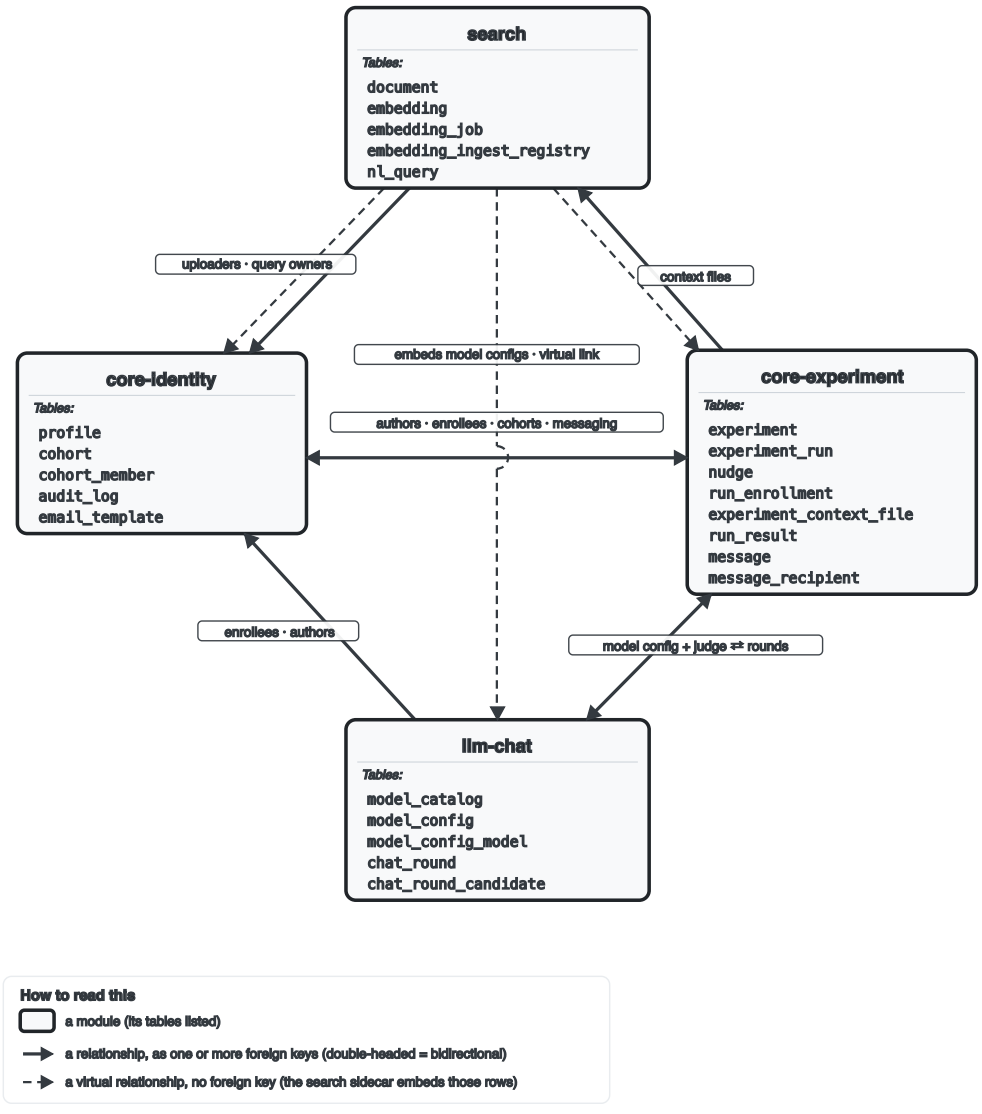
ChatMaestro — Data Model: Overview

Revision: 2026-08-30

For ease of interpretation, the tables of the schema (a database's complete set of table definitions) can be viewed as organized into four modules. A module is a reading aid only; it has no counterpart in the SQL (Structured Query Language) schema, where every table lives in one flat namespace (all table names in a single shared list, with no grouping). The map below shows the four modules and their interconnections. Each line between two modules stands for one or more relationships between individual tables of those two modules — the modules themselves are not connected, since they are not real database objects. A solid line marks real relationships, realized as foreign keys (a foreign key is a column whose value must match a row's identity in another table); a dashed line marks virtual relationships, which the search sidecar (a self-contained companion service that runs alongside the main app) forms by embedding another module's rows with no foreign key. An arrow points to the table, and so the module, being referenced; a double-headed line means each side references the other. One per-module ERD (entity-relationship diagram) backs each box.

Module Map



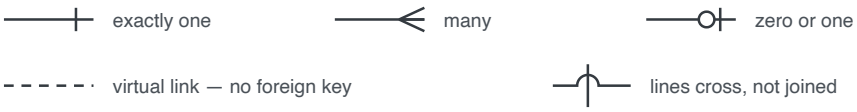
Entity-Relationship Diagram (ERD) Notation

The per-module ERDs share the following notation, loosely based on the crow's-foot, or Information Engineering, notation system (a widely used convention for drawing how database tables relate). Each box is a table, the badges mark its key columns, and the lines mark how tables relate. These diagrams — this module map and the per-module ERDs — are a visual aid for understanding the entities and their relationships; they are not a functionally complete definition of the schema, which lives in `schema.dbml`.

Column keys

PK	primary key — the row's identity
FK	foreign key — points to a row in another table
UK	unique — no two rows share this value
PK · FK	both at once — part of the key and a foreign key

Relationship lines



Narrative

The Four Modules

- **core-identity** (5 tables): the people, each with a role of admin, experimenter, or enrollee (a person who takes part in a study run); the cohorts they are grouped into; the full `audit_log`; and the outbound email templates. Almost every other module has a foreign key back to `profile`. There is no install-settings table, because install config lives in backend `.env` (a plain-text file of deployment settings), and the admin is the owner.
- **core-experiment** (8 tables): the reusable experiment definitions; each concrete run — one execution of an experiment — against a cohort (a named group of enrollees); the enrollees enrolled; the per-run `run_result` scorecard; any files attached as experiment context; and the out-of-band (outside the chat itself) operator↔enrollee messaging, scoped to a run. This is the spine of the platform.
- **llm-chat** (5 tables): the large-language-model (LLM) side, which is the admin-curated `model_catalog` and the reusable `model_config` with its constituent models, combine policy, and self-improvement; and the runtime, which is every scored `chat_round` and the per-model and per-iteration candidates behind it. One trusted judge (an LLM configured on the experiment to score answers) does all scoring.
- **search** (5 tables): a generic semantic-search sidecar, described in the note below, plus the `nl_query` Ask engine, which answers questions typed in plain language. It turns documents and other content into vectors (lists of numbers that place similar content close together) so the app can find things by meaning.

How the Modules' Tables Connect

Real relationships, the solid arrows, are foreign keys between tables in the two modules.

- **core-experiment and core-identity**, bidirectional: experiments are authored by people, each run records who launched it and targets a cohort, runs enroll enrollees, and messages carry a sender, recipients, and an optional cohort broadcast — through `experiment.created_by`, `experiment_run.launched_by / cohort_id`, `run_enrollment.enrollee_id`, and `message.*`. The reverse direction is one link: a run-scoped audit entry points back at its run, through `audit_log.run_id → experiment_run`, which is why this pair is bidirectional.
- **core-experiment and llm-chat**, bidirectional: a run picks the model configuration (which models generate answers, and how their outputs combine) it applies and the judge it scores with, through `experiment_run.model_config_id` and `experiment.score_judge_catalog_id → model_catalog`. And every scored `chat_round` and its candidates belong back to the run and its enrollees, through `chat_round.run_id / enrollment_id`. Both directions carry real foreign keys, so the pair is bidirectional. (The per-run `run_result` scorecard now lives in core-experiment, hanging off `experiment_run`.)
- **llm-chat to core-identity**: a model configuration records its author, and each chat round remembers the enrollee, through `model_config.created_by` and `chat_round.enrollee_id`.
- **core-experiment to search**: an experiment can attach documents as shared context for all its runs, through `experiment_context_file.document_id → document`.
- **search to core-identity**: an uploaded document remembers who added it, and a saved question remembers its owner, through `document.uploaded_by` and `nl_query.owner_id`.

Virtual relationships, the dashed arrows, are not foreign keys. They are opaque, generic row references: the search sidecar records, with each `embedding` row, a plain `(source_schema, source_table, source_id)` triple that names a target table and row by value, with no foreign-key constraint declared — which is what lets the target live in another schema or a remote system. The registered sources are individual tables, and the references run table to table: `embedding` to `cohort` and `email_template`; to `experiment`, `experiment_run`, and `nudge`; to `model_config` and `model_catalog`; and to `document` and `nl_query`. The map draws the search-to-llm-chat link explicitly, because `embedding`→`model_config` / `model_catalog` is the only relationship between any table in those two modules; the references from `embedding` into core-experiment and core-identity tables run alongside the existing foreign-key edges there. One more opaque row reference is internal to reporting: `audit_log.target_type / target_id` names a row in any table.

Indexing and Access Paths

Beyond the indexes that come with every primary key, unique constraint, and foreign key, the schema carries secondary indexes chosen from the documented query paths. They are declared next to each table, in the `indexes` blocks and table notes of `schema.dbml`. The families in use:

- **Secondary B-tree indexes** (B-tree is the default, general-purpose sorted index) for the hot read paths — reverse lookups (an enrollee's cohorts and runs, a document's runs, a reader's inbox), transcript order (`chat_round(enrollment_id, session_seq, round_seq)`), run- and enrollee-scoped aggregation, per-model analysis (`best_model`, `worst_model`, `chat_round_candidate(model_id)`), and queue polling (`embedding_job(status, requested_at)`).
- **Partial indexes** (each covers only the rows matching a condition) that both enforce a rule and speed the narrow scan — the single admin (`WHERE role = 'admin'`), one default email per kind, and one open embedding job per source.
- **HNSW vector indexes** (HNSW, Hierarchical Navigable Small World, is an index for fast nearest-vector search; supplied by pgvector, the PostgreSQL — Postgres — extension that adds vector columns) for semantic similarity — the one shared index on `embedding.vector`, and a separate one on `nl_query.query_vector` for matching a typed question to a saved one.
- **GIN full-text indexes** (GIN, Generalized Inverted Index, searches inside composite values such as text or JSON) for keyword search over the human free-text columns of each text-bearing table — the same columns each contributes to its embedding, plus `profile.notes` and `audit_log.note`, which keep keyword search but are held out of the semantic index for privacy.
- **A GIN index on `embedding.filter_values`** for scoped similarity search, which filters by jsonb (Postgres's binary JSON column type) containment (@>) so a search narrows to one experiment, cohort, or category without a join.

The highest-volume tables — `chat_round`, `chat_round_candidate`, and `audit_log` — are ordinary tables with simple `id` primary keys. Deep paging uses keyset pagination (paging by a cursor value rather than a numeric offset) — `ORDER BY (created_at, id)` with a `WHERE (created_at, id) < (:cursor)` cursor, backed by the `(created_at, id)` indexes on those tables — which stays fast at any depth, so partitioning is not needed for paging; counts use `count(*) OVER()`. Range partitioning (splitting one table into sub-tables by a key range, for instant DROP-PARTITION retention and smaller per-partition indexes) is a scale option to revisit only if these tables reach the tens of millions of rows, a known migration deliberately deferred to keep the primary keys and foreign keys simple now. Postgres has no maintained clustered index (rows kept physically ordered by a key), no persistent bitmap index (the planner builds bitmap scans from ordinary B-trees at query time), and hash indexes are avoided in favor of B-tree. The exhaustive audit of the remaining secondary indexes is deferred to a later performance-tuning pass: once the app runs against real data, they are added by reading query plans, not guessed at design time.

A Note on the Search Module

The search module is generic, and nothing in it is specific to ChatMaestro. It can embed the rows of any table, in this database, another schema, or even a remote system, and query them by similarity, through that virtual source pointer with no foreign key, which is exactly why its cross-module links are dashed. It is designed to be extracted into its own standalone service that ChatMaestro, and other applications, connect to. On this map it is shown as one of ChatMaestro's modules, but think of it as a self-contained building block plugged in here.