GroupBrain Talks

LLM Application Evaluations

From informal “vibe testing” to a rigorous, repeatable evaluation setup — how to measure whether an LLM-powered application actually works, at every level of the stack.

SECTION 01

The Problem With “Vibe Testing”

Most teams start out evaluating their LLM features the same way: they type in a few prompts, eyeball the responses, and decide “yeah, that feels right.” This is **vibe testing** — judging quality by gut feel. It is fast and comforting, but it quietly breaks down the moment you need to trust, compare, or ship a system. Two problems make it unreliable.

1 · Not repeatable	Run the same prompt tomorrow — or hand it to a teammate — and you get a different judgement. There is no fixed set of inputs and no recorded expectation, so you cannot reproduce a result or prove that a change actually helped.
2 · Subjective & informal	“Good” means something different to every reviewer, and nothing is written down. Without shared criteria the verdict depends on who happened to look and what mood they were in — not on the system itself.

The deeper danger is that vibe testing gives a **false sense of confidence**. Because it *feels* like testing, teams believe they have verified their system when they have really only sampled it once, informally. When a regression slips in — a prompt tweak that quietly degrades ten edge cases while improving the one you happened to check — vibe testing will not catch it, because there is no baseline to compare against and no record of what “working” looked like before. As soon as a system has more than one contributor, more than one release, or any obligation to behave consistently, vibe testing stops scaling.

SECTION 02

Why LLM Apps Are Harder to Evaluate Than Traditional ML / DL

Even teams comfortable evaluating classical machine-learning models find LLM applications slippery. Two structural differences are to blame.

Traditional ML / DL	LLM Applications
Deterministic-ish outputs. A classifier maps an input to one of a fixed set of labels; the same input gives the same answer every time.	Probabilistic, open-ended outputs. The same prompt can yield many different valid (and invalid) free-text answers, so there is no single "correct string" to match against.
A single benchmark. One scalar — accuracy, F1, AUC — usually captures "how good" the model is.	Multi-dimensional benchmarking. Quality is a bundle — correctness, faithfulness, tone, safety, latency, cost — and no single number sums it up.

These two differences compound each other. In traditional ML, a held-out test set plus one accuracy number tells you almost everything, and any engineer can reproduce it exactly. In an LLM application, the "correct" output is a distribution rather than a point, the surface area of possible failures is enormous, and each dimension may need its own dataset, its own scoring method, and its own threshold. That is precisely why a single test is never enough — and why evaluation for LLM apps must be designed as a **system of evals** rather than a single script.

SECTION 03

So, What Is an LLM Evaluation?

An evaluation is the disciplined opposite of vibe testing. It is **repeatable** (same inputs, same procedure, every time), **systematic** (applied the same way to every candidate), and built on **clear, written criteria** for what "good" means.

The key mental shift: **an eval is not just a metric — it is the entire testing setup built around a question.** A number like "0.82" is meaningless on its own. It only has meaning inside the scaffolding of what you asked, what you compared against, and how you scored it.

Concretely, that scaffolding turns a vague worry ("is the chatbot any good?") into an answerable, repeatable experiment. You first name the precise question, then assemble a

fixed dataset that represents it, then pick a scoring method appropriate to the question, and only then read the resulting metric. Change any one of those parts — a different dataset, a different judge — and the same model can score very differently, which is exactly why the number without the setup is meaningless.

SECTION 04

Two Branches: Model Evals vs. Application Evals

"LLM evaluation" splits into two distinct families, and it helps to keep them apart.

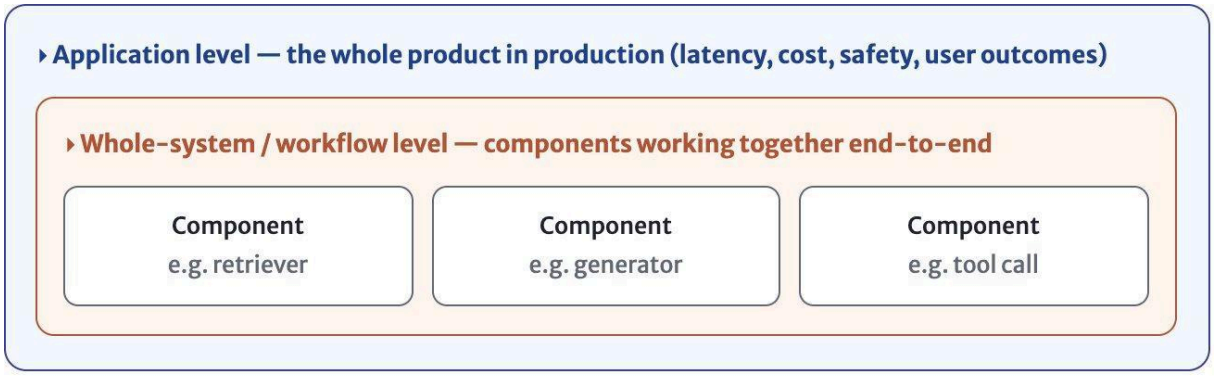
Model evaluations	Application evaluations ◂ our focus
Measure the raw model — reasoning, knowledge, safety — on broad academic benchmarks (MMLU, GSM8K, and the like). This is what model providers publish.	Measure your product built on top of a model — the prompts, retrieval, tools and glue — against the job it must actually do for real users.

For this lecture, we focus on LLM application evaluations from here on.

SECTION 05

Application Evals Operate at Three Levels

The same application is worth evaluating at three nested scopes — each catches failures the others miss.



Component evals nest inside workflow evals, which nest inside application-level evals.

Component level isolates a single building block (the retriever, one prompt, one tool call) and tests it in a vacuum, so you know precisely which piece is at fault. **Whole-system / workflow level** tests those components wired together, catching interaction bugs that no component reveals alone. **Application level** steps all the way back to the deployed product and asks the questions a user or a business actually cares about — is it fast

enough, cheap enough, safe enough, and does it solve the task end-to-end? A mature strategy runs all three, because a failure at each level is invisible to the others.

SECTION 06

The LLM Application Eval Workflow

Whatever the level, a good evaluation follows the same seven-step loop:

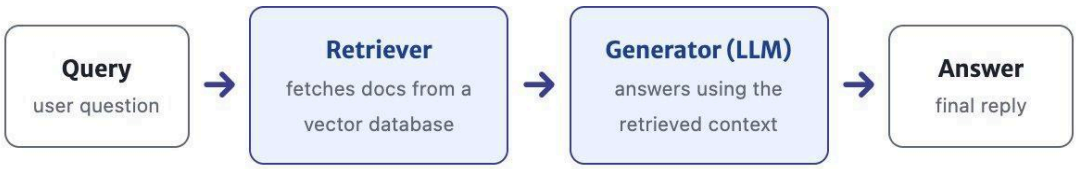
1. **Decide precisely what you are evaluating**
   Pin down the exact component, workflow, or application-level question.
2. **Define success criteria and metrics**
   Write down what "good" means and how it will be scored.
3. **Create the golden dataset**
   Curate trusted inputs (and expected outputs where relevant).
4. **Define an evaluation method**
   Choose LLM-as-judge, programmatic, human eval, or a mix.
5. **Run the model**
   Execute the system over the dataset and collect its outputs.
6. **Evaluate the results**
   Apply the scoring method to produce metrics.
7. **Analyze, improve, deploy, monitor**
   Act on findings, ship, then keep evaluating in production.

SECTION 07

One Application, Several Evals

Reason 1 — Multiple Failure Points

Why can't one eval cover a whole app? Because a real application is a *chain*, and every link can fail independently. Take a worked example: a **RAG chatbot** (Retrieval-Augmented Generation).



RAG architecture: Query → Retriever → Generator → Answer.

There are two obvious **component-level failure points**:

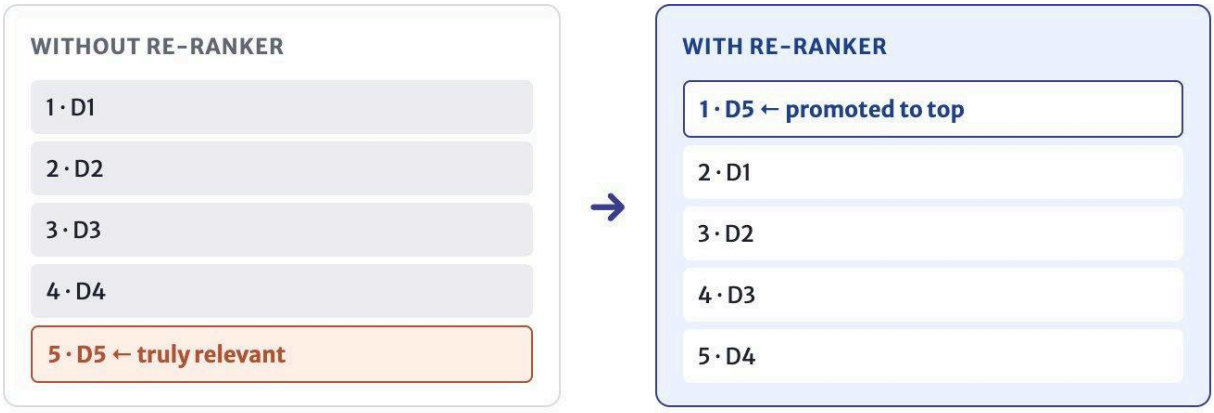
- **Retriever** — might fetch the wrong or irrelevant documents.
- **Generator** — might ignore correct context and hallucinate, or answer incorrectly even when the context was good.

So at minimum you need **one eval pipeline per component**:

Retriever eval	Generator eval
Given a query, are the relevant documents actually being retrieved? → Context Relevance.	Given a context, is the answer faithful and grounded in that context, without inventing outside facts? → Faithfulness / Groundedness.

Why component-level evals alone are not enough

Imagine both components pass their own evals — yet the chatbot still gives a weak answer. How? The retriever *does* return the truly relevant document (call it **D5**), but ranks it 5th, behind four merely-okay documents (D1–D4). Each component looks fine in isolation, but the combination fails: the generator leans on the top-ranked D1–D4 and never really uses D5.



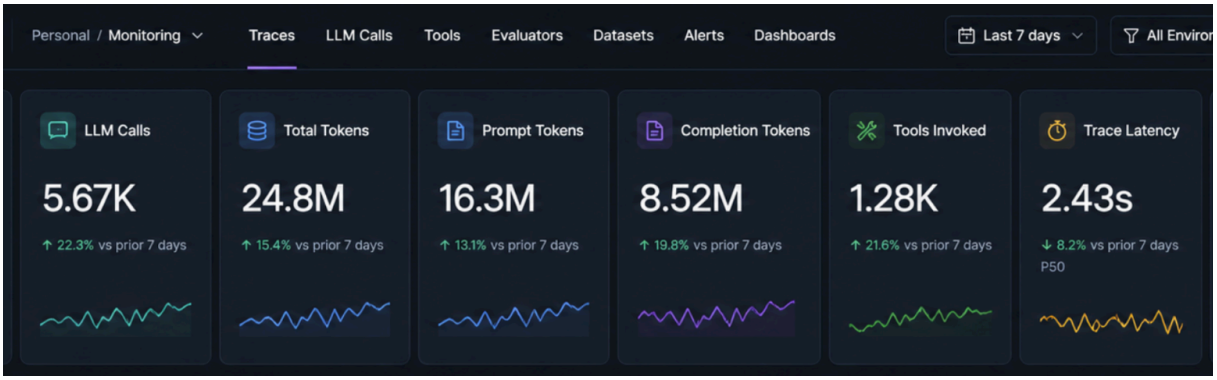
A workflow-level eval catches the ranking problem and points toward adding a **re-ranker**.

> Conclusion: component-level evals passing does not guarantee the workflow passes. You also need a workflow-level eval that tests the retriever + generator combination together — which here would point toward adding a re-ranker.

Even a passing workflow-level eval isn't the full picture

Now suppose the retriever + generator combination is finally correct. There is still a catch: answering a single query takes **10 seconds end-to-end** — far too slow for a production chatbot where a user is waiting after hitting enter. Correctness alone doesn't make a product usable.

> This is why an application-level eval is also needed — for example, checking that end-to-end latency stays under an acceptable threshold (and watching cost, safety, and user satisfaction too).



SECTION 08

LLM Application Eval Methods

Once you know what to measure, three broad methods do the measuring.

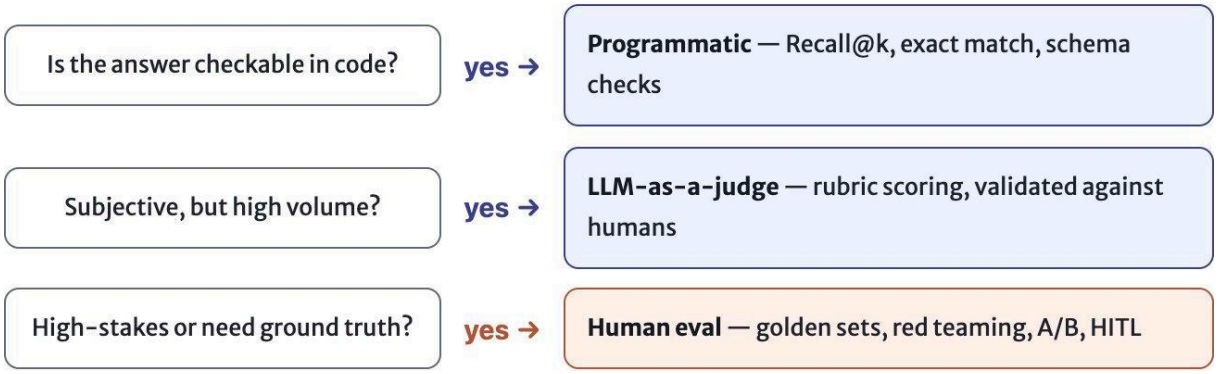
1 · LLM-as-a-judge	2 · Programmatic	3 · Human eval
A capable model scores the output against a rubric — scalable and flexible for open-ended, subjective qualities.	Deterministic code metrics — Recall@k, exact match, regex, JSON-schema checks. Cheap, fast, perfectly repeatable.	People judge quality directly — the gold standard, but slow and costly.

Human evaluation is an umbrella that includes **red teaming** (adversarially probing for failures), **A/B testing** (comparing variants on live traffic), **golden-dataset creation** (hand-curating trusted examples), and **human-in-the-loop (HITL)** review.

reference-based evals compare an output to a known correct answer, while *reference-free* evals judge quality without any ground-truth reference.

The three methods are complementary, not competing. Programmatic checks are the first line of defence wherever the correct answer is checkable in code. LLM-as-a-judge fills the gap for qualities that resist a formula — helpfulness, tone, faithfulness — though the judge itself must be validated against people. Human evaluation remains the ground truth you calibrate everything else against; too slow to run on every change, so you spend it deliberately.

CHOOSING A METHOD — A QUICK DECISION FLOW



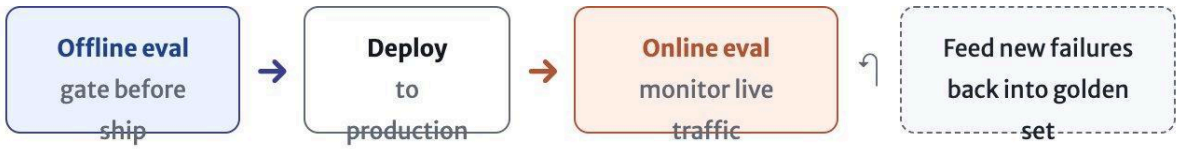
SECTION 09

Offline vs. Online Evaluations

A final axis is *when* you evaluate. **Offline** evals run before deployment on a fixed dataset; **online** evals run against live production traffic.

Offline evaluation	Online evaluation
Benefits	**Benefits**
• Fully repeatable — same fixed dataset every run. • Safe — no real users exposed to a bad version. • Fast to iterate and cheap to re-run in CI.	• Reflects true production behaviour and real users. • Catches drift and edge cases as they emerge. • Enables A/B tests and live guardrail monitoring.
Disadvantages	**Disadvantages**
• The dataset can drift from reality and go stale. • Misses real-world inputs you never anticipated. • Good offline scores don't guarantee live success.	• Not repeatable — traffic is never identical twice. • Risky — real users may hit a bad response. • Often lacks ground truth, so scoring is harder.

In practice the two are a loop, not a choice. Offline evals act as the gate before release; online evals watch what actually happens once real traffic hits the system; and the surprises online get folded back into the offline golden dataset so the gate keeps getting stronger.



Offline and online evaluation form a continuous improvement loop.

SECTION 10

Common Metrics Discussed in This Lecture

Inter-rater Agreement Metrics

Used to quantify consistency *between human raters*, ensuring they are aligned on subjective tasks.

Agreement Rate	A simple proportion of the time raters give the same response.
Cohen's Kappa	Adjusts the observed agreement rate relative to what would happen by random chance.
Fleiss' Kappa & Krippendorff's Alpha	Extensions of Kappa for scenarios involving more than two raters.

Why not just use the raw agreement rate? Because on tasks with few options, raters can agree a lot *purely by chance* — two people flipping coins agree half the time. Cohen's Kappa subtracts that expected-by-chance agreement, so a high Kappa means raters genuinely share the same standard. Fleiss' Kappa and Krippendorff's Alpha generalise the same correction to panels of many raters and to different data types.

Rule-based Metrics

Compare model outputs against fixed, human-provided reference texts.

METEOR	Metric for Evaluation of Translation with Explicit Ordering — a translation-focused metric combining precision, recall, and a penalty for poor word ordering.
BLEU	Bilingual Evaluation Understudy — a precision-focused metric comparing n-grams in the prediction to the reference, with a brevity penalty for overly short outputs.
ROUGE	Often used for summarization — measures overlap between the model output and the reference text.

SECTION 11 · CASE STUDY

Why This Matters for Project Sakhi

Every idea in these notes applies directly to Project Sakhi: It is exactly the kind of multi-component LLM application that vibe testing cannot safely ship. Because it runs unattended at a live lab bench and shapes how beginners learn, a wrong-but-confident answer is worse than no answer — so its quality has to be measured, not felt.

Application evals give the team a concrete advantage at each level of Sakhi's stack:

Sakhi component	Failure it can hide	Eval that catches it
RAG retriever (lab manual)	Pulls the wrong manual section, so guidance drifts from the syllabus.	Context Relevance / Recall@k
Tutoring LLM	Ignores the manual and invents a plausible-sounding but wrong step.	Faithfulness / Groundedness
Full guided session	Components each pass, yet the end-to-end explanation confuses a beginner.	Workflow-level eval on a golden set
Voice + memory loop	Response is correct but too slow, or recalls the wrong student's history.	Application-level: latency & recall %

This directly operationalises the metrics already named in the Sakhi proposal — **guidance correctness vs. the lab manual** is a faithfulness eval, **prior-session recall %** is a memory eval, and **wake-to-response latency** is an application-level eval.

- **Offline first:** build a golden set of real lab questions and mis-wired-circuit images to gate every prompt or model change before it ever reaches a bench.
- **Online in production:** log each session to the instructor dashboard, turning live traffic into an ever-growing eval set — the same offline→online loop described earlier.

> evals turn Sakhi's promises — syllabus-consistent guidance, reliable memory, sub-3-second responses — into numbers which can defend in a demo and improve week over week.