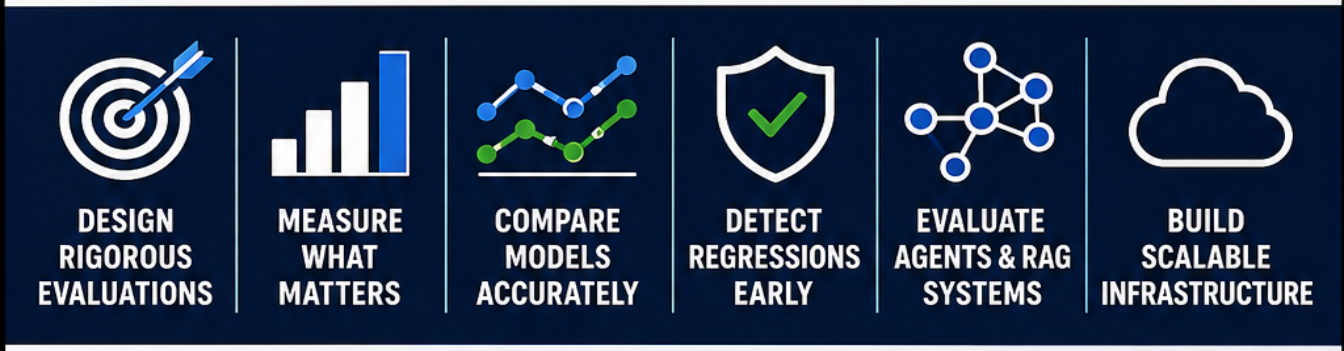
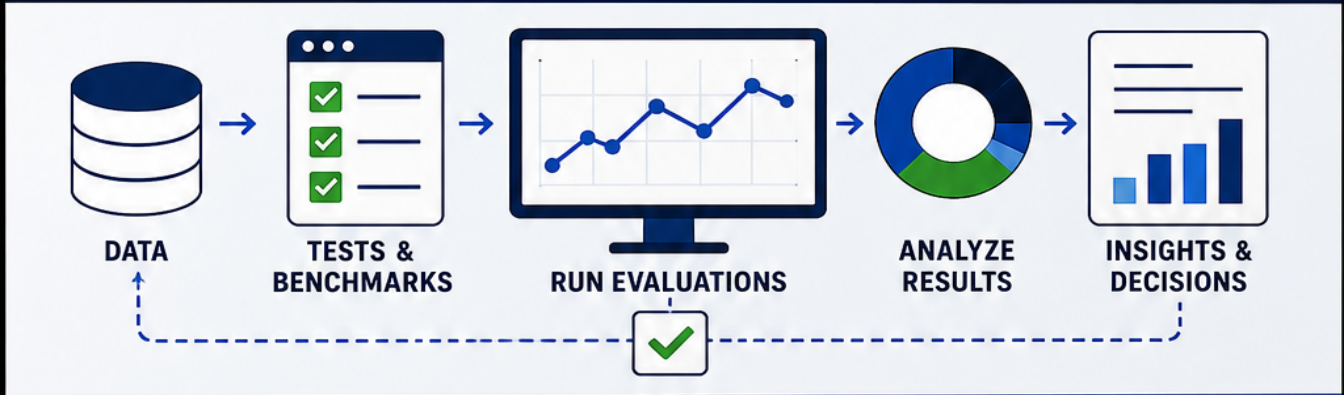
EVALUATION ENGINEERING FOR AI SYSTEMS

BUILDING RELIABLE EVALUATIONS FROM BENCHMARKS TO PRODUCTION





PHILLIPS MÜLLER

Evaluation Engineering for AI Systems

Building Reliable Evaluations from Benchmarks to Production

Steve Publications

This book is available at
https://leanpub.com/evaluationengineeringforaisystems

This version was published on 2026-07-25

Contents

Building Reliable Evaluations from Benchmarks to Production

This book is a comprehensive guide to evaluation engineering for artificial intelligence systems, written for machine learning engineers and AI practitioners who need to build trustworthy, production-ready AI applications. You will learn how to design rigorous evaluations, choose appropriate metrics, run statistically sound model comparisons, detect regressions before they reach users, evaluate agents and retrieval-augmented systems, test for safety and robustness, and build scalable evaluation infrastructure that integrates into your development pipeline. Every concept is explained with real-world examples, annotated Python code using modern frameworks, case studies from industry, and guidance drawn from the latest research. By the end, you will have a complete mental model of evaluation engineering as a first-class discipline and the practical skills to implement it in your own systems.

Introduction: Why Evaluation Is the Bottleneck

The Cost of Bad Evals: A Tale of Two Deployments

In early 2024, a healthcare startup deployed a large language model to summarize patient records for physicians. The model had scored well on standard benchmarks and passed a handful of manual tests by the engineering team. Within three weeks, two separate incidents came to light: the model had hallucinated a medication allergy in one patient's summary and omitted a critical lab result in another. Neither error would have caused immediate harm, but both revealed a pattern that the pre-deployment evaluation had missed. The evaluation consisted of twenty hand-picked examples, mostly straightforward cases, scored by engineers who were also the model's builders. There was no systematic coverage of edge cases, no statistical validation of the results, and no continuous monitoring after launch.

A few months later, a financial services company faced a similar decision: whether to deploy a customer service agent built on a fine-tuned language model. The team took a different approach. They constructed an evaluation suite of five hundred test cases drawn from real production conversations, covering happy paths, edge cases, and adversarial inputs. They used both programmatic checks for structured outputs and calibrated LLM judges for open-ended responses, validated against human expert ratings. They ran regression tests on every prompt change and fine-tuning iteration. Before deployment, they launched a shadow evaluation where the model processed live traffic without affecting customers, comparing its responses to the existing system. The shadow eval revealed a subtle but consistent failure mode: the model underperformed on questions involving multi-step account reconciliation. The team fixed the issue, re-ran the evaluation, and only then deployed to production. Six months later, with over a million interactions processed, the system had maintained its quality and caught zero safety incidents that would have required rollback.

These two stories illustrate a fundamental truth about modern AI development: the difference between a successful deployment and a problematic one is often not the model itself, but the quality of the evaluation process. Good evaluations do not guarantee success, but bad evaluations almost guarantee failure eventually. The question this book addresses is how to build good evaluations systematically, at scale, and with enough rigor that you can trust the results when making go or no-go decisions about your AI systems.

What Evaluation Engineering Actually Is

Evaluation engineering is the practice of designing, implementing, and maintaining systematic assessments of AI system behavior to answer specific questions: Does this model perform well enough for our use case? Did a change improve or degrade performance? Is the system safe and reliable in production? Are we measuring what actually matters to our users and stakeholders?

This definition deserves unpacking. First, evaluation engineering is not simply running a model on a benchmark dataset and reporting the score. Benchmarks are one tool among many, and treating them as the final word on model quality is a common mistake that leads to overconfidence in flawed evaluations. Second, evaluation engineering is not the same as testing in the traditional software engineering sense, although there is overlap. Software tests verify that deterministic code produces expected outputs for given inputs. AI systems are probabilistic and their behavior emerges from patterns learned during training, making evaluation more nuanced. A single input can produce different outputs across runs, and correctness often depends on context, intent, and domain knowledge rather than exact string matching.

Evaluation engineering sits at the intersection of several disciplines. It borrows from statistics (confidence intervals, significance testing, experimental design), software engineering (CI/CD pipelines, regression testing, test automation), machine learning research (benchmarking, ablation studies, error analysis), and domain expertise (understanding what quality means in healthcare, finance, customer service, or whatever your application targets). The “engineering” part refers to the systematic, repeatable, and scalable nature of the practice. You are building infrastructure and processes that produce trustworthy evidence about your AI system’s behavior, not just running ad-hoc checks.

A useful distinction to keep in mind is between three related but different concepts:

- **Benchmarking** measures model performance on standardized datasets, often for comparison against other models or tracking progress over time. Benchmarks are valuable for research and model selection, but they rarely capture the specifics of your production use case.
- **Testing** verifies that a system meets defined requirements, typically using a fixed set of cases with expected outcomes. Testing is essential for catching regressions and ensuring reliability, but it may not cover the full range of possible inputs or behaviors.
- **Evaluation** is broader: it encompasses benchmarking and testing, but also includes human assessment, statistical analysis, error investigation, continuous monitoring, and any other method that produces evidence about system quality. Evaluation engineering is the discipline of making all of these methods work together coherently.

The Eval Crisis in Modern AI Development

The field of AI evaluation is currently undergoing a crisis and a transformation simultaneously. Traditional evaluation metrics that served the machine learning community well for years are breaking down in the era of large language models and generative AI, while the stakes for getting evaluation right have never been higher.

Consider the classic metrics from natural language processing. BLEU, introduced in 2002, measures n-gram precision between generated text and reference translations. ROUGE, from 2004, measures recall-oriented overlap for summarization tasks. These metrics worked reasonably well when models produced relatively constrained outputs: machine translation with a narrow space of acceptable phrasings, or extractive summarization where the correct answer is a subset of the source text. They correlate imperfectly but acceptably with human judgment on those specific tasks.

For modern generative AI, these metrics largely fail. A language model can produce a perfectly accurate, fluent, and helpful response that shares almost no n-gram overlap with a reference answer. Conversely, it can reproduce the reference verbatim while hallucinating facts not present in the source material.

Metrics based on surface-level text similarity cannot distinguish between these cases. Research by Zhang et al. in their 2019 BERTScore paper demonstrated that traditional n-gram metrics fail to capture semantic equivalence in over sixty percent of paraphrase cases, and more recent work has shown that none of the standard metrics reliably detects factual errors.

The problem extends beyond metric failure to a deeper issue: the evaluation landscape is fragmented, opaque, and often misaligned with real-world needs. Model providers publish scores on curated benchmarks that showcase their strengths while obscuring weaknesses. These benchmarks become contaminated as models are trained on internet data that includes the benchmark itself, inflating reported performance. Research by Li et al. in 2024 found contamination levels reaching forty-five percent on commonly used benchmarks, meaning nearly half of the questions a model appears to "solve" may have been memorized rather than understood.

At the same time, organizations are deploying AI systems in high-stakes contexts where evaluation failures have real consequences. A hallucinated medical recommendation, a biased hiring decision, a jailbroken customer service agent leaking confidential information: these are not theoretical risks but documented incidents that have occurred in production. According to McKinsey's State of AI survey in early 2024, seventy-two percent of organizations had adopted AI in at least one business function, up from fifty-five percent a year earlier, with generative AI use nearly doubling. This rapid adoption pushed organizations to evaluate for real risks that traditional benchmarks do not measure.

The National Academy of Engineering published a report in Spring 2025 titled "Toward an Evaluation Science for Generative AI Systems" arguing that static benchmarks and ad-hoc audits fail to assess generative AI safety or performance in real-world contexts. The authors advocate for a rigorous evaluation science modeled on fields like aerospace, transportation, and pharmaceuticals, where systematic testing and continuous monitoring are not optional extras but fundamental requirements.

This is the landscape evaluation engineering must address: a field where the old tools no longer work reliably, the stakes are high, and the technology is advancing faster than the evaluation methods. The good news is that solutions are emerging. Techniques like LLM-as-a-judge evaluation, which uses strong language models to assess other models' outputs, have shown promising alignment with human preferences when implemented carefully. Frameworks

for RAG evaluation, agent evaluation, and safety testing are maturing rapidly. Statistical methods adapted from classical machine learning can be applied to LLM evaluation to produce defensible claims about model improvements. And the tooling ecosystem is growing, with open-source and commercial platforms that make it practical to build comprehensive evaluation systems.

This book is built on the conviction that evaluation engineering is not a luxury for well-funded research labs but a necessity for any organization deploying AI systems seriously. The methods and practices it describes are drawn from the best current research, industry experience, and practical engineering, and they are accessible to teams of varying sizes and budgets. You do not need an army of annotators or millions of dollars in compute to build good evaluations, but you do need a systematic approach and a clear understanding of what you are measuring and why.

How This Book Is Structured

The chapters that follow take you on a journey from foundational concepts through advanced techniques, always with a focus on practical application. The structure is designed so that each chapter builds on previous ones while remaining largely self-contained, allowing you to jump to topics most relevant to your current needs.

Chapter 1 establishes the foundations of AI evaluation: core terminology, the anatomy of an evaluation system, and where evaluations fit into the AI development lifecycle. If you are new to evaluation engineering, this is the place to start. Chapter 2 covers evaluation design principles that determine whether your evaluations will produce actionable, reliable results. These principles apply regardless of the specific metrics or tools you use, so they are worth understanding before diving into implementation details.

Chapters 3 through 5 cover the core toolkit of evaluation engineering: metrics for measuring model performance, benchmarks and datasets for standardized comparison, and automated evaluation techniques that scale beyond manual assessment. Chapter 6 is a deep dive into LLM-as-a-judge, one of the most important and widely used evaluation methods in modern AI development, covering both its power and its pitfalls.

Chapter 7 addresses human evaluation: when it is necessary, how to design it rigorously, and how to integrate human feedback with automated methods.

Chapter 8 brings statistical rigor into evaluation engineering, teaching you how to make defensible claims about model improvements using confidence intervals and significance testing. Chapter 9 covers systematic error analysis as a feedback loop for continuous improvement.

Chapters 10 and 11 move from offline evaluation to continuous practices: regression testing integrated into CI/CD pipelines, and production monitoring and observability for live systems. Chapter 12 addresses the unique evaluation challenges of agents and retrieval-augmented generation systems, which are increasingly common in production but break many standard evaluation assumptions.

Chapter 13 covers safety, robustness, and adversarial evaluation: how to test your systems for hallucinations, bias, jailbreak vulnerabilities, and other failure modes that can cause harm. Chapter 14 extends evaluation concepts to multimodal models and specialized domains like code generation. Chapter 15 addresses the practical realities of running evaluations at scale: cost optimization, infrastructure design, and the tradeoffs you will face in production environments.

The Conclusion synthesizes the book's themes and looks ahead to emerging challenges and opportunities in the field. Throughout, you will find annotated Python code examples using modern frameworks, case studies drawn from real-world deployments, comparisons between alternative approaches, and references to influential research papers and open-source projects for further exploration.

My hope is that this book becomes a reference you return to repeatedly as you build and maintain AI systems, not just something you read once and set aside. Evaluation engineering is a practice that improves with experience, and the field itself is still evolving. By understanding the principles, methods, and tradeoffs described here, you will be better equipped to navigate whatever comes next.

Chapter 1: Foundations of AI Evaluation

From Accuracy to Alignment: How Evaluation Evolved

To understand where AI evaluation is today, it helps to know where it came from. The history of ML evaluation is a story of increasing complexity, driven by the capabilities of the models we build and the demands of the applications they power.

In the early days of machine learning, evaluation was straightforward because the tasks were constrained. A spam classifier either correctly labeled an email or it did not. Metrics like accuracy, precision, recall, and F1 score provided a complete picture of performance for these binary classification tasks. When models improved, the metrics went up in ways that were easy to interpret and compare. The evaluation datasets were small enough to label by hand, and the relationship between metric improvements and real-world impact was direct: higher precision meant fewer legitimate emails marked as spam, higher recall meant more spam caught.

As natural language processing matured, the tasks became harder to evaluate. Machine translation, text summarization, and question answering produce open-ended outputs where multiple correct answers are possible. The community developed metrics like BLEU for translation and ROUGE for summarization, which measure overlap between generated output and human reference texts. These were imperfect proxies for quality but worked well enough to track progress over time. Researchers accepted that a ten-point improvement in BLEU score likely meant better translations, even if the metric did not capture every nuance of human judgment.

The arrival of large language models disrupted this comfortable arrangement. Modern LLMs can perform dozens of different tasks with varying degrees of competence, and their outputs are often long, creative, and context-dependent. Traditional metrics break down because they cannot handle the diversity and complexity of what these models produce. A model might

write excellent code but generate hallucinated medical advice, excel at logical reasoning but fail at following formatting instructions, or produce fluent text that is subtly biased in ways that only domain experts can detect.

This led to the emergence of new evaluation paradigms. The first was the multitask benchmark: instead of evaluating a model on a single task, test it across many to get a more complete picture of its capabilities. MMLU, introduced by Hendrycks et al. at ICLR 2021, became the canonical example, covering fifty-seven subjects from elementary mathematics to law and medicine through multiple-choice questions. Similarly, benchmarks like GSM8K for mathematical reasoning, HumanEval for code generation, and TruthfulQA for factuality emerged to measure specific dimensions of model behavior.

The second paradigm was human evaluation at scale. Since automated metrics proved inadequate, researchers turned to crowdsourcing platforms and expert panels to rate model outputs directly. The LMSYS Chatbot Arena, launched in 2023, pioneered a crowdsourced battle format where users compare two anonymous model responses and vote for the better one, generating an Elo-based leaderboard that tracks relative quality. This approach produces preference data that correlates well with human judgment but is expensive and slow to collect.

The third paradigm, which has become dominant in practice, is LLM-as-a-judge: using a strong language model to evaluate the outputs of other models. Zheng et al.'s 2023 NeurIPS paper demonstrated that GPT-4 could judge chatbot responses with over eighty percent agreement with human preferences, opening the door to scalable, automated evaluation of open-ended tasks. This technique is now used extensively in both research and production, though it comes with its own challenges around bias, calibration, and reliability.

The fourth paradigm is safety and alignment evaluation, driven by the recognition that capability alone is insufficient. As models became more powerful, the risks of deploying them without careful evaluation grew. Benchmarks like HELM from Stanford's CRFM, which evaluates models across capabilities, robustness, fairness, and toxicity, represent an effort to create holistic assessment frameworks. Regulatory requirements like the EU AI Act have added further pressure to evaluate systems for bias, transparency, and safety before deployment.

This evolution has not been linear or uncontested. Each new paradigm has

raised questions about validity, reproducibility, and whether we are measuring what matters. Benchmark contamination, where models memorize test data during training, has undermined the reliability of many standard benchmarks. LLM judges have their own biases and failure modes that can distort evaluation results. Human evaluation is subjective and expensive, making it impractical for continuous use. And safety evaluations often rely on static test sets that may not capture emerging risks or real-world deployment contexts.

The field is still finding its footing. What was once a relatively narrow discipline focused on classification metrics has become a broad, interdisciplinary practice encompassing statistics, software engineering, domain expertise, and ethics. Evaluation engineering, as we define it in this book, is the systematic application of all these methods to produce trustworthy evidence about AI system behavior. Understanding this history helps explain why the field looks the way it does today and why many evaluation practices feel provisional or contested: they are, because the technology has outpaced our ability to measure it rigorously.

The Anatomy of an Evaluation

Every evaluation, whether a simple accuracy calculation on a classification task or a complex multi-dimensional assessment of a production agent, consists of the same core components. Understanding these components gives you a vocabulary for thinking about evaluations systematically and a framework for designing new ones.

At the most basic level, an evaluation needs three things: test cases, a model to evaluate, and a way to score the model's outputs. But each of these elements has structure and choices that affect the quality of your evaluation.

Test cases are the inputs you present to the model, often paired with expected outcomes or reference answers. A test case might be as simple as a question and its correct answer, or as complex as a multi-turn conversation scenario with constraints on acceptable behavior. The quality of your test cases determines what your evaluation can and cannot tell you. If your test cases do not cover the kinds of inputs the model will see in production, your evaluation will give you false confidence.

Test cases have several important properties:

- **Representativeness**: Do they reflect the distribution of real-world inputs the model will encounter? An evaluation on medical questions is useless for a customer service bot.
- **Coverage**: Do they span the full range of tasks, edge cases, and difficulty levels relevant to your use case? Evaluating only easy cases misses critical failure modes.
- **Clarity**: Are the expected outcomes unambiguous? Vague test cases produce noisy scores that are hard to interpret.
- **Independence**: Are the test cases independent of each other and of the model's training data? Dependent or contaminated cases inflate scores artificially.

Building high-quality test cases is often the most labor-intensive part of evaluation engineering, and it deserves more attention than it typically gets. We will cover test case design in depth in Chapter 2.

Scorers (also called evaluators, judges, or metrics) determine whether a model's output is correct or good for each test case. Scorers can range from simple programmatic checks to complex human judgment protocols:

- **Exact match**: The output must equal the expected answer exactly. Useful for tasks with deterministic answers like arithmetic or code execution, but too strict for open-ended generation.
- **Programmatic validation**: Use rules, regex patterns, or schema checks to validate specific properties of the output. For example, checking that a JSON response has the required fields, or that a generated SQL query executes without error.
- **String similarity metrics**: BLEU, ROUGE, edit distance, and similar measures compare the model's output to reference text. These capture surface-level similarity but miss semantic equivalence and factual accuracy.
- **Semantic similarity**: Embedding-based methods like BERTScore or cosine similarity in a learned vector space measure meaning-level alignment between outputs and references. Better than n-gram metrics for paraphrases, but still require references and cannot detect hallucinations.
- **LLM-as-a-judge**: A language model evaluates the output against criteria specified in a prompt. Highly flexible and scalable, but subject to biases and calibration issues that must be managed carefully.

- **Human evaluation**: People rate outputs based on guidelines or rubrics. The gold standard for quality assessment, but slow, expensive, and subject to inter-rater variability.

The choice of scorer depends on your task, your tolerance for noise, and your resource constraints. In practice, you will often use multiple scorers in combination to get a more complete picture. For example, you might use programmatic checks for structural correctness, an LLM judge for content quality, and periodic human review to calibrate the automated metrics.

Aggregators combine individual scores across test cases into summary statistics that are interpretable and actionable. Common aggregations include:

- **Mean score**: The average across all test cases. Simple and intuitive, but sensitive to outliers and does not reveal the distribution of performance.
- **Percentiles**: The median, 90th percentile, or other quantiles give you a sense of typical and tail behavior. Important for production systems where worst-case performance matters.
- **Confidence intervals**: Statistical bounds around your mean estimate that quantify uncertainty. Essential for making defensible claims about model improvements.
- **Category breakdowns**: Scores grouped by task type, difficulty level, or other dimensions reveal patterns that overall averages hide.

The aggregation method you choose should reflect what you care about. If you are optimizing for typical user experience, the median might be more relevant than the mean. If you need to ensure reliability under edge cases, tail metrics matter more. We will cover statistical rigor in Chapter 8, but the key point here is that aggregation is not a mechanical step: it is a design choice that affects how you interpret your evaluation results.

Reports and dashboards present the evaluation results to stakeholders in a form they can understand and act on. A good report answers the questions your audience cares about: Did the change help or hurt? Are we meeting our quality targets? Where are the biggest failure modes? Reports should be clear, visual, and focused on actionable insights rather than raw numbers. They should also be reproducible: anyone who runs the same evaluation pipeline should get the same results.

Putting it all together, an evaluation system is a pipeline that takes test cases and model outputs, applies scorers to produce individual scores, aggregates

those scores into summary statistics, and presents the results through reports or dashboards. Each component in this pipeline is a design decision with tradeoffs, and the quality of your evaluation depends on getting each one right.

[Figure 1.1: Anatomy of an Evaluation System] A diagram showing the complete evaluation pipeline from left to right: (1) Test Cases dataset feeds into (2) the Model under evaluation, which produces (3) Outputs. Those outputs are scored by (4) Scorers (programmatic checks, LLM judges, human reviewers), producing individual scores that flow into (5) Aggregators computing summary statistics. Results feed into (6) Reports and Dashboards for stakeholders. Arrows indicate data flow; a feedback loop from Error Analysis back to Test Cases improvement is shown at the bottom.

Where Evals Live in the Development Lifecycle

Evaluations are not a single event that happens once before deployment. They are continuous activities that occur throughout the AI development lifecycle, each serving a different purpose and using different methods. Understanding where evaluations fit in this lifecycle helps you design the right evaluation for each stage and avoid the common mistake of treating pre-deployment testing as sufficient.

Model selection and benchmarking. Before you even start building your application, you need to choose which base model to use. This is typically done through benchmarking: running candidate models on standardized datasets to compare their capabilities. Benchmarks like MMLU, GSM8K, and HumanEval provide a common language for comparing models across different providers and sizes.

However, benchmark scores are only a starting point. A model that leads on general benchmarks may not be the best choice for your specific use case. If you are building a legal document summarizer, you need to evaluate candidates on legal texts, not just general knowledge questions. This is where custom evaluations come in: building test suites tailored to your domain and task to validate that benchmark performance translates to real-world utility.

Fine-tuning and prompt engineering. Once you have selected a base model, you may fine-tune it on domain-specific data or craft specialized prompts to improve its behavior. Each iteration of this process needs evaluation to determine whether the changes are helping or hurting. These

evaluations should be fast and automated so you can iterate quickly, but they also need to be reliable enough that you trust the results.

A common pattern here is the golden dataset: a small, carefully curated set of test cases with known correct answers that you run on every iteration. Golden datasets serve as regression tests, ensuring that improvements in one area do not come at the cost of degradation elsewhere. They should be protected from contamination and updated periodically to stay relevant.

Pre-deployment gates. Before releasing a model or system to users, you need a comprehensive evaluation that covers all dimensions of quality: accuracy, safety, robustness, latency, cost, and user experience. This is the final check before going live, and it should be rigorous enough to catch problems that earlier evaluations missed.

Pre-deployment evaluations often combine multiple methods: automated tests for regression detection, LLM judges for content quality, human review for edge cases and safety, and load testing for performance characteristics. The results feed into a go or no-go decision, ideally with clear, pre-defined criteria for what constitutes acceptable performance. Without explicit criteria, these gates become subjective and vulnerable to pressure to ship.

Continuous monitoring in production. Once your system is live, evaluation does not stop. Production traffic reveals edge cases and failure modes that were impossible to anticipate in testing. Continuous monitoring tracks quality metrics over time, detects drift and degradation, and provides data for ongoing improvement.

Production monitoring differs from offline evaluation in several ways. First, you often lack ground truth labels for real user inputs, so you need to rely on proxy metrics like user engagement, thumbs up/down feedback, or automated quality signals. Second, the stakes are higher: a quality regression in production affects real users immediately, so you need fast detection and response mechanisms. Third, the evaluation must be lightweight enough to run continuously without prohibitive cost or latency overhead.

We will cover production monitoring in depth in Chapter 11, but the key insight is that offline evaluations and production monitoring are complementary, not substitutes. Offline evals tell you whether your system is ready to deploy; production monitoring tells you whether it stays ready over time.

Iterative improvement. Evaluation data feeds back into the development cycle. Error analysis on evaluation failures reveals patterns that inform new

training data, prompt improvements, or architectural changes. User feedback from production identifies gaps in your test coverage. Each cycle of evaluate, analyze, improve, re-evaluate makes your system more reliable and your evaluations more effective.

This iterative loop is sometimes called the evaluation flywheel: good evaluations produce better models, which make it easier to build better evaluations, and so on. Teams that institutionalize this process tend to outperform those that treat evaluation as a one-time checkbox exercise.

Internal vs External Validation

A critical distinction in evaluation engineering is between internal and external validation, and the dangers of confusing the two. This distinction matters because practices that are appropriate for internal development can produce dangerously misleading results if applied to external assessment.

Internal validation uses data and methods that are accessible to the development team during model building and iteration. This includes:

- Development sets used to tune hyperparameters, select prompts, or guide fine-tuning
- Synthetic test cases generated by the team or by the model itself
- Evaluations run on models developed in-house, where the team has full visibility into training data and architecture
- Benchmarks that the team knows are not contaminated with training data

Internal validation is essential for rapid iteration and debugging. It tells you whether your changes are moving the needle in the right direction. But internal validation scores are inherently optimistic: they reflect performance under controlled conditions on data that may be easier, more representative of training, or otherwise biased in the model's favor.

External validation uses independent data and methods to assess model performance as it would appear to an outside evaluator or end user. This includes:

- Holdout test sets that have never been used during development and are strictly separated from training data

- Public benchmarks evaluated under controlled conditions to minimize contamination risk
- Human evaluations conducted by annotators who are not part of the development team
- Production monitoring on live traffic that the model has not seen before

External validation is what you rely on for go/no-go decisions, public claims about model quality, and regulatory compliance. It provides an unbiased estimate of how the model will perform in the real world. The key requirement is independence: the evaluation data must not have influenced the model's training or the team's development decisions.

The contamination problem. The line between internal and external validation has become dangerously blurred in the era of large language models, primarily due to data contamination. When models are trained on internet-scale corpora, they inevitably encounter some fraction of any public benchmark or dataset. A model that appears to solve a reasoning question may have simply memorized the answer from its training data rather than genuinely understanding the problem.

Contamination can be direct (the exact test case appears in training data) or indirect (related examples, tutorials, or solutions appear, giving the model patterns it can exploit). Detecting contamination is difficult because model providers rarely disclose their full training data, and even when they do, the scale makes comprehensive checking impractical.

Several strategies help mitigate contamination risk:

- **Canary insertion**: Embedding unique identifiers (like a UUID string) in benchmark datasets so you can definitively detect if they appear in model outputs. BIG-bench pioneered this approach.
- **Temporal filtering**: Using only data published after the model's training cutoff date. This is imperfect because the training cutoff is often unclear and data leaks happen, but it helps.
- **Private holdouts**: Maintaining evaluation datasets that are never published or shared, evaluated through server-side APIs so the raw data cannot leak. GLUE and SQuAD used this approach historically.
- **Dynamic benchmarks**: Continuously generating new test cases to stay ahead of contamination, though this requires ongoing effort and quality control.

- **Perplexity-based detection**: Contaminated models tend to show abnormally low perplexity on contaminated samples compared to reference models, which can flag suspicious performance.

Best practices for managing validation data. To keep internal and external validation distinct in your own work:

1. Maintain a strict separation between development, validation, and test sets. Never use test set data to guide model selection, prompt tuning, or hyperparameter optimization.
2. Treat your holdout test set as sacred. Limit access, log all uses, and rotate it periodically to prevent gradual contamination through repeated exposure.
3. Be transparent about what kind of validation your reported scores represent. A score on a development set is not the same claim as a score on an independent holdout.
4. When using public benchmarks, be aware of contamination risk and interpret high scores skeptically, especially for models trained after the benchmark was published.
5. Build custom evaluations for your specific use case that cannot be contaminated because they are based on proprietary data or dynamically generated scenarios.

Understanding the distinction between internal and external validation, and the risks of contamination, is fundamental to evaluation engineering. It protects you from overconfidence in misleading results and ensures that your claims about model quality are defensible.

Key Takeaways

- AI evaluation has evolved from simple classification metrics to complex, multi-dimensional assessment frameworks driven by the capabilities and risks of modern models.
- Every evaluation consists of test cases, scorers, aggregators, and reports, each involving design choices that affect the quality and interpretability of results.

- Evaluations occur throughout the development lifecycle: model selection, fine-tuning, pre-deployment gates, production monitoring, and iterative improvement. Each stage has different requirements.
- Internal validation supports rapid iteration but produces optimistic estimates; external validation provides unbiased assessment for go/no-go decisions and public claims.
- Data contamination threatens the validity of many standard benchmarks, requiring careful management of evaluation data and skeptical interpretation of reported scores.
- Evaluation engineering is the systematic practice of combining all these methods to produce trustworthy evidence about AI system behavior, not just running ad-hoc tests or chasing benchmark leaderboards.

Chapter 2: Designing Effective Evaluations

Principles of Good Evaluation Design

Before you write a single line of evaluation code or collect a single test case, you need to think about what makes an evaluation good. A poorly designed evaluation can be worse than no evaluation at all, because it gives you false confidence in your system's quality. This chapter establishes the principles that guide all the specific techniques and tools we will cover later.

Five core principles define effective evaluation design: validity, reliability, coverage, actionability, and efficiency. Each addresses a different potential failure mode, and together they form a checklist you can apply to any evaluation you build.

Validity asks whether your evaluation measures what it claims to measure. A valid evaluation produces scores that correspond to real-world quality as defined by your stakeholders. If you are building a customer service bot, validity means your evaluation metrics correlate with actual customer satisfaction, not just internal engineering benchmarks.

Validity has several dimensions:

- **Construct validity**: Does your metric capture the underlying concept you care about? For example, does BLEU score actually measure translation quality, or just n-gram overlap? Research shows it captures only part of the picture.
- **Content validity**: Does your evaluation cover the full range of behaviors and scenarios that matter? Evaluating a medical AI only on straightforward diagnostic questions misses rare but critical edge cases.
- **Criterion validity**: Do your evaluation scores correlate with an external gold standard, such as human judgment or business outcomes? This is the strongest form of validation.

Common threats to validity include using proxies that diverge from actual quality (optimizing for benchmark scores while users experience worse service), narrow test sets that miss important cases, and scorers that reward the wrong behaviors (an LLM judge that prefers verbose responses regardless of accuracy).

Reliability asks whether your evaluation produces consistent results when repeated. A reliable evaluation gives you the same answer today and tomorrow, assuming the model has not changed. Unreliable evaluations make it impossible to tell whether a score change reflects a real model improvement or just noise in the measurement process.

Sources of unreliability include:

- **Stochastic model behavior**: Language models with temperature greater than zero produce different outputs on each run, so a single evaluation may not represent typical performance.
- **Subjective scorers**: Human annotators and LLM judges introduce variability based on individual interpretation, mood, or prompt sensitivity.
- **Environmental factors**: API latency, rate limiting, or infrastructure issues can cause evaluations to fail or produce degraded outputs unrelated to model quality.

Strategies for improving reliability include running multiple trials and averaging results, using deterministic settings when possible (temperature of zero), calibrating scorers with clear guidelines and examples, and implementing retry logic for transient failures. We will cover statistical methods for quantifying and managing unreliability in Chapter 8.

Coverage asks whether your evaluation tests enough of the relevant behavior space to give you confidence that the system will work in production. An evaluation with perfect validity and reliability on ten test cases tells you almost nothing about how the system handles the millions of diverse inputs it will see in the real world.

Good coverage requires:

- **Diversity**: Test cases should span different topics, formats, difficulty levels, and edge conditions relevant to your use case.
- **Representativeness**: The distribution of test cases should match the distribution of production traffic as closely as possible.

- **Edge cases**: Include inputs that are unusual, ambiguous, adversarial, or otherwise likely to expose failure modes.
- **Balance**: Avoid over-representing any single category unless it is proportionally important in production.

Coverage is often the hardest principle to satisfy because building comprehensive test suites is labor-intensive and requires domain expertise. The techniques we will cover for synthetic data generation, golden dataset construction, and continuous test case creation all address this challenge.

Actionability asks whether your evaluation results tell you something you can do. An evaluation is not useful if it produces a single aggregate score that goes up or down without explaining why or indicating what to fix. Actionable evaluations break down performance into dimensions you can improve, highlight specific failure patterns, and connect to concrete engineering decisions.

Examples of actionable design:

- Instead of reporting only overall accuracy, report accuracy by task type so you know which capabilities need improvement.
- Instead of flagging a response as "bad," classify the error type (hallucination, formatting violation, safety issue) so engineers can target fixes.
- Include reference outputs or expected behaviors alongside scores so reviewers can understand what went wrong.
- Tie evaluation metrics to business KPIs so stakeholders understand the impact of quality changes.

Actionability is especially important for iterative development. When you are tuning prompts, selecting models, or fine-tuning on new data, you need evaluations that tell you not just whether a change helped but where and why, so you can make informed tradeoffs.

Efficiency asks whether your evaluation can run fast enough and cheaply enough to be practical. An evaluation that takes three days to complete and costs ten thousand dollars is technically sound but operationally useless if you need to run it on every pull request. Efficient evaluations find the right balance between rigor and resource constraints.

Strategies for efficiency:

- **Tiered evaluation**: Run fast, cheap checks (programmatic validation, small test sets) frequently, and reserve expensive methods (human review, large-scale LLM judging) for critical milestones.
- **Caching**: Store model outputs and scores so you do not recompute them unnecessarily when only one component of the pipeline changes.
- **Parallelization**: Run evaluations across multiple test cases or models concurrently to reduce wall-clock time.
- **Strategic sampling**: Evaluate a representative subset of cases when full evaluation is too expensive, using statistical methods to estimate population-level performance.

Efficiency is not about cutting corners; it is about allocating resources wisely so that you can maintain rigorous evaluation practices continuously rather than as occasional, heroic efforts.

These five principles are interdependent and sometimes in tension. Improving coverage may reduce efficiency. Increasing rigor for validity may introduce complexity that reduces reliability if not managed carefully. Good evaluation design is the art of satisfying all five principles well enough for your specific context, recognizing that perfection is rarely achievable but systematic improvement is always possible.

Defining What Matters: Requirements-Driven Evaluation

One of the most common evaluation failures is measuring the wrong thing. Teams optimize for benchmark scores, latency, or token efficiency while their users experience hallucinated answers, biased outputs, or frustrating interactions. This happens when evaluation is disconnected from the actual requirements of the system and the needs of its stakeholders.

Requirements-driven evaluation starts by asking: what does success look like for this system, as defined by the people who will use it and depend on it? The answer determines what you measure, how you measure it, and what thresholds constitute acceptable performance.

The first step is stakeholder alignment. Different stakeholders care about different things: engineers care about correctness and reliability, product managers care about user satisfaction and engagement, legal teams care about

compliance and liability, executives care about cost and competitive positioning. A requirements-driven evaluation process explicitly surfaces these perspectives and translates them into measurable criteria.

For example, consider a RAG-based internal knowledge assistant for a large company:

- **Engineering requirement**: The system should answer questions accurately based on the provided documents, with hallucination rate below five percent.
- **Product requirement**: Users should find answers within two interactions, with satisfaction scores above four out of five.
- **Legal requirement**: The system must not reveal confidential information from restricted documents, and all responses must be traceable to source materials.
- **Executive requirement**: The system should reduce time spent searching for information by at least twenty percent, measured through usage analytics.

Each of these requirements maps to specific evaluation metrics: faithfulness and factual accuracy for engineering, task completion rate and user ratings for product, access control validation and citation checks for legal, and before-and-after productivity studies for executive impact. Without this mapping, you might build an evaluation that optimizes for retrieval precision while ignoring whether users actually find the answers helpful or whether confidential data leaks occur.

The second step is defining acceptance criteria: explicit thresholds that determine whether the system is ready to deploy or whether a change is acceptable. Vague goals like "improve quality" or "reduce hallucinations" are not actionable because they do not specify how much improvement is needed or how to measure it. Clear criteria might look like:

- Context recall above eighty-five percent on the golden dataset
- Faithfulness score above ninety percent as measured by LLM judge calibrated against human review
- Zero critical safety violations (PII leakage, policy breaches) in pre-deployment testing
- P95 latency below two seconds under expected load

Acceptance criteria should be set before development begins, not retrofitted after you see the results. Setting thresholds post-hoc invites cherry-picking and undermines the integrity of the evaluation process. They should also be revisited periodically as requirements evolve and as you gain more data about what levels of performance are achievable and necessary.

The third step is connecting evaluation metrics to downstream decisions. An evaluation is only valuable if its results influence what happens next. Define in advance what triggers which actions:

- If faithfulness drops below ninety percent on the regression suite, block the deployment.
- If hallucination rate exceeds five percent in production monitoring, trigger an incident review and potential rollback.
- If a model change improves accuracy by more than three percentage points with statistical significance, promote it to the candidate pool for A/B testing.

This connection between evaluation and decision-making is what transforms evaluation from a reporting exercise into an engineering control mechanism. It ensures that quality gates are enforced consistently rather than overridden by shipping pressure or optimism bias.

A practical technique for requirements-driven evaluation is to write user stories or scenarios that capture real use cases, then derive test cases directly from them. For each scenario, define what a successful interaction looks like and what constitutes failure. This grounds your evaluation in concrete behaviors rather than abstract metrics and makes it easier to communicate with non-technical stakeholders about what quality means for the system.

Case Study: Requirements-Driven Red Teaming at a Global Financial Services Firm. A global financial services company used machine learning models to detect and intercept fraud during real-time financial transactions. Before deploying model updates, they relied on standard accuracy metrics on historical transaction data. However, their security team raised concerns: traditional metrics measure how well the model classifies known fraud patterns, but do not test whether an adversary could deliberately craft transactions that evade detection.

The company engaged HiddenLayer's professional services team to conduct a structured red team evaluation. The engagement began with requirements gathering across stakeholders:

- **Security requirement**: Adversarial attack success rate must be below five percent; the model should resist deliberate evasion attempts.
- **Business requirement**: False positive rate must stay below two percent to avoid disrupting legitimate customer transactions.
- **Regulatory requirement**: The system must maintain audit trails for all flagged and cleared transactions, and model decisions must be explainable to regulators.

The red teaming exercise was structured in four phases, aligned with NIST AI RMF guidance:

1. **Planning** (one week): Defined scope covering three ML models used in fraud detection, established rules of engagement, and selected attack vectors based on MITRE ATLAS threat taxonomy.
2. **Execution** (three weeks): Conducted adversarial testing using both automated tools and human experts. Attacks included evasion attempts (crafting transactions that look legitimate but are fraudulent), poisoning scenarios (testing model robustness to corrupted training data), and prompt injection tests for any LLM components in the investigation workflow.
3. **Evaluation**: Measured Attack Success Rate (ASR) across all models and attack types. Results showed ASR ranging from eight to twenty-three percent on the initial models, well above the five percent threshold.
4. **Remediation** (four weeks): The team implemented input validation hardening, ensemble defenses combining multiple detection models, and anomaly scoring that flagged suspicious patterns even when individual classifiers were uncertain.

After remediation, re-testing showed ASR dropped to below three percent across all models, meeting the security requirement. The false positive rate remained at 1.7 percent, satisfying the business constraint. The entire engagement cost approximately $85,000 in professional services fees plus two weeks of internal engineering time for remediation, but identified vulnerabilities that could have resulted in millions in fraudulent transactions if exploited.

This case study illustrates several principles: requirements from multiple stakeholders (security, business, regulatory) shaped the evaluation design; explicit thresholds (ASR below five percent) made results actionable; and

structured methodology (planning, execution, evaluation, remediation) ensured comprehensive coverage rather than ad-hoc testing. Most importantly, the evaluation tested for what actually matters in production: whether an intelligent adversary could break the system, not just how well it performs on clean historical data.

Constructing Test Cases That Reveal Truth

Test cases are the foundation of any evaluation, and their quality determines everything that follows. A sophisticated scorer running on garbage test cases produces garbage results. This section covers how to construct test cases that actually reveal how your system will behave, rather than giving you comforting but misleading scores.

Start from production data. The best source of test cases is real user inputs from production or closely related systems. Production data captures the actual distribution of queries, edge cases, ambiguities, and failure modes that engineering intuition alone would miss. If you do not yet have production data, use data from pilot deployments, beta tests, or analogous systems in similar domains.

When using production data, be careful about privacy and compliance. Anonymize personally identifiable information, respect data usage agreements, and ensure that test cases do not contain sensitive information that should not be shared with evaluation teams or external annotators. A common pattern is to create synthetic variants of real queries that preserve the structure and difficulty while removing identifying details.

Ensure diversity across dimensions. A representative test suite must span multiple dimensions of variation:

- **Task types**: If your system handles multiple kinds of requests (questions, commands, creative tasks, analysis), include all of them in proportion to their production frequency.
- **Difficulty levels**: Include easy cases that the system should handle reliably, medium cases that are challenging but solvable, and hard cases that probe the limits of capability. A suite with only easy cases produces inflated scores; a suite with only hard cases may not reflect typical performance.

- **Topics and domains**: Cover the full range of subjects your system will encounter, including less common topics that may expose knowledge gaps.
- **Formats and styles**: Test different input formats (short queries, long documents, structured data, natural language), different tones (formal, casual, technical), and different languages if applicable.
- **Edge cases**: Include ambiguous inputs, contradictory information, out-of-domain queries, adversarial prompts, and other scenarios that are likely to cause failures.

A useful technique is stratified sampling: divide your production data into strata based on relevant dimensions (task type, topic, difficulty), then sample proportionally from each stratum to ensure coverage. This prevents any single category from dominating the evaluation and makes it easier to diagnose where problems occur.

Calibrate difficulty. Not all test cases are equally informative. Very easy cases do not distinguish between good and mediocre models; very hard cases may be impossible for any current model, making them useless for tracking improvement. The most valuable test cases are those at the "right" difficulty level: hard enough to expose real differences in capability, but solvable by a competent system.

You can calibrate difficulty empirically by running candidate models on a large pool of test cases and analyzing the score distribution. Cases where all models score near one hundred percent are too easy; cases where all models score near zero are too hard (or poorly specified). Focus your evaluation on cases in the middle range where scores vary, because those are the cases that will differentiate model quality and guide improvement.

Some frameworks formalize this through item response theory, a statistical method originally developed for educational testing that models the probability of a correct response as a function of both the examinee's ability and the item's difficulty. While full IRT analysis is complex, the basic intuition is simple: track which test cases discriminate well between models and prioritize those in your evaluation suite.

Write unambiguous expected outcomes. A test case is only useful if you can determine whether the model's response is correct or good. Ambiguous expected outcomes introduce noise that makes scores unreliable and hard to interpret.

For tasks with deterministic answers (arithmetic, code execution, factual lookup), the expected outcome is clear: the model must produce the correct answer. For open-ended tasks, you need to define acceptance criteria explicitly:

- Instead of "the response should be helpful," specify what helpful means: "the response should directly address the user's question, provide accurate information, and suggest next steps if applicable."
- Instead of "the response should follow the format," provide a concrete schema or example of acceptable formatting.
- For rubric-based scoring, define each level of the rubric with specific, observable criteria rather than vague descriptors like "good" or "poor."

Few-shot examples are particularly effective for clarifying expected outcomes. Including three to five examples of inputs paired with ideal outputs in your evaluation guidelines dramatically improves scorer consistency, whether the scorer is human or an LLM judge.

Include negative and adversarial cases. Most test suites over-represent positive cases: inputs where the system is expected to succeed. This creates a coverage gap that lets critical failures slip through. A well-designed evaluation includes:

- **Negative cases**: Inputs where the correct behavior is to decline, refuse, or respond cautiously (e.g., out-of-domain queries, requests for harmful content, questions the system does not have enough information to answer). These test whether the system knows its limits.
- **Adversarial cases**: Inputs designed to provoke failures through prompt injection, jailbreak attempts, distribution shift, or other attack vectors. These test robustness and safety.
- **Contrastive pairs**: Two similar inputs that should elicit different responses (e.g., one asking for general information and another asking for medical advice). These test whether the system can distinguish between subtly different requirements.

Adversarial testing is especially important for systems deployed in open environments where malicious users will actively try to break them. We will cover adversarial evaluation in depth in Chapter 13, but the key point here

is that your test suite should include some adversarial cases as a standard practice, not just as an afterthought.

Maintain version control and provenance. Test cases are first-class artifacts in your codebase, deserving the same treatment as model code or application logic. Version control your test suites so you can track changes, revert problematic updates, and understand how the evaluation has evolved over time. Record the provenance of each test case: where it came from (production data, synthetic generation, manual creation), when it was added, and any known limitations or biases.

This discipline pays off when you need to debug evaluation failures, investigate score changes, or audit your evaluation process for compliance or research purposes. It also enables collaboration across teams: anyone can understand what the evaluation tests and why, not just the original authors.

The Golden Dataset Pattern

The golden dataset is one of the most important patterns in evaluation engineering. A golden dataset is a small, carefully curated collection of test cases with high-quality reference answers that serve as the canonical regression suite for your system. It is run on every model change, prompt update, or configuration modification to ensure that quality does not regress.

The golden dataset pattern is borrowed from software engineering's concept of regression tests: a fixed set of tests that verify known-correct behavior is preserved as the system evolves. In AI, the analogy holds but with important differences due to the probabilistic nature of model outputs and the open-endedness of many tasks.

Why golden datasets matter. Without a golden dataset, you have no reliable way to detect regressions. You might notice quality degradation through user complaints or production monitoring, but by then the damage is done. A golden dataset catches problems early, before they reach users, and provides fast feedback during development so engineers can iterate confidently.

Golden datasets also serve as a stable reference point for model comparison. When you are evaluating multiple candidate models or fine-tuning variants, running them all on the same golden dataset ensures that differences in scores reflect real capability differences rather than variations in test coverage.

Building a golden dataset. A good golden dataset has the following characteristics:

- **Small but representative**: Typically fifty to two hundred cases, chosen to cover the most important task types, difficulty levels, and edge cases. The goal is not comprehensive coverage (that is the job of larger evaluation suites) but high signal-to-noise ratio for regression detection.
- **High-quality references**: Each case has a carefully validated expected outcome, reviewed by domain experts if necessary. The quality of the references determines the reliability of the scores.
- **Stable**: Once established, the golden dataset changes infrequently. Frequent changes make it hard to track trends over time and defeat the purpose of regression testing. When you do add cases, mark them as new so you can analyze their impact separately.
- **Protected from contamination**: The golden dataset should never be used for training, fine-tuning, or prompt optimization. It must remain an independent holdout to provide unbiased assessment. Store it separately from development data and restrict access to prevent accidental leakage.

The process for building a golden dataset typically involves:

1. Collecting candidate cases from production data, user stories, or domain experts.
2. Filtering for diversity across task types, difficulty levels, and edge cases.
3. Writing or validating reference answers with clear acceptance criteria.
4. Reviewing the suite to ensure coverage of critical scenarios and removal of ambiguous or broken cases.
5. Establishing baseline scores on the current production model to use as a regression threshold.

Using golden datasets in CI/CD. Golden datasets are most powerful when integrated into your continuous integration pipeline, running automatically on every relevant change. A typical workflow:

1. Developer submits a pull request with a prompt change, model swap, or configuration update.
2. CI pipeline runs the golden dataset evaluation against the proposed change.

3. If scores drop below the regression threshold (e.g., more than two percentage points on overall accuracy, or any critical safety violation), the pipeline fails and blocks merge.
4. Developer investigates failures, makes corrections, and resubmits.

This process requires the golden dataset evaluation to be fast enough to run within CI time constraints (typically minutes, not hours) and robust enough to produce reliable results without manual intervention. These requirements often mean using programmatic scorers or calibrated LLM judges rather than human review for the golden dataset, reserving human evaluation for periodic audits and complex cases.

Maintaining golden datasets over time. Golden datasets are not set-and-forget artifacts. As your system evolves, the types of queries it handles change, new failure modes emerge, and the difficulty level of your model may shift relative to the test cases. A golden dataset that was well-calibrated six months ago may no longer be informative today.

Maintenance practices include:

- **Periodic review**: Every quarter or so, review the golden dataset to ensure cases are still relevant, references are still correct, and coverage matches current production patterns.
- **Gradual refresh**: Add new cases to cover emerging use cases and failure modes, while retiring cases that have become too easy or irrelevant. Do this incrementally to preserve historical comparability.
- **Difficulty calibration**: If all models consistently score near one hundred percent on the golden dataset, it has become too easy and is no longer discriminating between quality levels. Replace some cases with harder ones.
- **Cross-validation with production**: Regularly compare golden dataset scores with production quality metrics. If they diverge significantly (e.g., golden dataset scores are stable but user satisfaction is declining), your golden dataset may have lost representativeness and needs updating.

The golden dataset pattern is simple but powerful. Teams that maintain a well-curated golden dataset integrated into their CI/CD pipeline catch regressions early, iterate with confidence, and maintain higher quality over time than teams that rely on ad-hoc testing or benchmark scores alone.

Key Takeaways

- Good evaluation design satisfies five principles: validity (measuring what matters), reliability (consistent results), coverage (testing enough of the behavior space), actionability (producing insights you can act on), and efficiency (running fast and cheap enough to be practical).
- Requirements-driven evaluation starts with stakeholder needs and translates them into measurable criteria, acceptance thresholds, and decision rules, ensuring that evaluation is aligned with real-world success rather than abstract metrics.
- Effective test cases are derived from production data, diverse across relevant dimensions, calibrated for appropriate difficulty, specified with unambiguous expected outcomes, and include negative and adversarial scenarios.
- The golden dataset pattern provides a stable, high-quality regression suite that runs on every change to detect quality degradation early and enable confident iteration.
- Test cases and golden datasets are first-class engineering artifacts that deserve version control, provenance tracking, and ongoing maintenance to stay relevant and reliable over time.

Chapter 3: Metrics for AI Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Classical Metrics Revisited

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Text Generation Metrics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Semantics-Aware Metrics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Choosing the Right Metric for Your Task

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 4: Benchmarks and Datasets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

The Benchmark Ecosystem

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

How Benchmarks Become Obsolete

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Data Contamination: The Silent Killer of Benchmarks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Building Custom Benchmarks That Outlast Trends

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 5: Automated Evaluation at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Rule-Based and Programmatic Evaluators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Model-Assisted Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Test Case Generation at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Automation Pitfalls

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

End-to-End Evaluation Pipeline Example

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 6: LLM-as-a-Judge

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

The Rise of LLM-as-a-Judge

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Prompt Design for Reliable Judging

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Known Failure Modes

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Best Practices and Emerging Techniques

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 7: Human Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

When Humans Are Indispensable

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Designing Rigorous Human Studies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Crowdsourcing vs Expert Review

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Integrating Human Feedback Into Evals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 8: Statistical Rigor in Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Why Statistics Matter in Evals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Confidence Intervals for Model Scores

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Significance Testing: Did Your Change Actually Help?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Effect Sizes and Practical Significance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 9: Error Analysis and Iteration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Systematic Error Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

From Errors to Features

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Case Study: Iterating on a Production System Through Error Analysis

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 10: Regression Testing and Continuous Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Eval as Code

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

CI/CD Integration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Regression Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 11: Production Monitoring and Observability

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

From Offline to Online Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Production Metrics That Matter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Drift Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Building an Observability Stack for AI

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 12: Evaluating Agents and RAG Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Why Agents Break Standard Evals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Evaluating RAG Systems

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Agent Evaluation Frameworks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Tools and Patterns for Agent/RAG Evals

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 13: Safety, Robustness, and Adversarial Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Evaluating for Hallucinations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Bias and Fairness Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Adversarial Testing and Jailbreak Detection

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Robustness Under Perturbation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 14: Multimodal and Specialized Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Evaluating Vision-Language Models

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Code Generation Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Audio and Speech Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Cross-Modal Alignment

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Chapter 15: Cost, Tradeoffs, and Scaling Eval Infrastructure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

The Economics of Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Cost-Performance Tradeoffs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Building Eval Infrastructure at Scale

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Design Patterns and Anti-Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Case Study: Scaling Evaluation Infrastructure at a Mid-Sized AI Company

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Key Takeaways

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Conclusion: The Future of Evaluation Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Where We Are Now

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Emerging Frontiers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

The Maturation of Eval as a Discipline

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

Final Thoughts: Building Trust Through Evaluation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/evaluationengineeringforaisystems.