ChatMaestro

Controlled experiments for large language model (LLM) answer quality.

DATA SHEET
TECHNICAL

Revision: 2026-09-06

ChatMaestro is an experiment-orchestration platform for chat-based LLM studies. An experiment fixes a shared set-up — its condition — and a run is one execution of it against a cohort of human enrollees; across a group of runs you vary a single parameter and hold the rest constant. 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.

ARCHITECTURE AT A GLANCE

Front end	A single-page web app (React/Vite). It calls the middle tier over HTTPS for everything, and reaches Supabase directly only to sign in and to receive live messages.	**In each user's browser, from the Cloudflare Pages CDN**
Middle tier	A persistent Python service: the sole path to the database, the router for every model call (via LiteLLM), and the runner of OCR, embedding, scoring, NL→SQL, live-message broadcast, and email. It holds the provider API keys.	**One long-running container on fly.io — the only server you operate**
Data tier	Postgres (with the pgvector extension), Auth, and Realtime — managed services you configure but Supabase hosts and runs.	**Supabase**
Object storage	Uploaded files, extracted-text caches, and database snapshots, referenced from Postgres by key.	**Cloudflare R2 (S3-compatible)**
Models	Large language models for chat and scoring, plus embedding models for search. A provider-agnostic catalog the admin curates live by adding a provider's API key — no redeploy.	**Any hosted or local model**

THE DATA MODEL — 24 TABLES, 4 MODULES

core-identity (5 tables) holds profiles and roles, cohorts (defined groups of enrollees) and their membership, the append-only audit trail whose rows are only ever added, and invite email templates.

core-experiment (8 tables) holds experiments and their runs, the reusable nudges runs apply, enrollees, the per-run scorecard, the experiment's context documents, and the operator↔enrollee message channel, which runs out of band, as a side channel separate from the experiment chat.

llm-chat (5 tables) holds the model catalog, reusable model configurations and their member models, and scored chat rounds with per-candidate detail.

search (5 tables) holds documents, their embeddings (stored with the pgvector extension and indexed with HNSW, a hierarchical navigable small world index, for fast similarity search), the queue and registry that ingest them, and the saved natural-language queries behind the Ask feature.

SECURITY MODEL

- **One data path, guarded by row-level security**
  Every database read and write goes through the middle tier — the Supabase Data API is off — and Postgres row-level security enforces the three roles (admin ▸ experimenter ▸ enrollee) inside the database. Enforcement never depends on the client: the same JavaScript bundle ships to every user, so hiding a control changes only the user experience (UX) and grants no access.
- **End-user identity reaches the database**
  The middle tier forwards the caller's signed identity token, a JSON Web Token (JWT), with each database transaction. The database reads that token to identify the user, so its row-level rules apply to that specific person. A separate service key can bypass those rules, and it is never used on any path that touches user data.
- **Provider API keys never leave the server**
  The provider API keys exist only as fly.io secrets on the Python service. Enrollees hold no API keys, so they carry no provider cost and face no provider rate limits.
- **Passwordless sign-in, blinded enrollees**
  Sign-in uses Supabase Auth, either through an OAuth provider such as Google or Microsoft or a one-time code sent by email. Each identity is a stable UUID (universally unique identifier), independent of the email address. Each enrollee gets a system-assigned handle that keeps them blinded.

SCORING METHODOLOGY

One trusted judge. Each experiment names a single judge model that does all the scoring — the industry-standard, reference-free "LLM-as-a-judge" approach, needing no reference answer. During a round it grades every candidate to pick the winning answer and drive any self-improvement loop; the chosen answer's grades are the round's scores. Its factors are configurable per experiment (built-in default: groundedness, relevance, coherence, instruction-following → a weighted 0–100 composite). One judge on one rubric, applied identically to every run of an experiment, is what makes comparison across that experiment's runs valid — no per-configuration judge, no second scoring pass. Validating the judge against people, if ever wanted, is an offline step from an export of rounds and their scores.

Analytics. Comparison happens at query time, one variable at a time: the system compares only runs that share every value except the one under study, so any difference in their scorecards traces to that variable. Runs can be produced in any order, with no pre-planned sequence — the comparison stays valid because one trusted judge graded them all the same way.

CAPABILITY MATRIX

Model configs	Six modes: `single` · `synthesize` · `vote` · `consensus` · `judge_best` · `moe`. An optional Mixture-of-Agents pass, in which several models improve on each other's answers, loops improve → combine → score → refine, stopping on a score target, a minimum gain, or a maximum number of iterations, and pruning models that underperform.
Mixture of experts	The `moe` mode routes instead of running every model: a router picks one member — the expert — so a round costs one generation whatever the member count. It matches the prompt by meaning against each model's capability sentence (free, no model call), escalating only a tie to the trusted judge at temperature 0. Every routing decision is recorded per round.
Prompt enhancement	Optional switch (`enable_prompt_enhancer`, off by default): before answering, the middle tier rewrites the enrollee's raw prompt into a sharper one, aimed at the rubric — one pass on the judge model, under a versioned enhancer asset. Both raw and effective prompts are kept; frozen per experiment, so a study stays comparable.
Conversation	**Chat memory** (`enable_chat_history`) replays earlier turns, bounded by the **smallest** context window in the configuration. **Socratic mode** (`socratic_enabled`) decides how the sufficiency gate replies when it stops a request — the judge asks for one missing piece at a time, or asks for a rewrite — under wording frozen on the experiment. Stopped attempts count in `retry_count`.
Controlled parameters	Each run varies exactly four: the nudge (guidance shown to the enrollee), the model configuration, the cohort, and the Socratic switch. Everything else is held constant across the experiment's runs — the system prompt (built as a two-layer cascade), the opening message, the scenario, the context files, and the scoring configuration.
Scoring rubric	One trusted judge scores on a rubric that is configurable per experiment — the default four factors (groundedness, relevance, coherence, instruction-following) or a custom rubric picked from the middle tier's bundled registry — yielding a weighted 0–100 composite. The same judge grades every run, so scores are comparable within an experiment.
Live observation	Experimenters watch a run stream live, drill into any enrollee's session exchange-by-exchange, and message enrollees out-of-band. Operational recovery — retry a failed model call, pause or abort a run, withdraw an enrollee — runs alongside a live session without changing its frozen set-up.
Integrity	A run's set-up stays fixed because the definitions it references — the experiment, nudge, model configuration, and cohort — are held immutable while any run references them, so a finished run's results stay valid without a stored copy. A disjoint-cohort gate keeps any enrollee from joining two live runs at once. Cost is split per round between generating answers and judging them.
Ask engine	A question in plain language becomes a read-only SQL (Structured Query Language) database query, run under the asker's own row-level security. Saved queries take parameters and are matched to a question by meaning. Each answer is a short narrative plus zero or more downloadable tables and/or charts; run results (CSV/XLSX/JSON) and transcripts (text) also export for outside tools.
Retrieval (RAG)	Retrieval-augmented generation (RAG) brings relevant document text into a run's chats. Each document is extracted, using optical character recognition (OCR) for scanned files, then split into chunks and embedded into one shared HNSW index with pgvector. The relevant chunks are retrieved into a run's chats.

SPECIFICATIONS

Database	PostgreSQL · 24 tables across 4 modules · row-level security throughout
Roles	admin · experimenter · enrollee
Scoring	rubric factors (4 by default), chosen from a built-in list → weighted 0–100 composite · one trusted judge, applied identically across every run
Models	Provider-agnostic catalog · single or ensemble · hosted or local
Embeddings	pgvector · HNSW (cosine) · fixed 1024-dim
Front end	Single-page app on Cloudflare Pages
Middle tier	Persistent Python service on fly.io (holds the provider API keys)
Auth / Realtime	Supabase · passwordless (OAuth + email one-time passcode, or OTP)