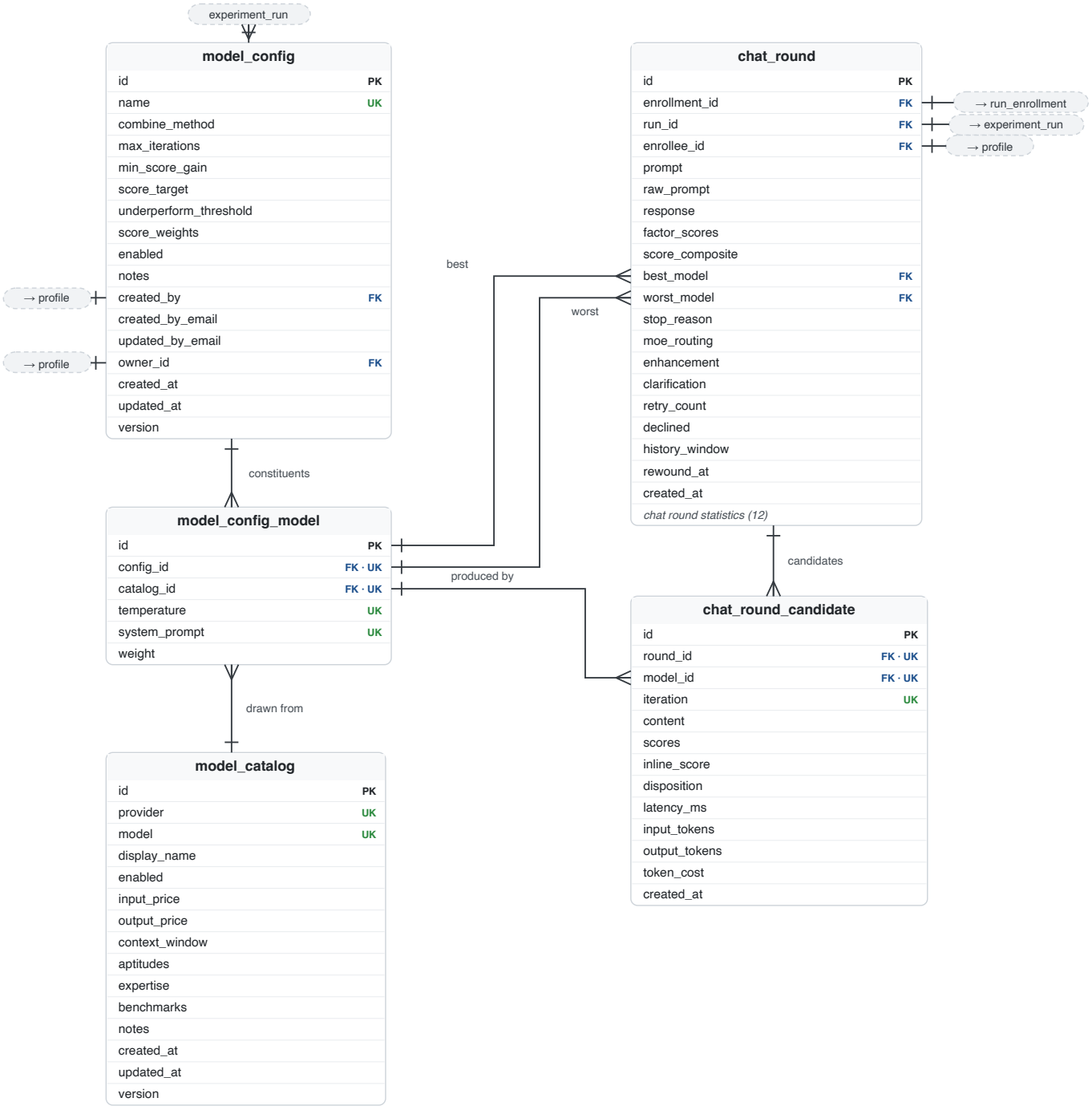
ChatMaestro — Data Model: `llm-chat`

Revision: 2026-09-20

This module comprises the reusable model side — the model catalog, a model configuration, and its constituent models — and the runtime: each scored chat round and the model-by-iteration candidates behind it.

Entity-Relationship Diagram



Narrative

What This Module Does

This module covers the model side and the runtime of the chat task, the conversation between an enrollee and a large language model (LLM). The model side is one logical model to answer with: a reusable `model_config`, whose constituent models are each drawn from an admin-curated `model_catalog`. The runtime is what actually happened: each scored `chat_round`, which is one prompt-to-best-response exchange; and the model-by-iteration `chat_round_candidate` rows behind it. The prompts, the one trusted judge, and the per-run `run_result` scorecard all live on the experiment side, in core-experiment.

It is simple by default and sophisticated on demand. The common case is one model: a config with a single constituent and `combine_method = single`, which short-circuits the whole ensemble, judge, and iterate machinery. Adding more constituents, a combine method, self-improvement, or per-factor score weights is all opt-in.

model_catalog — The Install'S Model Menu (Admin-Curated)

The list of models this install offers. It is database-backed, so the admin curates it live: the admin adds a newly released model or retires a disappointing one by editing rows, with no code deploy. It holds no secrets. The API key is a backend secret in `.env`, and keys are per-provider rather than per-model, so a new model from a provider you already use needs no new key.

Column	Meaning · values · when populated
`id`	The surrogate primary key for the row, marked **PK**.
`provider · model`	The vendor, such as `anthropic`, `openai`, or `google`, and its API model-id string, for example `claude-opus-4-8`. They are unique together, marked **UK**, so a model's identity is defined once here.
`display_name`	A friendly label shown in pickers, for example "Claude Opus 4.8."
`input_price · output_price`	Per-token prices in US dollars per 1,000,000 tokens. They are separate because providers bill prompt (input) and completion (output) tokens at different rates, and output is typically several times input. A self-hosted or local model is set to 0, or a near-zero figure covering compute, which is how it shows up as near-free in cost reports. The prices are read at round time to freeze each candidate's cost, so a later price edit never rewrites history.
`context_window`	How many tokens this model can accept in one call, prompt and history together. It is **mandatory**, curated alongside the prices and for the same reason: nothing else in the system knows it, and a wrong value is not recoverable from a provider's response. It is what bounds the replayed chat history. A configuration's members may have very different windows, and every member has to receive the identical input, or the comparison stops being between models and becomes a comparison of how much each was told — so the budget comes from the **smallest** `context_window` among the 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. Room for the system prompt, the retrieved context, the current prompt and a reserve for the answer all come off this figure before history is fitted (see `ALGORITHMS §23`).
`enabled`	An admin toggle. A disabled model drops out of pickers but still resolves for existing configs and history, so a model is retired rather than deleted.
`aptitudes`	Advisory tags, of type `text[]`, describing what the model is good at, such as `judge`, `coder`, `reasoning`, `vision`, `long-context`, or `general`, used to filter the picker. They are admin-curated, and the system never makes a correctness decision from them.
`expertise`	One curated sentence saying what this model is good at, written for meaning rather than for filtering, such as "Long-form reasoning and careful multi-step analysis over large documents." It is the text the mixture-of-experts (MoE) router matches an incoming prompt against, by embedding the sentence once and comparing it with the prompt by meaning, so it is the one column here the system itself reads. Where `aptitudes` is a short tag list for people, this is prose for the router, and when it is left empty the router falls back to the tags joined with `notes`, a weaker match that the authoring screen warns about.
`benchmarks`	Optional published scores, as jsonb, for reference and display, for example `{"swe_bench":0.65,"mmlu":0.88}`.
`notes`	Free-text notes a user writes about this model. 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.

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."

model_config — One Logical Model (Reusable, Named)

It is authored independently, like a cohort. A run picks one and applies it uniformly, not per enrollee.

Column	Meaning · values · when populated
`id`	The surrogate primary key for the row, marked **PK**.
`name`	A reusable name, marked **UK**, for example "GPT-5 + Claude ensemble."
`combine_method`	How the constituent generators become one output: • `single` — the one member answers, the short-circuit. Required for a one-member config, disallowed for a multi-member one. • `synthesize` — the highest-weight member merges all answers, so it doubles as the aggregator. • `vote` — the generators peer-score on the rubric; the highest weighted tally wins. • `consensus` — the answers are grouped by meaning; the largest weighted cluster's most central answer wins. • `judge_best` — all models answer, the experiment's judge scores them, and the top wins, with a tie broken at random. • `moe` — mixture of experts: a router picks ONE member to answer instead of running them all, so the round costs one generation however many members there are. The router embeds the prompt and compares it with each member's `model_catalog.expertise`; a clear winner answers with no extra model call, and a tie escalates to the experiment's trusted judge at temperature 0, which predicts the best contender. The decision is recorded on `chat_round.moe_routing`. It needs at least two members.
`max_iterations`	The self-improvement cap: stop after this many looped improve-then-score passes. It is method-general and applies to every combine method. 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`	The convergence gate: stop once a pass gains less aggregate score than this. It is also method-general.
`score_target`	The good-enough bar, from 0 to 100, 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.
`underperform_threshold`	A value from 0 to 100, defaulting to 50. A generator scoring below this in an iteration is pruned from later iterations.
`score_weights`	Optional per-factor weights for the composite, over groundedness, relevance, coherence, and instruction-following, held as jsonb. A null value means equal weights, the arithmetic mean. It applies wherever factor-scoring is used.
`enabled`	An owner or admin toggle. A disabled configuration drops out of pickers for new runs but still resolves for existing runs and history. It is the way to retire a configuration that a past run references and so cannot be deleted.
`notes`	Free-text notes a user writes about this configuration. They are searchable by SQL keyword, and in the semantic index by meaning.
`created_by`	A foreign key to `profile` naming the composer who first authored it. It never changes.
`owner_id`	A foreign key to `profile` naming the current owner, which defaults to the composer. The owner edits the configuration, and peers read it and use it in their own runs. The admin can transfer ownership. The value goes null if the owner is deleted, which leaves the configuration admin-managed.
`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, bumped on every save. Its constituent `model_config_model` rows are versioned through this parent, so a child edit bumps it too, and the whole configuration is guarded as one unit. A stale save is rejected with the message "record changed — reload."

model_config_model — The Constituent Models (All Generators)

Each row is one model's use inside this config. A model's identity lives once in `model_catalog`, and the knobs here are per-use junction attributes. The same catalog model can appear in another config — or twice in one config at a different temperature or prompt override — so they are not redundancy. A distinct-member unique on `(config_id, catalog_id, temperature, system_prompt)` (with `NULLS NOT DISTINCT`) permits exactly those variations while blocking exact-duplicate rows; its leading `config_id` also serves config-scoped lookups, so no separate index on it is needed.

Column	Meaning
`id`	The surrogate primary key for the row, marked **PK**.
`config_id`	A foreign key to `model_config`, the config this model belongs to.
`catalog_id`	A foreign key to `model_catalog`, naming which model this constituent is, since its provider and model-id are named once in the catalog. A catalog model in use cannot be hard-deleted; it is retired through `enabled`.
`temperature`	The per-use temperature, the one broadly supported generation dial, trading output diversity against determinism, so it is the deliberate per-model knob. Reasoning models that reject a temperature other than 1 are skipped by a capability list.
`system_prompt`	The per-model layer of the two-layer system-prompt cascade. It is combined, through its mode of `inherit` or `append`, with `experiment.system_prompt` (the base) to give this model its final prompt. It lets the same catalog model play a different role inside one configuration, such as skeptic against optimist, which is what makes ensembles meaningful.
`weight`	A relative number, defaulting to `1.0`, not a percentage. It is used for `vote` (weighting each ballot), `consensus` (summing to give each cluster its support), `synthesize` (the highest-weight member is the aggregator that merges the rest, ties by a stable order), and `moe` (it doubles as the member's tier, a low weight reading as the cheap or fast expert and a high one as the strong expert, and the highest-weight contender is the router's deterministic tie-break), and only for generators. There is no separate lead flag; the `single` method uses a one-member configuration. Weights are normalized at combine time, as $w_i / \Sigma w$, so a value does not depend on how many models are in the config, and all weights of `1.0` means equal. The config author sets it. You would weight unequally not only for "this model is better," but also for diversity, to keep a cheaper model for a different view while trusting it less; for domain fit, to up-weight a specialist; or for a sensitivity sweep.

Why only these per-use knobs? `temperature` and the prompt override are the two levers that meaningfully vary an experiment and are portable across providers. Other generation parameters — `top_p`, `max_tokens`, and frequency and presence penalties — are provider-specific, rarely the experimental variable, or redundant with temperature, so they are left at defaults. Any specific one can be added if a study needs it.

chat_round — The Scored Exchange (Analytics Unit + Transcript)

One prompt-to-best-response exchange, already scored. The quality scores are the experiment's one trusted judge's scores for the chosen response, applied identically to every run so the experiment's runs stay comparable within it, and the objective figures are recorded at runtime.

Column	Meaning
`id`	The surrogate primary key, marked **PK**. The table is an ordinary, unpartitioned table; deep paging uses keyset pagination on `(created_at, id)` rather than OFFSET.
`enrollment_id · run_id · enrollee_id`	The foreign keys: the enrollee's membership, in `run_enrollment`, and the run and enrollee, which are a deliberate copy for run-scoped analytics without a join. `run_id` cascades on run purge, and `enrollee_id` is set null on profile delete.
`prompt · raw_prompt · response`	The effective prompt actually submitted, the enrollee's original message, and the single shown response (a copy of the chosen candidate's text). When the experiment's `enable_prompt_enhancer` is on, `prompt` holds the enhanced rewrite and `raw_prompt` the enrollee's original; when off, `prompt` is the original and `raw_prompt` is null (see ALGORITHMS §18).
`factor_scores`	The judge's per-factor scores for the chosen response, 0 to 100, as a jsonb bag keyed by the experiment's rubric factors — for the built-in default rubric `groundedness` (supported by a source), `relevance` (addresses the prompt), `coherence` (internally consistent), and `instruction_following` (did what was asked); a custom rubric defines its own, say funniness or originality. A factor that did not apply this round, such as groundedness with no grounding source, is absent rather than zero. It is a jsonb bag, not fixed columns, so the factor set is configurable per experiment.

Column	Meaning
`score_composite`	The weighted mean of the round's applicable factors, following `experiment.score_weights` renormalized over the factors that applied, from 0 to 100. It is the headline the comparisons rank by, kept as its own column rather than inside `factor_scores`.
`best_model · worst_model`	Foreign keys to `model_config_model`: the generator that produced the chosen response, and the lowest-scoring generator this round, the dud signal, resolved to catalog identity.
`stop_reason`	Why the loop ended: `score_target`, `min_gain`, `max_iterations`, or `single_pass`.
`moe_routing`	The mixture-of-experts routing decision for this round, as jsonb, and null under every other combine method. It holds the stage that decided (embedding or judge), the member routed to, the per-member similarity scores, the contenders a tie considered, and any fallback that fired. `best_model` already names the expert that answered; this records *why* it was picked, which is what makes the router auditable and lets a run report how often it escalated. The judge-router's own tokens land in `judge_tokens` and `judge_cost`, since the router is that same trusted judge.
`enhancement`	What prompt enhancement did to this round, as jsonb, and null when the experiment's `enable_prompt_enhancer` is off. It records the outcome (`applied` when the rewrite passed every check, `fell_back` when it did not), the transformations the enhancer declared, the check that rejected a failed rewrite, the enhancer version that ran, and the tokens and latency the call spent. It exists because a fallback is otherwise invisible: a rejected rewrite leaves `prompt` holding the enrollee's original, which looks exactly like enhancement having been switched off. The study argument rests on the enhancement policy applying uniformly to every enrollee, so a silent fallback is precisely the event that has to leave a trace. See `ALGORITHMS §18`.
`clarification`	The exchange that preceded this round when the enrollee's earlier attempts did not pass the sufficiency gate, as jsonb, 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 it spent, and which reply style was in force. 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, while a rewrite has nothing to pair and the earlier drafts stay here as record only. A turned-back attempt is deliberately not a round of its own, since rounds are the unit every statistic counts. See `ALGORITHMS §21`.
`retry_count`	How many attempts the sufficiency gate turned back before the attempt 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 therefore has to be groupable and averagable without digging into jsonb, and because it survives if the clarification detail is ever trimmed. Both settings of `experiment_run.socratic_enabled` increment it, which is what makes them comparable: the question a study asks is whether assembling a request through dialogue takes fewer attempts than asking the enrollee to rewrite it. It counts attempts rather than questions, so a reply asking for two tightly linked pieces at once still counts as one. See `ALGORITHMS §21`, rolled up by `§6`.
`declined`	Whether the response refused the request rather than attempting it, as a single enum, and null when it attempted it. `appropriate` means the refusal was warranted, by the system prompt's own bounds or by there being no source to answer from; `unwarranted` means it declined with nothing to justify that. 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. 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. It is distinct from `clarification`, which turns a request back before any round exists; this records a refusal by the models under test. See `ALGORITHMS §4`.
`history_window`	What conversation history this round actually carried, as jsonb, and null when the experiment's `enable_chat_history` is off. It records the turns included and the turns available, the tokens the history occupied, the budget it was fitted into, 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. A truncation that is not recorded cannot be told from one that never happened, and the stakes are higher here than for a fallback: `model_config` is a swept variable, so two arms of one comparison can have different smallest windows and therefore different effective memory. Without this the experimenter reads that as a quality difference. `alert_shown` is kept for a parallel reason — a warned enrollee may write more tersely or wrap up, and that lands only on long sessions, which is itself an outcome (see `ALGORITHMS §23`).

Column	Meaning
`rewound_at`	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 discards everything after it: those rounds stop being replayed to the models, 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. It is a timestamp rather than a flag so the order of rewinds within a session stays legible.
chat round statistics — the 12 detail columns	The round's ordering, selection scores, and objective figures: • Transcript ordering — `session_seq` names the session (it increments on clear-and-new, while a reconnect keeps the session), and `round_seq` is the round's order within it. • Self-improvement scores — `inline_score` is the judge's selection score that picked the winner, on the same scale as the factor scores; `best_score` and `worst_score` are the best and worst generators' inline scores this round; and `num_iterations` counts the self-improvement passes, where 0 is a single pass. • Cost and usage — `latency_ms` is the parallel-aware wall-clock the enrollee waited (the span of the slowest generator when they run at once, not the sum of the calls); `tokens` is the round total, input plus output across candidates; `token_cost` is the generator cost in US dollars, the sum of `chat_round_candidate.token_cost` across every candidate and iteration, excluding the judge; and `judge_tokens` and `judge_cost` are the judge's tokens and US-dollar cost, kept separate because the judge produces no candidate row of its own, so the full round spend is `token_cost + judge_cost`. • Outcome — `error` is the failure detail (hung, error, or sysfail) as jsonb, null on success, and `created_at` is the round's plain timestamp.

chat_round_candidate — The Competition Behind a Round (Optional Depth)

Column	Meaning
`id`	The surrogate primary key, marked **PK**. The table is ordinary and unpartitioned.
`round_id`	A foreign key to `chat_round.id`, with `ON DELETE CASCADE`, so a candidate is removed with its round. It also joins the distinct-member unique, so it is marked **FK · UK**.
`model_id · iteration`	The rest of the distinct-member unique `(round_id, model_id, iteration)` — one candidate per generator per self-improvement pass. `model_id` points to the generator, in `model_config_model`, so it is marked **FK · UK**; `iteration` is marked **UK**.
`content`	The candidate text. It is retained for every candidate, winners and losers, so the "see all" side-by-side view works, and the chosen one's text is also copied to `chat_round.response`. Retention is governed by `experiment_run.keep_candidates`: true keeps them for review, false clears them as each round finalizes.
`scores`	The judge's per-factor breakdown for this candidate, as jsonb, over groundedness, relevance, coherence, and instruction-following.
`inline_score`	The judge's aggregate score for this candidate, from 0 to 100, that drove selection and convergence.
`disposition`	This candidate's final state, as a single enum: • `chosen` — its text became `chat_round.response`, the single winner across all iterations, since the loop is not monotonic. • `pruned` — it fell below `underperform_threshold` and was dropped from later passes. • `none` — it was generated and scored but was neither. There is exactly one `chosen` per round.
`latency_ms`	The wall-clock time in milliseconds for this one model call. Wall-clock includes queue wait, network round-trips, and provider-side scheduling, so summing these across a round's candidates gives cumulative latency — not compute time, and not the round's parallel-aware wall-clock, which the candidates overlap.
`input_tokens · output_tokens`	The prompt and completion token counts for this call, from the provider's API usage field, or the model's own tokenizer for a local model. They are per-candidate rather than per-round, because tokenizers differ by model, per-model prompt overrides change the text, and each self-improvement pass refines the input.

Column	Meaning
`token_cost`	The candidate's cost in US dollars, frozen at round time, computed as `input_tokens/1e6 × input_price + output_tokens/1e6 × output_price`, using this model's catalog prices then in effect. There is one model per candidate, so it is a single model's cost, and the round total is the sum over candidates. It is near-zero for a local model.
`created_at`	When this candidate was produced.

How They Relate (Reading the Lines)

- constituents: one `model_config` has many constituent models, all generators (one is the primary).
- drawn from: each constituent names which model it is, from the `model_catalog`.
- candidates: one `chat_round` has many candidates, over generators and iterations.
- produced by, best, and worst: each candidate, and each round's best and worst, name a constituent generator.
- dashed tags: cross-module foreign keys to `profile` for the config author and the round's enrollee; and to the `run_enrollment` and `experiment_run` a round belongs to. The per-run `run_result` scorecard lives in core-experiment, where it hangs off `experiment_run`.

Combining & Self-Improvement (Behavioral, Not Columns)

The middle tier runs the generators and applies `combine_method` to get the one output. If it is `single`, it simply calls the one member and skips all of this. If it is `moe`, it inverts the shape: rather than fan out and reduce, it *routes* first and calls one member. It embeds the prompt and compares it with each member's `model_catalog.expertise` by meaning; when one member is a clear winner it answers, and when several tie the experiment's trusted judge is asked, at temperature 0, to predict which of the tied contenders will answer best. Either way one member generates, the decision is written to `chat_round.moe_routing`, and the answer is scored exactly as under `single`. When self-improvement is enabled, it loops improve, score, and refine, a Mixture-of-Agents (MoA) approach that feeds the best or aggregate answer back to the generators. It stops as soon as `score_target` is reached, the gain drops below `min_score_gain`, or `max_iterations` is hit. Each pass is persisted as a candidate, and generators below `underperform_threshold` are pruned.

One trusted judge does all the scoring. The experiment's judge (`experiment.score_judge_*`) grades each candidate on the experiment's rubric: that score picks the winning candidate and iteration and decides when to stop improving (`inline_score`), and the chosen response's scores are the round's `factor_scores` and `score_composite`. Because it is one judge on one rubric applied identically to every run, the scores are comparable within the experiment without a second pass. For `judge_best` the judge also selects the top candidate; for `single`, `synthesize`, `vote`, and `consensus` it scores whatever answer the method produces, so the loop always has a number on the same scale to act on.