

BUILD ORDER

How dual-mode was built — 10 phases

The refactor is one move — "put an interface between code and backend" — repeated for the three things that touch Redis and Celery. Each phase has a known-good starting point and a verification step.

0

Branch & baseline

Create `feat/dual-mode`, confirm the app runs, install dev tooling. Nothing changes behaviorally.

`touches: nothing`

1

Config foundation

Layered settings resolver + first-boot secret bootstrap. Lets a package user install with no `.env`.

touches: maybe merges settings.py

2

Rate limiter abstraction

`RateLimiter` Protocol + Redis and in-memory backends. New files only.

touches: new files

3

Task queue abstraction

`TaskQueue` interface + Celery and inline asyncio backends. Job status still flows through Postgres.

touches: new files

4

Cache abstraction

`Cache` interface with Redis and in-memory TTL backends.

touches: new files

5

Runtime factory + lifespan

The keystone. `runtime.py` reads the flag and builds backends; lifespan attaches them to `app.state` and registers the real Celery task.

touches: launch_server.py

6

Migrate call sites

Every `redis.xxx` and `task.delay(...)` becomes an `app.state` call. The mechanical pass.

touches: routers / services

7

Guardrails

import-linter, the grep guard, and the lite-boot test. Once required, a PR that breaks lite mode can't merge.

touches: new files

8

Packaging

Optional `full` extra, the `fagoon` console script, Dockerfile, entrypoint, both compose files, and the Typer CLI.

touches: merges pyproject.toml

9

GitHub automation

CI + release workflows, CODEOWNERS, PR template. Plus manual UI steps: branch protection, PyPI trusted publishing, GHCR visibility.

`touches: new files`

INVARIANTS

What not to do

The constraints that keep both modes honest. Breaking any of these is how lite installs crash and how concurrency bugs creep in.

- ✕ Don't **import redis or celery** directly in routers or general service modules. Use `app.state.limiter`, `app.state.queue`, or `app.state.cache`.
- ✕ Don't put **core business logic inside router functions** — delegate to services.
- ✕ Don't use **synchronous code** for DB operations or remote calls. Use async variants, or run blocking sync code in a thread executor.
- ✕ Don't **instantiate heavy clients per request** — DB pools, HTTP clients, model registries are lifespan singletons on `app.state`.
- ✕ Don't **bypass AuthMiddleware** for protected routes, or add unprotected pathways without approval.
- ✕ Don't **suppress Pydantic validation**. Maintain strict schemas.
- ✕ Don't **hot-swap database pools** without a server restart. Establish a restart mechanism instead.

The mandate. Any major architectural, structural, or infrastructural change must be documented by updating `GEMINI.md` immediately, so all future agents understand the current system state.
