ChatMaestro — Algorithms Reference

Revision: 2026-09-20

Overview

This document is the implementation specification for the non-trivial logic in ChatMaestro: the places where a silent bug produces a wrong but plausible number rather than a crash. Ordinary plumbing — fan-out, retries, create-read-update-delete operations, queue draining — is out of scope, except where one subtle rule sits inside it.

Notation and Conventions

An algorithm whose essence is control flow, meaning a loop, a branch, a state machine, or literal SQL, is given as pseudocode. One whose essence is a formula, an aggregation, or a set of declarative rules is described in prose. A state machine additionally gets a state-transition diagram.

The pseudocode is written in an Algol/Pascal style, in which assignment is `:=`. It is illustrative notation rather than the target implementation language: the system is built in Python.

Calling an implementation **hand-coded** — the per-factor scorer (§4), for example — means its logic is written directly in the middle tier rather than handed to a third-party evaluation library such as RAGAS, DeepEval, or TruLens, so that the rubric and its scoring remain ours to freeze and version. The term says nothing about who wrote the code; in this project it is written by Claude, the AI assistant building the application.

Tunable Constants

Every numeric threshold, cap and timeout follows one convention: a reasonable default is hard-coded and a correspondingly named `.env` parameter optionally overrides it. Each section lists its own under *Tunables*, where the quoted value is that default; the appendix collects them all.

Core Concepts

Three ideas frame every section below.

An experiment is an immutable comparison frame. Once it has runs, its substantive fields — the base `system_prompt`, `opening_message`, `scenario`, `blinded` flag, attached context documents, and the whole scoring configuration — become read-only.

A run's fidelity comes from immutability rather than from a snapshot. An `experiment_run` is one cell of a factorial grid: foreign keys to the three swept definitions (`nudge_id`, `model_config_id`, `cohort_id`), the `socratic_enabled` switch that is the fourth swept variable, plus `state`, `keep_candidates` and timestamps. Only `state` mutates, and there is no `condition_snapshot`. The referenced definitions are held immutable while any run references them, through a database `RESTRICT` rule, so a past run's exact conditions can always be resolved again — the comparison conditions are reproducible even though a single generation is not.

One trusted judge does all the scoring. Every experiment names one judge model and one rubric, frozen on the `experiment` (`score_judge_provider`, `score_judge_model`, `score_rubric_version`, `score_weights`). That judge scores candidates to pick a round's winner and to drive the self-improvement loop, and the winning answer's factor scores are the round's analytics scores — one judge, one rubric, one 0–100 scale, no second pass. Its spend is captured per round in `chat_round.judge_cost` and kept separate from `total_generator_cost`.

For the chosen candidate, the selection score and the analytics score are the same number, produced by the same judge in the same call.

Checklist of Algorithms

Algorithms

1. Combine Methods — Many Candidates into One Answer

Purpose. A `model_config` may hold several constituent models. This algorithm turns whatever they produce in one round into the single `response` shown to the enrollee. The method is chosen by `model_config.combine_method`, one of `single`, `synthesize`, `vote`, `consensus`, `judge_best`, or `moe`.

Five of the six **fan out and then reduce**: every member answers, and a reduce step picks or merges. `moe` inverts that — it **routes first and calls one member** — so it is described here in outline and specified in full in §19.

Inputs / outputs.

- Inputs: the round's assembled context (resolved system prompt per model, the replayed chat history, the enrollee prompt, retrieved RAG context) — the enrollee prompt here is the *effective* one, enhanced per §18 when `enable_prompt_enhancer` is on, and the history is the window §23 assembled, empty when `enable_chat_history` is off; the generator rows (`model_config_model`) and their `weight` (whose maximum designates the `synthesize` aggregator); the experiment's one trusted judge (`experiment.score_judge_*`) and its `score_weights`.
- Outputs: one chosen answer written to `chat_round.response`, plus the selection bookkeeping `best_model`, `worst_model`, `best_score`, `worst_score`, and `inline_score` (the winner's composite, the number that *picked* it). Every generation is persisted as a `chat_round_candidate`.

Procedure. Under the five fan-out methods all generators are called **in parallel** and each output is persisted as a `chat_round_candidate`; the reduce step then depends on `combine_method`. Under `moe` the router runs *before* any generation and only the routed expert is called.

`single` — the configuration's lone model answers. `single` is **required** for a one-member configuration and is not a valid choice for a multi-member one (enforced in the application and the authoring UI). One candidate, no contest; the one trusted judge (§4) still scores it, so the round has its analytics scores and the loop has a number. `best_model` = that model.

`synthesize` — all generators answer; the highest-`weight` member (the *aggregator*) merges them into one response aimed at the rubric, with the candidates anonymized so it weighs content over brand. The one trusted judge then scores the merge.

```
candidates := parallel_generate(generators)      # each answer is persisted as a candidate
aggregator := the generator with the greatest weight     # break ties with a stable order

# 1. Build the merge prompt deterministically. It contains the aggregator's role;
#    the instruction "merge these into ONE answer better than any single candidate:
#    keep the complementary points, on a conflict prefer the better-supported claim,
#    drop the unsupported, and do not simply concatenate them"; the original prompt
#    and its retrieved context; the rubric's factor names as the target; and the
#    candidate answers, shown anonymously as "Answer 1, Answer 2, ..." and never
#    labeled by model, so the aggregator cannot favor its own.
merge_prompt := build_merge_prompt(aggregator, prompt, context,
                                   rubric.factors, anonymize(candidates))

# 2. A single merge call produces the round's response. Unlike every other method,
#    `synthesize` does not PICK a winner from what the members wrote — it manufactures
#    a new answer none of them produced. So the round persists N + 1 candidate rows:
```

```
#    the N member answers above, which stay `none` because they were neither chosen
#    nor pruned, and this merged one, which becomes chat_round.response.
response := aggregator.call(merge_prompt)
persist candidate(response, disposition = chosen)

# 3. The one trusted judge scores the merged answer (see §4); best_model is the aggregator.
inline_score := judge.score(response)
```

`vote` — a rubric-based *peer vote*: every generator scores every *other* candidate on the same rubric the judge uses, and the highest voter-weighted tally wins. The tally only *selects* — the one trusted judge then scores the winner for the official number.

```
candidates := parallel_generate(generators)      # each answer is persisted as a candidate

# 1. Peer scoring. Each generator scores every OTHER candidate on the same rubric
#    the judge uses (the render step, the applicable factors, and score_weights from
#    §4). A generator never scores its own answer, which blunts self-preference.
for each voter in generators:
    for each candidate in candidates whose model is not the voter:
        peer_score[candidate][voter] := voter.score(candidate, rubric)

# 2. Tally each candidate's peer scores, weighting each ballot by the voter's weight.
for each candidate in candidates:
    tally[candidate] := 0
    for each voter who scored this candidate:
        tally[candidate] := tally[candidate] + voter.weight * peer_score[candidate][voter]
winner := the candidate with the highest tally       # break ties at random

# 3. Selecting is not scoring: the one trusted judge scores the winner (see §4), and
#    THAT composite -- not the peer tally -- becomes the analytics score / inline_score.
response := winner.text
inline_score := judge.score(winner)                  # the peer tally is kept as metadata
```

`consensus` — the answer the models most *agree* on wins. The candidates are embedded into meaning vectors, grouped into clusters of near-identical meaning, and the cluster with the greatest summed generator weight is the majority; its most central answer is returned. As with `vote`, agreement only *selects* — the one trusted judge then scores that winner for the official number.

```
candidates := parallel_generate(generators)      # each answer is persisted as a candidate

# 1. Embed each candidate into a meaning vector (the same embedding used for RAG),
#    so answers that mean the same thing sit close together. active_embedding_model is
#    the install's EMBEDDING_MODEL setting, the same one RAG uses (§7).
for each candidate in candidates:
    vector[candidate] := embed(candidate.text, active_embedding_model)

# 2. Group the candidates into clusters of near-identical meaning: two answers join the
#    same cluster when their cosine similarity is at or above CONSENSUS_SIMILARITY.
```

```
clusters := group the candidates by meaning, using vector[...] and CONSENSUS_SIMILARITY

# 3. Each cluster's support is the summed weight of the generators that landed in it;
#    the cluster with the greatest total weight is the majority opinion.
for each cluster in clusters:
    cluster.support := 0
    for each candidate in cluster:
        cluster.support := cluster.support + candidate.model.weight
winning_cluster := the cluster with the greatest support   # break ties at random

# 4. Return the winning cluster's MEDOID: the actual answer closest to all the others
#    in the cluster (a real answer, never an average).
winner := the medoid of winning_cluster

# 5. Selecting is not scoring: the one trusted judge scores the winner (see §4), and
#    THAT composite becomes the analytics score / inline_score.
response := winner.text
inline_score := judge.score(winner)              # cluster support kept as metadata
best_model := winner.model
```

`judge_best` — the one trusted judge scores every candidate on the rubric and the highest composite wins, so here selecting and scoring are the *same* pass.

```
candidates := parallel_generate(generators)      # each answer is persisted as a candidate

# 1. The one trusted judge scores EVERY candidate on the rubric's factors, weighted
#    by experiment.score_weights (see §4), giving one composite score per candidate.
for each candidate in candidates:
    composite[candidate] := judge.score(candidate, rubric)

# 2. The highest composite wins (break ties at random). Selecting IS scoring here, so
#    the winner's composite is directly the analytics score / inline_score.
winner := the candidate with the highest composite
response := winner.text
inline_score := composite[winner]
best_model := winner.model
```

`moe` — *mixture of experts*: a router picks **one** member to answer instead of running them all. It decides in two stages — embedding similarity between the prompt and each member's capability text, which is free, deterministic and needs no model

call; then, only when several members tie, the experiment's trusted judge at temperature 0 predicting which contender suits the prompt. Downstream is identical to `single`: one candidate, scored by that same judge.

```
# The router runs BEFORE generation and returns ONE member. Full specification in §19;
# in outline, and with MOE_TIE_MARGIN deciding what counts as a clear win:
#   sim[m]     := cosine(embed(prompt), capability_vector(m))  for each member
#   contenders := every m within MOE_TIE_MARGIN of the leader and above MOE_MIN_SIMILARITY
#   one contender  -> that member answers; no model call was needed
#   several        -> a tie: the trusted judge picks among them at temperature 0
expert := route(prompt, generators, experiment)
persist chat_round.moe_routing := the routing decision   # which stage decided, and why

candidates := [ expert.generate(prompt, context) ]   # ONE generation, however many members exist
response   := candidates[0].text

# The one trusted judge scores it exactly as under `single` (see §4).
inline_score := judge.score(candidates[0])
best_model   := expert
```

Regardless of method, record `best_model` / `worst_model` (both resolve to a constituent model's catalog identity), `best_score` / `worst_score`, and `inline_score` on `chat_round`.

Where the token counts come from. Every provider the router targets returns a `usage` field on the response, and that is the number used, because it is the number the provider bills. Counting locally with the model's tokenizer is a fallback for a self-hosted model that reports nothing, and it is **approximate**: a local count sees only the text, not the chat-template wrapping and special tokens the provider includes in its own total, so a cost derived from it will not reconcile with an invoice. A candidate whose count came from a local tokenizer is recorded as such, so a cost report can say which figures are exact.

Key decisions & edge cases.

- **`inline_score` and the analytics score are the same number for the winner.** Both are the trusted judge's composite for the chosen candidate, on one 0–100 scale — because one judge produces both. There is no separate, less-comparable "config score."
- **`synthesize` is the one method whose winner no member wrote.** Its round holds N + 1 candidates: the N member answers at `none`, and the merge at `chosen`. Nothing on the row says which kind it is, and nothing needs to — the round's `model_config.combine_method` settles it, since under `synthesize` the chosen candidate is always the merge.
- **Selecting is not scoring.** Under every method the winner is chosen by that method's own signal — peer tally, cluster support, judge composite — and then scored by the one trusted judge. That composite is the round's `inline_score` and its analytics score, so all six methods stay on one comparable 0–100 scale. A peer tally or cluster support is kept as candidate metadata.
- **`consensus` selects by agreement, not quality.** Agreement is not correctness: models confidently wrong in the same way produce a wrong majority cluster.
- **A singled-out member is derived, never flagged.** `single` uses a one-member config, and `synthesize`'s aggregator is the highest-`weight` member, ties broken by a stable order so a re-run is reproducible. `vote`, `consensus` and `judge_best` single out no member.
- **`vote` is the costliest method to select with; `moe` is the cheapest.** Under `vote`, peer-scoring calls grow with the square of the member count, all before the judge's pass. `judge_best` costs one judge pass; `single` and

`synthesize` one call each; `consensus` only embeds and clusters. `moe` generates once however many members exist.

- **`weight` applies to generators only,** never to the judge (the judge is not a config member). In `vote` it weights each voter's ballot, in `consensus` it sizes each answer's contribution to its cluster's support, and in `moe` it is the member's **tier** and the router's tie-break.
- **Ties → random** in `judge_best`, `vote`, and `consensus` (a tie in cluster support). Use a real random tiebreak, not "first wins," so no model gets a positional advantage.
  - **`moe` is the deliberate exception: a routing tie never breaks at random.** A random *router* would make the model set-up itself vary run to run — a confound in the one place the platform cannot afford one. A tie escalates to the judge, and if that call cannot decide, it falls to the highest-`weight` contender (§19).
- **`single` gives you one answer graded by the same trusted judge** as every other method, so its score is directly comparable — the only thing it skips is the multi-candidate contest. See §2.
- **`single` and `moe` both produce one candidate, and they are not the same thing.** `single` has one member and no choice to make; `moe` has several and makes a recorded, per-round choice. That is what makes `moe` a research variable: vary only the config and the platform measures routing against merging, voting and judging on one rubric.
- **Member-count rules.** `single` requires exactly one member; `synthesize`, `vote`, `consensus`, `judge_best`, and `moe` all require at least two. Both rules are enforced in the application and the authoring UI, not by a database constraint, because they are counts over child rows.

Tunables (`.env`, with hard-coded defaults): `CONSENSUS_SIMILARITY` (**0.90**) — the cosine similarity at or above which two `consensus` candidates join the same cluster.

Schema touchpoints. Reads `model_config.combine_method`, `model_config_model` (`weight`), and `experiment.score_judge_*` / `score_weights`; `consensus` additionally embeds the candidate answers with the active embedding model, and `moe` reads `model_catalog.expertise` (§19). Writes `chat_round` (`response`, `best_model`, `worst_model`, `best_score`, `worst_score`, `inline_score`, and `moe_routing` under `moe`) and one `chat_round_candidate` per generation (`content`, `iteration`, `scores`, `inline_score`, `disposition`, token/latency fields).

---

2. The Self-Improvement Loop (Mixture-of-Agents)

Purpose. Optionally iterate "refine → score → check stop conditions" so a round's answer improves across passes. The scheme is **Mixture-of-Agents (MoA)**: after a pass, **every** generator is shown **all** of that pass's answers and re-answers. The MoA insight is that the diverse peer drafts — not a single collapsed summary — are what lift the next pass, so the whole set is broadcast, not just the current best.

Inputs / outputs.

- Inputs: the combine method and its generators (from §1); the loop knobs on `model_config` — `max_iterations`, `min_score_gain`, `score_target`, `underperform_threshold`; the per-method score source.
- Outputs: `chat_round.num_iterations` and `chat_round.stop_reason`; the winning answer chosen **across all iterations**; per-candidate `disposition`, one of `chosen`, `pruned`, or `none`.

Procedure.

```
iteration       := 0
prev_score      := negative infinity
best_overall    := null            # the winner may come from ANY iteration
prev_candidates := null            # prior pass's answers; null on the first pass
max_passes      := model_config.max_iterations or 1

loop:
    # Every pass forks to all active generators and joins before the next one.
    # prev_candidates is null on pass 0, so the first pass has no peers to show.
    candidates := parallel_generate(active_generators, context, peers = prev_candidates)
    winner, score := combine(candidates)      # the §1 reduce, plus the inline score
    persist candidates(iteration)
    best_overall := winner on the first pass, else whichever of the two scores higher

    if score >= model_config.score_target:
        stop_reason := score_target
        break
    if iteration > 0 and score - prev_score < min_score_gain:
        stop_reason := min_gain
        break
    if iteration + 1 >= max_passes:
        # max_passes = 1 lands here on pass 0, which is the single-pass case.
        stop_reason := max_iterations if max_passes > 1 else single_pass
        break

    prune the generators whose factor score is below underperform_threshold
    prev_candidates := the surviving candidates
    prev_score      := score
    iteration       := iteration + 1

mark best_overall's candidate as the chosen one       # exactly one per round
chat_round.response       := best_overall.text
chat_round.num_iterations := iteration               # 0 means a single pass
chat_round.stop_reason    := stop_reason
```

The **score that drives the loop is always the trusted judge's composite** for that pass's winning answer — one source for every combine method, on one 0–100 scale, so the stop conditions mean the same thing everywhere. The **stop trio** is `score ≥ score_target` OR `gain < min_score_gain` OR `iter ≥ max_iterations`; any one ends the loop.

Key decisions & edge cases.

- **The loop is not monotonic.** A later pass can be worse (models plateau and can even regress with feedback). So the winner is the best answer across *all* iterations, tracked as you go — not blindly the last iteration's. Exactly one candidate ends `chosen` per round.
- **Generators see peers' answers, not scores.** Each refinement pass hands every generator all of the previous pass's answers (anonymized), never the numeric judge scores.

    - Feeding the score back would push models to game the rubric — Goodhart's law, where a measure that becomes a target stops measuring — instead of genuinely improving, and would blur cross-run comparability. The score's only jobs are to drive the stop conditions and to pick the across-iterations winner.

- **Reflexion-style critique is deferred.** Feeding the judge's short verbal per-factor critique back to each generator is a stronger signal than bare peer answers, and is left as a future toggle the platform can measure: it pays off only on the minority of prompts that both iterate and fail in critique-actionable ways, and it costs net-new judge work under `vote` and `synthesize`.
- **Persistence split.** `disposition` is per-**candidate** on `chat_round_candidate`; `stop_reason` and `num_iterations` are per-**round** on `chat_round`. The threshold column is `model_config.min_score_gain`; the recorded enum value is `min_gain` — deliberately different names for the knob versus the reason.
- **Pruning** drops a generator scoring below `underperform_threshold` from *later* iterations (its last candidate is `disposition = pruned`). Null threshold means no pruning.
- **Cost shape.** Passes are sequential, but within each pass every active generator is called in parallel and the pass joins before the next one forks. Worst case is about `generators × iterations` generation calls plus scoring, and per-call tokens grow with the generator count because each refinement carries the prior pass's peer answers. The `score_target` gate is the biggest saver, since most prompts clear it on pass 0.
- **With one generator the loop still runs, it just has no peers.** Under `single`, and under `moe` where the router has already narrowed the round to one expert, each pass shows the generator its own previous answer and nothing else. The stop trio and the across-iterations winner work unchanged.
- **Retention of losers is not this algorithm's job** — it is governed later by the `experiment_run.keep_candidates` boolean (keep the losing candidate answers after the run, or sweep them).

Schema touchpoints. Reads `model_config` (`max_iterations`, `min_score_gain`, `score_target`, `underperform_threshold`, `score_weights`, `combine_method`), `model_config_model`. Writes `chat_round` (`num_iterations`, `stop_reason`, `inline_score`, `best_*` / `worst_*`, `response`) and `chat_round_candidate` (`iteration`, `content`, `scores`, `inline_score`, `disposition`).

---

3. The Single-Variable / Factorial Analytics Report

Purpose. Isolate the effect of one experimental factor. An experiment's runs form a factorial grid over the four swept axes `(nudge, model_config, cohort, socratic_enabled)`. To measure factor *i* cleanly, you hold the other three fixed and compare across *i*'s values. There is no "run set" table — the grouping is done at query time.

Inputs / outputs.

- Inputs: an `experiment_id`; the factor to isolate, *i*, one of `nudge`, `model_config`, or `cohort`; the metric of interest (usually `composite_mean` or `composite_median`).
- Outputs: a comparison table — for each fixed combination of the other three factors, a series of `run_result` metrics across factor *i*'s values, labeled with the definitions' human names.

What this does not do. It reports the difference between conditions, not whether that difference exceeds chance. Significance testing is left to an export — a repeated-measures analysis of variance, or a Friedman test — because the right test depends on the study's design.

Procedure.

```
-- One query does the whole report. `isolate` is the axis being varied; the other three
-- are held, so every row within a held group differs only in `isolate`.
--   isolate  in (nudge_id, model_config_id, cohort_id, socratic_enabled)
--   metric   is usually composite_mean or composite_median
SELECT  <held_1>, <held_2>, <held_3>,        -- the three axes other than `isolate`
        <isolate>,                            -- the axis being compared across
        rr.<metric>
FROM    run_result rr
JOIN    experiment_run er ON er.id = rr.run_id
WHERE   er.experiment_id = :experiment_id     -- never span experiments: a different
                                              -- frame gives incomparable numbers
ORDER BY <held_1>, <held_2>, <held_3>, <isolate>;

-- The rows arrive already grouped by the ORDER BY, so the caller only walks them:
--   each run of equal held values is one series;
--   label each point by its definition's .name, joined by id, which is safe because a
--   definition is immutable while a run references it. socratic_enabled is a boolean,
--   so its two values label as "Socratic off" and "Socratic on".
-- A cell with no run is absent from the result and is rendered as a gap, never as 0.
-- There is no aggregate here: run_result already holds one row per run.
```

Key decisions & edge cases.

- **Mean and median can disagree, and that gap is a finding.** All scores are enrollee-unit (per-enrollee means), so a chatty enrollee never dominates. Report `composite_mean` alongside `composite_median`: when the mean sits well above or below the median, the per-enrollee distribution is skewed, which is itself worth knowing.
- **Medians, percentiles, and standard deviations cannot be re-pooled across runs.** Averaging stored per-run `_median`s is invalid; a cross-run median or percentile must be recomputed from the raw `chat_round` rows. After a Cleanup those rows are gone and only the frozen per-run statistics survive, so a cross-run median is unavailable and the report must say so rather than fake it by pooling.
- **Comparison is valid only because one trusted judge grades every run identically** (same frozen scoring config on the `experiment`). Never compare `run_result`s across *different* experiments — different frames, incomparable numbers.
- **Grid gaps.** A factor value with no run under some holding is a missing cell, rendered as absent, never as zero.

Schema touchpoints. Reads `run_result` (all statistic columns, `n_rounds`, `n_enrollees`, cost totals), `experiment_run` (`experiment_id`, `nudge_id`, `model_config_id`, `cohort_id`), and `nudge.name` / `model_config.name` / `cohort.name`. For re-pooled medians, reads raw `chat_round`.

4. The Per-Factor Scorer

Purpose. Turn one answer into its per-factor scores and a weighted `composite`, all on 0–100. For the built-in default rubric the factors are `groundedness`, `relevance`, `coherence`, `instruction_following`; the rubric (hence the factor set) is configurable per experiment.

Every other scoring step calls it: §1 once per candidate to pick a winner, §2 each pass for the stop condition, and the round's saved analytics scores are simply the winner's result.

Inputs / outputs.

- Inputs: the answer text; the originating enrollee prompt — `raw_prompt` when prompt enhancement rewrote it (§18), so the answer is measured against what the enrollee actually asked, and `prompt` otherwise; the same replayed chat history the generators received (§23), without which a follow-up such as "shorter" cannot be scored for relevance at all; the retrieved RAG context that produced the answer (the exact chunks, §7) — **which may be empty**, since many experiments run with no context files; the experiment's frozen scoring config — `score_judge_provider` / `score_judge_model`, `score_rubric_version`, `score_weights`.
- Outputs: a struct `{groundedness, relevance, coherence, instruction_following, composite}` on 0–100, plus a `declined` verdict and the judge's token/cost usage. `groundedness` is **N/A (null)** when the round had no context to ground against, and `relevance` is **N/A** when the answer appropriately declined to answer at all (see *Declining is not a bad answer*, below), and `composite` is then the weighted mean of whichever factors applied (see *Graceful degradation with no context*, below).

The factors — configurable per experiment. The rubric (§5) decides which factors are scored; an experiment may pick one whose criteria fit its study (say funniness, originality, timeliness for a joke experiment). The built-in **default rubric** has the four below, and `composite` is always derived, not a rated factor:

- **groundedness** — are the answer's claims supported by the retrieved context, with nothing invented or contradicted? The retrieved context is the only source that counts; material a prompt happens to carry is not treated as one, because applicability has to be uniform across the experiment rather than varying by round.
  - Groundedness is undefined only when there is no source at all. Open-domain "grounded" would collapse into factual correctness, which the judge is not asked to guess, so the factor is N/A rather than forced to a number.
- **relevance** — does the answer address what the enrollee actually asked? When enhancement ran, that is the raw prompt, not the rewrite.
- **coherence** — is it internally consistent and well-structured?
- **instruction_following** — did it obey the instructions in the system prompt (format, tone, constraints)? This is the factor on which models differ most.

Provenance of the default factors. These four are standard, reference-free **LLM-as-a-judge** criteria (an LLM scores each answer against the rubric, no gold reference answer needed), drawn from established evaluation work:

- **groundedness** (a.k.a. *faithfulness*) and **relevance** (*answer relevance*) — from RAG evaluation (the "RAG triad" in TruLens; core metrics in RAGAS).
- **coherence** — from summarization evaluation (G-Eval / SummEval).
- **instruction_following** — from instruction-following benchmarks (IFEval).

There is no single industry-standard name for this exact four-factor bundle; it is a curated default, implemented by hand rather than by pulling in any of those libraries.

Procedure.

```
score(answer, prompt, context, cfg) -> { factor_scores, composite }:
  # Any factor may be N/A (null).

  # 1. Build ONE structured judge prompt for the APPLICABLE factors only. The rubric
  #    is versioned and immutable: task framing, a definition per factor, five anchors.
  rubric := load_rubric(cfg.score_rubric_version)
  judge_prompt, factors := render(rubric, prompt, answer, context)
  # The `factors` are the applicable factor keys. groundedness drops out when `context`
  # is empty (there is nothing to ground against), so it is scored only when context is
  # present. The rubric asks for strict JSON: one integer from 0 to 100 per applicable
  # factor, plus a one-line rationale each (kept in the transcript, not parsed as a score).

  # 2. Call the one trusted judge deterministically, re-asking a bounded number of times
  #    if the output is malformed.
  for attempt from 1 to MAX_REASK (which is 3):
      # A temperature of 0 makes the call repeatable.
      raw := call_model(cfg.score_judge_provider, cfg.score_judge_model,
                        judge_prompt, temperature = 0)
      parsed := parse_strict_json(raw)
      if parsed is well-formed and has an integer for every key in `factors`:
          break
      # Re-ask, requesting only JSON with the applicable factor keys, each 0 to 100.
      judge_prompt := judge_prompt + reask_note(factors)
  else:
      # The caller writes chat_round.error and marks the round failed.
      raise ScoringError

  # 3. A response that declines to answer is not a bad answer. The judge returns a
  #    `declined` verdict alongside the scores, since it already holds the system prompt
  #    and can say whether the refusal was warranted by it.
  #      none        — the response attempted the request
  #      appropriate — it declined, and the system prompt or the absent source called for it
  #      unwarranted — it declined with nothing to justify the refusal
  if parsed.declined = 'appropriate':
      factors := factors - relevance      # N/A, exactly as groundedness drops out
  # 'unwarranted' keeps relevance scored, so a model that simply will not answer is
  # still penalized. That asymmetry is what stops the verdict becoming a way to
  # dodge a low score.

  # 4. Clamp each applicable factor to the range 0 to 100. A factor that does not
  #    apply stays NULL (N/A), never 0.
  score := empty map
  for each factor in factors:
      score[factor] := clamp(parsed[factor], 0, 100)
  # A factor the rubric did not apply has no key at all, which is what N/A means here.
```

```
    # 5. Weighted composite over the APPLICABLE factors, with the weights renormalized
    #    so that they sum to 1.
    w := cfg.score_weights (or equal weights if none are set)
    weighted_sum := 0
    weight_total := 0
    for each factor in factors:
        weighted_sum := weighted_sum + w[factor] * score[factor]
        weight_total := weight_total + w[factor]
    composite := weighted_sum / weight_total

    return { factor_scores := score, composite }
```

The two seams — `load_rubric` and `render`. Step 1 hides the two functions worth spelling out; both live in the middle tier (there is no rubric table in the schema — `experiment.score_rubric_version` is a *label*, not the text).

- **`load_rubric(version)` — versioned template lookup.** Rubrics are immutable, versioned assets bundled with the middle-tier release (`rubrics/<version>.yaml`), loaded once at boot into an in-memory registry keyed by version string.
  - The registry holds every bundled rubric, and **`list_rubrics()`** enumerates it (version, display name, factor list) to fill the composer's drop-down, so the experimenter picks a version rather than typing one.
  - `load_rubric` returns the rubric for `version` or fails fast with `UnknownRubricVersion`. It never substitutes a different rubric, since a silent swap would break cross-run comparability.
  - A rubric is structured rather than a blob: a task-framing header plus factors, each `{key, name, definition, five anchors, requires_context}` . A published version's content never changes — new wording means a new version — so it is safe to cache and `score_rubric_version` is a real provenance label.
- **`render(rubric, prompt, answer, context) -> (judge_prompt, factors)` — applicability, then deterministic fill.**
  - **Which factors apply.** Drop any factor whose `requires_context` is set when no grounding source is available. Today only groundedness sets it, so with no context `factors` is `{relevance, coherence, instruction_following}` .
  - **Assemble the prompt in fixed order, with no randomness:** task framing, the prompt and answer, the context block when present or an explicit *"No source material was provided; do not assess factual grounding"* line when absent, the definition and five anchors for each applicable factor, then the output contract naming exactly those keys.
  - Returning `factors` alongside the string lets the later steps require the right keys and renormalize over them. The same inputs render the same prompt, so a past round re-renders identically. **Key decisions & edge cases.**
- **One call, all applicable factors.** Every applicable score comes from a single judge response, so they are internally consistent and the call cost is one grading, not one per factor. The composite is computed in code, never asked of the model.
- **Declining is not a bad answer.** A refusal scores badly on `relevance` and well on `instruction_following` , so a model that correctly refuses would otherwise rank below one that answered anyway. The judge returns a `declined` verdict alongside the scores: `appropriate` takes `relevance` to N/A, `unwarranted` keeps it scored.
- **No round is ever a question.** The models under test never ask for a missing detail — eliciting one is §21's work, and it happens before a round exists. So every response this scorer sees is an attempt or a refusal, and the verdict needs no third value.

- **The verdict is recorded, not just applied.** `chat_round.declined` keeps it, so a run can report how often it refused and how often that was warranted. §6 rolls the verdicts into `run_result.declined_dist`, which outlives a Cleanup of the rounds.
- **Graceful degradation with no context.** Context files attach to the `experiment`, so one with none never grounds. `render` then drops groundedness and it is **absent from `chat_round.factor_scores`** rather than written as 0, which would be a real and damning score. The composite is the weighted mean of the factors that do apply, their weights renormalized to sum to 1.
    - Because context presence is experiment-level, every round scores the same factor set and composites stay comparable. The one exception is a context-using experiment whose retrieval returns nothing for a single query, which drops groundedness for that round alone.
- **Grounding is inferred from context-file presence, with no override.** At compose time an experiment with no context files warns the experimenter that groundedness will not be scored, so the omission is deliberate rather than silent.
- **The rationale is never a score.** The rubric asks for a short per-factor rationale to make the transcript auditable; the parser reads only the integers. A model that returns prose instead of a parseable integer triggers the re-ask, then a hard failure — it never silently scores 0.
- **Determinism.** `temperature = 0` and a frozen `score_rubric_version` make the same answer score the same way on re-run — the property that lets a past round be re-scored and reproduced.
- **Clamp, don't trust.** A judge can emit 105 or -3; clamp to `[0, 100]` before the composite so one stray value cannot skew the weighted mean.
- **Weights live on the `experiment`,** so every factor-scoring call in the run uses the same weights — the composite means the same thing in selection (§1), in the loop (§2), and in the saved analytics score.

Schema touchpoints. Reads `experiment` (`score_judge_provider`, `score_judge_model`, `score_rubric_version`, `score_weights`) and the round's RAG context via `experiment_context_file` → `document`. Its outputs are written by the callers onto `chat_round_candidate.scores` / `inline_score` and, for the winner, `chat_round.factor_scores` / `score_composite` (see §5).

---

5. The Scoring Pass — One Trusted Judge

Purpose. Tie the per-factor scorer (§4) to the round lifecycle. **One** judge, frozen on the `experiment`, scores the round's answer(s) on the rubric's applicable factors — groundedness, when the rubric includes it, drops out on a no-context round — and the winning answer's scores become the round's durable analytics scores.

Because the same judge and rubric apply to every round of every run, all runs of that experiment are comparable within it, with **no** separate second scoring pass.

Inputs / outputs.

- Inputs, per round: the candidate answer(s), the originating enrollee prompt (`raw_prompt` when enhancement ran, `prompt` otherwise — see §18), and the retrieved RAG context; the experiment's scoring config — `score_judge_provider` / `score_judge_model`, `score_rubric_version`, `score_weights`.
- Outputs: `chat_round.factor_scores` (a jsonb bag keyed by the rubric's factors, each 0–100), `score_composite`, `inline_score`, and the judge's `judge_tokens` / `judge_cost`.

Procedure.

```
# Called once per round as it settles. During the combine step (§1) the ONE trusted
# judge has already scored each candidate via §4; here we promote the winner's scores
# to the round and freeze them.
scoring_pass(round):
    # 1. The winner is the chosen candidate (§1): the highest composite, or for
    #    `single` the one answer. Each candidate keeps its OWN scores in
    #    chat_round_candidate.scores / inline_score.
    winner := round.chosen_candidate

    # 2. Promote the winner's scores to the round's DURABLE analytics fields.
    chat_round.factor_scores   := winner.factor_scores   # jsonb, factors 0 to 100
    chat_round.score_composite := winner.composite
    chat_round.inline_score    := winner.composite       # the same number, from one judge

    # 3. If a factor call was still malformed after §4's bounded re-ask, FAIL the round
    #    rather than record a guess.
    if the winner's scoring failed:
        chat_round.error := the scoring error   # jsonb, set only on failure
        mark the round failed

# The judge's spend is captured (judge_tokens / judge_cost) and reported apart from
# generator cost (§11), never hidden.
```

Key decisions & edge cases.

- **The judge's cost is reported separately.** Per-round spend lands in `judge_tokens` / `judge_cost` and rolls to `run_result.total_judge_cost`, kept apart from `total_generator_cost` so a report can isolate the variable under test (§11). Under `moe` the routing call (§19) is judge spend too, which keeps `total_generator_cost` a clean measure of the generation being compared.
- **Self-preference guard — a warning, not a block.** Because `score_judge` is a real catalog foreign key, the app checks at launch whether the judge shares a provider with, or is the same model as, a generator under comparison, and warns rather than blocking. Sometimes that overlap is exactly what a study means to observe.
- **The rubric fixes the factor set; the composite is always derived.** `composite` is not a rated dimension (it appears as a row only in statistics). A factor that does not apply to a round — groundedness with no source (§4) — drops out, and the composite renormalizes so it stays on 0–100.
- **One judge is the entire point:** same judge, same weights, same rubric, on every round of every run. Because the scoring config lives immutably on the `experiment`, no per-run snapshot is needed to preserve it — re-reading the experiment reproduces it.

Schema touchpoints. Reads `experiment` (`score_judge_provider`, `score_judge_model`, `score_rubric_version`, `score_weights`) and the round's RAG context via `experiment_context_file` → `document`. Writes `chat_round` (`factor_scores`, `score_composite`, `inline_score`, `judge_tokens`, `judge_cost`, and `error` on failure).

6. `run_result` Aggregation

Purpose. Compute the durable per-run scorecard exactly once, when the run finishes, and freeze it.

Inputs / outputs.

- Inputs: every `chat_round` of the run (with `enrollee_id` for the enrollee-weighting), and their `chat_round_candidate` cost detail.
- Outputs: one `run_result` row.

Procedure. On `experiment_run.state` reaching `done`:

1. Gather the run's `chat_round` rows.

2. Score statistics — the enrollee is the unit. Everything is computed over **per-enrollee means** (average within each enrollee, then summarize across them), so a chatty enrollee cannot dominate:

```
-- 2a/2b. Two levels: average WITHIN each enrollee, then summarize ACROSS them. A NULL
-- factor is skipped by avg(), never read as 0, which is why this is not one flat mean.
WITH per_enrollee AS (
    SELECT  enrollee_id,
            avg(score_composite)                      AS composite,
            avg((factor_scores->>'groundedness')::numeric) AS groundedness,
            avg((factor_scores->>'relevance')::numeric)    AS relevance
            -- ... one avg() per factor in the experiment's rubric
    FROM    chat_round
    WHERE   run_id = :run_id          -- rewound rounds INCLUDED: they keep their scores (§23);
                                      -- only the replayed history excludes them
    GROUP BY enrollee_id
)
SELECT  avg(composite)                                    AS composite_mean,
        percentile_cont(0.5) WITHIN GROUP (ORDER BY composite) AS composite_median
FROM    per_enrollee;

-- 2c. factor_stats (jsonb): a five-number summary per dimension — each rubric factor
-- and the composite — computed over the SAME per_enrollee rows. There is no standard
-- deviation; q3 - q1 is the distribution-free measure of spread.
SELECT  min(x), percentile_cont(0.25) WITHIN GROUP (ORDER BY x),
        percentile_cont(0.5)  WITHIN GROUP (ORDER BY x),
        percentile_cont(0.75) WITHIN GROUP (ORDER BY x), max(x)
FROM    per_enrollee, LATERAL (SELECT <dimension> AS x) v
WHERE   x IS NOT NULL;
-- A dimension with no non-null value gets no key at all: groundedness on a run with no
-- source simply does not appear, rather than appearing as zero.
```

3. Operational grid — 12 columns (three metrics, four statistics each). For each of `iterations` (`num_iterations`), `latency_ms`, and `tokens`, compute `_mean`, `_median`, `_min`, and `_max`.

4. **`stop_reason_dist`** (jsonb) — tally the round-level `stop_reason` values; this shows whether the loop was converging or just hitting its iteration ceiling.

4b. `declined_dist` (jsonb) — tally the round-level `declined` verdicts into `appropriate`, `unwarranted`, and `attempted` (the rounds that did not refuse); the three sum to `n_rounds`. This is what lets a finished run report its refusal rate, and how much of it was warranted, after the rounds themselves are gone. Requests the §21 gate turned back are **not** counted — they never became rounds.

2d. `final_answer_stats` (jsonb) — the same summary as step 2, computed over each session's **last round** rather than every round: the highest `round_seq` that was not rewound. A session whose rounds were all rewound contributes nothing, and the column is null when no session contributed one. It says where a session ended up, which a long session can drift away from; read it beside the all-rounds figure, never instead of it.

4d. `retry_stats` (jsonb) — what §21's gate cost this run in enrollee effort, and null when no gate could fire. 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:

```
{"rounds_with_retries": 41, "retries_mean": 0.7, "retries_max": 4,
 "reasons": {"missing_slots": 38, "out_of_scope": 9}}
```

This is the outcome measure of the Socratic switch (§30). Both settings turn attempts back and both count them here, so the two arms are directly comparable on how many attempts an answer took — which is the question the switch exists to ask, since answer quality is already in `composite_mean`.

4c. `history_stats` (jsonb) — null when `enable_chat_history` is off; otherwise `context_window_min` (the smallest window among the configuration's members, which bounded every round), the truncation and alert counts, the rewind count, and the mean and maximum turns replayed. `model_config` is swept, so one arm can have systematically shallower memory than another; these figures make that visible. See §23.

5. `model_perf` (jsonb) — per constituent model, its wins/losses/`avg_score` aggregated from `best_model` / `worst_model` across the run; the "dud model" signal. Under `moe` a "win" is a round that model was **routed to**, so this same column doubles as the run's routing distribution — and, because `run_result` outlives a cleanup, that distribution survives when the raw rounds do not.

6. Costs and counts: `total_generator_cost` is the sum of `chat_round.token_cost` across the run; `total_judge_cost` is the sum of `chat_round.judge_cost` (the one trusted judge's grading spend); then `total_tokens`, `total_latency_ms` (a cumulative sum), `n_rounds`, `n_enrollees`, and `computed_at`.

7. Freeze the row.

Key decisions & edge cases.

- **Enrollee-unit means are two-level, not plain means.** Every statistic starts from per-enrollee means (average within each enrollee, then summarize across enrollees). Computing a plain mean over raw rounds instead is a silent, plausible-looking corruption of every comparison — the highest-consequence subtlety in the system.
- **Medians and quantiles are recomputed from raw rounds,** never pooled from other runs' stored statistics.
- `run_result` **is durable and survives a Cleanup** of the raw rounds. The aggregation therefore runs *at run end* and must never assume its inputs still exist afterward — everything cross-run must be readable from the frozen row alone.
- `total_latency_ms` **is a cumulative sum** across rounds — a different figure from each round's parallel-aware wall-clock `latency_ms` (see §11) and from the run's calendar duration.
- **The judge's cost is** `total_judge_cost`**,** reported separately from `total_generator_cost`; the two sum to the run's total spend.

Schema touchpoints. Reads `chat_round` (`factor_scores` , `score_composite` , `num_iterations` , `latency_ms` , `tokens` , `token_cost` , `judge_cost` , `stop_reason` , `declined` , `history_window` , `rewound_at` , `retry_count` , `clarification` , `best_model` , `worst_model` , `enrollee_id`). Writes `run_result` (`composite_mean` , `composite_median` , `factor_stats` , `final_answer_stats` , the operational stats, `stop_reason_dist` , `declined_dist` , `retry_stats` , `history_stats` , `model_perf` , `total_*` , `n_rounds` , `n_enrollees` , `computed_at`).

---

7. Retrieval-Augmented Generation (RAG)

Purpose. Supply relevant source text so answers are grounded in real material. Two kinds of source are indexed: documents, and rows of tables named in `embedding_ingest_registry` — a `profile.notes` field is as retrievable as a PDF. Two consumers exist: enrollee chat, scoped to the experiment's context documents, and the Ask engine (§8), scoped to reference documents **and** to the registered table rows, which it reads under the asker's own row-level security.

Inputs / outputs.

- Inputs: the query text; the scope (which documents are eligible); the active `embedding_model` ; the mandatory cosine floor `RAG_MIN_SIMILARITY` (default **0.25**); a top-K and an optional minimum-similarity threshold.
- Outputs: the top-K chunk texts, assembled into the generation prompt.

Procedure.

Ingest — per source (a document, or a registered row), run when the source changes (see §9's queue):

```
# TWO kinds of source are ingested. They differ in how the text is rendered and how the
# scope key is derived; everything after that is identical.
ingest(source):

    if source is a document:
        text   := extract(source)              # a scanned document is OCR'd first
        policy := fixed(1000, 150)             # long-form prose has to be cut
        scope  := { source_table: 'document', document_id: source.id,
                    purpose: source.purpose,   # context | reference
                    experiment_id: the experiment it is attached to, when purpose = context }
    else:                                      # one row of a registered table
        text   := render(source, registry.text_template)   # e.g. "{{name}}\n{{notes}}"
        policy := none                         # the template already yields a sentence or two
        scope  := source[registry.filter_columns]

    policy := registry.chunking_policy or policy           # an explicit setting wins

    for each chunk, chunk_index in split(text, policy):
        write embedding(chunk_index, chunk_text = chunk, vector = embed(chunk, model),
                        embedding_model = model, chunking_used = policy,
                        filter_values = scope)
```

Retrieval:

```
# For enrollee chat (the experiment's context docs) or the Ask engine (§8, reference).
# min_sim always has a value: RAG_MIN_SIMILARITY is hard-coded to 0.25 and only
# overridden by .env, so there is no "no floor" case to guard against. It matters that
# it is always applied, because §21's gate depends on retrieval being able to return
# nothing.
retrieve(query, scope, top_k, min_sim = RAG_MIN_SIMILARITY):
    # 1. Embed the query with the SAME embedding_model as the index; vectors from
    #    different models are not comparable.
    qvec := embed(query, active_embedding_model)

    # 2. Find the approximate nearest neighbors of embedding.vector using an HNSW
    #    index (Hierarchical Navigable Small World), by cosine distance (vector <=> qvec).
    # 3. Scope by jsonb containment, using a GIN index (filter_values @> scope):
    #      enrollee RAG uses the EXPERIMENT's docs where purpose = 'context';
    #      the Ask engine uses docs where purpose = 'reference', pulled by meaning.
    hits := nearest(embedding.vector, qvec)
            where filter_values @> scope
            and cosine_similarity >= min_sim

    # 4. Take the top few hits and paste their chunk_text into the generation prompt.
    return assemble_prompt(top_k_of(hits, top_k))
```

Key decisions & edge cases.

- **Enrollee RAG scopes to the experiment frame, not the run.** All runs of an experiment share the same immutable context corpus, which is part of what makes them comparable.
- **The `purpose` scoping (`context` vs `reference`) is application-enforced,** not a database check — so a reference document never leaks into enrollee RAG context, and vice versa.
- **Bulk-rebuild guard.** With a single `vector(1024)` slot, a model switch overwrites vectors in place. While a source's bulk (re)embed is in progress, similarity queries that reference that source are **rejected** (mid-switch the source holds a non-comparable mix of old and new vectors). Other sources stay queryable. See §9.
- **Chunking has a default per source kind, and the split respects text boundaries.** `chunking_policy` on the registry entry decides how a source is cut up; when null, the default for that kind applies — **`fixed(1000, 150)`** for a document, **`none`** for a registered table row.
    - **Why those two differ.** A document is long-form prose that must be cut, and 1000 characters is about 250 tokens: small enough that a chunk describes one thing, large enough to keep a supporting sentence in context. The 150-character overlap keeps a sentence straddling a boundary retrievable from either side. A registered table row is already a sentence or two, so one row is one chunk.
- **Chunking policy.** Either `none` (the whole rendered text, `chunk_index = 0`) or `fixed(size, overlap)` — a sliding window that backs off to a paragraph or sentence boundary, so an idea is not cut mid-thought. The overlap is carried as whole sentences rather than a raw character count, which is where the mainstream splitters have converged.
- **Top-K and the minimum-similarity threshold have defaults, overridable via `.env` .** Top-K defaults to **5** (`RAG_TOP_K`). The cosine floor, `RAG_MIN_SIMILARITY` , defaults to **0.25** and is always in force — it is not optional, because §21 relies on retrieval returning nothing to turn an unanswerable request back.

  - **The floor is a hallucination guard.** `RAG_MIN_SIMILARITY` stops retrieval handing the model confident-looking context of little or no relevance. It is set low on purpose: it detects that the corpus has no opinion rather than ranking chunks that are already relevant.

- **The value is calibrated to the embedding model.** Cosine scales differ between models, so 0.25 suits the reference model `openai/text-embedding-3-large`. Re-tune it whenever the model changes, as part of the rebuild §9 already attaches to that event.
- **An empty result is handed to §21, not to the model.** Retrieval reports that nothing cleared the floor, and the gate turns the request back with a scoped reply rather than answering it.
- **Chunking degrades gracefully:** `fixed(size)` over text shorter than `size` yields a single chunk, the same as `none`.

Tunables (`.env`, with hard-coded defaults): `RAG_TOP_K` (**5**) — how many chunks are pasted into the prompt; `RAG_MIN_SIMILARITY` (**0.25**) — the mandatory cosine floor, below which retrieval returns nothing; `CHUNK_TARGET_TOKENS` (**800**) — the chunk size the splitter aims at.

Schema touchpoints. Reads `document`, `embedding` (`vector`, `filter_values`, `embedding_model`, `chunk_index`, `chunk_text`), `embedding_ingest_registry` (`text_template`, `chunking_policy`, `filter_columns`), and `experiment_context_file` → `document`.

---

8. The NL→SQL Ask Engine — Question Reuse, SQL Caching, and Execution

Purpose. Answer a question typed in plain language, always executing under the asker's own row-level security so an answer can never reveal a row the asker could not see on a screen.

Three paths converge here. Only the third uses a model to **write SQL**; every path uses one to write the narrative that accompanies the tables, since that prose depends on the values returned:

- **(A) A saved button.** The row already holds its SQL: bind the values and run.
- **(B) A typed question that matches a saved one.** Embed it, find the nearest saved question, reuse its SQL.
- **(C) A typed question that matches nothing.** An agent answers it, with the database as a tool.

Inputs / outputs.

- Inputs: the typed question, or a button click; the asker's identity and visibility; the screen's `scope_key`; for path C, the schema context pack and the install's `ASK_MODEL`.
- Outputs: a short narrative plus 0..N result tables, each rendered as a table, bar, line, or pie chart. Zero tables is a valid answer — see the abstention rule below.

Procedure — paths A and B, which cost no model call.

```
# (A) A saved-question BUTTON skips embedding and the agent entirely.
if button_click:
    row := the saved nl_query
    bind row.params values into row.derived_sqls        # named placeholders, %(name)s
    validate that every statement is read-only
    tables := execute under the caller's RLS, capped at ASK_ROW_CAP
    render tables, then narrate(resolved_question, tables)   # the narrative needs a model
    return
```

```
# (B) A typed question, matched against what is already saved.
qvec  := embed(question, active_embedding_model)
match := the HNSW nearest neighbor on nl_query.query_vector
         AMONG the rows the asker may see: their own drafts, drafts shared with them,
         and the approved and system rows WHERE retired_at IS NULL

if match.similarity >= ASK_MATCH_THRESHOLD:
    if match.schema_fingerprint is stale:
        # The database structure THIS QUESTION READS has moved — only the tables and
        # columns its own statements name are covered, and only changes that can alter
        # what a SELECT returns count (see "What counts as a change", below).
        # Regenerate, then flag for re-approval; do NOT run it silently.
        # model_version is NOT a trigger: an upgraded compiler leaves working SQL working.
        regenerate match.derived_sqls from match.canonical_prompt
    slot_fill any params the question supplies; ask for the rest on a small form
    validate; tables := execute under the caller's RLS, capped at ASK_ROW_CAP
    render tables, then narrate(resolved_question, tables)   # the narrative needs a model
    return
```

Procedure — path C, the agent. A miss is answered by a tool-using agent rather than by a one-shot compiler, because a real analytical question is often answered in more than one query, and because a follow-up ("now split that by cohort") is the normal way people ask.

```
# The system prompt carries the SCHEMA CONTEXT PACK: the tables, the preferred join
# paths, and the GLOBAL RULES that keep an answer from being plausibly wrong — that a
# mean of stored means is invalid, that run cost is generator plus judge, that a
# cross-run median must be recomputed from raw rounds. This prompt, not the shape of
# the loop, is what makes an answer trustworthy.
tools := [ query_database ]                      # a single SELECT or WITH statement
       + [ join_path ]                           # how do tables A and B connect? (FK graph walk)
       + [ one tool per saved question the asker may see ]

loop:
    reply := call(ASK_MODEL, conversation, tools)
    if reply asks for no tool:
        break                                    # it has its answer, or it abstains
    for each tool call in reply:
        if it names a saved question:
            result := bind and run that row's derived_sqls
        else:
            result := query_database(sql)
        append result to the conversation
narrate(question, tables)      # every path ends here, including A and B

narrate(question, tables):
    # The question is the CANONICAL prompt with its parameters bound, so the narrative
    # names the values actually asked about rather than the placeholders.
    resolved := substitute(nl_query.canonical_prompt, bound params)  # or the typed question
```

```
    # Small model, low temperature, and the rows themselves — never the SQL, which would
    # invite the model to describe the query instead of the answer.
    return call(ASK_FAST_MODEL, narrative_prompt(resolved, tables), temperature = 0)


join_path(table_a, table_b):
    # A deterministic walk of the 40-edge foreign-key graph, no model call. Returns the
    # shortest join path, with preferred edges weighted lower so the DENORM shortcuts win
    # (chat_round.run_id beats the two-hop route through run_enrollment). Answers the
    # ambiguous-join trap by construction rather than from the model's recall.
    # Polymorphic pseudo-edges (embedding.source_*, audit_log.target_*) are NOT in the
    # graph: those joins need the source_table discriminator and the pack's rules.


query_database(sql):
    # Three guards, in order. Each is cheap, and each refuses rather than repairs.
    if sql does not begin with SELECT or WITH:  return REFUSED    # read-only at the app layer
    if EXPLAIN sql fails:                       return REFUSED(the planner's message)
    rows := execute under the caller's JWT/RLS, capped at ASK_ROW_CAP + 1
    # Asking for one row beyond the cap is how truncation is detected at all.
    if count(rows) > ASK_ROW_CAP:
        return first ASK_ROW_CAP rows, marked truncated   # the answer says so; see below
    return rows
```

Saving. Nothing is cached automatically. If the asker chooses to save, the statements the agent actually ran are harvested into `derived_sqls`, the literals it bound are offered as parameters, and the question is canonicalized (§20) and embedded. The row therefore stores three things, not one: `derived_sqls` (the statements), `canonical_prompt` (the normalized wording, which is what a later question is matched against and what a stale statement is regenerated from), and `query_vector` (its embedding). The row lands as `status = draft`; an admin's approval promotes it, and an approved row becomes both a one-click button and an exemplar for later questions.

Lifecycle after saving. Every step below is an admin action on the Saved-questions screen, writes one `audit_log` row, and is guarded by the row's `version`.

```
approve(id, note):    status := approved;  append the stamped note to notes
unapprove(id):        status := draft      # back to its owner; off everyone else's Ask at next load
retire(id):           retired_at := now()  # approved rows only (CHECK); hidden from Ask and from matching
restore(id):          retired_at := NULL
delete(id):           permanent. A draft: its owner or an admin. An approved row, retired or
                      not: an admin. Never a system row. Retire is the undoable alternative.
```

Key decisions & edge cases.

- **The context pack does the safety work.** The system prompt carries the schema context pack — the tables, the preferred join paths, and the global rules that keep an answer from being plausibly wrong. The loop's guards catch unsafe SQL; only the pack catches SQL that is safe and wrong.
- **The FULL pack goes in the system prompt, not a per-question slice** (`ASK_PACK_MODE`, default `full`; decided 2026-09-18, see `DECISIONS.md`). Three reasons, all consequences of the agent loop.

- **The prompt is static, so the pack is cached.** The same bytes every session is a prompt-cache hit. A per-question slice changes each time and is a fresh write, so it costs more than the larger static pack it was meant to save.
- **A follow-up pivots the question.** "Now split that by cohort" is normal, and the loop does not re-run table selection. A slice chosen for the opening question leaves the agent unable to join a table it was never shown.
- **At 24 tables the pack is small.** ~11.2k tokens for everything. Pruning buys distraction relief, not headroom.
- **`ASK_PACK_MODE=slice` switches** to the static core plus a selected slice (`nl-sql-mapping-design.md` §5.2). Build the selector either way: it generates the PREFERRED JOIN PATHS section and backs the `join_path` tool.
- **Refuse, don't repair, inside the tool.** The read-only check and the `EXPLAIN` both reject and hand the reason back to the agent, which then rewrites its own query. That is the repair loop, and it costs no extra machinery.
- **Parameter binding is by named placeholder** (`%(name)s`, psycopg style). Assert that every placeholder in the SQL has a matching `params` entry and vice versa; a mismatch is a generation bug, not a runtime error.
- **Execute under the caller's JWT/RLS, never a service-role key.** This is the property that makes "you can never see rows you couldn't see on a screen" true, and it is why the agent needs no permission logic of its own.
- **A truncated result says so.** `ASK_ROW_CAP` bounds any single query, and a result that hits it is returned marked truncated. The answer states plainly that it shows the first `ASK_ROW_CAP` rows of a larger set and asks the reader to narrow the question — a silently clipped table is a wrong answer wearing the shape of a right one.
- **Abstention is a designed outcome.** A question about data the schema does not hold is declined plainly. `derived_sqls` defaults to `[]` precisely so a saved question can be narrative-only.
- **The match set is visibility-filtered** — only rows the asker may see: their own drafts, drafts shared with them, and the `approved` and `system` rows whose `retired_at` is null. Both correctness and security; a retired question is invisible to matching as well as to the buttons.
- **Staleness invalidation is `schema_fingerprint` alone.** A stale hit is regenerated from `canonical_prompt` and flagged for re-approval rather than run silently.
- **The fingerprint covers only the tables a question actually reads.** Hashing the whole schema would let any migration anywhere invalidate every saved question at once. The statements in `derived_sqls` are already parsed for the read-only check, so the objects they name are in hand and the fingerprint covers just those. A question about run costs is then untouched by a migration to the issue tracker.
- **What counts as a change.** Only what can make stored SQL wrong rather than merely slow: type, nullability, enum values, unique constraints including partial ones, and foreign keys. Plain indexes, defaults, comments and row-level-security policies are excluded, since none changes what a `SELECT` returns — and counting indexes would let one performance-tuning migration invalidate the entire saved library.
- **The model is an install setting.** `ASK_MODEL` answers questions and `ASK_FAST_MODEL` does the small jobs (slot-filling, a result caption). Both live in `.env`, like the embedding model, and neither is a `model_catalog` row, because that catalog is the menu of models *under study*. An experiment's judge is never used here.
- **Thresholds and caps are `.env` constants.** `ASK_MATCH_THRESHOLD` defaults to **0.85** cosine: a saved question is reused only on a strong match, because running the wrong query is worse than asking the agent. `ASK_ROW_CAP` defaults to **100** rows per statement.

Tunables (`.env`, with hard-coded defaults): `ASK_MODEL` — the model that answers; `ASK_FAST_MODEL` (falls back to `ASK_MODEL`) — the small jobs, slot-filling and the narrative; `ASK_MATCH_THRESHOLD` (**0.85**) — the cosine similarity at which a typed question reuses a saved one; `ASK_ROW_CAP` (**100**) — the row ceiling on any single query; `ASK_PACK_MODE` (**full**) — `full` sends the whole schema context pack in the system prompt, `slice` sends the static core plus a per-question slice.

Schema touchpoints. Reads/writes `nl_query` (`canonical_prompt`, `query_vector`, `derived_sqls`, `params`, `status`, `retired_at`, `notes`, `schema_fingerprint`, `model_version`, `scope_key`, `owner_id`, `is_shared`, `hit_count`, `last_used_at`, `version`); writes `audit_log` for each lifecycle step. Reads `embedding` for the reference-document narrative pull.

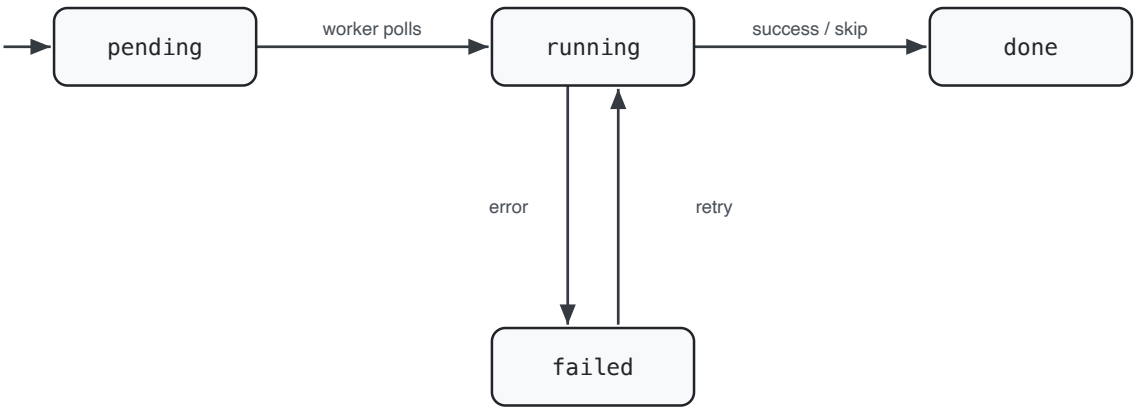
9. Embedding Reconciliation (The Job Queue)

Purpose. Keep the embedding index in sync with source rows through a transient work queue, without scanning a whole table on an ordinary edit. The queue's "already enqueued?" guard is a database partial-unique index rather than an in-memory check.

Inputs / outputs.

- Inputs: source-row inserts/updates/deletes; the `embedding_job` queue; the active embedding model.
- Outputs: fresh `embedding` rows; job status transitions through { `pending` , `running` , `done` , `failed` }.

Job status. An `embedding_job` moves through this state machine:



Procedure.

```
# Enqueue side — a database trigger fires on a local source row.
on insert or update:
    # Only a source the admin has registered and left active produces work. The job
    # table's foreign key points at the registry, so an unregistered table would not
    # merely enqueue uselessly — the insert would fail and take the caller's write
    # with it. An inactive source is skipped for the plainer reason that the admin
    # switched it off.
    if this table has no active embedding_ingest_registry row:
        do nothing
    # On update, compare the TEMPLATED text rather than the row. The trigger fires on
    # any column, so without this a counter bumped on nl_query or a run's state
    # transition would queue a re-embedding of text nobody touched. This is the same
    # comparison the worker's skip guard makes, applied one step earlier.
    if this is an update and render(new, template) = render(old, template):
        do nothing
    enqueue embedding_job(source_id, status = pending)
    # A partial-unique index keeps this unique: at most one OPEN job per
    # (source_schema, source_table, source_id) whose status is pending or running.
```

```
on delete:
    clear that source's embedding rows

# Worker side — drains the queue.
loop:
    job := poll the oldest pending embedding_job     # take the oldest first (FIFO)
    if job is null:
        wait briefly, then continue
    job.status := running

    # Skip guard. Three things decide the stored chunks: the source row, the template
    # that renders it, and the policy that splits it. A change to ANY of them means the
    # text or its chunking differs from what is stored, so all three are checked.
    # embedding rows are replaced wholesale rather than updated in place, so created_at
    # IS the write time; the OLDEST of the set is used, because a partially failed
    # earlier run can leave a newer row beside stale ones.
    stored := that source's embedding rows
    if source.updated_at   <= min(stored.created_at)
       and stored.embedding_model = active_model
       and stored.chunking_used   = resolved_policy(source, registry):
        job.status := done          # text, model and chunking all still current
        continue

    try:
        chunks := split(render(source_row, registry.text_template),
                        registry.chunking_policy)
        for each chunk in chunks:
            write embedding(chunk_text = chunk, vector = embed(chunk, active_model),
                            embedding_model = active_model, filter_values)
        replace the source's old embedding rows with the new set
        job.status := done
    on error:
        job.attempts := job.attempts + 1
        job.error := the error message
        job.status := failed        # visible; retried with backoff

# Model switch: re-embed rows whose embedding_model differs from active_model. An
# install-wide switch is ONE table-wide job (source_id = NULL) with a progress cursor;
# while it runs, similarity queries on that source are rejected (see §7).
```

Key decisions & edge cases.

- **The one-open-job dedup and the one-table-wide-job constraint are database-enforced** partial-unique indexes; NULL `source_id` is distinct in a unique index, so the two constraints coexist.
- **Throughput comes from worker-side batching at the chunk level** (embedding APIs take ~96–2048 texts per call), independent of the per-row queue unit — so per-row jobs are not a throughput bottleneck.
- **The skip guard hinges on comparing timestamps against the *templated* text,** not the whole row. **Schema touchpoints.** Reads `embedding_ingest_registry` (`text_template` , `chunking_policy` , `filter_columns` ,

`is_active`) and the registered source rows. Writes `embedding` (`chunk_text`, `vector`, `embedding_model`, `chunking_used`, `filter_values`) and `embedding_job` (`status`, `attempts`, `error`).

---

10. The 2-Layer Prompt Cascade

Purpose. Resolve the one system prompt a given model receives for a round. Two layers combine, in a fixed order: the model's own standing instruction, then the experiment's.

Inputs / outputs.

- Inputs: `model_config_model.system_prompt` (the model's role inside the configuration, null when it has none); `experiment.system_prompt` (the study's standing instruction).
- Outputs: the resolved system prompt for that model, held in memory for the call. Nothing is written.

Procedure.

```
# Layer 1 — the model's own role, when it has one. Null means it has none.
final := model_config_model.system_prompt or ""

# Layer 2 — the study's standing instruction, appended AFTER the model's own, so the
# experiment always has the last word. It is the invariant frame; a model_config is a
# swept variable and must never be able to displace it.
final := join(final, experiment.system_prompt)

# `final` is the whole system prompt, and those two layers are all of it. Everything the
# round carries — the retrieved context (§7), the replayed history (§23), and the
# enrollee's effective prompt (§18) — follows it, in that order, as the user-side turns.
send system = final, then context, history, and the effective prompt, to that model

join(a, b):
    return b if a is empty else a + "\n" + b     # never emit a stray separator
```

Key decisions & edge cases.

- **Two layers, and the experiment's comes last.** A `model_config` is one of the four swept variables while the `experiment` is the invariant frame of the study, so the study's instruction is appended after the model's and is never displaced by it. There is no per-run layer and no mode to choose: a model either carries a role of its own or it does not.
- **No snapshot is stored.** The resolved prompt is transient. Both layers are immutable while a run references them, so re-resolving reproduces exactly what a past round received.
- **A null model prompt reduces to the experiment's alone,** which is the ordinary case: most members carry no role of their own.
- **The nudge is not part of this cascade — or of the model's prompt at all.** The run's `nudge` is enrollee-facing: a steering message shown beside the enrollee's prompt input to guide how they phrase their prompts. It never reaches the model.
- **Neither is the Socratic switch.** `experiment_run.socratic_enabled` (§30) selects how the trusted judge words a reply when §21's gate stops a message. It is read by the judge, before any model is called, and nothing is appended here on account of it. So the cascade is two layers on every run of every experiment.

- **The separator is a newline** — an implementation choice, kept consistent because it affects the exact bytes the model sees.

Schema touchpoints. Reads `model_config_model.system_prompt` and `experiment.system_prompt`. Writes nothing.

---

11. Cost, Tokens and Latency — Freezing and Roll-Ups

Purpose. Record what a round actually consumed — **tokens**, **dollars** and **wall-clock time** — and roll each up to the round and the run.

Only the dollar figure is *frozen*: it is computed at the moment the round runs, from the prices then in effect, so a later `model_catalog` price edit never rewrites history. Tokens and latency are measurements rather than derivations, so they are simply recorded as the call reports them.

Inputs / outputs.

- Inputs: per candidate, `input_tokens` / `output_tokens` (see *Where the token counts come from*, below) and the measured `latency_ms`; `model_catalog.input_price` / `output_price` as they stand at that moment (US dollars per 1,000,000 tokens).
- Outputs: `chat_round_candidate.token_cost`; `chat_round.token_cost` / `tokens` / `judge_tokens` / `judge_cost` / `latency_ms`; `run_result.total_generator_cost` / `total_judge_cost` / `total_tokens` / `total_latency_ms`.

Procedure.

```
# 1. Per candidate — freeze the cost AT ROUND TIME from the prices then in effect,
#    so a later model_catalog price edit never rewrites history. Prices are in US
#    dollars per 1,000,000 tokens.
for each candidate in round.candidates:
    prices := model_catalog[candidate.model]      # read the current prices now
    candidate.token_cost := candidate.input_tokens  / 1e6 * prices.input_price
                          + candidate.output_tokens / 1e6 * prices.output_price

# 2. Round roll-up over ALL generators and ALL iterations (summed over the round's
#    candidates). The JUDGE's usage is kept separate: it scores the candidates but
#    produces no candidate row of its own.
chat_round.token_cost := sum of candidate.token_cost
chat_round.tokens     := sum of (candidate.input_tokens + candidate.output_tokens)
chat_round.judge_tokens, chat_round.judge_cost := the judge's own usage

# 2b. Latency is wall-clock, so it does NOT sum the way cost does. Within one pass the
#     generators run in parallel, so that pass costs the SLOWEST of them; passes are
#     sequential, so a round costs the sum of its passes (§2).
for each pass in round.passes:
    pass.latency_ms := max(candidate.latency_ms for candidates in that pass)
chat_round.latency_ms := sum of pass.latency_ms  +  the judge's own latency

# 3. Run roll-up. Generator and judge cost are recorded SEPARATELY and together
#    make up the run's total spend.
```

```
run_result.total_generator_cost := sum of chat_round.token_cost over the run's rounds
run_result.total_judge_cost     := sum of chat_round.judge_cost over the run's rounds
run_result.total_latency_ms     := sum of chat_round.latency_ms over the run's rounds
run_result.latency_ms_{min,mean,median,max} := the five-number summary of
                                  chat_round.latency_ms across the run's rounds
```

Key decisions & edge cases.

- **Freeze at round time.** A later `model_catalog` price edit must never rewrite historical cost. This is why the price is read and multiplied in at round time, not derived on read.
- **A local/self-hosted model has price 0** (or near-zero) → it shows up as near-free.
- **Generator and judge costs are kept apart** so a report can isolate the variable under test; the judge is scoring overhead, not the generation being measured. The `moe` router's call (§19) is judge overhead by the same rule — it happens before generation and produces no candidate.
- **`moe` is where the split earns its keep.** A `moe` round pays for **one** generation whatever the member count, so `total_generator_cost` divided by `composite_mean` is a quality-per-dollar figure that can be set directly against a `synthesize` or `judge_best` config, which pays for every member on every round.
- **Latency is parallel-aware at the round level.** `chat_round.latency_ms` is the span of the **slowest** concurrent call (what the enrollee actually waited), not the sum of per-candidate `latency_ms`. Summing candidates gives cumulative compute-ish time — a different figure. `run_result.total_latency_ms` sums the per-round wall-clock times.

Schema touchpoints. Reads `model_catalog` (`input_price` , `output_price`). Writes `chat_round_candidate` (`input_tokens` , `output_tokens` , `token_cost` , `latency_ms`), `chat_round` (`token_cost` , `tokens` , `judge_tokens` , `judge_cost` , `latency_ms`), `run_result` (`total_generator_cost` , `total_judge_cost` , `total_tokens` , `total_latency_ms`).

---

12. Cohort Resolution and the Disjoint-Cohort Gate

Purpose. Turn a cohort into the concrete `run_enrollment` set at launch, and refuse to launch a run whose enrollees overlap another live run.

Inputs / outputs.

- Inputs: the target `cohort` and its `cohort_member` rows; the `run_enrollment` sets of all currently live runs.
- Outputs: `run_enrollment` rows for the new run, or a refusal with nothing written.

Procedure.

```
# At launch, in ONE middle-tier transaction. The transaction takes a lock (a Postgres
# advisory lock keyed to the experiment, or SELECT ... FOR UPDATE on the contended rows)
# so two overlapping launches cannot both pass the gate at once.

# Resolve the members: a cohort is an EXPLICIT roster (exact and reproducible).
# Anyone who has left the pool is skipped — their cohort_member row stays, because it
# records who the cohort held at the time, but they are not enrolled again (§28).
members := the cohort's cohort_member rows whose profile.opted_out_at is null

# Disjoint-cohort gate — evaluated BEFORE any write.
```

```
# live is the enrolled set of every currently-live run (paused counts as live).
live := the union of run_enrollment rows where run.state is running or paused
if any member also appears in live:
    refuse the launch and write nothing
    return

# The gate passed cleanly, so materialize the enrollment.
for each member in members:
    write run_enrollment(run_id, enrollee_id = member)
```

Key decisions & edge cases.

- **The gate is one locked transaction.** The check and the insert run together in a transaction that takes a lock (a Postgres advisory lock keyed to the experiment, or `SELECT … FOR UPDATE` on the contended rows), so two simultaneous launches cannot both pass a stale gate.
- **The gate must run before inserting the run.** Evaluating it after the insert is the classic bug that contaminates two runs at once. Order matters more than anything here.
- **Membership is fixed at launch.** Because the roster is explicit and the cohort is immutable while referenced, the enrolled set of a run never drifts after launch.

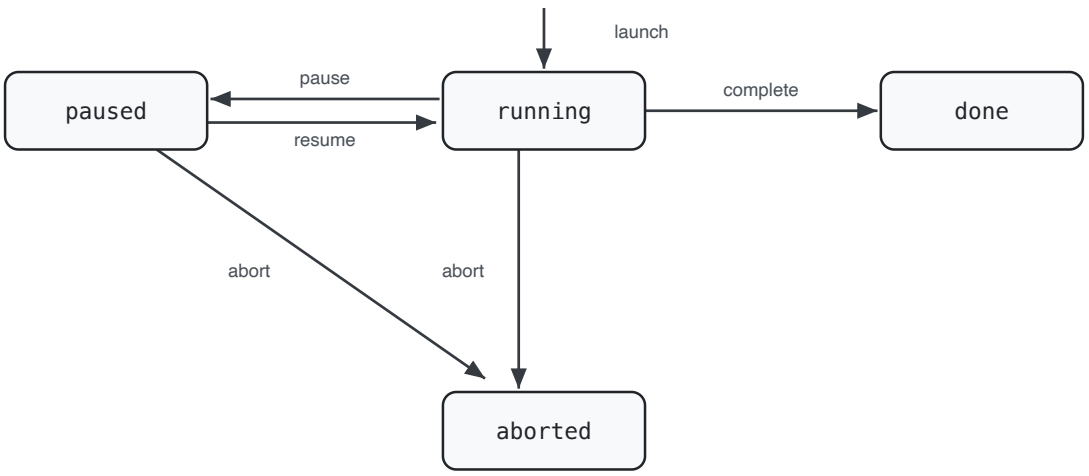
Schema touchpoints. Reads `cohort`, `cohort_member`, and `experiment_run.state` with its `run_enrollment` rows. Writes `run_enrollment`.

---

13. Clearing a Run — Cleanup and Purge

Purpose. Reclaim a finished run's storage in one of two modes — **Cleanup**, which keeps the run and its scorecard, and **Purge**, which removes the run altogether — protecting both the durable scorecard and the immutable definitions.

Run lifecycle. A run must reach a terminal state (`done` or `aborted`) before it can be cleared. `experiment_run.state` moves through this state machine:



Only `done` and `aborted` are clearable; the disjoint-cohort gate (§12) counts `running` **and** `paused` as live.

Inputs / outputs.

- Inputs: the target run (its `state` must be `done` or `aborted`); the actor's role; the mode (**Cleanup** or **Purge**).
- Outputs: deleted rows per mode, an export file, and audit entries.

Procedure.

```
clear_run(run, mode, actor):     # mode is either cleanup or purge
    # 1. Guard: only a finished run may be cleared.
    if run.state is neither done nor aborted:
        return REFUSED           # a running or paused run is never cleared

    # 2. Authorize by mode:
    #      cleanup is allowed for an admin, or for the owning experimenter (the one
    #              who composed it via experiment.created_by, or launched it via
    #              launched_by);
    #      purge   is allowed for an admin only.
    if not authorized(actor, mode, run):
        return REFUSED

    # 3. Export first and abort on failure — a delete before a confirmed export
    #    can lose data forever. The archive is COMPLETE: it is not what the
    #    operator ticked on an export screen, because anything left out here is
    #    gone for good a few lines below.
    archive := every chat_round and chat_round_candidate of the run,
               plus run_result, plus a header naming the run's resolved conditions
    # The operator chooses. Exporting is the default and the dialog pre-selects it,
    # because a delete before a confirmed export loses the data for good. Skipping is
    # allowed — clearing a botched run should not force an archive nobody wants — but
    # it is a deliberate, warned choice, and the audit entry records which was taken.
    if the operator asked for an archive:
        if write(archive, to durable storage, format is one of xlsx, csv, json) failed:
            record the failure and tell the operator
            return               # delete NOTHING: an export that was asked for and
                                 # failed is not consent to delete
    else:
        require the operator to confirm the warning that this data cannot be recovered

    # 4. Both modes reclaim the heavy produced data, but KEEP experiment_run and
    #    run_result so that analytics stay queryable.
    delete run_enrollment; chat_round and its chat_round_candidate rows;
           message and its message_recipient rows; and the run's audit_log rows

    # 5. Purge (admin only) additionally removes experiment_run, which CASCADES
    #    run_result away, so run_result never dangles, and NULLS issue.run_id on any
    #    issue filed while looking at this run. The issue itself survives: oversight
    #    data records what a person reported, and must not be erased by an operational
    #    cleanup or hold one up.
    if mode is purge:
```

```
        delete experiment_run    # cascades to run_result; nulls issue.run_id

    # 6. Record the clearing where the clearing cannot reach it. run_id is NULL on purpose:
    #    step 4 deletes the run's own audit rows and step 5 cascades them, so a run-scoped
    #    entry would erase the only record of where the archive went.
    insert audit_log(action := mode, target_type := 'experiment_run', target_id := run.id,
                     run_id := NULL, actor := actor,
                     after_state := { mode, archive: taken | declined, format, object_key })

# Definitions are immutable while referenced (RESTRICT, database-enforced), so deleting
# an experiment, nudge, model_config, or cohort requires PURGING its referencing
# runs first.
```

Key decisions & edge cases.

- **Export-first with abort-on-export-failure** is the rule that prevents irrecoverable data loss.
  - **The archive is complete, and it is not the selectable export.** Whatever is left out no longer exists once the delete runs, so Cleanup writes everything the run produced regardless of any screen setting. A run launched with `keep_candidates` has per-model candidates, and they go in.
  - **It carries the run's conditions, not only its rounds.** After a Purge the `run_result` scorecard is gone too, so this file may be all that survives. The conditions are their own part of the archive rather than a preamble to parse around.
  - **It goes to durable storage before the delete, never to a browser download.** A download can fail after the server believes it succeeded, and the delete would proceed on that false assurance. The archive is written to object storage and confirmed there, and the dialog stays up until that write completes.
  - **The object key is recorded in the audit entry.** Reaching the archive again is the exception, so the key lives in the clearing entry's `audit_log.after_state` — `{mode, archive: taken | declined, format, object_key}` — rather than earning a column of its own. That entry is written with `run_id` **null**, naming the run through `target_type` / `target_id`: a run-scoped entry would be deleted by the very cleanup it records, or cascaded away by the purge. The bucket is not public, so a link is a short-lived signed URL minted from that key.
  - **Shape follows the format, with the same four parts in each:** conditions, rounds, candidates and the scorecard. A workbook gives each its own sheet. Comma-separated output cannot hold four tables in one file, so it is a zip of four, which the interface should say plainly. The JSON form is one object whose `rounds` carry their `candidates` inside them.
  - **The detail is one table per master, and the keys are kept.** A single candidates sheet covers every round, each row carrying its `round_id`, and every round carries its `run_id`, so the parts rejoin exactly as the tables do.
  - **Identifiers travel with a label.** `chat_round_candidate.model_id` and the round's `best_model` / `worst_model` reference `model_config_model`, which is protected only while a run references it. A Purge lifts that protection, so an archive of bare identifiers can end up pointing at nothing. Each is written with its identifier for joining and the model's display name beside it, and the conditions carry the nudge and cohort names the same way.
- **Cleanup keeps `run_result`,** which is exactly why §6 aggregation must never assume the raw rounds still exist.
- **Context documents are not touched by either mode.** In this model context files attach to the *experiment* (`experiment_context_file`), shared across runs, not to the run.
- **`RESTRICT` on the definitions is database-enforced,** so the application must purge dependent runs before it can remove a referenced definition — surface that as an ordered operation, not a single delete.

Schema touchpoints. Reads `experiment_run.state`, `experiment.created_by`, `experiment_run.launched_by`. Deletes (Cleanup) `run_enrollment`, `chat_round`, `chat_round_candidate`, `message`, `message_recipient`, run

`audit_log`; (Purge) also `experiment_run` → cascades `run_result`. Depends on `RESTRICT` foreign keys from `experiment_run` to `experiment` / `nudge` / `model_config` / `cohort`.

---

14. Owner Reconciliation on Boot

Purpose. Guarantee exactly one admin — the install owner — on every boot. The owner is the `profile` whose email matches the backend `OWNER_EMAIL`; the middle tier reconciles to it at startup, promoting that account and demoting any other admin. There is no `app_settings` table and no stored UUID — the `admin` role on `profile` *is* the owner record, kept unique by a partial index.

Bootstrap (initial setup, and every later owner change). Supabase sign-up is invite-only, so the owner's account cannot be self-created: the installer creates it once in the dashboard and sets `OWNER_EMAIL` to that address, keeping the two equal. Every later owner change works the same way — point `OWNER_EMAIL` at the intended account and reboot. There is deliberately no in-app owner transfer.

Inputs / outputs.

- Inputs: the `.env` `OWNER_EMAIL`; Supabase Auth (`auth.users`), to resolve that email to an account id; the `profile` rows (`id`, `email`, `role`).
- Outputs: exactly one `profile.role = 'admin'`; any other admin demoted to `experimenter`; audit entries.

Procedure.

```
# Runs on middle-tier boot. Idempotent; converges to exactly one admin every boot.
# The owner is anchored by EMAIL: the account whose email matches OWNER_EMAIL. There
# is no stored UUID and no settings table.

# 1. Resolve OWNER_EMAIL to a Supabase Auth account. It may have no profile row yet,
#    because the account is created in the dashboard before its first login.
# Compare case-insensitively. Supabase Auth holds addresses lowercased, so an
# OWNER_EMAIL typed with different capitalization would resolve to nothing and send
# this straight to the fail-loud branch below, leaving the install with no admin.
owner_account := ask Supabase Auth for the user whose email = lower(OWNER_EMAIL)
if owner_account is null:
    # Installer error: OWNER_EMAIL names no Supabase user. Do NOT guess.
    log a loud, actionable error ("create this user in the Supabase dashboard")
    leave the install with no admin, then stop

# 2. Enforce exactly-one-admin, DEMOTE-FIRST so there are never two admins at once.
for each profile p where p.role = 'admin' and p.id is not owner_account.id:
    p.role := 'experimenter'

# 3. Create the owner's profile if absent, and make it admin. Match by the resolved
#    account id, so a stale profile.email cannot fool the reconciler.
upsert profile (id = owner_account.id, email = owner_account.email, role = 'admin')
#    The email written is the one Auth holds, not the .env spelling, so the mirror
#    matches its source exactly.
```

Key decisions & edge cases.

- **Idempotent, and the single source of truth.** It runs on every boot and converges to the same state — the `OWNER_EMAIL` account is admin, no one else is. Because it re-asserts this on every boot, `OWNER_EMAIL` is authoritative, which is *why* there is no in-app owner change: a boot would revert it.
- **Anchored on email by design — no stored UUID, no settings table.** The `admin` role on `profile` is the owner record; `OWNER_EMAIL` names which account holds it. The reconciler resolves the account through Supabase Auth and writes only the `role` — it never writes to Auth.
- **`profile.email` is written here, never read here.** The one comparison that decides anything is `OWNER_EMAIL` against Auth, and the profile is then found by the resolved account id. `profile.email` is a mirror of what Auth holds, so it takes no part in resolving the owner and none in authorizing a request: row-level security reads `profile.role` by `id` on every request, and `profile.id` equals `auth.users.id`. A corrupted `profile.email` cannot cost the owner their admin role, and the next boot repairs it.
- **Fail loud, never guess.** If `OWNER_EMAIL` names no Supabase account, the boot logs an actionable error and leaves the install with **no** admin, rather than promoting a wrong account or crashing. This also guards the only admin from being demoted when nothing matches.
- **Changing the owner is an operator action — Supabase dashboard, `.env`, reboot.** To change the owner's own email, edit that account in the dashboard and set `OWNER_EMAIL` to match; the id is unchanged, so it stays admin. To hand ownership over, point `OWNER_EMAIL` at another existing account: the next boot promotes them and demotes the old owner.
- **Application reconciliation and the database partial-unique index work together** — the index makes two admins impossible to persist; the boot logic decides *which* account is the admin.

Schema touchpoints. Resolves `OWNER_EMAIL` against Supabase Auth (`auth.users`); reads/writes `profile` (`id`, `email`, `role`); writes `audit_log`.

---

15. Keyset Pagination With `count(*) OVER()`

Purpose. Page through high-volume tables (`chat_round`, `chat_round_candidate`, `audit_log`) without `OFFSET`, which re-scans and discards all skipped rows and degrades linearly as you page deeper.

Procedure. Order by a stable, unique tuple — here `(created_at, id)` — and **seek** past the last row of the previous page instead of counting off rows:

```
SELECT ..., count(*) OVER() AS total_count
FROM chat_round
-- cursor from the previous page:
WHERE (created_at, id) < (:last_created_at, :last_id)
ORDER BY created_at DESC, id DESC
LIMIT :page_size;
```

Key decisions & edge cases. The cursor is the last row's `(created_at, id)`, not a page number. `count(*) OVER()` returns the grand total in the same round trip as the page, avoiding a second count query. The ordering tuple must be unique (hence including `id`) so the seek never straddles or skips ties.

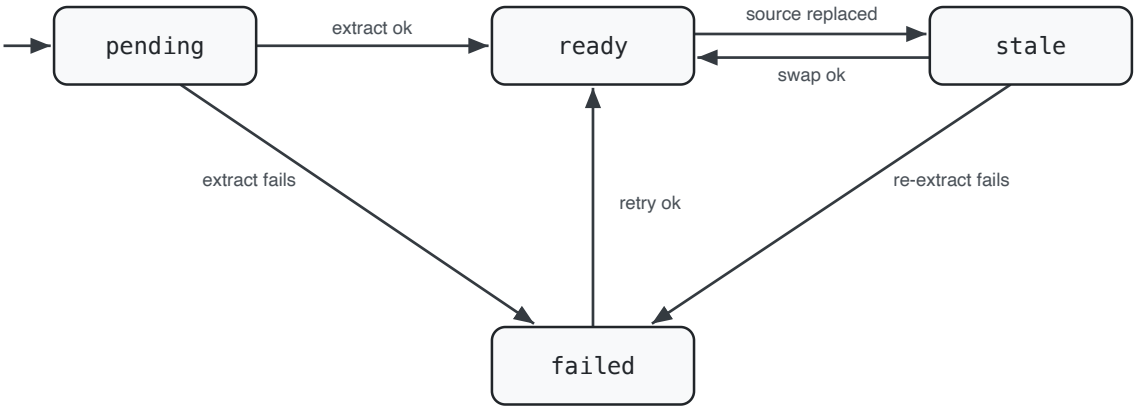
Schema touchpoints. Each high-volume table paged this way carries a `(created_at, id)` index for exactly this cursor. Keyset pagination alone keeps deep paging fast; range-partitioning these tables is a deferred, separate scaling option, not a

prerequisite for this algorithm.

16. Document Extraction: The Staging-Then-Swap State Machine

Purpose. Re-extract a document's text without ever taking the live snapshot offline, and record truthfully what the file currently holds. `document.extraction_status`, one of `pending`, `ready`, `stale`, or `failed`, is that truth.

Procedure. A re-extraction writes to a **staging** object key and **atomically overwrites** `document.extracted_text_path` only on success. `document.extraction_status`, one of `pending`, `ready`, `stale`, or `failed`, moves through this state machine:




```
# On the source being replaced: mark the current snapshot stale (it stays live).
on source replaced:
    document.extraction_status := stale

# Re-extraction never takes the live snapshot offline: build the new text off to
# the side, then flip ONE reference atomically so readers never see a torn state.
re_extract(document):
    if a live run references document (via experiment_context_file):
        return        # run-fidelity freeze — the swap is blocked mid-run

    # extract_text is format-driven. A PDF, Word or PowerPoint file goes through the
    # extraction libraries named in the architecture (Tika, unstructured, python-docx,
    # openpyxl, pdfminer), and a scanned PDF adds Tesseract OCR first. A plain-text,
    # Markdown or CSV file needs no extraction at all: the snapshot is a copy of the
    # file. Either way the result is UTF-8 text and nothing else.
    staging := extract_text(document.source)      # written to a staging key
    if extraction failed:
        document.extraction_status := failed      # old snapshot stays live*
        return
```



```
    document.extracted_text_path := staging       # atomic overwrite (the swap)
    document.extraction_status := ready

# * except a FIRST extraction, which has no prior snapshot to keep serving.
```

Key decisions & edge cases.

- `stale` **is a readable state, not an outage.** The snapshot is readable in every state **except** `pending` and a first-extraction `failed` that has no prior snapshot.
- `stale` **never returns to** `pending` — a document that once extracted is never empty again.
- **The swap is blocked while a live run references the document** (through its experiment's `experiment_context_file` links) — a run-fidelity freeze, so retrieval cannot shift under an in-flight run.
- **The swap is atomic.** The new text is built off to the side and swapped in with a single reference flip, so readers of the old snapshot never see a torn state.

Schema touchpoints. `document` (`extracted_text_path`, `extraction_status`, `updated_at`); the freeze checks live runs via `experiment_context_file`.

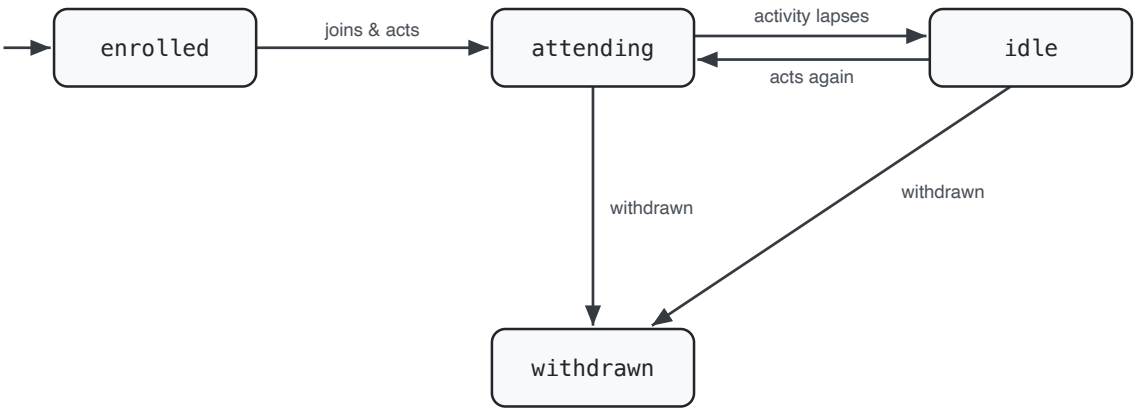
---

17. Attending Status — Enrolled vs Actively Present

Purpose. Distinguish an enrollee who is merely *enrolled* in a run from one who is *attending* it — taking part right now. This is what the live monitor's two roster counts read from, and what the operator↔enrollee panel uses to know who can see a message immediately.

Why derived, not stored. Presence drifts continuously as an enrollee acts, disconnects, and returns. A stored `is_attending` flag would need a writer on every heartbeat and would be wrong the moment activity lapses. So attending is computed from the fields that already exist on `run_enrollment`, and only the raw signal (`last_activity_at`) is written.

Presence states. An enrollment moves through these states. `attending` and `idle` are **derived** from `last_activity_at` recency, not stored; only `joined_at`, `withdrawn_at`, and `last_activity_at` are written.



```
# attending / idle are DERIVED from last_activity_at recency (NOT stored). The
# parent run must be running or paused for anyone to be attending. Withdraw can
# apply from any non-withdrawn state.
presence_state(enrollment):
    if enrollment.withdrawn_at is not null:
        return withdrawn
    if enrollment.joined_at is null:
        return enrolled       # in the roster but has not entered the run yet
    if now() - enrollment.last_activity_at <= presence_window:
        return attending
    return idle               # joined and not withdrawn, but activity has gone stale

# The live monitor shows two counts per run:
#   enrolled  = count of non-withdrawn enrollments        (stable after launch)
#   attending = the subset whose presence_state = attending   (drifts over time)
```

Key decisions & edge cases.

- **Withdrawn is not opted out.** `run_enrollment.withdrawn_at` ends one enrollee's part in **one run**, and the experimenter usually sets it. `profile.opted_out_at` is pool-level and the enrollee's own act: it withdraws every live enrollment at once and blocks future ones (§28). Opting out therefore *causes* withdrawal but is not a state of it — the two live on different rows, at different scopes, and this state machine models only the enrollment.
- **Attending is per-*enrollment*, not per-person** — the same enrollee can be attending run A while idle in a paused run B.
- **Membership is fixed at launch** (§12); only presence drifts, so the enrolled count is stable and only the attending count moves.
- **`last_activity_at` is refreshed by enrollee actions** (sending a prompt, opening the session), on the same cadence the middle tier already uses to drive Realtime presence.
- **The Realtime layer's own presence timeout wins when it exposes one,** so the "attending" cutoff matches the heartbeat that drives it.

Tunables (`.env` , with a hard-coded default): `PRESENCE_WINDOW_MINUTES` (**5**) — how many minutes of quiet before an attending enrollee reads as idle.

---

18. Prompt Enhancement — Rewriting the Enrollee Prompt

Purpose. When `experiment.enable_prompt_enhancer` is on, rewrite the enrollee's raw message into a better-engineered prompt before generation. It runs once per turn, after §21 accepts the request and before §1. In a controlled study it is a frozen, uniform condition element.

Where this comes from. Two established patterns meet here. Expanding a terse prompt with a fixed rewriter before generation is **prompt upsampling**, the caption-rewriting step image generators put in front of the model. Resolving a follow-up against the conversation is the **history-aware question contextualizer** from retrieval stacks such as LangChain's. Neither is reused as a library: both assume the rewrite is a quality improvement to be adopted freely, whereas here it is a *condition element* that must be frozen, versioned on the experiment, and applied identically to every enrollee — and must be provably faithful, since a rewrite that quietly adds a requirement changes what the study measured. That is what the seven validation checks and the `prompt_enhancer_version` asset exist for, and no off-the-shelf rewriter offers them. Internally it is the same skeleton as §20: deterministic preparation, one rewrite call, mechanical validation, safe fallback.

The idea that makes the rewrite auditable is that the enhancer may not improve freely. It picks from a **closed taxonomy of transformations** and declares which it applied, which turns an unfalsifiable instruction into something a validator can check.

Inputs / outputs.

- Inputs: the raw enrollee message; the round's attached context (the RAG block, if any — retrieval has already run, so it is an input here rather than a consumer); the last `ENHANCER_HISTORY_TURNS` turns of the session; the enhancer instructions named by `experiment.prompt_enhancer_version` (a versioned middle-tier asset); the enhancer model — the experiment's judge model (`experiment.score_judge_*`); the rubric factor names (`experiment.score_rubric_version`) as the rewrite's target.
- Outputs: `chat_round.prompt` (the effective prompt handed to generation), `chat_round.raw_prompt` (the original, when enhancement ran), and `chat_round.enhancement` (what happened).

The transformation taxonomy. Six transformations, each aimed at a rubric factor and each with a precondition, so an inapplicable one is refused rather than invented — the same applicability rule §4 uses for `groundedness`.

Transformation	Aims at	Precondition
`scope`	relevance	always
`ground`	groundedness	a source is attached, as context or as material carried in the prompt
`structure`	coherence	always
`constrain`	instruction_following	the raw message states a constraint
`disambiguate`	relevance	the round is not the first of its session
`neutralize`	relevance	always

`constrain` restates constraints the enrollee gave. It never introduces one, and that distinction is the whole fidelity guarantee — check 2 below is what enforces it.

`neutralize` removes the packaging, not the request: greetings, hedges, flattery and hostility go; what was asked stays. The line it must not cross is a constraint wearing politeness — in "could you please keep it short," the *please* goes and *keep it short* survives as a constraint.

Procedure.

```
enhance(round, experiment):
    if not experiment.enable_prompt_enhancer:
        chat_round.raw_prompt   := null           # nothing was rewritten
        chat_round.prompt       := round.message  # the original is the effective prompt
        chat_round.enhancement  := null
        return round.message

    raw     := round.message
    source  := attached_context(round)            # the RAG block, or material in the prompt
    history := last_turns(round, ENHANCER_HISTORY_TURNS)
    allowed := applicable(TAXONOMY, source, round.round_seq)
```

The cache key is what makes the policy uniform in practice and not merely in principle: two enrollees who type the same thing under the same frozen settings get the same rewrite.

```
    key := hash(experiment.id, experiment.prompt_enhancer_version,
                experiment.score_rubric_version, raw, digest(source), digest(history))
    if cached(key):
        return apply(cached(key))

    instructions := load_enhancer(experiment.prompt_enhancer_version)
    out := call(experiment.score_judge_*,
                assemble(instructions, raw, source, history, allowed,
                         rubric_factors(experiment.score_rubric_version)),
                temperature := 0)                # a frozen condition demands a fixed rewrite

    if out is error or timed out after ENHANCER_TIMEOUT_MS:
        return fall_back(raw, reason := 'enhancer_unavailable')
    if not valid(out, raw, source, allowed):
        return fall_back(raw, reason := first_failed_check)

    chat_round.raw_prompt  := raw                # never lose the enrollee's actual words
    chat_round.prompt      := out.enhanced       # the effective prompt for §1
    chat_round.enhancement := { status: 'applied', applied: out.applied,
                                failed_check: null, version, tokens, latency_ms }
    cache(key, out)
    return out.enhanced
```

A fallback is a recorded event, never a silent one.

```
fall_back(raw, reason):
    chat_round.raw_prompt  := raw                # set even though prompt equals it, so the
    chat_round.prompt      := raw                #   attempt is distinguishable from "off"
    chat_round.enhancement := { status: 'fell_back', applied: [], failed_check: reason,
                                version, tokens, latency_ms }
    return raw
```

The enhancer asset. A versioned middle-tier artifact, sitting beside the rubric registry and named by `experiment.prompt_enhancer_version`.

```
You rewrite a user's message into a better-engineered prompt for an AI assistant.

You do NOT answer it. You produce the request the user should have made.

Apply ONLY these transformations, and only the ones listed as available:
  scope        — make the implicit ask explicit; name the deliverable.
  ground       — bind the request to the attached source material.
  structure    — name the shape the answer should take.
  constrain    — restate constraints the user STATED, explicitly.
```

```
  disambiguate — resolve pronouns and ellipsis against the conversation.
  neutralize   — remove greetings, hedges, flattery and hostility; keep the request.

You must NOT:
  - Add any requirement, quantity, length, format or deadline the user did not state.
  - Introduce any name, term, figure or quotation absent from the message and source.
  - Answer the question, or include any part of an answer.
  - Issue instructions about the assistant's identity, role, or prior instructions.
  - Drop a constraint because it was phrased politely. "Please keep it short" keeps "keep it short".

Available transformations: {{allowed}}
The answer will be judged on: {{factors}}

List every constraint the user stated, quoting their words, and carry each one into
the rewrite. A constraint is anything bounding the answer: a prohibition, a limit, a
required format, a scope. Do not add one that is not there.

Return only JSON:
  {"enhanced": "<the rewritten request>", "applied": ["<transformation>", ...],
   "constraints": ["<the user's own words>", ...]}
```

Validation. Seven checks, all mechanical and all cheap. Any failure falls back to the raw message; there is no retry and no human in the loop, the same discipline §20 applies.

```
valid(out, raw, source, allowed):
    return  out.enhanced is non-empty and is a single message
    and     length(out.enhanced) <= ENHANCER_MAX_RATIO * length(raw)
    and     numbers(out.enhanced) subset of numbers(raw) + numbers(source)
    and     proper_nouns(out.enhanced) + quoted(out.enhanced)
                subset of proper_nouns(raw) + quoted(raw) + terms(source)
    and     no directive-override phrasing in out.enhanced
    and     out.applied subset of allowed
    and     cosine(embed(out.enhanced), embed(raw)) >= ENHANCER_MIN_SIMILARITY
    and     every constraint in out.constraints is present in out.enhanced
    and     every negation-bearing clause in raw is covered by out.constraints
```

Reading them in order: the first bounds size, so the enhancer cannot answer instead of asking. The second is the fidelity guarantee — a quantity absent from both message and source is an invented requirement, which is how "in exactly 500 words" gets caught. The third stops fabricated specifics. The fourth stops a rewrite issuing role instructions, which would escape the study's own framing. The fifth checks that every declared transformation was permitted.

The seventh guards the opposite failure from the second: where the second stops the enhancer inventing a requirement, the seventh stops it losing one. A rewrite that tidies "don't use any external libraries" out of existence passes every other check. The enhancer declares the constraints it found in the user's words and each must survive into the rewrite; because a declared list can be gamed by omission, it is paired with a scan for negation-bearing clauses in the original.

This check is deliberately weaker than the second through the fifth, which are exact set comparisons. A constraint is a phrase rather than a token, so matching one is a judgment and the scan will not be airtight. It earns its place anyway: every other

prohibition in the enhancer's instructions has a mechanical check behind it, and an instruction to a model is a request rather than a guarantee.

Tunables (`.env` , with hard-coded defaults): `ENHANCER_MAX_RATIO` (**6**) — the ceiling on how much longer the rewrite may be than the raw message, which stops the enhancer answering instead of asking; `ENHANCER_MIN_SIMILARITY` (**0.60**) — the cosine floor the rewrite must hold against the original, so it stays the same request; `ENHANCER_HISTORY_TURNS` (**4**) — how many prior turns the rewrite may read when resolving a follow-up; `ENHANCER_TIMEOUT_MS` (**20000**) — how long to wait before falling back to the untouched prompt.

Key decisions & edge cases.

- **Frozen, uniform policy.** `enable_prompt_enhancer` and `prompt_enhancer_version` are substantive experiment fields, locked once runs exist and applied identically to every enrollee. What varies per enrollee is their own message; the policy is constant, which is what a fixed condition requires — comparability never demanded identical prompts. The call runs at temperature 0 with the result cached, since a stochastic rewriter would leave the policy uniform while its application varied.
- **The judge scores against the raw prompt.** §4 measures the answer against `raw_prompt` , the enrollee's own words, so enhancement cannot inflate a score by rewriting the question the answer is judged against.
- **Store both, always.** `raw_prompt` preserves the enrollee's actual words; `prompt` is what was submitted. The original is never lost, which is what lets a not-blinded enrollee see the rewrite, and what a later approve-or-edit step would act on.
- **A fallback is recorded, not swallowed.** `chat_round.enhancement` distinguishes an applied rewrite from an attempted one, because otherwise a rejected rewrite is indistinguishable from enhancement having been off — and the study argument rests on uniform application, so exactly that event is the one that has to leave a trace. A run's fallback rate is read from these rows the way its escalation rate is read from `moe_routing` .
- **The enhancer is the judge model.** It crafts a better question and does not generate the answer it later scores, so §5's self-preference concern does not apply. It does aim the rewrite at the rubric the judge grades on, which is the intent. A separately-pinned enhancer model is a future option.
- **Aims at the rubric, degrades with no source.** The rewrite targets the experiment's rubric factors. If a factor needs a grounding source and none is attached, `ground` is not among the available transformations and check 5 rejects a rewrite that claims it, so the enhancer cannot fabricate a "ground this in the source" instruction.
- **Faithful, single pass.** One enhancer call; it sharpens and structures without adding requirements the user did not ask for. Iterating or searching over enhancement policies is offline prompt optimization (a future direction), not this per-turn step.
- **`neutralize` makes tone a controlled variable.** Affect in a prompt steers an answer: flattery invites agreement, hostility invites terseness. Stripping it means the answer depends on what was asked rather than how it was packaged. Because the enhancer is a versioned asset, one arm can run instructions that include this transformation and another omit it, which makes the effect itself measurable.
- **Multi-turn is resolved against history.** A follow-up such as "what about the second one?" is meaningless alone, so prior turns are supplied as read-only context and `disambiguate` resolves the reference. The rewrite stays scoped to the current request.
    - The turns available are the ones §23 assembled, so the enhancer never sees more of the conversation than the models will. `ENHANCER_HISTORY_TURNS` narrows that window further; with `enable_chat_history` off there is no history and `disambiguate` cannot fire.
- **What downstream reads.** This step changes what `chat_round.prompt` holds, so every later consumer of that field sees the rewritten text — generation in §1, and the router under `moe` (§19), which is simply one of those consumers rather than a feature that interacts with this one. Retrieval is upstream and unaffected. Scoring reads `raw_prompt` instead, per the decision above.

- **Visibility rides `blinded`.** In a blinded experiment the enrollee sees only their own words; unblinded, the enhanced prompt is shown alongside the original. Both are stored either way, so the choice affects display and never the record.
  Schema touchpoints. Reads `experiment.enable_prompt_enhancer`, `experiment.prompt_enhancer_version`, `experiment.human_review_enabled`, `experiment.score_judge_*` (as the enhancer model), `experiment.score_rubric_version`, `experiment.blinded`. Writes `chat_round.raw_prompt`, `chat_round.prompt`, and `chat_round.enhancement`. The enhancer's tokens are counted in `chat_round.judge_tokens` / `judge_cost`, because the call is made by the trusted judge and is overhead on the round rather than generation being compared — the same accounting §19 applies to the router's call.

---

19. MoE Routing — Embedding First, the Judge on a Tie

Purpose. Under `combine_method = moe`, pick **which single member of the config answers this prompt**, before any generation happens. "MoE" here is *orchestration* mixture-of-experts — a router choosing one whole model — not the architectural kind baked inside a model like Mixtral. The config's members are the experts; the router is a dispatcher in front of them.

It runs in **two stages, and the second one usually does not run**:

1. **Embedding similarity** — compare the prompt with each member's capability text. Free, deterministic, no model call. When one member wins clearly, that is the answer.
2. **The judge as router** — only when several members **tie**, the experiment's one trusted judge is asked, at temperature 0, to **predict** which of the tied contenders will answer this prompt best.

The second stage exists because of a known limit of the first: embeddings capture **topic**, not difficulty. A hard question and an easy one on the same subject sit next to each other in embedding space, so similarity alone cannot separate a strong expert from a cheap one when both are on-topic. That case is exactly what reads as a tie.

Inputs / outputs.

- Inputs: the round's **effective** prompt (enhanced per §18 when enabled — the router must see what the expert will see); the config's members (`model_config_model` with their `weight`) and each member's capability text (`model_catalog.expertise`, falling back to `aptitudes` + `notes`); the active `embedding_model`; the experiment's trusted judge (`experiment.score_judge_*`) and its rubric factors; the `.env` constants `MOE_MIN_SIMILARITY`, `MOE_TIE_MARGIN`, `MOE_ESCALATE`.
- Outputs: the routed member, handed to §1 as the round's only generator; and `chat_round.moe_routing`, the recorded decision.

Procedure — stage 1, the embedding router. Free, deterministic, and no model call. It ends the round's routing outright whenever one member is a clear winner.

```
route(prompt, members, experiment):

    # Embed the EFFECTIVE prompt with the same model the capability vectors used;
    # vectors from different embedding models are not comparable (see §7).
    qvec := embed(prompt, active_embedding_model)
    for each m in members:
        sim[m] := cosine_similarity(qvec, capability_vector(m))

    ranked := members sorted by sim, descending
```

```
    top    := ranked[0]

    # The CONTENDERS are every member within MOE_TIE_MARGIN of the leader and at or
    # above the floor. Exactly one contender means the embeddings were decisive.
    contenders := [ m in ranked where sim[m] >= sim[top] - MOE_TIE_MARGIN
                                and sim[m] >= MOE_MIN_SIMILARITY ]

    # Nothing cleared the floor: the embeddings have no opinion at all, which is a
    # tie of the widest kind. Every member becomes a contender.
    if sim[top] < MOE_MIN_SIMILARITY:
        contenders := members

    if count(contenders) = 1:
        return decide(contenders[0], stage = 'embedding')      # DONE — no LLM call

    if not MOE_ESCALATE:                                       # stage 2 switched off
        return decide(highest_weight(contenders), stage = 'embedding',
                      fallback = 'tie_by_weight')
```

Stage 2 — the judge as router. Reached only on a tie. The experiment's one trusted judge is asked, before anything is generated, to predict which contender suits this prompt.

```
    # No separate router model, and no leakage: this is a PRE-generation call about
    # the PROMPT, while scoring is a POST-generation call about an ANSWER.
    router := experiment.score_judge_*
    ballot := build_router_prompt(
                  prompt,
                  # each contender: its display name, its capability text, and its tier
                  [ (m.display_name, capability_text(m), tier(m.weight)) for m in contenders ],
                  rubric_factors(experiment.score_rubric_version))   # what "best" means here

    # temp 0 -> the same prompt routes the same way, run after run
    pick := call(router, ballot, temperature = 0)

    # Never fail a round on the router: an unparseable, timed-out, or errored call
    # falls back deterministically to the strongest contender.
    if pick is not one of contenders:
        return decide(highest_weight(contenders), stage = 'judge', fallback = 'router_error')

    return decide(pick, stage = 'judge')
```

The two helpers. `decide` is where the decision is recorded; `capability_vector` is the cache that keeps stage 1 free.

```
decide(member, stage, fallback = null):
    chat_round.moe_routing := { stage, chosen: member.id, sim, contenders, fallback }
    return member
```

```
capability_vector(m):
    # Embedded ONCE per (catalog model, capability text, embedding model) and cached in
    # the middle tier. A catalog edit or an embedding-model switch invalidates the entry,
    # exactly as it invalidates the document index (§9).
    return cached_embed(capability_text(m), active_embedding_model)

capability_text(m):
    # The curated sentence if there is one; otherwise the thin fallback.
    return m.catalog.expertise
           or join(m.catalog.aptitudes, ', ') + '. ' + m.catalog.notes

build_router_prompt(prompt, contenders, factors):
    # A ballot, not a conversation: the judge is asked to PREDICT, before anything is
    # generated, which contender will answer this prompt best. It returns one name and
    # nothing else, so an unparseable reply is detectable and falls back deterministically.
    return  "You are choosing which assistant should answer a question. "
            "Do NOT answer it yourself."
          + "Question: " + prompt
          + "Candidates, each with what it is good at:"
          + for each c in contenders: c.display_name + " (tier " + c.tier + ") — " + c.capability_text
          + "The answer will be judged on: " + factors
          + "Reply with exactly one candidate name and no other text."
```

Key decisions & edge cases.

- `MOE_TIE_MARGIN` **is the knob that trades cost for care.** Larger margin → more rounds treated as ties → more judge calls, better-considered routing, higher cost. Zero margin → stage 2 effectively never fires. The escalation rate is observable (`chat_round.moe_routing.stage`), so a study can tune this from evidence rather than guess.
- **Reproducibility is the reason for every determinism choice here.** Stage 1 is arithmetic on fixed vectors, stage 2 runs at temperature 0, and a tie the judge cannot resolve falls to the highest `weight` rather than to chance. The same prompt therefore routes the same way every time, keeping the model set-up a fixed condition.
- **The router's spend is judge spend.** Its tokens go to `chat_round.judge_tokens` / `judge_cost` , never to `token_cost` (§11). That keeps `total_generator_cost` a clean measure of the generation under test, which matters here more than anywhere: MoE's headline claim is *one cheap generation per round*, and it would be self-defeating to let routing overhead contaminate the number that claim is read from.
- **Routing is per round, not per session.** Each prompt is routed on its own merits, so a multi-turn session can switch experts mid-conversation; `chat_round.moe_routing` makes each switch visible. Pinning one expert for a whole session (for conversational consistency) is an additive later option, not V1 — it would trade the per-prompt fit that is the point of routing.
- **The router is decided once per round, and the self-improvement loop does not re-route.** With one generator, §2's broadcast degenerates to the shape of `single` : the routed expert refines its own answer with no peers to show it. Re-routing between iterations is the escalation cascade, a deliberate non-goal for V1.
- **The capability vectors are the router's own cache.** Embedded once per (catalog model, capability text, embedding model) in the middle tier, synchronously, so a just-added model routes correctly on its first round.
- **A member with no capability text can still be routed to.** Its similarity is effectively 0, so it never wins stage 1 alone, but a round where nothing clears the floor makes every member a contender and the judge can pick it on the prompt alone. The authoring UI warns at compose time when a `moe` config has members without `expertise` .

- **Blinding is unaffected.** The routing record is staff-facing telemetry; the enrollee sees one answer, as under every other method.

Tunables (`.env` , with hard-coded defaults): `MOE_MIN_SIMILARITY` (**0.15**) — the floor a member must clear to contend; `MOE_TIE_MARGIN` (**0.05**) — how close to the leader still counts as a tie; `MOE_ESCALATE` (**true**) — whether a tie may escalate to the judge.

Schema touchpoints. Reads `model_config.combine_method` , `model_config_model` (`catalog_id` , `weight`), `model_catalog` (`expertise` , `aptitudes` , `notes` , `display_name`), and `experiment` (`score_judge_provider` / `score_judge_model` , `score_rubric_version`). Writes `chat_round.moe_routing` , and `chat_round.judge_tokens` / `judge_cost` for the stage-2 call. Rolls up through `run_result.model_perf` , where under `moe` a "win" is a round the model was routed to — so the run's routing distribution survives a cleanup of the raw rounds.

---

20. Canonicalizing a Saved Question

Purpose. Turn the question a person typed into the **generalized, normalized** form stored in `nl_query.canonical_prompt` . Two things happen. The literal values the agent bound become named placeholders, so *"what were the top 10 runs by score?"* is stored as *"what were the top `{{k}}` runs by score?"* — one saved question that serves any *k* rather than one that serves only 10. And the wording is normalized, so that everyone who asks the same thing in different words reaches that same row. It runs once, when a question is saved (§8), and never on the reading path.

The stored text has three jobs, and they pull against each other: it is embedded into `query_vector` so a *paraphrase* finds it; it is the text re-read to regenerate `derived_sqls` when the schema moves, so its *meaning* must not drift; and it is the full-question subtitle in the interface, so it must stay readable.

Inputs / outputs.

- Inputs: the question as typed; the value-to-placeholder mapping the agent reports for the literals it bound; the values the user chose to parameterize; the canonical lexicon; `ASK_FAST_MODEL` .
- Outputs: `canonical_prompt` , and the `params` entries for the placeholders it contains.

Procedure.

```
canonicalize(typed, bound_values, chosen, lexicon):

    # 1. Substitute, deterministically. The literals come from the AGENT, which knows
    #    exactly which values it bound; they are never re-detected by parsing English.
    text := typed
    for each value in chosen:
        replace value's span in text with '{{' + its param name + '}}'
    text := trim(text); collapse runs of whitespace; fold smart quotes and dashes to ASCII
    substituted := text                       # always acceptable; the fallback below

    # 2. Normalize the WORDING with a model. It may not decide what the question MEANS.
    rewritten := call(ASK_FAST_MODEL, canonicalizer_prompt(lexicon), substituted)

    # 3. Validate mechanically. Any failure falls straight back — there is no retry,
    #    because `substituted` is always good enough and a retry only adds latency.
    if not valid(rewritten, substituted, lexicon):
```

```
        log the rejection with the token that caused it     # this log grows the lexicon
        return substituted
    return rewritten


valid(out, src, lexicon):                      # six checks, all cheap and mechanical
    placeholders(out) == placeholders(src)                          # same set, same spelling
    and numbers(out) == numbers(src)                                # multiset; none added or lost
    and no name from IDENTS appears in out that is absent from src  # blocks 'composite_mean'
    and every content word in out but not in src is a lexicon TARGET
    and words(out) <= 1.3 * words(src)                              # blocks elaboration
    and cosine(embed(out), embed(src)) >= 0.85                      # a semantic floor
```

The canonicalizer prompt. It is a versioned middle-tier asset, like the rubric registry, and the lexicon is injected into it.

```
You normalize a saved question into its canonical form for a research platform.

Your only job is to normalize how the question is WORDED. You must not decide what it MEANS.

Rewrite it so that people asking the same thing in different words produce the same text:
  - Remove conversational framing: greetings, hedges, "can you", "please", "just".
  - Replace informal wording with the platform term, using the vocabulary below.
  - Resolve elliptical references ("the X one") to the full noun phrase.
  - Use a plain, direct question. Keep one clause per thing asked.

You must NOT:
  - Resolve a metric, measure, statistic or formula. Words like "best", "average", "top",
    "score" and "cost" must survive as written. Never name a column, table or calculation.
  - Add, remove or change any number.
  - Add, remove, rename or reorder any {{placeholder}}. Copy each one exactly.
  - Add any filter, grouping or time range the question did not state.
  - Answer the question or comment on it.

Platform vocabulary — informal wording on the left, the platform's term on the right:
  study, trial, run group ................ experiment
  coaching tip, hint, steering message ... nudge
  group of people, participant group ..... cohort
  subject, participant, respondent ....... enrollee
  model set-up, model bundle ............. model configuration
  spend, what we spent ................... cost
  ended up on top, came out best ......... performed best
  [...the rest of the lexicon]

Return only JSON: {"canonical": "<rewritten question>"}
```

The lexicon asset. The vocabulary block above is generated from `canonical-lexicon.v1.yaml`, which ships beside the rubric registry. Version 1 carries 36 target terms — entities, roles, artifacts, lifecycle states, actions, and metric-neutral qualifier phrases — and 168 variant phrasings.

Two properties of that file are worth stating, because both are load-bearing rather than incidental:

- **No metric, statistic or measure name is a target.** Resolving *average* to a particular mean, or *cost* to a particular sum, is precisely what the third and fourth checks exist to catch, so admitting those words to the allowlist would defeat the validator. Every qualifier entry is neutral about the measure: *performed best* says who came out ahead without saying what decided it.
- **Each variant records where it came from.** A variant is `attested` when the phrase occurs in one of the platform's existing natural-language questions, and `proposed` when it is an unverified guess. Version 1 is 15 attested against 153 proposed, the expected shape before the platform has users. The rejection log corrects that ratio. **A worked example.** On the Results screen, someone types:

> *hey — for the terse-prompt one, can you pull up which coaching tip ended up on top across the runs, and roughly what we spent on each? just the top 3 please*

The agent answers it, binding two literals: the experiment name, and a limit of 3. The user saves, and parameterizes both.

Step 1, substitution — reproducible, no model involved:

> `hey — for the {{experiment}} one, can you pull up which coaching tip ended up on top across the runs, and roughly what we spent on each? just the top {{limit}} please`

Step 2, the rewrite:

> `For the {{experiment}} experiment, which nudge performed best across its runs, and what did each one cost? Show the top {{limit}}.`

It dropped the framing, mapped *coaching tip* to **nudge**, resolved *the terse-prompt one*, and folded *ended up on top* and *what we spent* into the platform's wording. It left **"performed best"** and **"cost"** unresolved, which is the point: whether "best" means the composite mean, and whether "cost" is generator plus judge, are decisions that belong in `derived_sqls` under the context pack's rules, not in the question.

Step 3, validation — placeholders unchanged; no number added or lost; no schema identifier introduced; every new content word (*experiment*, *nudge*, *cost*, *performed best*) is a lexicon target; 21 words against 34; cosine well above the floor. Stored.

And the rejection it exists for. Had the model returned the tidier-sounding:

> `Top {{limit}} nudges in experiment {{experiment}} ranked by mean composite score, with total cost.`

— the third check fires on `composite`, and the fourth on *ranked*, *mean* and *total*, none of them lexicon targets. It is rejected and the substituted form is stored. The rewrite that reads best is exactly the one that quietly resolved two of the traps in §8.

Where this comes from. Query normalization before caching is standard practice — search engines normalize before a cache lookup, and semantic caches such as GPTCache embed a normalized form rather than the raw string. What differs here is the consequence of a wrong match: those systems risk a stale answer, while a wrong match here runs the wrong SQL under the asker's own permissions. That is why the rewrite is followed by six mechanical checks and falls back to the deterministic

substitution on any failure. No library pairs normalization with that kind of validation, and the checks depend on platform specifics — the schema's own identifier set, the canonical lexicon — so it is written here rather than pulled in.

Key decisions & edge cases.

- **Substitution is deterministic, and the rewrite is disposable.** The stored text is the input to a later regeneration, so nothing a model produced may reach it unchecked. The substituted form is always a valid answer, which is what makes discarding a bad rewrite free.
- **The values come from the agent, never from parsing the question.** The agent already knows which literals it bound; re-detecting them in English would be a second, worse extractor.
- **Embeddings handle paraphrase; substitution handles values.** That division is the whole design. A number or a name is a weak signal in an embedding, so those are the parts replaced by placeholders, and the wording is left for the vector to match.
- **A question with no parameters is stored as typed**, bar the mechanical tidying. Two people wording it differently produce two rows, and matching still works, because the embeddings sit close together. The duplication is in storage, not in retrieval.
- **The lexicon is one asset with two jobs:** it is pasted into the canonicalizer's prompt as the vocabulary to prefer, and it is the allowlist the validator checks new content words against. One file keeps the two from drifting apart, which would let the model introduce a term the validator then rejects.
- **A thin lexicon fails safe and grows itself.** Early on, the fourth check rejects most rewrites and almost everything falls back to substitution. Each rejection is logged with the token that caused it, and that log is the backlog for the next revision of the lexicon.

Schema touchpoints. Writes `nl_query.canonical_prompt`, `params`, and `query_vector`. Reads the canonical lexicon and the enhancer-style prompt asset from the middle tier; neither is in the database.

---

21. The Sufficiency Gate — Turning a Request Back Before It Is Answered

Purpose. Decide whether an enrollee's message can be answered at all, and turn it back with a reply when it cannot. It runs before §18, because a request that cannot be answered is not worth rewriting. Three tests catch three failures: a message about nothing the corpus holds, a message that leaves out something the experiment declared it needs, and — only on a run that is eliciting — a message the judge reads as unanswerable with no checklist to test it against.

The gate is also where the Socratic switch lands. `experiment_run.socratic_enabled` (§30) chooses how the trusted judge words the reply: a question asking for one missing piece, or an instruction to rewrite the request and resend it. Detection is the same either way, so the switch is a clean single-variable manipulation.

Why a turned-back attempt is not a round. A round is the unit every statistic counts — round count, cost per round, score distribution — so admitting an answerless exchange would distort all three. The gate loops outside the round: attempts and the replies they drew accumulate, and one `chat_round` is written when an attempt passes, carrying the detail in `clarification` and the tally in `retry_count`.

Inputs / outputs.

- Inputs: the enrollee's message; the experiment's context documents, if it has any; `experiment.required_slots`; `experiment_run.socratic_enabled` and `experiment.socratic_version`; `RAG_MIN_SIMILARITY` and `RAG_TOP_K`; the trusted judge model (`experiment.score_judge_*`), which runs the tests and writes the reply.
- Outputs: either a reply shown to the enrollee with no round written, or the composed request handed on to §18, together with `chat_round.clarification` and `chat_round.retry_count` recording any attempts that came first.

Procedure.

```
gate(message, experiment, run, attempts):
    # Test 1 — is anything in the corpus close enough to be worth grounding in?
    if experiment has context documents:
        hits := retrieve(message, RAG_TOP_K, RAG_MIN_SIMILARITY)      # §7
        if hits is empty:
            return DECLINE(scope := experiment.scenario)
    else:
        hits := []                    # no corpus, so this test cannot fire

    # Test 2 — does the request carry what this experiment declared it needs?
    if experiment.required_slots is not empty:
        # Under the questioning style the enrollee supplies pieces one at a time, so
        # everything said so far counts. Under the rewrite style they were asked for one
        # complete request, so each attempt is judged on its own.
        known := message
        if run.socratic_enabled:
            known := message + every earlier attempt in this exchange
        missing := [ s for s in experiment.required_slots
                     where s.required and not filled(s, known) ]
        if missing is not empty:
            return CLARIFY(missing)
        return PROCEED(hits)

    # Test 3 — nothing was declared, so there is no checklist to test against. Only a run
    # that is eliciting asks anyway, and it asks the judge to read the request instead (§30).
    if run.socratic_enabled:
        verdict := call(experiment.score_judge_*,
                        completeness_prompt(message, experiment.system_prompt),
                        temperature := 0)
        if verdict names anything missing:
            return CLARIFY(verdict.missing)

    return PROCEED(hits)
```

The two failing outcomes differ in what they are about. A `DECLINE` means the corpus has nothing on the topic. A `CLARIFY` means the topic is covered but the request is incomplete. The switch then decides how each is worded, which is four phrasings in all.

```
compose(session, experiment, run):
    attempts := []
    forever:
        message := the enrollee's next message
        outcome := gate(message, experiment, run, attempts)
        if outcome is PROCEED:
            break
        reply := call(experiment.score_judge_*,
```

```
                    gate_reply_prompt(outcome, run.socratic_enabled,
                                      experiment.socratic_version),
                    temperature := 0)
        show reply to the enrollee
        append { message, reason: outcome, reply, tokens,
                 socratic: run.socratic_enabled } to attempts
        # deliberately no chat_round, and the models under test never see this message

    if run.socratic_enabled:
        # The PAIRS, not the bare answers: "crude oil" is meaningless without the question
        # that drew it, so joining only the enrollee's messages would hand the models a run
        # of fragments and throw away the very information the dialogue collected.
        raw := render_exchange(attempts, message)
            # each attempt as  "<enrollee's words>" then "Asked: <the gate's reply>",
            # in order, ending with the message that finally passed
    else:
        # Each attempt was judged on its own, and the enrollee was asked for one complete
        # request, so the one that passed stands alone. Folding the rejected drafts back in
        # would hand the models the incomplete versions the enrollee was told to replace.
        raw := message

    write one chat_round with
        raw_prompt    := raw                          # §18 enhances this
        clarification := attempts or null
        retry_count   := count(attempts)

gate_reply_prompt(outcome, socratic_enabled, version):
    # The wording is a versioned middle-tier asset, bundled with the release exactly as a
    # rubric is (§4's load_rubric) and loaded once at boot into a registry keyed by version
    # string. It carries four phrasings, one per (style, outcome) pair; this selects the one
    # in force and hands the judge the specifics — which pieces are missing, or what the
    # experiment does cover. The experimenter writes none of it, and a published version's
    # wording never changes, so both arms of a comparison speak in frozen terms.
    return the bundled instructions for `version`,
           at [socratic_enabled ? 'socratic' : 'direct'][outcome.kind]
```

Key decisions & edge cases.

- **What `required_slots` looks like.** A short declared checklist on the experiment, not free text. A study about energy markets might require `commodity` ("which commodity — crude, natural gas, power?"), `region`, and `time_frame`, each with the wording the gate uses when it turns a request back. "What happened to prices?" is missing all three; "what happened to European gas prices last winter?" fills them and passes.
- **The switch changes the reply, not the detection.** With a checklist declared, the same missing piece stops the same message under either setting; only the wording the enrollee gets back differs. That is what makes `socratic_enabled` a clean single-variable manipulation — the arms are not being held to different standards, they are being repaired by different means.
- **Test 3 runs only when there is nothing to test against, and only when eliciting.** An experiment that declares no `required_slots` has no checklist, so a study of eliciting would have nothing to elicit. On such a run the judge reads the request itself. It costs a judge call on **every** message rather than only on the ones it stops, since a message cannot be

known to be complete without being read — visible in `judge_cost`, and the price of being able to run this study at all. Where a checklist exists, the checklist decides and this test never runs, so both arms detect identically.

- **With the switch off and nothing declared, there is no gate.** The models answer whatever the enrollee typed. That is not a degraded control: it is ordinary chatbot behavior, and it is the honest baseline for asking whether eliciting is worth doing.
- **Attempts accumulate only under the questioning style.** An enrollee being asked one thing at a time is not made to restate the whole request, so everything said so far counts. An enrollee asked to rewrite and resend was asked for something self-contained, so each attempt is judged alone — otherwise "resend a complete request" would be untrue, and a request could pass on pieces scattered across drafts the enrollee had been told to replace.
- **What reaches the models differs by style, for the same reason.** Under questioning the exchange is rendered as question-and-answer pairs, because an answer such as "crude oil" is meaningless without the question that drew it. Under rewriting the final request already stands alone, so it is used as it is and the rejected drafts stay in `clarification` as record.
- **The gate is central, and it is the only thing that elicits.** Refusing to open a round makes the standard uniform: every enrollee meets the same test, so it is a frozen condition element rather than model-dependent behavior. The models under test never ask for a missing detail — nothing in their system prompt tells them to, and if one does anyway it is scored as the answer it is.
- **There is no attempt cap.** A request is not accepted until it is well formed, so the loop ends when the enrollee produces something answerable or leaves. An enrollee who gives up shows as an abandoned session rather than as a bad answer, which is the truthful place for it — though the judge cost their attempts spent has no round to be charged to, so it is not counted anywhere.
- **Every stopped attempt counts in `retry_count`, under both settings.** That is what makes the two arms comparable: the study asks whether dialogue reaches an answerable request in fewer attempts than a demand to rewrite. §6 rolls the counts into `run_result.retry_stats`.
- **The gate runs on the trusted judge at temperature 0.** Both tests and the reply come from the model that scores the experiment, for the reason §18 and §19 use it: a pre-generation decision has to be identical for every enrollee, and the models under test must not make it, or the gate becomes part of what is being compared instead of a constant.
- **Cost lands in `judge_tokens` and `judge_cost`.** The gate's calls are overhead on the round that eventually gets written, so counting them there keeps `total_generator_cost` an honest measure of the generation under comparison.
- **The relevance test cannot fire without a corpus.** Many experiments run with no context documents, and for those the floor has nothing to measure against. Those experiments bound their topic through `system_prompt`, or through `required_slots`, or not at all.
- **`required_slots` is off by default.** An empty or absent list disables the second test, so an experiment that wants no gate does not get one — unless the run is eliciting, which is test 3. The column is frozen once the experiment has runs, like the other substantive fields.
- **A gate that can fire needs wording to fire with.** An experiment that attaches context documents or declares `required_slots` must also name an `experiment.socratic_version`, because either one lets a test stop a message and the judge then has nothing to compose a reply from. The compose screen requires it in those cases. Only an experiment doing neither may leave it null, and §30 carries the matching check at launch.
- **Slot filling is a judgment, not a parse.** Whether a message supplies "which market or instrument" is decided by the judge model reading the declared description, so the declaration is written for a reader rather than as a grammar.
- **The composer's Required Details section writes `required_slots`.** Each entry is authored in prose — a name, a phrase saying what it is, and whether it is required — and the same screen carries the guidance on what a Socratic comparison compares with and without a list.

Schema touchpoints. Reads `experiment.required_slots`, `experiment.scenario`, `experiment.system_prompt`, `experiment.socratic_version`, `experiment.score_judge_*`, `experiment_run.socratic_enabled`, and the

experiment's `document` rows through §7. Writes `chat_round.clarification`, `chat_round.retry_count`, and `chat_round.judge_tokens` / `judge_cost` for the gate's calls. Writes no row for a turned-back attempt. Rolled up to `run_result.retry_stats` by §6.

---

22. Sending an Email — Binding the Merge Tokens and Minting the Invite Link

Purpose. Turn a template plus a recipient into a sent message, giving every merge token a value and minting the one-time sign-in link the two invitation kinds carry. Every enrollee-facing kind also carries `{{app_url}}`, the installation's permanent address, so a recipient whose link has expired knows where to sign in. The token vocabulary is fixed per kind and enforced when the template is saved, so what remains is what happens at send time when a token has no value.

Inputs / outputs.

- Inputs: the `email_template` row (`kind`, `subject`, `body`, `variables`); the recipient's `profile`; the experiment or run the kind needs; the SMTP settings and sender address from `.env`; `INVITE_LINK_EXPIRY_HOURS`.
- Outputs: a sent message and an `audit_log` entry with `action` of `invite` or `notify`, or a refusal with nothing sent and the reason recorded.

Procedure.

```
send(template, recipient, context):
    values := bind(template.kind, recipient, context)      # data tokens only; see below

    # Only the tokens this template actually uses need a value. A token the kind
    # allows but the template never mentions has nothing to do with this send.
    for each token in tokens_used_by(template.subject, template.body):
        if values[token] is present:
            continue
        if token carries a link:                  # invite_link, run_link
            # A link is RESOLVED BY MINTING, not looked up. bind() never produces one,
            # so arriving here is the normal path for an invitation, not a fault.
            values[token] := mint_invite_link(recipient, template.kind)
            if minting failed:
                record the failure
                return REFUSED    # only a MINTING failure is fatal: an email whose
                                  # whole purpose is the link is worthless without one
        else:
            values[token] := fallback(token)      # recorded alongside the send

    text := substitute(template.subject and template.body, values)
    if any {{token}} survives in text:
        record the surviving token
        return REFUSED                            # never let one reach a recipient

    send text over SMTP from the configured sender
    write audit_log with action invite or notify
```

The link itself is not ours to make.

```
bind(kind, recipient, context):
    # Resolve every DATA token the kind allows, from rows this send already holds.
    # It calls no other service and mints nothing, which is why a link token is absent
    # from what it returns and is minted by the caller instead.
    return { app_url       : APP_URL,                   # the install's permanent address
             enrollee_name : recipient.username,
             inviter       : the experimenter for an enrollee invitation,
                             the installation name when that profile is gone,
             study         : context.experiment.enrollee_title,
             run           : context.run.name }         # one entry per data token

mint_invite_link(recipient, kind):
    return the link Supabase Auth generates for that address and kind
    # any previously issued link for the same address stops working
```

Key decisions & edge cases.

- **Supabase Auth mints the link; the platform embeds it.** Signing in already belongs to Supabase Auth, so expiry, single use and replay protection come from the service that implements them, and the platform stores no credential of its own. Self-hosting that service later carries this behavior along unchanged.
- **A link is good for 72 hours, and the authentication service is the authority.** `INVITE_LINK_EXPIRY_HOURS` defaults to 72 and is what the email states, but the token's real lifetime is a setting on the authentication service. **The two must be set to the same value**, since a mismatch has the email promising a validity the link does not have. Three days covers a cohort invited on a Friday afternoon.
- **The link is not the way back in.** An invitation carries a one-time link *and* `{{app_url}}`, the installation's permanent address. The link is a first-entry convenience that dies in 72 hours; the address is where the recipient signs in for the rest of the study. Every enrollee-facing template states it in text, not only behind the link.
- **`{{app_url}}` is bound from the `APP_URL` install setting** and is a cosmetic token in the sense that it never expires and never needs minting — but it is treated as required in the enrollee kinds, because an invitation that omits it is the failure above.
- **Nothing is sent to somebody who has left the pool.** A `run_ready` notice is skipped when the recipient's `profile.opted_out_at` is set (§28); mailing a study announcement to a person who withdrew is precisely what opting out is meant to stop.
- **Re-sending invalidates the earlier link.** Otherwise a live sign-in link sits in an old inbox message that nobody is tracking, which is a credential with no owner.
- **A cosmetic token falls back; a link token refuses.** `{{inviter}}` can genuinely have no value, since a profile can be hard-deleted and the authoring columns go null by design, so the invitation falls back to the installation's name and goes out slightly less personal. An email whose whole purpose is a link is worthless without one, so those refuse and leave a record.
- **An unbound token never reaches a recipient.** After substitution the text is scanned for a surviving `{{token}}`, and anything left refuses the send. A reader seeing "You have been invited by" with nothing after it is worse than either a fallback or a failure.
- **The refusal is recorded, not swallowed.** A send that does not happen leaves an entry naming the token that had no value, so an invitation that never arrived is diagnosable rather than silent.

Tunables (`.env` , with hard-coded defaults): `INVITE_LINK_EXPIRY_HOURS` (**72**) — what the email states, which must match the authentication service's own setting; `SMTP_PORT` (**587**) — the mail submission port; `SMTP_USER` and `SMTP_PASSWORD` — the relay credentials, where unset means the relay takes no authentication.

Schema touchpoints. Reads `email_template` (`kind` , `subject` , `body` , `variables`), `profile` (`email` , `username`), `experiment` (`enrollee_title`), and `experiment_run` for the run the notice concerns. Writes `audit_log` .

---

23. Chat History — What the Models Remember, and What Happens When It Will Not Fit

Purpose. Decide how much of an enrollee's earlier conversation is replayed to the models on each round, keep that identical across every member of the configuration, tell the enrollee when the room is running out, and record what was actually sent. It runs after §21 has accepted the request and §18 has produced the effective prompt, and its output is part of the assembled context §1 hands to the generators.

Why the smallest window governs. A configuration's members can have very different `context_window` figures, and every member must receive identical input or the comparison becomes one of how much each was told. The budget therefore comes from the smallest window among the members — the largest they all share — and that one assembled history goes to all of them.

Why there is no per-experiment turn count. `experiment.enable_chat_history` is a plain switch: memory is on or off, and its depth is a platform constant rather than a per-experiment setting. Holding the depth platform-wide keeps one limit in force everywhere, and it keeps memory from varying with `model_config` , which is itself a swept variable — an experiment-level depth would make one arm systematically shallower than another and read as a quality difference.

Inputs / outputs.

- Inputs: `experiment.enable_chat_history` ; the rounds of the enrollee's current session (`session_seq`) whose `rewound_at` is null; the configuration's members and their `model_catalog.context_window` ; the resolved system prompt (§10); the retrieved context (§7); the effective prompt; `CHAT_HISTORY_MAX_TOKENS` , `CHAT_OUTPUT_RESERVE` , `CONTEXT_ALERT_THRESHOLD` .
- Outputs: the ordered turns to replay, an alert state for the enrollee, and `chat_round.history_window` recording all of it.

Procedure.

```
assemble_history(session, experiment, config, system_prompt, source, prompt):
    if not experiment.enable_chat_history:
        return { turns := [], alert := none, window := null }   # every round self-contained

    # The largest window every member can accept — not the largest any one of them can.
    window        := min(m.context_window for m in config.members)
    fixed         := tokens(system_prompt) + tokens(source) + tokens(prompt)
    window_budget := window - fixed - CHAT_OUTPUT_RESERVE
    cap_budget    := CHAT_HISTORY_MAX_TOKENS
    budget        := min(cap_budget, window_budget)
    bound_by      := 'window' if window_budget < cap_budget else 'cap'

    if window_budget <= 0:
        fail the round with an error naming the smallest member, and return
```

```
        # the configuration cannot hold one prompt; the launch gate rejects this case

    # Newest first, so the oldest turns are the ones dropped.
    turns     := rounds of session where rewound_at is null, newest first
    available := count(turns)
    included  := empty list
    used      := 0
    for each turn in turns:
        cost := tokens(turn.raw_prompt) + tokens(turn.response)
        if used + cost > budget:
            break                       # everything older is dropped too
        prepend turn to included
        used := used + cost

    truncated := count(included) < available
    if truncated:
        alert := 'truncating'
    else if used >= budget * CONTEXT_ALERT_THRESHOLD:
        alert := 'approaching'
    else:
        alert := none

    return { turns := included, alert := alert, window := {
        turns_included := count(included),  turns_available := available,
        tokens_used    := used,             usable_budget   := budget,
        context_window := window,           bound_by        := bound_by,
        truncated      := truncated,        alert_shown     := alert is not none } }
```

Rewind, and starting fresh. Two enrollee controls act on the session, and they are not the same thing.

```
rewind(session, to_round):                  # forget everything after a chosen point
    now := current time
    for each r in session where r.round_seq > to_round.round_seq and r.rewound_at is null:
        r.rewound_at := now                 # dropped from history, kept in the transcript

clear_and_start_new(session):               # forget the whole conversation
    session.session_seq := session.session_seq + 1     # history assembles from empty
```

A **rewind** discards the tail and keeps the earlier context. **Clear and start new** discards all of it by beginning a new session. Neither deletes anything: a rewound round keeps its scores and still counts in `n_rounds` and every statistic built on rounds, because an enrollee rewinds when an exchange went badly and removing it would bias the run upward.

Key decisions & edge cases.

- **Truncation continues the session; it never stops it.** At the limit the oldest turns drop and the conversation carries on. Forgetting the early part of a long session is a legitimate thing to want, and a forced ending would be a worse experience than a shorter memory.

- **The enrollee is told, and told what it means.** At `CONTEXT_ALERT_THRESHOLD` of the budget they see a notice that the session is filling, recommending — not requiring — that they wrap up and start a new one, and naming the consequence: earlier parts of this conversation will stop being remembered. It is a recommendation because only the enrollee knows whether the early context still matters.
- **The round records whether the enrollee was warned.** `chat_round.history_window.alert_shown` carries it, and §6 rolls the count into `run_result.history_stats`. A warned enrollee may write more tersely, wrap up, or start over, and that behavior change lands only on enrollees with long sessions — so recording it turns an uncontrolled variable into a measured one.
- **The judge sees exactly what the generators saw.** §4 is handed the same assembled turns. Scoring "Shorter." without the antecedent measures the answer against a prompt that does not explain it, and relevance would be wrong in a way that looks like a model failure.
- **History is the enrollee's own words.** `raw_prompt` is replayed, not the enhanced rewrite, so the conversation the model sees is the one that happened. §18 then rewrites only the current turn.
- **Turned-back attempts are not turns.** A request §21 refused produced no answer and no round, so there is nothing to replay.
- **Socratic mode does not require chat history.** §30's eliciting happens inside §21's gate, which holds an exchange's attempts in its own loop rather than in the replayed history, so the two settings are independent and any combination of them is valid.
- **A round's cost rises with its `round_seq`,** because each round replays more history than the last until the budget is reached, and history is input tokens. Per-round cost is therefore not comparable between round 2 and round 30 of the same session, and `round_seq` is the covariate that makes the difference visible.
- **A configuration too small for one prompt fails at launch, not at round time.** `window_budget <= 0` means the system prompt, retrieved context and one message do not fit the smallest member, which is a broken configuration regardless of conversation length.

Tunables (`.env`, each with the hard-coded default shown): `CHAT_HISTORY_MAX_TOKENS` (**16384**) — the platform depth cap; `CHAT_OUTPUT_RESERVE` (**4096**) — tokens held back for the answer; `CONTEXT_ALERT_THRESHOLD` (**0.80**) — the fraction of the budget at which the enrollee is warned.

Schema touchpoints. Reads `experiment.enable_chat_history`, `chat_round` (`session_seq`, `round_seq`, `raw_prompt`, `response`, `rewound_at`), and `model_catalog.context_window` through `model_config_model`. Writes `chat_round.history_window` and `chat_round.rewound_at`; rolled up to `run_result.history_stats` by §6.

---

24. Bulk Import — Parse, Validate, Then Insert What Survives

Purpose. Turn an uploaded CSV, XLSX or JSON file into rows of one table, reporting every row it could not accept instead of failing the whole file. It is offered on the list screens that curate reusable definitions — enrollees, experiments, nudges — and takes the same path for each.

Inputs / outputs.

- Inputs: the uploaded file; the target table; the acting user; the target's required columns, enum domains and natural key.
- Outputs: the inserted rows, a per-row rejection report, and one `audit_log` entry.

Procedure.

```
import(file, target, actor):
    rows := parse(file)                      # csv / xlsx / json  ->  list of maps
    if rows is empty:            return REJECT('the file has no rows')
    if count(rows) > IMPORT_MAX_ROWS:  return REJECT('too many rows')

    accepted := empty list
    rejected := empty list
    seen     := empty set                    # natural keys seen earlier in THIS file
    for each row, line in rows:
        errs := validate(row, target)        # required present, types, enum domains,
                                             # referenced names resolve to real rows
        key  := natural_key(row, target)     # email / name — the table's unique key
        if key in seen:
            append 'duplicated earlier in this file' to errs
        else if exists(target, key):
            append 'already exists' to errs   # import never updates; see below

        if errs is empty:
            add key to seen
            append row to accepted
        else:
            append { line, errs } to rejected

    if accepted is empty:
        return REPORT(accepted := [], rejected)      # nothing is written

    begin transaction
        insert every row in accepted
        write one audit_log entry: the file name, the counts, and the rejected lines
    commit

    return REPORT(accepted, rejected)
```

Key decisions & edge cases.

- **Every row is judged before any row is written.** Validation is a separate pass, so the report is complete and the insert is a single decision rather than a partial walk that stops at the first bad row.
- **Partial acceptance, atomic insert.** A 500-row file with three bad rows imports 497 — failing all of it over three typos would be useless. But the accepted set goes in one transaction, so a failure mid-insert leaves nothing behind.
- **Import never updates.** An existing natural key is a rejection, not an upsert. Definitions are immutable while runs reference them, so a silent overwrite could redefine a study that finished runs belong to. Changing an existing row is an edit or a clone, both of which go through the rules that protect it.
- **References are given by name, not by id.** A spreadsheet author cannot type a uuid, so a referenced cohort or model is resolved from its unique name and an unresolved name is a row error.
- **The database is still the authority on uniqueness.** The pre-check exists to produce a readable report; the unique constraint is what actually guarantees it, and a concurrent import that wins the race turns into a rejected row rather than a

duplicate.

- **One audit entry per import,** naming the file and the counts — not one per row, which would bury the trail.

Tunables (`.env` , with hard-coded defaults): `IMPORT_MAX_ROWS` (**5000**) — the row count above which a file is rejected before any validation; `IMPORT_MAX_BYTES` (**10485760**, ten megabytes) — the upload size ceiling.

Schema touchpoints. Writes the target table and `audit_log` . Reads the target's unique indexes and any table it resolves a name against.

---

25. Cloning a Definition — The Sanctioned Way to Change a Frozen Frame

Purpose. Produce an independent copy of an `experiment` , `model_config` , `cohort` or `nudge` , so a definition that is locked because runs reference it can still be evolved. Cloning is what the interface offers in place of editing a frozen record.

A clone is a head start, not a duplicate. It carries the child rows that *constitute* the definition — a configuration's member models, an experiment's attached corpus — and not the ones that merely *populate* it, which is why a cloned cohort arrives with an empty roster.

Inputs / outputs.

- Inputs: the source row; the acting user.
- Outputs: a new row with a unique name, its copied children, and an `audit_log` entry.

Procedure.

```
clone(source, actor):
    copy := every substantive column of source          # prompts, config, notes, flags
    copy.name        := unique_name(source.name)        # 'X (copy)', then '(copy 2)', …
    copy.created_by  := actor                           # authorship of the COPY
    copy.owner_id    := actor
    copy.version     := 1
    copy.id                          := a fresh UUID          # database defaults supply
    copy.created_at, copy.updated_at := now()                 #   all three on insert

    begin transaction
        insert copy
        # Each constituting child is a JUNCTION row. The clone gets its own junction rows,
        # pointing at the SAME documents, catalog models and people — nothing downstream is
        # duplicated, only the associations to it.
        for each child in constituting_children(source):
            # experiment   -> experiment_context_file   -> the same document rows
            # model_config -> model_config_model        -> the same model_catalog rows
            # cohort       -> cohort_member             -> the same profile rows
            # nudge        -> (has no children)
            insert child with its parent key set to copy.id
        write audit_log naming source and copy
    commit
    return copy
```

Key decisions & edge cases.

- **History is not copied.** Runs, results and rounds belong to the original. A clone is a fresh frame with no runs, which is exactly why it is unfrozen and editable — and why cloning is the answer to "I need to change a study that has already run."
- **Ownership resets to whoever cloned it.** A peer starting from somebody else's design owns their copy; inheriting the source's owner would leave the cloner unable to edit the thing they just made.
- **The name is made unique, not rejected.** Names carry a unique constraint, so the copy takes a suffix and the user renames it if they like. Refusing the clone over a name collision would be a pointless stop.
- **Children that *constitute* the definition are copied; children that *populate* it are not.** A `model_config` without its `model_config_model` rows is not a configuration at all. An `experiment` normally holds its corpus constant while the prompt, rubric or scoring varies, so its `experiment_context_file` rows come along; detaching one afterward is trivial, re-attaching a corpus from memory is not.
- **References are not copied, only re-pointed.** An experiment's clone attaches the *same* `document` rows through new `experiment_context_file` rows. The documents are shared library items; duplicating them would fork a corpus that is meant to be one thing.
- **A clone is comparable to nothing.** Its runs form a new comparison; scores from the original's runs are not comparable to the clone's unless the scoring config happens to match, and even then the platform only claims comparability *within* an experiment.

Schema touchpoints. Reads and writes `experiment`, `model_config`, `cohort`, `nudge` and their child tables (`experiment_context_file`, `model_config_model`, `cohort_member`). Writes `audit_log`.

---

26. Ownership Transfer and Hand-Off on Deactivation

Purpose. Move `owner_id` on one object, or move everything a departing person owns and disable their account in the same step. It exists because `owner_id` governs who may edit a definition, so deactivating an owner without reassigning their work would leave that work uneditable by anyone.

This is object ownership, and is unrelated to the *install* owner, which is reconciled from `.env` at boot (§14).

Inputs / outputs.

- Inputs: the object or the departing user; the receiving user; the acting user, who must be an admin.
- Outputs: updated `owner_id` values, possibly `profile.enabled = false`, and an `audit_log` entry carrying both states.

Procedure.

```
transfer_one(object, to_user, actor):
    require actor.role = 'admin'                      # the owner cannot hand off their own
    require to_user.enabled and to_user.role in { 'experimenter', 'admin' }
    require no run in state 'running' or 'paused' references object
    object.owner_id := to_user.id
    write audit_log with before_state and after_state

hand_off_all(from_user, to_user, actor):
    require actor.role = 'admin'
```

```
    require to_user ≠ from_user and to_user.enabled
    require from_user owns no run in state 'running' or 'paused'

    begin transaction
        for each t in [experiment, nudge, model_config, cohort, document]:
            update t set owner_id := to_user.id where owner_id = from_user.id
        from_user.enabled := false
        write one audit_log entry naming both people and the count moved per type
    commit
```

Key decisions & edge cases.

- **Reassign and disable are one transaction.** Otherwise a failure between them leaves either an orphaned set of definitions or a disabled account still owning live work.
- **Live runs block it.** Reassigning a definition that a running study depends on is refused; wait for the run to finish or abort it first.
- `created_by` **never moves.** It records who authored the row, which is history. `owner_id` records who may edit it, which is permission. Rewriting authorship to tidy a hand-off would falsify the record.
- **The account is disabled, never deleted.** `enabled = false` blocks sign-in and keeps every row the person produced correctly attributed, which is the same rule the users screen applies everywhere.
- **An admin may transfer to self,** which is the usual outcome when somebody leaves and no successor is named yet.
- **Only an admin does this.** An owner cannot hand their own work away, so ownership cannot be used to quietly shed responsibility for a study.

Schema touchpoints. Writes `owner_id` on `experiment`, `nudge`, `model_config`, `cohort`, `document`; writes `profile.enabled`; writes `audit_log`.

---

27. Install Reset and Demo Reseed

Purpose. Empty every data table, optionally reloading the shipped demonstration dataset. It is an operator action for preparing a new install or resetting a demonstration, not a backup or recovery mechanism.

Inputs / outputs.

- Inputs: the mode, `empty` or `demo`; the acting admin; a typed confirmation.
- Outputs: emptied tables, optionally the seed fixture loaded, and an `audit_log` entry that survives the wipe.

Procedure.

```
reset(mode, actor):
    require actor.role = 'admin'
    require the typed confirmation matches the install name    # destructive, not undoable
    require no run in state 'running' or 'paused'              # never wipe under a live run

    begin transaction
        for each table in DELETION_ORDER:      # children strictly before parents
            delete every row
```

```
        # DELETION_ORDER runs leaf-first, and is exactly:
        #   chat_round_candidate, chat_round, run_enrollment, message_recipient, message,
        #   run_result, experiment_run, experiment_context_file, experiment, nudge,
        #   model_config_model, model_config, cohort_member, cohort,
        #   embedding, embedding_job, document,
        #   nl_query, issue, audit_log, then profile last
        keep the install owner's profile row
        keep every embedding_ingest_registry row
        if mode = 'demo':
            load the shipped seed fixture in the same order, parents first
        write audit_log recording the reset and its mode
    commit
```

Key decisions & edge cases.

- **Explicit deletion order, not `TRUNCATE … CASCADE`.** Several foreign keys are `RESTRICT` precisely so a definition cannot be deleted out from under a run; a cascade would defeat that protection wholesale, and a forgotten table would silently take rows with it. An explicit leaf-first order fails loudly instead.
- **The install owner's `profile` survives.** Deleting it would lock every human out of the install. §14 re-asserts that row from `OWNER_EMAIL` on the next boot regardless, so keeping it also keeps the two consistent.
- **`embedding_ingest_registry` survives too, for the same reason.** It is deploy-time configuration seeded by the schema script, not data the install produced, and nothing at runtime writes it back. Wiping it would leave the enqueue trigger correctly declining to enqueue for every source, so the semantic index would quietly never fill again — no error, just nothing happening. A reset clears what the install made, not how it was set up.
- **Refused while a run is live.** Wiping under a running study would leave enrollees mid-conversation with their run gone.
- **The audit entry is written inside the transaction, after the wipe,** so the reset itself is the first row of the new trail rather than something the wipe erased.
- **The demo fixture is a shipped, versioned artifact,** loaded parents-first. It is a fixture, not a migration: it never runs on an install that holds real data except through this deliberate action.
- **This is not a backup.** There is no undo, and the confirmation phrase exists to make that unmissable.

Schema touchpoints. Deletes from every data table; preserves one `profile` row and every `embedding_ingest_registry` row; writes `audit_log`.

---

28. Enrollee Opt-Out — Leaving the Pool

Purpose. Let an enrollee withdraw themselves from the study pool, ending every run they are currently in, stopping further invitations, and keeping everything they already produced. It is theirs to trigger; staff cannot opt somebody out on their behalf, and `enabled` is a different thing.

Why it is pool-wide and not per study. An enrollee accepts an invitation to the pool, not to a named study — studies reach them afterward through cohort membership, and §12 writes their enrollment at launch without asking. The switch therefore sits at the level the consent did.

Inputs / outputs.

- Inputs: the acting enrollee (never another user); a typed confirmation.
- Outputs: `profile.opted_out_at`, a `withdrawn_at` on each live enrollment, and an `audit_log` entry.

Procedure.

```
opt_out(actor):
    require actor.role = 'enrollee'          # staff have no equivalent; see below
    require actor.opted_out_at is null
    require the confirmation was given       # a consequential, though reversible, act

    now := current time
    begin transaction
        actor.opted_out_at := now
        for each e in run_enrollment where e.enrollee_id = actor.id
                                       and e.withdrawn_at is null:
            e.withdrawn_at := now            # every live run ends at the same instant
        write audit_log: actor is the enrollee, naming the runs ended
    commit
    # cohort_member rows are NOT deleted — §12 filters instead

opt_back_in(actor):
    require actor.opted_out_at is not null
    actor.opted_out_at := null               # eligible for FUTURE runs only
    write audit_log
```

Key decisions & edge cases.

- **Opting out is not being switched off.** `enabled` is a suspension somebody with more privilege applies; `opted_out_at` is a decision the participant makes about their own involvement. Both can hold at once, neither implies the other, and conflating them would make "I left" indistinguishable from "I was removed" in the record.
- **They can still sign in.** An opted-out enrollee reaches a My-studies screen that says so and offers to opt back in, plus read-only access to summaries of studies they finished. Locking them out would punish the decision, strand their completed work, and leave them no way to change their mind.
- **Cohort rows are filtered, never deleted.** §12 skips opted-out members when it resolves a cohort, so a later launch cannot quietly re-enroll them. Deleting the `cohort_member` rows instead would rewrite who a cohort held at the time, falsifying the history of runs that already used it.
- **Their data stays and still counts.** Rounds already answered keep their scores and remain in the statistics of the runs they belong to — the work happened, and removing it would bias those runs upward, exactly as dropping a rewound round would.
- **Reversible, but not retroactive.** Opting back in clears the column and makes them eligible for future runs. It does not restore the enrollments it ended, because those runs have moved on and re-entering one mid-flight would put a participant into a condition partway through.
- **The mail stops.** §22 skips `run_ready` for an opted-out recipient. Nothing else is suppressed, because nothing else is sent to enrollees.
- **Staff have no opt-out.** An experimenter or admin who wants to leave is deactivated by an admin, with their work handed off first (§26). The two flows solve different problems and share no code.
- **A run mid-conversation ends cleanly.** The enrollee's current chat closes on the next request; rounds already written are untouched, and the run's `n_enrollees` still counts them, since they did take part.

Schema touchpoints. Writes `profile.opted_out_at`, `run_enrollment.withdrawn_at`, and `audit_log`. Read by §12 (cohort resolution) and §22 (suppressing `run_ready`).

---

29. Sessions — Signed in Until You Sign Out

Purpose. State what happens to a signed-in person over time. The answer is deliberately short: **nothing**. There is no session timeout, no idle expiry and no maximum age. A session ends when the person signs out, when an admin disables the account, or when the enrollee leaves the pool — never on a clock.

Why a timeout would not buy security here. The habit comes from systems where the token *is* the authorization, so a stolen token stays privileged until it expires. Here the token proves only identity; what the caller may do is re-read from the database on every query through row-level security, so a role change takes effect on their next query.

And it would cost something real. Expiring a session mid-conversation drops an enrollee out of a study they are part-way through and loses the round they were composing. Timeouts also train people to re-authenticate reflexively on demand, which is the reflex phishing depends on.

What actually protects the session.

- **A short-lived access token, refreshed silently.** The authentication service issues an access token with a lifetime of about an hour and a refresh token beside it; the client renews in the background. That expiry is an internal refresh cadence, **not** a user-visible timeout, and it is deliberately not exposed as a `.env` knob — turning it into a setting would invite somebody to mistake it for a session policy.
- **Refresh-token rotation with reuse detection.** Each refresh issues a new token and retires the old one. A stolen token replayed after rotation is detected as reuse and the whole token family is revoked — a sharper response than expiry, because it triggers on evidence of theft rather than on the calendar.
- **Authorization re-read per request.** `role`, `enabled` and `opted_out_at` are read from the database on every query. **They must never be cached into the token as claims** — a claim is a snapshot, and snapshotting authorization into a long-lived credential is the one change that would make indefinite sessions genuinely unsafe. This is the single implementation rule this section exists to state.
- **Sign-out, on every screen.** It is the user's own control, and the answer for a shared or public computer. Signing out everywhere revokes the refresh-token family for all devices, which is the answer for a lost one.

Key decisions & edge cases.

- **No `.env` variable governs session length,** because there is no policy to tune. A setting here would imply the platform expires sessions, which it does not.
- **A disabled or opted-out person may still hold a valid token,** and that is fine: the next request they make is refused on the database's authority, not the token's.
- **An enrollee returning after weeks signs in exactly as before** — or is still signed in. The one-time code path costs an inbox round-trip only when the session has genuinely ended, which now means only after an explicit sign-out.

Schema touchpoints. Reads `profile.role`, `profile.enabled`, `profile.opted_out_at` on every request through row-level security. Writes nothing — the session lives entirely in the authentication service.

---

30. Socratic Mode — How the Gate Replies

Purpose. Choose how §21's sufficiency gate answers an enrollee whose message it has stopped, and make that choice a swept variable so a series can measure which way of repairing an incomplete request produces better answers, and in fewer attempts. It is switched on per run by `experiment_run.socratic_enabled`; both phrasings live in `experiment.socratic_version` and never vary within an experiment.

The two styles.

- **On — the judge asks.** It asks for one missing piece, or for a small group that only makes sense together such as the start and end of a date range. It does not list everything that is missing. The enrollee answers, the gate tests again, and the request is assembled through dialogue.
- **Off — the judge asks for a rewrite.** It names everything missing in one reply, and asks the enrollee to rewrite the request so that it carries all of it and to send the rewritten request as a single self-contained message.

Why the judge does the asking, and not the models under test. A `model_config` may hold several members, and under every fan-out combine method all of them answer each round. Were the models doing the asking, one member could ask a question while another answered, and §1's reduce step would have to merge, peer-score or cluster a question against an answer — which none of the methods can do coherently. `chat_round.declined` is one verdict per round, so a mixed round would have no single value to record either. Putting the asking on the one trusted judge, ahead of any member call, removes the problem instead of special-casing it: every member answers the same well-formed request, and the fairness rule that a round's members receive identical input is untouched.

The cost is that a model's own aptitude for asking a good question is not measurable here. That is a real difference between models, and it is out of scope.

Inputs / outputs.

- Inputs: `experiment_run.socratic_enabled`; `experiment.socratic_version`, naming a versioned instructions artifact in the middle-tier registry; `experiment.required_slots`, which decides whether the gate has a checklist to work from.
- Outputs: no output of its own. The switch selects a branch inside §21, whose effects are the reply the enrollee sees, the `raw_prompt` the models receive, and `chat_round.retry_count`.

Procedure.

```
# The switch is read in exactly two places, both inside §21's gate.
#
#   1. Detection, in one case only. With no required_slots declared there is no
#      checklist, so a run that is eliciting has the judge read the request itself
#      (§21 test 3), and a run that is not has no gate at all.
#
#   2. The reply, always. The style selects one of four bundled phrasings, one per
#      (style, outcome) pair, and the judge composes the reply under it.
#
# Nothing is appended to any model's system prompt (§10), and no model under test is
# called until a message has passed the gate.
```

What the comparison measures. Hold the experiment, the model configuration, the cohort and the nudge steady, and run once with the switch on and once with it off. Two figures move. `composite_mean` moves because one arm's models receive a request assembled through dialogue while the other's receive one the enrollee composed in a single piece.

`run_result.retry_stats` moves because it counts how many attempts each arm needed before an answer existed, which is the direct measure of the manipulation. Judge spend rises on the asking arm, and `total_judge_cost` is reported apart from `total_generator_cost`, so that extra spend is visible rather than confounded with generation.

Key decisions & edge cases.

- **A switch, not prompt text.** The experimenter never edits wording to run this study. Two runs of one experiment differ in which of two frozen phrasings the judge uses, and in nothing else.
- **The launch form refuses the switch when the experiment names no wording.** With `experiment.socratic_version` null the judge has nothing to ask with. §21 already requires a version from any experiment whose gate can fire on its own, so this check catches the remaining case: an experiment with neither context documents nor `required_slots`, which is allowed to leave the version null and is exactly the experiment where test 3 would otherwise run. It has to be checked at launch, since the experiment locks on its first run and a version omitted then can never be added.
- **Both phrasings are versioned together, in one artifact.** The Socratic wording is being compared against the non-Socratic wording, so freezing one and hard-coding the other would leave the control arm free to change between releases with nothing in the experiment's record to show that it had. One version name pins the whole reply policy.
- **The switch needs something to fire on, and what it compares depends on that.** With `required_slots` declared, or context documents attached, both settings have a working gate and the comparison is between two ways of repairing a request. With neither, only the on setting has a gate, and the comparison is between eliciting and not eliciting at all. Both are legitimate studies, but they answer different questions, so the compose screen names which shape the experiment is in.
- **Chat memory is not required.** The gate holds a dialogue's attempts in its own loop, outside the replayed history, so this works with `experiment.enable_chat_history` off. Memory and eliciting are independent settings.
- **An exchange that never converges is possible under either setting.** The enrollee supplies what is asked, or leaves; there is no attempt cap. It is likelier under the rewrite style, since nothing already supplied is carried forward, and it shows as an abandoned session rather than as a bad answer.
- **The models never ask.** Nothing in the resolved system prompt tells them to. A model that asks anyway has produced its answer to that round and is scored as such — the round is not treated as a question.

Schema touchpoints. Reads `experiment_run.socratic_enabled`, `experiment.socratic_version`, `experiment.required_slots`. Drives §21, which writes `chat_round.clarification` and `chat_round.retry_count`. Aggregated into `run_result.retry_stats` by §6.

---

31. Prompt Decomposition — Splitting a Request Across Models

Status — a candidate for a future version, not part of V1. Nothing described here is built and no schema supports it. It is written down so the shape is settled before anyone starts, and because the reasoning behind two of its constraints is easy to get wrong.

Purpose. Answer a request that carries several distinct asks by splitting it into sub-prompts, routing each to the member that suits it, and having a master combine the results. It is pure prompting throughout: no tool use, no external data sources, no state outside the round.

Shape. This is a **combine method**, not an agent framework. Structurally it is close to `synthesize` (§1), which already fans out to every member and already ends in a merge call. The two new ideas are that the sub-prompts *differ* from one another and that something has to produce them.

The procedure is **recursive**: a node the planner emits may itself be compound, in which case it is split again rather than answered, and only a leaf node is routed to a model. A node's `depends_on` edges make a plan a **directed acyclic graph** — a node may feed several others and draw on several — and nothing can depend on itself, because a node may only reference nodes the same planner call already emitted. Expanding an intermediate node substitutes its own graph in place of that node, so the composition stays acyclic and the whole execution path is one directed acyclic graph. `dependency_waves` is a topological layering of that graph: the nodes in a wave are independent of each other, so a wave runs in parallel and the next one waits for it.

Procedure — a sketch, not a specification.

```
# `inherited` carries the answers this request already depends on. The top-level call
# has none; a recursive call is given the inputs of the node it was expanded from, so a
# sub-plan can see what its parent drew on.
decompose(request, config, experiment, depth = 0, inherited = {}):

    # Model-blind: the planner sees the request and nothing about the configuration.
    plan := split(request, experiment.planner_version, temperature = 0)

    # Two ways to stop splitting: the planner found nothing to split, or the recursion
    # has gone as deep as it may. Either way this request is answered whole.
    if count(plan.nodes) < 2 or depth >= DECOMPOSE_MAX_DEPTH:
        member := route(request, config.members, experiment)
        return generate(member, request, inherited)

    answers := {}                            # node -> the one answer that node produced
    for wave in dependency_waves(plan):      # one topological layer of the plan's DAG
        for node in wave in parallel:        # the nodes in a wave are independent
            inputs := inherited + answers[node.depends_on]
            if node is a leaf:
                # A leaf is answered: routed on its own merits, then generated.
                member        := route(node.prompt, config.members, experiment)   # §19
                answers[node] := generate(member, node.prompt, inputs)
            else:
                # Still compound, so split it the same way, handing down what it depends
                # on. The call returns ONE answer, already merged at its own level, so a
                # sub-plan reaches its parent as a single node's worth of result.
                answers[node] := decompose(node.prompt, config, experiment,
                                           depth + 1, inputs)

    # The merge IS a node: the DAG's terminal node depends on every other, carries the
    # combining instruction, and was routed and generated by the loop above like any
    # leaf. Returning its answer is what makes every invocation — the top-level call and
    # each recursive one alike — hand back a single merged result.
    return answers[plan.terminal]
```

Key decisions & edge cases.

- **The planner is model-blind.** It splits by task structure and never sees which models are configured. One that could see them would split the same request differently per configuration, and `model_config` is a swept variable, so two arms

would answer different questions and no quality difference would trace to the models.

- **The master is the plan's last node, routed like any other.** Under `synthesize` the merger combines the same prompt answered several ways, so there is no merge instruction with a topic of its own. Here the merge step has its own instruction, which carries a topic and therefore routes.
- **It sharpens the router rather than merely adding a stage.** A compound request embeds to the midpoint of its parts and matches no capability sentence cleanly, which is the case §19 reads as a tie and pays the judge to break; each split part matches decisively on the free stage. If decomposition works, the escalation rate should fall, which `chat_round.moe_routing.stage` already records.
- **Cost is roughly neutral; latency is not.** Each sub-answer is billed once as output and again as input to every dependent node, cancelling most of what cheaper members save, and the plan sequences into several waves. The case rests on answer quality rather than spend, so a layer choosing between strategies has to weigh quality.
- **Errors propagate silently.** Nothing checks the first node before later nodes build on it, and the master presents an inherited mistake in the same confident register as a sound one. A single-model answer is at least wrong coherently.
- **Reproducibility needs a frozen planner.** Temperature 0, and the planner instructions held as a versioned artifact, exactly as `socratic_version` and `prompt_enhancer_version` are. What must be uniform across a comparison is the *policy*, not the output: one frozen planner applied identically to every enrollee is a fixed condition even though it emits a different plan per request.

What it would need. A `combine_method` value; a planner-instructions version on the experiment; a `DECOMPOSE_MAX_DEPTH` ceiling, since a recursive planner without one can split without end; per-node storage for the sub-prompts and their answers, since `chat_round_candidate` today assumes every candidate answered the *same* prompt. None of that exists.

Inspiration, and its limit. The design takes its shape from query optimization and access path selection in relational databases: decide before executing, and price a plan from statistics gathered over past executions rather than by trial run. The structural borrowing is that an optimizer consults both a **catalog** and **statistics** — statistics say which shape of plan has paid off, the catalog says which model applies to a part, which is what `model_catalog.expertise` holds. What carries across is the principle, not the machinery: language has no algebra of legal rewrites to enumerate, and no plan can be priced without generating it.

Appendix — `.env` Variables

Every tunable in this document follows one convention: a reasonable default is hard-coded and the `.env` parameter below optionally overrides it. Only the variables marked **mandatory** must be present for an installation to work; everything else can be left unset.

Variable	Required	Example	Default if unset
`DB_SCHEMA`	mandatory	`chat_maestro`	—
`DATABASE_URL`	mandatory	`postgresql://…/postgres`	—
`SUPABASE_URL`	mandatory	`https://abcd.supabase.co`	—
`SUPABASE_SERVICE_KEY`	mandatory	`eyJhbGciOi…`	—
`OWNER_EMAIL`	mandatory	`owner@example.org`	— (§14)

Variable	Required	Example	Default if unset
`APP_URL`	mandatory	`https://study.example.org`	— (the address invitations tell enrollees to return to, §22)
`SMTP_HOST`	mandatory	`smtp.example.org`	—
`SMTP_PORT`	optional	`587`	`587`
`SMTP_USER`	optional	`mailer@example.org`	unset (no SMTP auth)
`SMTP_PASSWORD`	optional	`…`	unset (no SMTP auth)
`MAIL_FROM`	mandatory	`ChatMaestro <no-reply@example.org>`	—
`INVITE_LINK_EXPIRY_HOURS`	optional	`72`	`72` (§22)
`ASK_MODEL`	mandatory	`anthropic/claude-opus-4-5`	— (§8)
`ASK_FAST_MODEL`	optional	`anthropic/claude-haiku-4-5`	falls back to `ASK_MODEL`
`ASK_MATCH_THRESHOLD`	optional	`0.85`	`0.85` (§8)
`ASK_ROW_CAP`	optional	`100`	`100` (§8)
`ASK_PACK_MODE`	optional	`full`	`full` — `slice` uses core + per-question slice (§8)
`EMBEDDING_MODEL`	mandatory	`voyage/voyage-3`	— (§7)
`RAG_TOP_K`	optional	`5`	`5` (§7)
`RAG_MIN_SIMILARITY`	optional	`0.25`	`0.25` (§7)
`CHUNK_TARGET_TOKENS`	optional	`800`	`800` (§7)
`MOE_MIN_SIMILARITY`	optional	`0.15`	`0.15` (§19)
`MOE_TIE_MARGIN`	optional	`0.05`	`0.05` (§19)
`MOE_ESCALATE`	optional	`true`	`true` (§19)
`ENHANCER_MAX_RATIO`	optional	`6`	`6` (§18)
`ENHANCER_MIN_SIMILARITY`	optional	`0.60`	`0.60` (§18)
`ENHANCER_HISTORY_TURNS`	optional	`4`	`4` (§18)
`ENHANCER_TIMEOUT_MS`	optional	`20000`	`20000` (§18)
`CONSENSUS_SIMILARITY`	optional	`0.90`	`0.90` (§1)
`CHAT_HISTORY_MAX_TOKENS`	optional	`16384`	`16384` (§23)

Variable	Required	Example	Default if unset
`CHAT_OUTPUT_RESERVE`	optional	`4096`	`4096` (§23)
`CONTEXT_ALERT_THRESHOLD`	optional	`0.80`	`0.80` (§23)
`PRESENCE_WINDOW_MINUTES`	optional	`5`	`5`, or the Realtime layer's own presence timeout when it exposes one (§17)
`IMPORT_MAX_ROWS`	optional	`5000`	`5000` (§24)
`IMPORT_MAX_BYTES`	optional	`10485760`	`10485760` (§24)

A note on secrets. The four mandatory credentials — the database URL, the Supabase service key, the SMTP password and the provider API keys the LLM router reads — are the only values here that must never reach the browser or a repository. Everything else is configuration, not secret.