ChatGPT

Evaluation Strategy for LabAcc Copilot: Test-Driven and Evaluation-Driven Development

1. Why the current workflow is inadequate

The repository's `start-dev.sh` launches a FastAPI backend and a React front-end, sets the project root to `data/alice_projects` and exposes the service on `localhost:8002` [1]. Testing currently requires editing the code, re-running this script, opening the UI, asking questions and manually copying the output back into the conversation. The cycle is slow and error-prone. A better workflow should:

- **Automate test runs.** Relying on manual UI interactions makes it difficult to detect regressions, especially when responses are non-deterministic.
- **Define expected behaviour.** Without a formal specification, it's unclear whether the agent's answers are correct.
- **Evaluate more than answer text.** Agents have to plan, call tools, remember context and adapt; merely checking final output ignores these behaviours.

2. Lessons from state-of-the-art agent evaluation

Recent research shows that traditional test-driven development (TDD) or behaviour-driven development (BDD) is insufficient for LLM-based agents because outputs are non-deterministic and goals evolve over time. To address these issues, **evaluation-driven development (EDD)** integrates continuous offline and online evaluations into the development lifecycle [2]. Key take-aways from surveys and frameworks are summarised below.

2.1 What to evaluate

- **Fundamental capabilities.** Agents need to perform multi-step planning, call external tools correctly, manage memory, and self-reflect [3]. Benchmarks evaluate whether an agent decomposes complex tasks into sequential steps, invokes the right tools with proper arguments and uses outputs correctly [4]. Memory benchmarks test whether agents retain and recall information across long interactions [5]. Self-reflection benchmarks give the agent feedback and assess whether it can correct its own mistakes [6].
- **Process, not just outcome.** A recent survey argues that future frameworks should evaluate the *quality of reasoning chains*, efficiency of tool selection and the agent's adaptability when initial approaches fail [7]. Merely judging success/failure misses critical behaviours.
- **Agent self-improvement and safety.** Frameworks should measure learning curves across repeated tasks, adaptation to feedback, context retention and safety (e.g., toxicity, bias) [8] [9].

2.2 Evaluation methods

- **Static vs. dynamic evaluation.** Traditional offline tests (fixed prompts with expected outputs) provide repeatability but fail to capture agents' adaptation in open-ended environments [10]. Dynamic evaluation uses simulated environments or web tasks to observe the agent's decision-making over time [10].
- **LLM-as-a-judge.** For open-ended answers where accuracy metrics are hard to define, a large language model can grade responses. EvidentlyAI explains that LLM-as-judge evaluation can provide pairwise rankings or scalar scores when provided with clear criteria [11]. The method is flexible and cheaper than human evaluation [12].
- **Feedback functions.** Frameworks like **TruLens** use feedback functions to score aspects such as context relevance, groundedness and coherence [13] [14]. The library emits OpenTelemetry traces and allows comparison across versions, making it suitable for continuous evaluation [15].
- **Agent evaluation frameworks.**

Framework	Key features (with citations)	Applicability
OpenAI Evals	Provides a registry of evaluation templates and allows custom evals; emphasises that high-quality evals are critical for understanding model performance [16].	Good starting point for custom, model-graded evals; limited support for multi-turn agents.
DeepEval	PyTest-style framework to unit-test LLM outputs. Supports metrics such as G-Eval, hallucination, answer relevancy and RAGAS; can evaluate task completion and tool correctness [17] [18]. Runs locally and integrates into CI/CD pipelines [19].	Useful for writing repeatable tests against agent outputs; allows custom metrics.
TruLens	Evaluates agent executions using feedback functions for context relevance, groundedness, answer relevance and harmful language [20] [14]. Provides tracing, version comparison and OpenTelemetry integration [15].	Suitable for monitoring real runs and comparing versions; good for RAG and memory-oriented evaluation.
Phoenix Evals (Arize)	LLM-based evaluation library with pre-built templates for common tasks (e.g., RAG, function calling); runs evaluations on your own data frames, supports batched scoring and provides model-generated explanations for each judgment [21].	Helpful for large-scale, fast evaluation and tasks where explanations are valuable.
AWS Agent Evaluation	A framework that uses a built-in evaluator agent to simulate multi-turn conversations with your agent and evaluate its responses [22]. Supports integration with CI/CD pipelines and allows custom hooks [23].	Provides automated conversation evaluation and suitable for enterprises using AWS services.

Framework	Key features (with citations)	Applicability
LangSmith	Defines datasets (input + reference output pairs) and evaluators for scoring outputs [24]. Recommends starting with 10–20 high-quality manually curated examples [25] and augmenting with historical traces and user feedback [26]. Supports custom evaluators, pairwise comparisons and integration with Pytest/ Vitest tests.	Suitable for managing evaluation datasets and aligning with test-driven workflows.

2.3 Limitations in current benchmarks

A 2025 survey of agent evaluation frameworks notes that benchmarks often optimise for showcasing strengths rather than providing comprehensive evaluation [27]. It argues that future frameworks should include standardized environments with variable difficulty, multi-dimensional scoring (task completion rate, time/resource efficiency, safety compliance, user satisfaction), and human-AI collaborative evaluation [28]. As LabAcc Copilot is a wet-lab assistant, evaluations must consider domain-specific factors (e.g., safety of recommended protocols and alignment with lab best practices) in addition to general agent metrics.

3. Designing test cases for LabAcc Copilot

3.1 Identify key user journeys

The agent will need to handle typical lab queries, for example:

1. **Project-level questions:** “What is this project about?” or “List all experiments in this project.” It should summarise the README and list experiments.
2. **Experiment-level questions:** “What is in this experiment?” or “Describe the results in the README for experiment X.” It must fetch structured YAML and free-text notes.
3. **File-specific queries:** “Tell me about this file” or “How many files are in this folder?” It must read the file, count items and report data statistics.
4. **Recommendation and optimisation:** “What should I optimise next?” or “Suggest changes to improve this protocol.” It should analyse previous successes and failures, identify patterns and propose next steps using memory and literature tools.
5. **Multilingual queries:** Ask the same questions in Chinese to ensure the system does not rely on keyword matching. Multilingual support should be verified by evaluating both English and Chinese prompts.

3.2 Create an evaluation dataset

Follow LangSmith’s advice: start with **10–20 manually curated examples** covering typical and edge cases [25]. For each example:

- Provide the **input prompt** (English or Chinese).
- Provide a **reference answer or rubric** describing the expected content (e.g., the agent should mention there are three experiments and summarise each; the count of files should match the test folder; the recommendation should consider previous failures).

- Optionally include **metadata** such as difficulty and test category to help filter tests later [29].

As the application evolves, augment this dataset with:

- **Historical traces and user feedback.** Real user interactions reveal edge cases and failure modes [26]. Add examples where the agent performed poorly to ensure future versions fix regressions.
- **Synthetic examples.** Frameworks like DeepEval can generate adversarial prompts to test hallucination, prompt injection and safety [30]. This is especially important for ensuring the agent avoids dangerous lab suggestions.

3.3 Write a test harness

1. **Isolate the backend.** The FastAPI backend launched by `start-dev.sh` exposes endpoints at `localhost:8002`. Interact with these endpoints directly using Python scripts (e.g., via `requests`) instead of the front-end.
2. **Script test cases.** For each dataset example, write a test function that:
3. Sends the input prompt to the API.
4. Captures the JSON response, including the final answer, any tool calls and reasoning traces.
5. Logs metadata such as latency and number of tool invocations.
6. **Evaluate responses.** Use an evaluation framework: for example, DeepEval or a custom script that calls an LLM to grade the answer against the reference rubric. Pass the agent's answer and the expected answer to the evaluator; prompt the LLM to score relevancy, completeness and correctness on a 0–10 scale and provide justification. This implements an **LLM-as-judge** method [11] [12].
7. **Record metrics.** Store scores for each dimension (task completion, reasoning quality, tool correctness, multilingual accuracy) along with raw outputs. Also record system metrics like latency and memory usage.
8. **Automate via CI/CD.** Integrate the test harness into your continuous integration pipeline. Use frameworks like DeepEval or AWS Agent Evaluation to run tests on each pull request and fail if average scores fall below thresholds. DeepEval integrates seamlessly with any CI/CD environment [19] and supports custom metrics and red-teaming tests [30].

3.4 Define evaluation metrics and thresholds

The evaluation should combine objective metrics and model-graded scores:

- **Task completion rate.** Percentage of test cases where the agent meets the core requirements of the prompt.
- **Answer relevance and completeness.** Use metrics like answer relevancy and faithfulness from DeepEval [31] or context relevance from TruLens [14] to ensure the agent uses the correct information and cites experimental data.
- **Tool correctness and planning efficiency.** Determine whether the agent selected appropriate tools and executed them in the correct order. Count unnecessary tool calls and penalise inefficient plans.
- **Memory retention.** Check whether the agent incorporates historical context (previous experiments) when answering. For example, when asked what to optimise next, the agent should consider recorded success rates and patterns【15†L934-L942】.
- **Multilingual robustness.** Score how well the agent handles Chinese prompts compared with English. Compare response structures and information content.

- **Safety and compliance.** Use toxicity and harmful-language evaluators (e.g., TruLens or Phoenix) [14] to ensure the agent avoids unsafe lab instructions.

Set pass/fail thresholds for each metric (e.g., average task completion > 0.8, no critical safety violations) and track trends over time. Multi-dimensional scoring is important for capturing performance trade-offs [32].

4. Implementing evaluation-driven development for LabAcc Copilot

1. **Embed evaluation in the development cycle.** Adopt evaluation-driven development: after each code change, automatically run the test harness. The results should feed back into development decisions. This aligns with the EDD process model where evaluation guides iterative refinement and ensures the agent adapts to evolving goals and user needs [2].
2. **Use modular evaluation frameworks.** Start with DeepEval or LangSmith for offline unit tests. Use TruLens or Phoenix for online evaluation on live runs to monitor context usage, groundedness and user sentiment. Consider AWS Agent Evaluation for end-to-end multi-turn simulation if you integrate with AWS services [22].
3. **Leverage manual and synthetic data.** Combine curated examples with synthetic adversarial prompts (e.g., asking the agent to misinterpret file counts or propose unsafe protocols). Regularly expand the dataset with real user interactions and negative feedback, as recommended by LangSmith [26].
4. **Analyse process traces.** Capture agent reasoning steps, tool invocations and memory accesses. Evaluate the quality of the reasoning chain and efficiency of tool use [7]. Tools like TruLens and Phoenix provide tracing and comparison across versions [15].
5. **Iterate and refine.** Use evaluation results to prioritise improvements. For instance, if Chinese queries score lower, improve translation or intent detection. If recommendations ignore past failures, enhance the memory retrieval and analysis tools. Document these changes and update tests accordingly.

5. Architectural considerations for a wet-lab copilot

- **Maintain transparent and persistent memory.** The current design uses README files and YAML blocks to store experimental context [1]. This file-based memory should be validated and versioned to ensure consistency. Consider indexing experiments using embeddings for semantic search if the number grows large.
- **Use modular tools.** Implement the agent as a single LLM orchestrating modular tools for reading/writing memory, analysing data and performing literature searches. Keep tool interfaces clean; use Python code for heavy data analysis and LLM for reasoning and summarisation. Add new tools (e.g., PCR primer design, statistical analysis) gradually to avoid scope creep.
- **Guard against hallucination and unsafe suggestions.** Use evaluators to detect hallucinations, fact-check suggestions and ensure safety. Provide citations and require the agent to justify recommendations with evidence.
- **Plan for multilingual support.** Ensure the LLM can handle Chinese prompts and produce answers in Chinese. Evaluate cross-lingual performance and, if necessary, introduce translation modules or fine-tuning.

6. Conclusion

Migrating from manual testing to an evaluation-driven, test-driven workflow will accelerate development of the LabAcc Copilot and improve reliability. Modern evaluation frameworks emphasise *multi-dimensional metrics*, *process-oriented analysis* and *continuous integration*. By designing a comprehensive test harness, curating an evaluation dataset, and adopting tools like DeepEval, TruLens, Phoenix or LangSmith, you can systematically measure and improve your agent's capabilities. Such an approach ensures the wet-lab copilot not only answers questions correctly but also plans effectively, uses tools appropriately, retains context, operates safely and adapts over time.

---

1 start-dev.sh
https://github.com/LuyiTian/Labacc_copilot/blob/1ac4b2f9cf8268eac48c967b331a8db9d1ce4634/start-dev.sh

2 [2411.13768] Evaluation-Driven Development of LLM Agents: A Process Model and Reference Architecture
https://arxiv.org/abs/2411.13768

3 4 5 6 Evaluation Methodologies for LLM-Based Agents in Real-World Applications | by Adnan Masood, PhD. | Jul, 2025 | Medium
https://medium.com/@adnanmasood/evaluation-methodologies-for-llm-based-agents-in-real-world-applications-83bf87c2d37c

7 8 27 28 32 A Survey of Agent Evaluation Frameworks: Benchmarking the Benchmarks
https://www.getmaxim.ai/blog/llm-agent-evaluation-framework-comparison/

9 13 14 15 20 TruLens: Evals and Tracing for Agents
https://www.trulens.org/

10 Evaluation and Benchmarking of LLM Agents: A Survey
https://arxiv.org/html/2507.21504v1

11 12 LLM-as-a-judge: a complete guide to using LLMs for evaluations
https://www.evidentlyai.com/llm-guide/llm-as-a-judge

16 raw.githubusercontent.com
https://raw.githubusercontent.com/openai/evals/main/README.md

17 18 19 30 31 raw.githubusercontent.com
https://raw.githubusercontent.com/Confident-AI/deepeval/main/README.md

21 Overview: Evals | Phoenix
https://arize.com/docs/phoenix/evaluation/llm-evals

22 23 Agent Evaluation
https://awslabs.github.io/agent-evaluation/

24 25 26 29 Evaluation concepts | LangSmith
https://docs.smith.langchain.com/evaluation/concepts