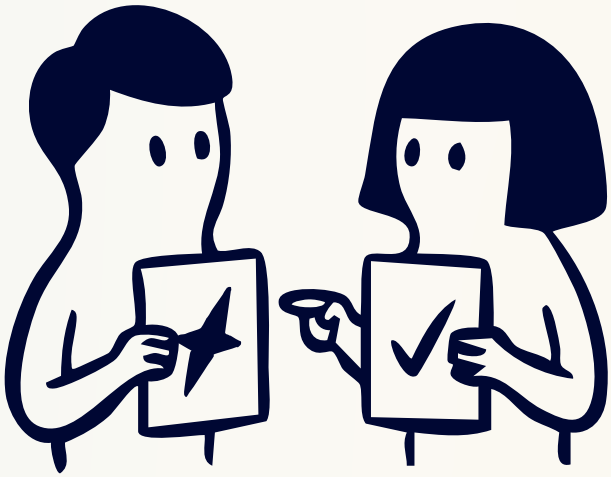
Oasium AI

GUIDE № 03



Evals before code: a practical handbook for AI builds.

Engineers, tech leads, and AI builders shipping production systems. Useful as a starter handbook before a system goes live, or as a structured way to evaluate whether an existing system has the eval and observability discipline to survive month four.

ENGINEERING · 36 PAGES · PDF

MAY 2026 EDITION

CONTENTS

What's inside

01

Why evals before code

The demo is always good.

We have sat through enough AI demos — as buyers and as the consultants called in to clean up after them — to know that the demo is always good. The agent does the thing. The model writes the report. The dashboard updates. Everyone leaves the meeting impressed. Six months later the system is either working or it is not, and what decided that was almost never the part of the work that anyone showed in the demo.

This guide is about the work that decides whether your AI system is still working in month four. It's not about model choice, prompt engineering, or framework selection — those are the visible parts. This is about the part most teams skip and most teams regret skipping: the evaluation suite.

The thesis

Build the eval suite before you build the system.

Three reasons this matters more than it looks:

One. An eval suite written *after* the system biases toward what the system already does. By the time you've shipped a working prototype and your team has formed opinions about what "good" output looks like, the eval cases you write will tend to confirm what you've already decided to do. An eval suite written *before* the system, by contrast, tells you what the system needs to do — independent of what you've built. The order of operations matters.

Two. A 30-case eval suite catches roughly 80% of regressions for roughly 1% of the effort of catching all regressions. The marginal value of an eval suite is enormous and almost entirely front-loaded. Once it exists, every subsequent decision (model swap, prompt change, retrieval tweak) becomes measurable. Without it, every decision is a vibe.

Three. The most common production-AI failure pattern in 2026 is *invisible drift*: a system that worked at launch slowly degrades because the underlying model changed underneath, or your data shifted, or a vendor updated their API behavior in a "minor" release. Without an eval suite running continuously, you don't notice until a customer does. With one, you notice within a day.

Three failure patterns from production agents

To make the case for evals concrete, here are three failure patterns we've seen in client engagements over the past year. All three were preventable with eval suites in place. None were detected by the teams running the systems until users complained.

Pattern 1: The silent model migration regression. A client's customer-support routing system, in production for 8 months, started routing 12% more tickets to the "general" queue (lowest priority)

starting on a specific Tuesday in March. Users noticed only after a backlog built up. Root cause: the API provider had migrated their model to a new version with subtly different classification behavior. The change was announced in release notes nobody read. An eval suite running daily would have caught the regression within hours.

Pattern 2: The retrieval drift. A research-Q&A system worked well at launch, then gradually degraded over six months. Users started complaining that answers cited papers that didn't actually contain the cited content. Root cause: the underlying corpus had grown 4x since launch; the retrieval system was returning more documents but the relevance ranking had shifted. The team had no eval suite measuring "answer faithfulness to retrieved documents," so they couldn't see the regression until it was severe.

Pattern 3: The prompt update that broke things subtly. A team updated their system prompt to add a new instruction. The instruction worked correctly for the new use case it was intended to handle. But it changed model behavior on three other workloads in ways nobody noticed for a month. By the time it was caught, the team had received unrelated complaints they'd already attributed to other causes.

In all three cases, the system "worked" by every visible metric (uptime, latency, no errors). The quality regression was invisible until human users surfaced it. An eval suite is what makes quality regressions visible.

What this guide is and isn't

This guide is a practical handbook for building eval suites that work in production. It covers test set design, scoring rubrics, tool selection (Inspect AI, Braintrust, Promptfoo, LangSmith, and friends), production observability, three worked examples at different scales, anti-patterns, and operational reality.

It is not a survey of every eval framework in the market. It is not a research-grade discussion of LLM evaluation theory. It is the thing we'd want a new technical lead to read before they shipped their first AI system in production.

02

Designing a held-out test set

A held-out test set is the foundation everything else rests on. Get this wrong and no amount of tooling helps. Three principles.

Principle 1: Cases come from real history, not from your imagination

The biggest mistake in eval design is generating synthetic test cases. *"We need 30 examples of customer support tickets for our routing test"* — and someone writes 30 plausible-looking tickets. These cases are useless. They reflect the case-writer's assumptions about the workload, not the actual distribution of cases the system will see in production.

Where cases should come from:

- **Production logs** if the system is already running. Anonymize, sample stratified across categories, hand-pick edge cases.
- **Pre-AI process records** if the workflow exists and was previously done by humans. Pull the last 200 cases your team handled manually; pick a representative 30.
- **Customer interviews** if no historical data exists. Ask 5-10 customers to walk you through 3-5 real cases each. Capture verbatim. Anonymize.
- **Field collection** as last resort: have someone shadow the workflow for a week and document every case they see.

What never works: case generation by the team building the system. The bias is too strong.

Principle 2: 20-50 cases is usually enough

Production AI teams routinely overshoot on test set size. We've seen teams collect 500 test cases before they ship anything; the cases get stale, the team loses momentum, and the eval suite becomes a project rather than a tool.

Twenty to fifty cases is usually enough to detect regressions and make confident decisions. Specifically:

- **20 cases**: enough to detect whether a system meets a baseline quality bar. Use this for first-pass go/no-go decisions.
- **30 cases**: enough to detect regressions in subsequent versions with reasonable statistical confidence. The right size for ongoing CI evaluation.
- **50 cases**: enough to stratify across categories and edge cases. Use this for systems with multiple distinct workloads or when the cost of a regression is high.

If you find yourself wanting 200+ cases, you're either (a) using the eval suite as a substitute for production monitoring (it isn't one — production observability is separate, see section 05), or (b) trying to cover every edge case rather than the modal cases (don't — edge cases get caught by production observability).

Principle 3: Cases should be representative, dated, and re-runnable

Representative means the distribution of your test cases should look roughly like the distribution of cases the system will see in production. If 70% of your real workload is "easy" cases, 70% of your test cases should be easy. The other 30% should split across whatever specific patterns matter — edge cases, common failure modes, sensitive scenarios.

A common error: stuffing the test set with hard cases because they feel important. The result is an eval that constantly fails, becomes noise, and gets ignored. Match production distribution.

Dated means each case carries the date it was added and ideally the date the case originated (if pulled from production logs). Cases age. When the distribution shifts, you'll need to refresh the test set; knowing when each case was added makes that maintainable.

Re-runnable means a non-author can run the entire suite, get the same results, in under five minutes. Practically: cases should be data, not hand-coded specifically for one model. Inputs go into a structured format (CSV, JSON, YAML — all work). Expected behavior or grading rubric goes alongside. The harness handles the rest.

If running your eval suite requires a human to manually evaluate each case, you've built a "manual test plan," not an eval suite. You'll run it twice and stop.

How many test sets do you need?

For most systems: one. A single 30-case suite covering the modal workload.

For systems with structurally distinct workloads: one per workload. A customer-support system that does both routing and reply drafting needs two suites — different inputs, different scoring criteria, different patterns.

For evolving systems: keep the original suite stable and add a smaller "expansion" suite for new patterns. Don't constantly modify the original — you'll lose the ability to compare across versions.

Anti-patterns to avoid

- **The synthetic-case suite.** Generated by ChatGPT or by your team. Useless.
- **The accidentally-easy suite.** Pulled from production logs but unintentionally weighted toward routine cases your system already handles well. Always score the suite on a baseline (a known-good system or a manual baseline) to confirm there's signal in the cases.

- **The 500-case suite.** Stale, expensive to run, demoralizing to update.
- **The frozen-since-launch suite.** Made sense at launch; doesn't reflect current workload. Refresh annually.
- **The single-stage suite for a multi-stage system.** If your system has retrieval + generation + verification, you need to be able to evaluate each stage. A single end-to-end score hides where the failures are.

03

Scoring rubrics that work

Once you have your cases, you need a way to score them. The choice of scoring methodology matters more than most teams realize.

Three approaches, when each works

Binary scoring (pass / fail). The simplest possible approach: for each case, the system either gets it right or wrong. Best for: classification tasks, structured extraction, code that runs vs. doesn't. Not great for: generation tasks where partial credit matters.

Graded scoring (0-N point scale). Each case is scored on a 0-3, 0-5, or 0-10 scale based on a rubric. Best for: generation tasks, summarization, anything with degrees of correctness. The challenge: making the rubric specific enough that two different graders score the same output similarly. Untrained graders typically vary by 20-40% on graded scoring without explicit rubrics.

Multi-dimensional rubrics. Each case is scored on multiple dimensions: factual accuracy, helpfulness, tone, completeness, etc. Best for: complex generation tasks where you care about multiple qualities. The cost: more grading work per case, harder to summarize into a single number.

The judge: human, LLM, or both

Once you've decided on a scoring approach, you need a judge. Three options:

Human judges. The gold standard, especially for nuanced or domain-specific evaluation. The cost: human time. Reasonable for an initial 30-50 case suite that gets graded once or twice; impractical for daily CI runs.

LLM-as-judge. Use a strong frontier model (Claude Opus, GPT-5, Gemini 3 Pro) to grade your system's outputs against the rubric. Fast, cheap, scales to thousands of cases per run. The risk: LLM judges have biases — they prefer longer responses, prefer their own family's outputs, can be misled by confidence. Calibrate against human judgment before trusting LLM judges.

Hybrid. Use LLM-as-judge for routine grading (CI runs, frequent re-evaluations) and human judges for periodic calibration (every 1-3 months, sample 10% of cases for human grading and confirm the LLM judge scores correlate). This is the right setup for most production systems.

Calibrating an LLM judge

The single most important step in setting up LLM-as-judge that most teams skip. The procedure:

01 Have a human (ideally domain expert) grade the full eval set manually. Save scores.

02 Have your LLM judge grade the same eval set. Save scores.

03 Compute correlation between human and LLM scores per case.

04 Identify cases where the two disagree by more than one point.

05 Look at the disagreements. Three possibilities:

06 **The human was wrong** — refine the rubric to be clearer.

07 **The LLM judge was wrong** — refine the LLM judge's prompt with examples.

08 **The case is genuinely ambiguous** — flag it; potentially remove from the suite.

09 Iterate until correlation > 0.85 across the full suite.

Skip this step and your LLM judge is producing scores that look real but aren't. We've seen teams ship with judges showing 60% correlation to human grading — meaning their CI scores were nearly noise. They didn't know it.

Rubric authoring — what good looks like

A scoring rubric is a small document, typically 100-300 words, that says exactly what counts as a 5/3/1 (or whatever your scale is) for each dimension being scored.

Bad rubric example:

> *Score 1-5 on overall quality, where 5 is excellent and 1 is poor.*

This is a vibes test. Two graders will produce wildly different scores.

Good rubric example:

> *Score 1-5 on factual accuracy:*
>
> - *5: Every claim in the response is supported by the source documents. No hallucinated facts. No misattributed claims.*
> - *4: All claims are correct, but one or two minor details could have been more precisely stated.*
> - *3: Most claims are correct; one factual error that doesn't materially change the answer's correctness.*
> - *2: Multiple factual errors, including at least one that materially changes the answer.*
> - *1: Substantial fabrication; the answer doesn't reflect the source content.*

Specific, anchored, verifiable. Two different graders applying this rubric will produce similar scores.

Rubrics should be authored by someone in the domain who can recognize a right answer. If a non-domain expert wrote the rubric, you'll grade against the wrong criteria.

04

Tooling: where each tool fits

Five categories of evaluation tooling exist as of mid-2026. Each has a clear best use case. Picking the right one matters less than committing to one and running it consistently.

The serious eval frameworks

Inspect AI (https://inspect.aisi.org.uk/) (UK AI Security Institute, open source). The most respected eval framework in the AI safety community. Created by the UK AISI, used by METR, Apollo Research, and other safety-focused organizations. Strengths: typed, reproducible, introspectable. Comes with 200+ pre-built evaluations covering coding, agentic tasks, reasoning, knowledge, behavior, and multi-modal understanding. Active development with backing from a serious institutional sponsor. Weakness: more academic register than commercial; setup feels closer to a research tool than a production CI service.

Use Inspect AI when: You're a research team, an AI safety org, or an enterprise that values reproducibility and open-source provenance. Or when you want to leverage the 200+ pre-built evaluations as a starting point.

Braintrust (https://www.braintrust.dev/) (commercial). The most production-oriented eval platform in 2026. Connects evaluation scoring with production tracing, dataset management, cross-functional review, and CI-based release enforcement inside a single system. Strengths: end-to-end trace-to-test workflows, CI/CD integration, hybrid and self-hosted deployment options for enterprises with on-prem requirements, hosted UI that makes evals reviewable by non-engineers. Weakness: commercial pricing.

Use Braintrust when: You're shipping production AI systems and need eval-driven release enforcement. The CI integration alone justifies the cost for most production teams.

Promptfoo (https://www.promptfoo.dev/) (open source CLI). A simpler, command-line-driven eval tool. Strengths: extremely easy to integrate into existing development workflows, especially in Node.js environments. Built-in security testing and red-teaming features. Lightweight. Weakness: less production-grade than Braintrust; not designed for cross-team collaboration.

Use Promptfoo when: You're an individual developer or small team running evals from the command line. Or when LLM security testing / red teaming is the primary concern.

LangSmith (https://smith.langchain.com/) (commercial). LangChain's eval and observability platform. Strengths: integrated tightly with LangChain / LangGraph applications. Decent end-to-end coverage from development to production. Weakness: best fit if you're already in the LangChain ecosystem; less compelling otherwise.

Use LangSmith when: Your codebase already uses LangChain / LangGraph. The integration value is the primary driver.

Galileo (https://galileo.ai/) (commercial). Specialized in real-time safety guardrails and low-latency scoring for production environments. Strengths: built for runtime evaluation rather than offline test suites. Compliance-focused features for regulated industries. Weakness: narrower use case than Braintrust or Inspect AI.

Use Galileo when: You need real-time safety guardrails on production traffic, especially in regulated contexts where compliance posture matters.

Recommended pairings

For most production teams, the right combination is:

01 **One eval framework** for offline test suites and CI: Braintrust if budget allows; Inspect AI if you want open source; Promptfoo for solo developers.

02 **One observability tool** for production monitoring (see section 05).

Don't try to use multiple eval frameworks in parallel. The overhead of maintaining two sets of test cases isn't worth the marginal coverage benefit.

05

Production observability

Eval suites catch regressions in the development cycle. Production observability catches them in the wild. Both matter; neither replaces the other.

The pattern

Production observability for AI systems works on a sample-and-score pattern:

01 Sample some fraction of production traffic — typically 1-10%, depending on volume.

02 Re-score the sampled cases against your eval rubric (or a simplified version of it).

03 Track the distribution of scores over time.

04 Alert when the distribution shifts beyond a defined threshold.

This is fundamentally different from traditional observability (uptime, latency, error rate). Traditional observability tells you the system is *running*. AI observability tells you the system is *working* — that the outputs are still high-quality.

Tools

Three production-observability tools have settled into the market in 2026:

Helicone (https://www.helicone.ai/) (commercial, with open source self-hosted option). Captures every LLM API call your system makes, lets you sample, score, and compare outputs over time. Strong at the unit-economics layer (cost tracking, request logging) as well as quality scoring.

Langfuse (https://langfuse.com/) (open source, with commercial cloud option). Similar feature set to Helicone, with a stronger open-source posture. Good fit for teams that want to self-host their observability stack.

Arize Phoenix (https://phoenix.arize.com/) (open source, with Arize commercial cloud option). Comes from a longer ML-observability heritage; strong tooling for embedding-based drift detection, retrieval-quality scoring, and hallucination detection.

For most teams, Helicone or Langfuse are the right starting points. Phoenix earns its keep when retrieval-quality monitoring matters specifically.

Alerting patterns

Two alerts every production AI system should have:

Quality-score distribution shift. When the rolling 7-day average of your eval score drops below a threshold (typically 1 standard deviation below the historical mean), alert. This catches model

migration regressions, retrieval drift, and prompt-update bugs.

Tail latency spike. When p99 latency spikes above 2x the historical mean for more than 30 minutes, alert. This catches model deprecation, API issues, and infrastructure problems.

A third alert worth considering for high-volume systems: **cost spike.** When daily API spend exceeds 1.5x the rolling 30-day average, alert. This catches usage spikes, cost-bloat from prompt changes, and accidentally-uncached-input scenarios.

What "good" production observability looks like

The goal of production observability is that the alert reaches you before the customer ticket does. Concrete: when the model migration regression we described in section 01 happens, an observability system should fire an alert within hours of the regression, allowing you to investigate and roll back before customers are materially affected.

The difference between "good" observability and "stale dashboard nobody looks at" is the alerting. A dashboard you check weekly catches issues weekly. An alert system catches them within hours.

06

Three worked examples

Three eval setups at different scales, drawn from common production scenarios. Use these as starting points; adapt to your specific case.

Small: contract-clause extraction (5-case suite)

Workload: Extract specific clauses (termination, indemnification, payment terms, liability cap, governing law) from contracts. Output a structured JSON with each clause's text and metadata.

Test set design:

- 5 cases. Two from "easy" contracts (templated MSAs with standard structure), two from "medium" complexity (negotiated terms, non-standard formatting), one edge case (multi-jurisdiction contract with conflicting clauses).
- Each case includes: input contract text (anonymized), expected JSON output, and notes on what the right answer looks like.

Scoring approach:

- Binary scoring per clause type: did the system correctly identify and extract the clause text?
- Bonus dimension: did the system correctly classify the clause type (termination vs. indemnification vs. ...)?
- Per case: 5 binary scores (one per clause type) + 5 classification scores = 10 binary outcomes.
- Total possible: 50 across the 5 cases. Pass threshold: 45/50.

Tooling: Promptfoo or Inspect AI. Run on every commit to the prompt or model. Total run time: <30 seconds.

What this catches: Most regressions on extraction tasks. Misses: subtle issues like incorrect clause-type classification when extraction is correct; that's why the bonus dimension matters.

Medium: customer support routing (20-case suite)

Workload: Classify incoming customer support tickets into one of 7 routes (billing, technical, account, sales, refunds, escalation, general). Output the route + confidence score + brief reasoning.

Test set design:

- 20 cases drawn from the last 6 months of production tickets, sampled stratified across the 7 routes.

- Distribution: ~3 each from billing, technical, account; ~2 each from sales, refunds, escalation, general; plus 5 edge cases (multi-route tickets, ambiguous, sentiment-driven misroutes).
- Each case includes: ticket subject, ticket body, expected route, secondary acceptable route (if any), and grader notes.

Scoring approach:

- Multi-dimensional rubric scored 0-5 per dimension:
  - **Route correctness**: 5 = correct primary route; 3 = secondary acceptable route; 1 = wrong route.
  - **Confidence calibration**: 5 = high confidence on easy cases, low confidence on hard cases; 1 = miscalibrated.
  - **Reasoning quality**: 5 = clear, actionable reasoning; 1 = generic boilerplate.
- Per case: 3 dimensions × 5 = 15 max. Total possible: 300 across 20 cases.
- Pass threshold: 240/300 (80%).

Tooling: Braintrust. LLM-as-judge using GPT-5.5 for the multi-dimensional scoring; calibrated against human grading on a 25% sample. Run daily in CI.

Production observability: Sample 5% of production tickets, re-score using same LLM judge, alert if rolling 7-day average drops below 80% of historical baseline.

What this catches: Regressions in any of the three dimensions. Catches the silent model migration regression we described in section 01 — that exact pattern.

Large: multi-step research agent (50-case suite)

Workload: A research agent that takes a research question, performs retrieval over an internal corpus, synthesizes findings, and produces a 1-2 page memo with cited sources.

Test set design:

- 50 cases. 30 from real historical research questions the team has answered manually; 15 simulated cases covering edge categories (questions outside the corpus, multi-language source content, ambiguous questions); 5 deliberately impossible cases (testing the agent's ability to say "I can't answer this from the available sources").
- Each case includes: input question, expected key findings (the 3-5 facts the answer should contain), expected source citations, and grader notes on what good vs. bad output looks like.

Scoring approach:

- Multi-dimensional rubric scored 0-5 per dimension:
  - **Retrieval quality**: did the agent retrieve the right source documents?
  - **Factual accuracy**: are the claims in the memo supported by the retrieved sources?
  - **Citation correctness**: do the citations actually contain the cited content?
  - **Synthesis quality**: does the memo answer the question, or just regurgitate sources?
  - **Refusal calibration**: does the agent correctly refuse impossible cases?
- Per case: 5 dimensions × 5 = 25 max. Total possible: 1250 across 50 cases.
- Pass threshold: 1000/1250 (80%).
- Stage-wise scoring: retrieval, generation, and refusal evaluated separately so regressions can be localized.

Tooling: Braintrust + Phoenix for retrieval-specific drift detection. LLM-as-judge with GPT-5.5 for generation scoring; calibrated against human grading every quarter on a 20% sample. Human grading on the 5 refusal cases (LLM judges struggle with refusal calibration). Run weekly in CI; daily on production sample (10% sample rate).

Production observability: Three separate observability streams:

01 Retrieval-quality drift via Phoenix.

02 Generation-quality drift via Helicone or Langfuse with LLM judge on sampled traffic.

03 Refusal-rate tracking — alert if the agent's refusal rate moves >2 standard deviations from baseline.

What this catches: Almost all production regressions specific to multi-stage agent systems. The stage-wise scoring is critical — without it, a retrieval regression and a generation regression look identical at the end-to-end level, but require completely different fixes.

What to learn from these three

The size and complexity of the eval suite scale with the system's complexity, not the team's preference. A 5-case suite is right for a small narrow system. A 50-case suite is right for a multi-stage agent. Don't over-build the suite for a small system; don't under-build it for a large one.

The scoring approach scales similarly: binary for narrow tasks, multi-dimensional for tasks where multiple qualities matter, stage-wise for multi-stage systems.

07

Anti-patterns

A non-exhaustive list of eval anti-patterns we've seen in client engagements. Each has a specific failure mode worth knowing about.

The eval suite written after the system. As discussed in section 01, this biases toward what the system already does. Always write evals first.

The scoring rubric written by a non-domain expert. A system engineer writing the rubric for a legal-document workflow will measure the wrong things. The grader needs domain expertise.

LLM-as-judge with no calibration. Using GPT-5.5 to grade outputs against a rubric without ever validating that GPT-5.5 actually grades like a human would. Produces scores that look real but aren't.

Single-metric tyranny. Measuring only one number (e.g., "accuracy" on classification, "pass rate" on extraction). Hides multi-dimensional regressions. Always score multiple dimensions for non-trivial systems.

The "we'll add evals later" promise. "We need to ship; we'll come back and add evals." We've never seen this happen. Six months later there are no evals and the team is debugging production regressions blind.

The frozen eval suite. Made sense at launch, hasn't been updated in 14 months, no longer reflects current production patterns. Refresh the suite annually or whenever you observe substantial workload distribution shifts.

The 500-case eval suite. Too large to run frequently, too expensive to refresh, too demoralizing to maintain. The marginal value of cases past 50 is approximately zero for most workloads. Stop at 50.

Treating eval scores as ground truth. Eval scores are a useful signal, not the truth. A system that passes the eval but performs poorly in production is a sign the eval suite has gaps, not a sign the production users are wrong. Always weight observed user feedback over eval scores.

Eval-driven prompt engineering. Iterating on the prompt against the eval suite until you maximize the score. This produces prompts that are over-fit to the test cases. The result: high test scores, poor generalization. Mitigate by holding back 20% of cases as a "blind set" that you don't iterate against.

Working in test, broken in production. A pattern where the eval suite passes but production fails. Usually root-caused to: (a) eval cases don't reflect production distribution, (b) eval setup uses different infrastructure than production, (c) the system fails on cases the eval suite doesn't cover. Always validate the eval suite against production data periodically.

Skipping the refusal cases. Forgetting to test that the system correctly says "I don't know" when appropriate. Especially important for systems that synthesize from retrieved sources or that handle out-of-scope queries.

08

Operational reality

Once you have the eval suite running, what does month four look like?

Day-to-day

For a well-set-up eval suite running in CI:

- **Every commit** to model, prompt, or system code: full eval suite runs, scores update on the dashboard, regressions block merges.
- **Daily** in production: 5-10% sample of traffic re-scored, observability dashboard updates, alerts fire on score-distribution shifts.
- **Weekly**: a 30-minute review of the dashboard. Look at trends, recent alerts, anything that needs attention.
- **Monthly**: refresh ~10% of eval cases — replace stale cases, add new patterns observed in production, retire cases that have become trivial.

When something goes wrong

The alert fires. Score distribution dropped. Now what?

01 **Identify the regression.** Look at the failing cases. Is it specific cases, or a broad shift? If specific cases, what's different about them? If broad, what changed in the system or its dependencies?

02 **Triangulate the cause.** Three common culprits: (a) underlying model changed (check API provider's release notes), (b) input distribution shifted (check whether the categories of recent traffic match historical), (c) something in the system's code changed (check recent commits, prompt edits, retrieval index updates).

03 **Roll back if possible.** If the regression is severe and the cause isn't immediately fixable, roll back to the last known-good state and investigate offline.

04 **Add the regression case to the eval suite.** Whatever pattern triggered the regression should now be in the eval suite as a permanent test case. This way, the next similar regression gets caught earlier.

05 **Update the rubric if needed.** Sometimes regressions reveal that the original rubric was incomplete or unclear. Refine.

Refreshing the eval suite

Every quarter, spend 2-4 hours on suite maintenance:

- Add 3-5 cases from recent production for new patterns observed.

- Retire 2-3 cases that have become trivial (every model gets them right).
- Re-grade the full suite manually on a 10-case sample to confirm LLM judge calibration is still good.
- Review the rubric for any ambiguities surfaced during the quarter.

This maintenance compounds. A suite that's been continuously refreshed for 18 months is far more valuable than one that's been stable for 18 months.

When to add new evals

Three triggers:

01 **A regression in production that the eval suite missed.** Add the failing pattern as a permanent test case.

02 **A new feature or workload added to the system.** New workloads need their own eval cases, ideally in their own suite.

03 **A user-facing complaint that surfaces a quality issue.** Convert the complaint into an eval case if the pattern is likely to recur.

When the eval suite becomes the bottleneck

A sign your eval suite has grown too large: it takes more than 5-10 minutes to run on every commit. At this point, you have three options:

- **Stratified sampling**: run a 30-case sample on every commit; full suite weekly.
- **Suite splitting**: separate fast (under 1 minute) and slow (full) suites; fast on every commit, full nightly.
- **Smarter caching**: cache results for cases that haven't changed inputs, only re-run when inputs differ.

Most teams hit this point around the 50-100 case mark. Plan for it.

APPENDIX A

Eval starter template

A minimal starting point for a new eval suite. Adapt to your specific system. Use this format with Inspect AI, Braintrust, or roll your own; the format is straightforward enough to translate.

```
# eval-suite.yaml — minimal eval starter
suite:
  name: contract-clause-extraction
  version: 1
  description: Extract specific clauses from MSAs and produce structured JSON.

scoring:
  type: multi-dimensional
  dimensions:
    - name: clause-identification
      type: binary
      weight: 1
      rubric: |
        Did the system correctly identify the presence of the clause type
        (termination, indemnification, payment, liability, governing-law) in
        the contract? Score 1 if all clauses present in the source were
        correctly identified, 0 if any were missed or false-positive.
    - name: clause-extraction
      type: binary
      weight: 2
      rubric: |
        Was the extracted clause text the actual text of the clause from
        the source contract? Score 1 if extracted text matches source
        within 95% character similarity, 0 otherwise.
    - name: structured-output
      type: binary
      weight: 1
      rubric: |
        Was the output valid JSON with the expected schema? Score 1 if
        passes JSON schema validation, 0 otherwise.

cases:
  - id: case-001
    type: easy-templated-msa
    input: ./cases/case-001-input.txt
    expected: ./cases/case-001-expected.json
    notes: Standard MSA with all five clause types in canonical positions.
  - id: case-002
    type: easy-templated-msa
    input: ./cases/case-002-input.txt
    expected: ./cases/case-002-expected.json
    notes: Standard MSA with reordered clauses.
  - id: case-003
    type: medium-negotiated
    input: ./cases/case-003-input.txt
    expected: ./cases/case-003-expected.json
    notes: Heavily negotiated MSA with non-standard formatting; one clause
      buried inside an unrelated section.
  # ... and so on
```

The same shape works for code, scoring, and case management — the format is just data. Keep it in your repo alongside the system code so changes to either are visible in pull requests.

APPENDIX B

About Oasium AI

Oasium AI is an applied AI consultancy. Two PhDs (UC Berkeley and UCLA), based across San Francisco and Dubai, who use AI in their own daily work — and help organizations do the same. Three service lines: **Educate** (workshops and training), **Strategize** (AI feasibility and roadmaps), and **Implement** (production AI workflows, agent systems, **and the evaluation harnesses behind them**).

We wrote this guide because the eval discipline is something we end up rebuilding for every client engagement. Most published guidance on AI evaluation is research-grade or marketing-led. This one is the practical handbook we wish existed when we started.

If you're building an AI system and want a structured second opinion on your evaluation approach — or if any part of this framework is unclear in your specific situation — we offer free 30-minute discovery calls. No slides, no script, just a conversation.

Book a discovery call → (https://oasium.ai/contact#book-a-call)

Or write us at hello@oasium.ai.

If this guide was useful and you'd like more like it, the rest of our writing lives at oasium.ai/writing (https://oasium.ai/writing). We post when something's worth saying — usually a few times a quarter.

— Ziad and Jonah Co-founders, Oasium AI

Oasium AI

COLOPHON

About this guide

How to build an evaluation harness before you build the AI system. Held-out test sets, scoring rubrics, tool selection (Inspect AI, Braintrust, Promptfoo, LangSmith), production observability, three worked examples at different scales, anti-patterns, and what month-four operational reality looks like.

AUTHORS

Ziad Yassine and **Jonah Lipsitt**

Co-founders, Oasium AI

San Francisco + Dubai

CONTACT

oasium.ai

hello@oasium.ai

For consulting work, the workshop, or to point out something we got wrong.

EDITION

Guide № 03 · May 2026

The latest version is always at oasium.ai/guides.