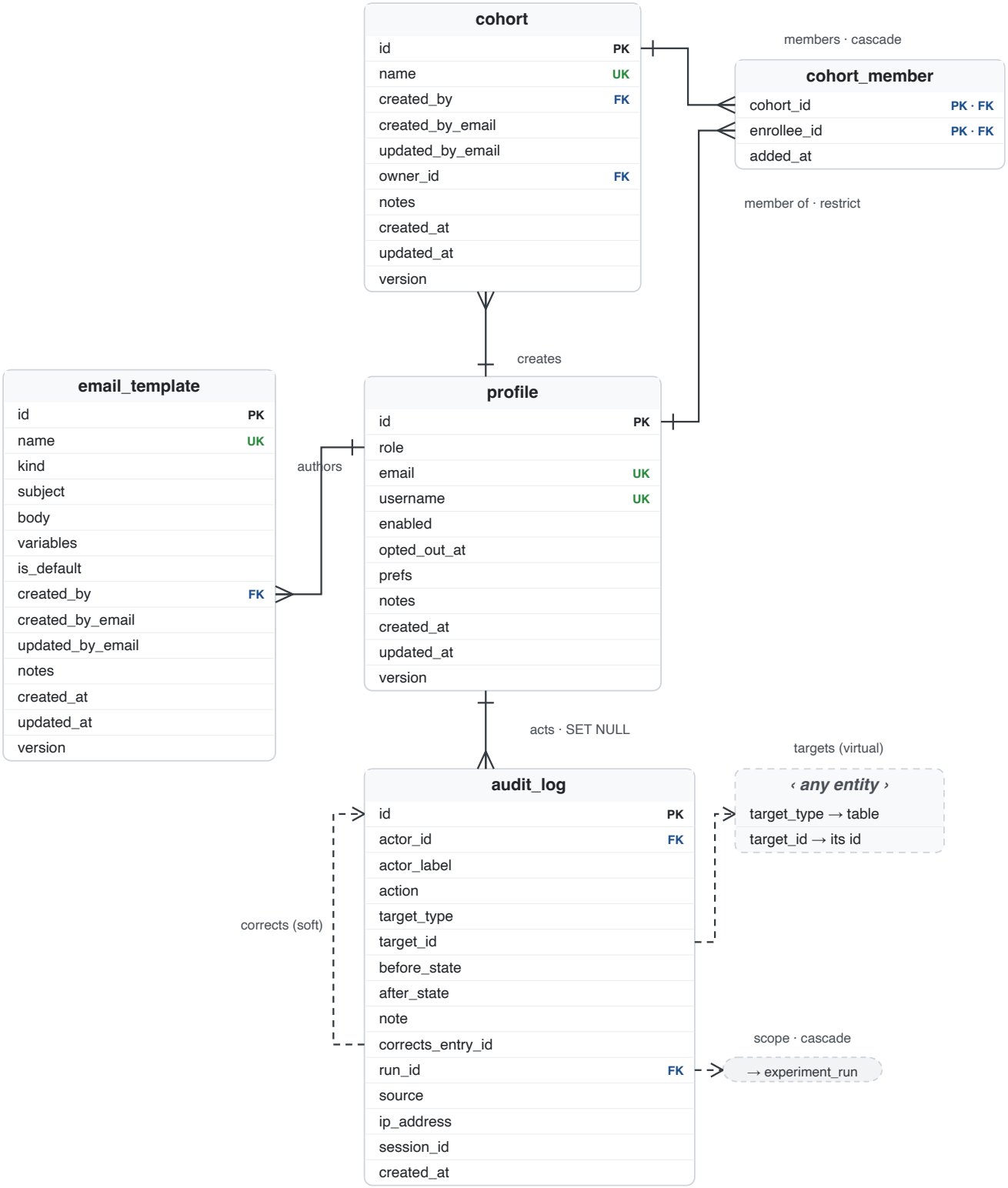
ChatMaestro — Data Model: `core-identity`

Revision: 2026-09-20

This module holds the users, the cohorts they form, and the full audit trail.

Entity-Relationship Diagram



Narrative

What This Module Does

This module is the foundation that the rest of the schema points back to. It holds the users, each with a role; the cohorts, which are named groups of enrollees used to decide who a run — one execution of an experiment — is given to; the audit trail of privileged actions; and the outbound email templates. Nearly every other module has a foreign key into `profile`.

The Tables and Every Significant Column

profile — One Row per User

Column	Meaning · values · when populated
`id`	The user's identity, equal to the Supabase auth user id. It is a UUID, a universally unique identifier. Every relationship in the schema joins on this UUID rather than on email or handle, so it is the one stable anchor. It is the **PK**.
`role`	The user's role, one of `admin`, `experimenter`, or `enrollee`. This is what row-level security (RLS), the database's per-user access rules, keys off. The `admin` role is the install owner. There is exactly one admin, the profile whose email matches the backend `OWNER_EMAIL`, reconciled on boot and held unique by a partial index. Only the admin adds and manages users, both experimenters and enrollees, and experimenters group enrollees into cohorts.
`email`	The account's email address, which is personal data (PII). It is unique — case-insensitively, through a unique index on `lower(email)` — so there is one account per address, which is why it is marked **UK**. Supabase Auth treats one address as one account whatever its 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 matching the account id, and a request is authorized by reading `profile.role` for the id in the caller's token. This email is written by that reconciliation rather than read by it. Supabase Auth is the authority for it, and this column is a copy the app keeps so it can display and query the email under RLS without calling the auth admin API. A user changes their email through Supabase's verified change-email flow, which re-verifies and updates Auth first, and this copy then follows. Because Auth is authoritative and this column only mirrors it, the two never drift.
`username`	A display-only handle, for example `carol_p`. It is unique and normalized to lowercase — a unique index on `lower(username)` enforces the normalization rather than trusting the writer to apply it — which is why it is marked **UK**. It is not the login, and nothing has a foreign key to it. Enrollee handles come from a single running counter, for example `enrollee-0001`, so they are unique by construction; staff handles are readable. They keep names off the screens rather than providing anonymity, since they are sequential and the same in every study.
`enabled`	The suspension gate, which defaults to `true`. A higher-privilege user can disable, or lock out, a lower-privilege one. A disabled user cannot sign in or act, but the account and its data remain. This is independent of deletion: a profile can be hard-deleted, and its audit rows keep the actor through `actor_label`. So `enabled` suspends a user rather than blocking deletion.
`opted_out_at`	When this person withdrew themselves from the enrollee pool, and null while they are still taking part. The enrollee sets it, never staff, which is what separates it from `enabled`: that is a suspension somebody with more privilege applies, this is a decision the participant makes about their own involvement. Both can hold at once and neither implies the other. It is **pool-wide** because the consent was: an enrollee accepts an invitation to the pool rather than to a named study, and studies reach them afterward through cohort membership, so a per-study refusal would decline something they were never separately asked about. Setting it withdraws them from every live run in the same transaction, stops further `run_ready` mail, and removes them from cohort resolution so a later launch cannot quietly re-enroll them — their `cohort_member` rows are filtered rather than deleted, since those record who a cohort held at the time. Everything they produced stays and still counts. Opting back in clears the column but does not restore the enrollments it ended (see `ALGORITHMS §28`).
`prefs`	The user's interface preferences, held as free-form jsonb.
`notes`	Free-text notes a user writes about this user, for example an admin's note on an enrollee or staff account. They are searchable by SQL. They are kept out of the semantic index for privacy, because they are personal data about users, and they are not shown to the user themselves.
`created_at / updated_at`	These record when the row was created and when it was last updated.

Column	Meaning · values · when populated
`version`	A counter that supports optimistic concurrency control (OCC), the check that stops two users from overwriting each other's edits. The counter increases on every save. A save built on a stale copy is rejected, and the edit screen shows the message "record changed — reload."

cohort — A Named Group of Enrollees

Column	Meaning · values · when populated
`id`	The surrogate primary key for the row, marked **PK**.
`name`	A unique, human-readable name, marked **UK**. Membership is an explicit list of `cohort_member` rows; there is no dynamic or sampled membership.
`created_by`	A foreign key to `profile` naming the composer who first created it. It never changes.
`owner_id`	A foreign key to `profile` naming the current owner, which defaults to the composer. The owner edits the cohort, and peers read it and use it to launch their own runs. The admin can transfer ownership. The value goes null if the owner is deleted, which leaves the cohort admin-managed.
`notes`	Free-text notes a user writes about this cohort. They are searchable by SQL keyword, and in the semantic index by meaning.
`created_at / updated_at`	These record when the row was created and when it was last updated.
`version`	A counter that supports optimistic concurrency control (OCC), the check that stops two users from overwriting each other's edits. The counter increases on every save. A save built on a stale copy is rejected, and the edit screen shows the message "record changed — reload."

cohort_member — Who Is in Which Cohort

Column	Meaning · values · when populated
`cohort_id + enrollee_id`	The pair of columns is the row. It is a composite **PK**, and each column is also an **FK**, which is why they are marked **PK·FK**. The row links one cohort to one enrollee, and an enrollee may belong to many cohorts. `cohort_id` is `ON DELETE CASCADE`, so deleting a cohort clears its memberships and leaves the enrollees in place. `enrollee_id` is `ON DELETE RESTRICT`, so an enrollee who is a member of any cohort cannot be deleted until they are removed from every cohort.
`added_at`	This records when the enrollee was added.

Only the admin adds and removes enrollees, and experimenters group those enrollees into cohorts. An enrollee who has taken part in a run is also pinned by `run_enrollment`, so a deletable enrollee is one in no cohort and no run.

audit_log — The Full-Featured, Append-Only Forensic Trail

This table records every consequential action, such as create, update, delete, launch, abort, disable, invite, notify, share, transfer, cleanup, purge, and correct.

Column	Meaning · values · when populated
`id`	The surrogate primary key for the row, marked **PK**. The table is ordinary and unpartitioned; deep paging uses keyset pagination on `(created_at, id)` rather than OFFSET.
`actor_id`	Who acted. It is a soft **FK** to `profile.id`, with `ON DELETE SET NULL`, and it is nullable. It is set while the actor's profile exists, and it goes null if that profile is hard-deleted, after which `actor_label` preserves the identity.
`actor_label`	A deliberate copy of the actor's username and email, captured when the entry is written. It makes the entry self-contained. The origin survives profile edits and deletion or purge, so the log never loses who acted.
`action`	The verb for the action, extensible free 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`.
`target_type · target_id`	A virtual, or polymorphic, foreign key: the kind of entity acted on and its id. It can point at a different table per entry, so it carries no database foreign key, which is the dashed line to "any entity."

Column	Meaning · values · when populated
`before_state · after_state`	jsonb snapshots of the row before and after the change. The before value is null for a create, and the after value is null for a delete — except on the entry that records a run being cleared, whose after value carries the facts that exist nowhere else once the delete has run: the mode, whether an archive was taken or declined, its format, and the object key of the archive in storage. These drive the field-level before-and-after diff view.
`note`	A free-text annotation on an 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 does not create a new row — unlike a correction, which appends a new row via `corrects_entry_id`. It is searchable by SQL keyword, and kept out of the semantic index for privacy.
`corrects_entry_id`	A **soft** self-reference, the corrects loop: a `correct` entry holds the id of the earlier entry it supersedes. It is application-enforced, not a database foreign key — kept a soft reference by design so auditing stays app-driven and uniform — so the diagram draws it as a dashed self-link. Originals are never edited or deleted; a correction is a new appended row. It is null for normal entries.
`run_id`	A scope foreign key to `experiment_run`, crossing modules, with `ON DELETE CASCADE`. It is set for run-scoped actions, so purging a run erases its audit. It is null for account-level actions such as invites and role changes, which survive run purges. It is also null on the entry that records the clearing itself, which names the run through `target_type` and `target_id` instead; a run-scoped entry would be deleted by the very cleanup it records, or cascaded away by the purge.
`source · ip_address · session_id`	Forensic context on the actor at the time: the channel (`web` or `api`), the IP address (`inet`), and the sign-in session, for correlating a sequence of actions. Each is null when not available, such as a system-internal action.
`created_at`	When the action happened. It backs the `(created_at, id)` keyset-pagination index for the append-only trail.

email_template — The Outbound Email Templates (Admin-Managed)

Column	Meaning · values · when populated
`id`	The surrogate primary key for the row, marked **PK**.
`name`	A unique template name, marked **UK**.
`kind`	Which outbound email this template is for: • `invite_experimenter` — the admin invites a new experimenter. • `invite_enrollee` — an enrollee joins the pool. • `run_ready` — an enrollee is notified a run is ready. All kinds are admin-managed.
`subject`	This is the full email subject line.
`body`	The full email body, with merge tokens such as `{{invite_link}}`. It is provider-agnostic: the app composes the whole email and sends it through Supabase or SMTP, with the transport chosen in `.env`. There is no Supabase “standard template” to inject into.
`variables`	This declares the available merge tokens.
`is_default`	There is exactly one default per `kind`, enforced by a partial-unique index on `kind`. The default of a kind is the template used whenever that email is sent.
`created_by`	A foreign key to the admin who authored it. Templates are admin-only, so there is no owner and no transfer, and RLS keeps them invisible to experimenters and enrollees.
`notes`	Free-text notes a user writes about this template. They are searchable by SQL keyword, and in the semantic index by meaning.
`created_at / updated_at`	These record when the row was created and when it was last updated.
`version`	A counter that supports optimistic concurrency control (OCC), the check that stops two users from overwriting each other’s edits. The counter increases on every save. A save built on a stale copy is rejected, and the edit screen shows the message “record changed — reload.”

Each send is recorded in `audit_log`, with action set to `invite` for invitations or `notify` for run-ready notices, capturing the actor, recipient, provider, and result. That is why there is no separate email_log. Detailed email delivery logs are available in the email service provider's console.

How They Relate (Reading the Lines)

- Membership is many-to-many. The members line, from cohort, and the member of line, from profile, both run through the `cohort_member` intersection table, and together they form the many-to-many link.
- creates: one user authors many cohorts, through `cohort.created_by`.
- authors: one user authors many email templates, through `email_template.created_by`.
- acts, with SET NULL: one user accounts for many audit entries. On the rare forced delete the link goes null, but `actor_label` keeps the origin.
- targets, the dashed line: the polymorphic `target_type/target_id` virtual foreign key to any entity.
- corrects, the self-loop: a correction entry points back at the entry it supersedes.
- scope, with cascade, drawn dashed and off-page: the run-scope foreign key to `experiment_run`.

Audit Behavior — Append-Only, Corrections, Visibility, Clearing

- Append-only, with one permitted edit: adding a note annotation (RLS allows updating only the `note` column). To fix an entry you annotate it, or correct it with a new entry that points back — the history extends, never mutates. RLS blocks all other updates, and blocks deletes except the paths below.
- Role-scoped visibility (RLS): the admin sees everything, an experimenter their own and their enrollees' actions, an enrollee only their own.
- Three sanctioned deletes: the admin's age-based sweep of old entries (optionally archived to file first); the run-scoped cascade when a run is purged; and a run-scoped Cleanup by an admin or the owning experimenter, deleting that run's audit entries while keeping the run.

Clearing a Run (And What It Takes with It)

An `experiment_run` may be cleared only when it is done or aborted, and a running or paused run is protected. The run's transcripts and scorecard are always archived to a file, in `xlsx`, `csv`, or `json` form, before anything is deleted. If the export fails, nothing is deleted. There are two ways of clearing a run, with distinct intents:

- **Cleanup**, run by an admin or by the experimenter who composed the run, through `experiment.created_by`, or launched it, through `experiment_run.launched_by`, reclaims everything the run produced: enrollees, chat rounds and candidates, operator messages, and the run's own audit entries. It keeps the run row and its scorecard, `run_result`, as the durable record, so the analytics survive.
- **Purge**, admin only, additionally removes the `experiment_run` itself, which cascades the scorecard away too.

Neither touches the reusable definitions: the experiment, the model configuration and its models, the cohort and its members, the underlying documents, or the experiment's context files. So clearing a run removes output, never design.

The run's `audit_log` rows are part of the produced data that both reclaim, through the `run_id` cascade above.

Note: a database reset for demos (reseeding from a seed dump) is a middle-tier operation recorded in `audit_log`, not a schema table; general backup and restore is left to the Postgres/Supabase infrastructure, which the app does not duplicate.

Cohorts and Parallel Runs

Cohorts are a shared, reusable pool: an enrollee can be in one cohort for one study and another later. But two runs in flight at once must target non-overlapping cohorts — no enrollee may be live in two runs simultaneously, or their data is contaminated by two conditions in parallel. This is checked at launch (the run lives in core-experiment): sequential overlap is fine, concurrent overlap is not.