ChatMaestro — Technical Architecture Specification

Revision: 2026-09-20

Overview

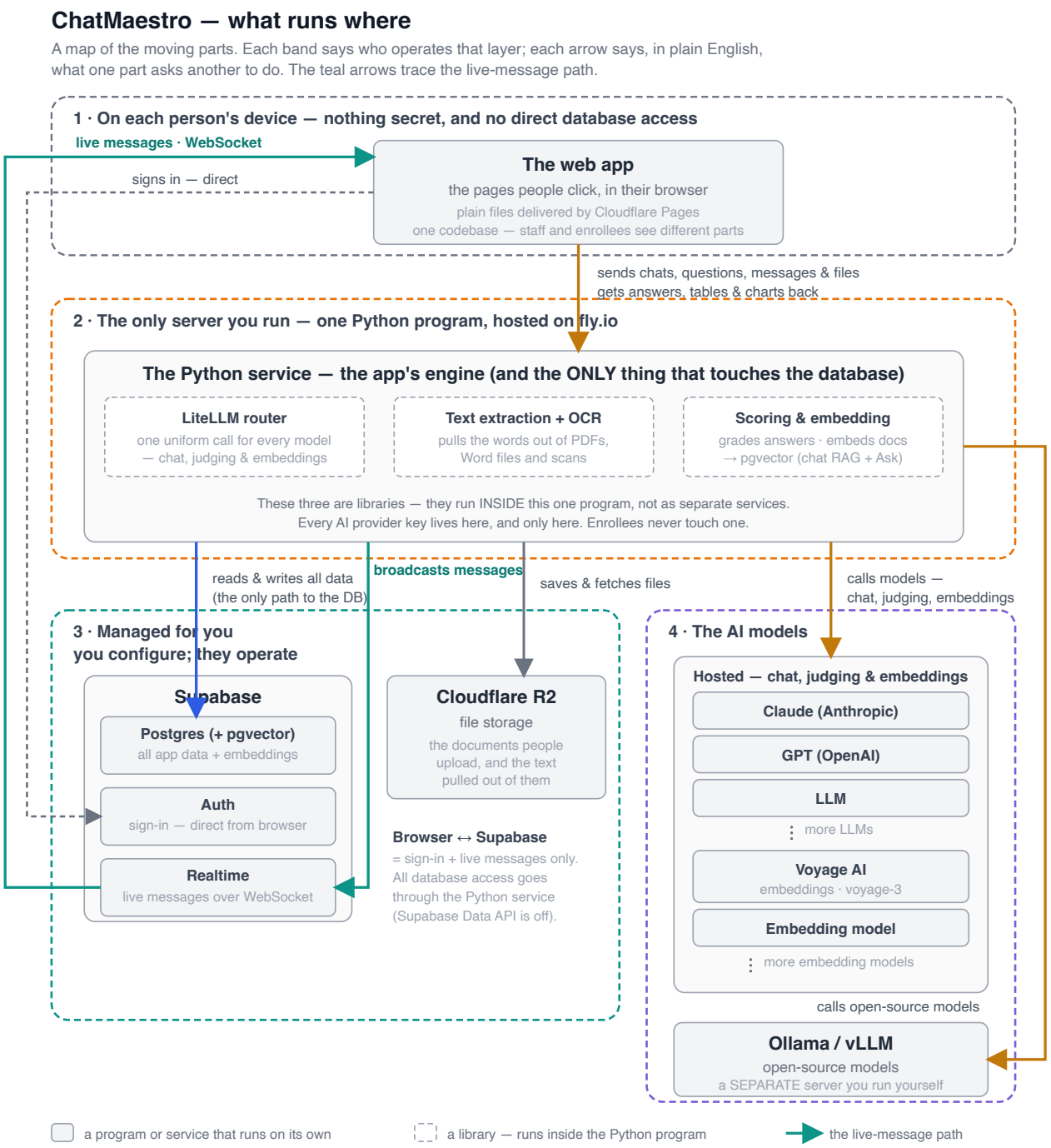
ChatMaestro is an experiment-orchestration platform for controlled, chat-based studies of how the way you configure or prompt a large language model (LLM) changes the quality of its answers. An **experiment** fixes a shared set-up — its *condition* — and lets a single parameter vary across its **runs**, each executed against a **cohort** (a group of enrollees); one trusted judge scores every run the same way, so runs are comparable within their experiment and each run's exact set-up stays reproducible for audit. Architecturally, the platform is a browser single-page app over a persistent middle-tier service that holds the provider API credentials, routes every model call, drives each enrollee's chat, and lets experimenters observe live runs in real time. It captures fine-grained telemetry of every interaction and includes a built-in analytics engine that answers plain-language questions as downloadable tables and charts, with export to outside tools when you prefer.

Context

This document describes ChatMaestro's system-level architecture. It is recommended that you first review the Overview datasheet, the User Interface mockup screens, and the Scenarios documents. The complete set of functional specifications and design documentation is accessible from the "Design & Documentation" home page. It is listed below in the recommended order of reading.

#	Document	What it covers
1	Overview Datasheet	What ChatMaestro is, the problem it solves, what it does, and who it is for.
2	User Interface	Mockup screens that illustrate the workflow from the user's perspective, viewed through an interactive Mockup Viewer app — navigate the screens, search for a specific one, and read on-screen annotations. A static screen gallery is also available.
3	Use Case Scenarios	Formal walkthroughs of common scenarios, such as composing and running experiments. Each has a Given/When/Then spine, a table of variations and exceptions (if this happens, then this is what the system does), an acceptance test, a screen-by-screen journey, state machines for entities with many lifecycle states, a behind-the-scenes sequence that references the system-architecture components, and a plain-English narrative.
4	This System Architecture document	The document you are reading — ChatMaestro's system-level architecture.
5	Technical Datasheet	A complement to this architecture document.
6	Data Model & ERDs	The application's database schema — entities, relationships, and constraint definitions — with narratives. Defined in DBML, a text language for describing database schemas, and rendered as modular entity-relationship diagrams (ERDs), each with its narrative.

System Architecture Diagram



The model names in the diagram (Claude, GPT, Voyage AI) are illustrative — the actual models are configured per install and per experiment, not fixed by the platform.

System Architecture & Topology

System layer	Implementation technology	Execution environment & host	Primary responsibilities
Client frontend	HTML / CSS / JavaScript single-page app (React / Vite)	User's browser, served from the Cloudflare Pages content delivery network (CDN)	Renders both the staff and enrollee interfaces from a single code bundle. For all mainstream operations it talks to the middle tier over HTTPS. It stores no secrets, and contacts Supabase directly only to sign in and to receive live messages.
Middle tier	Python 3 persistent container	Long-running container hosted on the fly.io cloud — the only server that is yours to operate (fly.io hosts it; nothing runs on your own machines)	Isolates model-provider secrets from clients and routes every model call through LiteLLM, a library that presents one interface to many model providers. It is the only path to the database, performing every read and write. It runs optical character recognition (OCR), which reads text out of images, and embedding as background jobs; scores answers on a per-experiment rubric of factors; and answers plain-language questions about the data, reusing a saved question's SQL when one matches and otherwise running an agent that writes and executes read-only SQL — the NL→SQL step (natural language to SQL). It also transmits operator messages to enrollees by broadcasting over Realtime, and sends transactional email such as experiment invitations.
Data tier	PostgreSQL (+ pgvector), Auth, Realtime	Managed Supabase Cloud	Provides relational storage and vector storage, the latter through pgvector, a PostgreSQL extension for vector similarity search. It handles sign-in through Supabase Auth and delivers live messages through Supabase Realtime over a WebSocket, a persistent two-way connection. The middle tier is the only path to this database.
Object storage	Cloudflare R2 (S3-compatible API)	Managed Cloudflare infrastructure	Stores document files, the attachments that supply a run's context, the cache of text extracted from those documents, and the archive written before a run is cleaned up or purged, whose object key is recorded in the audit entry.
Model tier	Hosted provider APIs + open-source models	Provider HTTPS endpoints / self-hosted Ollama or vLLM	Runs the language-model computation (inference) behind chat, scoring judges, and query generation. It also supplies the embedding model that turns text into the vectors used for retrieval (RAG).

The infrastructure footprint is deliberately light: every component is a managed cloud service, including the one server you operate — a single persistent Python container hosted on the fly.io cloud, so you run nothing on your own hardware.

Core System Capabilities

1. Controlled Experimental Conditions

- **Reusable Invariant Frame:** An `experiment` holds the reusable frame of a study: its topic, base system prompt, scenario, enrollee controls, context documents, and a seeded default scoring configuration. It stays editable until its first run; once any run references it, its substantive fields are locked, so every run in the series shares one fixed frame. To change the frame, you clone the experiment. A run is a single execution of that study, launched against a chosen cohort.
- **Four Swept Variables:** An experiment fixes a shared set-up — its condition — and lets exactly four things vary across its runs: the model configuration (`model_config`), the nudge, the target cohort, and the Socratic switch (`socratic_enabled`). A nudge is a short coaching message shown beside the enrollee's prompt box to steer how the person phrases a request; it guides the enrollee and never reaches the model. Everything else is held constant at the experiment level: the prompts, scenario, context documents, enrollee controls, and scoring configuration. A comparison is a group of runs within one experiment that holds three of the four steady and varies the remaining one, so any difference between the runs traces to that parameter.
- **Immutability by Reference:** A run keeps no copy of its set-up. Every definition it points at — the `experiment` , the `nudge` , the `model_config` and its member models, and the `cohort` and its membership — is held immutable while any run references it: its substantive fields become read-only and it cannot be deleted (`RESTRICT`). A finished run's exact set-up is therefore always recoverable by reading those definitions directly. To adjust a locked definition, you clone it, which yields a new, distinctly named definition.
- **Cohort Isolation:** Targeting is per cohort, never per individual. Variation is introduced across cohorts, either concurrently as A/B and multi-arm designs over non-overlapping cohorts, or sequentially by running the same cohort again under a changed set-up. The middle tier applies each run's set-up server-side as it drives every enrollee's chat.

2. Live Observation & Operational Recovery

This layer exists so an experimenter can keep a run healthy while it happens: watch it unfold, message enrollees, and step in when something needs attention.

- **Real-Time Streaming:** An experimenter follows a run as it streams and can drill into any one enrollee's chat session live, exchange by exchange. Enrollees, in turn, receive operator messages in real time. Both arrive in the browser over its single Supabase Realtime WebSocket, described under *Technical Security & Deployment Model* below.
- **Non-Contaminating Interventions:** Two kinds of action run alongside a live session without touching the frozen set-up. The first is operational recovery: retrying a failed model call, pausing or aborting a run, or withdrawing an enrollee. The second is out-of-band operator messaging, meaning messages sent outside the normal chat flow, whether to one enrollee, to a cohort, or as a broadcast; it uses the `message` and `message_recipient` channel, which the middle tier writes and broadcasts.
- **Departure From Either Side:** Two actors can end an enrollee's part in a study, at two scopes. The experimenter withdraws one person from one run (`run_enrollment.withdrawn_at`). The enrollee's own act is pool-wide: opting out sets `profile.opted_out_at` , which withdraws every live enrollment at once and makes the person ineligible for future invitations, and it needs nobody's permission; there is no enrollee control for leaving a single study. Both land the same way: the rounds already recorded stay, so a departure is a recorded event and a finished comparison is never silently rewritten. Specified in ALGORITHMS §28.

3. Provider Access & Credential Isolation

- **Zero-Key Client:** Provider API keys live as fly.io secrets inside the Python service. Enrollees bring no keys and no accounts, so they carry no cost or rate-limit exposure.
- **One Interface to Every Provider:** A LiteLLM router inside the Python service gives every model call one interface, whether the model is hosted by a provider (Anthropic, OpenAI, Google) or self-hosted (Ollama, vLLM), and that same interface covers embeddings. A small model-capability skip-list drops parameters that particular models reject; OpenAI reasoning models, for example, ignore `temperature` .

4. Model Configurations & Combine Methods

- **A Configuration Is a Routing Recipe:** A `model_config` names its member LLMs — each a `model_config_model` carrying a relative `weight` — and sets the `combine_method` that turns whatever they produce into the single response the enrollee sees. The middle tier reads the recipe and issues the calls. The `weight` also settles which member leads where a method needs a leader, and the judge is named on the `experiment` , so a recipe needs neither a lead-member flag nor a judge member.
- **Six Methods, Five of Which Fan Out:** `single` is the one-member case, and is required for one. `synthesize` , `vote` , `consensus` , and `judge_best` call every member in parallel and then reduce. The reduce step is what separates them: the highest-weight member merges the anonymized candidates; the members peer-score each other on the rubric; the largest cluster of candidates that agree yields its most central answer; or the trusted judge picks the best. `moe` inverts the shape — a router chooses one member before anything is generated, so a round costs one generation however many members the configuration holds. Each method is specified in ALGORITHMS §1.
- **Mixture of Experts Routes Deterministically:** The `moe` router compares the prompt by meaning against each member's curated capability sentence (`model_catalog.expertise`), which is arithmetic over cached vectors, so it is instant and repeats exactly. It escalates to the experiment's trusted judge — at temperature 0, before anything is generated — only when several members tie, because embeddings capture a prompt's topic and not its difficulty. An unresolved tie falls to the highest- `weight` member, so the model set-up stays a fixed, reproducible condition. Every decision is recorded per round in `chat_round.moe_routing` , along with the stage that made it. Specified in ALGORITHMS §19.
- **Routing Is Something the Platform Measures:** The combine method is part of a configuration, and a configuration is one of the four swept variables, so `moe` can be compared against merging, voting, and judging on the same rubric. Because generator cost is recorded apart from judge cost, the comparison extends to quality per dollar.

5. Prompt Enhancement

- **Rewriting a Request Before It Is Answered:** With `experiment.enable_prompt_enhancer` on, the middle tier rewrites the enrollee's raw prompt into a better-engineered one before the configuration answers — a single pass on the experiment's judge model, under a versioned enhancer-instructions asset (`prompt_enhancer_version`) and aimed at the scoring rubric, turning "summarize this" into a well-scoped instruction that exploits the attached context. Both the raw and the effective prompt are stored on `chat_round` , and when the experiment is not blinded the enrollee sees the enhanced version.
- **A Frozen Element of the Condition:** Enhancement is off by default, and the *policy* — on or off, under one instruction version — is held constant across a comparison, which is what keeps a study comparable. Its main use is the best-output mode, where the goal is the strongest answer. Specified in ALGORITHMS §18.

6. Asynchronous Ingestion Pipeline

- **Two Kinds of Source:** The pipeline indexes documents and, separately, rows of the tables named in `embedding_ingest_registry` — so a `profile.notes` field is as retrievable by meaning as a PDF is. The two differ only in how their text is produced: a document is extracted, while a table row is rendered into a sentence or two by the registry's template. Everything after that is identical.
- **Off the Request Path:** A source is loaded and processed once, then embedded in the background, so ingestion runs independently of any live request. Embedding turns a passage of text into a numeric vector that captures its meaning.
- **Multi-Format Extraction:** Documents arrive in many formats, and the pipeline extracts text from Word, Excel, PowerPoint, PDF, and CSV, TXT, MD, and HTML files, drawing on the Tika, unstructured, python-docx, openpyxl, pdfminer, and Tesseract libraries; a scanned PDF has optical character recognition applied first. A plain-text file needs no extraction at all, so its snapshot is a copy.
- **Embedded for Two Consumers:** The embeddings serve two consumers. They give the chat sessions retrieval-augmented generation (RAG), which retrieves relevant passages of context to ground a model's answer. They also give the Ask engine semantic search over both the reference documents and the registered table rows, which it reads under the asker's own row-level security, finding content by meaning rather than by exact words. The embeddings are produced through the same LiteLLM router, for example with Voyage AI's `voyage-3` model, and are stored in Postgres through pgvector.
- **Decoupled Status Tracking:** Text extraction (`document.extraction_status`) and embedding work (`embedding_job.status`) are tracked independently, so each stage's state is observable on its own.

7. Chat Sessions & Memory

- **A Session That Lasts the Run:** An enrollee's chat with a run is one continuous session. It has no idle timeout and no lifetime cap — someone can close the laptop, travel for a week, and return to the same conversation. Ending it is an act rather than an expiry: the enrollee starts a fresh session, the run finishes, or the experimenter withdraws them. A session's turns are its `chat_round` rows in order, and one round is one completed exchange.
- **Chat Memory, Bounded by the Smallest Member:** With `experiment.enable_chat_history` on (the default), earlier turns replay alongside each new message, so a model can resolve "and what about the second one?" Off, every round stands alone. Depth is the platform constant `CHAT_HISTORY_MAX_TOKENS` (default 16,384), held under the **smallest** `model_catalog.context_window` among the configuration's members and net of everything else the round must carry. Taking the smallest is what keeps a multi-model round fair: every member receives identical input, so a difference between their answers is a difference between the models. What a round actually carried is recorded in `chat_round.history_window` , and a run's totals in `run_result.history_stats` . Specified in ALGORITHMS §23.
- **Filling Up Warns, Never Stops:** At `CONTEXT_ALERT_THRESHOLD` (default 0.80) of the budget the enrollee sees a notice that the session is filling and what will happen at the limit; it recommends starting fresh rather than requiring it. At the limit the oldest turns drop and the conversation continues. That the warning fired is itself recorded, since a warned enrollee may behave differently from an unwarned one.
- **Rewind, and Start Fresh:** An enrollee can turn the session back to an earlier point and continue from there (`chat_round.rewound_at`). Rewound rounds are marked rather than deleted — they stay in the transcript, keep their scores, still count in the run's statistics, and simply drop out of the replay — so the record of what was said survives while the live conversation resumes from the chosen point. Clear chat and start new is the blunter sibling: it begins a new session (`chat_round.session_seq` increments) so history assembles from empty, and again nothing is deleted. Both controls are always available to the enrollee and neither is part of the condition.

8. Handling an Underspecified Request

Every message an enrollee sends is checked before any model under study sees it. The check is one mechanism; the Socratic switch decides what the enrollee is told when it stops them.

- **The Sufficiency Gate:** The trusted judge tests each message twice — against whether the corpus holds anything on the topic, and against a checklist the experiment can declare of what a question must contain (`experiment.required_slots`) — and turns back what cannot be answered. A turned-back attempt writes **no round**, since a round is the unit every statistic counts; it rides along on the round that finally passes (`chat_round.clarification`), and its judge spend is charged there. The models under study never see a message that was stopped, and never ask for a missing detail themselves. Specified in ALGORITHMS §21.
- **Socratic Mode — the Fourth Swept Variable:** The switch (`experiment_run.socratic_enabled`) selects how the judge words a stop. With it on, the judge asks for one missing piece at a time and the request is assembled through dialogue; with it off, it names everything missing and asks the enrollee to rewrite the request and resend it. The models then receive either that dialogue, as question-and-answer pairs, or the single rewritten request. Detection is identical either way, so the arms are repaired differently rather than held to different standards. Both phrasings are frozen on the experiment (`experiment.socratic_version`), so an experimenter never edits prompt text to run this study. Specified in ALGORITHMS §30.
- **What the Switch Compares Depends on the Experiment:** With a checklist declared or context documents attached, both settings have a working gate and the comparison is between two ways of repairing a request. With neither, the off setting has no gate at all and its models answer whatever is typed, while the on setting has the judge read each request on its own merits — the plainer question of whether eliciting is worth doing.

- **Measuring It:** Answer quality is already in the rubric scores. What the switch adds is `chat_round.retry_count`, the number of attempts turned back before an answer existed, rolled up per run into `run_result.retry_stats`. Both settings count it, so the arms compare directly on how many attempts an answer took. The asking arm spends more on the judge, and because judge cost is recorded apart from generator cost, that shows as its own figure.

9. Interaction Capture

- **Granular Telemetry:** Every exchange records one `chat_round`. It holds the prompt, the winning `response`, which member model produced it (`best_model`, a reference to the winning `model_config_model` — the configuration itself is fixed for the whole run, so what varies round to round is which of its members won), and the exchange's measurements: the number of iterations, the stop reason, the latency, and the token counts and cost, split between generating the answers and judging them. Under `moe` it also carries the routing decision (`chat_round.moe_routing`), so *which* expert answered is joined by *why*. Each competing answer is kept as a `chat_round_candidate`.
- **Roll-Up Into a Scorecard:** When a run finishes, its rounds roll up into one `run_result` scorecard, together with any operator notes for that run. There, `model_perf` records each member model's wins, which under `moe` is the run's routing distribution.

10. Scoring & the Trusted Judge

- **One Judge, Named on the Experiment:** The judge is named on the experiment (`experiment.score_judge_*`), picked from the curated `model_catalog`, and it does *all* of the scoring — grading each candidate during a round to pick the winner and drive the self-improvement loop, then supplying the chosen answer's comparable scores. A judge is a model that scores answers instead of generating them: the industry-standard, reference-free **LLM-as-a-judge** approach, which needs no gold reference answer. It rides the same router that drives chat and is pinned for the life of a comparison, so every run of an experiment is graded by the same grader.
- **Configurable Rubric:** The judge grades each finished exchange on the experiment's rubric factors, stored as a jsonb bag (`chat_round.factor_scores`) so the factor set is configurable per experiment. The built-in **default rubric** has four factors, and an experiment may pick a different rubric whose criteria suit its study (say funniness, originality, timeliness for a joke experiment). The experimenter selects it from a drop-down populated by the middle tier's rubric registry, a set of versioned configuration files bundled with the release, so no exact name is typed by hand. The default four:
  - `groundedness` — how faithful the answer is to a designated source, whether retrieved context or a reference the prompt carries; scored only when a source exists. From retrieval-augmented-generation evaluation ("faithfulness" in RAGAS, part of the TruLens RAG triad).
  - `relevance` — how well the answer aligns with the prompt. From answer relevance in RAG evaluation (RAGAS, TruLens).
  - `coherence` — the answer's structural and logical flow. From summarization evaluation (G-Eval, SummEval).
  - `instruction_following` — how well the answer obeys the instructions in the system prompt: its format, tone, and constraints. This is the factor on which models differ most. From instruction-following benchmarks (IFEval).
- **Weighted Composite:** The rubric's factors combine into one headline number, `score_composite` (kept as its own column). By default each factor carries equal weight; an experiment's `score_weights` can change the mix, and because the experiment is held immutable while any run references it, the weights in force are fixed for the life of the comparison, so a run's composite can always be recomputed from the experiment's own record.
- **A Refusal Is Not a Bad Answer:** Alongside the scores the judge returns a `declined` verdict when a response declines the request rather than attempting it, marked `appropriate` or `unwarranted` against the system prompt the judge already holds. An appropriate refusal takes relevance to N/A instead of a low score, so a model that correctly declines is not ranked below one that answers anyway; an unwarranted one stays scored, so declining is never a way to dodge a low score. The run's scorecard freezes the tally in `run_result.declined_dist`. Specified in ALGORITHMS §4.
- **A Scorer That Ships With the Release:** The scorer is about 200 lines of hand-written Python in the middle tier, so the judge's prompt is part of the experiment's set-up and travels with the release rather than sitting inside a dependency. `experiment.score_rubric_version` records any deliberate change to the rubric. Specified in ALGORITHMS §4.

11. Descriptive Analytics — The Ask Engine

- **Three Paths to an Answer:** Experimenters query the captured records in plain language, and a question reaches the data one of three ways; only the third spends tokens on writing SQL. A saved-question **button** already holds its statements: the engine binds the inputs and runs. A **typed** question is embedded and matched by meaning against the saved questions the asker may see; a close match (`ASK_MATCH_THRESHOLD`) reuses that question's SQL with the values lifted from the words. A question that matches nothing goes to a tool-using **agent** — the NL→SQL step, an LLM through the same router — whose system prompt carries the full schema context pack (`ASK_PACK_MODE = full`) and whose tools are `query_database`, which runs a single read-only statement, and `join_path`, a deterministic walk of the foreign-key graph that answers the ambiguous-join question by construction rather than from the model's recall. The pack is generated from two deterministic sources, `schema.dbml` (what exists) and `semantic-layer.v1.yaml` (what it means and how to compute it: each table's grain, the preferred joins, the metric definitions, and the rules that keep an answer from being plausibly wrong), so the agent's knowledge of the data is a build artifact rather than something it remembers. Every path ends the same way, with a small model writing the narrative from the rows returned. Specified in ALGORITHMS §8.

- **Scoped & Read-Only:** Generated SQL is refused unless it begins with `SELECT` or `WITH`, is tested with the `EXPLAIN` command before it runs, and executes under the asker's own row-level security — never a service-role key — so a query can return only rows the asker could already see on a screen, with no permission logic in the engine itself. Each answer is a short written summary plus zero or more result tables, and each result table can be shown as a downloadable table or a chart. The generated SQL is available on demand.
- **Parameterized Saved Questions:** A question can be saved as a reusable `nl_query`, which turns it into a fill-in-the-blanks form: a user picks the saved question and supplies parameter values. Because the SQL is already compiled, a saved question runs faster and spends no tokens on the NL→SQL step. It stores prepared statements (`derived_sqls`) — pre-written SQL with placeholders — beside a typed parameter specification (`params`), and values always pass in as bound parameters, never pasted into the SQL text, which keeps the query safe. Comparing results across runs is one such saved question: it applies a `GROUP BY` at query time over `run_result`, or over the raw `chat_round` records, on whatever varied between the runs. A saved question begins as its author's private draft. An admin approves it into a button everyone in that context sees, and can take it back: unapprove it to its owner, retire it (a soft `retired_at`, undoable, which removes it from the buttons and from matching), or delete it. The approval note travels with the row in `nl_query.notes`, and the Saved-questions screen is one screen for both roles, scoped by row-level security.
- **Export for External Tools:** The data tables a query returns can be exported as CSV, XLSX, or JSON for the analytics tool of your choice; chat transcripts export as text. This sits alongside the built-in Ask engine.

Technical Security & Deployment Model

- **Single Data Path:** Every database read and write goes through the Python service. The Supabase Data API (PostgREST) is turned off, so the only Supabase features the browser reaches are Auth and Realtime. The service connects to the database over the pooled Postgres protocol, which reuses a shared set of connections.
- **Two Direct Browser–Supabase Links:** The browser talks to Supabase directly for two things: to sign in through Auth, and to receive live messages through Realtime over a WebSocket — a persistent two-way connection on which the browser only *receives*, while the middle tier *publishes*. Signing in yields a Supabase token, which the browser uses to open that socket, and access to the socket is gated by authorization. Every other call the browser makes is an HTTPS (encrypted web) request to the Python service.
- **Role-Based Access, Enforced by the Database:** Authorization follows a three-role model with ownership scoping. Enrollees see parts of their assigned runs and read and write their own chats and messages. Experimenters compose and update experiments and cohorts, launch runs, message enrollees within a run, manage invitations for experiments they own, and run analytics over their own results. Admins additionally manage experimenters, see all experimenters and enrollees, and run analytics across every experiment. The rules are Postgres row-level security (RLS) policies keyed to `profile.role`, `enabled` and `opted_out_at`, re-read on every request. The middle tier is the only database caller, but it does not stand in for the caller: on every query it presents the signed-in person's own token (setting `request.jwt.claims`) so the policies apply to that person, and it never uses the service-role key on a path that reads or writes user data. A role change or a deactivation therefore takes effect on the person's next query, and generated SQL inherits the same boundary with no permission logic of its own.
- **The Two Authorization Surfaces Beyond Plain CRUD:** Most access is plain create, read, update, and delete (CRUD) of records. Two surfaces go beyond that. The Realtime channel delivers straight from Supabase to the browser, so the channel's own authorization, keyed to the enrollee and the run, governs which messages a session receives. And the Ask engine runs LLM-generated SQL under the caller's own row-level security, so a query stays within the caller's permissions.
- **Messaging Flow:** A message sent from one browser to another first goes to the Python service. The service writes the `message`, creates one `message_recipient` row per recipient, and broadcasts on the recipient's private Realtime channel. The recipient's browser receives it over its WebSocket. The recipient sees the message as coming from the operator, so the real sender stays hidden; this concealment is called blinding.
- **Persistent Container and Scale:** The fly.io Python tier runs continuously. It holds its own pool of database connections, calls external OCR programs, and runs long-lived background jobs. It scales horizontally, adding more instances, behind the single data path, which keeps connection pooling and caching in one place.
- **Portability:** Supabase Auth is built on GoTrue, which can be self-hosted, and Postgres is portable, so both can move off Supabase later. Realtime is the Supabase-specific piece, so a future migration would need to add a replacement for it.
- **High-Volume Tables:** The high-volume tables `chat_round`, `chat_round_candidate`, and `audit_log` are ordinary tables with simple primary keys. Deep paging uses keyset pagination — paging by a sort key `(created_at, id)` rather than a slow numeric offset — which stays fast at any depth, so partitioning is not needed for it. Range partitioning is a scale option left for later, if these tables ever reach the tens of millions of rows; it is a known migration, deliberately deferred to keep the keys simple now.

Scientific Guarantees

Each row names a property a study relies on, the mechanism that secures it, and where the guarantee stops.

Scientific target	Architectural guarantee	Where the guarantee stops
Set-up rigor	Every definition a run references — experiment, nudge, model configuration, cohort — is held immutable while referenced, so a run's exact set-up is always recoverable without a stored copy.	A run's set-up is fixed once it launches. To change it, clone the definition or the experiment and launch a new run.
Comparative validity	One trusted judge grades every run of an experiment identically, so within that experiment the relative ordering between runs is dependable.	Runs are comparable only within their experiment, never across experiments; the scores are best read as a comparison between runs rather than an absolute measure of correctness.
Equal footing within a round	Chat memory is trimmed to the smallest `context_window` among a configuration's members, so every model answering a round receives identical input and a difference between their answers is a difference between the models.	Two arms of one comparison can still sit under different smallest-windows when their configurations differ. `run_result.history_stats` records `context_window_min`, so the analysis can say which arm was working with less.
Socratic comparison	The Socratic setting is a switch over frozen wording, so the arms differ in how the judge words a stopped request and in nothing else. Where the experiment declares a checklist, both settings detect identically and both count every stopped attempt, so the arms compare on equal terms.	They diverge in detection only when no checklist is declared, where the off setting has no gate at all — a comparison against an ungated baseline rather than between two repair styles. The compose screen names which shape an experiment is in.
Groundedness	This factor scores how faithful an answer is to the context it was given.	It reflects how well the answer uses the provided context; the quality of the retrieval that supplied that context is a separate measure. Retrieval is by similarity over independent chunks, so an answer that missed a relationship stated across two passages is scored as faithful to what it saw, not penalized for what it never received (see *Chunk-level retrieval* under Architectural Trade-Offs).
Composite score	`score_composite` summarizes the rubric's factors. By default it weights them equally, and `score_weights` can change those weights.	The per-factor breakdown in `factor_scores` shows where a composite score comes from, so read it alongside.
Routing determinism	Under `moe` the choice of expert is reproducible: the embedding stage is arithmetic on fixed vectors, the judge stage runs at temperature 0, and an unresolved tie falls to the highest-weight member. Every decision is recorded per round.	The routed model is fixed for a given prompt and configuration; the *answer* it then generates carries the usual LLM variability. Changing the embedding model, or editing a model's capability sentence, changes routing from that point on — both are recorded changes.
Judge reliability	One trusted judge does all scoring on one rubric, so scores are internally consistent and comparable across an experiment's runs. Validating the judge against people is an optional offline step — export the rounds and their scores and compare — not modeled in the app.	Like any LLM judge, it carries biases such as verbosity, self-preference, and top-scale compression, along with run-to-run variation; an offline human-agreement check is how you would surface these.

The rubric is general-purpose: it measures answer quality against stated criteria, which suits it to comparing runs across a wide range of open-ended, language-based tasks. It grades language quality rather than task-specific ground truth, so for work demanding verifiable correctness — code generation, numeric computation, or any task with hard checkable constraints — pair the rubric with your own checks, which live outside it.

Architectural Trade-Offs

Each row is a design decision with an alternative that was given up. The reasoning behind the more recent ones is recorded in `DECISIONS.md`; the rest is stated where the decision applies, in the capability sections above and in ALGORITHMS.

Decision	What it buys	What it gives up	Revisit when
Immutability by reference, not by snapshot. A run points at its definitions and locks them; it keeps no copy.	A finished run's exact set-up is always the live definition, so there is no copy to drift and nothing to reconcile.	A referenced definition cannot be edited or deleted while any run points at it; changing one means cloning it under a new name.	Editing a definition under live runs becomes a genuine need rather than an error to prevent.
One middle tier as the only data path. The Supabase Data API is off; the browser reaches Supabase only for Auth and Realtime.	Provider secrets never leave the service; one place holds connection pooling, caching, background jobs, and the audit point.	Every read is a round-trip through fly.io; the browser cannot query the database directly, however simple the read.	A read path proves latency-sensitive enough to justify a second, tightly scoped route.
Row-level security as the enforcement, with the service impersonating the caller.	Authorization lives with the data and cannot be bypassed by a code path; the Ask engine's generated SQL inherits it with no permission logic of its own.	The policies are Postgres-specific and must be tested as code; the service must present the caller's token on every user-data query and never take the service-role shortcut.	A move off Postgres, which would carry the policies with it as a rewrite.
Supabase Realtime for live delivery.	Managed WebSocket fan-out to enrollees and experimenters with no server of your own to run.	It is the one vendor-specific component; Auth (GoTrue) and Postgres are portable, Realtime is not.	A migration off Supabase, which must add a replacement first.
Keyset pagination instead of table partitioning on the high-volume tables.	Simple primary keys, and deep paging that stays fast at any depth today.	Nothing yet; partitioning is a known migration deliberately not paid for in advance.	`chat_round`, `chat_round_candidate` or `audit_log` reach the tens of millions of rows.
The full schema context pack in the Ask agent's system prompt (`ASK_PACK_MODE = full`), not a per-question slice.	The prompt is static, so it is a prompt-cache hit; a follow-up can pivot to any table the agent was never asked about.	About 11k tokens up front per session, most of them irrelevant to any one question.	The pack passes roughly 25k tokens as the schema grows, or the Ask model runs on a provider without prompt caching (then `slice`, which is built and tested behind the flag).
Chunk-level retrieval, no knowledge graph. Documents and registered table rows are cut into independent chunks and retrieved by similarity alone.	One ingestion path for every source kind, no extraction model in the pipeline, and retrieval that is arithmetic over stored vectors: fast, deterministic, and free of a second model's failure modes.	Relationships between concepts are implicit. A question that must join facts stated in different places is answered correctly only if both passages rank in the top five and the model connects them; two documents naming one thing differently are not reconciled; and groundedness scores an answer against the chunks it received, so a missed relationship is invisible to the score.	A study's context is a corpus whose parts refer to one another — a handbook and its amendments, a case file, a body of regulations — and the questions are about how the parts relate (see *Knowledge graph over the context corpus* under Future Direction).
One trusted judge, not a panel.	Every run of an experiment is graded identically, so runs compare cleanly, and judge cost is one line.	The judge's biases — verbosity, self-preference, top-of-scale compression — go uncorrected in the app; validating it against people is an offline export step.	Human scoring lands (see Future Direction), which brings calibration into the app without touching the single-judge path.
No stored comparison object. A cross-run comparison is a query-time grouping of `run_result` rows on whatever varied.	Nothing to keep in sync with the runs; any subset of runs can be compared after the fact.	A comparison has no identity of its own to name, annotate, or share; it is reconstructed each time from its runs.	Experimenters want named, saved comparisons with their own notes.

Future Direction

These directions are planned rather than built. Each one extends the platform without changing the guarantees above.

- **Predictive and prescriptive analytics:** The analytics layer is currently descriptive — it reports what happened. A predictive tier would forecast which run or model configuration is likely to score best, or where a run is trending. A prescriptive tier would go further and recommend the next action.
- **Headless-enrollee experiments:** High-bandwidth studies that put LLM-driven simulated enrollees, fed synthesized prompts, in place of human ones — a way to scale a design past the throughput of live enrollees, or to dry-run it before people take part.
- **Deployment flexibility:** Run the middle tier on any container service (self-hosting included), serve the front end from any web host, authenticate through any provider, and store data in any managed PostgreSQL — so no single vendor is load-bearing.
- **Ground-truth scoring and objective-benchmark studies:** Deterministic checks that verify an answer against a known-correct result — a multiple-choice letter matching the key, code passing its tests, a value being exactly right — for tasks whose answers are checkable. As a full capability it lets an experiment run an **objective benchmark with human participants**: a reusable **item bank**, drawn reproducibly from a benchmark such as MMLU, is walked past each enrollee, who prompts the model in a tone or style coached by the nudge, while a `scoring_mode` switch replaces the judge with a deterministic parse-and-match against the key at no judge cost. Because a person supplies the phrasing, the platform also measures the **enrollee's own prompt**, gains per-model generation controls, and reports **accuracy against inference cost**. Detailed in the objective-benchmark study design note.
- **Human scoring and judge calibration:** Alongside the automated judge, let enrollees rate answers in the flow (a satisfaction signal) and experimenters hand-score them (gold labels). Because many people can rate the same answer, these land in a new `rating` table keyed to `chat_round` — an additive change that leaves the single-judge path and its within-experiment comparability untouched. It brings judge calibration, currently an offline export step, into the app.
- **Multimodal context:** Context is currently **text**, end to end: the ingestion pipeline reduces every format it accepts down to words, and a chat session sends the models text alone. Supporting images, audio, and video as first-class context would let a study put the artifact itself in front of the models — ask about what a diagram shows, or about a recording rather than its transcript. It reaches further than the ingestion pipeline: each modality needs its own embedding path, the round's stored prompt has to reference the attachment instead of inlining it, and the smallest-member budget that keeps a round fair has to be computed in whatever units each provider bills a non-text input in. Because member models differ in which modalities they accept, the `model_catalog` gains a capability declaration, and the fairness rule already enforced on context length extends to modality.
- **Knowledge graph over the context corpus (GraphRAG):** Context retrieval today is chunk-level: a document is cut into windows of about a thousand characters, each embedded on its own, and a round receives the five nearest. Relationships between concepts — that one document's addendum revises a procedure another document defines, that two documents name the same thing differently, that a term defined in one place is used in another — exist only implicitly, in the embedding geometry and inside whatever chunks happen to co-occur. A knowledge-graph layer would extract entities and the relations between them at ingest, store them beside the chunks (a graph keyed to `embedding` and `document`), and retrieve by graph neighborhood as well as by similarity, so a multi-hop question is answered from the passages that are actually connected rather than the ones that happen to be nearest. It matters for studies whose context is a corpus rather than a single reference — a handbook plus its amendments, a case file, a body of regulations — where what the study asks enrollees to explore is precisely how the parts relate; for a study grounded in one self-contained document, chunk retrieval is enough. It changes what *context* means for the groundedness factor, since an answer would be graded against connected passages rather than the nearest ones; it adds an extraction step to the ingestion pipeline with a model call and failure modes of its own; and it is a new stored structure. Under the engineering standards it begins as a design review of two or three candidate shapes — entity and relation extraction into a graph table, hierarchical parent-and-child chunking so a hit can pull its surrounding section, and document-level summaries indexed alongside the chunks — before anything is built.
- **Prompt decomposition:** A request carrying several distinct asks currently goes to one routed model, strong at only part of it. Decomposition splits the request, routes each part on its own merits, and has a master combine the results — pure prompting, with no external tools or data sources. The shape is analogous to relational database query optimization and access path selection. The planner splits by task structure and stays blind to which models are configured, which keeps both arms of a comparison answering the same question. On a worked example the cost was close to a wash, so the case rests on answer quality. Sketched in ALGORITHMS §31.
- **Advanced prompt enhancement:** The built-in prompt enhancer (see *Prompt Enhancement* above) is currently a single deterministic rewrite. Future work makes it *learn*: automatic prompt-optimization techniques — OPRO (an LLM proposes better prompts, guided by past prompt-and-score pairs), DSPy (compiles and tunes prompts against a metric), and evolutionary search — would fit its instructions to measured response scores, and critique-guided rewriting (Reflexion-style) could refine a prompt before it is sent. These sit alongside the offline prompt-eval work, which searches over *system-prompt* variants. Detailed in the prompt-enhancement design note.