

baswe.Ai Engineer Accelerator™

Ops & EvaluationCheckpoint

Format:

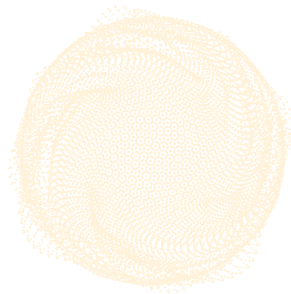
Production hardening + Loom Walkthrough + Technical Exam

Time investment:

Weekend sprint (10–14 hours)

Stack:

Python · LangSmith · Arize Phoenix · Great Expectations



baswe's

AiEngineerAccelerator

Pre-checkpoint checklist

Before you start, confirm all of the following.

Accounts & API access

- You've completed the Module 1 checkpoint.
- OpenAI API key (carried from Module 1).
- Pinecone account (carried from Module 1).
- LangSmith account (free tier is sufficient).
- Arize Phoenix (Arize AX (hosted) is acceptable if you prefer it)
- GitHub account Loom account (free tier is fine for a single recording).

Local environment

- Python 3.10 or later.
- Git, configured with your name and email.
- A code editor with Python tooling.
- Terminal comfort with venv / uv / poetry.
- Your LLMs & RAG build runs end-to-end and passes its own tests before you begin.

Stack

- LangSmith (latest stable) for tracing, eval datasets, and LLM-as-Judge evaluators.
- Arize Phoenix for span-level observability and trace analysis on your RAG pipeline.
- Great Expectations (latest stable) for data quality validation on your corpus and ingestion pipeline.
- Pytest for your extended test suite.
- LLMs & RAG stack (LangChain, OpenAI, Pinecone) carried forward.

AiEngineerAccelerator

Part 1: The Build

Scenario

Six weeks after you shipped your RAG-powered support assistant, the VP of Engineering wants a go / no-go decision on rolling it out to all 40 senior CSMs. She has seen demos that look great and demos that fall apart in production. She is not interested in a third demo. She is interested in evidence.

Your task: take your Module 1 RAG and bring it to production maturity. Instrument it for tracing, build a real evaluation suite, lock down data quality on the corpus, define a monitoring plan, and produce a production readiness report with a defensible go / no-go recommendation. This is the work that separates an AI engineer from someone who can ship a notebook.

Required architecture

Tracing	LangSmith instrumented across the full ingestion → retrieval → generation pipeline. Every chain run produces a trace with inputs, outputs, and intermediate steps.
Evaluation suite	LangSmith eval dataset with at minimum 3 evaluators. 2 standard (faithfulness, answer relevance) and 1 custom LLM-as-Judge designed specifically for the CSM support context. Eval runs are reproducible and versioned.
Observability	Arize Phoenix instrumented on the RAG pipeline. Spans capture retrieval latency, generation latency, token usage, and document IDs returned. You must use Phoenix to identify at least one specific weakness in your system and document it.
Data quality	Great Expectations expectation suite on the ingestion pipeline. At minimum: chunk length distribution, metadata completeness, embedding dimension consistency, no detected PII patterns. Validation runs as part of ingestion and writes a report.
Monitoring plan	Documented monitoring specification covering quality, system, and cost metrics. Each metric paired with a threshold and an alert condition. You do not deploy monitoring; the plan must be specific enough that an SRE could implement it from your document.
Production report	Markdown document with a go / hold / kill recommendation backed by your eval results, observability findings, data quality verdict, and identified risks.
Tests	Pytest suite extended to cover the new ops layer. New tests must run under <code>python -m pytest tests/</code> .

Repo structure (extending LLMs & RAG)

```
checkpoint-rag/
├── README.md                # Updated with Module 3 sections
├── pyproject.toml           # Pinned versions for ops dependencies
├── .env.example             # New env vars: LANGSMITH, PHOENIX
├── src/                    # Unchanged from Module 1
├── evals/                 # Unchanged from Module 1
├── tests/
│   ├── (existing Module 1 tests)
│   └── test_ops.py         # NEW – tests for ops layer
├── ops/                   # NEW – operational maturity layer
│   ├── langsmith/
│   │   ├── setup.py       # project, dataset, tracer init
│   │   └── evaluators/
│   │       ├── faithfulness.py
│   │       ├── answer_relevance.py
│   │       └── custom_judge.py # your custom LLM-as-Judge
│   │       └── run_evals.py
│   ├── phoenix/
│   │   ├── instrument.py  # Phoenix tracer + span exporter
│   │   └── analysis.md    # findings from your trace review
│   ├── data_quality/
│   │   ├── expectations.py # Great Expectations suite definition
│   │   ├── run_validations.py # runs the suite, writes report
│   │   └── results/
│   ├── monitoring/
│   │   └── monitoring_spec.md # production monitoring plan
│   └── reports/
│       └── production_readiness.md # go / hold / kill report
```

baswe's

Required technical checks

1. `python -m pytest tests/` exits 0 (Module 1 + new ops tests).
2. LangSmith project URL is included in your README, and traces from at least one full eval run are visible at that URL.
3. `python -m ops.langsmith.run_evals` executes end-to-end against your eval dataset and writes results to LangSmith.
4. `python -m ops.phoenix.instrument` launches a Phoenix session, runs at least 10 sample queries through the RAG, and produces traces. Screenshots of your trace analysis are committed in `ops/phoenix/`.
5. `python -m ops.data_quality.run_validations` executes the Great Expectations suite and writes a validation report.
6. Your custom LLM-as-Judge averages ≥ 0.75 on the eval set, AND aggregate faithfulness improves by ≥ 0.05 over your Module 1 baseline.

README additions

- **Operational maturity summary:** what you added on top of Module 1, in one paragraph.
- **Eval methodology:** which evaluators, why each, how you calibrated the custom LLM-as-Judge against ground truth.
- **Observability findings:** what Phoenix surfaced, what you changed in response, what you measured before vs after.
- **Data quality contract:** which expectations, what failure each guards against, what happens when one fails in CI.
- **Monitoring plan summary:** top 5 metrics with thresholds and rationale.
- **Production readiness verdict:** ship / hold / kill, with the three pieces of evidence that would change your mind.

Part 2: Loom Walkthrough

Format

- Single Loom recording, 10–30 minutes, screen share with face cam on.
- Public or unlisted Loom link, included in your README

Required structure

Your Loom must hit each of the following sections, in order.

1. Operational maturity overview (3–5 minutes)

- Walk through what you added on top of Module 1.
- Explain how tracing, evaluation, observability, data quality, and monitoring fit together end to end.
- Call out where you made a non-obvious decision and why.

2. Live eval & trace demo (4–6 minutes)

- Run your LangSmith eval suite live, or walk through a recent run.
- Open Phoenix and walk through a specific trace where retrieval or generation failed.
- Narrate: what does the trace tell you, what would you change, how would you measure that change.

3. Data quality demo (3–5 minutes)

- Run Great Expectations live against your corpus.
- Walk through 2 expectations: what each guards against, what happens when one fails, why those over others.

4. Monitoring plan defense (4–6 minutes)

- Walk through your monitoring specification.
- Pick 3 metrics. For each: why this metric, why this threshold, what action the alert triggers.
- Generic answers ('I would monitor latency') score low. Specific answers ('I would alert on p95 retrieval latency above 1.2s sustained for 5 minutes, because my Phoenix traces show baseline p95 at 600ms and a sustained doubling indicates Pinecone slowdown rather than a single bad query') score high. **Companies hire engineers who can defend their numbers.**

5. Production readiness verdict (3–5 minutes)

- State your ship / hold / kill recommendation.

- Walk through the three pieces of evidence behind it.
- State the conditions under which you would change your verdict.

Part 3: Technical Exam

Format

- Open-book. Open-internet.
- Do **NOT** use an LLM to generate your answers. Do **NOT** Google the entire question. Cheating on these only cheats yourself. You are evaluating your judgment, not your prompting.
- Time-boxed: 1–2 hours. We do not enforce the timer, but if you spend 6+ hours on this section, that is a signal you might not have understood the concepts deep enough.
- Written answers in a single Markdown file (`exam-answers.md`),

Section A: LLMOps & MLOps Foundations (8 points)

Q1 (3 pts)

Explain the differences between MLOps and LLMOps. Where do they overlap, where do they diverge, and which Module 1 → Module 3 design decisions in your own build map cleanly to one discipline versus the other? Use your own system as the example.

Q2 (3 pts)

Walk through the LLM application lifecycle from ideation to operational deployment. For each stage, name one decision a senior engineer makes that a junior engineer would miss.

Q3 (2 pts)

Some LLM apps need full MLOps maturity (CI/CD, registry, lineage, governance). Others can ship with a lighter LLMOps footprint. Give two concrete examples — one of each — and explain what determines which side of the line a system falls on.

Section B: Evaluation (12 points)

Q4 (3 pts)

Define eval-driven development for LLM applications. How does it differ from test-driven development for traditional software? What changes in your workflow when an evaluator can be wrong, expensive to run, and non-deterministic?

Q5 (3 pts)

LLM-as-Judge: when is it the right call, when is it the wrong call, and how do you calibrate one so you can trust its output? Describe a concrete calibration procedure you would run on the custom evaluator you built in Part 1.

Q6 (3 pts)

Compare algorithmic evaluators, human evaluators, and LLM-as-Judge evaluators across reliability, cost, latency, and what they can measure. Give one example metric that is best measured by each.

Q7 (3 pts)

Evaluating a multi-step agent is meaningfully harder than evaluating a single-shot RAG. Name three specific failure modes that show up in agents but not in your Module 1 RAG, and explain how you would catch each with Arize-style agent evaluation.

Section C: Observability & Monitoring (10 points)

Q8 (3 pts)

Distinguish tracing, logging, and metrics. For your Module 3 build, give one concrete example of each — what specific signal you capture, in which tool, and what production question that signal lets you answer that the other two cannot.

Q9 (4 pts)

List four production failure modes for a deployed RAG system that you would only catch with proper observability — i.e., that would not show up in a passing eval run. For each: the metric, the threshold, the alert action, and which observability tool surfaces it.

Q10 (3 pts)

Distinguish data drift, concept drift, and prompt drift in an LLM application. For each, give a realistic scenario in your CSM RAG context, the signal that would let you detect it in monitoring, and one mitigation.

Section D: Data Quality (10 points)

Q11 (4 pts)

Walk through the data quality dimensions (completeness, accuracy, consistency, timeliness, validity, uniqueness) and explain which apply to a RAG corpus, which apply to retrieval results, and which apply to generation outputs. For three of those dimensions, write the Great Expectations expectation you would author and what failure it would catch.

Q12 (3 pts)

You discover your CSM RAG occasionally surfaces customer PII from old support tickets in its retrieved context. Walk through how you would (a) detect this in your existing data quality suite, (b) prevent it from reaching generation, (c) prevent it from re-entering the corpus on the next ingestion run.

Q13 (3 pts)

The corpus you indexed in Module 1 will not be the corpus you serve in month six. Describe a corpus change validation strategy: what you check on every ingestion, what you check weekly, what you check before promoting an updated index to production. Be specific about the expectations that gate each step.