Analyze the Results

Experimenter Read the scores, compare runs, check cost (dollars split between doer and judge), ask or reuse saved questions, and export.

Screens Results `results-analytics.html` | See-all `see-all-responses.html` | Ask panel `ask-surface.html`

Reads the run scorecards `run_result` | each exchange `chat_round` | the runs `experiment_run` | saved questions `nl_query`

Key fields headline score `composite_mean · composite_median` | per-factor five-number `factor_stats` | spread `IQR = q3 − q1` | effort `iterations_* · latency_ms_*` | tokens `total_tokens` | $ split `total_generator_cost (doer) · total_judge_cost` | per model `model_perf`

Ask one-click saved buttons `nl_query` | save a new question `status = draft → approved`

1 · Given / When / Then

Human wording first; the exact table/field in `mono`.

GIVEN	At least one run has reached `state = done`, so its scorecard (`run_result`) has been built.
WHEN	On the **Results Screen** the experimenter reads the run's scores: the composite headline (`composite_mean` and `composite_median`) and the per-factor five-number summaries in `factor_stats` (min, q1, median, q3, max). They compare runs across the series and check cost and effort: the token count (`total_tokens`) and the US dollar spend, split between the doer models and the judge (`total_generator_cost` versus `total_judge_cost`), plus passes and latency. To dig deeper they open the **Ask** panel, either clicking a saved question (a one-click `nl_query` button) or typing a new one in plain language. The answer comes back as a narrative plus zero or more result tables, each viewable as a downloadable table or chart. A useful new question can be saved for the team. Anything on screen exports to a file.
THEN	Reading and comparing write nothing, because analysis is read-only. The cross-run comparison is a query-time grouping of the series' `run_result` rows on the two variables held constant, varying the third — the nudge, the model set-up, or the cohort — with readable labels joined straight from the immutable definitions. There is deliberately no stored “series” object. Cost is reported two ways that add up: `total_generator_cost` for the doer models (the variable under test) and `total_judge_cost` for the judge overhead, so a fast-but-expensive set-up cannot hide. A plain-language question is matched against saved questions or translated to SQL by an AI model, then run read-only, with the generated SQL always shown next to the answer. Only an explicit Save writes an `nl_query` row. **reads** `run_result`, `chat_round`, `experiment_run`, `nl_query` **writes** `nl_query` (only on Save)

Variations and exceptions

IF…	THEN…
Fewer than two runs of the series have finished	There is nothing to compare; the comparison view waits for a second `done` run.

IF…	THEN…
A typed question would modify data	Refused: the plain-language path compiles and runs only reads, and the SQL is shown beside the answer (*Ask a Question in Plain Language*).

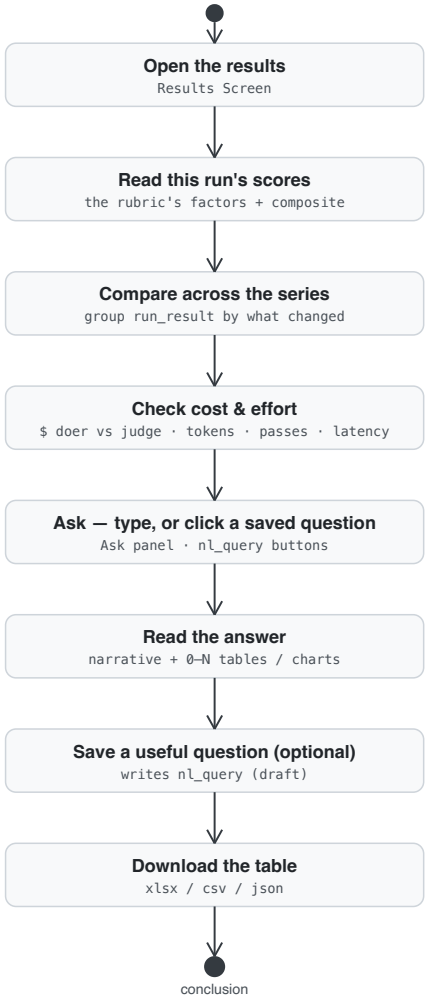
Reading the numbers

1. **Mean and median are both enrollee-unit.** Every score is computed per person (the mean of per-enrollee means), so one chatty enrollee cannot dominate. Read `composite_mean` alongside `composite_median`: when they diverge, the per-enrollee distribution is skewed, which is itself a finding.
2. **Cost is split on purpose.** `total_generator_cost` (the doer models under test) is kept separate from `total_judge_cost` (the one trusted judge's overhead, for both picking winners and grading), so the measurement cost is visible and cannot masquerade as answer cost.
3. **Tokens and dollars can point different ways.** A model that is cheap per token but needs many self-improvement passes can still be the pricier set-up, which is why both the token count and the US dollar figure are shown.
4. **Quantiles cannot be re-pooled.** A median or quartile across runs must be recomputed from the raw exchanges; you cannot average stored medians.
5. **Memory depth can differ between arms.** When the study has chat history on, the depth is bounded by the smallest `model_catalog.context_window` in each run's configuration — and the configuration is a swept variable, so one arm can have systematically shallower memory than another. Read alone that looks like a quality difference. `run_result.history_stats` puts the figures beside the scores: the smallest window in force, the rounds truncated, the rounds where the enrollee was alerted, the rewinds, and the turns actually replayed.
6. **Refusals read separately from low scores.** A response that declined the request rather than attempting it is counted from `chat_round.declined`, split into `appropriate` and `unwarranted`. An appropriate refusal takes relevance to N/A rather than scoring it badly, so a model that correctly declines is not ranked below one that answers anyway; an unwarranted refusal stays scored. The scorecard freezes the tally in `run_result.declined_dist`, so the refusal rate still reads after a cleanup has removed the rounds it came from. Requests the sufficiency gate turned back are not in it — those never became rounds.
7. **Scores compare only within an experiment.** Comparison across an experiment's runs is valid because the same trusted judge grades every one identically, on one frozen rubric; runs of different experiments are not comparable.

Acceptance test — *Given* two or more finished runs of one study differing in a single parameter, *When* the experimenter opens the Results Screen, *Then* they see per-factor and composite scores per run, a by-parameter comparison of `composite_mean`, the effort/cost stats, and can export — with no rows written.

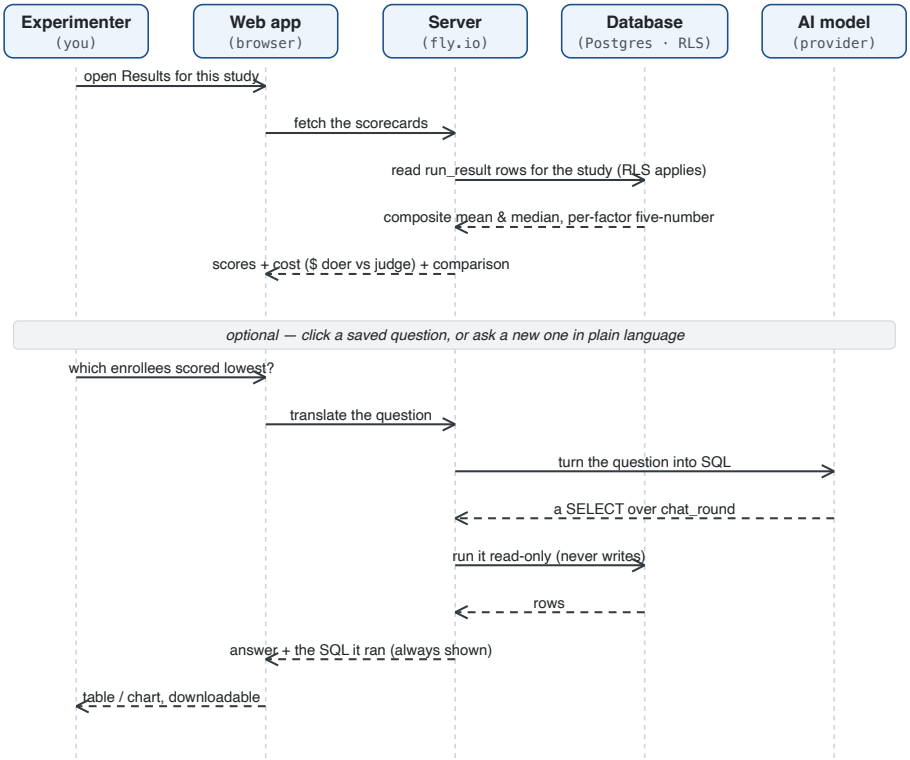
2 · The Journey, Screen by Screen

All read-only; the comparison is a query, not a stored object.



3 · Behind the Scenes

Where the numbers actually come from, including the plain-language query path.



A dashed line is a reply. The comparison is query-time SQL over the `run_result` rows; the plain-language path adds an AI translation step but still runs read-only.

4 · In Plain English

This is where the study pays off. Every exchange in every run was graded the same way by a single judge AI, so the numbers are comparable, and the run's scorecard rolls them into one line.

You get two headline averages, and it is worth knowing why. One treats every question-and-answer equally. The other treats every person equally, so somebody who sent fifty prompts does not drown out somebody who sent five. If those two numbers disagree, that tells you something real about who drove the result.

Alongside quality you can see cost and effort: how many improvement passes it took, how slow it was, how many tokens it used, and the actual dollar cost, split two ways. The doer models (the ones under test) are billed separately from the judge, so you can tell whether an expensive set-up is expensive because of the answer or because of the grading. A model that is cheap per token but needs many passes can still end up the pricey one.

If you would rather just ask, open the Ask panel. You can click a saved question, a labeled one-click button someone already built, or type a new one in ordinary English. ChatMaestro answers with a short write-up plus any number of result tables, each of which you can flip to a chart. It also shows you the query it used, so you can check its work rather than trust it blindly. If the question turns out to be worth keeping, you save it as a button for next time. Anything you see, you can download.

Fits the schema cleanly. Everything read here is `run_result` (the score matrix and the operational and cost matrix, including `total_generator_cost` versus `total_judge_cost`) and `chat_round`. Saved questions are `nl_query` rows (see *Ask a question in plain language*). There are no analytics tables and no run-set table; comparisons are grouped queries.