ChatMaestro — schema.dbml

Revision: 2026-09-20

This is the authoritative definition of the database, rendered for reading: every table, column, data type, enumerated value, and foreign key. It is written in DBML, a plain-text language for describing a database schema. The entity-relationship diagrams are generated from this file, so they are views of it rather than a separate source. The unformatted source text is also reproduced in full at the foot of this document.

CONTENTS

core-identity profile · cohort · cohort_member · audit_log · email_template

core-experiment experiment · nudge · experiment_run · run_enrollment · experiment_context_file · message · message_recipient

llm-chat model_catalog · model_config · model_config_model · chat_round · chat_round_candidate · run_result

Search and embeddings (a generic bring-your-own-index sidecar) document · embedding · embedding_job · embedding_ingest_registry

The Ask engine (natural-language queries) nl_query

Issue tracker (bug reports and product feedback) issue

Overview

ChatMaestro — the authoritative database schema

This file is the machine-readable specification of the database. It is written in DBML, a plain-text language for describing a database schema. Three other artifacts are generated from it, so each is a view of this file rather than a separate source:

1. the entity-relationship diagrams, which are hand-authored vector drawings of the tables and of the relationships between them;
2. the SQL data-definition language, or DDL — the CREATE statements that build the database. Those come from the declarative body below together with the raw DDL supplement at the foot of this file. The supplement carries the constraints and indexes that DBML syntax cannot express: CHECK constraints; partial, expression and vector indexes; GIN indexes, GIN being the generalized inverted index type; and extensions. The body and the supplement together are the complete and accurate DDL.

This is a physical model, so it describes the database as it is actually built. Table names are singular, and the file carries every primary key, foreign key, unique constraint and CHECK constraint. It targets PostgreSQL 15 or newer, the floor set by the NULLS NOT DISTINCT clause on the `model_config_model` member-uniqueness index; Supabase is well past it.

A few values are stored twice on purpose. Each deliberate duplicate is marked DENORM, short for denormalization, together with the reason it is safe: it copies either a value that never changes or a value captured at a deliberate point in time, so the two copies cannot drift apart.

Every app-minted primary key defaults to gen_random_uuid(), which is built into PostgreSQL and needs no extension. The middle tier still generates ids itself on the normal path; the default is what lets a seed script, a migration or a hand-written insert work without one. Two uuid primary keys deliberately carry no default, because neither is the app's to mint: `profile.id`, which equals auth.users.id, and `run_result.run_id`, which is the `experiment_run` it scores.

No installation settings live in the database. The email and storage providers, every credential, the feature flags and OWNER_EMAIL all live in the backend .env file, and the service is rebooted to apply a change. The admin role is the installation's owner: the single profile whose email address matches OWNER_EMAIL, reconciled each time the service boots. The note headed "Admin and owner" below explains how.

Key Concepts

Installation, Schemas and Modules

This is a single-tenant install, meaning one organization per deployment. All objects live in a single Postgres schema, whose name is configurable and defaults to chat_maestro; several such schemas can coexist in one database (one per project or version), each dropped in one shot with DROP SCHEMA.

The modules — core_identity (profile, cohort, `audit_log`, `email_template`), core_experiment (including messaging), llm_chat (including `chat_round` analytics and the `run_result` scorecard), and search — are logical sub-domains, each with its own entity-relationship diagram (ERD). They are not separate Postgres schemas. Row-level security (RLS), the database's per-user access rules, enforces the ranking of admin over experimenter over enrollee.

Two operational points are not columns. The first is connection handling: the middle tier is a persistent Python service on fly.io that holds its own connection pool, so Supavisor, the Supabase connection pooler, can run in session mode and prepared statements can stay on. It propagates the end-user's JWT (JSON Web Token, the signed login token) by setting request.jwt.claims so that RLS applies, and it never queries high-volume paths with the service-role key, which would bypass RLS.

The second is paging. The high-volume tables — `chat_round`, `chat_round_candidate` and `audit_log` — stay ordinary tables. Analytics queries page through them with keyset pagination, which asks for the rows after a given key value, (`created_at`, id), rather than counting past them with a numeric offset, so deep pages stay fast without partitioning. Those queries obtain their total row count with count(*) OVER().

Optimistic Concurrency Control

Optimistic concurrency control, abbreviated OCC, stops two people from silently overwriting each other's edits. Every row that two people can edit through a form carries a version counter. An update supplies the version the form loaded and succeeds only if the stored version still matches, and the counter then increases. If it does not match, someone else saved first, no row is updated, and the edit screen reports that the record changed and asks the reader to reload it. This protection is applied only where two users can edit the same row through a form.

Nine tables carry a version counter: experiment, `model_config`, `model_catalog`, cohort, nudge, `email_template`, profile, document and `nl_query`.

The other sixteen tables carry none, for one of six reasons.

- Rows that are only ever added, or written once and never changed, are never read, modified and written back, so no update can be lost: `audit_log`, `chat_round`, `chat_round_candidate`, `run_result`, `run_enrollment`, message and `message_recipient`.
- Rows driven by a state machine are guarded instead by an update that names the state it expects to find: `experiment_run`, whose setup is frozen at launch so that only its state changes, and `embedding_job`, which is a work queue.
- A child row that belongs to a parent is versioned through that parent, so editing a member raises `model_config.version`: `model_config_model`.
- A row that records membership is inserted or deleted under a unique key and is never edited in place: `cohort_member` and `experiment_context_file`.
- Derived data is regenerated rather than hand-edited: embedding.
- A row with a single owner and little contention needs no counter: `embedding_ingest_registry`, which holds installation configuration that the admin maintains.

Admin and Owner

(a developer note)

There is exactly one admin, and that admin is the installation's owner. The admin role is the owner record itself: there is no settings table and no stored identifier. OWNER_EMAIL in the .env file names which account holds it, and that is re-asserted every time the service boots.

At boot the middle tier resolves OWNER_EMAIL to a Supabase Auth account — case-insensitively, since Auth holds addresses lowercased — and writes the admin role onto the profile with that account's id, creating the profile if the account has never logged in. The match is on the account id and never on `profile.email`, which the same step writes rather than reads, so a drifted email cannot mislead the reconciler or cost the owner their role. It then demotes any other admin to experimenter. It demotes first, so there are never two admins at once. The database enforces the single admin with a partial unique index, which is a uniqueness rule that applies only to the rows matching a condition — here, only to rows whose role is admin. See `profile.role`.

Sign-up is by invitation only, so the owner's account is created in the Supabase dashboard rather than in the app. Because OWNER_EMAIL is re-asserted at every boot, the app offers no way to transfer ownership or to change the owner's email address. Changing the owner means editing the account in the dashboard, editing OWNER_EMAIL, and rebooting. If OWNER_EMAIL names no Supabase account, the boot fails loudly and leaves the installation with no admin.

Audit entries written under a previous owner keep the name recorded at the time, in `audit_log.actor_label`. They are never rewritten, because they record who acted then. Only the admin adds and manages enrollees; experimenters group the existing pool of enrollees into cohorts.

Ownership and Transfer

(a developer note)

Every definition row that can be owned records two separate facts. The first is `created_by`, or `uploaded_by` on a document: the composer who first authored the row. It is a record of provenance and never changes. The second is `owner_id`, the current owner; it defaults to the composer at creation and can be reassigned. If the owner's profile is deleted, `owner_id` is set to null, which leaves the row orphaned and managed by the admin.

Row-level security grants the owner full access to create, read, update and delete the row. Peers may read it and use it in their own work, but not edit it. The admin overrides all of this.

The admin can transfer ownership by reassigning `owner_id`, which the audit trail records as a transfer action. The new owner may be another experimenter or the admin. A transfer is permitted only while the current owner's experiments have no running or paused runs.

Five tables carry `owner_id`: experiment, cohort, `model_config`, document and `nl_query`. Three others are admin-managed and are never transferred: `model_catalog`, `email_template` and `embedding_ingest_registry`. Enrollees are implicitly owned by the admin.

Rows that a run produces keep a mark of whoever initiated them and are never transferred: `experiment_run.launched_by`, `chat_round`, message and `audit_log`.

A document's ownership depends on its purpose. A context document, which grounds an experiment's chat, is owned by an experimenter and can be transferred under the rules above. A reference document, which is methodology the Ask engine draws on, is admin-managed in the same way as the model catalog: the admin creates, reads, updates and deletes it, experimenters read and use it, and it is never transferred.

Scoring: One Trusted Judge

(a developer note; no table of its own)

Scoring is done by one trusted judge per experiment. The judge is named in experiment.score_judge_*, and it applies a rubric that is configurable per experiment — four factors by default — with weights, both seeded from an installation default. This is the reference-free method known as LLM-as-a-judge, where LLM stands for large language model: the model grades each answer against stated criteria, with no human-written model answer to compare it against. The same judge grades every candidate to pick a round's winner and to drive the self-improvement loop, and its per-round scores, held in `chat_round.factor_scores` and `score_composite`, are what make an experiment's runs comparable within it: one judge, one scale, and no second scoring pass. Scoring is a measurement. It fills the score columns that the analytic queries rank by, while the aggregation and comparison themselves are plain SQL over `chat_round` and `run_result`. Four things are deliberately not modeled: several judges, re-judging, a judgment row per turn, and human calibration inside the app. Validating the judge, if that is ever wanted, is done in a separate tool fed by an export of the rounds and their scores.

Semantic Index and Embeddings

(a developer note)

One shared vector index makes curated text searchable by meaning. It is a single HNSW index over `embedding.vector`, HNSW being the hierarchical navigable small world graph, the index type that makes nearest-vector search fast. Document chunks and the rows of registered database tables are not indexed separately: they are all rows of embedding and they share that one index, with a query narrowed to the part it wants through `filter_values` rather than by picking a different index. The one vector index that stands apart is on `nl_query.query_vector`, which matches a typed question against saved questions and so answers a different question entirely.

Keyword search stays available on the free-text columns through GIN full-text indexes, described in the keyword and full-text note below. Meaning-based search is therefore additive: it adds a second way to find something and never replaces keyword search, so a short note is reachable either way.

The rule is that every notes or free-text annotation column is embedded into the shared index, with two privacy exceptions that stay searchable by keyword alone. Those are `profile.notes`, which holds personally identifiable information (PII) about people, and `audit_log.note`, which is part of the forensic trail. Holding those two out keeps personal data in the shared index to a minimum, as a second line of defense.

Content tables also embed their richer text. A document contributes its name and its extracted-text snapshot; an experiment contributes its name, description, scenario, prompts and opening message; and a nudge contributes its name and its steering text.

Some tables are deliberately not sources. The sidecar's own plumbing — embedding, `embedding_job` and `embedding_ingest_registry` — does not embed itself. The chat transcript (`chat_round` and `chat_round_candidate`) and the out-of-band message are held out because of their volume and because blinding limits what may be surfaced; they are reached instead through the natural-language-to-SQL side. `nl_query` embeds its notes into the shared index like any other table, and separately keeps its own `query_vector` on an index of its own for matching typed questions.

Which columns each source contributes is recorded in `embedding_ingest_registry.text_template`.

Keyword Search, Full-Text Search, and jsonb Access Paths

(a developer note; the exact CREATE statements are in the raw DDL supplement)

Beyond the primary-key, unique and foreign-key indexes that arrive with the constraints, the documented query paths need three further kinds of index.

1. Keyword and full-text search over text that people write. Each text-bearing table carries one GIN full-text index over its free-text columns, meaning its names, content and notes. That includes the two columns held out of the semantic index, `profile.notes` and `audit_log.note`, which are searchable by keyword even though they are not embedded. Substring and fuzzy matching would need a pg_trgm GIN index instead; that is a separate tool and is not added.

2. Filtering by scope inside a jsonb column, jsonb being Postgres's binary JSON type. `embedding.filter_values` carries a GIN index built with jsonb_path_ops and is queried by containment, using the @> operator. Every other jsonb column is read whole rather than filtered by key, so none carries a jsonb index yet. One is added when a real per-key filter path appears and its query plan calls for it — see the deferred index audit.
3. Vector similarity. There is one HNSW index on `embedding.vector` for the shared content, and one on `nl_query.query_vector`.

Three properties of Postgres itself are worth stating. It has no persistent bitmap index; the planner builds bitmap scans from ordinary B-tree indexes at query time. It has no maintained clustered index, so append-heavy data is ordered through the (`created_at`, id) B-tree that keyset pagination already uses. Hash indexes are avoided in favor of B-tree. The exhaustive audit of the remaining secondary indexes, driven by reading real query plans with EXPLAIN, waits for a later performance-tuning pass, once the application runs on real data.

Enums

role `admin` `experimenter` `enrollee`

run_state `running` `paused` `done` `aborted`

combine_method `single` `synthesize` `vote` `consensus` `judge_best` `moe`

moe is MoE (mixture of experts) routing: a router picks one member, the expert, to answer, instead of running them all. ALGORITHMS §19 specifies it.

decline_kind `appropriate` `unwarranted`

How a response answered short of attempting the request, and null when it attempted it. An appropriate refusal takes relevance to N/A, since it is not trying to address the request; an unwarranted refusal keeps relevance scored. ALGORITHMS §4 specifies the verdict. The models under test never ask clarifying questions, so no round is ever a question: eliciting a missing detail is the sufficiency gate's work, ahead of the round (ALGORITHMS §21, §30).

stop_reason `score_target` `min_gain` `max_iterations` `single_pass`

Why a round's self-improvement loop ended, where self-improvement is the loop that repeatedly refines an answer.

message_target `direct` `cohort` `everyone` `operator` *The operator value marks a message an enrollee sends to the run staff.*

candidate_disposition `chosen` `pruned` `none` *A candidate's final state: chosen is the winner, pruned was dropped mid-loop, and none was generated but neither chosen nor pruned.*

embedding_job_status `pending` `running` `done` `failed`

extraction_status `pending` `ready` `stale` `failed` *The state of a document's cached extracted-text snapshot.*

query_status `draft` `approved` `system` *The trust and visibility level of a saved `nl_query`, a saved natural-language question.*

email_kind `invite_experimenter` `invite_enrollee` `run_ready`

The kind of outbound email: `invite_experimenter` invites an experimenter, `invite_enrollee` invites an enrollee to the pool, and `run_ready` tells an enrollee their run is ready. All kinds are admin-managed.

document_purpose `context` `reference`

A document's category. A context document grounds the enrollee chat for one run through retrieval-augmented generation (RAG), which supplies relevant text to the model. A reference document is admin-owned methodology that the Ask engine pulls by meaning. The set is extensible, so each new category becomes a new retrieval scope.

issue_kind `bug` `feedback`

The two kinds an issue row can be: a bug report or a piece of product feedback. One table backs both surfaces; severity is gated to the bug kind by a CHECK (see the issue Note).

issue_severity `low` `med` `high` *Set on bug rows only (null on feedback): how bad the reported defect is.*

issue_status `new` `triaged` `closed` *The triage lifecycle of an issue: new (unseen), triaged (an admin has dispositioned it), closed (resolved or dismissed).*

core-identity

> Tables in this module: profile, cohort, `cohort_member`, `audit_log` and `email_template`.

profile TABLE

11 columns

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	This id equals auth.users.id, so a person's identity is this UUID (universally unique identifier). The sign-in itself lives in Supabase Auth, whether through a third-party provider (OAuth) or a one-time passcode (OTP). Every join in the schema uses id, never email or username. It is the one surrogate key with no gen_random_uuid() default. The value belongs to Auth, so defaulting it would let an insert mint a fresh id and leave a profile with no account behind it.
role	role	NN	The account's role, one of admin, experimenter, or enrollee. Row-level security (RLS), the database's per-user access rules, keys off this value. The admin role is the install owner, so there is no separate owner record. At most one admin exists. A partial unique index that covers only rows where role = 'admin' enforces the single admin. On boot the middle tier reconciles this column to the OWNER_EMAIL setting in the .env file. It resolves that address to its Supabase Auth account and writes the matching profile as admin, creating the profile if the account has never logged in. It then demotes any other admin to experimenter.

			The owner account is created in the Supabase dashboard, since sign-up is invite-only; every owner change is a dashboard + OWNER_EMAIL + reboot operation, not an in-app one. Only the admin adds and manages enrollees; experimenters group the existing pool into cohorts but cannot add or remove enrollees.
email	text	UK1 NN	The account's email address, which is personal data (PII). Supabase Auth is the authority for it; this column is a copy the app keeps so it can display and query the email without calling the auth service. A user changes their email through Supabase's verified flow, which updates Auth first, and this copy then follows. Uniqueness is case-insensitive, enforced by a unique index on lower(email) in the raw DDL supplement rather than by the plain UNIQUE alone. Supabase Auth treats one address as one account regardless of capitalization, and this column mirrors Auth, so without that index the mirror could hold two rows for what Auth considers a single person. Nothing resolves anyone by this column. The owner is found by resolving OWNER_EMAIL against Auth and then matching the account id, and row-level security authorizes a request by reading `profile.role` for the id in the caller's token. This email is written by that reconciliation, never read by it, so a stale value costs nobody their access and the next boot corrects it. ALGORITHMS §14.
username	text	UK2 NN	A display-only handle, separate from the email and the login. It is assigned once, uses a restricted character set, and is normalized to lowercase — with a unique index on lower(username) in the raw DDL supplement enforcing that normalization rather than trusting the writer to apply it. Staff handles are readable. Enrollee handles are system-assigned from a single monotonically increasing counter, giving enrollee-0001, enrollee-0002 and so on, so they are unique by construction and need no collision check. They keep an enrollee's name off the screens an experimenter reads. That is a convenience rather than a guarantee, in two ways. A sequential handle reveals the order accounts were created, and the handle is the same in every study, so an experimenter who sees one across two runs knows it is one person. Both were accepted deliberately, because a handle that resisted correlation would also make ordinary analysis across a person's runs harder. The blinding the platform does enforce is `experiment.blinded`, which hides the study's conditions. No foreign key points to this column, so changing the username or the email never breaks a reference, because every join uses id.
enabled	boolean	NN	The suspension gate for the account. A higher-privilege user can disable, or lock out, a lower-privilege one, following the ranking of admin over experimenter over enrollee. A disabled account cannot sign in or act, but the account and its data remain. This is independent of deletion: a profile can also be hard-deleted, in which case its audit rows keep the actor through `actor_label` while `actor_id` is set to null. Setting enabled to false suspends the account; it does not prevent deletion.
opted_out_at	timestamptz		When this person withdrew themselves from the enrollee pool, and null while they are still taking part. It is set by the enrollee, never by staff, which is what separates it from enabled: enabled is a suspension somebody else applies, `opted_out_at` is a decision the participant makes about their own involvement. Both can hold at once and neither implies the other.

			Opting out is pool-wide because opting in was. An enrollee accepts an invitation to the pool rather than to a named study, and studies then reach them through cohort membership, so a per-study refusal would be declining something they were never separately asked about. Setting it withdraws them from every run they are currently in — each live `run_enrollment` takes a `withdrawn_at` in the same transaction — stops any further `run_ready` mail, and takes them out of cohort resolution, so a later launch cannot quietly re-enroll them. Their `cohort_member` rows are kept and filtered rather than deleted, because those rows record who a cohort held at the time and rewriting them would falsify the history of runs that already used it. Everything they produced stays. Rounds already answered keep their scores and still count in the statistics of the runs they belong to, for the same reason a rewound round does: the work happened, and removing it would bias those runs upward. The decision is reversible — an enrollee can opt back in, which clears this column but does not restore the enrollments it ended, since those runs have moved on. ALGORITHMS §28 specifies it.
prefs	`jsonb`		The person's own interface preferences, as jsonb. The keys the app writes today are theme (light / dark / system), density (comfortable / compact), text_size, and time_zone; enrollees write only theme and text_size, since the other two belong to the staff console. An absent key means the app default, so a null prefs is a valid, complete state rather than a missing row. It is jsonb rather than typed columns because these settings carry no behavior. Nothing in an experiment, a run or the scoring reads them, so adding a preference must never need a migration, and an unrecognized key left by an older or newer build is ignored rather than breaking a sign-in. Anything that DOES affect what an enrollee experiences belongs on the experiment, where it is frozen and uniform, not here where each person could set it differently.
notes	`text`		Free-text human notes about this person, for example an admin's note about an enrollee or a staff account. The notes are searchable through SQL keyword search but are kept out of the shared semantic index for privacy, because they are personal data (PII) about people. They are not shown to the person themselves.
created_at	`timestamptz`	NN	
updated_at	`timestamptz`	NN	
version	`int`	NN	A counter for optimistic concurrency control (OCC), a way to prevent lost updates. It is increased on every edit so that two people editing the same row at once cannot silently overwrite each other.

INDEXES

(id) **primary key** **unique**

(email) **unique**

(username) **unique**

`cohort` TABLE

10 columns

A reusable, named group of enrollees, assembled once and targeted by any number of runs.

Its membership lives in `cohort_member`. Two runs that are live at the same time must target cohorts with no enrollee in common, a rule the app checks at launch.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
name	text	UK NN	
created_by	uuid	FK	A foreign key to `profile.id` recording who first created the cohort, kept as a record of its origin. It does not change on an ownership transfer, and it is set to null when that profile is hard-deleted (see the Ref below); its email is snapshotted in `created_by_email` so the origin survives.
created_by_email	text	NN	The creator's email address, copied here when the cohort is created. It is a deliberate duplicate, marked DENORM for denormalization, following the same pattern as `audit_log.actor_label`. The copy means the record still shows who composed the cohort after that profile is hard-deleted and `created_by` becomes null.
updated_by_email	text		The email of whoever last edited this row, captured on each update (DENORM), preserving the last editor after their profile is deleted.
owner_id	uuid	FK	A foreign key to `profile.id`; when that profile is deleted this is set to null (see the Ref below). It is the current owner, which can change, and it defaults to `created_by` when the cohort is created; the admin can transfer it. Under row-level security (RLS), the database's per-user access rules, the owner can create, read, update, and delete the cohort, while peers can read it and use it to launch their own runs. It is null only when the owner was hard-deleted, which leaves the cohort orphaned and admin-managed.
notes	text		Free-text human notes about this cohort. They are searchable by SQL keyword and, as part of the shared semantic index that supports meaning-based search, by meaning.
created_at	timestamptz	NN	
updated_at	timestamptz	NN	

version	int	NN	A counter for optimistic concurrency control (OCC), a way to prevent lost updates. It is increased on every edit so that two people editing the same row at once cannot silently overwrite each other.

INDEXES

(id) primary key unique

(name) unique

`cohort.owner_id` → `profile.id` delete: set null

cohort_member TABLE

3 columns

Records which enrollees are in which cohort; an enrollee can belong to many cohorts.

Only the admin adds and removes enrollees from the pool, while experimenters compose cohorts from that existing pool. Deleting a cohort cascades these rows away, and the enrollees survive. A cohort's membership is immutable while any run references that cohort — no rows may be added or removed once it is in use, so "same cohort" always means the same population; to use a different membership, clone the cohort.

Deleting an enrollee is blocked while any membership row references them. Separately, `run_enrollment` also pins an enrollee who has taken part in a run through its not-null reference, so an enrollee can be deleted only when they are in no cohort and no run.

COLUMN	TYPE	KEYS	NOTES
cohort_id	uuid	PK FK NN	A foreign key to `cohort.id`; deleting a cohort cascades, removing its membership rows but never the enrollees themselves (see the Ref below).
enrollee_id	uuid	PK FK NN	A foreign key to `profile.id`, restricted on delete (see the Ref below): an enrollee who belongs to any cohort cannot be hard-deleted, so the admin must remove them from every cohort first.
added_at	timestamptz	NN	When this enrollee was added to the cohort. A cohort is edited freely until a run targets it, so this records the roster's own history rather than anything about a run.

INDEXES

(cohort_id, enrollee_id) primary key unique

(enrollee_id) non-unique *reverse lookup: which cohorts an enrollee is in*

> The delete rules on `cohort_member`'s foreign keys, stated explicitly: deleting a cohort clears its memberships, and deleting an enrollee is blocked while that enrollee is still a member of one.

`cohort_member.cohort_id` → `cohort.id` delete: cascade

`cohort_member.enrollee_id` → `profile.id` delete: restrict

audit_log TABLE

15 columns

The audit trail. It is anchored to the profile of the person who acted.

Rows are appended and are otherwise immutable, with one permitted edit: adding a note annotation, where RLS allows updating only the note column. RLS blocks every other update. It also blocks deletes, with three exceptions. The first is the admin's age-based purge. The second is the automatic cleanup that runs when a run is deleted, through the `run_id` cascade. The third is a run-scoped cleanup by an admin or the owning experimenter, which deletes that run's audit entries while keeping the run itself. Corrections never change an existing row; they append a new one, linked by the soft `corrects_entry_id` reference (kept an app-enforced soft reference, not a database self-foreign-key).

RLS also scopes visibility, so an admin sees all entries, an experimenter sees their own and those of their enrollees, and an enrollee sees their own. It is append-heavy and high-volume. Deep paging walks it with keyset pagination, asking for the rows after a given (`created_at`, id) key instead of counting past them with OFFSET.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
actor_id	uuid	FK	Who acted. It is a nullable foreign key to `profile.id`, enforced by a real database constraint whose delete rule is ON DELETE SET NULL; the Ref below declares it. The column holds a value while the actor's profile exists, and goes null if that profile is hard-deleted. From then on, `actor_label` preserves the identity. Unlike `corrects_entry_id` (a soft, app-enforced reference with no database foreign key), this is a genuine, database-enforced foreign key.
actor_label	text	NN	The actor's username or email, captured when the entry is written. It is a deliberate duplicate, which the schema marks DENORM, and it makes the entry self-contained so that the origin survives profile edits, deletion, and purge.
action	text	NN	The verb describing what happened, stored as extensible text rather than a fixed enum: create, update, delete, launch, abort, disable, invite, notify, share, transfer, cleanup, purge, or correct. (Adding a note to an entry is not an action row — it edits that entry's note column; see note and the table Note.)
target_type	text	NN	Which kind of entity was acted on, for example profile, `experiment_run`, or cohort. Every audited action names the entity it acted on, so this is always set. It is a virtual, polymorphic foreign key, meaning it can point at different tables, so the database enforces no foreign key on it.

target_id	uuid	NN	That entity's id, resolved through `target_type` for each entry. It is always set, for the same reason `target_type` is. It is a virtual, polymorphic foreign key, so the database enforces no foreign key on it.
before_state	jsonb		A snapshot of the row before the action, which is null for a create. It feeds the field-level before-and-after comparison view.
after_state	jsonb		A snapshot of the row after the action, which is null for a delete — with one exception. The entry that records a run being cleared (action cleanup or purge, `target_type` `experiment_run`) carries here the facts that exist nowhere else once the delete has run: {"mode": "cleanup" | "purge", "archive": "taken" | "declined", "format": "xlsx" | "csv" | "json", "object_key": "..."}, the key being where the archive was written in object storage (ALGORITHMS §13). Reaching an archive again is the exception, so its key lives in this entry rather than in a column of its own.
note	text		A free-text annotation ON this entry, such as "intended, not a typo", added later. Writing it is the one permitted update to an audit row — RLS allows updating only this column — and it does not create a new row. This is different from a correction, which never edits an entry but appends a new row pointing back at the one it supersedes via `corrects_entry_id`.
corrects_entry_id	uuid		A soft self-reference forming a correction chain: a `correct` entry holds the id of the earlier entry it supersedes. It is an application-enforced reference, not a database foreign key: the chain is validated by the middle tier, keeping auditing app-driven and uniform. Original entries are never edited or deleted; corrections are appended. This is null for normal entries.
run_id	uuid	FK	The run this entry belongs to (see the Ref below, which cascades on delete): it is set for run-scoped actions, so clearing a run erases its audit entries. It is null for account-level actions, such as invites and role changes, which outlive the runs. It is also null on the entry that records the clearing itself, which names the run through `target_type` and `target_id` instead: were it run-scoped, a cleanup would delete the only record of where its own archive went, and a purge would cascade it away.
source	text		This records the channel the action came through, for example web or api. It is null for actions internal to the system, which have no external caller.
ip_address	inet		The actor's IP address at the time of the action, kept as forensic context (type inet). Null when it is not available, such as a system-internal action.
session_id	text		The actor's session identifier when the action happened, for correlating a sequence of actions within one sign-in session. Null when not applicable.

created_at	timestamptz	NN	

INDEXES

(id) primary key unique

(actor_id) non-unique

(run_id) non-unique

(created_at, id) non-unique *keyset pagination over the append-only trail*

(target_type, target_id) non-unique

(corrects_entry_id) non-unique

The delete rules on the audit trail's foreign keys, stated explicitly. DBML records a delete action on a standalone Ref line rather than on the column itself.

`audit_log.actor_id` → `profile.id` delete: set null

`experiment.score_judge_catalog_id` → `model_catalog.id` delete: restrict

`audit_log.corrects_entry_id` points at another row of the same table, but it is not a database foreign key. The middle tier validates the chain of corrections instead, which keeps auditing driven by the application and uniform across every table.

`audit_log.run_id` → `experiment_run.id` delete: cascade

email_template TABLE

14 columns

The templates for the email the app addresses to a person, part of the core-identity module. There are three kinds, named by the kind column: an invitation to a new experimenter, an invitation to an enrollee joining the pool, and the notice that a run is ready. Every kind is admin-managed. Experimenters neither compose nor customize email. Exactly one template per kind is the default, a rule a partial unique index on kind enforces by covering only the rows whose `is_default` is true. Under row-level security (RLS), the database's per-user access rules, the admin can read and write these rows, and they are invisible to experimenters and enrollees.

How the mail is sent is separate from what these rows hold. The middle tier fills the merge tokens, sends over SMTP from the sender address configured in the .env file, and records each send in `audit_log` with action set to invite or notify. The one-time sign-in link the invitation kinds carry is minted by Supabase Auth rather than by this application, so the platform stores no credential of its own, and issuing a new one stops the previous link working. Its lifetime is a setting on that service, which INVITE_LINK_EXPIRY_HOURS mirrors at 72 hours so the mail can state it; the operator guide keeps the two in step.

The app also sends mail that has no template here at all, such as the notifications that go out when someone files a bug report or a piece of feedback; those are composed in the middle tier.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
name	text	UK NN	
kind	email_kind	NN	This names which outbound email this template is for. The value `invite_experimenter` marks the invitation an admin sends to a new experimenter. The value `invite_enrollee` marks the invitation to an enrollee joining the pool. The value `run_ready` marks the notice an enrollee receives when a run they are in is ready. All three kinds are admin-managed, and experimenters do not compose or customize email. Exactly one default exists per kind; see `is_default`.
subject	text	NN	The full email subject line. It accepts the same merge tokens as the body, so a subject can name the study or the recipient; see variables below for the list.
body	text	NN	The full email body, including the merge tokens that are filled in for each recipient when the email is sent; see variables below for the list.
variables	jsonb		Records the merge tokens this template uses. The vocabulary is not free-form: it is fixed for each kind by the middle tier, which binds every token at send time. The template editor lists the tokens its kind allows and inserts one on click, so a composer never types a token by hand, and a token outside that kind's vocabulary is refused when the template is saved. A token can still have no value when the mail goes out, since a profile can be hard-deleted and the authoring columns go null by design. A cosmetic token then falls back, so {{inviter}} becomes the installation's name and the mail goes out slightly less personal rather than not at all. The two link tokens refuse the send instead, because a message whose purpose is a link is worthless without one, and that refusal is recorded so an invitation nobody received can be diagnosed. ALGORITHMS §22 specifies the binding. The tokens are: 1. **{{username}}** — the recipient's display name, available to every kind 2. **{{study}}** — the experiment's enrollee-facing title, for `invite_enrollee` and `run_ready` 3. **{{inviter}}** — the person the invitation comes from: the experimenter running the study for an enrollee invitation, and the admin for an experimenter invitation 4. **{{invite_link}}** — the recipient's one-time sign-in link, for the two invite kinds 5. **{{run_link}}** — the link that opens the run that is ready, for `run_ready`.
is_default	boolean	NN	Marks the one default template per kind, enforced by a partial unique index on kind that covers only rows where `is_default` is true.

			The default of a kind is the template used whenever that email is sent. Experimenters cannot override it, because email is admin-only.
created_by	uuid	FK	A foreign key to `profile.id` recording the admin who authored the template, kept as a record of its origin. It is set to null when that profile is hard-deleted (see the Ref below); its email is snapshotted in `created_by_email`. Templates are admin-only, so there is no `owner_id` and no transfer.
created_by_email	text	NN	The author's email captured at creation time (DENORM, like `audit_log.actor_label`), preserving the origin after the profile is hard-deleted and `created_by` goes null.
updated_by_email	text		The email of whoever last edited this template, captured on each update (DENORM).
notes	text		Free-text human notes about this template. They are searchable by SQL keyword and, as part of the shared semantic index that supports meaning-based search, by meaning.
created_at	timestamptz	NN	
updated_at	timestamptz	NN	
version	int	NN	A counter for optimistic concurrency control (OCC), a way to prevent lost updates. It is increased on every edit so that two people editing the same row at once cannot silently overwrite each other.

INDEXES

(id) primary key unique

(name) unique

core-experiment

> Tables in this module: experiment, nudge, `experiment_run`, `run_enrollment`, `experiment_context_file`, message and `message_recipient`.

`experiment` TABLE

29 columns

The reusable study and comparison container.

It holds the invariant frame shared across all its runs: the prompts (system prompt, opening message, scenario), the enrollee-interface settings, the scoring configuration, and the attached context documents (`experiment_context_file`). The three per-run swept variables — the nudge, the model configuration, and the cohort — are referenced by `experiment_run`. The score_* fields are overridable copies of an install-level global scoring default, seeded when the experiment is composed.

The experiment is held immutable while any run references it. Its substantive fields and its set of context files are read-only once it has runs, and only notes and `run_defaults` stay editable. A finished run's setup therefore never drifts, and no per-run snapshot is needed. It becomes editable and deletable again only after its last run is purged.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
name	text	UK NN	The experiment's name, shown to staff. Enrollees see `enrollee_title` instead, so the name can describe the study's intent without unblinding them.
enrollee_title	text		The enrollee-safe short title shown to enrollees (for example "Open-topic chat"), kept distinct from name (which is staff-facing and may reveal the study's intent). The app prefills a head-started title derived from the experiment and scenario when the experiment is created, and the composer can edit it; if left blank, the interface falls back to a generic title.
description	text		A free-text description of the study, shown to staff only. Like name, it describes the study and can reveal its intent, so it is hidden from enrollees; enrollees see the enrollee-safe `enrollee_title`, scenario, and `opening_message` instead.
system_prompt	text		The standing instruction to the model, hidden from the enrollee, and the base of the system-prompt cascade. Two things are worth writing into it. Bound the topic, saying what the assistant does and does not cover, since an experiment with no context documents has no other way to state its scope. Then tell the model to ask what is missing rather than guess when a request is unclear. The sufficiency gate in ALGORITHMS §21 already turns back a request that leaves out a declared required slot, or that no context document comes close to; this instruction covers the requests no declared slot describes. There is no attempt limit, because a request is not accepted until it is well formed. It is part of the experiment's fixed frame — a control held constant across the experiment's runs, and read-only once the experiment has runs. The cascade resolves in two layers, in a fixed order: `model_config_model.system_prompt`, the model's own role inside the configuration, comes first, and this prompt is appended after it, so the study's instruction always has the last word. There is no mode to choose and no way for a model configuration to displace this text. The fully resolved prompt is what the model receives. There is no per-run layer, because the system

			prompt is not one of the swept variables; those three are the nudge, the model configuration and the cohort. The experiment and the model configuration are both immutable while a run references them, so the prompt a finished run resolved to is always recoverable by resolving the cascade again. No snapshot is kept.
scenario	text		The enrollee-safe framing, which is the session topic.
opening_message	text		This is the enrollee-safe first message from the assistant.
blinded	boolean	NN	The blinding gate for the enrollee interface, defaulting to true (blinded) so it fails safe. When true, the enrollee sees only the single chosen response and none of the machinery behind it. The "See all" viewer, which shows every model's output at every iteration, is hidden. File upload and download are disabled as well, so nothing about the condition can leave or enter the session. When false (visible), those capabilities are enabled. Chat conveniences unrelated to blinding (clear-session, rewind, example prompts) are always available and are not configured here.
time_limit_min	int		This sets an optional per-session time limit for enrollees; when set, the countdown is always shown to the enrollee.
score_judge_catalog_id	uuid	FK NN	A foreign key to `model_catalog.id` naming the one trusted judge for this experiment: the single model that does all the scoring. During a round it grades each candidate, to pick a winner and to drive the self-improvement loop; after the round it produces the comparable per-round scores. One judge on one rubric, applied identically to every run, is what makes an experiment's runs comparable within it and removes the need for a separate per-configuration judge. It is picked from the curated catalog rather than free-typed, so a model's identity lives once in `model_catalog`, the same rule `model_config_model` follows. A catalog row that any experiment names as its judge cannot be deleted; the delete rule is RESTRICT, the same protection a catalog row gets when a model configuration uses it. A model is taken out of service by setting `model_catalog.enabled` to false, which hides it from the pickers while every experiment that already names it keeps working. The `score_judge_provider` and `score_judge_model` columns below record which judge produced the scores in readable form. Keeping it a real foreign key also makes the anti-self-preference check possible at launch, warning when the judge shares a provider with one of the generators it will score.

score_judge_provider	text	NN	The provider of the experiment's one trusted judge, stored as a literal value. It is seeded from the install-level global default when the experiment is composed and can be overridden here. It is a deliberate duplicate (DENORM): it is the historical record of what actually scored the rounds and must survive catalog edits, retirement, and deletion, following the same pattern as `audit_log`'s `actor_label`.
score_judge_model	text	NN	This holds the model id of the experiment's one judge, and it is read together with `score_judge_provider`.
score_rubric_version	text	NN	The identifier of the rubric the judge applies — an immutable, versioned artifact that defines which factors are scored, their definitions, and their anchored 0-to-100 scales. It is not stored in the database; it names one entry in the middle tier's rubric registry, which bundles every available rubric (the built-in default plus any others shipped with the release). The composer does not free-type this value: the middle tier publishes the registry's list of rubrics, and the experimenter picks one from a drop-down, so the stored string is always a real, resolvable version. The built-in default is a four-factor answer-quality rubric (groundedness, relevance, coherence, instruction-following), but an experiment may select a different rubric whose factors suit its study — for example funniness, originality, and timeliness for a joke experiment. A rubric is frozen per experiment, so all its runs are scored the same way and stay comparable within the experiment; versioning lets a reworded rubric become a new version rather than silently changing past scores.
score_weights	jsonb		The per-factor weights for the composite score, keyed by the chosen rubric's factor keys, on a 0 to 100 scale. It is the one part of the scoring configuration that may be left unset. When it is null, the rubric's factors are weighted equally, as an arithmetic mean. A factor that does not apply to a round, for example groundedness where there is no grounding source, is dropped, and the remaining weights renormalize. The weights are seeded from the install default and can be overridden.
enable_prompt_enhancer	boolean	NN	Whether the middle tier rewrites the enrollee's raw prompt into a better-engineered one before the model set-up answers (see ALGORITHMS §18). It defaults to false, so the enrollee's exact words are used unless enhancement is opted into. It is a substantive field: like the prompts and scoring config, it is frozen once the experiment has runs, so the enhancement policy is uniform across a comparison and never confounds the swept variable. When true, the rewrite runs on the experiment's judge model (score_judge_*) under the instructions named by `prompt_enhancer_version`, and both the original and the enhanced prompt are stored on `chat_round`.
prompt_enhancer_version	text		The identifier of the prompt-enhancer instructions the rewrite applies — a versioned artifact, exactly like `score_rubric_version`.

			It is not stored in the database; it names one entry in the middle tier's enhancer registry, which the composer picks from a drop-down rather than free-typing. It is used only when `enable_prompt_enhancer` is true, and it is frozen per experiment so every run enhances prompts the same way. The rewrite is executed by the experiment's one trusted judge model (score_judge_*); a separately-pinned enhancer model is a future option.
human_review_enabled	boolean	NN	Whether a person approves or edits the enhanced prompt before it is submitted, rather than the rewrite going straight to the model set-up. It is gated on `enable_prompt_enhancer`, since there is nothing to review when no rewrite happens. In this version the column exists but is always false, and the compose and detail screens show it read-only so that its presence is visible and its state is unambiguous. Turning it on is a later version, and belongs to ad-hoc uncontrolled use rather than to a controlled study: a per-enrollee hand-edit is not a uniform condition, so it would break the comparability that makes a study's runs measurable. Reserving the column now means that change adds a screen rather than a migration.
enable_chat_history	boolean	NN	Whether an enrollee's earlier exchanges are replayed to the models on later rounds — that is, whether the assistant remembers the conversation. It is frozen once the experiment has runs, like the scoring and enhancer configuration, so memory is a uniform condition rather than something that varies by enrollee. When false, every round is self-contained: the models receive the system prompt, the retrieved context and this prompt alone. That is the honest setting for a controlled comparison of single answers, because it removes conversation length as a variable entirely. When true, the round carries as much recent history as fits. There is deliberately no per-experiment turn count beside this switch. A second limit on the compose screen invites confusion about which one is in force. Worse, an experiment-level depth would let memory vary with the model configuration, which is itself a swept variable. The depth is instead a platform constant (CHAT_HISTORY_MAX_TOKENS), bounded by the configuration's smallest `context_window`, so it is identical across every arm of a comparison. ALGORITHMS §23 specifies the assembly, the enrollee alert and the truncation behavior.
socratic_version	text		The versioned instructions the trusted judge composes the sufficiency gate's replies under, and null only on an experiment whose gate can never fire. It names an artifact in the middle-tier registry, chosen from a drop-down as the rubric and enhancer versions are, and it is frozen once the experiment has runs. A single artifact holds all four phrasings the gate can need: asking for a missing piece, or redirecting a request the material does not cover, each worded once as a question and once as a demand to rewrite. They are versioned together because the questioning wording and the rewrite wording are the two arms of the comparison, and freezing one while the other could change between releases would move the control arm without recording it. A version is required whenever the experiment attaches context documents or declares `required_slots`, and a run cannot set `socratic_enabled` without one. Which of the two styles a given run uses is `experiment_run.socratic_enabled`. ALGORITHMS §21 specifies the gate and §30 the switch.

required_slots	jsonb		The pieces of information a request must carry before this experiment will answer it, declared by the experimenter because the experimenter is the one who knows what the study is about. It is a list of entries, each naming a slot, saying in a phrase what it is, and marking whether it is required — for example [{"name":"topic","note":"which instrument, sector or market","required":true}, {"name":"time_horizon","note":"over what period","required":false}]. A null or empty list turns the check off, which is the default, so an experiment that wants no gate does not get one. The exception is a run with `socratic_enabled` set, where the judge falls back to reading each request itself, since a study of eliciting would otherwise have nothing to elicit. That fallback costs a judge call on every message rather than only on the ones it stops, and it never applies where a list is declared. It exists to make sufficiency a checkable property rather than a matter of each model deciding for itself. The alternative is a general catalog of task classes, which has to classify an incoming request before it can judge it, and a misclassification then demands information the request never needed. Scoping the declaration to the experiment removes the classification step entirely. Like the other substantive fields it is frozen once the experiment has runs, so every enrollee is held to the same standard. ALGORITHMS §21 specifies the gate that reads it. The experiment composer's Required Details section writes it, one prose entry per slot.
run_defaults	jsonb		A bag of seed values for a run's per-run settings, used only to pre-fill the launch form. It is not authoritative, because the run stores its own typed, foreign-key-enforced values that are frozen at launch. Its keys, for now, are: `model_config_id` and model_config_name (the default model configuration, with the name stored alongside so it displays readably without a join), nudge (a baseline steering message), and `keep_candidates`. Its keys are not free-form: each is set through a typed control on the experiment composer's Run-defaults tab, so nobody types a key name. A stale default, such as a reference to a deleted `model_config`, is simply ignored at launch, because it was never a foreign key. Cohort is deliberately not defaulted, because concurrent runs must target non-overlapping cohorts.
notes	text		Free-text human notes about this experiment. They are searchable by SQL keyword and, as part of the shared semantic index that supports meaning-based search, by meaning.
created_by	uuid	FK	A foreign key to `profile.id` recording who first authored the experiment, kept as a record of its origin. It does not change on ownership transfer, and it is set to null when that profile is hard-deleted (see the Ref below); its email is snapshotted in `created_by_email`.
created_by_email	text	NN	The author's email captured at creation time (DENORM, like `audit_log.actor_label`), preserving the origin after the profile is hard-deleted and `created_by` goes null.
updated_by_email	text		The email of whoever last edited this experiment, captured on each update (DENORM).

owner_id	uuid	FK	A foreign key to `profile.id` that is set to null when that profile is deleted (see the Ref below). It is the current owner, which can change, and it defaults to `created_by` at creation; the admin can transfer it to another experimenter or to themselves, but only while this experiment has no running or paused runs. Under row-level security (RLS), the database's per-user access rules, the owner can create, read, update, and delete the experiment, while peers can read it and use it to launch their own runs. It is null only when the owner was hard-deleted, which leaves the experiment orphaned and admin-managed.
created_at	timestamptz	NN	
updated_at	timestamptz	NN	
version	int	NN	A counter for optimistic concurrency control (OCC), a way to prevent lost updates. It is increased on every edit so that two people editing the same row at once cannot silently overwrite each other.

INDEXES

(id) primary key unique

(name) unique

nudge TABLE

11 columns

A reusable, named steering message (the nudge): an enrollee-safe instruction a study varies across its runs to see how it changes results.

A run references one nudge through `experiment_run.nudge_id`. Authored independently and reused, like a cohort or a model configuration. It is immutable while any run references it — its substantive fields (name, text) are read-only and it cannot be hard-deleted (RESTRICT); to change a nudge you clone it, which yields a new, distinctly-labeled condition. Only notes stay editable while referenced.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
name	text	UK NN	A short, unique display label for this nudge, for example "concise" or "thorough". It is the stable key that analytics comparisons group and label by, distinct from the message text below. Renames are blocked while any run references the nudge, so the label stays stable across a study.
text	text	NN	The enrollee-safe steering message shown to enrollees — the nudge itself, the thing a study deliberately changes between runs. It is authored once and reused across runs and experiments, like a cohort or a model configuration.

notes	text		Free-text human notes about this nudge. They are searchable by SQL keyword and, as part of the shared semantic index that supports meaning-based search, by meaning.
created_by	uuid	FK	A foreign key to `profile.id` recording who first authored the nudge, kept as a record of its origin. It does not change on ownership transfer, and it is set to null when that profile is hard-deleted (see the Ref below); its email is snapshotted in `created_by_email`.
created_by_email	text	NN	The author's email captured at creation time (DENORM, like `audit_log.actor_label`), preserving the origin after the profile is hard-deleted and `created_by` goes null.
updated_by_email	text		The email of whoever last edited this nudge, captured on each update (DENORM).
owner_id	uuid	FK	A foreign key to `profile.id` that is set to null when that profile is deleted (see the Ref below). It is the current owner, which can change and defaults to `created_by` at creation; the admin can transfer it. Under row-level security (RLS), the owner can create, read, update, and delete the nudge, while peers read it and use it in their own runs. It is null only when the owner was hard-deleted, leaving the nudge admin-managed.
created_at	timestamptz	NN	
updated_at	timestamptz	NN	
version	int	NN	A counter for optimistic concurrency control (OCC), a way to prevent lost updates. It is increased on every edit so that two people editing the same row at once cannot silently overwrite each other.

INDEXES

(id) **primary key** **unique**

(name) **unique**

`nudge.owner_id` → `profile.id` delete: set null

`experiment_run` TABLE

15 columns

One cell of an experiment's comparison grid, identified by the tuple (`experiment_id`, `nudge_id`, `model_config_id`, `cohort_id`) — the run's three per-run independent variables plus its container. The row is append-frozen: only state mutates after launch. A finished run stays faithful with no per-run copy to keep. The definitions it references — experiment, nudge, `model_config` and its models, and cohort with its `cohort_member` rows — are held immutable while referenced, enforced by a RESTRICT delete rule. Analytics therefore resolve a run's labels and setup by joining straight to those definitions.

Comparisons are done at query time: to isolate one variable, group an experiment's runs by the other two variables' ids and compare across the studied one.

A finished run can be cleared to reclaim its storage, and clearing is guarded and recorded. It is allowed only when the state is done or aborted, so running and paused runs are protected. It archives first by default: the transcripts, every candidate answer, the scorecard and the run's resolved conditions are written to object storage as xlsx, csv or json before anything is deleted, and if that write fails nothing is deleted. The operator may decline the archive — a botched run should not force an archive nobody wants — and that is a warned choice. Either way one `audit_log` entry records the mode, whether an archive was taken or declined, and the archive's object key; it is written with a null `run_id` so that the operation it records cannot erase it (see `audit_log.run_id`).

Clearing removes only what the run produced. It never touches the reusable definitions or the experiment's context files (`experiment_context_file`), and a definition becomes deletable only after the last run that references it is purged. ALGORITHMS §13 specifies the two ways of clearing, which have distinct intents:

1. **Cleanup** — reclaims space and keeps the record. It deletes the heavy data the run produced: `run_enrollment`, `chat_round` and its candidates, message and its recipients, and the run's own `audit_log` rows. This `experiment_run` row and its `run_result` scorecard stay, so the analytics remain queryable. The run's owner or an admin can perform a cleanup, the owner being whoever composed the experiment (`experiment.created_by`) or launched the run (`launched_by`).
2. **Purge** — removes the run altogether. It drops this `experiment_run` row and cascades `run_result` away with it (see the run Refs), so no scorecard is ever left dangling. Only an admin can perform a purge.

COLUMN	TYPE	KEYS	NOTES
`id`	`uuid`	PK	
`experiment_id`	`uuid`	FK UK NN	A foreign key to the experiment this run belongs to — the comparison container. It uses the RESTRICT delete rule (see the Ref below): an experiment cannot be hard-deleted while any of its runs exist.
`name`	`text`	UK NN	A human-readable label for the run, unique within the experiment (see the index). If left blank at launch, a default is generated and frozen from the experiment, cohort, `model_config`, and datetime.
`nudge_id`	`uuid`	FK NN	A foreign key to the nudge (a reusable, named steering message) applied in this run — one of the three per-run independent variables. The nudge is immutable while any run references it (RESTRICT; see the Ref below), so a finished run always shows the exact steering message it used, with no per-run copy to keep.
`cohort_id`	`uuid`	FK NN	A foreign key to the cohort applied in this run — a second per-run independent variable. The cohort is immutable while referenced (RESTRICT; see the Ref below), so its identity and membership at run time are preserved without a copy; the actual enrollees are pinned in `run_enrollment`.

model_config_id	uuid	FK NN	A foreign key to the model configuration applied uniformly to every enrollee in this run — the third per-run independent variable. The configuration and its constituent models are immutable while any run references them (RESTRICT; see the Ref below), so a finished run always resolves to the exact models, temperatures, and per-model prompts it ran with.
launched_by	uuid	FK	A foreign key to `profile.id` that is set to null when that profile is deleted (see the Ref below). It records the experimenter who launched this run, which serves both as a record of origin and as authorization to clear the run. An experimenter may clean up runs they composed (`experiment.created_by`) or launched (`launched_by`), and only when the run is not in progress.
state	run_state	NN	The run's lifecycle state: running, paused, done, or aborted. Only done or aborted runs may be cleared, and a paused run still counts as live for the non-overlapping-cohort rule. Reaching done is what triggers the `run_result` scorecard to be built.
socratic_enabled	boolean	NN	How the sufficiency gate replies when it stops an enrollee's message: with the switch on the trusted judge asks for one missing piece at a time and the request is built through dialogue, and with it off the judge names everything missing and asks for the whole request to be rewritten and resent. It is pre-filled from `experiment.run_defaults` at launch and can be changed there, which makes it the fourth per-run variable beside the nudge, the model configuration and the cohort. The switch changes the reply and not the detection, so the same missing piece stops the same message either way. An experiment declaring no `required_slots` is the exception: with nothing to test against, the judge falls back to reading the request itself, and only when this switch is on. What the models eventually receive differs too. A dialogue reaches them as question-and-answer pairs, while a rewrite reaches them as the enrollee's final request alone. They never see a stopped message and never ask anything themselves. Both phrasings live in `experiment.socratic_version`, so two runs of one experiment differ in the style the judge uses and in nothing else, and the launch form refuses the switch when no version is named. Stopped attempts are counted in `chat_round.retry_count` under either setting. ALGORITHMS §30.
keep_candidates	boolean	NN	Whether to retain this run's `chat_round_candidate` rows — the per-model and per-iteration detail behind each round — after the round is finalized. Default false clears them as each round completes (leanest storage, and the "See all" viewer still works live during the round); true keeps them for later side-by-side review. It is pre-filled from `experiment.run_defaults` at launch and can be overridden here. Candidate rows are also deleted whenever their `chat_round` or run is deleted, regardless of this setting.
notes	text		Free-text human notes about this run.

			They are searchable by SQL keyword and, as part of the shared semantic index that supports meaning-based search, by meaning. They are visible only to admins and experimenters, and they are frozen at run end.
started_at	timestamptz	NN	When the run's execution began, marking the start of the run window. It is distinct from `created_at`.
ended_at	timestamptz		When the run reached done or aborted, marking the end of the run window.
created_at	timestamptz	NN	When the run row was composed. It can precede `started_at` if the run is set up as a draft before launch.
updated_at	timestamptz	NN	When the row last changed. It is set when the run is created and refreshed on each state change, so it equals `created_at` until the run first moves, and it is never null.

INDEXES

(id) primary key unique

(experiment_id, name) unique

(cohort_id) non-unique

run_enrollment TABLE

6 columns

A reusable, named group of enrollees, assembled once and targeted by any number of runs.

Its membership lives in `cohort_member`. Two runs that are live at the same time must target cohorts with no enrollee in common, which the app checks at launch.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
run_id	uuid	FK UK NN	A foreign key to `experiment_run.id`; it cascades on delete, so purging a run removes this row (see the run-purge Refs).
enrollee_id	uuid	FK UK NN	Which enrollee this enrollment is for. The reference declares no delete rule, so the database refuses to remove a profile while any enrollment points at it, which is the same protection `cohort_member.enrollee_id` states as RESTRICT. That protection is why `chat_round.enrollee_id` can afford to be set to null instead. A person cannot be hard-deleted while their run still holds them, and by the time a Cleanup has removed the enrollments it has removed the rounds too. The consequence worth knowing is that an enrollee stays undeletable for as long as any run they took part in is kept.

joined_at	timestamptz	NN	When the enrollee was enrolled in this run, which is the moment the run was launched, because membership is resolved once at launch and fixed from then on. It is not when they first opened the chat.
withdrawn_at	timestamptz		When the enrollee stopped taking part in this run, and null while they are still in it. Their captured data is retained either way: rounds already answered keep their scores and still count in the run's statistics. Two things set it. Staff remove somebody from a run on the watch drawer, which is the operational case — a person who cannot continue, or should not. And the enrollee's own pool-wide opt-out sets it on every live enrollment they hold at once (ALGORITHMS §28). There is deliberately no per-study opt-out for the enrollee, because they were never asked per study: they accepted the pool, and studies reach them through cohort membership. A withdrawn enrollment stops the condition being served and is one of the presence states in ALGORITHMS §17. It is not a deletion and not reversible in place; re-including somebody means a later run.
last_activity_at	timestamptz		The time of the enrollee's last activity, used to show presence. An enrollee can leave and return, and doing so continues the same session.

INDEXES

(id) primary key unique

(run_id, enrollee_id) unique

(enrollee_id) non-unique *reverse lookup: runs an enrollee joined*

experiment_context_file TABLE

3 columns

A pure link table recording which documents are attached to an experiment as retrieval context, shared across all of that experiment's runs — context is an experiment-level constant, not a per-run variable. It scopes every run's enrollee retrieval-augmented generation (RAG), in which relevant document text is supplied to the model, to the experiment's documents. It links only documents whose purpose is 'context', a rule the app enforces rather than a database check, so a reference document never becomes enrollee RAG context.

The context set is part of the experiment's locked frame: it is immutable while the experiment has runs. Embedding readiness is derived from the search sidecar through the presence of `embedding_job` and embedding rows, not stored here, because a document is embedded once and shared across experiments and runs.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
experiment_id	uuid	FK UK NN	A foreign key to `experiment.id`; deleting the experiment cascades this link away (see the Ref below). Context files belong to the experiment's fixed frame, shared by every one of its runs, not to any single run.

document_id	uuid	FK UK NN	A foreign key to `document.id` with the RESTRICT delete rule (see the Ref below): a document attached to any experiment cannot be hard-deleted until it is detached. The shared document itself is never removed by an experiment delete.

INDEXES

(id) primary key unique

(experiment_id, document_id) unique

(document_id) non-unique *reverse lookup: experiments referencing a document*

message TABLE

7 columns

This is the out-of-band operator channel, not the chat with the language model.

It is two-way: `sender_id` can be any profile, so an enrollee can send a message with `target_kind` set to operator. A recipient row is created for each intended reader of every message (enrollees for direct, cohort, and everyone; the run staff for operator), so `read_at` works in both directions. A check constraint requires that `target_kind` is 'cohort' exactly when `target_cohort_id` is not null.

The sender's identity on the display side is not stored. The enrollee client shows a sender who is not themselves as "from Experimenter", which keeps the enrollee blinded. Row-level security (RLS), the database's per-user access rules, means it never reads the operator's profile.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
run_id	uuid	FK NN	A foreign key to `experiment_run.id`; it cascades on delete, so purging a run removes this row (see the run-purge Refs).
sender_id	uuid	FK NN	Who sent the message. The enrollee client shows any sender other than the enrollee themselves as "from Experimenter", so this identifies the author without revealing which staff member wrote it.
target_kind	message_target	NN	The kind of message target: direct is one-to-one to an enrollee, cohort targets a cohort, and everyone targets all enrollees, all of which go from an operator to enrollees; operator marks a message from an enrollee to the run staff. This column carries both the direction and the distinction between a direct message and a broadcast, so there is no separate kind column.
target_cohort_id	uuid	FK	The target cohort. It is set exactly when `target_kind` is cohort, and is otherwise null.

body	text	NN	The message text as written. It carries no merge tokens, because this is a message somebody typed rather than a template the middle tier fills in; `email_template` covers the templated mail.
created_at	timestamptz	NN	

INDEXES

(id) primary key unique

(run_id) non-unique

message_recipient TABLE

3 columns

COLUMN	TYPE	KEYS	NOTES
message_id	uuid	PK FK NN	A foreign key to `message.id`; it cascades on delete, so purging a run removes this row.
recipient_id	uuid	PK FK NN	The intended reader: an enrollee for operator-to-enrollee messages, or a staff member for an enrollee-to-operator message. It is a generic profile reference, not restricted to enrollees.
read_at	timestamptz		A per-recipient read receipt. This is why every message creates one recipient row per reader.

INDEXES

(message_id, recipient_id) primary key unique

(recipient_id) non-unique *a reader's inbox*

llm-chat

Tables in this module:

- `model_catalog` — the installation's menu of models.
- `model_config`, with `model_config_model` — the reusable model side.
- `chat_round` — both the analytics unit and the transcript.
- `chat_round_candidate` — the depth behind a round, for the see-all view and for forensics.

- `run_result` — the per-run scorecard.

The prompt and task side — `system_prompt`, scenario, `opening_message` and blinded — belongs to experiment, and the nudge is referenced by `experiment_run`.

`model_catalog` TABLE

15 columns

The install's model menu, backed by the database so the admin curates it live without a redeploy.

Only the API keys are .env secrets, and those are per provider rather than per model. The `model_config_model` table references a model's identity from here, so identity is defined once. The aptitudes and benchmarks columns are advisory, meant for humans choosing a model; expertise is the one column the system itself reads, as the text the MoE router matches prompts against.

The temperature constraint on reasoning models is handled by an app-level capability skip-list, not by a column.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
provider	text	UK NN	The language-model vendor, such as anthropic, openai, or google. The API key for each provider is a secret held in the middle tier's environment, as a fly.io secret or in the .env file. One key unlocks all of that provider's models, so adding a new model from an existing provider needs no new secret and no redeploy.
model	text	UK NN	The provider's API model-id string, for example claude-opus-4-8 or gpt-5. It is not a secret, only a reference.
display_name	text	NN	This is a friendly label shown in pickers, for example "Claude Opus 4.8".
enabled	boolean	NN	An admin toggle. A disabled model drops out of pickers but still resolves for existing configurations and history. Adding a newly released model or retiring a disappointing one is done by editing rows, with no code deploy.
input_price	numeric	NN	The price of prompt tokens, the input tokens, in US dollars per 1,000,000 tokens. Providers bill input and output at different rates, so the two are stored separately. A self-hosted or local model is set to 0, or to a near-zero figure covering compute, which is how a locally hosted language model shows up as near-free in cost reports. This price is read at round time to freeze each candidate's `token_cost`, so a later price edit never rewrites historical cost.
output_price	numeric	NN	The price of completion tokens, the output tokens, in US dollars per 1,000,000 tokens, typically several times `input_price`. It follows the same freezing rule as `input_price`.

context_window	int	NN	How many tokens this model can accept in one call, prompt and history together. It is curated alongside the prices and mandatory for the same reason: nothing else in the system knows it, and a wrong value is not recoverable from the provider's response. It is what bounds the replayed chat history. A `model_config`'s members may have very different windows, and every member has to receive the identical input or the comparison stops being between models and starts being between how much each one was told. So the usable budget is taken from the SMALLEST `context_window` among the config's members — the largest window they all share — and that same assembled history goes to every one of them. Sizing to the largest instead would leave the smallest member's provider silently truncating its input, differently from its peers and with nothing recorded. Reserve for the answer and the room taken by the system prompt, the retrieved context and the current prompt all come off this figure before history is fitted. ALGORITHMS §23 specifies the arithmetic.
aptitudes	"text[]"		Advisory selection tags describing what the model is especially good at, for example judge, coder, reasoning, vision, long-context, or general. They filter the picker, for instance to show good judge models. They are admin-curated and may go stale, and the system never makes a correctness decision from them.
expertise	text		One curated sentence saying what this model is good at, written as prose rather than as tags. An example is "Long-form reasoning and careful multi-step analysis over large documents." It is the text the MoE (mixture of experts) router matches a prompt against. The router embeds this sentence once and caches the vector, then compares it with each incoming prompt by meaning (ALGORITHMS §19). Where aptitudes is a list of short tags for filtering a picker, this is prose for matching by meaning. When expertise is null the router falls back to the aptitudes tags joined with notes. That is a weaker match, so the authoring screen warns when a model used in an MoE configuration has no sentence here. It never changes how the model is called, only which prompts reach it. A poor match sends the round down the router's tie-break path rather than producing a wrong answer.
benchmarks	jsonb		Optional published benchmark scores for reference and display, for example {"swe_bench":0.65,"gpqa":0.50,"mmlu":0.88}. They are advisory only.
notes	text		Free-text human notes about this model. They are searchable by SQL keyword and, as part of the shared semantic index that supports meaning-based search, by meaning.
created_at	timestamptz	NN	
updated_at	timestamptz	NN	

version	int	NN	A counter for optimistic concurrency control (OCC), a way to prevent lost updates. It is increased on every edit so that two people editing the same row at once cannot silently overwrite each other.

INDEXES

(id) primary key unique

(provider, model) unique

model_config TABLE

17 columns

One logical model as far as its users are concerned, with an optional "see all constituents and iterations" diagnostic.

Its members are all generators (primary or extra); there is no judge member — the experiment's one trusted judge does all scoring. Every method except moe fans out and then reduces. MoE routes to one member and calls only that one, so it is the cheapest multi-member method per round. Self-improvement uses mixture-of-agents (MoA) feedback, feeding the best or aggregate answer back to the generators, and stops on `score_target`, `min_score_gain`, or `max_iterations`. That judge scores each candidate (0 to 100, a weighted mean of the rubric's factors) to pick a winner and drive the loop, and the same scores serve within-experiment cross-run comparison — one judge, one scale, no second pass.

Each candidate is persisted as a `chat_round_candidate`, keyed by its iteration index. The configuration and its constituent `model_config_model` rows are immutable while any run references them (read-only, delete blocked by RESTRICT); to change one, clone it, which yields a new configuration under its own name. Only notes stays editable while it is referenced.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
name	text	UK NN	The name of this reusable configuration.
combine_method	combine_method	NN	How the constituent generator models become one logical output. The six methods are: 1. **single** — the configuration's one member answers and the ensemble is skipped. It is required for a one-member configuration and is not allowed for a multi-member one, a rule the application and the authoring screen enforce. The self-improvement loop still applies to it. 2. **synthesize** — every member answers, and the highest-weight member merges all the answers, so that member also acts as the aggregator. 3. **vote** — every generator scores the others on the experiment's rubric, and the highest voter-weighted tally wins. 4. **consensus** — the answer the models most agree on wins: the candidates are embedded and grouped by meaning, and the most central answer of the cluster with the greatest summed weight is returned.

			5. **`judge_best`** — every member answers, the experiment's one judge scores them, and the top answer wins. 6. **moe** — MoE, or mixture of experts, which routes rather than fanning out. A router picks one member, the expert, to answer, so a round costs one generation however many members the configuration holds. It matches the prompt against each member's capability text in `model_catalog.expertise` and escalates a tie to the experiment's judge, recording what it decided in `chat_round.moe_routing`; ALGORITHMS §19 specifies the router. It requires at least two members, and its routed answer is scored exactly as under single. Ties break at random under `judge_best`, vote and consensus. All scoring is done by the experiment's single trusted judge (experiment.score_judge_*) and never by a member of the configuration. That covers the `judge_best` selection, the official number for the vote and consensus winners, the MoE router's prediction, and the stop condition of the self-improvement loop in every method.
`max_iterations`	`int`		An optional cap on self-improvement: stop after this many looped improve-then-score passes. When null, there is a single pass. It applies to all combine methods; the score that drives the loop always comes from the experiment's one trusted judge, so the stop condition is on the same scale for every method.
`min_score_gain`	`numeric`		An optional convergence gate on a 0 to 100 scale: stop once a pass gains less aggregate score than this compared with the previous pass. When null, there is no gate. It applies to all methods. Any single stop condition, whether this one, `max_iterations`, or `score_target`, ends the loop.
`score_target`	`numeric`		An optional good-enough bar on a 0 to 100 scale, defaulting to 90: stop as soon as the combined score reaches it. It is the gate that skips refinement when the first pass is already good. When null, there is no target.
`underperform_threshold`	`numeric`		An optional threshold on a 0 to 100 scale, defaulting to 50: a generator that scores below it in an iteration is pruned, meaning dropped, from later iterations. This is the pruning step of mixture of agents (MoA), an approach that combines several models. When null, there is no pruning.
`score_weights`	`jsonb`		Optional per-factor weights for the composite score across groundedness, relevance, coherence, and instruction-following, on a 0 to 100 scale. When null, the factors are weighted equally as an arithmetic mean. These weights apply wherever factor-scoring is used, not only in `judge_best`.
`enabled`	`boolean`	NN	An owner or admin toggle, matching `model_catalog.enabled`.

			A disabled configuration drops out of pickers for new runs but still resolves for existing runs and history. This is how to retire a configuration that cannot be hard-deleted because a past run's `experiment_run.model_config_id` references it under a no-action delete rule.
notes	text		Free-text human notes about this configuration. They are searchable by SQL keyword and, as part of the shared semantic index that supports meaning-based search, by meaning.
created_by	uuid	FK	A foreign key to `profile.id` recording who first authored the configuration, kept as a record of its origin. It does not change on ownership transfer, and it is set to null when that profile is hard-deleted (see the Ref below); its email is snapshotted in `created_by_email`. A configuration is authored independently, like a cohort, by an experimenter or an admin.
created_by_email	text	NN	The author's email captured at creation time (DENORM, like `audit_log.actor_label`), preserving the origin after the profile is hard-deleted and `created_by` goes null.
updated_by_email	text		The email of whoever last edited this configuration, captured on each update (DENORM).
owner_id	uuid	FK	A foreign key to `profile.id` that is set to null when that profile is deleted (see the Ref below). It is the current owner, which can change, and it defaults to `created_by` at creation; the admin can transfer it. Under row-level security (RLS), the database's per-user access rules, the owner can create, read, update, and delete the configuration, while peers can read it and use it in their own runs. It is null only when the owner was hard-deleted, which leaves the configuration orphaned and admin-managed.
created_at	timestamptz	NN	
updated_at	timestamptz	NN	
version	int	NN	A counter for optimistic concurrency control (OCC), a way to prevent lost updates. It is increased on every edit. Its constituent `model_config_model` rows are versioned through this parent, so any edit to a child bumps this counter, and the configuration is guarded as a single unit.

INDEXES

(id) primary key unique

(name) unique

`model_config_model` TABLE

6 columns

A model's identity, its provider and model-id, lives once in `model_catalog`. The per-use settings here (temperature, prompt override, mode, and weight) are attributes of this junction row: the same catalog model can appear in another configuration with different settings, so they are not redundant duplicates. The pair (`config_id`, `catalog_id`) alone is not unique, because a temperature sweep may reuse one model at several temperatures. The uniqueness rule that distinguishes members therefore covers (`config_id`, `catalog_id`, temperature, `system_prompt`), declared with NULLS NOT DISTINCT so that two rows with no override count as duplicates.

The rule permits temperature sweeps, and same-temperature prompt variants such as a skeptic and an optimist, while blocking exactly duplicated constituent rows. There is no lead-member flag: the single method uses a one-member configuration, and the highest-weight member is the aggregator under synthesize. All members are generators; there is no judge member — the experiment's one trusted judge does all scoring, and under MoE that same judge is also the router. The capability text MoE routes on lives once on `model_catalog.expertise`.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
config_id	uuid	FK UK NN	A foreign key to the `model_config` this row belongs to, identifying the models that make up the configuration. It uses the default no-action delete rule, because a `model_config` is a reusable definition and is not removed when a run is cleared.
catalog_id	uuid	FK UK NN	Which model this is, drawn from the admin-curated `model_catalog`, so a model's identity is defined once. It uses the default no-action delete rule, so a catalog model that is in use cannot be hard-deleted; to retire it, set enabled to false.
temperature	numeric	UK NN	The temperature for this specific use of the model, that is, this model within this configuration. Note that OpenAI reasoning models (the o-series and GPT-5-class models) reject any temperature other than 1, so it is omitted for those through a model-capability skip-list.
system_prompt	text	UK	This model's own standing instruction — its role inside the configuration, for example casting one member of an ensemble as a skeptic and another as an optimist. It is the FIRST layer of the system-prompt cascade: `experiment.system_prompt` is appended after it, so the study's instruction always has the last word (ALGORITHMS §10). Null means this model carries no role of its own and receives the experiment prompt alone.
weight	numeric	NN	The relative weight (trust/importance) of this member. It weights each ballot under vote, sums to give a cluster its support under consensus, and marks the highest-weight member as the aggregator under synthesize (ties broken by a stable order, for reproducibility). Under MoE it doubles as the member's tier: a low weight reads as the cheap or fast expert and a high weight as the strong one, and the highest-weight contender is the router's tie-break. It applies to generators only. There is no lead flag: the single method uses a one-member configuration.

INDEXES

(id) primary key unique

(config_id, catalog_id, temperature, system_prompt) unique

NULLS NOT DISTINCT (PG15+) — see RAW DDL SUPPLEMENT; permits temperature sweeps + same-temp prompt variants, blocks exact-duplicate constituent rows. Its leading `config_id` also serves `config_id` -only lookups (leftmost-prefix), so no separate (`config_id`) index.

chat_round TABLE

32 columns

One prompt-to-best-response exchange, already scored.

It is both the analytics unit and the transcript. The quality scores (factor_scores plus the headline score_composite) are the experiment's one trusted judge's scores for the chosen response, applied identically to every run so its runs are comparable within the experiment. The objective columns (latency, tokens, iterations, and stop_reason) are recorded at runtime, and moe_routing records the MoE router's decision when that method is in force. It is a high-volume table.

Deep paging walks it with keyset pagination, which asks for the rows after a given (created_at, id) key instead of counting past them with OFFSET, so a deep page stays fast. Its inbound foreign key from chat_round_candidate references its simple primary key, id.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
enrollment_id	uuid	FK NN	A foreign key to run_enrollment.id; it cascades on delete, so purging a run removes this row. It identifies the enrollee whose exchange this is.
run_id	uuid	FK NN	A foreign key to experiment_run.id; it cascades on delete. It is a deliberate duplicate (DENORM), safe because the chain is frozen at insert, and it supports run-scoped analytics and access rules without the two-hop join through run_enrollment.
enrollee_id	uuid	FK	A foreign key to profile.id that is set to null when that profile is deleted. It is a deliberate duplicate (DENORM) that enables the optional per-enrollee cut, such as enrollee-weighted means. It is not the default aggregation axis.
session_seq	int	NN	Which session this round belongs to. It increments when the enrollee clears the chat and starts a new one, while a reconnect keeps the same session.
round_seq	int	NN	The order of this round within its session.

prompt	text		The effective prompt for this round — the message actually submitted to the model set-up. When the experiment's `enable_prompt_enhancer` is on, this holds the enhanced (rewritten) prompt and `raw_prompt` holds the enrollee's original; when off, this is the enrollee's message verbatim and `raw_prompt` is null.
raw_prompt	text		The enrollee's original message, preserved when prompt enhancement ran (see ALGORITHMS §18). It is null when enhancement was off, in which case prompt already is the original. Storing both means the rewrite never loses the enrollee's actual words. It is also what the judge scores against. When enhancement ran, ALGORITHMS §4 and §5 are handed this column rather than prompt, so relevance and instruction-following measure the answer against what the enrollee asked rather than against the rewrite the enhancer produced. That is what keeps enhancement accountable: a rewrite that drifted from the request scores worse, not better. A rewrite that was attempted and rejected also lands here, holding the same text as prompt; enhancement records which case applies.
enhancement	jsonb		The prompt-enhancement outcome for this round, and null when the experiment's `enable_prompt_enhancer` is off. It records five things: 1. **status** — applied when the rewrite passed every check, and fell_back when it did not 2. **applied** — the transformations the enhancer declared 3. **failed_check** — the check that rejected a failed rewrite 4. **version** — the enhancer instructions that ran 5. **tokens and `latency_ms`** — what the call spent. For example {"status":"applied","applied": ["scope","structure"],"failed_check":null,"version":"v1","tokens":412,"`latency_ms`":830}. It exists because a fallback is otherwise invisible: a rejected rewrite leaves prompt holding the enrollee's original, which is indistinguishable from enhancement having been off. Since the study argument rests on the enhancement policy applying uniformly to every enrollee, a silent fallback is exactly the event that has to leave a trace. ALGORITHMS §18 specifies the algorithm and the checks.
history_window	jsonb		What conversation history this round actually carried, and null when the experiment's `enable_chat_history` is off. It records turns_included and turns_available, the tokens the history occupied, and the usable budget it was fitted into. It also records the `context_window` that budget came from (the smallest among the configuration's members), whether truncation dropped the oldest turns, which limit bound — the platform cap or the window budget — and whether the enrollee saw the approaching-limit alert. For example {"turns_included":6,"turns_available":9,"tokens_used":14320,"usable_budget":16384,"`context_window`":32768, "bound_by":"window","truncated":true,"alert_shown":true}. It exists because a truncation that is not recorded cannot be told apart from one that never happened, which is the same reason `chat_round.enhancement` records a fallback. The stakes are higher here: `model_config` is a swept variable, so two arms of one comparison can have different smallest windows and therefore different effective memory.

			Without this column the experimenter reads that as a difference in model quality. With it, `run_result.history_stats` can say plainly that one arm was working with less. alert_shown is recorded for the same reason. An enrollee warned that their session is filling may write more tersely, wrap up, or start fresh — a behavior change that lands only on enrollees with long sessions, which is itself an outcome. Recording it turns an uncontrolled variable into a measured one. ALGORITHMS §23.
rewound_at	`timestamptz`		When this round was rewound away, and null for a round still in the session's working context. An enrollee who rewinds to an earlier point in the conversation discards everything after it: those rounds stop being replayed to the models, exactly as if they had not happened, while the earlier context is kept — which is what separates a rewind from clearing the chat and starting a new session. A rewound round is excluded from history assembly and from nothing else. It stays in the transcript, keeps its scores, and still counts in `n_rounds` and every statistic built on rounds. Dropping it from the analytics would bias a run upward, because an enrollee rewinds when an exchange went badly, and that is data. The column is a timestamp rather than a flag so the order of rewinds within a session is legible.
clarification	`jsonb`		The exchange that preceded this round when the enrollee's earlier attempts did not pass the sufficiency gate, and null when the first attempt passed. Each entry holds the message they sent, why it was turned back — an unmet relevance floor, or the pieces it left out — the reply they were shown, the tokens that reply spent, and which reply style was in force, since a cleared run may no longer have its `experiment_run` row to consult. That style also decides what reaches the models. A dialogue is rendered into `raw_prompt` as question-and-answer pairs, because an answer such as "crude oil" means nothing without the question that drew it. A rewrite has nothing to pair, so `raw_prompt` is the final request alone and the earlier drafts stay here as record only. A turned-back attempt is deliberately not a round of its own, because rounds are the unit every statistic counts and an answerless exchange would distort the round count, the cost per round and the score distribution alike. Keeping it here leaves those numbers about answered questions while the detail survives. Its tokens still count in `judge_tokens` and `judge_cost`, since the gate runs on the trusted judge. ALGORITHMS §21 specifies the gate.
retry_count	`int`	NN	How many attempts the sufficiency gate turned back before the one that produced this round, and 0 when the first attempt passed. It is the length of the clarification list, kept as its own column because it is the outcome measure of the Socratic switch and so has to be grouped and averaged without reading into jsonb — and because it outlives any trimming of that detail. Both settings of `experiment_run.socratic_enabled` increment it, which is what makes the two arms comparable. It counts attempts rather than questions, so a reply asking for two closely linked pieces at once still counts as one. ALGORITHMS §21 writes it and §6 rolls it up into `run_result.retry_stats`.
response	`text`		The single response shown to the enrollee, a copy of the chosen candidate's text.
declined	`decline_kind`		Whether the response refused the request rather than attempting it, and null when it attempted it. A value of appropriate means the refusal was warranted, by the system prompt's own bounds or by there being no source to

			answer from; a value of unwarranted means the response declined with nothing to justify it. It exists because refusing and failing look identical to the scorer otherwise. Relevance asks whether the answer addressed what the enrollee asked, so a refusal scores badly on it while scoring well on instruction-following, and a model that correctly refuses would rank below one that answers anyway. An appropriate refusal therefore takes relevance to N/A, exactly as an absent source takes groundedness to N/A, and the composite is the weighted mean of whichever factors applied. An unwarranted refusal keeps relevance scored, so a model that simply will not answer is still penalized, which is what stops the verdict becoming a way to dodge a low score. The judge sets it in the same call that produces the scores, since it already holds both the response and the system prompt. Keeping it as a column rather than inferring it from an absent relevance score lets a run report how often it refused and how often that was warranted. It is distinct from the sufficiency gate in ALGORITHMS §21, which turns a request back before any round exists; this records a refusal by the models under test. ALGORITHMS §4 specifies the verdict.
factor_scores	jsonb		The judge's per-factor scores for the chosen response on a 0 to 100 scale, keyed by the experiment's rubric factor keys — for the built-in default rubric {groundedness, relevance, coherence, instruction_following}, for a custom rubric whatever it defines (say {funniness, originality, timeliness}). It is a jsonb bag rather than fixed columns so the factor set is configurable per experiment. A factor that did not apply this round (for example groundedness with no grounding source) is absent, never zero; `experiment.score_rubric_version` records which rubric produced the set.
score_composite	numeric		The weighted mean of the round's applicable factor scores, using `experiment.score_weights` renormalized over the factors that applied, on a 0 to 100 scale. It is kept as its own column rather than inside `factor_scores` because it is the headline the comparisons rank by, the primary sort and comparison key.
inline_score	numeric		The judge's selection score that picked this round's winning candidate. With one trusted judge doing all scoring, it is on the same 0 to 100 scale as the factor scores; for the chosen candidate it equals `score_composite`.
best_model	uuid	FK	Which constituent generator produced the chosen response. It resolves to the model's catalog identity. It is null only when the round failed before any answer was chosen, in which case error carries the reason.
worst_model	uuid	FK	The lowest-scoring generator this round. Aggregated across runs, it signals a consistently poor model. It is null when the round failed, and also when only one generator answered, since a single candidate is both the best and the worst.
best_score	numeric		The judge's score of the best generator, on a 0 to 100 scale.
worst_score	numeric		The judge's score of the worst generator, on a 0 to 100 scale.
num_iterations	int	NN	The number of self-improvement passes for this round, where 0 means a single pass.

stop_reason	stop_reason		Why the loop ended: score_target, min_gain, max_iterations, or single_pass.
moe_routing	jsonb		The MoE (mixture of experts) routing decision for this round, and null under every other combine method. It records which stage decided (embedding or judge), the member that was routed to, the per-member cosine similarities, the contender set a tie considered, and any fallback that fired — for example {"stage":"judge","chosen":"<model_config_model id>","sim":{...},"contenders":[...],"fallback":null}. It is what makes the router auditable: best_model names the expert that answered, and this records why it was chosen. A run''s escalation rate is read from these rows. The judge-router''s own tokens are counted in judge_tokens and judge_cost, because the router is that same trusted judge (see ALGORITHMS §19).
latency_ms	int		The wall-clock time, in milliseconds, that the enrollee waited for this round. It is parallel-aware: when the generators run at once, it is the span of the slowest, not the sum of the calls. The per-candidate call times are on chat_round_candidate.latency_ms; summing those gives cumulative latency, which is a different figure from this parallel-aware wall-clock time (and is not compute time, since latency includes queue and network wait).
tokens	int		The total token count for the round, which is the sum of input_tokens and output_tokens across all candidate generations. The per-candidate breakdown is on chat_round_candidate.
token_cost	numeric		The US dollar cost of the round, which is the sum of chat_round_candidate.token_cost across every candidate, covering all generators and all iterations. It does not include the judge, which is judge_cost below; the complete round spend is token_cost plus judge_cost.
judge_tokens	int		The tokens consumed by the judge across its calls this round — its scoring calls, and under MoE the pre-generation routing call as well, since the router is that same judge. It is captured separately for two reasons. The judge produces no candidate row of its own, so its usage would otherwise be invisible to cost reports. Keeping it out of token_cost also leaves the generator spend a clean measure of the variable under test.
judge_cost	numeric		The US dollar cost of the judge's calls this round, frozen at round time from the judge model's catalog prices. Add it to token_cost for the full round spend.
error	jsonb		Details of a failure, whether the round hung, errored, or hit a system failure. It is null on success.
created_at	timestamptz	NN	

INDEXES

(id) primary key unique

(enrollment_id, session_seq, round_seq) non-unique *transcript order*

(run_id) non-unique *run-scoped aggregation*

(enrollee_id) non-unique

(best_model) non-unique

(worst_model) non-unique

(created_at, id) non-unique *keyset pagination over the transcript / high-volume scans*

chat_round_candidate TABLE

13 columns

The competition behind a round.

One row per generator per self-improvement pass, supporting the "see all" side-by-side view and per-model and poor-performer analysis. When `experiment_run.keep_candidates` is true the text of every candidate is kept; when it is false, only the content column is emptied as each round finalizes, and the rest of the row survives for per-model analysis. Either way the rows go with their `chat_round` when the run is cleared. This table is high-volume, growing with generators times iterations; its primary key is a simple id, its distinct-member unique is (`round_id`, `model_id`, iteration), and its foreign key to `chat_round` is a plain `round_id`.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
round_id	uuid	FK UK NN	A foreign key to `chat_round.id`, part of the uniqueness key; it cascades on delete, so a candidate is removed with its round.
model_id	uuid	FK UK NN	Which generator (primary or extra) produced this candidate. It is part of the uniqueness key.
iteration	int	UK NN	The 0-based index of the self-improvement pass. It is part of the uniqueness key.
content	text		The candidate's text, kept for every candidate, winners and losers alike, so that the see-all side-by-side view works. The chosen candidate's text is also copied to `chat_round.response`. It is nullable because it is the one part of the row that is ever cleared.

			When experiment_run.keep_candidates is false, this text is set to null as each round finalizes and the row itself stays. The candidate's score, tokens, cost and latency therefore remain available for per-model analysis.
scores	jsonb		The judge's per-factor breakdown for this candidate, keyed by the experiment's rubric factor keys (for the built-in default rubric: groundedness, relevance, coherence, instruction_following).
inline_score	numeric		The judge's aggregate score for this candidate, on a 0 to 100 scale, that drove selection and convergence.
disposition	candidate_disposition	NN	This records the candidate's final state, as a single enum. chosen means its text became chat_round.response and is the single winner across all iterations, since the loop is not monotonic. pruned means it fell below underperform_threshold and was dropped from later iterations. none means it was generated and scored but neither chosen nor pruned. Exactly one candidate is chosen per round. Under synthesize the chosen row is the merged answer rather than any member's own, and the N answers that fed the merge stay none; the round's combine_method is what tells the two cases apart.
latency_ms	int		This records the wall-clock time, in milliseconds, for this model call.
input_tokens	int		The prompt tokens this model consumed on this call, reported by the provider's API usage field or counted with the model's tokenizer for a local model. They are tracked per candidate, not per round, because tokenizers differ by model, per-model system-prompt overrides change the text, and each self-improvement pass refines the input.
output_tokens	int		The completion tokens this model produced on this call, from the same API usage field.
token_cost	numeric		The US dollar cost of this candidate, frozen at round time as input_tokens/1e6 * model.input_price + output_tokens/1e6 * model.output_price, using the model_catalog prices in effect then. There is one model per candidate, so this is a single model's cost; the round total is the sum over candidates (chat_round.token_cost). It is near-zero for a local model.
created_at	timestamptz	NN	

INDEXES

(id) primary key unique

(round_id, model_id, iteration) unique

(model_id) non-unique *per-model analysis*

(created_at, id) non-unique *keyset pagination over the high-volume candidate detail*

`run_result` TABLE

29 columns

operational statistics (12) — a 3×4 grid: {iterations, `latency_ms`, tokens} × {_mean, _median, _min, _max}. (Full explanation in the table Note.) score statistics — composite headline as two fixed columns; every dimension's robust five-number summary in `factor_stats`. All enrollee-unit (per-enrollee means), so no round-weighted / enrollee-weighted split.

The flat per-run scorecard, computed once at run end from the run's chat_rounds. Every measure below is nullable, because a run can end with nothing to compute from. An aborted run with no scored rounds leaves them unset, which reads as "not computable" and not as a zero. It is durable. It survives both a retention sweep of the raw chat_rounds and a cleanup, because it hangs off `experiment_run` rather than off the rounds. It is removed only when an admin purges the run, so it never dangles. The comparison queries read it, grouping an experiment's runs by the parameter that varies, and are valid within that experiment only, never across experiments.

For the score statistics, the composite headline is two fixed columns (`composite_mean` and `composite_median`) and every dimension — each rubric factor plus the composite — carries a nonparametric five-number summary {min, q1, median, q3, max} in `factor_stats`, all enrollee-unit (per-enrollee means). Standard deviation is not stored: the interquartile range is the distribution-free spread. The per-factor stats are a jsonb bag because the factor set is configurable per experiment. For the operational statistics, iterations, latency, and tokens each carry mean, median, min, and max, for 12 columns, plus `total_tokens` as the run's sum. The dimension and metric sets are fixed, so these are flat columns rather than a child table.

Medians, quantiles, and any drill-down need the raw rounds, because quantiles cannot be re-pooled across runs.

COLUMN	TYPE	KEYS	NOTES
`run_id`	`uuid`	PK FK	This is both the primary key and a foreign key to `experiment_run.id`; it cascades on delete. There is one scorecard per run, a one-to-one relationship. It carries no gen_random_uuid() default, because the value is not the app's to invent: it is the id of the run this scorecard is for. It survives a cleanup, which keeps the `experiment_run` row and this scorecard; the cascade fires only when an admin purges the run, so it never dangles.
`n_rounds`	`int`		The number of scored rounds aggregated, which is the sample size.
`n_enrollees`	`int`		The number of distinct enrollees who contributed rounds.
`stop_reason_dist`	`jsonb`		A tally of why the loops stopped, across `score_target`, `min_gain`, `max_iterations`, and `single_pass`.
`final_answer_stats`	`jsonb`		The score distribution over each session's LAST round rather than over every round, and null when no session contributed one — a run with no rounds, or one whose rounds were all rewound away. It carries the same five-number summary and composite mean and median that `factor_stats` and `composite_mean` hold, computed on that one round per session. It exists because the last answer is what the enrollee walked away with, and a long session can drift from where it started. Read it beside `composite_mean` rather than instead of it: the all-rounds figure says what the whole session cost and scored, and this one says where it ended up. ALGORITHMS §6 aggregates it.

retry_stats	jsonb		What the sufficiency gate cost this run in enrollee effort, and null when no gate could fire. It records rounds_with_retries, the mean and maximum `chat_round.retry_count` across the run's rounds, and a tally of why attempts were turned back — for example {"rounds_with_retries":41,"retries_mean":0.7,"retries_max":4, "reasons":{"missing_slots":38,"out_of_scope":9}}. It is the outcome measure of the Socratic switch. Answer quality between the two arms is already in `composite_mean`; what this adds is how many attempts each arm needed before an answer existed. Like the rest of `run_result` it outlives a Cleanup of the rounds it was computed from. ALGORITHMS §6 aggregates it.
history_stats	jsonb		What conversation memory did across this run, and null when the experiment's `enable_chat_history` is off. It records context_window_min (the smallest among the configuration's members, which is what bounded every round), rounds_truncated, rounds_alerted, rewinds, and the mean and maximum turns actually replayed — for example {"context_window_min":32768,"rounds_truncated":34,"rounds_alerted":12,"rewinds":5, "turns_mean":5.8,"turns_max":9}. It is the roll-up that makes a between-arm difference visible. `model_config` is a swept variable, so one arm of a comparison can have a smaller context_window_min than another and therefore systematically shallower memory. Read as a quality difference that is a wrong conclusion; read beside these figures it is a condition difference the experimenter can see and account for. Like the rest of `run_result` it outlives a Cleanup of the rounds it was computed from. ALGORITHMS §6 aggregates it, §23 produces the per-round input.
declined_dist	jsonb		A tally of how the run's rounds answered the request, across appropriate, unwarranted, and attempted — the first two counting the rounds whose `chat_round.declined` carries that verdict, and the third the rounds that attempted the request rather than refusing it. The three sum to `n_rounds`. It exists for the same reason `model_perf` does: `run_result` outlives a Cleanup of the raw rounds, so a figure that is only derivable from `chat_round` disappears exactly when the run becomes a historical record. ALGORITHMS §4 makes refusal a first-class verdict precisely so a run can report how often it refused and how often that was warranted, and that report has to survive the rounds it was computed from. It counts refusals by the models under test only. Requests the sufficiency gate turned back never became rounds at all, so they are not here and not in `n_rounds`; `chat_round.clarification` holds those, on the round they eventually led to. ALGORITHMS §6 specifies the aggregation.
model_perf	jsonb		A per-model summary, mapping each catalog_model to its wins, losses, and avg_score. It is the per-run input for identifying poor performers. Under MoE a win is a round the router sent to that model, so this column doubles as the run's routing distribution and survives a purge of the raw rounds. The finer detail of how each round was routed lives on `chat_round.moe_routing`.
total_tokens	bigint		The sum of the token count across the whole run. It is distinct from the per-round token distribution in the operational stats below.

total_generator_cost	numeric		The run's US dollar spend on generator model calls, which is the sum over its rounds of `chat_round.token_cost`. This is the cost that tracks the experimental variable, the `model_config` or nudge under test, kept separate from judge cost so a report can isolate the variable's effect. It is frozen at run end from the per-round costs, which are themselves frozen at round time.
total_judge_cost	numeric		The run's US dollar spend on the judge, which is the sum over its rounds of `chat_round.judge_cost`. It is kept separate from `total_generator_cost` because judging is scoring overhead, not the generation being measured, so a report can isolate the variable's effect; the run's total spend is `total_generator_cost` plus `total_judge_cost`. It is frozen at run end from the per-round costs.
total_latency_ms	bigint		The cumulative per-round wall-clock time, in milliseconds, summed across the run's rounds. It is the run's cumulative latency, not its compute time, and is distinct from the per-round latency distribution in the operational stats and from the run's calendar duration (`experiment_run.started_at` to `ended_at`).
computed_at	timestamptz		When the scorecard was frozen, at run end.
iterations_mean	numeric		The mean number of self-improvement passes per round.
iterations_median	numeric		The median self-improvement passes per round across the run's rounds.
iterations_min	numeric		The lowest self-improvement passes per round across the run's rounds.
iterations_max	numeric		The highest self-improvement passes per round across the run's rounds.
latency_ms_mean	numeric		The mean per-round latency, in milliseconds, measured as app wall-clock time.
latency_ms_median	numeric		The median per-round latency, in milliseconds, across the run's rounds.
latency_ms_min	numeric		The lowest per-round latency, in milliseconds, across the run's rounds.
latency_ms_max	numeric		The highest per-round latency, in milliseconds, across the run's rounds.
tokens_mean	numeric		The mean per-round token count.
tokens_median	numeric		The median per-round token count across the run's rounds.
tokens_min	numeric		The lowest per-round token count across the run's rounds.
tokens_max	numeric		The highest per-round token count across the run's rounds.

composite_mean	numeric		The enrollee-unit mean of the composite: the mean of each enrollee's mean composite, so a chatty enrollee does not dominate. A fixed column because it is a primary ranking key for cross-run comparison.
composite_median	numeric		The enrollee-unit median of the composite, the robust headline center, kept as a fixed column alongside `composite_mean` as the two keys the comparisons sort by.
factor_stats	jsonb		The robust distribution summary for every score dimension, keyed by factor key plus "composite". Each value is a nonparametric five-number summary {min, q1, median, q3, max} on a 0 to 100 scale, computed with the enrollee as the unit (over per-enrollee means). A jsonb bag rather than fixed columns so it adapts to a configurable rubric's factor set. Standard deviation is deliberately omitted: the summary assumes no distribution shape, with the interquartile range (q3 minus q1) as the spread. Drill-down beyond this needs the raw rounds, because quantiles cannot be re-pooled across runs.

INDEXES

(run_id) primary key unique

Search and embeddings (a generic bring-your-own-index sidecar)

document TABLE

16 columns

The source is a single origin field: a web URL identified by its scheme, or otherwise an uploaded-file object key.

The cached `extracted_text_path` is the frozen snapshot the embeddings are built from and the unit used for retrieval-augmented generation (RAG) retrieval (ALGORITHMS §7). It is stable and is never set to null, and `extraction_status` reports its state. A re-extraction is written to a staging location and swapped in atomically, so retrieval never drops offline: the snapshot is readable in every state except pending, and except a first extraction that failed and so has no prior snapshot.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
name	text	NN	
purpose	document_purpose	NN	The document's category, which scopes retrieval. The set is extensible, so a new category becomes a new retrieval scope with only a new enum value. There are two today:

			1. **A `context` document** — grounds the enrollee chat through retrieval-augmented generation (RAG), in which relevant text is supplied to the model. It is attached to an experiment through `experiment_context_file` and owned by the experimenter who uploaded it (`owner_id`). 2. **A `reference` document** — methodology or reference material, owned by the admin, that the Ask engine pulls by meaning. It takes the top-K nearest vector matches over the documents whose purpose is reference, with no explicit link to anything.
`source`	`text`	NN	The document's origin, held in one field so a user never fills two. It is either an http(s):// URL for a web-hosted document fetched over the Internet, or the object-storage key of an uploaded file the app holds. The object-storage key applies when there is no http(s):// prefix. The uploaded file lives in R2 by default, with the provider set in .env. The app branches on the URL scheme.
`extracted_text_path`	`text`		The stable object-storage key of the cached extracted-text snapshot, derived once from id, for example extracted/{id}.txt. The middle tier extracts text from the file (Word, Excel, or PDF, including scanned files through optical character recognition, or OCR), and the embeddings and RAG retrieval are built from it. A file that is already plain text is not converted at all: the extraction step simply copies it to this key, which fixes a stable snapshot that later edits to the source cannot move. It is never nulled or repointed: a re-extraction writes to a staging key and atomically overwrites this one on success, so the live snapshot never goes offline. The `extraction_status` column is the truth about what this file currently holds. The key is kept human-locatable for diagnostics. While an active run references the document, the snapshot swap is blocked, which freezes the document to preserve run fidelity.
`extraction_status`	`extraction_status`	NN	The truth about the `extracted_text_path` file, and about whether there is anything to read at all. This column exists for concurrency: the snapshot is readable in every state except pending and a first-extraction failed that has no prior snapshot. A re-extraction writes to a staging key and is swapped in atomically (ALGORITHMS §16), so retrieval keeps serving the existing text for the whole time a new extraction is running, and keeps serving it even if that extraction fails. There are four values. pending means the file was never extracted, with nothing to read yet. ready means it holds the current, complete extraction. stale means it holds a complete but outdated snapshot, because the source changed and a re-extraction is queued or running, while retrieval keeps using this text until the swap. failed means the last attempt errored; if a previous snapshot exists it stays readable and is still served, but a first-extraction failure has nothing to serve. So the unreadable states are pending and a never-yet-extracted failed; ready, stale, and a failed-after-success all serve text. The transitions are as follows. A first extraction moves pending to ready, or to failed. Replacing the source moves ready to stale. A successful swap moves stale back to ready. An error during re-extraction moves stale to failed, and the old text stays live.

			When the admin fixes the source and a retry succeeds, failed moves to ready, or to stale if the source changed in the meantime. stale never returns to pending, because pending means empty and a document that once extracted is never empty again.
mime	text		The file's content type (MIME type), for example application/pdf, text/plain, or one of the Office formats. It guides how the middle tier extracts text at ingest (choosing the PDF, Office, or plain-text path, and whether optical character recognition is needed for a scanned file).
uploaded_by	uuid	FK	A foreign key to `profile.id` recording who first uploaded the document, kept as a record of its origin. It does not change on ownership transfer, and it is set to null when that profile is hard-deleted (see the Ref below); its email is snapshotted in `uploaded_by_email`.
uploaded_by_email	text	NN	The uploader's email captured at upload time (DENORM, like `audit_log.actor_label`), preserving the origin after the profile is hard-deleted and `uploaded_by` goes null.
updated_by_email	text		The email of whoever last edited this document's metadata, captured on each update (DENORM).
owner_id	uuid	FK	A foreign key to `profile.id` that is set to null when that profile is deleted (see the Ref below). It is the current owner, which can change. For a context document, it defaults to `uploaded_by`; the owning experimenter can create, read, update, and delete it (re-upload, decommission, or retire), while peers read and reference it as run context, and the admin can transfer it. A reference document is admin-managed, so the admin has full create, read, update, and delete rights, experimenters read and use it, it is not transferred among experimenters, and its `owner_id` stays the admin. It is null only when the owner was hard-deleted, which leaves the document orphaned and admin-managed.
notes	text		Free-text human notes about this document. They are searchable by SQL keyword and, as part of the shared semantic index that supports meaning-based search, by meaning.
created_at	timestamptz	NN	
updated_at	timestamptz	NN	This drives embedding staleness detection.
retired_at	timestamptz		A soft-retire timestamp that hides the document rather than deleting it. When non-null, it removes the document from active experiment composition, meaning the pickers, while keeping it for the historical runs that froze it as context. This is distinct from deletion: decommissioning is a hard delete, allowed only when no experiment references the document, and `experiment_context_file.document_id` uses a RESTRICT delete rule, so a referenced document cannot be deleted anyway. The `retired_at` column exists precisely for the still-referenced document that should leave the active pool: it cannot be deleted, so it is hidden.

version	int	NN	A counter for optimistic concurrency control (OCC), a way to prevent lost updates, covering the editable metadata (name, source, owner, and `retired_at`). It is increased on every edit so that two people editing at once cannot silently overwrite each other. The extracted-text snapshot has its own concurrency control through the `extraction_status` staging-then-swap process.

INDEXES

(id) primary key unique

embedding TABLE

11 columns

There is a single HNSW index, the vector-index type used for similarity search, created with CREATE INDEX ON embedding USING hnsw (vector vector_cosine_ops).

A second index, a GIN on `filter_values` (CREATE INDEX ON embedding USING gin (`filter_values` jsonb_path_ops)), serves the scoped-search containment filter (`filter_values` @> ...). The pair (`source_schema`, `source_table`) is a composite foreign key to `embedding_ingest_registry`. For local sources, a trigger that fires after an insert or an update enqueues a pending `embedding_job`. The one-open-job partial unique index keeps duplicates out of the queue, and the reconciler worker skips a job whose templated text has not changed.

A trigger that fires after a delete clears the matching rows. The virtual source pointer, which has no foreign key, is what lets any source be embedded, whether a second Postgres schema or a remote, external table.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
source_schema	text	FK UK NN	A logical namespace naming where the source row lives. For ChatMaestro's own tables it is this install's app schema (current_schema(), chat_maestro by default), since every ChatMaestro table lives in that one schema; for an external source it is that source's namespace, such as remote_crm. It is part of the polymorphic source key and of the composite foreign key to the registry.
source_table	text	FK UK NN	Which source table this row comes from. Together with `source_schema`, it forms the composite foreign key to `embedding_ingest_registry`.
source_id	uuid	UK NN	The primary-key value of the source row. It is polymorphic, so there is no foreign key; together with `source_schema` and `source_table`, it pinpoints the exact source row. A document's own chunks are found this way: `source_table` holds document and `source_id` holds that document's id, with `chunk_index` telling one chunk of it from another.

chunk_index	int	UK NN	The 0-based chunk number within the source row. This is what lets one source row map to many embedding rows.
chunk_text	text		A stored copy of the exact slice of source text this vector was built from. It lets the app show a search result and rebuild the vector later without fetching the original document again. The copy is intentional, the schema marks such a deliberate duplicate DENORM, and it is safe because the stored slice never changes.
vector	"vector(1024)"	NN	The embedding vector, with one fixed dimension for the whole install. Larger models shorten their output to 1024 dimensions using Matryoshka representation learning (MRL), a way to shorten an embedding vector, and renormalize it. The distance metric, cosine, is fixed on the HNSW index (the vector-index type used for similarity search) and is not stored.
embedding_model	text	NN	Which model produced this vector, given as a provider-qualified id such as openai/text-embedding-3-large. Vectors from different embedding models are not comparable, and every vector shares one index, so each row has to say what produced it. It earns its place in two ways. It is the reconciler's work-list and its resume marker (ALGORITHMS §9): a model switch is worked through row by row, and the rows still to convert are exactly those whose model differs from the active one. It is also provenance in a subsystem meant to be lifted out and reused, where one installation-wide setting cannot be assumed.
chunking_used	text	NN	The chunking policy that produced this chunk, recorded for the same reason as `embedding_model` beside it: the policy is a setting, and this is what actually happened. Changing the policy invalidates every chunk cut under the old one, because a query then compares text split one way against text split another. Without this column that change is silent, since nothing else records how an existing chunk was made. With it, the rows still to convert are exactly those whose policy differs from the one now in force, so a policy change reuses the model-change machinery in ALGORITHMS §9 rather than needing its own. A rebuild is not atomic, so the index legitimately holds both policies while it runs, and this is what keeps that state legible. The active model itself is an installation setting in .env, not a column, and a source that is mid-rebuild is fenced off by its `embedding_job` rather than by reading this column. Provider flexibility is constrained by the fixed vector(1024): a model must produce 1024 dimensions, or be truncatable to 1024 using Matryoshka representation learning (MRL), a way to shorten an embedding vector.
filter_values	jsonb		Copied source columns used to scope or filter similarity search, for example `run_id` and `cohort_id`. Which columns are copied is set by registry.filter_columns.
created_at	timestamptz	NN	When the row was embedded. It is compared to the source row's `updated_at` to detect staleness.

INDEXES

(id) primary key unique

(source_schema, source_table, source_id, chunk_index) unique

embedding_job TABLE

10 columns

A transient work queue, not a history log.

When at rest, with all embeddings fresh, it is nearly empty; it holds only outstanding and in-flight work, bounded by the number of source rows, because completed rows are pruned and failed rows are kept until resolved. Jobs are enqueued by per-source after-insert or after-update triggers for local sources, or by the app, a connector, or change-data-capture (CDC), which streams row changes, for remote sources.

The state of a source row follows from its jobs: in-progress means a running job, queued or stale means an open pending job, and up-to-date means no open job. A constraint allows at most one open job per (source_schema, source_table, source_id), enforced by a partial unique index covering rows where status is pending or running.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
source_schema	text	FK NN	Together with source_table, this forms the composite foreign key to embedding_ingest_registry.
source_table	text	FK NN	The table the row to embed belongs to, completing the composite foreign key to the registry with source_schema.
source_id	uuid	NN	The identifier of the one source row this job covers. With source_schema and source_table it names the row exactly, in the same polymorphic style embedding.source_id uses, and the three together are what the one-open-job-per-row constraint is built on.
status	embedding_job_status	NN	Where the job stands: pending until a worker picks it up, running while it is in flight, and then removed on success. A failed job is kept rather than pruned, so a source row that cannot be embedded stays visible instead of quietly dropping out of the index. The index on (status, requested_at) is what lets the reconciler take the oldest pending job first.
attempts	int	NN	The retry count, which drives backoff and the eventual decision to give up.
error	text		This holds the detail of the last failure.

requested_at	timestamptz	NN	When the job was enqueued, which is the order the reconciler works in. It is set once and not moved by a retry, so a repeatedly failing job keeps its place in the queue rather than drifting to the back.
started_at	timestamptz		When a worker last picked the job up, and null while it is still pending. A retry overwrites it, so it reads as the start of the current attempt rather than of the first; attempts counts the tries.
finished_at	timestamptz		When the job stopped, whether it succeeded or gave up. A successful job is pruned soon afterwards, so a row carrying this in a table at rest is a failed one still waiting to be resolved.

INDEXES

(id) primary key unique

(status, requested_at) non-unique *reconciler polls oldest pending first*

embedding_ingest_registry TABLE

9 columns

The natural key is (source_schema, source_table). This is a tiny deploy-time configuration table referenced by name, so it has no surrogate id. There is one row per registered source.

The schema script seeds it, and an install reset preserves it the way it preserves the owner's profile row: it is how the install was set up rather than something the install produced, and nothing at runtime writes it back. Clearing it would leave the enqueue trigger correctly declining to enqueue for every source, so the semantic index would quietly stop filling with no error to show for it. ALGORITHMS §27.

COLUMN	TYPE	KEYS	NOTES
source_schema	text	PK NN	This is a logical namespace: the app schema (chat_maestro by default) for ChatMaestro's own tables, or an external source's namespace such as remote_crm.
source_table	text	PK NN	This names the source table to index.
text_template	text		A deploy-time setting for how to render each source row into embeddable text, for example {{name}}\n{{extracted_text}}.

chunking_policy	text		How to split this source's text into chunks: none, or fixed(size,overlap). When it is null the default for the kind of source applies, which is fixed(1000,150) for a document and none for a registered table row. The two differ because a document is long-form prose that has to be cut, while a table row is already a sentence or two once text_template has rendered it, so cutting it would separate a row from itself. Setting this column overrides the default for one source, which is an advanced adjustment rather than something an ordinary install touches. The split backs off to a paragraph or sentence boundary rather than cutting at an exact character count; ALGORITHMS §7 specifies it.
filter_columns	jsonb		Which source columns to copy into embedding.filter_values for scoped search. For the document source, this includes purpose, so embedding.filter_values carries the category and a semantic pull scopes by it. Enrollee RAG restricts to the experiment's context documents through experiment_context_file, while the Ask engine pulls chunks whose purpose is reference. Being searchable is not the same as being embedded: every text column stays searchable on its own through SQL and full-text search, so embedding is additive. Every notes or free-text column is embedded into the shared semantic index, except the privacy carve-outs profile.notes and audit_log.note (see the semantic-index developer note). The registered sources, namely document, experiment, experiment_run, cohort, nudge, model_config, model_catalog, email_template, and nl_query, along with their text_template and filter_columns, are enumerated in the search-module narrative (schema-search). The chat transcript (chat_round and chat_round_candidate) and the out-of-band message stay out, held back by scale and blinding and reached through the natural-language-to-SQL side. The nl_query table embeds its notes here like any other table, and separately keeps its own query_vector on an index of its own for matching typed questions.
is_active	boolean	NN	Enables or disables embedding this source without deleting the configuration.
notes	text		Free-text human notes about this registered source and its embedding configuration. They are searchable through SQL keyword search; the sidecar's own configuration is not itself embedded.
created_at	timestamptz	NN	
updated_at	timestamptz	NN	Configuration edits bump this timestamp.

INDEXES

(source_schema, source_table) primary key unique

The composite foreign keys into the registry. DBML cannot declare a reference spanning several columns on the column itself, so these are written as standalone Ref lines.

`embedding.(source_schema, source_table)` → `embedding_ingest_registry.(source_schema, source_table)`

`embedding_job.(source_schema, source_table)` → `embedding_ingest_registry.(source_schema, source_table)`

The Ask engine (natural-language queries)

nl_query TABLE

22 columns

A saved natural-language question and the read-only queries, if any, it stands for. Answering a question yields a short narrative plus zero or more result tables, each shown as a table or a chart; a question that needs no database query is answered by the narrative alone. The narrative may be augmented by a semantic pull of document chunks whose purpose is reference. The pull takes the top-K nearest vector matches across all reference documents, scoped through `embedding.filter_values` with no explicit link, so the Ask engine can ground an answer in methodology material. Enrollee RAG scopes to the experiment's context documents through `experiment_context_file`.

A newly typed question is matched by meaning, using `query_vector`, against the saved questions the asker may see; a match reuses the saved statements, filling in any values the question supplied. A question with no match is answered once and is kept only if the user chooses to save it. A saved question's label is unique per owner among labeled rows, retired ones included so that a restore can never collide, enforced by a partial unique index on (`owner_id`, label). Identical labels can still occur across different owners; they are disambiguated in the interface by origin and by the full-question `canonical_prompt` subtitle.

Row-level security scopes the table by role. An experimenter reads their own drafts, drafts a teammate has shared, and the approved and system questions that are not retired, and writes only their own drafts. An administrator reads and writes every row, retired ones included. The Saved-questions screen is the same screen for both roles, showing whatever those rules let through.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
label	text		A short display name for the button or saved-question list entry, for example "Model results", chosen by the user when they save the question; a built-in system question ships with one. It is deliberately short for the interface, distinct from `canonical_prompt`, which is the full question and may be long. If left empty, the interface falls back to a shortened `canonical_prompt`. It is unique per owner among labeled questions: a partial unique index on (`owner_id`, label) stops one owner from saving two questions under the same name (see the RAW DDL SUPPLEMENT), and it does not constrain across owners. In a combined list that spans owners, two identical labels from different owners are told apart in the interface by their origin — grouped and badged as the viewer's own, shared by a named teammate, approved, or built-in — and by the `canonical_prompt` shown as the full-question subtitle, which differs even when the short labels match.
canonical_prompt	text	NN	The saved question in natural language. It is the form of the question that every other way of asking the same thing maps to, which is what makes one saved question answer a whole family of phrasings.

			It may carry named parameters for the values its author chose to leave open, written in double braces, as in "top {{n}} model configurations for experiment {{experiment}}". Those values come from the agent that answered the question, never from re-reading the text. The same saved question then serves every value of n and every experiment. A value the author chose to keep fixed is written literally instead, as in "the top 5 model configurations", and never reaches params. ALGORITHMS §20 specifies how it is produced. It is used three ways. It is the text embedded into `query_vector`, so a newly typed question can be matched against it. It is the text the language model re-reads when `derived_sqls` has to be regenerated. It is also the full-question subtitle in the interface. It is not the button caption, which is the short name in label.
query_vector	`"vector(1024)"`	NN	The embedding of `canonical_prompt`, written at save time. It has its own HNSW index, the vector-index type used for similarity search, on this table, separate from the shared content index, so a newly typed question is matched against the saved questions in the same request (ALGORITHMS §8).
derived_sqls	`jsonb`	NN	The SQL SELECT statements that answer this question, whether zero, one, or several, kept together as an ordered list of {name, sql, display} entries. They are harvested from the statements the Ask agent actually ran, at the moment the asker saves the question (ALGORITHMS §8). Zero statements means a natural-language-only question, answered by the narrative alone with no database query. One question can produce several labeled result tables. The display field hints how each result is shown, as a plain table or as a chart such as bar, line, or pie. The statements are parameterized, using named placeholders such as %(model)s; the actual values are supplied at run time, so one saved question serves every value of its parameters. They are stored together in one field rather than split into a child table, because the statements of one question are always used as a set, never individually.
params	`jsonb`	NN	The question's parameters, one entry each, as {name, type, required, default, label} — where the inner label is the caption of the form field, unrelated to the row's button label above. Each entry declares the kind of value the parameter takes, so a value supplied for it can be checked before the query runs. One definition is used three ways: it validates the values supplied at run time, it builds the small form on which the user fills those values in, and it tells the language model what to fill in. Parameterizing is a save-time choice: when the user saves a question, the app offers the values it detected, and the user picks which become parameters. A value left un-parameterized is frozen as a constant inside `derived_sqls` and does not appear here, so "the top 5 models" can keep 5 fixed with one click and no form, while "results for model %(model)s" prompts for the model. Parameters apply whether or not the question runs a database query, since a natural-language-only question can be parameterized too, so one saved question serves every value. A question with no parameters has an empty list.
status	`query_status`	NN	Who may use this saved question, and how trusted it is.

			An approved question becomes both a one-click button and an exemplar the Ask agent can draw on (ALGORITHMS §8). The level is independent of how the question is answered: a question at any level may be natural-language-only, giving a narrative answer, or may produce one or several result tables. Matching offers the approved and system questions to everyone, and offers each user their own drafts. The three values are: 1. **A value of `draft`** — a user saved it. It is private to its owner, who may edit it, unless it is shared read-only. 2. **A value of `approved`** — an administrator has vetted it. Everyone in scope may read and run it, and only an administrator may edit it. 3. **A value of `system`** — it shipped with the installation as seeded data. It carries the same access as approved, so administrators read and write it and experimenters read it. An administrator moves a question between draft and approved in either direction: approving promotes a draft, and unapproving returns an approved question to its owner as a draft, which takes the button off everyone else's Ask surfaces at their next load. Each move writes an `audit_log` row. Withdrawing an approved question without demoting it is retirement, recorded in `retired_at`, not a status value.
`schema_fingerprint`	`text`		A fingerprint of the database structure the `derived_sqls` were written against. If the structure has since changed, the saved statements may not fit, so they are regenerated from `canonical_prompt` before use. This is the only stamp that forces a regeneration. It covers only the tables and columns the saved statements actually read, not the whole schema, so a migration elsewhere leaves this question alone. It takes in each object's name, its columns' types and nullability, the values of an enum-typed column, and the table's constraints, counting a unique index as the constraint it is. It deliberately ignores plain indexes, defaults, comments and row-level-security policies, since none of them changes what a query returns. ALGORITHMS §8 specifies the computation.
`model_version`	`text`		Which language model wrote `derived_sqls`, meaning the install's ASK_MODEL at the time (ALGORITHMS §8). It is a record of origin, useful when a saved question starts returning something odd and the question is which model produced it. It does not decide when a saved question is regenerated: `schema_fingerprint` does that, because a moved database structure is what actually breaks a stored statement, while an upgraded compiler leaves working SQL working.
`scope_key`	`text`		A tag for the kind of screen a saved question belongs to, such as `results`, `experiments`, or `cohorts` (ALGORITHMS §8). It controls where the question is offered and which starting values a screen fills in. It grants no access of its own: whenever the question runs, the results are limited to what the person running it is already allowed to see.
`created_by`	`uuid`	FK	A foreign key to `profile.id` that is set to null when that profile is deleted (see the Ref below). It records the author who first composed the question, kept as a record of origin, and it does not change when ownership is transferred; its email is snapshotted in `created_by_email`. It is null for built-in system queries.

created_by_email	text		The author's email captured at creation (DENORM, like `audit_log.actor_label`), preserving the origin after the profile is deleted. It is null for built-in system queries.
updated_by_email	text		The email of whoever last edited this saved query, captured on each update (DENORM).
owner_id	uuid	FK	A foreign key to profile that is set to null when that profile is deleted. It is the current owner, which can change and defaults to `created_by` at save; the admin can transfer it. It determines who may edit the draft, and it is null for built-in system queries. It is distinct from `created_by`, which records the original author.
is_shared	boolean	NN	The owner's read-sharing switch on a draft: true lets teammates see and run it read-only, while the owner keeps sole edit rights. approved and system queries are readable by everyone in scope regardless.
hit_count	int	NN	How many times this saved query has been reused. It surfaces frequently used queries and flags popular drafts as candidates for an administrator to approve.
last_used_at	timestamptz		When the query was last run. It powers a "recently used" list.
version	int	NN	A counter for optimistic concurrency control (OCC), a way to prevent lost updates. It is increased on every edit so that two people editing the same row at once cannot silently overwrite each other.
notes	text		Free-text human notes about this saved query, edited on the Saved-questions screen by whoever may edit the row: the owner of a draft, an administrator for anything. An administrator's approval note is appended here, stamped with who wrote it and when, so the reason a question was approved travels with the question; the `audit_log` row for the approval carries the same text. They are searchable by SQL keyword and, as part of the shared semantic index that supports meaning-based search, by meaning, like any notes column. They are separate from `query_vector`, which embeds only `canonical_prompt` for question-matching.
retired_at	timestamptz		A soft-retire timestamp that withdraws an approved question without deleting it. When non-null, the question disappears from every Ask surface and from the match set a typed question is compared against, while the row, its SQL, its `hit_count` and its notes stay in place. Only an administrator sets it, and only on an approved question (the CHECK in the RAW DDL SUPPLEMENT forbids it on a draft or a system row); the same administrator clears it to restore the question, so retirement is the undoable way to withdraw a shared button. Deletion is the separate, permanent way: a draft may be deleted by its owner or an administrator, and an approved question, retired or not, by an administrator. A system question is never retired or deleted.
created_at	timestamptz	NN	

updated_at	timestamptz	NN	

INDEXES

(id) primary key unique

Issue tracker (bug reports and product feedback)

issue TABLE

17 columns

The Issue Tracker's one backing table, covering both bug reports and product feedback. It is a generic, portable oversight subsystem, clamped onto the app in the same spirit as the Ask engine.

There is one row per submission, and kind selects between a bug report and a piece of feedback. The only column that belongs to one kind alone is severity, and a CHECK constraint holds it to the bug kind. That constraint lives in the raw DDL supplement, because DBML syntax cannot express it.

Feedback carries no category. An earlier design tagged it like, dislike or idea, which forced a second nullable column and a two-branch CHECK for a value nothing could act on: a like recorded no reference to what was liked, so it was a mood with no subject. Rating an answer is a real measurement and belongs to the deferred human-scoring work, keyed to `chat_round`, not to an app-feedback surface.

An admin triages every issue. An experimenter reads the issues filed by the enrollees of their own runs, which exposes nothing new, because the live monitor already shows those enrollees' transcripts to that same experimenter. A reporter does not read their own issues back, and enrollees have no surface for it.

Operations on issue are not tracked in `audit_log`, as issue tracking is a separate subsystem.

COLUMN	TYPE	KEYS	NOTES
id	uuid	PK	
kind	issue_kind	NN	Whether this row is a bug report or a piece of product feedback. One table backs both the "Report a bug" and the "Send feedback" surfaces, because the two share nearly all of their shape. Only severity is kind-specific, and a CHECK constraint ties the two together so a row can never be malformed for its kind.
summary	text	NN	The reporter's free-text description — what went wrong (bug) or what they think (feedback). Searchable by SQL keyword and, through the shared semantic index, by meaning.
severity	issue_severity		How bad the defect is. Set on bug rows and null on feedback rows, which is the whole of the per-kind CHECK.

screenshot	bytea		The page snapshot (PNG) captured when the reporter opened the bug dialog, so triage sees the screen as it was. Bug rows only, and null on feedback. It is taken from the page itself before the dialog mounts, which keeps the dialog out of its own picture, and it renders at a reduced scale so the encoded image stays under the size the request accepts. A browser that cannot produce an image this way is detected first and the step is skipped. When the capture fails anyway, which a page holding a cross-origin frame can cause, the report submits without it and says so rather than blocking the reporter. There is no retry, because these failures are structural rather than intermittent and a second attempt on the same page fails the same way. The bytes are stored inline rather than in object storage. These images are small and written rarely, and inline bytes cannot leave a row pointing at an object that never uploaded.
attachment	bytea		An optional file the reporter chose to attach (a log, an extra screenshot, a zip). Either kind may carry one. Persisted so an admin can re-download it from the triage detail later, and stored inline for the same reason screenshot is.
attachment_filename	text		This keeps the attachment's original filename, so the app can display it and offer it for download again.
attachment_mime	text		The MIME type of attachment, so a re-download serves it correctly.
context	jsonb	NN	Ambient telemetry captured with the report: recent route history, failed requests, console errors, viewport, user agent, app version/build, and current URL. It arms triage with reproduction context without the reporter having to describe their environment.
run_id	uuid	FK	The run the reporter was looking at when they filed, and null when they were not on a run screen. It is the one piece of reproduction context that is specific to this platform: context records the URL, but a typed reference is what lets triage open the run, read its transcripts, and see its configuration. It is nulled if the run is purged (see the Ref below), which keeps the issue readable after the run it described is gone. It carries no delete protection in the other direction: an open issue never blocks a purge, because oversight data must not hold operational data hostage.
status	issue_status	NN	The triage lifecycle: new (nobody has looked), triaged (an admin has dispositioned it), closed (resolved or dismissed). Drives the default triage-board filter.
reporter_id	uuid	FK	A foreign key to `profile.id` for who filed the issue, set to null if that profile is hard-deleted (see the Ref below); the reporter's email is snapshotted in `reporter_email` so attribution survives.
reporter_email	text	NN	The reporter's email captured at submit time (DENORM, like `audit_log.actor_label`), preserving attribution after the profile is hard-deleted and `reporter_id` goes null.

triaged_by	uuid	FK	A foreign key to `profile.id` for the admin who dispositioned the issue, set to null if that profile is hard-deleted. Null while the issue is still new.
triaged_at	timestamptz		When the issue was dispositioned. Null while new.
triage_note	text		The admin's free-text disposition note added during triage.
created_at	timestamptz	NN	

INDEXES

(id) primary key unique

`issue.reporter_id` → `profile.id` delete: set null

`issue.triaged_by` → `profile.id` delete: set null

`issue.run_id` → `experiment_run.id` delete: set null

Foreign keys and delete rules

A foreign key whose delete rule DBML cannot write on the column itself is declared here instead, as a standalone Ref line. These are integrity constraints like any other: each one names the child column, the parent it points at, and what happens to the child when the parent row is deleted.

Ownership and Authorship

The current-owner columns, and the composer columns `created_by` and `uploaded_by`. Deleting a profile sets these to null; the composer's email address survives in the matching `created_by_email` or `uploaded_by_email` snapshot, so provenance is never lost.

`nl_query.owner_id` → `profile.id` delete: set null

`nl_query.created_by` → `profile.id` delete: set null

`experiment.owner_id` → `profile.id` delete: set null

`model_config.owner_id` → `profile.id` delete: set null

`document.owner_id` → `profile.id` delete: set null

`cohort.created_by` → `profile.id` delete: set null

`experiment.created_by` → `profile.id` delete: set null

`nudge.created_by` → `profile.id` delete: set null

`model_config.created_by` → `profile.id` delete: set null

`email_template.created_by` → `profile.id` delete: set null

`document.uploaded_by` → `profile.id` delete: set null

> **Clearing a Run**
>
> A run can be deleted only once it is done or aborted; see the Note on `experiment_run`. Deleting an `experiment_run` cascades: it erases everything the run produced, and nothing that is a reusable definition. The standalone Ref lines below pin the delete actions that the inline column notes refer to. `experiment_run` references its three swept-variable definitions, and its container, under a RESTRICT rule, so none of them can be hard-deleted while a run references it. This is the immutable-while-referenced rule. `experiment_context_file`, which holds the experiment's context documents, is erased along with the experiment, while the shared document itself is protected by RESTRICT.

`experiment_run.experiment_id` → `experiment.id` delete: restrict

`experiment_run.nudge_id` → `nudge.id` delete: restrict

`experiment_run.cohort_id` → `cohort.id` delete: restrict

`experiment_run.model_config_id` → `model_config.id` delete: restrict

`experiment_context_file.experiment_id` → `experiment.id` delete: cascade

`experiment_context_file.document_id` → `document.id` delete: restrict

`experiment_run.launched_by` → `profile.id` delete: set null

`run_enrollment.run_id` → `experiment_run.id` delete: cascade

`chat_round.enrollment_id` → `run_enrollment.id` delete: cascade

`chat_round.run_id` → `experiment_run.id` delete: cascade

`chat_round.enrollee_id` → `profile.id` delete: set null

`chat_round_candidate.round_id` → `chat_round.id` delete: cascade

`run_result.run_id` → `experiment_run.id` delete: cascade

`message.run_id` → `experiment_run.id` delete: cascade

`message_recipient.message_id` → `message.id` delete: cascade

=

Raw DDL Supplement

```
Constraints and indexes that DBML syntax cannot express (CHECK, partial / expression / vector / GIN indexes,
extensions). A DDL generator emits the declarative body above and then these statements
VERBATIM; body + supplement = complete, accurate DDL. Standard DBML tooling ignores this block (it is
comments); a generator reads the SQL between the BEGIN and END markers.

--- Prerequisite: pgvector (no SQL here; see the note) ---
The extension is enabled ONCE per database from the Supabase dashboard (Database > Extensions >
"vector"), which installs it into the shared "extensions" schema that the generated script's
search_path already reaches. Nothing creates it from here: extension creation is database-scoped
state and needs rights the migration role should not carry, and an extension installed into the app
schema would be destroyed by DROP SCHEMA <app> CASCADE, taking every sibling schema's vector
columns with it.

The generated script asserts the type resolves before it creates anything, so a forgotten dashboard
step reads as one sentence instead of "type vector does not exist" thrown at a CREATE TABLE three
hundred lines below. That assertion is script scaffolding, emitted by _ddl_gen.py beside the
CREATE SCHEMA and SET search_path lines, which is why it is not written out here.

--- CHECK Constraints ---
ALTER TABLE message ADD CONSTRAINT message_cohort_target_ck
CHECK ((target_kind = 'cohort') = (target_cohort_id IS NOT NULL));   -- cohort target set iff kind = cohort
ALTER TABLE issue ADD CONSTRAINT issue_kind_shape_ck
CHECK ((kind = 'bug') = (severity IS NOT NULL));   -- severity set iff kind = bug
ALTER TABLE nl_query ADD CONSTRAINT nl_query_retired_ck
CHECK (retired_at IS NULL OR status = 'approved');   -- only an approved question can be retired

--- Partial UNIQUE Indexes (Enforcement Rules DBML Cannot Express) ---
CREATE UNIQUE INDEX one_admin                     ON profile          (role)  WHERE role = 'admin';
CREATE UNIQUE INDEX email_template_one_default    ON email_template   (kind)  WHERE is_default;
CREATE UNIQUE INDEX nl_query_owner_label          ON nl_query         (owner_id, label)
WHERE label IS NOT NULL;   -- one label per owner among the labeled saved questions
CREATE UNIQUE INDEX embedding_job_one_open        ON embedding_job    (source_schema, source_table, source_id)
WHERE status IN ('pending','running');

--- Case-Insensitive UNIQUE (Expression Indexes DBML Cannot Express) ---
The plain UNIQUE declared on each column keeps its identity as a key visible in the ERDs and lets
two rows differ only in capitalization; these indexes are what actually forbid that. It matters
```

```
most for email, because the owner is resolved by matching OWNER_EMAIL against profile.email on
every boot, so a capitalization difference would silently fail to find the installation's owner.
CREATE UNIQUE INDEX profile_email_lower_key    ON profile (lower(email));
CREATE UNIQUE INDEX profile_username_lower_key ON profile (lower(username));

--- Index Modifier DBML Cannot Express (Applied to an Index Declared in Its Table) ---
The model_config_model distinct-member unique index (declared in that table's `indexes` block) is generated
WITH the modifier NULLS NOT DISTINCT (PG15+) — flagged in that index's comment — so two rows with a null
system_prompt at the same (config_id, catalog_id, temperature) are treated as duplicates and blocked.
The DDL generator appends that clause to the in-table unique; it is not a separate statement here.

--- triggers + trigger functions (the ONLY triggers by design; the DDL generator's dollar-quote-aware
extractor emits them VERBATIM into schema.sql after all tables, foreign keys, and indexes). Auditing is
deliberately app-driven, not trigger-driven, so the actor, before/after, and intent are captured. ---

(1) Embedding lifecycle — keep embedding_job / embedding in sync with each LOCAL registered source table.
An insert or update enqueues a pending job (deduped by the one-open-job partial unique; the worker skips
if already fresh); a delete clears the source row's vectors and any open job. TG_TABLE_SCHEMA makes the
recorded source_schema self-adjust to whatever app schema the trigger fires in.
-- Renders a row through its registry text_template, which is the same text the worker embeds, so
-- "did the embedded text change" is answered by the template rather than by guessing at columns.
-- It takes jsonb rather than a polymorphic row: the caller converts with to_jsonb(NEW), which
-- keeps the parameter a concrete type instead of resolving anyelement against a trigger RECORD.
-- Defined BEFORE its caller. PL/pgSQL's validator checks syntax and not identifier resolution, so
-- a forward reference would compile and only fail at runtime; ordering it this way means the
-- question never has to be asked.
CREATE OR REPLACE FUNCTION embedding_source_text(sch text, tbl text, row_json jsonb)
RETURNS text LANGUAGE plpgsql STABLE AS $$
DECLARE tmpl text; rendered text;
BEGIN
SELECT text_template INTO tmpl FROM embedding_ingest_registry
WHERE source_schema = sch AND source_table = tbl;
IF tmpl IS NULL THEN RETURN NULL; END IF;
-- Splitting on the braces leaves the placeholder names at the EVEN ordinalities.
SELECT string_agg(coalesce(row_json ->> k, ''), E'\n' ORDER BY ord)
INTO rendered
FROM unnest(regexp_split_to_array(tmpl, '\{\{|\}\}')) WITH ORDINALITY AS x(k, ord)
WHERE ord % 2 = 0;
RETURN rendered;
END $$;
CREATE OR REPLACE FUNCTION embedding_enqueue() RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN
-- Enqueue only for a source the admin has registered AND left active. Without this guard the
-- INSERT below violates embedding_job's foreign key to embedding_ingest_registry, so on a fresh
-- install -- where the registry is empty -- the FIRST insert into ANY of the nine watched tables
-- aborts and the database is unusable. The is_active half matters just as much: a source the
-- admin switched off must stop producing work rather than keep filling the queue.
```

```
IF NOT EXISTS (SELECT 1 FROM embedding_ingest_registry
WHERE source_schema = TG_TABLE_SCHEMA
AND source_table  = TG_TABLE_NAME
AND is_active) THEN
RETURN NULL;
END IF;
-- On UPDATE, re-embed only when the row's text actually moved. The trigger is FOR EACH ROW on
-- any column, so without this a bump to nl_query.hit_count or a run's state transition would
-- queue a re-embedding of text nobody touched.
IF TG_OP = 'UPDATE' AND embedding_source_text(TG_TABLE_SCHEMA, TG_TABLE_NAME, to_jsonb(NEW))
IS NOT DISTINCT FROM embedding_source_text(TG_TABLE_SCHEMA, TG_TABLE_NAME, to_jsonb(OLD)) THEN
RETURN NULL;
END IF;
INSERT INTO embedding_job (id, source_schema, source_table, source_id, status)
VALUES (gen_random_uuid(), TG_TABLE_SCHEMA, TG_TABLE_NAME, NEW.id, 'pending')
ON CONFLICT (source_schema, source_table, source_id) WHERE status IN ('pending','running') DO NOTHING;
RETURN NULL;
END $$;
CREATE OR REPLACE FUNCTION embedding_clear() RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN
DELETE FROM embedding     WHERE source_schema = TG_TABLE_SCHEMA AND source_table = TG_TABLE_NAME AND source_id = OLD.id;
DELETE FROM embedding_job WHERE source_schema = TG_TABLE_SCHEMA AND source_table = TG_TABLE_NAME AND source_id = OLD.id;
RETURN NULL;
END $$;
DO $$
DECLARE t text;
BEGIN
FOREACH t IN ARRAY ARRAY['document','experiment','experiment_run','cohort','nudge','model_config',
'model_catalog','email_template','nl_query'] LOOP
EXECUTE format('CREATE TRIGGER %I AFTER INSERT OR UPDATE ON %I FOR EACH ROW EXECUTE FUNCTION embedding_enqueue();', t||'_embed_aiu', t);
EXECUTE format('CREATE TRIGGER %I AFTER DELETE          ON %I FOR EACH ROW EXECUTE FUNCTION embedding_clear();',   t||'_embed_ad',  t);
END LOOP;
END $$;

(1a) Registry seed — the nine sources the triggers above watch. The triggers no-op for a table that
is not registered, so without this seed the semantic index would simply never fill; with the
unguarded trigger it was worse, and every insert failed. Idempotent, and current_schema() makes
it self-adjust to whatever app schema the script is run into, matching TG_TABLE_SCHEMA.

Each text_template names exactly the columns that table's GIN full-text index covers, so the
two ways of finding a row -- by keyword and by meaning -- read the same text. profile and
audit_log are deliberately absent: their free text is PII and forensic trail respectively, and
stays keyword-searchable only. chunking_policy is left null so the per-kind default applies.
filter_columns carries owner_id for the six owned tables, which is what narrows a search to
what the asker may see.
INSERT INTO embedding_ingest_registry (source_schema, source_table, text_template, filter_columns) VALUES
(current_schema(), 'document',       E'{{name}}\n{{notes}}',                                                '["owner_id"]'),
```

```
(current_schema(), 'experiment',     E'{{name}}\n{{description}}\n{{scenario}}\n{{system_prompt}}\n{{opening_message}}\n{{notes}}', '["owner_id"]'),
(current_schema(), 'experiment_run', E'{{name}}\n{{notes}}',                                                                      NULL),
(current_schema(), 'cohort',         E'{{name}}\n{{notes}}',                                                                      '["owner_id"]'),
(current_schema(), 'nudge',          E'{{name}}\n{{text}}\n{{notes}}',                                                             '["owner_id"]'),
(current_schema(), 'model_config',   E'{{name}}\n{{notes}}',                                                                      '["owner_id"]'),
(current_schema(), 'model_catalog',  E'{{display_name}}\n{{expertise}}\n{{notes}}',                                                 NULL),
(current_schema(), 'email_template', E'{{name}}\n{{subject}}\n{{body}}\n{{notes}}',                                                 NULL),
(current_schema(), 'nl_query',       E'{{label}}\n{{canonical_prompt}}\n{{notes}}',                                                '["owner_id"]')
ON CONFLICT (source_schema, source_table) DO NOTHING;

--- Vector (HNSW), jsonb (GIN), and Keyword/Full-Text (GIN) Indexes ---
CREATE INDEX embedding_vector_hnsw ON embedding USING hnsw (vector vector_cosine_ops);
CREATE INDEX nl_query_vector_hnsw  ON nl_query  USING hnsw (query_vector vector_cosine_ops);
CREATE INDEX embedding_filter_gin  ON embedding USING gin  (filter_values jsonb_path_ops);
-- one GIN full-text index per text-bearing table, over its free-text columns (queried with
--   WHERE to_tsvector('english', <same expr>) @@ plainto_tsquery('english', :q) ):
CREATE INDEX profile_fts        ON profile        USING gin (to_tsvector('english', coalesce(notes,'')));
CREATE INDEX cohort_fts         ON cohort         USING gin (to_tsvector('english', coalesce(name,'')||' '||coalesce(notes,'')));
CREATE INDEX audit_log_fts      ON audit_log      USING gin (to_tsvector('english', coalesce(actor_label,'')||' '||coalesce(action,'')||'
'||coalesce(note,'')));
CREATE INDEX email_template_fts ON email_template USING gin (to_tsvector('english', coalesce(name,'')||' '||coalesce(subject,'')||'
'||coalesce(body,'')||' '||coalesce(notes,'')));
CREATE INDEX experiment_fts     ON experiment     USING gin (to_tsvector('english', coalesce(name,'')||' '||coalesce(description,'')||'
'||coalesce(scenario,'')||' '||coalesce(system_prompt,'')||' '||coalesce(opening_message,'')||' '||coalesce(notes,'')));
CREATE INDEX experiment_run_fts ON experiment_run USING gin (to_tsvector('english', coalesce(name,'')||' '||coalesce(notes,'')));
CREATE INDEX nudge_fts          ON nudge          USING gin (to_tsvector('english', coalesce(name,'')||' '||coalesce(text,'')||' '||coalesce(notes,'')));
CREATE INDEX model_config_fts   ON model_config   USING gin (to_tsvector('english', coalesce(name,'')||' '||coalesce(notes,'')));
CREATE INDEX model_catalog_fts  ON model_catalog  USING gin (to_tsvector('english', coalesce(display_name,'')||' '||coalesce(expertise,'')||'
'||coalesce(notes,'')));
CREATE INDEX nl_query_fts       ON nl_query       USING gin (to_tsvector('english', coalesce(label,'')||' '||coalesce(canonical_prompt,'')||'
'||coalesce(notes,'')));
CREATE INDEX document_fts       ON document       USING gin (to_tsvector('english', coalesce(name,'')||' '||coalesce(notes,'')));
CREATE INDEX issue_fts          ON issue          USING gin (to_tsvector('english', coalesce(summary,'')||' '||coalesce(triage_note,'')));

--- Issue-Tracker Board Access Paths (btree) ---
CREATE INDEX issue_status_idx   ON issue (status, created_at DESC);   -- default triage-board sort/filter
CREATE INDEX issue_kind_idx     ON issue (kind, created_at DESC);     -- bug/feedback split
CREATE INDEX issue_reporter_idx ON issue (reporter_id);               -- "my submissions"
CREATE INDEX issue_run_idx      ON issue (run_id) WHERE run_id IS NOT NULL;  -- "issues filed against this run"

--- High-Volume Tables: Plain Tables + Keyset Pagination (No Partitioning) ---
chat_round, chat_round_candidate, and audit_log are ordinary tables with simple id primary keys. Deep paging
uses KEYSET pagination — ORDER BY (created_at, id) with a WHERE (created_at, id) < (:cursor) cursor, backed by
the (created_at, id) indexes declared in those tables — which stays O(log n) per page at any depth, so
partitioning is not needed for paging. Range partitioning (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; it
```

```
is a known migration, deliberately deferred rather than pre-wired, to keep the primary keys and foreign keys
simple now.
```

=

▶ **Raw DBML source (verbatim)**