

# LLM Observability with Arize Phoenix

*Open-Source Tracing, Evaluation, and Debugging for AI Applications*

Caelan Wexford

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law.

Published by NobleTrex Press



© 2026 by NOBTREX LLC. All rights reserved.

For permissions and other inquiries, write to:

P.O. Box 3132, Framingham, MA 01701, USA

This book is for educational and informational purposes only. The author and publisher make no guarantees about the accuracy or completeness of its contents and accept no liability for any loss or damage resulting from its use. Software tools such as large language models have been applied to assist in the writing and editing of this book. All product names, logos, and trademarks are the property of their respective owners. References to third-party products or names are for identification only and do not imply endorsement.

# Contents

|          |                                                                                 |           |
|----------|---------------------------------------------------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                                                             | <b>1</b>  |
| <b>2</b> | <b>Positioning Phoenix in the Modern LLM Application Stack</b>                  | <b>5</b>  |
| 2.1      | Why LLM Applications Need a New Observability Model                             | 6         |
| 2.2      | What Arize Phoenix OSS Is and Where It Fits . . . . .                           | 15        |
| 2.3      | The Core Product Pillars of Phoenix . . . . .                                   | 24        |
| 2.4      | Version Boundaries and Current Guidance . . . . .                               | 34        |
| <b>3</b> | <b>Tracing Foundations: OpenTelemetry, OpenInference, and Phoenix Data Flow</b> | <b>43</b> |
| 3.1      | Traces, Spans, Attributes, and Context Propagation . .                          | 44        |
| 3.2      | OTLP Ingestion and the Phoenix Telemetry Pipeline . .                           | 53        |
| 3.3      | OpenInference Semantic Conventions for AI Workloads                             | 63        |
| 3.4      | Reading Phoenix Traces as Execution Narratives . . . .                          | 72        |

|          |                                                                                  |            |
|----------|----------------------------------------------------------------------------------|------------|
| <b>4</b> | <b>Instrumentation Strategies for Real-World AI Systems</b>                      | <b>83</b>  |
| 4.1      | Fast-Path Auto-Instrumentation Across Providers and Frameworks . . . . .         | 84         |
| 4.2      | Choosing What to Instrument First . . . . .                                      | 94         |
| 4.3      | Implementing Manual Spans for Custom Logic and Business-Critical Steps . . . . . | 104        |
| 4.4      | Designing High-Signal Trace Schemas . . . . .                                    | 114        |
| 4.5      | Applying Version-Safe Instrumentation Patterns . . . .                           | 124        |
| <b>5</b> | <b>Debugging with Phoenix Traces: From LLM Calls to RAG Pipelines</b>            | <b>135</b> |
| 5.1      | Trace-Driven Debugging of Model and Agent Behavior                               | 136        |
| 5.1.1    | Converting symptoms into span-level hypotheses                                   | 139        |
| 5.1.2    | Evidence to inspect inside suspicious spans . . .                                | 142        |
| 5.2      | Inspecting Retrieval Operations and Context Assembly                             | 145        |
| 5.2.1    | Reading retrieval failures as trace patterns . . .                               | 150        |
| 5.2.2    | Operational trade-offs in retrieval observability                                | 153        |
| 5.3      | Diagnosing Grounding, Relevance, and Hallucination Failures . . . . .            | 155        |
| 5.4      | Building Reusable Debugging Playbooks . . . . .                                  | 163        |
| 5.4.1    | A playbook template for bad final answers . . .                                  | 167        |
| 5.4.2    | Session-level playbooks for conversational failures                              | 169        |
| 5.4.3    | Make privacy and version context first-class evidence . . . . .                  | 170        |

|          |                                                                                                |            |
|----------|------------------------------------------------------------------------------------------------|------------|
| <b>6</b> | <b>Evaluations, Datasets, and Experiments as a Quality System</b>                              | <b>175</b> |
| 6.1      | Evaluation Primitives in Phoenix . . . . .                                                     | 176        |
| 6.2      | Choosing Between LLM Judges, Code-Based Checks, and Human Review . . . . .                     | 185        |
| 6.3      | Deciding Between Online Trace-Linked Evals and Offline Dataset Evals . . . . .                 | 194        |
| 6.4      | Curating Regression Datasets from Real Failures and Critical Scenarios . . . . .               | 203        |
| 6.5      | Running Comparative Experiments for Prompts, Models, and Application Variants . . . . .        | 212        |
| <b>7</b> | <b>Iterative Improvement and Programmatic Phoenix Workflows</b>                                | <b>223</b> |
| 7.1      | Closing the Loop from Observation to Action . . . . .                                          | 223        |
| 7.2      | Operating Prompt and Application Iteration as an Engineering Discipline . . . . .              | 234        |
| 7.3      | Automating Trace, Evaluation, and Annotation Workflows with Phoenix Clients and APIs . . . . . | 243        |
| 7.4      | Designing CI and Release Gates with Phoenix Quality Signals . . . . .                          | 252        |
| <b>8</b> | <b>Self-Hosting Phoenix for Secure, Scalable Team Use</b>                                      | <b>263</b> |
| 8.1      | Choosing a Phoenix Deployment Mode from Workstation to Kubernetes . . . . .                    | 264        |

|       |                                                                                     |     |
|-------|-------------------------------------------------------------------------------------|-----|
| 8.1.1 | Local CLI mode: fast feedback, weak operational guarantees . . . . .                | 265 |
| 8.1.2 | Single-container deployment: shared access with minimal platform overhead . . . . . | 267 |
| 8.1.3 | Compose-style environments: the practical team sandbox . . . . .                    | 269 |
| 8.1.4 | Kubernetes deployment: platform integration, not automatic virtue . . . . .         | 271 |
| 8.2   | Designing Storage for Durable, Multi-User Phoenix Operations . . . . .              | 275 |
| 8.2.1 | From convenient local storage to production persistence . . . . .                   | 276 |
| 8.2.2 | What PostgreSQL changes in practice . . . . .                                       | 277 |
| 8.2.3 | Storage patterns that map cleanly to Phoenix use . . . . .                          | 279 |
| 8.2.4 | Retention is part of storage design . . . . .                                       | 281 |
| 8.2.5 | Separate application ownership from database ownership . . . . .                    | 282 |
| 8.3   | Structuring Topology, Isolation, and Horizontal Scale . . . . .                     | 284 |
| 8.3.1 | When one Phoenix instance is still enough . . . . .                                 | 286 |
| 8.3.2 | Common topology patterns . . . . .                                                  | 287 |
| 8.3.3 | Implementation progression that keeps topology manageable . . . . .                 | 289 |
| 8.3.4 | Horizontal scale and rollout sequencing . . . . .                                   | 291 |
| 8.4   | Operating Authentication, RBAC, and Trace Data Governance . . . . .                 | 294 |

|       |                                                                                 |     |
|-------|---------------------------------------------------------------------------------|-----|
| 8.4.1 | Authentication defines the first trust boundary .                               | 296 |
| 8.4.2 | RBAC turns authentication into enforceable access                               | 297 |
| 8.4.3 | Retention, minimization, and the lifecycle of trace data . . . . .              | 299 |
| 8.4.4 | Network controls and privacy posture complete the governance envelope . . . . . | 301 |
| 8.4.5 | Operational habits that keep governance real . .                                | 302 |
| 8.5   | Migrating and Operating Phoenix Under Current Version Boundaries . . . . .      | 303 |
| 8.5.1 | The upgrade unit is application plus schema . .                                 | 305 |
| 8.5.2 | A release-aware upgrade flow that scales beyond one operator . . . . .          | 306 |
| 8.5.3 | Migration timing, index strategy, and write availability . . . . .              | 308 |
| 8.5.4 | Kubernetes and multi-replica rollout boundaries                                 | 309 |
| 8.5.5 | Automation drift is a versioned interface problem                               | 311 |
| 8.5.6 | A compact operational checklist . . . . .                                       | 312 |

# Chapter 1

## Introduction

Large language model applications have introduced a new class of software behavior that is difficult to inspect with conventional observability practices alone. In traditional systems, logs, metrics, and error reports are often sufficient to explain failures and performance regressions. In LLM-powered systems, however, the most important behaviors frequently occur inside sequences of prompt construction, model invocation, retrieval, tool use, ranking, and orchestration logic. These steps are often dynamic, probabilistic, and highly dependent on context. As a result, teams need more than infrastructure visibility. They need a way to understand what the application attempted, what information it used, how intermediate decisions were made, and why the final output met or failed expectations.

This book is about using Arize Phoenix to provide that visibility. Phoenix is an open-source platform for tracing, evaluation, and debugging of AI applications, including LLM workflows, agent systems, and retrieval-augmented generation pipelines. It helps practitioners capture execution data in a structured form, inspect



behavior at the level of traces and spans, attach quality signals to observed runs, and build repeatable workflows for improvement. Throughout the chapters that follow, Phoenix is treated not as a passive monitoring interface, but as an operational system for understanding and improving application quality.

The book begins by positioning Phoenix within the modern LLM application stack. It explains why AI systems require an observability model that extends beyond standard application telemetry, and it clarifies how Phoenix relates to model providers, agent frameworks, vector stores, and serving infrastructure. From there, the discussion moves into the technical foundations that make Phoenix effective: OpenTelemetry as the transport and tracing substrate, OpenInference as the semantic layer for AI workloads, and the data flow by which instrumented executions become usable debugging artifacts.

A major emphasis of this book is instrumentation. Effective observability depends on collecting the right signals with the right level of detail. For that reason, the text covers both rapid auto-instrumentation paths and more deliberate manual instrumentation patterns for custom logic, business-critical workflows, and production-grade systems. The goal is not only to produce traces, but to produce traces that are meaningful, searchable, and aligned with the way teams actually investigate failures.

The middle chapters focus on practical debugging and evaluation. Readers will learn how to interpret traces, analyze model and agent behavior, inspect retrieval operations, and isolate failures in grounding, relevance, latency, and execution flow. Phoenix’s evaluation capabilities are then introduced as the complementary layer that turns observation into measurement. The book explains how to use model-based judges, code-based checks, human annotation, datasets, and experiments to create a disciplined quality system for LLM applications.

---

The later chapters address iterative improvement and operational maturity. They show how traces, evaluations, and experiments can be connected into continuous engineering workflows, including client and API-based automation, regression testing, and release validation. The final chapter turns to self-hosting and production deployment, with attention to storage choices, scaling patterns, access control, and governance for sensitive trace data.

This book assumes that readers want practical, current guidance. It therefore follows modern Phoenix workflows and version-aware conventions, emphasizing supported instrumentation and deployment patterns rather than outdated approaches. The aim is to provide a clear foundation for teams building reliable AI systems and to show how observability, evaluation, and debugging can be integrated into everyday engineering practice.



## Chapter 2

# Positioning Phoenix in the Modern LLM Application Stack

*Most failures in AI applications do not look like ordinary software defects: the system returns something plausible, slow, expensive, off-policy, or subtly ungrounded, and the root cause may sit across prompts, retrieval, orchestration, and model behavior at once. This chapter gives the reader the operating vocabulary and product boundaries needed to understand Phoenix as the connective tissue between runtime evidence and quality improvement, while also clarifying the current-version assumptions that keep the rest of the book technically aligned.*

## 2.1. Why LLM Applications Need a New Observability Model

A conventional service can be healthy in every operational dashboard and still deliver a bad AI experience. Your request rate can be normal, the HTTP success rate can sit above 99%, CPU and memory can remain flat, and yet users still receive irrelevant answers, unsupported claims, malformed tool actions, or responses that are technically complete but operationally unusable. That mismatch is the defining observability problem of LLM systems: many important failures occur inside requests that look successful to ordinary monitoring.

A simple request log makes the gap visible. Consider the kind of record many teams start with:

```
{
  "request_id": "req_9f3a",
  "path": "/chat",
  "status": 200,
  "latency_ms": 1842,
  "model": "gpt-4.1-mini",
  "tokens_in": 4210,
  "tokens_out": 312
}
```

That log is useful. You can alert on latency, count traffic, estimate cost, and correlate spikes with deployments. You still cannot answer the questions that matter when a user says, “the answer was wrong.” You do not know which prompt template rendered, whether the retrieval step returned weak evidence, whether context was truncated to fit the token budget, whether a tool call failed and the model improvised around it, or whether the response violated a policy despite returning HTTP 200. The transport completed; the application may still have failed semantically.

**Availability is not quality**

Classical observability is built around infrastructure and software reliability. You watch logs for exceptions, metrics for saturation and throughput, and traces for distributed latency across services. That model assumes failure is usually explicit: a timeout, a crash, a dependency outage, a malformed request, a slow database query. When the system fails, the request often fails too.

LLM applications break that assumption. They regularly produce low-quality results while remaining operationally healthy. A retrieval-augmented assistant can answer confidently from irrelevant passages. A support bot can ignore the user's account state because session context was not attached. An agent can choose the wrong tool and still produce a fluent answer. A summarizer can omit a critical clause because the prompt budget forced truncation. None of these conditions necessarily raise an exception. From a classical monitoring perspective, the service succeeded.

That distinction between *system availability* and *application quality* matters because it changes what you instrument and what you investigate. If you only measure whether the request completed, you miss whether it completed correctly, groundedly, efficiently, and consistently. For AI systems, user-visible correctness is often a property of the entire execution path, not the final HTTP response.

### **An AI request is an execution narrative**

You get more leverage once you stop thinking of an AI request as “one model call” and start treating it as a structured execution narrative. A single user message may trigger prompt assembly, retrieval, reranking, tool selection, tool invocation, response synthesis, guardrail checks, and post-processing. In multi-turn systems, the request may also depend on prior turns, user metadata, and session memory. Each stage can change the answer without causing a crash.

That framing introduces a core term used throughout the rest of the dis-

cussion: **execution path**. The execution path is the ordered sequence of meaningful steps that transformed an input into an output. In a traditional microservice trace, the path might mostly explain latency and dependency behavior. In an LLM application, the path must also explain semantic behavior: what evidence the model saw, how it was instructed, which tools it used, and what intermediate outputs shaped the result.

The need for **intermediate step visibility** follows directly from that model. If you only store the final answer, you have almost no fault localization. A hallucinated answer could result from missing retrieval, weak retrieval, prompt truncation, poor tool output, a model reasoning error, or aggressive post-processing. These all look similar from the outside and require different fixes. One may need indexing changes, another prompt edits, another schema validation, another model replacement. Observability has to preserve enough evidence to distinguish them.

### Why logs, metrics, and exceptions stop short

Logs remain useful, but ordinary application logs are usually too coarse and too unstructured for AI debugging. Teams often log the user message, the final response, maybe a model name, and perhaps a few timing values. That record tells you *that* something happened, not *why* it happened. Free-form logs also make comparison difficult. If one engineer logs retrieved documents as a JSON blob, another logs only document IDs, and a third logs nothing unless debug mode is on, you cannot reliably reconstruct behavior across requests.

Metrics have a different limitation. They aggregate by design. That makes them excellent for detecting trends and poor for reconstructing semantic failures. Average latency can reveal that retrieval is slowing down. Token usage metrics can reveal an expensive prompt change. Error-rate metrics can reveal tool outages. None of those metrics can

explain whether a specific bad answer came from irrelevant context, a dropped system instruction, or an incorrect tool argument. Metrics tell you where to look; they rarely contain the evidence needed to explain the defect.

Exception tracking narrows the blind spot only slightly. It captures explicit software failures, but many of the expensive and user-visible LLM failures are not exceptional in the programming sense. A poorly grounded answer is not an uncaught exception. A wrong citation attached to a plausible answer is not a stack trace. An agent loop that terminates after choosing an inferior plan may satisfy every retry and timeout policy. Exception tracking still matters, especially around tools and orchestration layers, but it does not represent semantic quality.

Distributed traces get closer because they preserve request structure. A span remains the right basic unit of work: one retrieval call, one model invocation, one tool execution, one reranker step. The problem is not that tracing is irrelevant; it is that generic tracing semantics are incomplete for AI workloads. A classic span can tell you duration, parent-child relationships, and a few generic attributes. That is not enough when the diagnostic question is, “what prompt was rendered, what documents were retrieved, what tool arguments were sent, how many tokens were consumed, and which session state influenced the answer?” AI observability needs spans enriched with **semantic metadata** specific to model-driven systems.

### **Semantic metadata and traceable context assembly**

The phrase **semantic metadata** refers to structured attributes that capture the meaning of AI execution rather than just its timing. Examples include prompt templates, message arrays, retrieved passages, tool names and arguments, model identifiers, token counts, evaluator labels, and user or session context. These attributes let you inspect



what the model actually saw and did.

A closely related term is **traceable context assembly**. In many LLM applications, the final prompt is assembled dynamically from several sources: the system instruction, the current user message, conversation history, retrieved documents, tool outputs, policy snippets, and business rules. If you cannot trace that assembly process, you cannot debug a large class of failures. You may know that the answer was wrong, but not whether the model lacked necessary evidence or received the wrong instructions.

This is especially important in retrieval-augmented generation. Teams often label any unsupported answer a hallucination, but that label is diagnostically weak. The model may have hallucinated despite strong evidence. It may also have answered from irrelevant or empty retrieval results. The fix is very different in each case. When retrieval returns poor passages, the bug may live in chunking, indexing, query construction, metadata filters, or reranking. When strong passages are present but ignored, the bug may live in prompt wording, ranking order, context budget, or model capability. Without traceable context assembly, both failure modes collapse into the same vague complaint.

### **The major failure surfaces in LLM systems**

Several failure surfaces recur across modern AI applications, and each one can degrade user outcomes without producing conventional software errors.

- **Prompt construction.** Prompt templates, hidden instructions, and dynamic variables strongly influence behavior. Small changes in wording, ordering, or delimiters can shift answer quality, refusal behavior, citation style, and tool use. Prompt bugs are often silent regressions: the code deploys, the request returns 200, but answer quality drops.

- **Model invocation.** Model choice, decoding parameters, context limits, and provider-side behavior affect both quality and cost. A response can be coherent but weaker than expected because a smaller model was selected, a context window was exceeded, or inputs were truncated before inference. Ordinary infrastructure telemetry rarely records enough detail to expose that chain.
- **Retrieval quality.** In RAG systems, retrieval often determines whether the answer can be grounded at all. Empty results, near-duplicate passages, stale documents, overly broad chunks, weak metadata filtering, or query drift can all degrade output while leaving the service technically healthy.
- **Tool selection and invocation.** Agentic systems add another layer of fragility. The model may choose the wrong tool, construct invalid arguments, ignore a necessary tool, or overuse tools when direct reasoning would suffice. Some of these produce explicit errors; many do not. The agent can recover partially, fabricate around failures, or exit with a plausible but wrong answer.
- **Planning and control flow.** Multi-step agents create stateful execution paths that are difficult to reconstruct from coarse logs. A bad final answer may originate several steps earlier in a wrong branching decision or a loop that consumed budget on low-value actions.
- **Post-processing and guardrails.** Final outputs are often transformed after generation through parsers, classifiers, filters, redactors, or formatters. These components can remove important context, introduce formatting defects, or overfilter safe content. If you only observe the final response, you cannot tell whether the model or the downstream post-processor caused the issue.

- **Session context.** Multi-turn systems depend on continuity. Missing session identifiers, stale conversation memory, or incorrect user scoping can make a later turn fail even if each individual request appears well formed. A chat application without session-aware observability fragments the very context you need to debug conversational behavior.

### Quality regression is an operational event

Traditional operations teams are trained to treat regressions as changes in latency, availability, or resource consumption. LLM systems add another class: **quality regression**. A quality regression occurs when outputs become less useful, less grounded, less safe, less compliant, or less consistent after a prompt change, retrieval change, model change, data refresh, or orchestration change. The service may remain perfectly available. From the user's perspective, it still got worse.

This is why request-level success rate is a dangerous primary health signal for AI features. A stable 200-rate can hide substantial quality drift. The same applies to storing only final outputs. Final outputs are necessary but insufficient. They tell you what the user saw, not what caused it. If your debugging workflow begins after the intermediate evidence has already been discarded, you are forced into speculation.

### Evaluation-linked debugging

A useful observability model for AI systems has to support not only inspection but also judgment. That requirement motivates another key term: **evaluation-linked debugging**. The idea is straightforward. You do not merely inspect traces to understand execution; you also connect them to explicit judgments about quality, such as relevance, groundedness, correctness, policy compliance, or document usefulness. Once debugging is linked to evaluation, you can move from

anecdotal complaints to repeatable diagnosis.

That linkage matters because semantic failures are often ambiguous. If users report that answers feel “off,” you need a way to sort requests into failure categories and compare them over time. Evaluation creates a bridge between individual bad traces and recurring patterns. It helps you answer questions such as these: Are failures concentrated on one retriever configuration? Did a prompt edit reduce citation quality? Did a new model cut latency but increase unsupported claims? Observability without evaluation leaves you with rich evidence but weak prioritization. Evaluation without observability leaves you with scores but poor root-cause analysis.

### **Capture strategy is a design problem, not a logging toggle**

Once you accept that AI observability requires semantic execution detail, the next question is how much to capture. There is no universal answer. Recording every prompt, document chunk, tool argument, and response artifact maximizes diagnostic value, but it increases storage, processing cost, and privacy risk. Recording only aggregate timings and final outputs is cheaper and safer, but it strips away fault-localization power.

You need a deliberate capture strategy. Start by asking which classes of failures you must be able to explain. If grounding failures are expensive, preserve retrieved document identifiers and enough content to assess relevance. If tool misuse is a known risk, capture tool names, invocation arguments, and outputs. If prompt changes are frequent, preserve prompt templates and rendered inputs. If conversational continuity matters, preserve session identifiers and turn structure. This is not “more logging” in a generic sense. It is selective preservation of high-value artifacts from the execution path.

Privacy and governance shape these decisions immediately. Some teams can store prompt and response text in full; others need redac-

tion, hashing, field-level suppression, or short retention windows. Cost also matters. Rich traces can become expensive if you capture large document payloads for every request. The right strategy balances diagnostic value against sensitivity and volume, but the governing principle stays the same: if you do not capture the artifacts that determine behavior, you cannot reliably debug behavior.

### **Anti-patterns that slow teams down**

Three anti-patterns appear repeatedly.

- **Overreliance on request success rates.** This causes teams to declare AI features healthy while users quietly experience quality degradation. Success rates measure transport completion, not semantic adequacy.
- **Storing only final outputs.** This turns every investigation into guesswork. You can observe the symptom but not the mechanism. In multi-step systems, the mechanism often matters more than the symptom.
- **Calling every bad answer a hallucination.** That label compresses multiple root causes into one vague category and leads to shallow fixes. Unsupported answers can come from absent evidence, weak retrieval, poor prompt assembly, context truncation, model weakness, or synthesis errors. Effective observability keeps these possibilities distinct long enough to localize the defect.

### **The operating model you need**

The practical shift is to treat each AI request as a traceable, inspectable, semantically rich execution path whose quality must be judged separately from its transport success. Logs, metrics, and exception tracking

still belong in the stack; they remain necessary for infrastructure, service reliability, and dependency behavior. They are just not sufficient for stochastic, multi-step, model-driven systems. You need visibility into prompts, retrieved context, tool usage, intermediate steps, session state, and outcome quality so that a bad answer is not merely observed but explainable, classifiable, and fixable.

### 2.2. What Arize Phoenix OSS Is and Where It Fits

Once you accept that AI failures often hide inside successful requests, the next practical question is where the observability system belongs. If a user reports that an answer was irrelevant or unsupported, you need a place where execution evidence from prompts, retrieval, tools, and model calls can be collected, inspected, and compared. Phoenix fills that role. It is officially described as an open-source AI observability and evaluation tool. That wording matters because it places Phoenix in the evidence and improvement layer of an AI stack, not in the execution layer that generates the behavior in the first place.

A minimal configuration makes the boundary concrete. An instrumented application emits telemetry to a Phoenix collector over OTLP/HTTP. The application remains your runtime; Phoenix receives and presents the resulting traces.

```
export PHOENIX_COLLECTOR_ENDPOINT=http://localhost:6006
export PHOENIX_API_KEY=dev-key
```

In Python, current tracing setup uses the Phoenix OpenTelemetry package directly from the application process:

```
from phoenix.otel import register

tracer_provider = register()
```

Those two fragments already define the core relationship. Your appli-

cation, framework, retriever, and tools perform the work. Instrumentation inside that application emits spans. Phoenix acts as the collector and user interface for those spans. If the application is not instrumented, Phoenix has nothing meaningful to inspect. If the application is instrumented only with generic spans and weak semantic metadata, Phoenix can receive the data, but your debugging power will still be limited.

### **Phoenix is a runtime evidence layer**

A precise mental model is to treat Phoenix as a *runtime evidence layer*. It sits after execution and before improvement workflows. Upstream systems generate behavior: model providers produce completions, retrievers return documents, tools execute side effects, orchestration frameworks choose steps, and your application code assembles prompts and state. Downstream workflows consume evidence: debugging, annotation, evaluation, regression analysis, dataset curation, and experiments.

That placement is more specific than saying Phoenix is “an observability tool.” Many systems claim observability, but they observe different things and operate at different layers. Traditional APM products focus on service health, infrastructure dependencies, and exception paths. Provider dashboards focus on quota, billing, or provider-specific request diagnostics. Vector database tooling focuses on index health and search behavior. Phoenix focuses on AI application behavior as reconstructed from traces and AI-specific semantic metadata.

This distinction matters most when teams try to assign ownership. If your vector database returns high latency, Phoenix is not the authoritative source for low-level index tuning. If your GPU host is throttling, Phoenix is not the primary system for node-level saturation analysis. If your model provider rejects requests due to account limits, the provider console remains authoritative for that event class. Phoenix be-

comes authoritative when you need to answer questions such as these: What did the application actually send to the model? Which documents were retrieved? Which tools ran? Where in the execution path did latency accumulate? Which classes of traces correlate with poor outcomes?

### What Phoenix is not

The fastest way to place Phoenix correctly is to rule out several common category errors.

- **Phoenix is not a model serving layer.** It does not host the model that generates your outputs. Your model provider or inference stack still owns inference execution, model lifecycle, quotas, and provider-specific runtime behavior.
- **Phoenix is not a vector database.** It does not index your corpus, compute nearest neighbors, or replace the retrieval engine that serves passages to the application. Retrieval quality may be inspected through traces, but the underlying search stack remains separate.
- **Phoenix is not a workflow engine or agent runtime.** It does not orchestrate your tools, manage your state machine, or choose the next action in an agent loop. LangChain, LlamaIndex, custom orchestration code, or another runtime layer owns that control flow.
- **Phoenix is not a policy engine.** It does not replace the classifiers, rule engines, or enforcement points you use to govern content and actions. It can help you observe where policy failures occur and evaluate their frequency, but it is not the enforcement mechanism itself.
- **Phoenix is not a general-purpose APM replacement.** You still need ordinary monitoring for CPU, memory, host availabil-



ity, network failures, queue depth, database saturation, and infrastructure error budgets. AI observability adds a semantic execution layer; it does not eliminate the need for operational telemetry.

These boundaries are not marketing distinctions. They determine the shape of the data Phoenix receives and the kinds of questions it can answer. Teams run into trouble when they expect Phoenix to compensate for missing runtime design, weak retrieval engineering, or absent infrastructure monitoring. Phoenix can expose evidence of those issues, but it does not substitute for the systems that actually execute or govern the workload.

### **How Phoenix connects to the modern LLM stack**

The stack placement becomes clearer if you walk through a typical AI request. A user sends a message to your application. Application code normalizes the input, resolves user and session state, and may call an orchestration framework. That framework may retrieve documents from a vector store, call one or more models, invoke tools, and synthesize the final answer. During this execution, instrumentation emits spans enriched with AI-specific attributes. Those spans are transported using OpenTelemetry protocols and AI-aware semantics. Phoenix receives them over OTLP/HTTP, stores them as traces, and exposes them for investigation and later quality workflows.

That architecture has two important consequences.

First, Phoenix is *downstream of execution*. It does not need to be the framework you use to build the application. Because it is built on OpenTelemetry and uses OpenInference for AI-specific semantics, it operates as a vendor-, language-, and framework-agnostic tracing layer. That design is why Phoenix can fit into heterogeneous environments instead of requiring a single orchestration stack.

Second, Phoenix is *upstream of analysis and iteration*. Raw runtime evidence becomes valuable only when a team can inspect it, compare it, score it, and turn recurring failures into repeatable fixes. Phoenix occupies that handoff point between behavior and learning. That is why its official positioning includes both observability and evaluation. The two belong together operationally. Traces tell you what happened. Evaluation helps you decide whether what happened was acceptable.

### **OpenTelemetry and OpenInference define the tracing contract**

Phoenix's architectural foundation is not a proprietary telemetry protocol. It is built on OpenTelemetry and uses OpenInference for AI-specific semantics. That matters for interoperability more than branding. OpenTelemetry gives you the transport and trace model: traces, spans, parent-child relationships, timing, and attribute propagation. OpenInference adds the semantic layer that generic telemetry lacks for AI systems: model calls, prompts, retrieved documents, tool usage, message content, token economics, and related execution evidence.

Without that semantic layer, Phoenix would receive structurally valid traces that still fail to explain model behavior. A generic span named `llm_call` with a duration value is better than nothing, but it does not tell you what prompt was rendered, which model was called, or whether retrieved context was attached. OpenInference-style semantics make traces useful for AI debugging rather than merely compliant with distributed tracing infrastructure.

This is also why Phoenix should be thought of as *collector plus interpretation layer*. It does not merely store arbitrary spans. It benefits from spans that describe AI work in a structured, inspectable form. When those attributes are present, Phoenix can present execution paths in a way that matches how practitioners actually debug AI systems. When they are absent, you fall back toward generic trace inspection.

## **Manual and automatic instrumentation remain upstream concerns**

Because Phoenix is a collector and interface, instrumentation happens in the application. You can instrument manually or use automatic integrations where available, but either way the application is responsible for emitting spans. This boundary is easy to overlook when teams encounter Phoenix for the first time. They may assume that deploying Phoenix alone will reveal model prompts, retrieval outputs, and agent steps. It will not. Phoenix receives what your application sends.

That has practical implications for adoption. If your application already uses libraries or frameworks with OpenInference-compatible instrumentors, Phoenix can often become useful quickly because rich spans already exist or can be enabled with modest changes. If your application is custom and the high-value behaviors live in bespoke orchestration code, you need a manual instrumentation strategy. In that case, Phoenix still fits cleanly in the stack, but the effort sits upstream in trace design.

This division of labor is healthy. It keeps Phoenix decoupled from your business logic and runtime choices. It also means the quality of your Phoenix experience depends directly on the quality of your instrumentation plan. A well-placed collector cannot recover information that the application never emitted.

## **What remains authoritative outside Phoenix**

Advanced teams benefit from being explicit about which system remains authoritative for which decisions.

Use your infrastructure monitoring stack for host health, service saturation, process crashes, queue backlogs, network failures, and generic SLO management. That is the right toolchain for capacity planning and operational incident response.

Use provider dashboards and billing systems for account limits, provider-side status, invoice reconciliation, and provider-specific error reporting. Those systems remain closest to the commercial and operational behavior of the model endpoint.

Use vector database and search infrastructure tooling for indexing pipelines, embedding refreshes, ANN tuning, filter performance, and low-level retrieval mechanics. Retrieval quality as experienced by the application can be observed in Phoenix, but search engine internals still live elsewhere.

Use your application logs and security systems for access control enforcement, audit obligations, and non-AI business events that have regulatory or operational importance beyond the AI request itself.

Use Phoenix when the question is about the AI application's end-to-end behavior: how a request moved through retrieval, prompting, tool usage, model inference, and response generation, and how that behavior relates to quality. That boundary keeps teams from overloading one platform with responsibilities it was not built to own.

### **Operating modes change the ergonomics, not the role**

Phoenix can serve the same conceptual role in very different environments. The most immediate mode is local developer use. In a local setup, you instrument an application, send traces to a locally reachable collector, and inspect them while iterating on prompts, retrieval logic, or tool wiring. The benefit is short feedback loops. A single engineer can reproduce a failure, inspect the trace, edit code or prompts, and retry immediately. Local mode is ideal when your main constraint is development speed rather than shared governance.

A self-hosted team deployment changes the operating context. Now Phoenix becomes shared infrastructure for multiple engineers, services, or environments. Access control, retention, storage sizing, net-

work paths, and operational ownership become real concerns. The reason Phoenix exists does not change: it still collects AI telemetry and supports investigation and evaluation. What changes is the burden of operating it reliably and integrating it with the rest of your platform standards. This mode tends to fit organizations with privacy requirements, internal network constraints, or a need to keep runtime evidence inside controlled infrastructure.

Managed or cloud-adjacent usage changes the boundary again. Operational burden can decrease because the surrounding platform handles more of the service lifecycle. At the same time, service boundaries, data movement expectations, and governance responsibilities need to be evaluated more carefully. The same questions still apply: what telemetry leaves the application, where it is stored, who can access it, and which other systems remain authoritative. Deployment mode affects trust boundaries, scaling assumptions, and procurement constraints, but it does not change Phoenix’s conceptual place between execution and improvement.

### **Decision signals for adopting Phoenix**

You usually adopt Phoenix first as a debugging aid and later, if the AI feature matters operationally, as shared team infrastructure. Several signals indicate that local use is enough for now. The application is still exploratory, one or two engineers own it, traces are mainly needed during prompt or retrieval development, and governance requirements are light. In that environment, the highest value comes from rapid fault localization.

Different signals suggest that Phoenix should become a shared platform component. Multiple teams need visibility into traces. Production failures need reproducible investigation paths. Quality regressions need to be compared across releases. Trace-linked evaluation or dataset curation starts to matter. Privacy rules require consistent

storage and access controls. At that point, Phoenix stops being just a developer convenience and becomes part of the operating model for AI quality.

A third threshold appears when teams already have mature observability and governance platforms. Then the question is not whether Phoenix replaces them, but how it complements them. You may continue to route infrastructure telemetry to existing APM systems, preserve security events in established audit pipelines, and use Phoenix specifically for AI execution evidence and evaluation workflows. That split is often healthier than trying to force every telemetry type into one system.

### **Common misplacements and the cost of getting them wrong**

One common mistake is treating Phoenix as a substitute for instrumentation. Teams deploy the collector and UI, then expect insight to appear automatically. Because Phoenix sits downstream of emitted traces, this produces a disappointing experience: few spans, weak semantic detail, and little diagnostic value.

Another mistake is treating Phoenix as the only observability layer you need. That often leads to neglected host monitoring, poor service-level alerting, and confusion during infrastructure incidents. AI observability and classical observability solve related but distinct problems.

A subtler mistake is treating Phoenix as a runtime component that should own orchestration decisions. That confuses observation with execution. Once the same system both runs and judges the workflow, debugging boundaries become muddy. Phoenix is more valuable when it remains a stable evidence layer across changing runtimes, frameworks, and provider choices.

The healthiest placement is simple: instrument the application where meaningful AI work occurs, emit traces over OTLP/HTTP using Open-

Telemetry with OpenInference semantics, collect them in Phoenix, and use Phoenix as the place where runtime evidence becomes inspectable, comparable, and actionable. That separation keeps the stack modular. Your models, tools, retrievers, and orchestration code can evolve independently, while Phoenix remains the system that helps you understand what they actually did.

### 2.3. The Core Product Pillars of Phoenix

Once Phoenix is placed correctly in the stack, the more useful question is how to think with it day to day. A product name is not yet an operating model. You need a compact mental structure that explains why Phoenix includes tracing, evaluation, datasets, experiments, and prompt or application iteration in one workflow instead of presenting them as unrelated features. The organizing idea is that Phoenix closes a feedback loop: runtime evidence becomes judgment, judgment becomes reusable test material, reusable test material supports controlled comparison, and controlled comparison drives changes back into the application.

A concrete evaluation example makes that loop easier to anchor. Suppose your support assistant uses retrieval, and users report that answers often cite the wrong policy passages. Before you change prompts or swap models, you want a repeatable relevance signal over retrieved documents. Phoenix Evals exposes built-in evaluators and DataFrame-based APIs for exactly that kind of quality measurement.

```
import pandas as pd

from phoenix.evals import LLM, async_evaluate_dataframe
from phoenix.evals.metrics import DocumentRelevanceEvaluator

df = pd.DataFrame(
    [
        {
            "input": "Can I return an opened laptop in 30 days?",
```

```
        "reference": (
            "Laptops may be returned within 30 days. "
            "Opened items incur a restocking fee."
        ),
    },
    {
        "input": "Do you ship replacement parts internationally?",
        "reference": (
            "Replacement parts ship only to domestic addresses."
        ),
    },
]
)

llm = LLM(provider="openai", model="gpt-4.1-mini")
evaluator = DocumentRelevanceEvaluator(llm=llm)

# Run from an async context in real application code.
# results = await async_evaluate_dataframe(
#     dataframe=df,
#     evaluators=[evaluator],
# )
```

Even without running it here, the pattern is already visible. You start from concrete examples, apply an evaluator with an explicit quality criterion, and produce artifacts that can be compared over time. That pattern only works well if you can connect evaluation back to execution evidence. Otherwise you know that quality changed, but not why. Phoenix's core pillars are designed to keep those two halves together.

### Tracing is the runtime evidence pillar

Tracing is the first pillar because everything else depends on having credible runtime evidence. In Phoenix, traces represent the execution path of an AI request as a set of related spans. Those spans can describe LLM calls, retrieval operations, tool invocations, agent steps, evaluator runs, prompt rendering, and application-defined work. The central value is not merely chronology. It is *structured causality*: which step happened, in what order, with what inputs, for how long, and with which semantic attributes.



For ordinary software, a trace often answers questions about latency and dependency behavior. For AI applications, the trace also has to answer semantic questions. Which prompt template was rendered? Which documents were retrieved? Did a tool call happen before the model responded? Did the agent take three steps or ten? Was the final answer assembled from strong evidence or from weak context and guesswork? Phoenix uses tracing as the evidence backbone because these questions require a request-scoped execution narrative, not isolated log lines.

This is why tracing should be read as more than observability plumbing. It is the substrate that preserves debuggable context. If a model output is wrong, the trace gives you a path for fault localization. If retrieval failed, you can inspect the retrieval span rather than guessing at the model. If a tool returned malformed output, you can isolate the defect without conflating it with prompt quality. If a prompt edit caused a regression, traces help determine whether the regression arose from instruction changes, context packing, or some interaction with downstream synthesis.

There is a practical asymmetry here: teams can do useful tracing without immediate formal evaluation, but they cannot do robust evaluation-linked debugging without traces. Scores alone do not preserve context. A relevance or correctness label tells you that something went wrong, not what the application actually did to produce the failure.

### **Sessions extend tracing to conversational systems**

Multi-turn systems force an additional requirement: the unit of debugging is often larger than one request. A support bot may answer each turn plausibly in isolation and still fail across the conversation because it loses a user identifier, drops an order number, or forgets a policy constraint stated earlier. That is where sessions matter. Sessions extend tracing to conversation observability by grouping related traces with a

shared session ID.

This is not just a convenience for chat interfaces. It changes what you can diagnose. Suppose a bot repeatedly asks for the same account number despite the user having already provided it. A single trace may show a reasonable prompt and answer. The problem becomes obvious only when you inspect the group of traces sharing the same session: somewhere between turns, the application stopped carrying forward a critical piece of state. Sessions expose that failure mode because they preserve continuity across otherwise independent requests.

Phoenix provides helpers for session context in tracing APIs, but the operational point matters more than the syntax here. If your product involves conversations, agents, or multi-step user tasks, request-scoped traces are not enough. You need grouped traces that let you reason about state continuity, user intent over time, and repeated error patterns within the same interaction.

### **Evaluations are the quality measurement pillar**

Tracing preserves what happened. Evaluation asks whether what happened was good enough. That distinction is essential. Many AI failures are visible only after you impose an explicit standard such as groundedness, relevance, classification accuracy, policy compliance, answer helpfulness, or tool correctness. Phoenix Evals provides the quality measurement pillar through built-in evaluators, a unified LLM wrapper, and synchronous or asynchronous DataFrame evaluation APIs.

The DataFrame interface reflects a practical engineering choice. Evaluation becomes more tractable when you can organize examples as rows with stable columns for inputs, outputs, references, retrieved content, metadata, or labels. Once examples are tabular, you can run evaluators systematically, aggregate results, and compare outcomes across prompt or model changes. You move from anecdotal debugging to repeatable analysis.

The important architectural point is that evaluation in Phoenix is not isolated benchmarking. It is designed to sit near runtime evidence. That proximity matters because quality scores are most useful when you can explain them. If a document relevance evaluator flags weak retrieval, you want to inspect the associated retrieval behavior. If a classification evaluator catches policy failures, you want to inspect the prompt and response context around those examples. Evaluation without observability produces a leaderboard. Evaluation linked to traces produces an engineering workflow.

This also clarifies why Phoenix includes evaluator spans in the tracing vocabulary. Evaluations are themselves part of the system's learning loop. They are not external commentary glued onto the side of production behavior. They become inspectable steps in the broader process of debugging and improvement.

### **Datasets are the memory of recurring problems**

Once you can inspect traces and score behavior, the next challenge is persistence. Teams often discover the same failure class repeatedly and still fail to preserve it as a reusable test asset. A support bot mishandles return-policy edge cases, an internal assistant cites stale documents, or an agent keeps choosing a search tool when a calculator is required. If those cases remain trapped in screenshots, chat transcripts, or ad hoc notebooks, you do not yet have durable engineering memory.

Datasets provide that memory. In Phoenix, datasets store examples with inputs, optional outputs, and optional metadata. They can be populated from production, staging, evaluations, or manual review. That makes datasets the bridge between observation and disciplined improvement. A production trace that reveals a subtle grounding failure can become a dataset example. A curated edge case from QA can sit beside it. A manually labeled sample from an evaluator disagreement can join the same corpus. Over time, the dataset becomes a more

faithful representation of your application’s real failure surface than a synthetic benchmark assembled in isolation.

The deeper reason datasets matter is representativeness. Teams frequently try to improve prompts or model choices against whatever examples are easiest to collect, not whatever examples are most operationally important. Phoenix’s workflow encourages the opposite direction: start from observed behavior, preserve representative failures, then reuse them consistently. That is why datasets belong in the same conceptual loop as traces and evaluations. Without them, you keep rediscovering the same defects and relearning the same lessons.

### **Experiments turn changes into controlled comparisons**

Once you have a dataset, you can compare alternatives on the same examples instead of relying on impressionistic testing. Experiments are the pillar that makes iteration rigorous. You hold the evaluation corpus relatively stable, vary the thing you want to test—prompt wording, retrieval configuration, model choice, tool policy, orchestration logic—and examine how outcomes change.

This is where many teams reverse the order and get poor results. They begin with prompt tinkering, run a few spot checks, see mixed outcomes, and then debate subjectively which variant “feels better.” Without traces, they lack root-cause context. Without evaluations, they lack explicit quality criteria. Without datasets, they lack representative cases. The experiment becomes theater rather than evidence.

Phoenix’s experiments capability matters because it places controlled comparison downstream of curated evidence. That order is technically sound. You do not ask which prompt is best in the abstract. You ask which prompt performs better on the failure modes your system actually encounters. You do not ask whether a new model is generally stronger. You ask whether it improves the dimensions you care about without unacceptable latency or cost trade-offs on your own dataset.

Datasets and experiments were introduced as a distinct capability area in Phoenix 4.6+, which is a useful reminder that this part of the product is not an optional accessory. It formalizes the transition from debugging individual failures to engineering against stable corpora.

### **Iteration is the point of the loop**

Prompt iteration and application iteration are where the workflow becomes operational rather than descriptive. You inspect a bad trace, identify a likely cause, preserve the example in a dataset, evaluate a candidate fix, compare alternatives in an experiment, and then ship the change that improves the target behavior. Iteration can happen at several layers: prompt text, retrieval strategy, chunking, reranking, model selection, tool schema, guardrail logic, or overall orchestration design.

Calling this a pillar is useful because it prevents a common misunderstanding. Some teams treat iteration as informal prompt editing done outside the observability system. That misses the point. Iteration is the engineering action that closes the loop. Without it, traces become passive diagnostics and evaluations become passive scorekeeping. With it, Phoenix becomes a system for shortening the path from observed runtime defect to validated improvement.

The scope of iteration also matters. Sometimes the fix really is prompt-level. A better instruction can reduce citation errors or make tool usage more deterministic. Sometimes prompt editing is the wrong tool. If retrieval is weak, rewriting instructions will not fix absent evidence. If a tool's output schema is unstable, changing model phrasing may only mask the defect. The right use of Phoenix is to keep those options distinguishable long enough to choose the correct layer of intervention.

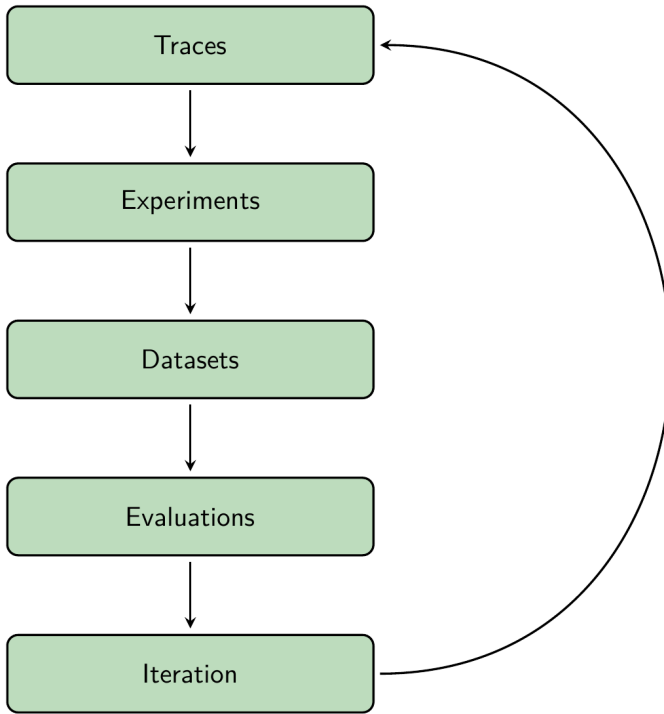
### **From isolated features to a closed-loop system**

The pillars become more coherent when you view them as a depen-

gency chain rather than a menu of capabilities.

- Tracing supplies runtime evidence.
- Evaluations attach quality judgments to that evidence or to curated examples.
- Datasets preserve representative cases so the same problems can be re-tested.
- Experiments compare candidate changes on those cases.
- Iteration applies the winning change back to the application.

That chain is not arbitrary. Each step stabilizes the one after it. Traces ground debugging in facts rather than anecdotes. Evaluations prevent inspection from remaining purely qualitative. Datasets prevent discoveries from evaporating after one incident. Experiments prevent change decisions from becoming subjective. Iteration keeps the workflow tied to shipped system behavior rather than detached analysis.



### Trade-offs inside the loop

Several real trade-offs appear once you start operating this loop seriously.

One trade-off is *live trace-linked evaluation versus offline dataset evaluation*. Live evaluation tied closely to production traces can reveal regressions quickly and keep scoring aligned with real traffic. It also raises cost, privacy, and operational-noise concerns. Offline dataset evaluation is cheaper to control, easier to reproduce, and better for disciplined comparisons, but it can drift away from production reality if the dataset stops reflecting actual usage.

Another trade-off is *qualitative debugging versus numeric aggregation*. Trace inspection is rich and explanatory. It tells you why one request failed. Aggregated evaluator scores tell you whether a change improved the population you care about. You need both. Overweight qualitative analysis and you risk overfitting to memorable examples. Overweight aggregate scores and you may miss subtle but operationally severe failure modes hidden inside the average.

A third trade-off is *prompt-level fixes versus broader redesign*. Prompt changes are cheap and often effective, so teams reach for them first. That is sensible when the failure truly stems from instruction quality or response formatting. It is wasteful when the defect sits in retrieval, tool contracts, session handling, or orchestration control flow. The loop helps you choose the correct intervention layer because it ties symptoms back to execution evidence and then tests fixes against preserved examples.

### **A compact debugging loop**

A good way to internalize the pillars is to imagine a support assistant that repeatedly loses an order ID in the second or third turn of a conversation. Traces reveal that the first turn extracted the ID correctly. Session-grouped traces show that later turns stopped carrying it forward. You preserve representative conversations as dataset examples. You run evaluations to measure answer quality or retrieval relevance on those cases. You compare two fixes in an experiment: a prompt-only change and an application change that persists session state explicitly. The experiment shows that the prompt edit helps little, while the state-handling fix resolves the repeated failure. That is Phoenix operating as a feedback system rather than as a collection of dashboards.

The strength of this model is that it scales from one bad trace to a disciplined improvement loop without changing tools or mental vocabulary. Traces give you evidence, evaluations give you standards, datasets give



you memory, experiments give you controlled comparison, and iteration turns findings into a better system.

## 2.4. Version Boundaries and Current Guidance

Fast-moving AI tooling creates a specific failure mode for experienced engineers: outdated examples often look plausible. An old import path still resembles the new one, a notebook still runs after minor edits, and a community blog post may describe a workflow that used to be preferred but is no longer the cleanest or safest way to build. Phoenix is under active development, with frequent releases through 2025 and 2026, so you should treat version context as part of the technical content rather than as incidental packaging detail.

A short compatibility check prevents most early mistakes. For current tracing examples, the boundary that matters most is the Phoenix OpenTelemetry package version. If you are using `arize-phoenix-otel` 0.16.0+, you can import tracing helpers directly from `phoenix.otel`. Earlier setups used `OpenInference` import paths instead.

```
# Current style for arize-phoenix-otel 0.16.0+
from phoenix.otel import register
from phoenix.otel import using_session, SpanAttributes
```

```
# Earlier style before arize-phoenix-otel 0.16.0+
# Imports came from OpenInference packages instead.
# Keep these examples as historical references only.
from openinference.instrumentation import ...
from openinference.semconv.trace import ...
```

The same rule applies to environment configuration. The verified current identifiers are:

```
export PHOENIX_API_KEY=your-api-key
export PHOENIX_COLLECTOR_ENDPOINT=http://localhost:6006
```

Those names are worth memorizing because stale tutorials often drift

first at the package and configuration boundary. If an example uses different environment variable names or an older import pattern without clearly explaining why, stop and verify it before you copy it into production code or internal documentation.

**Why version boundaries matter more in AI tooling** Version drift is not unique to Phoenix, but AI tooling amplifies it. A framework can change package boundaries, semantic conventions, and evaluation workflows within a short period while the broader ecosystem continues to circulate older notebooks and screenshots. This creates a dangerous kind of ambiguity: the code may look semantically right even when it is operationally stale.

That matters because Phoenix sits at the intersection of several moving layers. OpenTelemetry evolves. OpenInference semantics evolve. Agent frameworks evolve. Model APIs evolve. Phoenix itself evolves as it integrates those layers into a coherent tracing and evaluation workflow. Small mismatches across those boundaries can produce confusing failures: traces ingest but render poorly, semantic attributes do not map as expected, or evaluation examples appear structurally correct but reflect historical workflow assumptions.

You reduce that risk by treating version-sensitive guidance as scoped guidance. Whenever you document a concrete Phoenix workflow, you should be able to answer three questions:

- Which package or feature boundary does this example assume?
- Which parts are durable concepts and which parts are versioned API details?
- Which official release notes or current documentation would you check before publishing or operationalizing the example?

That posture is more disciplined than “use the latest.” It accepts that concepts such as traces, evaluations, datasets, and sessions are durable, while import paths, helper locations, and feature packaging may shift.

**The current tracing assumption** The cleanest current tracing assumption for Python is explicit: use `arize-phoenix-otel 0.16.0+` and import helpers from `phoenix.otel`. This boundary matters because it is one of the easiest places to accidentally mix eras. Older code may still circulate using `OpenInference` imports directly. That is historically meaningful, but it should not be presented as the default modern path unless you are deliberately documenting a legacy environment.

A compact compatibility table helps make this visible:

| Area                 | Current guidance                                                       | Historical pattern                                                                                   |
|----------------------|------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| Tracing imports      | <code>phoenix.otel</code> with <code>arize-phoenix-otel 0.16.0+</code> | Imports from <code>openinference.instrumentation</code> and <code>openinference.semconv.trace</code> |
| Collector config     | <code>PHOENIX_API_KEY</code> , <code>PHOENIX_COLLECTOR_ENDPOINT</code> | Verify carefully; old examples may use stale setup conventions or omit current names                 |
| Evals package        | <code>arize-phoenix-evals&gt;=3</code>                                 | Older eval examples may reflect earlier packaging or preferred flows                                 |
| Datasets/Experiments | Available in Phoenix 4.6+                                              | Phoenix 4.5 and earlier may not show these workflows                                                 |

This table is not merely housekeeping. It defines the assumptions under which code listings and walkthroughs remain trustworthy. If you work in a mixed codebase or inherit internal examples from several time periods, start by normalizing these boundaries before you debug deeper application behavior.

**The evals boundary is separate from tracing** Another common mistake is assuming that tracing and evaluation package guidance move together. They do not always. Current Python examples for

Phoenix Evals use `arize-Phoenix-evals>=3`. That should be treated as its own compatibility boundary.

A minimal current import pattern looks like this:

```
from phoenix.evals import LLM, evaluate_dataframe
from phoenix.evals.metrics import DocumentRelevanceEvaluator
```

This matters for two reasons. First, teams often find old notebooks that use older evaluation wrappers or slightly different packaging assumptions, then try to combine them with current tracing setup. Second, eval workflows are especially prone to cargo-cult adoption because they often appear in notebooks rather than in production repositories. A notebook from a previous release cycle can remain visually convincing long after the preferred import path or execution pattern has changed.

When you review evaluation examples, verify three things before adopting them: the package lineage, the evaluator import path, and whether the example assumes features introduced after older Phoenix releases. That discipline prevents a large class of subtle integration problems.

**Datasets and experiments have a historical feature boundary** Datasets and experiments are now central to the Phoenix feedback loop, but they were introduced in Phoenix 4.6+. That has a practical consequence when you read older material: screenshots, tutorials, or recordings from Phoenix 4.5 and earlier may not show those capabilities at all. Their absence does not mean your installation is broken; it may simply mean the artifact predates the feature boundary.

This is a useful example of why visual material ages differently from API examples. A stale import often causes an immediate runtime error. A stale screenshot causes a conceptual error instead. You may infer that a capability is optional, external, or missing from your deployment when in fact you are looking at a historical interface. For technical teams building internal enablement material, this is one more reason

to stamp screenshots and UI walkthroughs with a product version or release window.

The operational effect is larger than it first appears. If a team plans its process around old materials that lack datasets and experiments, it may end up treating Phoenix as a pure debugging UI and miss its role in systematic comparison and regression prevention. The result is an observability workflow that never matures into an engineering loop.

**Semantic conventions are still a compatibility surface** Tracing APIs are not the only moving part. The semantic layer around GenAI telemetry remains active as OpenTelemetry GenAI conventions and OpenInference mappings continue to evolve. You should therefore avoid writing as if every GenAI attribute name is permanently fixed across all versions and tools.

This does not mean the concepts are unstable in a useless sense. The concepts are clear and durable: prompts, model inputs and outputs, tool invocations, retrieved documents, token usage, and session context all belong in AI-aware telemetry. The caution applies to exact attribute mappings, naming details, and how different layers normalize those values over time.

That distinction changes how you should write and review technical material. If a concept can be explained without pinning a fragile attribute name, prefer the concept. If an exact key or mapping materially affects implementation, verify it against the current official documentation or release notes before treating it as normative. This is especially important for instrumentation chapters and internal platform standards, where teams may otherwise fossilize a transient convention into long-lived wrappers and policies.

**A practical taxonomy of change risk** Not every Phoenix topic carries the same compatibility risk. A useful way to reason about stability is to separate the stack into layers.

**Low-to-moderate risk: core concepts.** Traces, spans, evaluations, datasets, experiments, sessions, and OTLP transport are durable ideas. Their names and relationships are stable enough to anchor architectural thinking.

**Moderate risk: package boundaries and helper imports.** This is where many examples age first. The shift to `phoenixotel` for current tracing is the clearest example. These changes do not invalidate the underlying concept, but they can break code immediately.

**Moderate-to-high risk: workflow packaging.** Evaluation flows, client setup conventions, notebook patterns, and feature surfaces can change enough that old examples become misleading even if they still resemble current practice.

**High risk: semantic mapping details.** When integrations rely on evolving GenAI semantics, exact attribute names and rendering behavior can be a moving target. These details demand the closest tie to current docs.

This taxonomy helps you decide when to be strict. You do not need to re-verify the concept of a trace every month. You do need to re-verify helper imports, environment variable names, and version-sensitive examples before publishing or codifying them.

**How to read older Phoenix material safely** You will almost certainly encounter older examples in blog posts, notebooks, issue threads, and internal documents. Do not discard them automatically; many still teach valid ideas. Instead, read them with a migration lens.

First, identify whether the material is conceptual or procedural. Con-

ceptual material about why traces matter or why evaluations should connect to datasets often remains useful. Procedural material about imports, setup, UI steps, or package installation ages much faster.

Second, look for signals of historical context. If an example imports tracing components from OpenInference packages directly, treat it as `pre-arize-phoenix-otel 0.16.0+` style unless proven otherwise. If a walkthrough discusses Phoenix but never mentions datasets or experiments, it may predate Phoenix 4.6+ or simply focus narrowly on tracing; either way, verify before extrapolating.

Third, normalize examples into one era before you combine them. Mixing a legacy tracing setup with current evals and a modern UI mental model is a reliable way to confuse a team. Even when each individual piece once worked, the composite may not reflect a coherent supported workflow.

**Manuscript and team documentation hygiene** If you write internal standards, tutorials, or shared notebooks, version awareness needs to become a habit rather than a final review step. A few rules go a long way.

- Record the package boundary near every code listing that depends on it. If a tracing example assumes `arize-phoenix-otel 0.16.0+`, say so explicitly in prose or comments.
- Prefer verified configuration identifiers exactly as documented. For Phoenix collector setup, use `PHOENIX_API_KEY` and `PHOENIX_COLLECTOR_ENDPOINT` verbatim rather than paraphrased aliases.
- Date or version-stamp screenshots and UI walkthroughs. This matters most for features introduced after historical release boundaries, such as datasets and experiments in Phoenix 4.6+.

- Before publishing final examples, check the latest release notes for the areas most likely to drift: tracing imports, evaluation APIs, and semantic convention notes. This is not bureaucratic overhead. It is cheaper than asking every reader to rediscover compatibility mismatches independently.

**Anti-patterns to avoid** Several anti-patterns deserve explicit rejection because they look efficient at first and create long-lived confusion later.

**Copy-pasting plausible imports from old notebooks.** This often fails only after several layers of setup are already in place, wasting time and obscuring the real problem.

**Combining eras in one example.** A code snippet that mixes modern `phoenix.opentelemetry` imports with a stale evaluation pattern or a historical setup assumption becomes harder to debug than either era alone.

**Treating screenshots as current product specification.** UI artifacts are excellent learning aids and poor compatibility contracts.

**Writing versionless internal documentation.** Without explicit boundaries, your team cannot tell whether an example is authoritative, historical, or somewhere in between.

**Assuming GenAI semantic keys are timeless.** That invites brittle instrumentation wrappers and unnecessary migration pain.

**The operating rule** The safest working rule is straightforward: use current Phoenix examples only when you can tie them to verified package boundaries, configuration identifiers, and feature availability; treat older examples as historical references to be translated, not copied. For the workflows developed later, the modern baseline is `phoenix.opentelemetry` with `arize-phoenix-opentelemetry 0.16.0+` for tracing,



`arize-phoenix-evals>=3` for Python evals, `PHOENIX_API_KEY` and `PHOENIX_COLLECTOR_ENDPOINT` for collector configuration, and the understanding that datasets and experiments assume Phoenix 4.6+ era capabilities. Holding those boundaries steady lets you focus on the real engineering problems Phoenix is meant to solve instead of spending time untangling mismatched generations of setup guidance.

## Chapter 3

# Tracing Foundations: OpenTelemetry, OpenInference, and Phoenix Data Flow

*A good trace is not just a log with better formatting; it is an executable narrative of what an AI system believed, called, retrieved, waited on, and returned. This chapter shows how Phoenix depends on OpenTelemetry for transport and context, on OpenInference for AI-native meaning, and on careful span design for turning noisy distributed activity into evidence you can debug from with confidence.*

### 3.1. Traces, Spans, Attributes, and Context Propagation

A bad answer from an AI application rarely fails in one place. A user asks a short question. Your application retrieves documents, reformulates the prompt, calls an LLM, invokes a tool to fill a missing fact, retries after a timeout, and returns something that is still wrong. The individual components may each emit logs, but logs alone usually do not preserve the execution boundary of that one request or the causal structure inside it. You can see messages from the retriever, the model client, and the tool wrapper, yet still fail to answer the operational question that matters: *what happened, in what order, and which step shaped the final result?*

That is the job of tracing. Phoenix renders traces, but Phoenix is only readable if you already have a precise mental model of the objects inside the trace. Before you choose where to instrument, how to name spans, or which AI semantics to adopt, you need a strict vocabulary for execution.

#### A minimal concrete example

The shortest useful trace is not a full AI workflow. It is a root operation with a child operation. The point is to make identity and nesting explicit. In OpenTelemetry terms, a trace is composed of spans; each span has identifying context, timing, and structured metadata. The following Python example uses the standard OpenTelemetry tracing API to create a root span and one child span.

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("request") as root:
    root.set_attribute("user.id", "u-42")
    root.add_event("request.received")
```

```
with tracer.start_as_current_span("retrieve") as child:
    child.set_attribute("db.system", "vectorstore")
    child.set_attribute("query.text", "refund policy")
```

Even this tiny example encodes the core model. The span named `request` is the root of one trace. The span named `retrieve` is a child span nested within that root. Both spans have start and end times. Both can carry attributes. The root includes an event marking a meaningful point within the operation rather than a separate timed operation. That distinction becomes important once traces describe real AI pipelines.

#### **Trace and span as execution boundaries**

A *trace* is the complete record of one logical execution path. In an AI application, that usually means one user request, one agent turn, or one service operation that you want to reason about as a whole. The word *logical* matters. A trace is not every process on the machine and not the entire lifetime of a chat application. It is the bounded unit of work for which you need causal truth.

A *span* is a timed unit of work inside that trace. A span has a start time, an end time, a name, a `SpanContext` that identifies it within the trace, and optional structured data such as attributes and events. If the trace is the execution story, spans are the scenes with precise timing and parentage.

Parent-child relationships are semantic, not merely chronological. If retrieval runs and then the LLM call runs, that does not automatically mean one should be the child of the other. A child span should represent work causally contained by its parent. That is why the root span often maps to the request itself, while retrieval, prompt assembly, tool invocation, and model inference become children under that request. If a tool call is initiated by the LLM orchestration path, it may be a child of the LLM or of a broader orchestration span depending on where re-

sponsibility truly sits. A truthful hierarchy matters more than a convenient one.

Timing turns that hierarchy into something operationally useful. Start and end times let you decompose latency instead of treating total request time as a black box. If the request took four seconds, the trace can reveal whether three seconds were spent in retrieval, in a tool timeout, or in serial prompt assembly overhead that never touched the network. This is one reason spans should cover more than RPC boundaries. Internal application decisions can deserve their own spans when they materially affect latency, correctness, or control flow.

### **Trace identity and span identity**

OpenTelemetry separates the identity of a trace from the identity of a span. A trace has a `trace_id`. Each span has its own `span_id`. A child span shares the same `trace_id` as its parent but receives a new `span_id`. The child also records the parent relationship. This is how a backend such as Phoenix reconstructs the execution tree.

This distinction sounds mechanical until you debug a fragmented workflow. If your retrieval call and your LLM call appear under different `trace_id` values, they are not two steps in one execution narrative; they are two unrelated roots as far as the tracing system is concerned. You may know they belong together from application intuition, but Phoenix can only display the causality that the trace identity preserves.

### **Attributes as structured evidence**

Attributes are key-value pairs attached to a span. They are durable because they are structured, machine-readable, and available for filtering and grouping later. A log line like "retrieved 7 docs from kb-eu after fallback" is readable in the moment, but it is weak for analysis. Separate attributes such as `retrieval.document_count=7`, `kb.region="eu"`, and `fallback=true` are much more useful because

you can search, aggregate, and compare them across many traces.

For AI systems, attributes often carry the facts that explain behavior: model name, request mode, tool name, retrieval hit count, token counts, cache status, user tier, or a normalized error class. Phoenix relies on span structure to make traces analyzable; attributes are what let that analysis survive beyond one ad hoc incident.

#### **Attribute design and cardinality**

Not every fact should become an attribute. Two practical constraints dominate.

First, *cardinality* matters. Attributes with a small, stable set of values are easy to group and query. Attributes that contain highly unique values, such as raw prompts, generated text, or UUID-like payloads on every span, can create noise, storage pressure, and poor query behavior. Some AI payloads are worth keeping, but you should treat them as high-cost data and attach them only where they materially support debugging or evaluation.

Second, *shape stability* matters. If one span uses `model="gpt-4"` and another uses `llm.model_name="gpt-4"`, you have already reduced the analytical value of both. Later semantic conventions will narrow this further for AI workloads, but even at the foundational level the rule is simple: encode facts consistently enough that they remain queryable and comparable.

Attributes should also stay structured. Avoid packing multiple facts into one free-form string just because it feels convenient. If you need to filter on it later, break it into fields now.

#### **Events: points in time inside a span**

Events annotate something that happened *during* a span without declaring a new timed operation. They are timestamped records at-

tached to the span. Typical examples include `request.received`, `retry.started`, `first_token.received`, or `cache.miss`. An event is appropriate when the moment matters but the thing itself does not justify a nested span.

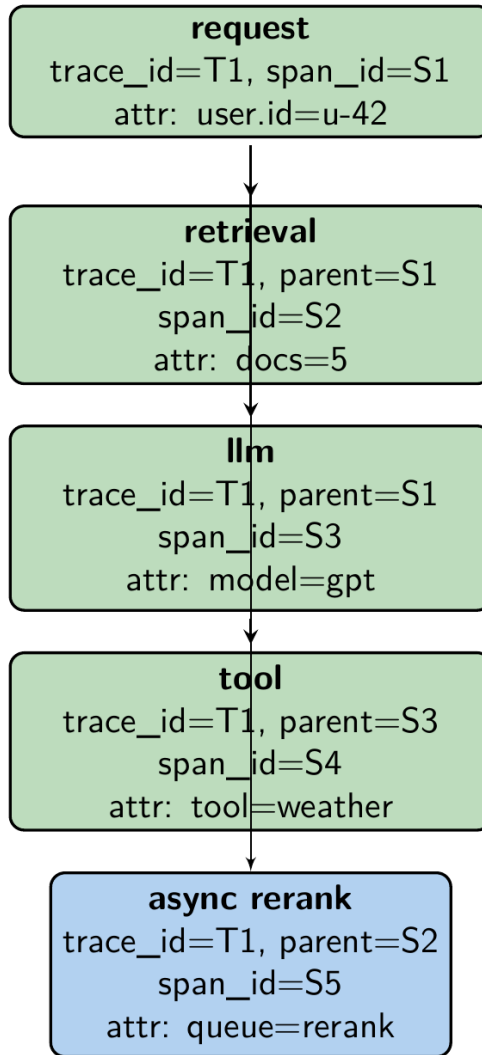
Use events sparingly. If the action has independent latency, status, or attributes that you need to analyze as an operation, model it as a child span instead. A tool call should almost always be a span. A retry marker inside a broader span may be an event, unless each retry attempt itself needs separate duration and outcome tracking.

### **Links: related, but not parent-child**

A hierarchy cannot express every execution relationship honestly. Some work is related to a trace without being causally nested under one parent at execution time. OpenTelemetry uses *links* for these non-hierarchical relationships.

Links are useful for fan-out and join patterns, asynchronous processing, batch consumption, or cases where a new span should refer to another span context without pretending to be its child. A common example in AI systems is a background job started from a request path but executed later by a worker. If the worker consumes a queued task after the original request scope has ended, you may choose to start a new trace and link it to the originating span rather than force a misleading parent-child tree.

Another case appears when sampled or partial upstream context prevents a full parent chain but you still want to preserve association. Links let you express relevance without falsifying containment.



**Context propagation preserves execution truth**



Tracing breaks down when identity fails to cross boundaries. OpenTelemetry uses *context* as the execution-scoped carrier for cross-cutting values, including the current span. Context is immutable: when code adds or changes values, it creates a new context rather than mutating a shared one in place. That immutability reduces accidental corruption, especially in concurrent code.

Within that context, `SpanContext` carries the identifiers needed to continue a trace. It is the information that identifies a span in the trace and must propagate to child spans and across process boundaries. OpenTelemetry also distinguishes *baggage*, which is a separate name/value propagation mechanism. Baggage is for small cross-service values you intentionally want to propagate. It is not a replacement for span attributes, and it is not the same thing as trace identity.

Across HTTP or messaging systems, propagation happens by injecting and extracting context through carriers such as request headers or message metadata. OpenTelemetry standardizes this through the `TextMapPropagator` abstraction. The mechanism varies by language and framework, but the conceptual rule does not: when a request crosses a boundary, you must transfer the trace context if you want downstream spans to join the same execution narrative.

### **Synchronous versus asynchronous boundaries**

Synchronous request chains are the easiest case. Service A receives a request, extracts incoming context, starts a span, and injects context into the outbound call to Service B. Service B extracts that context and starts its own child span. The trace remains contiguous.

Asynchronous paths are where teams often lose the narrative. Queues, worker pools, callbacks, thread hops, futures, and streaming handlers can break the current context if the runtime does not carry it automatically or if wrapper code ignores it. The symptom in Phoenix is familiar: one user action produces multiple root traces, each locally reasonable

but globally disconnected.

Streaming introduces a subtler version of the same problem. The root request span may remain active while token callbacks, tool completions, or post-processing handlers execute in different scheduling contexts. If those callbacks start new spans without the current context, the trace fragments even though the wall-clock sequence looks continuous in application logs.

### Common propagation failures

Several failure modes repeat across languages and frameworks.

- **Dropped headers or metadata.** A gateway, custom client, or proxy strips trace headers. Downstream code starts a new root because there is nothing to extract.
- **Broken async boundaries.** The current context is held in thread-local or task-local storage, but work resumes on a different execution unit without proper context transfer.
- **Wrapper code starts fresh roots.** Helper libraries call the tracer without using the current context, so each helper invocation creates a separate trace.
- **Background jobs are modeled as children by assumption.** Work that is delayed, retried, or independently scheduled is forced into a false hierarchy instead of being linked honestly.
- **Leaking stale context.** Reused worker threads accidentally carry old context into unrelated work, causing impossible parentage and misleading traces.

These are not cosmetic defects. A fragmented trace destroys latency decomposition, obscures retries, and can make a bad answer look like

a model issue when the real cause was a retrieval branch that silently escaped the request context.

### **Span granularity and readable traces**

A trace should be detailed enough to expose causality but not so granular that it becomes unreadable. Creating a span for every helper function, token callback, or formatting step usually produces a trace tree that is technically complete and operationally useless. A good span boundary corresponds to a decision, dependency, or resource cost you may need to isolate later.

For AI systems, useful boundaries often include the request root, retrieval, reranking when it is explicit and material, prompt assembly if it is a real source of latency or failure, LLM invocation, tool execution, and major post-processing stages. That is enough to preserve the execution narrative without drowning it in implementation noise.

### **Sensitive data and payload discipline**

Tracing systems make data easier to aggregate and inspect, which also makes careless payload capture more dangerous. Raw prompts, user documents, tool arguments, and generated outputs may contain regulated or sensitive material. Do not attach them indiscriminately just because the API allows arbitrary attributes. Prefer selective capture, redaction, or omission where appropriate, and reserve large payloads for places where they are essential to debugging or evaluation.

The same discipline helps performance. Rich AI spans can become expensive to export and store. Large, unstable attributes also make the trace harder to read. The goal is not maximum payload; it is maximum explanatory value per span.

### **A trace as the unit of execution truth**

Once you think in traces, many observability decisions become simpler.

A trace is the answer to “what happened for this one logical operation?” Spans decompose that answer into timed units of work. Attributes preserve structured evidence. Events mark meaningful points in time. Links model related work that hierarchy cannot represent truthfully. Context propagation keeps all of it attached to the same execution identity across services, threads, and asynchronous boundaries.

Phoenix displays these objects, but the display is only as honest as the model underneath. When a trace faithfully captures one request from entry to outcome, you can reason about a poor answer, a timeout, or a retrieval miss as one coherent system behavior rather than a pile of disconnected component logs.

## **3.2. OTLP Ingestion and the Phoenix Telemetry Pipeline**

You can instrument spans correctly and still fail to see reliable traces in Phoenix. That failure usually appears as intermittent gaps: one request arrives with full retrieval and LLM detail, the next shows only a root span, and a third never appears at all. At that point the problem is no longer trace semantics. It is transport, buffering, endpoint selection, shutdown behavior, or export failure. Treat the ingestion path as production infrastructure, because that is what it becomes the moment you rely on traces to debug user-facing failures.

A practical setup starts inside your application process. The tracer provider decides where finished spans go. A span processor decides when they leave memory. The exporter turns those spans into OTLP requests. Phoenix receives the exported telemetry, ingests it, and makes it queryable. If any hop is misconfigured, overloaded, or abruptly terminated, the trace you expected to analyze becomes partial or missing.

### **A concrete Phoenix-bound setup**

For Phoenix open source, use `phoenix.opel`, not `arize.opel`. Those package names are not interchangeable. A minimal direct export setup can be driven entirely by environment variables plus a Phoenix-aware tracer provider.

```
export PHOENIX_COLLECTOR_ENDPOINT=http://localhost:6006
export OTEL_BSP_SCHEDULE_DELAY=500
export OTEL_BSP_MAX_QUEUE_SIZE=2048
export OTEL_BSP_MAX_EXPORT_BATCH_SIZE=512
export OTEL_BSP_EXPORT_TIMEOUT=30000
```

```
from phoenix.opel import TracerProvider
from opentelemetry import trace

provider = TracerProvider(
    endpoint="http://localhost:6006"
)
trace.set_tracer_provider(provider)

tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("request"):
    pass
```

That example is intentionally small. It shows the key operational boundary: your code ends a span, the provider and its configured processor/exporter path take responsibility for shipping it to Phoenix. The exact defaults and ergonomics vary by release and language, but the transport model does not. Spans must be ended, processed, exported, transmitted, accepted, and indexed before they become visible.

## OTLP as the wire protocol

OTLP is the OpenTelemetry wire protocol for telemetry data between sources, collectors, and backends. It uses Protocol Buffers payloads over HTTP or gRPC and is stable for traces. That makes OTLP the common transport layer between your instrumented application and Phoenix, whether you export directly or route through an intermediate collector.

Two details matter operationally.

First, OTLP is *not* a magical stream where data simply exists once emitted. It is a request/response protocol. Your process constructs export requests, sends them to a destination, receives a response, and reacts to success or failure. That means network latency, timeouts, retries, TLS settings, proxies, and endpoint reachability are all part of tracing behavior.

Second, OTLP is hop-to-hop, not end-to-end exactly-once delivery. Duplicate telemetry is a known limitation. If an exporter times out waiting for a response and retries, the backend may receive the same batch twice. You should expect this possibility in production and avoid reasoning about trace counts as if every export were perfectly unique under failure.

#### **From ended span to visible trace**

The data path is easier to manage when you separate responsibilities:

- Your application creates and ends spans.
- A span processor receives completed spans.
- The processor either exports immediately or buffers.
- An exporter serializes spans as OTLP over HTTP or gRPC.
- Phoenix receives the payload and ingests it.
- Phoenix indexes the spans so the UI and query APIs can use them.

This pipeline explains several common surprises. A span that has started but not ended usually does not appear as a completed record. A span that ended moments ago may still be sitting in an in-process batch queue. A span may have been exported successfully but not yet surfaced where you expect because ingestion and indexing are not instantaneous. Conversely, a span missing from Phoenix may be absent

because the process died before the batch flushed, not because your instrumentation logic failed.

### **Batching versus immediate export**

Span processors determine how aggressively your process ships telemetry. The two operational extremes are immediate export and buffered batch export. Phoenix versioned API documentation verifies `phoenix.opentelemetry.BatchSpanProcessor`; it also documents Phoenix-aware exporter classes such as `phoenix.opentelemetry.HTTPSpanExporter` and `phoenix.opentelemetry.GRPCSpanExporter`. In practice, batching is the default production posture because it reduces per-span export overhead.

A batch processor accumulates finished spans in memory and exports them on a schedule or when a batch fills. The trade-off is straightforward:

- *Batching lowers overhead.* Fewer export calls mean less CPU, fewer sockets, and less request amplification.
- *Batching delays visibility.* A span may not appear in Phoenix immediately after it ends.
- *Batching risks loss on abrupt shutdown.* If the process crashes or is terminated before flush, buffered spans can disappear.
- *Batching introduces queue pressure.* Under burst load, the queue can fill and force drops or backpressure behavior depending on implementation.

Near-immediate export is useful during local debugging because you care more about fast feedback than efficiency. In production, immediate export can become surprisingly expensive when a request fans out into retrieval, reranking, tool calls, and streaming model activity. You trade observability freshness for request overhead on every completed span.

The batch knobs exposed through environment variables control this tension directly:

- `OTEL_BSP_SCHEDULE_DELAY` controls how long spans can wait before a scheduled flush.
- `OTEL_BSP_MAX_QUEUE_SIZE` bounds in-memory buffering.
- `OTEL_BSP_MAX_EXPORT_BATCH_SIZE` limits one export payload.
- `OTEL_BSP_EXPORT_TIMEOUT` bounds how long export can block.

A shorter schedule delay gives you fresher traces but more export calls. A larger queue absorbs bursty traffic but increases memory use and enlarges the amount of telemetry you can lose in a hard crash. A larger batch size improves export efficiency until payload size or timeout risk starts to dominate.

#### **Direct export versus collector-mediated topologies**

You have two broad topologies: export straight from the application to Phoenix, or export to an intermediate collector that then forwards to Phoenix.

*Direct export* is the simplest path. It minimizes moving parts, shortens failure chains, and works well for local development, single-service deployments, and environments where each application can reach the Phoenix collector endpoint directly. When you are establishing a known-good baseline, direct export is usually the right first step because it narrows the surface area of failure.

*Collector-mediated export* adds an operational hop but buys you central control. A collector can buffer, fan out to multiple destinations, apply policy, normalize network egress, and isolate application code from backend routing changes. In larger environments it often becomes the



practical place to enforce authentication, handle enterprise proxy constraints, or route telemetry across network boundaries the application itself should not manage.

The cost is complexity. Every additional hop creates new timeout modes, queue limits, and deployment dependencies. If traces disappear, you now need to decide whether the problem sits in the application SDK, the collector, the network path between them, or Phoenix ingestion itself. Use a collector because you need its capabilities, not because observability folklore says every deployment must have one.

### **HTTP versus gRPC transport**

OTLP supports both HTTP and gRPC transport. Both can carry trace telemetry correctly; the better choice depends on your deployment environment.

HTTP usually wins on deployment friction. It passes more easily through standard reverse proxies, ingress controllers, and enterprise networking policies. If your platform already standardizes on HTTP-based service egress and header-based authentication, OTLP/HTTP often reduces operational surprises.

gRPC can be a good fit when your environment already supports it cleanly and you want a transport optimized for protobuf-native RPC workflows. The gain is rarely conceptual; it is usually about consistency with the rest of your infrastructure and the behavior of your networking stack under load.

Do not make this choice doctrinally. If HTTP avoids proxy breakage and simplifies debugging, use HTTP. If gRPC integrates better with your internal telemetry routing and you know your platform handles it correctly, use gRPC. Operational fit matters more than transport fashion.

### **Authentication and headers**

Headers are part of the ingestion contract. Phoenix-related configuration may use `PHOENIX_CLIENT_HEADERS` or the more general `OTEL_EXPORTER_OTLP_HEADERS`, depending on how you configure export. These values become critical when telemetry must pass through authenticated gateways or when a managed environment expects API keys or tenant-routing metadata.

Keep the header path simple during initial validation. Start with the smallest direct path that can succeed, then add required auth headers, proxies, or collector forwarding rules one layer at a time. If you begin with a fully decorated production topology, you can spend hours debugging an authentication or routing issue while assuming your instrumentation is broken.

### **A validated exporter configuration**

When you need explicit control over the exporter rather than relying on inferred defaults, Phoenix’s versioned API documentation verifies exporter classes and batching support. The example below stays close to those documented identifiers.

```
from phoenix.otel import TracerProvider
from phoenix.otel import BatchSpanProcessor
from phoenix.otel import HTTPSpanExporter
from opentelemetry import trace

exporter = HTTPSpanExporter(
    endpoint="http://localhost:6006"
)
processor = BatchSpanProcessor(exporter)
provider = TracerProvider(endpoint="http://localhost:6006")

trace.set_tracer_provider(provider)
```

The exact way a given release wires provider, processor, and exporter may differ, so treat explicit construction as version-sensitive. The operational point is stable: processor choice controls buffering; exporter choice controls transport; endpoint choice controls where the telemetry leaves your process.

## Failure modes that create missing or partial traces

Most tracing outages are partial, not total. That is why they are hard to diagnose. Several patterns recur.

*Abrupt process death.* Serverless freezes, container kills, crash loops, and forced deploy restarts can terminate the process before buffered spans flush. The trace root may appear while late child spans vanish, especially if the root ended earlier and happened to export in a prior batch.

*Endpoint mismatch by environment.* Local development points at `localhost`, but the container runtime interprets that as the container itself rather than the host running Phoenix. The code appears correct, yet nothing arrives.

*Timeout and retry duplication.* The exporter sends a batch, Phoenix or an intermediary receives it, but the response is delayed or lost. The exporter retries. You now have duplicate spans unless the backend deduplicates or tolerates repeated data safely.

*Oversized payloads.* Large prompt or document attributes can inflate OTLP payload size enough to trigger timeouts, proxy limits, or collector rejection. Rich AI traces are valuable, but payload discipline is part of transport reliability.

*Backpressure in the middle hop.* A collector under load can accumulate queued telemetry faster than it forwards it. The application may still believe export succeeded to the collector while Phoenix visibility lags or drops.

*TLS or header misconfiguration.* The exporter reaches the endpoint but fails authentication or certificate validation. Depending on logging and retry behavior, this can look like silent trace loss until you inspect exporter diagnostics.

#### **Local success does not prove production success**

Tracing often works first on a developer laptop and fails in containers, jobs, or serverless runtimes. The reason is simple: local environments hide transport constraints. There may be no proxy, no egress policy, no sidecar, and no lifecycle pressure. In production, the telemetry path competes with memory limits, short shutdown windows, secret injection, and service mesh routing.

Validate in the shape you plan to run. If your workload will execute in Kubernetes, confirm that the pod can resolve and reach the configured endpoint. If it will run as a background worker, test shutdown behavior under termination. If it will run in a function platform, measure whether the runtime gives enough time for batch export on completion. Observability pipelines fail most often at deployment boundaries, not in isolated code snippets.

#### **Querying Phoenix to verify ingestion**

Do not rely only on the UI when validating ingestion. Phoenix supports programmatic query paths after data lands. Official documentation verifies `phoenix.client.Client` and `phoenix.trace.dsl.SpanQuery`, including filtering by project, time range, and `span_kind`. Even if your eventual workflow is UI-first, a small query makes ingestion validation more deterministic.

If you emit one known trace with one distinctive attribute and then query for it, you separate transport validation from UI navigation mistakes. This is especially useful in shared environments where multiple projects or time windows can make a trace appear “missing” when it is merely filtered out.

Because the exact query snippet can be version-sensitive, keep the principle fixed even if your local client code differs: generate a minimal trace, wait long enough for batch flush, and query for that trace or

a unique span attribute before diagnosing deeper instrumentation issues.

### **A practical validation sequence**

Use a short, disciplined checklist when you stand up or troubleshoot the ingestion path:

- Point one application directly at Phoenix with the simplest endpoint.
- Emit one minimal trace with a distinctive attribute.
- Confirm the process stays alive long enough to flush.
- Verify arrival through Phoenix UI or a client query.
- Increase load to confirm queue and batch behavior.
- Add required headers, TLS, or authentication.
- Insert a collector only if you need its control plane value.
- Repeat validation after each topology change.

This order matters. If you start with collector routing, custom headers, and production batch settings all at once, you cannot tell whether a missing trace reflects SDK setup, network policy, or backend ingestion.

### **Operational trade-offs that matter in practice**

A few decision signals are reliable enough to use repeatedly.

Choose *direct export* when you want fast setup, fewer moving parts, and clear blame assignment for early adoption or smaller deployments. Choose a *collector* when you need routing control, fan-out, buffering outside the application, or centralized policy.

Choose *shorter batch delays* when debugging freshness matters more than efficiency. Choose *larger, calmer batches* when production overhead dominates and traces do not need sub-second visibility.

Choose *HTTP* when proxies, ingress, and enterprise networking simplicity matter most. Choose *gRPC* when your environment already supports it well and you want consistency with existing protobuf RPC infrastructure.

Most importantly, do not interpret missing traces as proof of bad instrumentation until you have ruled out process lifecycle, buffering, endpoint reachability, and export failure. Telemetry pipelines lose data for the same ordinary reasons any distributed system does: queues fill, requests time out, processes die, headers disappear, and one environment behaves nothing like another. Once you start treating the path from ended span to Phoenix record as a real system with explicit trade-offs, the ingestion behavior becomes debuggable instead of mysterious.

### **3.3. OpenInference Semantic Conventions for AI Workloads**

A trace with perfect timing and propagation can still be analytically weak. If one library names a model call *completion*, another names it *chat*, and a third records the same operation as a generic application span, Phoenix can ingest all of them and still leave you with little basis for comparison. The transport succeeded, but the meaning did not. That is the gap OpenInference fills.

OpenInference is complementary to OpenTelemetry and is designed to enable tracing of AI applications. OpenTelemetry gives you the foundational execution model: traces, spans, context, and export. OpenInference adds an AI-aware semantic vocabulary on top of those primi-

tives so that model invocation, vector-store retrieval, external tool usage, and other AI-centric operations can be represented in a consistent way. The OpenInference specification is maintained in the `spec` directory and is transport- and file-format agnostic, which is exactly what you want from a semantic layer. It should survive changes in SDK ergonomics, wire protocols, and backend choices.

**A concrete example: semantics become queryable** The value of semantic conventions becomes obvious when you query traces after ingestion. Phoenix documentation shows filtering by `span_kind`, including values such as `RETRIEVER` and `LLM`. That means the semantic label is not decorative metadata. It becomes the basis for analysis.

```
from phoenix.client import Client
from phoenix.trace.dsl import SpanQuery
```

You would use a query like `SpanQuery` with a filter on `span_kind == 'RETRIEVER'` to isolate retrieval work across many traces. The exact method chain is version-sensitive and is best verified against the installed Phoenix client, but the operational point is stable: once your spans carry standardized AI semantics, Phoenix can separate retriever behavior from LLM behavior and make those categories analyzable at scale.

Without that semantic layer, the same query degenerates into brittle name matching such as “find spans whose name contains retrieval or search or vectordb.” That approach fails as soon as you switch frameworks, rename internal functions, or mix auto-instrumented spans from multiple libraries.

**Why generic tracing semantics are not enough** OpenTelemetry deliberately stays general. It can tell you that a span started at one time, ended later, belongs to a trace, and carries structured attributes.

That is necessary, but not sufficient, for AI systems. An LLM call is not merely an HTTP client operation. A retrieval span is not merely a database query. A tool span is not merely a subprocess or RPC. They have domain-specific meaning that affects how you debug, compare, and evaluate them.

You care whether a span represents retrieval because retrieval quality, document count, ranking behavior, and context contribution often determine answer quality. You care whether a span represents an LLM invocation because model choice, prompt structure, token usage, and latency shape both cost and output behavior. You care whether a span represents a tool call because tools can fail, retry, block, or silently return stale data while the outer request still succeeds.

Generic tracing semantics can encode these facts as arbitrary attributes, but arbitrary encoding does not create interoperability. OpenInference does that by defining a shared vocabulary for AI workloads so that different frameworks and libraries can emit semantically comparable telemetry.

**Semantic conventions as schema discipline** A semantic convention is a contract for how to describe an operation, not just that the operation occurred. In practice, that contract has several layers.

First, it gives you a **stable operation category**. Phoenix examples already verify live `span_kind` values such as `RETRIEVER` and `LLM`. Even these few verified kinds are enough to show the pattern: the backend can classify spans by AI role rather than by ad hoc span names.

Second, it gives you **stable attribute intent**. For an LLM span, you expect fields that describe model invocation context rather than transport trivia alone. For a retrieval span, you expect fields that expose query intent, returned document information, and ranking-related state. For a tool span, you expect invocation identity and outcome. The



exact key names should be verified against the current specification before production use, but the design principle is not version-sensitive.

Third, it gives you **provider neutrality**. A semantic span should keep meaning when you migrate from one model vendor, vector store, or orchestration framework to another. If your telemetry schema mirrors one vendor's API vocabulary too literally, your traces become less comparable the moment you change providers.

**Core AI operation classes** OpenInference's purpose explicitly includes representing LLM invocation context, vector-store retrieval, and external tool or API usage. Those three categories cover much of the operational surface of modern AI applications, and they expose distinct debugging questions.

**LLM invocation spans.** These spans represent model calls as first-class operations. Operationally useful metadata often includes model identity, request parameters, prompt-response linkage, and usage measurements. The semantic objective is not to capture every raw provider field. It is to preserve the information required to compare runs, explain latency and cost, and inspect output behavior without binding your traces to one provider's naming scheme.

**Retrieval spans.** These spans represent the search side of RAG systems: embedding lookup, vector similarity search, keyword retrieval, reranking, or a broader retriever abstraction, depending on how your system is structured. A retrieval span should tell you enough to answer whether the system found relevant context and what candidate set it considered. Document counts, selected document identifiers or summaries, and ranking hints usually matter more than low-level storage client details.

**Tool and external API spans.** These spans represent actions the system takes to gather facts, execute business logic, or interact with ex-

ternal systems. Tool semantics matter because a tool can dominate latency, fail intermittently, or produce low-quality inputs that the model then uses confidently. A generic “HTTP client” span often hides the operational purpose of the tool call; an AI-aware semantic layer preserves it.

OpenInference is also meant to support richer orchestration patterns such as agent behavior, but if your current Phoenix environment only verifies certain kinds directly, keep your implementation grounded in documented values and describe richer categories in provider-neutral prose until your local spec version confirms them.

**The difference between naming and meaning** Many teams think they can get equivalent value by carefully naming spans. A span named `rag.retrieve` is better than a span named `step2`, but naming alone does not create a durable schema. Names drift. Engineers abbreviate. Frameworks auto-generate names. New providers bring new vocabulary. A semantic convention gives you an explicit layer of meaning that survives those changes.

This distinction matters when you operate multiple services or multiple generations of the same system. Suppose your first implementation uses a chain framework that emits generic spans, your second uses more manual tracing, and your third migrates model providers. If you rely on span names alone, every dashboard and query becomes a set of historical exceptions. If you rely on stable semantic categories and attribute intent, Phoenix can compare traces across those migrations with much less custom translation.

**Why Phoenix benefits directly** Phoenix is valuable when it can treat AI operations as analyzable first-class events rather than opaque nested timing boxes. Semantic conventions are what make that possible. Once retrieval spans are represented consistently, Phoenix can iso-

late retrieval behavior across traces, group by retriever-specific fields, and support debugging patterns that are impossible with only transport metadata. Once LLM spans are represented consistently, Phoenix can compare model calls independently of how the orchestration code was written.

This is also where semantic conventions intersect with downstream evaluation. If retrieval spans consistently expose retrieval-specific structure, you can later join trace analysis with quality analysis without first reverse-engineering what each span meant. If LLM spans consistently expose invocation structure, you can compare prompt and response behavior across experiments using a common vocabulary instead of per-framework adapters.

**Schema design principles that hold up under change** A good AI trace schema is conservative, explicit, and stable.

**Use provider-neutral meaning.** Encode what the operation *is*, not just what one vendor called it. “LLM invocation” is durable. A provider-specific request object shape is not.

**Separate task semantics from transport metadata.** Network protocol details still matter for latency debugging, but they should not be the primary representation of AI work. A retriever span should first say that it is retrieval work, then optionally carry database or RPC details.

**Preserve comparison value.** Use field shapes that remain meaningful across framework upgrades and provider swaps. If a field is only interpretable when you know one library’s internal naming scheme, it will age poorly.

**Model intermediate operations explicitly when they affect diagnosis.** If reranking changes which documents reach the model, rep-

resent that step distinctly rather than burying it inside a generic retrieval blob. If a tool invocation decides whether the model sees external state, give it its own semantic operation.

**Control payload size and sensitivity.** AI spans invite overcapture. A semantic convention should help you decide what belongs as structured operational evidence and what should remain redacted, truncated, or omitted.

**Attribute richness without semantic sprawl** Semantic conventions solve one problem and create another: once you have a schema, it is tempting to encode everything. Resist that. You still need the attribute discipline established earlier. High-cardinality prompt text, full retrieved documents, and raw tool payloads may be useful in some traces and destructive in others. The presence of a semantic home for AI fields does not justify indiscriminate payload capture.

The best practice is to preserve *explanatory* data rather than maximal data. For retrieval, that may mean query intent, document counts, and selected identifiers before full document bodies. For LLM calls, that may mean structured model settings and prompt-response linkage before every intermediate token. For tool calls, that may mean tool identity, arguments summary, and outcome before whole response envelopes.

This is one reason the OpenInference specification being transport-agnostic matters. It focuses the schema on meaning, not on one backend's storage convenience or one exporter's payload quirks.

**Language and package boundaries** The OpenInference project spans multiple language ecosystems. Verified package identifiers include `openinference-semantic-conventions`, `openinference-instrumentation`, `@arizeai/openinference-semantic-convent`

and Go module paths under `github.com/Arize-ai/openinference/go/...`. That cross-language presence is a practical signal: the semantic layer is meant to travel across heterogeneous stacks, not remain trapped inside one Python-only tracing story.

Phoenix-specific compatibility deserves care as well. Official documentation notes that `arize-phoenix-otel 0.16.0+` allows importing `using_session` and `SpanAttributes` from `phoenix.otel`; older versions require imports from `openinference-instrumentation` and `openinference-semantic-conventions`. That boundary matters because it shows how the same semantic ideas can surface through different package entry points over time. If you are writing production code, verify the package surface of the version you have installed before adopting import paths from examples.

**Auto-instrumentation versus semantic sufficiency** Auto-instrumentation can produce spans quickly, but speed is not the same as semantic completeness. A framework integration may detect an outbound model call and produce a span that is technically useful. It may still miss your business-specific retrieval stages, reranking steps, or application-level tool orchestration. That gap is where teams often overestimate what “instrumented” means.

Use auto-instrumentation to capture common operations consistently, then layer manual spans where your orchestration adds meaning the framework cannot infer. The semantic convention layer helps here because it gives manual and automatic spans a shared vocabulary. Without that shared vocabulary, you end up with one family of auto-generated spans and a completely different family of custom spans that cannot be compared cleanly.

**Semantic drift and migration tolerance** Semantic drift is one of the quiet failure modes of observability systems. It happens when

traces remain technically valid but become less comparable over time. One team renames spans. Another adds custom attributes with different keys. A provider migration introduces new terminology. Six months later, the dashboard still renders, but the data no longer supports longitudinal analysis.

Prevent drift by treating semantic conventions as versioned schema, not casual naming guidance. Document the operation categories you rely on. Keep a small approved set of stable fields for each major AI operation type. When you adopt a new framework or provider, map its concepts into your established semantic model instead of letting the new vocabulary leak directly into production traces.

This matters most when you want to compare behavior across experiments, deployments, or incidents. Phoenix can only help you ask durable questions if the telemetry keeps durable meaning.

**Anti-patterns that weaken interoperability** A few mistakes repeat often.

- **Provider-specific taxonomies.** If your span model is built entirely around one vendor's product names or request types, migration becomes a schema rewrite rather than a configuration change.
- **Prompt dumping as schema design.** Storing entire prompts or responses in arbitrary attributes without stable surrounding semantics creates expensive traces that are hard to query and compare.
- **Mixing business outcome with model outcome.** A request can succeed at the business level while an individual model or tool span fails and triggers fallback logic. Keep those meanings separate so Phoenix traces remain truthful.

- **Assuming auto-instrumented spans are semantically complete.** They often capture framework behavior, not the full reasoning structure of your application.
- **Using span names as the only semantic contract.** Names help readability, but they do not replace a stable, queryable vocabulary.

The difference between merely transportable telemetry and reusable operational knowledge is the presence of disciplined semantics. OpenTelemetry tells Phoenix how spans relate and how they arrive. OpenInference tells Phoenix what those spans mean in an AI system, which is why retrieval, model invocation, and tool execution can become first-class evidence rather than a pile of generic timings.

### 3.4. Reading Phoenix Traces as Execution Narratives

A trace becomes useful when you stop reading it as a list of spans and start reading it as evidence. A user reports that the answer was slow, weakly grounded, and oddly confident. The request did not simply “call an LLM.” It moved through retrieval, perhaps embedding generation, prompt assembly, one or more model invocations, optional tool execution, and final response generation. Phoenix is built to show that sequence as a detailed timeline of a request. If you read the trace in the right order, it can tell you where time went, which intermediate steps shaped the outcome, and where the behavior first diverged from what you expected.

The interpretive mistake is to jump straight to the last visible error or the slowest child span. That often identifies a symptom rather than the cause. A model span may be long because retrieval returned too

many documents. A tool span may fail because the model asked the wrong question of the tool. A root span may report success even when a critical child span failed and fallback logic hid the defect. Reading Phoenix well means reconstructing causal structure, not merely spotting red rows.

**A concrete query-first example** A practical way to ground trace reading is to extract the fields you care about into a queryable table. Phoenix documentation verifies that the client and DSL support selecting fields such as `input.value`, `output.value`, `parent/child joins`, and exploding `retrieval.documents` into document-level rows.

```
from phoenix.client import Client
from phoenix.trace.dsl import SpanQuery
```

Using `SpanQuery`, you can select retrieval spans and project fields such as `input.value`, `output.value`, or `retrieval.documents`. You can also explode `retrieval.documents` so that one retrieval span becomes multiple rows, one per retrieved document. The exact method names are version-sensitive and should be verified in your installed Phoenix client, but the workflow is stable: query the spans, flatten the retrieval payload into per-document evidence, and compare what the retriever returned with what the model later used.

That single capability changes how you debug RAG systems. Instead of treating retrieval as a black box that “must have been fine because it returned 10 documents,” you can inspect the actual candidate set at the document level and ask whether the model ever had a chance to answer correctly.

**Start with the execution boundary** The first reading task is to identify what the trace claims to represent. In Phoenix, a trace should correspond to one logical request or one coherent unit of execution.



You want the root span because it defines the boundary of the story. It tells you what the system believed it was doing, when it started, when it ended, and often what high-level input and output were attached to the request.

If the root span is too broad, the narrative becomes muddy. A single trace for an entire long-lived chat session often hides causality because unrelated turns accumulate under one oversized root. If the root span is too narrow, meaningful work falls outside the trace and you lose the very path you were trying to explain. For most interactive AI systems, one trace per user turn or request is the cleanest unit of reasoning.

When you inspect the root, establish four facts before going deeper:

- What was the user or upstream service trying to do?
- Did the overall request succeed, fail, or fallback?
- How long did the entire request take?
- Which major child spans partition that time?

That framing prevents you from chasing local anomalies that did not materially affect the request outcome.

**Read the hierarchy before reading the details** Phoenix traces are strongest when spans correspond to semantically meaningful AI steps rather than arbitrary function boundaries. A good trace tends to expose a root request span, with child spans for retrieval, model invocation, tool execution, post-processing, or related orchestration steps. Read that hierarchy top-down.

Parent-child structure gives you causal decomposition. If retrieval is a child of the root and the LLM is another child, the trace claims that both are major components of the request. If a tool span is nested under the

LLM span, the trace claims that the tool call occurred within the model-driven reasoning path or orchestration around it. If a span appears at the same depth as the LLM instead of under it, that may signal a different design choice: perhaps the application orchestrator, not the model call itself, owns the tool invocation.

This matters because causal interpretation depends on structure. Two spans that happen near each other in time are not necessarily related in the way you think. Hierarchy lets you distinguish “this happened during the request” from “this happened because this earlier operation initiated it.”

**Use time to isolate latency contributors** After you understand the hierarchy, inspect duration. Phoenix traces are meant to show the sequence of retrieval, embedding generation, model invocation, and response generation as a detailed timeline. The timeline is not just a convenience for visualization. It is your first tool for latency decomposition.

Ask three questions in order:

- Which child spans dominate wall-clock time?
- Which spans overlap, and which serialize?
- Did the user-visible latency come from one expensive step or from many small steps chained poorly?

A long LLM span can mean slow model inference, but it can also indicate that the span includes streaming, retries, or tool-mediated waiting. A retrieval stage that looks cheap in isolation may still be the culprit if it triggered a much larger downstream prompt and inflated model latency. A short but repeated tool span can dominate the request when

retries stack up. The timeline helps you separate expensive work from merely visible work.

Be careful with overlap. If retrieval and embedding generation overlap, summing child durations will overstate total latency. Read duration as evidence of time spent in a component, not as a naive additive accounting system.

**Attributes turn spans into stateful evidence** Timing tells you where something happened. Attributes tell you what state the system was in when it happened. Once you have identified the important spans, inspect the fields that preserve inputs, outputs, model settings, retrieved documents, or tool metadata. Phoenix query support for `input.value` and `output.value` is especially useful because it lets you follow the transformation of state across operations rather than looking only at final output.

For a retrieval span, ask whether the query was aligned with user intent, how many documents were returned, and whether the retrieved set appears relevant. For an LLM span, compare the prompt-side input to the response-side output and ask whether the model had the right evidence and instructions. For a tool span, inspect the tool request and response at a level that explains behavior without assuming the tool is correct just because it returned successfully.

This is where semantic conventions pay off. If spans are labeled and attributed consistently, you can compare retrieval spans across many traces, model spans across providers, and tool spans across workflows without rebuilding meaning each time.

**Exploding retrieval documents exposes hidden context** RAG failures are often hidden in compressed retrieval payloads. A span may carry `retrieval.documents` as a collection, which is

informative but difficult to reason about when left nested. Phoenix supports exploding `retrieval.documents` into document-level rows. Operationally, that means you can inspect each retrieved document as its own record rather than as one large blob attached to a single span.

That capability is more powerful than it first appears. It lets you ask questions such as:

- Did the top-ranked documents actually mention the user topic?
- Were there duplicates or near-duplicates that crowded out diversity?
- Did the system retrieve stale policy text instead of the current one?
- Did the retrieved set include relevant material that the model ignored?

These are not merely retrieval questions. They are narrative questions. A weak final answer may reflect a model defect, but if the retrieval rows show low-signal documents, the trace already contains a stronger causal hypothesis: the model answered poorly because the context it received was poor.

**Look for the first meaningful divergence** A disciplined reading order keeps you from blaming the final symptom. Start at the root, identify the major stages, inspect the longest or most semantically central child spans, and then locate the first point where the actual path diverged from the expected one.

This idea is especially important in AI systems because late-stage spans often absorb the consequences of earlier errors. The LLM output may be obviously wrong, but the first meaningful divergence might be:

- a retrieval span that returned irrelevant documents,
- a prompt assembly step that dropped crucial context,
- a tool span that returned an empty result and forced hallucination,
- an orchestration span that retried unnecessarily and consumed the latency budget before the model ever answered.

The first meaningful divergence is the point at which a later bad outcome became likely. That is usually the most valuable span to study, because fixing it changes the request narrative rather than merely decorating the failure.

**Distinguish failure classes by narrative shape** Different failure modes leave different trace shapes.

*Retrieval failure.* Retrieval spans are present, but the retrieved documents are low-signal, duplicated, stale, or mismatched to user intent. The LLM may be fast and syntactically correct while still producing an ungrounded answer.

*Model-side failure.* Retrieval looks healthy, but the LLM span shows prompt-output behavior that suggests instruction failure, over-truncation, model mismatch, or poor reasoning despite good context.

*Tool failure.* A tool span is slow, retried, or semantically wrong. The outer request may still succeed, but the answer quality or latency reflects tool behavior more than model behavior.

*Orchestration failure.* The trace contains many small spans, repeated loops, or serial steps that should have been parallel. No single child fails dramatically, but the narrative reveals waste or poor control flow.

This is why Phoenix traces should be read as stories with causal structure. Failure classes are often visible in the shape of the trace before they are visible in the final text response.

**Interpret missing spans carefully** A missing span does not always mean missing work. It may mean broken context propagation, incomplete instrumentation, or export loss in the telemetry pipeline. Narrative continuity in Phoenix depends on correct OpenTelemetry context propagation; broken context produces missing or mis-parented steps. If you see a root request and an LLM span but no retrieval span, do not immediately conclude that retrieval never happened. Ask whether retrieval is instrumented, whether its context crossed async boundaries correctly, and whether the process lived long enough to flush all child spans.

This distinction matters because a trace can mislead you in two opposite directions. You can mistake an instrumentation gap for a system behavior gap, or you can ignore a real system omission because you assume “the trace must be incomplete.” Experienced operators learn to ask whether the observed structure is plausible given how the code actually runs.

**Sessions widen the narrative across turns** A single trace is usually the right unit for one request, but many AI applications are conversational. Phoenix sessions turn separate traces into conversation threads, enabling narrative analysis across turns instead of per-request isolation. That changes the interpretive question from “why did this one answer fail?” to “how is this interaction evolving over time?”

Consider a support bot handling a billing problem over five user turns. Each turn should remain a separate trace so that per-turn latency and causality stay readable. The shared session identity then ties those traces into a longer story. You can inspect whether the system repeat-

edly retrieved the same stale policy, whether tool failures clustered on certain turns, whether resolution time grew because the model kept missing the same entity, or whether a fallback path recovered after an earlier failure.

Sessions are especially useful for memory and persistence bugs. A single turn can look locally correct while the multi-turn narrative reveals that context was not carried forward, that the assistant asked for the same information repeatedly, or that earlier retrieval failures poisoned later reasoning.

**Common reading mistakes** Several anti-patterns show up even among experienced engineers.

*Overfocusing on the last error.* The final error may be downstream fallout from an earlier retrieval or orchestration problem.

*Ignoring successful but expensive spans.* A span does not need to fail to be the root cause of a bad user experience. A successful tool retry loop can still dominate latency.

*Confusing correlation with causation.* Two spans that are adjacent in time are not automatically causally related. Use hierarchy and attributes to verify the dependency.

*Treating every missing field as evidence.* Sometimes the real issue is incomplete instrumentation or semantic inconsistency rather than system behavior.

*Reading traces without an expected path in mind.* A trace is easiest to interpret when you know what a healthy execution would look like for the same workflow.

A useful habit is to articulate one concrete hypothesis after the first pass through a trace. For example: “The answer was weak because re-

trieval returned broad policy snippets instead of account-specific documents,” or “Latency was dominated by repeated tool retries, not by the model.” Then validate that hypothesis against span timing, attributes, and neighboring operations instead of browsing aimlessly.

**From trace inspection to operational judgment** Phoenix does not make debugging automatic. It gives you a representation rich enough to support disciplined reasoning. The strongest traces let you reconstruct a request as a sequence of semantically meaningful AI operations with timing, inputs, outputs, and preserved causality. When that representation is present, you can move from symptom hunting to evidence-based diagnosis: isolate the span where the path first went wrong, distinguish retrieval problems from model problems, separate tool unreliability from orchestration waste, and decide what deserves to be fixed before the next incident arrives.





## Chapter 4

# Instrumentation Strategies for Real-World AI Systems

*Instrumentation is where observability strategy becomes either durable signal or expensive noise. This chapter teaches readers how to get useful traces quickly, decide which execution boundaries deserve attention first, and evolve toward disciplined instrumentation that supports debugging, searchability, evaluation, and long-term maintainability without repeating foundational tracing concepts already covered in Chapter 2.*

## 4.1. Fast-Path Auto-Instrumentation Across Providers and Frameworks

Most teams delay tracing for the wrong reason. They assume the first useful deployment requires a schema workshop, a manual span plan, and code changes across every execution path. That assumption is expensive because it keeps you blind during the phase when your AI system is changing fastest. Phoenix gives you a much shorter path: emit OpenTelemetry traces, install the instrumentors that match your stack, and inspect real application traffic before you decide where deeper custom instrumentation is worth the effort.

A good first deployment has a narrow objective. You are not trying to model every business rule or every internal helper. You want coherent traces for live requests that reveal the major execution stages already exposed by supported SDKs and frameworks: model calls, retrieval where the framework emits it, orchestration boundaries such as chains or agents, and obvious failures. If your first trace shows where latency accumulates, which model call failed, and whether retrieval or tool execution occurred, you have enough signal to start making disciplined decisions.

### Fastest working path in Python

If your application already uses a supported provider SDK or framework in Python, the shortest setup is Phoenix OTEL registration with automatic discovery of installed OpenInference instrumentors. The important detail is that `auto_instrument=True` only activates instrumentors that are actually present in the environment. It does not synthesize support for packages you have not installed.

```
pip install arize-phoenix-otel \
  openinference-instrumentation \
  openinference-instrumentation-openai \
  openinference-instrumentation-langchain
```

You then point the application at a Phoenix collector and register tracing during startup.

```
import os

os.environ["PHOENIX_COLLECTOR_ENDPOINT"] = \
    "http://localhost:6006/v1/traces"
os.environ["PHOENIX_PROJECT_NAME"] = "support-assistant"

from phoenix.otel import register

tracer_provider = register(
    project_name="support-assistant",
    auto_instrument=True,
)
```

That is enough to establish a baseline for many Python applications. If the environment includes supported instrumentor packages and your runtime actually exercises those libraries, Phoenix will start receiving spans over OTLP.

### **A minimal request path to validate the setup**

Run representative traffic immediately after startup. Avoid synthetic no-op checks. You want a request that actually hits the provider or framework layers you expect to see in the trace.

```
from openai import OpenAI

client = OpenAI()

response = client.responses.create(
    model="gpt-4.1-mini",
    input="Give a one-sentence explanation of tracing.",
)

print(response)
```

The precise response shape depends on the SDK version and call you use, but the point of this first run is simple: force a real model invocation through a library that has an installed OpenInference instrumentor. If tracing is wired correctly, Phoenix should receive at least one

trace containing an LLM-related span and service metadata for your process.

### **What auto-instrumentation is actually doing**

Auto-instrumentation works by hooking supported SDK and framework execution points and translating them into OpenTelemetry spans enriched with OpenInference semantics. Phoenix accepts traces over OTLP, so from the application’s perspective the key job is to produce structurally correct spans and export them. Phoenix then interprets those spans using the standard OpenTelemetry model plus AI-specific conventions such as `openinference.span.kind`.

That separation matters operationally. Auto-instrumentation is not “Phoenix magic” inside your business logic. It is a set of library-specific interceptors and wrappers that observe calls your application already makes. If your app invokes OpenAI through an instrumented SDK, the model call can be captured. If you use LangChain or LlamaIndex through supported integration points, chain, retriever, or tool activity may also appear. If you bypass those surfaces with handwritten orchestration or internal adapters, the auto-instrumentor has nothing stable to attach to.

### **Selecting the right integration surface**

You usually have several possible attachment points:

- instrument the provider SDK directly;
- instrument the orchestration framework;
- instrument both;
- skip auto-instrumentation and move directly to manual spans.

For a fast baseline, start with the highest-level supported surface that still reflects the execution structure you care about. If your application

is mostly framework-driven, framework instrumentation often gives you the richest early visibility because it can expose retrievers, chains, agents, or tools in addition to the underlying model call. If your application is thin and mostly wraps a provider SDK directly, instrumenting the SDK may be enough to establish latency and failure visibility quickly.

Use both layers carefully. Layered instrumentation can be helpful when the framework and provider spans form a sensible hierarchy, but it can also produce redundant spans if both wrappers describe nearly the same operation. You should expect this risk whenever one integration delegates into another instrumented integration. The fix is not guesswork; inspect the resulting trace tree and decide whether the added detail is meaningful or just duplication.

### **A JavaScript example when your stack is not Python-first**

If your application runs in Node.js, the activation path is more explicit. You configure an OpenTelemetry tracer provider and register the specific instrumentation package for the framework you use. For LangChain.js, the documented instrumentation class is `LangChainInstrumentation`.

```
npm install --save \
  @opentelemetry/sdk-trace-node \
  @arizeai/openinference-instrumentation-langchain

const { NodeTracerProvider } =
  require("@opentelemetry/sdk-trace-node");
const { LangChainInstrumentation } =
  require(
    "@arizeai/openinference-instrumentation-langchain"
  );

const provider = new NodeTracerProvider();
provider.register();

new LangChainInstrumentation().setTracerProvider(provider);
```

The package boundary matters here. LangChain.js compatibility dif-

fers by major version, so you should verify that the instrumentation package matches the LangChain line you are actually running. Treat version compatibility as part of the setup, not as cleanup after traces look strange.

### **What you should expect to see in a healthy first trace**

A useful baseline trace is structurally coherent before it is comprehensive. Look for these signals:

- A stable service identity, usually anchored by `service.name`.
- A root or near-root span corresponding to the request path that triggered the AI work.
- Child spans for supported framework or provider operations.
- OpenInference span kinds appropriate to the work performed, such as CHAIN, RETRIEVER, TOOL, or LLM.
- Error status on failing operations rather than only application logs.
- Reasonable timing relationships: child spans should fit inside the parent execution window unless the work is intentionally decoupled.

A healthy minimal setup often looks incomplete in one specific way: it captures standardized library interactions but not the business logic that prepared them. That is acceptable at this stage. What is not acceptable is a trace that looks complete but is structurally misleading.

### **Structural validation before you trust the data**

The first traces should be validated like a protocol integration, not admired like a dashboard. Run a short checklist against a representative request.

- Trigger one request whose execution path you understand.
- Count the major steps you expect from the libraries in use.
- Inspect the trace tree in Phoenix.
- Verify parent-child relationships.
- Verify that failures appear on the span that actually failed.
- Record what important work is still invisible.

Suppose a RAG request should perform these steps: request entry, retrieval, model call, final response assembly. If your trace only shows the model call, auto-instrumentation is still useful, but you now know exactly what is missing. That is the right outcome for a first pass. It tells you where the supported wrappers stop and where your own code starts to matter.

### **Resource identity and project separation**

Even a fast-path deployment needs a small amount of discipline around identity. Set a project name and a stable service identity so traces are attributable across environments and processes. If you do not, your first week of data quickly becomes hard to interpret.

Phoenix supports environment variables such as `PHOENIX_COLLECTOR_ENDPOINT`, `PHOENIX_BASE_URL`, `PHOENIX_API_KEY`, `PHOENIX_PROJECT_NAME`, and `PHOENIX_CLIENT_HEADERS`. Use them deliberately. Keep the project boundary meaningful, and avoid mixing local development noise with staging or production traffic if multiple environments share a collector path.

```
export PHOENIX_COLLECTOR_ENDPOINT=\
http://localhost:6006/v1/traces
export PHOENIX_PROJECT_NAME=support-assistant
```



If your deployment model already distinguishes services and environments through OpenTelemetry resource attributes, preserve that model rather than inventing tracing-only naming rules. Stable identity pays off immediately once you compare traces across workers, services, or deployment tiers.

### **Where auto-instrumentation captures high value quickly**

Auto-instrumentation is strongest when the most important work in your application already flows through common provider or framework APIs. In practice, that gives you rapid visibility into several expensive or failure-prone operations:

- provider model invocations, which often dominate latency and cost;
- embeddings generation through supported SDK layers;
- retrieval and orchestration steps exposed by supported frameworks;
- tool or agent operations surfaced by those frameworks;
- exception paths attached to the intercepted call boundaries.

This is why the fast-path approach is so effective early in a project. It captures the parts of the system that are both operationally expensive and structurally standardized. Those spans are often enough to answer immediate questions such as:

- Is latency concentrated in retrieval or generation?
- Did the provider call fail or return normally?
- Did the agent invoke tools before responding?

- Did the application skip retrieval on a path where you expected it?

### **Where it stops**

The boundary is equally important. Auto-instrumentation is weak wherever the decisive work is custom, implicit, or outside the supported wrapper chain. You should expect missing visibility in these areas:

- prompt assembly performed in handwritten code;
- proprietary ranking or reranking logic;
- business-rule gates, policy checks, and entitlement decisions;
- custom routers that choose models, tools, or prompts;
- asynchronous background tasks that lose trace context;
- data transformations that change the meaning of downstream inputs.

These are not edge cases. In many production AI systems, the root cause of a bad answer or policy violation lives exactly in this hidden layer. Auto-instrumentation can tell you that a model call happened and how long it took. It usually cannot tell you why your custom retrieval filter silently dropped the only relevant document, why your router selected the wrong tool class, or why a post-processing step corrupted structured output.

That limitation should shape your expectations from day one. Treat auto-instrumentation as a visibility baseline, not as proof that the full execution path is represented.

### **Common failure modes during first deployment**

Three failure modes show up repeatedly.

*Missing spans because the instrumentor is not installed.* The call path is real, but the corresponding OpenInference package is absent, so `auto_instrument=True` has nothing to discover. This usually looks like partial traces that capture some libraries but not others.

*Duplicate spans from layered instrumentation.* You instrument both a framework and an underlying SDK, and the resulting trace tree becomes noisy or misleading. Duplicates often share very similar timing and payload semantics. If both layers are valuable, keep them. If one adds no diagnostic leverage, disable the extra layer.

*Broken parent-child structure across async boundaries.* A model call appears as an isolated trace or attaches to the wrong parent because context propagation was lost in background execution, callbacks, or task queues. Auto-instrumentation cannot repair missing application context on its own.

When you hit these issues, resist the urge to patch blindly. Run one known request, compare observed spans against the execution path you expected, and identify whether the problem is package discovery, integration overlap, or context propagation.

## Controlling noise while you validate

During rollout, you may need to suppress tracing in narrow code paths that distort the picture. Phoenix OTEL exposes a documented context manager for that purpose.

```
from phoenix.otel import suppress_tracing

def warm_cache():
    with suppress_tracing():
        # background work you do not want traced
    pass
```

Use suppression sparingly. If you suppress too early, you can hide the

very paths that would have shown you context propagation or duplication problems. Suppression is most useful after you have already confirmed the shape of a healthy request trace.

### **A practical validation routine**

A disciplined first-pass routine keeps you from over-investing in the wrong layer.

- Install only the instrumentors required for the stack you actually run.
- Register Phoenix tracing and export to the collector.
- Generate a small batch of representative requests.
- Inspect one successful trace and one failing trace.
- Check span kinds, names, durations, and parentage.
- Write down every important step missing from the trace.
- Decide whether missing steps are acceptable for now or require manual spans.

That last step is the real payoff. You do not need a perfect first trace. You need an honest one. Once the baseline tells you exactly which parts of the runtime are visible and which are not, you can instrument the missing business-critical boundaries with precision instead of wrapping code indiscriminately.

### **Version-sensitive habits that matter immediately**

Instrumentation packages and framework integrations evolve quickly enough that you should build a few safety habits into the fast path itself. Pin the instrumentation packages you install, especially when a framework has explicit compatibility boundaries across major versions.

Keep the package list small. More installed instrumentors do not automatically produce better traces; they increase the chance of overlap, version drift, and accidental noise. After each dependency upgrade, re-run the same representative trace checks rather than assuming a successful import means semantic compatibility.

That approach keeps the first deployment lightweight without making it disposable. You get useful Phoenix traces quickly, but you also establish the habit that matters most in production observability: trust traces only after you have validated their structure against the system you know is actually running.

## 4.2. Choosing What to Instrument First

Once you have a baseline trace pipeline working, the hard problem changes. You are no longer asking how to get *any* traces into Phoenix; you are deciding which execution boundaries deserve engineering effort before everything else. Mature teams often get this wrong by instrumenting whatever is easiest to wrap: helper functions, utility modules, or framework callbacks that are convenient but diagnostically weak. That produces a lot of spans and very little explanation.

A better approach is to rank instrumentation targets by how much they improve failure localization. If a span does not help you answer why the user saw a slow, expensive, incorrect, or non-compliant result, it is rarely a first-wave candidate. The fastest way to increase debugging value is to anchor the request narrative, capture the cost and latency drivers, and expose the steps that change control flow or semantic meaning.

### **Start with a concrete boundary map**

Before you add a single manual span, write down the major execution

boundaries for one real request path. Keep the list short and operational. For a typical retrieval-augmented assistant, a useful first-pass map looks like this:

```
BOUNDARIES = [  
    "request.entry",  
    "retrieve.documents",  
    "assemble.context",  
    "llm.generate",  
    "tool.execute",  
    "response.assemble",  
]
```

That list is not code you must run. It is a forcing function. It makes you name the places where latency accumulates, external cost is incurred, branching occurs, or user-visible meaning changes. If your system does not perform tool calls, drop that boundary. If it performs routing before generation, add a route-selection boundary. The point is to identify the smallest set of spans that can explain a bad outcome.

For AI systems instrumented with OpenInference semantics, the official span kinds give you a practical prioritization heuristic. High-value categories usually include AGENT, CHAIN, TOOL, LLM, and RETRIEVER. Those labels are useful not because they are fashionable, but because they correspond to execution stages that frequently dominate latency, cost, correctness, or branching behavior.

### **A simple scoring rubric**

When time is limited, score instrumentation candidates instead of debating them abstractly. Use a small set of criteria that map directly to operational pain:

- User impact: if this step fails, does the user notice?
- Debugging value: does this step isolate a common class of incidents?
- Latency and cost exposure: does this step consume time or paid

API calls?

- Branch complexity: does this step change downstream behavior?
- Business risk: can this step trigger policy, compliance, or entitlement errors?
- Reusability: does the same boundary exist across many flows?

A lightweight worksheet helps make the decision explicit.

```
Candidate Impact Debug Cost Branch Risk Reuse
request.entry 5 5 1 3 4 5
retrieve.documents 4 5 3 4 2 4
llm.generate 5 5 5 3 3 5
tool.execute 4 4 4 5 4 3
response.assemble 4 4 1 3 3 4
helper.normalize_text 1 1 1 1 1 2
```

You do not need perfect weights. You need a ranking that survives contact with production incidents. Notice how the trivial helper loses even though it may be easy to instrument. Ease of wrapping is not the same as explanatory value.

### Instrument the execution narrative before the details

The first priority is the request narrative: where work starts, how it branches, and where the user-visible response is finalized. If you miss those anchors, even well-instrumented internal steps become hard to interpret.

**Request entry** comes first because it establishes the trace root for a user operation. This is where you attach durable identifiers, service identity, and enough context to separate one request from another. Without a clear entry boundary, everything downstream looks detached or ambiguously grouped.

**Final response assembly** belongs in the first wave for the same reason. Many systems return a model output only after post-processing,

filtering, formatting, policy checks, or fallback handling. If your trace ends at the provider span, you cannot tell whether the application delivered the raw model output, rewrote it, dropped it, or rejected it.

These anchors are also where resource-level identity matters. OpenTelemetry resources default `service.name` to `unknown_service`. If you leave that default in place, cross-service filtering becomes needlessly weak, especially once multiple workers and environments send traces to the same backend. Set `service.name` explicitly, and use `service.namespace` and `deployment.environment.name` when they help partition traffic cleanly.

### **A schema stub for the first wave**

You do not need a full schema design exercise to make good prioritization decisions, but you do need a minimal identity model. A practical stub looks like this:

```
TRACE_SCHEMA = {
  "resource": {
    "service.name": "support-assistant-api",
    "service.namespace": "customer-support",
    "deployment.environment.name": "prod",
  },
  "span": {
    "openinference.span.kind": "CHAIN",
    "app.workflow": "answer_question",
    "app.route": "knowledge_base",
  },
}
```

This example is intentionally small. The resource block identifies the service and environment. The span block captures a stable AI span category plus two bounded business dimensions. Bounded matters. You want values that are searchable and comparable across many traces, not arbitrary strings that explode cardinality.

### **Model calls are nearly always first-tier**

Model invocation spans are almost always worth instrumenting early,



whether they arrive through auto-instrumentation or explicit spans around custom provider interactions. They typically score high on every practical axis:

- they consume external cost;
- they often dominate latency;
- they fail in ways that matter immediately;
- they reveal prompt and parameter problems;
- they are stable across many application flows.

This does not mean the model is always the root cause. It means model spans give you leverage. If a request is slow, they tell you whether the provider call was actually responsible. If a response is low quality, they let you distinguish model behavior from orchestration errors. If the call never happened, that is just as valuable: the missing LLM span tells you the fault lies earlier in the pipeline.

You should still resist overfitting the first wave to provider-centric telemetry. A system can show perfectly healthy model spans and still be wrong because the real defect lives in retrieval, routing, guardrails, or output transformation.

### **Retrieval is first-tier when evidence quality matters**

For RAG systems, retrieval boundaries usually deserve the same urgency as model calls. Wrong answers often start earlier than the generation step: the retriever chose poor documents, the context assembly truncated the relevant snippet, or a filtering rule removed the best evidence. If you only trace the model call, you are left diagnosing evidence problems indirectly.

Prioritize retrieval when any of these conditions hold:

- users complain that answers are plausible but unsupported;
- latency spikes correlate with vector or search backends;
- context windows are tight and document selection matters;
- multiple retrieval strategies or indexes exist;
- reranking or filtering meaningfully changes the final context.

In those systems, retrieval spans often explain more incident classes than additional detail inside the model span. Instrument the point where external data becomes model context. That boundary marks a semantic transition: before it, the system is gathering evidence; after it, the model is reasoning over a specific context set.

#### **Tool execution and routing deserve attention when control flow is the problem**

Some systems are not primarily retrieval-bound. They fail because the application chooses the wrong tool, enters unnecessary tool loops, or routes the request to the wrong workflow. In those cases, instrumentation should move closer to control-flow decisions.

Tool boundaries are especially valuable when tools are expensive, stateful, or business-critical. A database lookup, ticketing API mutation, payment action, or CRM update should appear as a distinct span because the downstream consequences are much larger than the code footprint suggests. These operations often deserve TOOL or AGENT categorization if they are part of an agentic control path.

Routing boundaries also deserve early instrumentation when they alter the semantics of everything that follows. If a classifier sends one request to a retrieval pipeline and another to a transactional tool path, that routing decision is not a minor implementation detail. It is a diagnostic hinge. One wrong route can make every subsequent span look superficially healthy while the overall behavior is still incorrect.

## **Quality-critical transformations are often invisible but decisive**

A common blind spot is the transformation layer between major system stages. Prompt construction, context trimming, document filtering, schema coercion, response validation, safety filtering, and fallback selection can all alter user-visible meaning without creating external API calls. They are less obvious than retrieval or generation, but they often determine whether the system behaves acceptably.

Do not instrument every transformation. Instrument the ones that satisfy at least one of these tests:

- the step changes the information seen by the model;
- the step applies a business rule or policy;
- the step can reject or rewrite the result;
- the step frequently appears in incident review;
- the step creates hidden branching not visible elsewhere.

This is where many teams either instrument too much or too little. They either wrap every helper function and drown in low-signal spans, or they omit transformations entirely and miss the step that actually explains the bug.

### **What not to instrument first**

The anti-pattern is straightforward: do not optimize for code coverage. A trace with fifty helper-level spans rarely beats a trace with six carefully chosen boundaries.

Consider the difference between these two approaches:

Bad first wave:

```
- tokenize_input
- normalize_text
- build_headers
- parse_docs
- sort_candidates
- trim_whitespace
- stringify_response
- map_error
```

Good first wave:

```
- request.entry
- retrieve.documents
- assemble.context
- llm.generate
- tool.execute
- response.assemble
```

The first list mirrors the source tree. The second mirrors the operational story. You debug systems through the operational story.

Avoid top-level spans for tiny pure functions, formatting helpers, or internal adapters unless they are directly implicated in failures. They add overhead, clutter the trace, and distract from boundaries that actually explain behavior. If a helper becomes important later, you can add an event or a scoped child span around the specific high-risk use site rather than promoting the helper itself to a global tracing concept.

### **Broad-but-shallow versus narrow-but-deep**

You will often face a strategic choice: instrument a few spans across the whole request path, or instrument one subsystem in rich detail. The right answer depends on the dominant incident pattern.

Choose **broad but shallow** when:

- the system is new and you do not yet know where failures cluster;
- incidents span multiple subsystems;
- you need end-to-end latency attribution quickly;

- teams are arguing about whether retrieval, routing, or generation is at fault.

Choose **narrow but deep** when:

- one subsystem already dominates incident reviews;
- you have clear evidence that a hidden transformation is decisive;
- a compliance or safety boundary requires auditable detail;
- one expensive external tool is driving cost or failures.

A practical pattern is to start broad across request entry, retrieval, model, tool, and response assembly, then deepen only the subsystem that remains ambiguous. This keeps the trace readable while still giving you a controlled path to finer-grained spans.

### **Identity and cardinality discipline affect prioritization**

Instrumentation scope is not only about where to place spans. It also depends on whether the resulting metadata will remain queryable. High-cardinality fields such as raw user identifiers are poor candidates for top-level schema dimensions. They make filtering noisy, complicate retention decisions, and can create privacy problems. If you need user context for debugging, store it with masking or in controlled metadata rather than promoting it to a primary slice-and-dice dimension.

Prefer bounded business attributes that support recurring operational questions: route name, retrieval strategy, model family, tool class, tenant tier, or workflow type. Those dimensions make early instrumentation more valuable because you can compare traces across stable categories instead of chasing one-off identifiers.

**Use the OpenTelemetry layering model to keep priorities clean**

The OpenTelemetry semantic model helps keep your instrumentation plan disciplined. Put service identity in resource attributes. Put operation-specific details in span attributes. Use events for notable moments inside a span, and links when causality matters without strict nesting. That layering prevents a common prioritization mistake: inventing new spans because you really just needed an attribute or event.

For example, if a request enters a retrieval span and later records that the top document was filtered for a policy reason, you may not need a separate span for the filter if it is instantaneous and not independently diagnosable. An event may be enough. If the filter is expensive, branch-heavy, or frequently wrong, then it may deserve its own span. The decision depends on diagnostic leverage, not on whether the code occupies its own function.

### **A practical first-wave checklist**

When deciding whether a boundary belongs in the first wave, ask these questions in order:

1. Does this boundary anchor the user-visible request narrative?
2. Does it materially affect cost, latency, or correctness?
3. Does it change control flow or semantic interpretation?
4. Does it correspond to an OpenInference category such as AGENT, CHAIN, TOOL, LLM, or RETRIEVER?
5. Will it stay meaningful as the implementation evolves?
6. Can you attach bounded, searchable attributes to it?

If the answer is consistently yes, instrument it early. If the boundary is only convenient, only local to one helper, or only interesting during a single debugging session, keep it out of the initial design. The strongest

first traces are not the ones with the most spans. They are the ones that turn ambiguous failures into a small number of concrete suspects.

### 4.3. Implementing Manual Spans for Custom Logic and Business-Critical Steps

Auto-instrumentation gives you a fast baseline, but it rarely captures the code that actually decides whether the system behaved correctly. A trace can show a healthy provider call and still fail to explain the incident because the decisive mistake happened in authorization logic, routing, retrieval filtering, output transformation, or a policy gate that lives entirely in your code. That is where manual spans matter. You add them at the points where your application changes control flow, applies business rules, or transforms data in ways that affect the user-visible result.

The goal is not to wrap every function. The goal is to make the trace tell the truth about the system. If a hidden step changes what the model sees, what tool gets called, whether the response is allowed to leave the system, or how downstream work is scheduled, that step deserves explicit representation.

**A concrete manual span for a policy gate** A policy or entitlement check is a strong first example because it is usually invisible to framework instrumentation and often decisive during incident review. The low-level Python API for manual spans is standard OpenTelemetry: obtain a tracer with `trace.get_tracer(__name__)` and create a span with `tracer.start_as_current_span("name")`.

```
from opentelemetry import trace
from opentelemetry.trace import Status, StatusCode

tracer = trace.get_tracer(__name__)
```

```
def enforce_budget_and_policy(user_tier, est_cost_cents, route):
    with tracer.start_as_current_span(
        "policy.check"
    ) as current_span:
        current_span.set_attribute(
            "openinference.span.kind", "CHAIN"
        )
        current_span.set_attribute("app.route", route)
        current_span.set_attribute("app.user_tier", user_tier)
        current_span.set_attribute(
            "app.estimated_cost_cents", est_cost_cents
        )
        current_span.add_event("policy.check.started")

        if user_tier == "free" and est_cost_cents > 5:
            current_span.set_attribute(
                "app.policy.decision", "deny"
            )
            current_span.set_status(
                Status(StatusCode.ERROR)
            )
            current_span.add_event("policy.check.denied")
            raise PermissionError("budget exceeded")

        current_span.set_attribute(
            "app.policy.decision", "allow"
        )
        current_span.add_event("policy.check.passed")
```

This span adds three kinds of signal. First, it makes the gate visible in the trace tree. Second, it records bounded attributes that explain the decision. Third, it marks failure on the span where the business failure actually occurred rather than letting the trace imply that the downstream model call was the problem.

The attributes stay bounded and queryable: tier, route, estimated cost, decision. You should avoid putting full user records, raw prompts, or serialized policy blobs into top-level span attributes. Those payloads are harder to search and increase privacy and cardinality risk.

**Choosing boundaries that deserve their own spans** The easiest mistake in manual instrumentation is to follow the code structure



rather than the operational semantics. A helper function boundary is not automatically a tracing boundary. Create a span when the unit of work satisfies at least one of these conditions:

- it changes user-visible meaning;
- it changes control flow;
- it can fail independently in a diagnostically useful way;
- it invokes another system or expensive local process;
- it applies a policy, ranking, or transformation that often explains incidents.

That guideline usually highlights a small set of candidates: authorization checks, prompt-template selection, retrieval filtering, reranking, tool routing, cache lookup decisions, response validation, guardrail enforcement, and asynchronous post-processing that affects delivery or persistence.

A useful mental test is this: if the span disappeared from the trace, would incident triage become substantially harder? If the answer is no, keep it as an event, an attribute, or leave it uninstrumented.

### **Use OpenInference categories to keep the trace readable**

Manual spans should not invent a separate taxonomy from the rest of your telemetry. AI-specific work benefits from the same OpenInference span kinds used by automatic instrumentors. If you are instrumenting custom business-critical tools, guardrails, evaluators, prompt-template boundaries, or orchestration steps, set the proper OpenInference kind so Phoenix can interpret the trace coherently.

In practice, that means choosing kinds that match the operational role of the step. A custom tool invocation should look like a `T00L`. A

guardrail or policy filter that mediates the response should not be mislabeled as an LLM. A multi-step orchestration boundary may be better represented as CHAIN or AGENT, depending on how it coordinates downstream work.

The exact kind does more than improve naming. It tells anyone reading the trace what sort of operation they are looking at before they inspect the attributes. That becomes important once traces include a mix of automatic provider spans and manual application spans.

**A custom tool span with bounded attributes** Tool execution is a common place where custom code carries most of the business risk. Auto-instrumentation may show the model deciding to call a tool, but the trace often needs more detail about how the tool actually behaved. A manual span around the tool boundary makes that visible.

```
from opentelemetry import trace
from opentelemetry.trace import Status, StatusCode

tracer = trace.get_tracer(__name__)

def run_customer_lookup_tool(customer_id, tenant_tier, cache):
    with tracer.start_as_current_span(
        "tool.customer_lookup"
    ) as current_span:
        current_span.set_attribute(
            "openinference.span.kind", "TOOL"
        )
        current_span.set_attribute(
            "tool.name", "customer_lookup"
        )
        current_span.set_attribute("app.attempt", 1)
        current_span.set_attribute(
            "app.tenant_tier", tenant_tier
        )

        cache_hit = customer_id in cache
        current_span.set_attribute("app.cache_hit", cache_hit)
        current_span.add_event("tool.lookup.started")

    try:
        if cache_hit:
```

```
        result = cache[customer_id]
    else:
        result = fetch_customer_record(customer_id)

    current_span.add_event("tool.lookup.completed")
    return result

except Exception as exc:
    current_span.record_exception(exc)
    current_span.set_status(
        Status(StatusCode.ERROR)
    )
    current_span.add_event("tool.lookup.failed")
    raise
```

This pattern is compact but expressive. The span name is stable. The attributes are bounded. Events mark milestones inside the work. Failures are recorded on the span with both `record_exception(...)` and an explicit error status. If the tool is retried, you can track an attempt counter without encoding retry numbers into the span name.

You should also notice what the example does *not* do. It does not attach the full customer record. It does not copy every argument into attributes. It does not create a child span for every helper called by the tool. The trace needs enough information to explain behavior, not a replay log of local execution.

**Decorators and context helpers for higher-level manual instrumentation** Phoenix/OpenInference also provide higher-level abstractions that can reduce boilerplate when your custom logic naturally maps to AI application concepts. Verified decorators include `@tracer.agent`, `@tracer.chain`, and `@tracer.tool`. Context managers such as `using_session`, `using_user`, and `using_metadata` let you enrich related spans with shared context.

These helpers are useful when you already know the semantic role of a function and want the trace to reflect that role consistently. They are

especially effective in orchestration code, where the trace readability benefit is larger than the small abstraction cost. Use them when they make the code clearer. Drop to the low-level OpenTelemetry API when you need precise control over events, links, error handling, or unconventional span boundaries.

If you use context helpers for user or session identity, apply the same cardinality discipline discussed earlier. Session-level grouping can be valuable. Raw, unbounded user identifiers as high-visibility attributes are usually not.

**Events versus spans** A trace becomes noisy quickly if every interesting moment becomes its own span. OpenTelemetry already gives you a lighter-weight mechanism: events inside a span. Use `current_span.add_event(...)` for notable milestones that matter to understanding the operation but do not deserve independent timing and hierarchy.

Good event candidates include:

- route chosen;
- cache miss observed;
- policy threshold exceeded;
- fallback path activated;
- reranker returned no candidates.

Create a child span only when the sub-operation has distinct duration, error behavior, or diagnostic value. For example, a costly reranking call to an external service probably deserves its own span. A quick local threshold comparison usually does not.

This distinction improves both performance and cognition. Spans should represent units of work. Events should represent meaningful moments within those units.

**Parent-child structure and granularity** A good trace reads like a faithful execution narrative. Parent-child nesting should therefore express real containment, not merely a convenient implementation detail. If your request handler invokes a routing function, which invokes retrieval, which invokes the model, nesting those spans is natural because the downstream work happens within the lifetime of the upstream orchestration.

Granularity becomes a design problem as soon as you instrument hidden logic. Too coarse, and one span called `process.request` conceals the actual failure boundary. Too fine, and the trace tree turns into a source-code outline. A practical rule is to create one span per business-relevant phase, not one span per function. Prompt construction with substantial branching may deserve a span. String concatenation helpers inside it do not.

Another useful rule is stability. If a span name would need to change every time you refactor a helper or move code across files, you are probably instrumenting too close to the implementation. Prefer names that describe durable operations such as `guardrail.validate_response`, `route.select_workflow`, or `retrieval.rerank`.

**Linked spans for causal relationships without nesting** Not every relationship is a clean parent-child tree. If one operation triggers deferred or asynchronous work, nesting it as a child of the original request can misrepresent the lifecycle. OpenTelemetry span links exist for exactly this case: the operations are causally related, but not temporally contained in the same execution scope.

A common example is post-processing scheduled after the user response has already been returned. You may enqueue transcript enrichment, indexing, or offline scoring for later execution. That work should be traceable back to the original request, but it should not appear as though the original request was blocked waiting for it.

The precise link-construction code depends on the surrounding OpenTelemetry setup, but the design rule is stable: use links when causality matters and parent-child nesting would tell the wrong timing story. That distinction prevents misleading latency interpretation in Phoenix.

**Handling failures correctly** Manual spans are most valuable when something goes wrong, so failure handling deserves deliberate treatment. Three steps matter:

1. record the exception on the current span;
2. set the span status with `Status(StatusCode.ERROR);`
3. add bounded attributes that explain the decision or failure class.

If you skip status and exception recording, Phoenix may show a suspiciously successful-looking trace with no obvious fault boundary. If you only log the exception elsewhere, you separate the failure from the timing and context needed to interpret it. The span that failed should carry the failure.

Be selective with failure metadata. Error type, retry attempt, route, policy verdict, or tool class are valuable. Dumping full stack traces or entire request payloads into attributes usually is not. Let the span capture searchable dimensions and let application logs carry verbose text when necessary.

**Context continuity across custom code** Manual instrumentation is only as good as its context propagation. If a span starts outside the current trace or loses the current context before downstream operations run, the trace becomes fragmented. This is especially easy to do in background tasks, callbacks, executors, or message-driven flows.

When you instrument custom async or deferred work, verify the resulting trace structure rather than assuming that a span was attached to the correct request. A manual span that appears as a separate trace can still be technically emitted, but it fails its real purpose: explaining how the user-visible request led to the later action.

That is also where links become useful. If lifecycle boundaries make strict nesting incorrect, preserve causal continuity with links rather than forcing a false hierarchy.

**Controlling trace noise in mixed workloads** Production systems often execute internal benchmark runs, offline evaluations, or notebook experiments alongside real application traffic. If those flows generate spans indiscriminately, they can distort the operational picture. Phoenix provides a documented way to suppress tracing inside a block of code.

```
from phoenix.opentelemetry import suppress_tracing

def run_offline_eval(eval_runner):
    with suppress_tracing():
        return eval_runner()
```

This is particularly useful when evaluation calls reuse the same provider SDKs and would otherwise look like customer-serving traffic. Suppress noise at the source when the activity is operationally irrelevant to production traces.

If you need to remove auto-instrumentation entirely in a test or notebook context, documented instrumentor instances also support

`.uninstrument()`. That is useful during partial rollouts or when you need to validate manual spans without interference from automatic wrappers.

**An iterative refinement workflow** The most reliable way to add manual spans is iterative and empirical.

1. Start from a real incident class: wrong route, bad retrieval, policy rejection, malformed response, or slow tool.
2. Identify the hidden step that would have explained it earlier.
3. Wrap that step in a manual span.
4. Add only the bounded attributes needed to explain the outcome.
5. Add events for important milestones inside the step.
6. Run representative traffic and inspect the trace tree in Phoenix.
7. Remove or merge spans that add noise without diagnostic gain.

This workflow keeps the trace grounded in actual debugging value. It also prevents a common failure mode where instrumentation grows faster than the team's ability to interpret it.

**Common anti-patterns** A few mistakes reliably reduce the value of manual spans.

*One span per helper function.* This mirrors the source tree instead of the operational story.

*Runtime values in span names.* Names such as `tool.lookup.customer_1234` destroy grouping and searchability.



*Large unstructured payload attributes.* They inflate storage, increase privacy risk, and are hard to query.

*Custom taxonomies that fight OpenInference semantics.* If your provider spans use AI-native kinds and your manual spans use ad hoc labels, the trace becomes harder to interpret.

*Suppressing or unlinking the wrong work.* Noise control is useful, but suppressing a boundary that often explains incidents removes the very signal you added tracing to capture.

The strongest manual spans are sparse, semantically stable, and anchored to work that actually changes outcomes. Once those spans exist, Phoenix stops being merely a record of provider calls and starts reflecting the real system you operate.

## 4.4. Designing High-Signal Trace Schemas

A trace can be perfectly emitted and still be operationally weak. That usually happens when the schema grows by accident: span names mirror source files, attributes capture whatever data happened to be nearby, and identifiers drift from service to service. You then have plenty of telemetry but very little analytical leverage. Months later, the team cannot answer simple questions such as which retrieval strategy correlates with bad outcomes, whether a cache miss pattern explains latency spikes, or which workflow variant broke after a release. High-signal trace design prevents that failure by treating telemetry as an information architecture problem rather than a code-generation problem.

The central design principle is simple: make traces easy to group, compare, and interpret over time. OpenTelemetry already gives you the right layering model. Use resources for service identity, span names for

operation labels, span attributes for bounded dimensions, and events for notable in-span milestones. OpenInference extends that model with AI-native semantics, especially `openinference.span.kind` and structured fields for messages, tool calls, outputs, and retrieval artifacts. If you follow those layers consistently, Phoenix can normalize and query your traces in ways that remain stable as the application evolves.

**Start with a small, explicit schema skeleton** Before you debate individual fields, establish a schema skeleton that separates durable identity from operation detail. A minimal but useful starting point looks like this:

```
TRACE_SCHEMA = {
  "resource": {
    "service.name": "support-assistant-api",
    "service.namespace": "customer-support",
    "deployment.environment.name": "prod",
  },
  "spans": {
    "agent.plan": {
      "openinference.span.kind": "AGENT",
      "app.workflow": "answer_question",
      "app.feature_flag": "router_v2",
    },
    "retriever.fetch": {
      "openinference.span.kind": "RETRIEVER",
      "app.retrieval_strategy": "hybrid",
      "app.retrieval_count": 8,
      "app.cache_hit": False,
    },
    "llm.generate": {
      "openinference.span.kind": "LLM",
      "app.model_provider": "openai",
      "app.model_name": "gpt-4.1-mini",
    },
    "tool.execute": {
      "openinference.span.kind": "TOOL",
      "tool.name": "customer_lookup",
      "app.tenant_tier": "enterprise",
    },
  },
}
```

This is a schema stub, not a runtime API. Its value is conceptual. It shows four layers working together:

- resource attributes identify where the trace came from;
- span names identify what operation occurred;
- `openinference.span.kind` classifies the AI role;
- bounded span attributes capture the dimensions you will actually filter and compare.

The schema is intentionally narrow. It records workflow name, feature flag, retrieval strategy, retrieval count, cache hit, model provider, model name, tool name, and tenant tier. Those are stable business or operational dimensions. They support grouping and comparison across many requests.

**Schema layers should have distinct jobs** A common source of noise is putting the right information in the wrong layer. Once that happens, searchability and durability both degrade.

*Resources* identify the emitting service or process. Put broad identity here: `service.name`, `service.namespace`, and `deployment.environment.name`. These values should be stable enough that you can compare traces across releases and deployments without rewriting every saved query.

*Span names* label operations. They should describe durable units of work such as `agent.plan`, `retriever.fetch`, `llm.generate`, `tool.execute`, or `response.render`. Span names should not encode runtime values, request IDs, model parameters, or user-specific data.

*Span attributes* hold bounded dimensions that explain differences among otherwise similar operations. Cache hit or miss, retrieval strategy, tool name, policy outcome, workflow name, and tenant tier all fit well here.

*Events* record notable moments inside a span without creating extra hierarchy. Retry attempts, fallback activation, partial parse failure, or a timeout threshold crossing often belong here.

That separation is not just stylistic. It controls query cost, keeps traces legible, and makes schema drift easier to manage.

**Prefer semantic conventions over ad hoc naming** OpenTelemetry’s primary recommendation is to use semantic conventions wherever possible. That advice is more valuable in AI systems than many teams initially realize. Without conventions, every service invents its own vocabulary. One team records `tool_type`; another uses `action`; a third uses `plugin_name`. Months later, cross-service analysis becomes a string-parsing exercise.

OpenInference gives you a strong anchor for LLM application semantics. At minimum, treat `openinference.span.kind` as a first-class part of the schema. It helps normalize interpretation across traces containing AGENT, CHAIN, LLM, RETRIEVER, TOOL, and other AI-native operations. Where OpenInference provides structured concepts for messages, tool calls, outputs, or retrieval artifacts, prefer those representations over ad hoc free-form blobs.

You will still need application-specific fields. Semantic conventions do not replace domain context. They give you a common frame so your custom fields remain interpretable instead of becoming a parallel telemetry language.

**Choose span names for operational durability** Span names are one of the most visible parts of the trace and one of the easiest places to introduce long-term pain. A good name is stable, semantic, and implementation-agnostic. A bad name is tied to a class name, file name, or temporary experiment.

These names age well:

```
agent.plan
retriever.fetch
llm.generate
tool.execute
response.render
guardrail.validate
route.select
retrieval.rerank
```

These names do not:

```
run_v3_chain
handler_step_2
call_openai_for_ticket_qa
customer_lookup_1234
rerank_docs_beta_path
```

The first set describes operations. The second set describes transient implementation details or embeds runtime values. Stable naming matters because saved searches, dashboards, regression checks, and evaluation workflows tend to depend on span names. If names change every release, the telemetry loses continuity exactly when you need historical comparison.

A practical naming rule is to combine a durable subsystem noun with an operation verb. Use lowercase dotted names consistently. Keep them terse enough to scan in a trace tree.

**Bounded attributes create analytical leverage** A span becomes analytically valuable when it includes attributes that explain

outcome differences without overwhelming the query surface. Favor dimensions that support recurring operational questions:

- Which workflow variant was used?
- Which feature flag was active?
- Which tenant tier experienced the failure?
- Was the cache hit or missed?
- Which retrieval strategy was selected?
- What was the policy outcome?
- How many documents were passed to the model?

These fields are high-signal because they form stable categories. You can group by them, compare them across time, and correlate them with latency or errors.

The opposite pattern is to export everything available: raw request IDs, entire prompt bodies, full retrieved documents, serialized tool payloads, and arbitrary internal variables. That feels thorough during implementation and becomes costly during operation. Large or unbounded values reduce signal-to-noise and hurt query performance. High-cardinality attributes are sometimes more tolerable in traces than in metrics, but they still increase storage cost and query complexity. Treat them as deliberate exceptions, not as defaults.

**A bad schema example** The easiest way to see the difference is to contrast a disciplined schema with a noisy one.

```
BAD_SCHEMA = {  
  "resource": {  
    "service.name": "assistant-prod-us-east-1-v7",  
  },  
}
```

```

"span": {
  "name": "call_openai_for_user_7f9a",
  "user.id": "7f9a30d8-11e2-4c7f-8b0a-8d1f5f...",
  "prompt.full": "...entire prompt body...",
  "docs.full": ["...doc1...", "...doc2..."],
  "internal.state": "{huge serialized object}",
  "tmp.debug.branch": "retry_then_parse_v2b",
},
}

```

Almost every field in this example is problematic. The resource name mixes service identity with region, environment, and release labels in one unstable string. The span name embeds a runtime-specific user context. Top-level attributes store raw UUIDs, full prompts, full documents, and a serialized internal object. A temporary debugging label becomes part of the operational schema. None of this scales well.

Now compare that with a compact, bounded version:

```

GOOD_SCHEMA = {
  "resource": {
    "service.name": "assistant-api",
    "service.namespace": "support",
    "deployment.environment.name": "prod",
  },
  "span": {
    "name": "llm.generate",
    "openinference.span.kind": "LLM",
    "app.workflow": "ticket_qa",
    "app.prompt_template": "ticket_answer_v3",
    "app.retrieval_count": 6,
    "app.cache_hit": False,
    "app.policy_outcome": "allow",
  },
}

```

The second version is much easier to search and compare. It preserves the dimensions that matter without promoting huge payloads or unstable labels to first-class query fields.

**Identity fields need discipline** Trace schemas often fail at identity because teams mix several concepts into a single field or scatter the same concept across multiple names. Keep identity layers separate:

- *Service identity*: use resource attributes such as `service.name` and `service.namespace`.
- *Environment identity*: use `deployment.environment.name`.
- *Request or run identity*: attach a stable request, conversation, or workflow run identifier where it supports debugging, but do not let it dominate the query surface.
- *Business grouping*: use bounded attributes such as workflow, feature flag, tenant tier, or route.

This separation lets you answer different classes of questions cleanly. You can filter by production versus staging without parsing project names. You can compare one workflow across services without relying on a release-specific naming convention. You can still drill into a single request when needed, but you do not force every dashboard and query to revolve around high-cardinality identifiers.

Sensitive identities deserve even more care. Raw user identifiers are poor candidates for top-level schema dimensions. If they must be present for debugging, mask them or store them in controlled metadata rather than promoting them to the main analytical vocabulary.

**Use events for milestones, not attributes for everything** A recurring schema mistake is to encode every interesting moment as a permanent attribute. That flattens a dynamic execution story into a bag of fields and often produces contradictions. If a span retries twice and then falls back to a secondary parser, what should the final attribute set say? One value may erase the sequence that actually mattered.



Events solve this more cleanly. Use them for milestones that are important to interpretation but not stable dimensions for grouping. Good event candidates include:

- `retry.started`
- `retry.exhausted`
- `fallback.selected`
- `parse.partial_failure`
- `policy.threshold_exceeded`

These moments are meaningful, but they are temporal facts about one span execution, not enduring dimensions you necessarily want in every aggregate query. If a fallback path itself becomes operationally central, give it its own span boundary. Until then, an event often carries the right amount of structure.

**Design for evaluation readiness** Trace schemas become far more useful when they are designed to support evaluation and comparison, not just one-off debugging. That does not require elaborate evaluator-specific fields everywhere. It requires consistency.

Evaluation-ready traces usually share three properties:

- stable span categories such as LLM, RETRIEVER, and TOOL;
- stable business dimensions such as workflow, prompt template, retrieval strategy, or policy outcome;
- enough request or run identity to connect traced executions to downstream quality assessment.

If those fields drift across services or releases, quality analysis becomes fragile. Saved slices stop working, historical comparisons break, and the team loses the ability to answer whether a new routing strategy or prompt revision improved anything. Schema stability is therefore not just an observability concern. It is also a prerequisite for trustworthy experimentation and evaluation.

**Keep the vocabulary controlled** As teams grow, schema drift often appears as uncontrolled synonyms. One service records `app.workflow`; another records `workflow_name`; a third records `pipeline`. This is a process problem as much as a technical one. Maintain a small controlled vocabulary for high-value dimensions and reuse it consistently.

A good controlled vocabulary favors:

- one canonical field name per concept;
- one small set of allowed values per bounded dimension;
- stable meanings that survive refactors and releases;
- explicit review before adding new top-level attributes.

This discipline prevents the schema from becoming an archaeological record of temporary experiments. It also reduces the temptation to encode multiple concepts in one string, such as combining environment, workflow, and version in a single label. Each dimension should be independently filterable.

**A practical design checklist** When deciding whether a field belongs in the schema, ask a few hard questions:

- Will you group or filter by this field repeatedly?

- Is the value bounded enough to remain queryable?
- Does it describe a stable business or operational dimension?
- Does it belong in a resource, a span name, an attribute, or an event?
- Would storing the full value create privacy, cost, or cardinality problems?
- Will the field still make sense after the next refactor or release?

If the answer is weak on several of those questions, the field probably does not belong as a first-class schema element. Capture it elsewhere or omit it.

A strong trace schema does not attempt to preserve every byte of execution context. It preserves the dimensions that make traces interpretable, comparable, and durable. Once that discipline is in place, Phoenix becomes much more than a place where spans land; it becomes a stable analytical layer for diagnosing failures, comparing workflows, and understanding how the AI system behaves as both the code and the organization evolve.

## **4.5. Applying Version-Safe Instrumentation Patterns**

Instrumentation usually breaks in quieter ways than application code. A request still completes, spans still arrive in Phoenix, and the team assumes everything is fine. Weeks later, someone notices that tool spans disappeared after a framework upgrade, a helper import moved, or a sampler example copied from an older blog post no longer reflects current guidance. The operational risk is not only outright failure. It is

semantic drift: traces keep flowing while becoming less complete, less consistent, or less comparable across releases.

That risk grows quickly in AI systems because the integration surface is wide and moving. Provider SDKs change, orchestration frameworks publish major-version breaks, OpenInference packages expand, and helper APIs evolve. If you treat instrumentation as a few lines of startup code that never need maintenance, the trace contract decays faster than the application itself.

### **Start with explicit, pinned dependencies**

The fastest way to make instrumentation brittle is to leave the dependency surface implicit. Pin the tracing stack the same way you pin your serving stack. That includes Phoenix-related packages, framework instrumentation packages, and any OpenTelemetry components that materially affect tracing behavior.

A Python dependency file should make the instrumentation surface visible rather than letting it hide behind transitive dependencies.

```
pip install \
  arize-Phoenix-otel==0.16.0 \
  openinference-instrumentation \
  openinference-instrumentation-openai \
  openinference-instrumentation-langchain
```

Two practices matter here. First, pin exact versions where helper availability or compatibility boundaries are known to matter. Phoenix documents OpenInference helper re-exports for `arize-Phoenix-otel >=0.16.0`, so examples and internal wrappers should reflect that boundary explicitly. Second, install only the instrumentors you actually need. Over-installation increases overlap risk and makes upgrades harder to reason about.

For JavaScript, treat the framework and the corresponding instrumentation package as a pair. `LangChain.js` has a verified compatibility

boundary: 0.x and 1.x require different OpenInference instrumentation packages. That is not a subtle edge case; it is exactly the kind of change that silently breaks observability if you upgrade the framework and forget the instrumentor.

```
npm install --save \
  @opentelemetry/sdk-trace-node \
  @arizeai/openinference-instrumentation-langchain
```

When you record dependencies for your service, capture both the framework version and the instrumentation package that matches it. A framework upgrade should always trigger an observability compatibility review, not just an application test run.

### **Document compatibility boundaries as code-adjacent operational facts**

Teams often know about version boundaries informally, but the knowledge lives in a chat thread or in one senior engineer's memory. That is not durable. Put compatibility notes next to the bootstrap code, in dependency manifests, or in a small instrumentation README checked into the same repository.

A concise note is enough:

```
LangChain.js 1.x uses:
  @arizeai/openinference-instrumentation-langchain

LangChain.js 0.x requires a different package.
Upgrade both together and re-verify trace structure.
```

This kind of note prevents a common failure mode: a routine framework bump keeps the application working but degrades trace coverage because the instrumentation package no longer matches the runtime surface.

The same rule applies to helper APIs. If your code relies on Phoenix OTEL helper re-exports such as session or metadata context helpers,

note the minimum package level that provides them. Do not assume that a new developer, a CI runner, or a notebook environment will resolve the same helper surface unless you pin and document it.

### Isolate bootstrap from business logic

The most maintainable instrumentation pattern is architectural rather than syntactic: isolate tracing bootstrap in one small module and keep application code dependent on your internal wrapper, not directly on every evolving package import. This reduces churn when packages, imports, or startup conventions shift.

A simple Python bootstrap module can centralize registration, project naming, and the imports you want the rest of the codebase to treat as stable.

```
# tracing_bootstrap.py
import os

from phoenix.otel import register
from phoenix.otel import using_metadata
from phoenix.otel import using_session
from phoenix.otel import using_user
from phoenix.otel import suppress_tracing

PROJECT_NAME = "support-assistant"

def init_tracing():
    os.environ.setdefault(
        "PHOENIX_PROJECT_NAME",
        PROJECT_NAME,
    )
    return register(
        project_name=PROJECT_NAME,
        auto_instrument=True,
    )
```

Application code can then import from your bootstrap module instead of binding itself directly to the external package layout.

```
from tracing_bootstrap import init_tracing
from tracing_bootstrap import using_session
from tracing_bootstrap import using_user
```

```
tracer_provider = init_tracing()
```

This pattern buys you three forms of stability. It reduces import churn, centralizes naming constants, and gives you one place to adapt if Phoenix or OpenInference packaging evolves. It also makes testing easier because you can stub or replace the bootstrap layer without touching business logic.

### **Treat semantic fields as a versioned contract**

Code compatibility is only half of the problem. Semantic compatibility matters just as much. If a release changes span names, resource labels, or important attributes, your traces may still emit but become analytically incompatible with saved searches, dashboards, and evaluation slices.

Treat these fields as a versioned contract:

- resource attributes such as `service.name`;
- stable span names such as `llm.generate` or `retriever.fetch`;
- core AI categorization through `openinference.span.kind`;
- bounded business dimensions such as `workflow`, `retrieval strategy`, `cache hit`, or `policy outcome`.

When you change any of them, document the change as you would an API change. If you rename `app.workflow` to `app.pipeline`, you have not merely refactored an internal field. You have changed the query contract used by operators, dashboards, experiments, and incident review.

This is why centralized constants matter. If your schema vocabulary lives in scattered string literals, semantic drift becomes almost impossible to control.

### **Use release validation, not just upgrade optimism**

Instrumentation upgrades should follow a repeatable validation path. A successful import or a passing unit test is not enough because the important failures are often semantic: missing child spans, changed hierarchy, dropped attributes, or duplicate traces.

A practical validation flow looks like this:

1. inspect release notes for Phoenix, the framework, and the instrumentor packages you use;
2. identify any changed package boundaries, imports, or helper availability;
3. run a small suite of representative traced scenarios;
4. inspect resulting traces in Phoenix;
5. compare span structure and attributes against your expected contract;
6. promote the upgrade only after semantic checks pass.

Your representative scenarios should cover the execution shapes that matter most operationally: a plain model call, a retrieval path, a tool-using path, an error path, and at least one flow containing manual spans. That test set is small enough to maintain and broad enough to expose most regressions that matter.

### **A migration trigger should be explicit**

Framework major-version upgrades deserve special scrutiny because they often change how instrumentors attach. A useful internal migration note should not merely say that a package changed. It should say what action the team must take.



Migration rule:

If `LangChain.js` crosses a major version boundary, review the matching `OpenInference` instrumentation package before deployment. Do not assume the old package remains compatible.

This turns a fragile memory into a stable upgrade gate. The same style of rule applies when helper APIs move, when server-side conventions evolve, or when trace conversion behavior changes in Phoenix releases. The details vary; the operational discipline does not.

### **Sampling deserves conservative, current guidance**

Sampling guidance in `OpenTelemetry` has evolved, and examples copied uncritically from older material can age badly. One concrete case is ratio-based sampling. `TraceIdRatioBased` is deprecated in favor of `ProbabilitySampler`, although it remains preserved through at least January 1, 2027. That means older examples may still work, but you should avoid presenting them as the durable long-term recommendation.

The more stable architectural lesson is to think about parent-child consistency first. When sampling consistency across a trace matters, `OpenTelemetry` recommends composition with `ParentBased` rather than using ratio-based samplers as child samplers. That advice reduces a subtle but common failure mode where one service samples a root span and another service makes a conflicting child decision.

If your stack uses sampling, document three things together:

- the sampling policy name;
- the rationale for it;
- the validation expectation for cross-service traces.

Even if you do not publish a full code sample for every sampler config-

uration, be explicit about whether a configuration is current guidance, compatibility-oriented, or legacy. That small labeling discipline prevents outdated examples from becoming accidental standards inside the organization.

### **Preserve cross-service propagation with standards-based headers**

Multi-service observability is only version-safe if trace context survives service boundaries in a standard way. The stable baseline is W3C Trace Context, specifically the `traceparent` header and optional `tracestate`. If an application, gateway, queue bridge, or worker layer strips or rewrites those headers inconsistently, you can upgrade every library correctly and still lose end-to-end continuity.

Treat propagation as part of your instrumentation contract. When you validate an upgrade, check not only local span generation but also whether traces remain continuous across HTTP or other service boundaries. The absence of shared trace context is easy to miss in development because each service still emits spans; they just no longer belong to the same causal chain.

This is another reason to keep bootstrap logic centralized. Whatever propagator configuration your environment requires should live in one place, not be recreated ad hoc across services.

### **Separate compatibility shims from application semantics**

As instrumentation stacks evolve, you may need small shims or wrappers to bridge changing APIs. That is reasonable. What you should not do is let those compatibility details leak into the business code or into the trace schema.

Keep three layers distinct:

1. **bootstrap compatibility:** package imports, registration, prop-

agators, exporter wiring;

2. **semantic contract:** span names, resource fields, and business attributes;
3. **application logic:** routing, retrieval, generation, tools, and policy code.

This separation lets you upgrade the bootstrap layer without renaming operational concepts, and it lets you refine business logic without repeatedly touching tracing infrastructure. The fewer places where those concerns are entangled, the less likely a framework or package change will ripple through the entire codebase.

### **What to pin, what to review, what to monitor**

Not every moving part deserves the same level of control. A practical policy is to pin the pieces with the highest compatibility risk and actively review the pieces with the highest semantic risk.

#### **Pin aggressively:**

- Phoenix OTEL packages;
- OpenInference instrumentors;
- framework major versions tied to instrumentation packages;
- any helper package whose imports appear in your code.

#### **Review on upgrade:**

- release notes that mention tracing, OpenTelemetry, or server-side conversion behavior;
- framework migration guides that may alter callback or execution boundaries;

- semantic convention changes that affect field names or span kinds.

#### **Monitor continuously:**

- span count changes on representative requests;
- disappearance of expected span kinds;
- drift in critical attributes;
- broken parent-child structure across services.

Monitoring for semantic regression is especially important because many instrumentation failures are not runtime errors. They are reductions in trace usefulness.

#### **Avoid common maintenance traps**

Several anti-patterns make instrumentation unnecessarily fragile.

**Copying stale snippets.** A code sample that imports correctly today may still encode an outdated compatibility assumption or deprecated sampler choice.

**Relying on transitive dependency behavior.** If tracing only works because another package happened to bring in a compatible instrumentor, your environment is already brittle.

**Mixing tracing setup with application startup logic.** This makes upgrades risky because changing instrumentation now requires touching unrelated business initialization.

**Changing semantic fields casually.** Renaming spans or attributes without a migration note breaks observability consumers even when application behavior is unchanged.

**Treating a passing smoke test as proof.** A single provider call that emits a span says very little about retrieval, tools, async paths, or cross-service propagation.

The antidote is not complexity. It is explicitness: explicit versions, explicit compatibility notes, explicit schema contracts, and explicit validation scenarios.

### **A compact operational checklist**

When you prepare any instrumentation-related upgrade, run a short checklist:

1. Pin the target package versions.
2. Check framework-to-instrumentor compatibility boundaries.
3. Review any helper API assumptions in bootstrap code.
4. Re-run representative traced scenarios.
5. Inspect trace hierarchy, span kinds, and key attributes.
6. Confirm cross-service propagation still uses traceparent correctly.
7. Record any semantic contract changes before release.

That process turns observability from a best-effort startup configuration into a maintained engineering interface. Once you treat instrumentation as a versioned contract rather than incidental glue code, Phoenix traces remain trustworthy even as the AI stack, framework ecosystem, and operational requirements keep changing.

## Chapter 5

# Debugging with Phoenix Traces: From LLM Calls to RAG Pipelines

*A bad answer from an LLM application is rarely a single failure. In Phoenix, the real task is to reconstruct the execution story: what the model saw, what the retriever supplied, what tools were invoked, where latency accumulated, and which step first diverged from intent. This chapter teaches a disciplined debugging progression, starting with model and agent traces, extending into retrieval and grounding analysis, and ending with reusable playbooks that help teams investigate recurring incidents with speed and consistency.*

## 5.1. Trace-Driven Debugging of Model and Agent Behavior

A failing LLM application rarely fails at only one layer. The user sees a bad answer, a stalled response, or a tool call that makes no sense, but the operational question is narrower and more useful: where did the execution first depart from intent? In a model-driven system, that departure may happen before the model call, inside the model call, in the handoff to a tool, in retry logic around a provider request, or in post-processing that discards a good intermediate result. Phoenix traces give you the artifact needed to answer that question without guessing.

A practical starting point is to instrument the application so that each request produces a trace with semantically meaningful spans. In Python, a minimal setup looks like this:

```
from phoenix.otel import register

tracer_provider = register(
    project_name="support-bot",
    auto_instrument=True,
)
```

For a conversational or agentic system, attach a session identifier so repeated failures can be grouped by user thread rather than treated as isolated incidents:

```
from phoenix.otel import register, using_session

register(
    project_name="support-bot",
    auto_instrument=True,
)

with using_session("session-42"):
    # Run your application logic here.
    pass
```

That setup does not solve debugging by itself. It changes the unit

of analysis. Instead of reading logs in chronological order or starting at the final prompt and response, you reconstruct a request as a tree of spans. Each span is a unit of work with timing, attributes, parent-child relationships, and a required AI operation type carried in `openinference.span.kind`. For debugging model and agent behavior, the most useful kinds are LLM, CHAIN, TOOL, AGENT, and, later in retrieval-heavy systems, RETRIEVER and RERANKER. Once those semantic boundaries are present, you can stop asking vague questions such as “why did the model hallucinate?” and start asking testable ones such as “did the planner omit the user constraint before the final LLM span?”

### **Treat the trace as the incident record**

When an application usually completes in four seconds and one request takes thirty, the bad experience is obvious but the cause is not. A single trace can separate provider latency from application retries, tool slowness, serialization bottlenecks, or a loop in the agent planner. The same logic applies to incorrect outputs. The final answer may be wrong because the prompt omitted a variable, because the model never saw the relevant tool schema, because a tool returned malformed data, or because post-processing clipped the response. Each of those leaves a different trace signature.

This is the central operating method: begin with the trace that corresponds to the failing request, not with source code guesses and not with aggregate dashboards alone. Aggregate latency and error rates tell you that a problem exists; the trace tells you how the particular execution unfolded. You use that execution history to formulate a causal hypothesis and then test it against span evidence.

### **Traverse from root to first suspicious span**

A disciplined traversal keeps you from jumping to the last visible error. Start at the top-level request span and identify the major phases of work. In a typical agent or orchestration pipeline, those phases often in-



clude request normalization, prompt construction, one or more model invocations, tool selection, tool execution, response synthesis, and final formatting. Even when framework auto-instrumentation names spans differently, the debugging method remains the same: classify each child by what operational role it plays.

The first pass through a trace should answer four questions:

- Is the trace complete enough to explain the failure?
- What are the major child spans, in execution order?
- Which span first shows input, output, or timing inconsistent with intent?
- Did downstream spans merely amplify an earlier defect?

That last question matters because the visibly bad output is often several spans removed from the first bad decision. If an agent calls the wrong tool three times, the problem may not be the repeated `TOOL` spans. The earliest suspicious span may be an `LLM` planning call that never received the user constraint, or an `AGENT` span whose state omitted a prior tool result. If a response is empty, the final `LLM` span may be fine and the defect may live in a post-processing `CHAIN` span that stripped content during validation.

### **Use span kinds to narrow the failure domain**

The required `openinference.span.kind` attribute is more than taxonomy. It constrains the search space. If the incident symptom is malformed natural-language output, inspect the `LLM` span first, but verify the `CHAIN` span that assembled its inputs. If the symptom is external side effects or wrong API calls, inspect `AGENT` and `TOOL` spans. If the symptom is runaway execution, look for repeated `AGENT`, `LLM`, or `TOOL` sequences with similar inputs and no narrowing state.

You should expect different evidence from different span kinds:

- **LLM spans** expose the model invocation boundary: messages, invocation parameters, token counts, latency, and output.
- **CHAIN spans** expose application logic that reshapes state: prompt templates, variable binding, output parsing, validation, and formatting.
- **TOOL spans** expose concrete side effects or data fetches: function arguments, return payloads, exceptions, and execution time.
- **AGENT spans** expose planning or coordination: what intermediate state existed, whether the loop advanced, and whether the model was given enough context to choose correctly.

Once you know the likely failure domain, inspect the attributes that matter to that domain rather than reading every field indiscriminately. Phoenix is most effective when the trace is semantically rich enough to make that filtering possible.

#### **5.1.1. Converting symptoms into span-level hypotheses**

A useful debugging habit is to translate each user-visible symptom into one or two falsifiable hypotheses before you inspect details. That prevents the common anti-pattern of browsing spans until something merely looks unusual.

##### **Malformed or implausible answers**

Suppose the response is fluent but wrong. The first hypothesis is not “the model is bad.” A better pair is:

- The final LLM span received incomplete or malformed inputs.

- The final LLM span received adequate inputs but produced an unsupported output.

To distinguish those, inspect the messages passed to the model, including flattened message attributes such as `llm.input_messages.<index>.message.role` and `llm.input_messages.<index>.message.content` when available. Confirm that system instructions, user constraints, and any inserted context all appear in the expected order and format. If required content is missing, the bug is upstream, usually in a `CHAIN` span that performed prompt construction or state assembly. If the prompt is intact but the answer is still unsupported, the issue moves to model behavior, parameter choice, or the need for downstream evaluation and guardrails.

Bad variable binding is especially common. You will often find a trace where the template exists, but a bound field is empty, stale, or derived from the wrong object. The final answer looks like a model failure only because the application silently degraded the prompt before the model saw it.

### **Wrong tool selection**

When an agent chooses the wrong tool, start with the `LLM` or `AGENT` span that produced the decision, not the tool execution itself. The immediate question is whether the tool schema and decision context were present at the moment of choice. If the trace shows valid `TOOL` spans but repeated misuse, inspect whether the planner was given the correct user goal, prior tool outputs, and available tool descriptions.

A strong trace pattern for tool misuse is a plausible but context-blind planner call: the model receives generic instructions and tool schemas, but not the state needed to discriminate among tools. Another pattern is state corruption between turns, where one `TOOL` span returns useful

data but a later AGENT or CHAIN span fails to carry it forward. The agent appears irrational, but the trace shows that it is operating on incomplete intermediate state.

### **Looping behavior**

Runaway loops are easier to diagnose in traces than in logs because repetition becomes structural. Look for a repeating sequence such as AGENT → LLM → TOOL → AGENT, especially where successive spans have near-identical inputs. If the loop count rises but the state does not materially change, the agent is not incorporating tool results or termination conditions. If the inputs do change slightly but move in a narrow band, the planner may be oscillating between two equivalent interpretations.

The first suspicious span is usually the first iteration where progress should have been made but was not. That may be the first TOOL result that returned an error and was not surfaced to the planner, or the first planning call that ignored a clear stop condition. Comparing iteration-to-iteration deltas is more informative than staring at the final repeated action.

### **Latency regressions**

A thirty-second response often looks like a slow model call from the outside, but traces frequently show a different decomposition: one fast planning call, three slow serial tool invocations, a retrying provider request, and a final summarization call that is normal. Phoenix lets you allocate wall-clock time to spans rather than to vague stages.

Build the latency hypothesis by asking where the extra time accumulated:

- One span became dramatically slower.
- Several nominally small spans became serialized.

- Retry logic amplified a single upstream fault.
- A loop multiplied otherwise acceptable step latencies.

Retry amplification deserves special attention. A provider-side failure that would have cost one second can turn into a thirty-second incident if wrapped in nested retries across application code, SDK behavior, and orchestration layers. In the trace, this appears as repeated sibling or nested spans with similar inputs and growing delays. If you only inspect the final exception or timeout, you miss the fact that the system was busy redoing the same work.

### **5.1.2. Evidence to inspect inside suspicious spans**

Once you isolate the likely first bad span, inspect three classes of evidence in a fixed order: inputs, outputs, and timing.

#### **Inputs**

Inputs determine what work the span was asked to do. For LLM spans, inspect prompt messages, invocation parameters, and any attached tool or function definitions. For TOOL spans, inspect parsed arguments and calling context. For CHAIN spans, inspect intermediate variables, especially if the span constructs a prompt or transforms a tool result. The key question is whether the span received enough correct state to behave as intended.

Privacy controls can limit what you see here. If production instrumentation hides sensitive inputs through OpenInference configuration, Phoenix may show redacted values such as `__REDACTED__`. That preserves structure but reduces diagnostic precision. You can still verify that a field exists, that propagation occurred, and that the right span kind executed, but some root-cause questions will require a lower environment or a carefully scoped debugging mode.

## Outputs

Outputs tell you whether the span did the wrong thing or merely exposed an upstream defect. If a planning LLM span emits a tool choice inconsistent with the visible prompt, the planner or tool descriptions are suspect. If the output is actually reasonable given the input, then the bug is earlier. For tool spans, check not only whether the tool succeeded but whether it returned data in the shape later spans expect. A tool can be semantically correct and still poison the pipeline if downstream code assumes a different schema.

## Timing

Timing helps separate correctness failures from orchestration failures. A span that returns the wrong output quickly is different from a span that repeatedly retries and finally succeeds with stale state. Duration, retries, and child-span accumulation reveal whether the incident is driven by a slow dependency, a synchronization bottleneck, or a control-flow defect. For agents, timing also exposes accidental serialization. Two tools that could have been independent may be invoked strictly one after another because the planner architecture forces serial dependence.

## Compare against a known-good trace

A single failing trace is necessary but not always sufficient. The most effective debugging move after identifying the suspicious span is to compare it with a known-good execution of the same logical path. Focus on deltas, not absolute complexity. Did a message disappear? Did tool availability change? Did latency shift from the provider span to a wrapper span? Did session state propagate in the good case but not in the bad one?

This comparison keeps the investigation grounded. Without it, engineers often overfit to incidental details in the failing trace.

## When auto-instrumentation is not enough

Framework instrumentation gives you a valuable baseline, but default traces often miss the exact state that explains a failure. A planner span may show that a model call happened without recording the compact decision state that the model should have seen. A chain span may capture timing but not the prompt variables that were bound incorrectly. When that happens repeatedly, add custom spans or attributes around the unstable boundary instead of collecting more undifferentiated logs.

The boundary for adding custom instrumentation is simple: if a class of incidents repeatedly forces you to reconstruct hidden state from code rather than inspect it directly in the trace, the semantic boundary is under-instrumented. Add enough information to test hypotheses, but do not dump every in-memory object. Over-instrumented traces are noisy in a different way: they create dozens of spans with no meaningful semantic distinctions, making the first suspicious span harder to find.

## A practical incident flow

Under production pressure, a lightweight procedure keeps the investigation repeatable:

- Capture the failing trace for the user-visible incident.
- Verify trace completeness and identify major child spans.
- Isolate the earliest span whose inputs, outputs, or timing diverge from expectation.
- Compare that span with a known-good counterpart.
- Inspect upstream state propagation before proposing a fix.
- Only after locating the first bad decision, change prompts, orchestration logic, retry policy, or tool handling.

That ordering matters. Teams often edit prompts because the final answer is bad, when the actual defect is a missing field in prompt assembly or an agent loop caused by state loss. They often blame a provider for latency when the trace shows retry multiplication or serial tool execution inside the application. A trace-driven workflow prevents the fix from drifting toward the most visible symptom.

The result is not merely better observability. You get a debugging grammar for AI systems: symptom, likely span kind, earliest suspicious boundary, evidence to inspect, comparison trace, and tested hypothesis. Once you work that way, malformed outputs, tool misuse, loops, and latency spikes stop looking like mysterious model behavior and start looking like ordinary execution failures with inspectable causes.

## **5.2. Inspecting Retrieval Operations and Context Assembly**

A RAG failure often looks like a model failure because the user only sees the final answer. The harder and more useful question is whether the model ever had the right evidence. If the retriever returned stale policy text, if reranking favored topical but non-answer-bearing chunks, or if prompt assembly dropped the decisive passage to stay within a context budget, the model may be behaving exactly as the application asked it to. Phoenix traces let you inspect retrieval as an observable subsystem rather than a black box that runs before generation.

A concrete trace is the fastest way to ground that investigation. The instrumentation setup remains the same as for general trace-driven debugging; what changes is what you look for inside the trace. In Python, you can register Phoenix tracing and run a RAG request under a project name that isolates retrieval-heavy workloads:

```
from phoenix.opel import register
```



```
tracer_provider = register(  
    project_name="rag-support-bot",  
    auto_instrument=True,  
)
```

Once the request is traced, inspect the span tree for retrieval-bearing work. OpenInference defines retrieval semantics through RETRIEVER spans and the `retrieval.documents` attribute. Each retrieved document object can carry structured fields such as `document.id`, `document.content`, `document.metadata`, and `document.score`. Those fields give you enough evidence to answer a question that logs rarely answer cleanly: what candidate evidence existed at retrieval time, and what happened to it afterward?

### **Separate retrieval from context assembly**

Treat retrieval and context assembly as distinct execution stages even when your framework collapses them into a small number of spans. Retrieval decides which chunks or documents become candidates. Context assembly decides which of those candidates are formatted, ordered, truncated, and inserted into the final prompt. Failures in these stages look similar to users but require different fixes.

Three failure classes are worth separating immediately:

- Wrong documents retrieved.
- Right documents retrieved but poorly assembled into context.
- Right context provided but ignored or overridden downstream by the model.

Without that distinction, teams tend to blame the retriever for every unsupported answer. In practice, many incidents originate after retrieval has already succeeded. A relevant chunk may appear in `retrieval.documents` but disappear during filtering, lose crucial

neighboring text during truncation, or be formatted so poorly that the model does not use it effectively. Phoenix traces are valuable because they preserve evidence across those boundaries. The trace becomes reusable forensic material rather than a one-off debugging log.

### **Start with the retrieval-bearing span**

When you inspect a failing RAG trace, begin with the earliest RETRIEVER span that materially affects the answer. Do not jump straight to the final LLM span. First confirm that retrieval occurred and that the span contains candidate evidence. Then inspect four elements in order:

- the originating query or rewritten query,
- the retrieved candidate set in `retrieval.documents`,
- document-level metadata such as identifiers, scores, and provenance,
- the relationship between retrieved candidates and the final prompt context.

That ordering matters. If the query itself is over-narrowed or pointed at the wrong corpus, later ranking and assembly details are secondary. If the candidate set already contains the decisive chunk, the diagnosis shifts away from retrieval recall and toward filtering, reranking, truncation, or prompt construction.

### **Inspect scores, but do not trust them as explanations**

A common mistake is to sort by `document.score`, observe that the top-ranked chunk looks plausible, and stop there. Scores can tell you how the retrieval system ranked candidates; they do not tell you whether the highest-ranked candidate actually answers the user's question. In RAG systems, semantically related but task-irrelevant chunks often score

well because they share vocabulary or topic, not because they contain the decisive fact.

Suppose a user asks, “What is the refund window for annual enterprise plans?” and the retriever returns these top candidates:

```
1. score=0.93 id=doc-41
   "Enterprise plans include dedicated support..."
2. score=0.91 id=doc-18
   "Annual billing is available for business customers..."
3. score=0.86 id=doc-77
   "Refunds are available within 30 days for annual
   enterprise plans..."
```

The highest-scoring chunk is topically related but does not answer the question. The decisive evidence is present, but it ranks third. If your context assembly logic keeps only the top two chunks, retrieval technically succeeded while answer generation still fails. The right fix is not necessarily a new embedding model. You may need better reranking, less aggressive top- $k$  filtering, or assembly rules that preserve answer-bearing chunks over generic descriptive text.

### **Use provenance to detect stale or contaminated retrieval**

Document identifiers and metadata are not bookkeeping details. They often reveal the actual root cause. If a retrieved chunk lacks `document.id`, you lose the ability to compare retrieved evidence against the source corpus or later verify whether the same stale passage keeps appearing across incidents. If `document.metadata` includes tenant, timestamp, product line, or source collection, you can detect cross-tenant contamination, stale indexes, and corpus-routing mistakes quickly.

A retrieval result that looks semantically relevant may still be operationally wrong. For example, a stale document with strong lexical overlap can outrank the current policy document if indexing or filtering lags behind source updates. In the trace, the content appears reasonable

until you inspect metadata and notice an outdated effective date or the wrong tenant label. That is one reason to record provenance-rich retrieval attributes even when content capture is redacted or sampled.

### **Query generation and rewriting can fail before retrieval does**

Many RAG pipelines rewrite user questions before retrieval. That can help recall, but it can also erase constraints. If the original question is “Can student accounts export invoices as CSV in the EU region?” and a rewrite collapses it to “invoice export CSV,” retrieval may return broadly relevant billing documents while losing the regional and account-type constraints that actually determine the answer.

You should therefore inspect the query artifact that reached the retriever, not only the original user message. If the rewritten query is narrower, broader, or differently scoped than the user’s intent, that is often the earliest suspicious span. A broad query tends to produce noisy candidates and force later filtering decisions. An over-narrow query may exclude the only useful document family entirely. Both can produce the same visible symptom: a plausible answer supported by the wrong context.

### **Candidate sets and selected sets are different evidence**

Many teams inspect only the final prompt context. That is useful but incomplete. You need to compare the candidate set returned by retrieval with the selected set passed into prompt assembly. A good diagnostic routine asks:

- Was the decisive chunk present anywhere in `retrieval.documents`?
- If present, where did it rank?
- Was it discarded by reranking or filtering?
- Was it selected but truncated into uselessness?

- Was it preserved in the prompt but placed so late or with so little context that generation ignored it?

This comparison is the boundary between retrieval debugging and prompt debugging. If the decisive evidence never entered the candidate set, you have a retrieval recall problem. If it entered but was later excluded, you have a ranking or assembly problem. If it survived into the final context and the answer still contradicts it, the problem shifts downstream toward generation, synthesis, or evaluation.

**Figure 5.1:** *Evidence loss often occurs before the final model call. Tracing each transformation stage turns a vague “bad answer” into a specific retrieval or assembly diagnosis.*

### 5.2.1. Reading retrieval failures as trace patterns

The most productive RAG debugging sessions rely on recurring patterns rather than one-off intuition. Phoenix traces make these patterns visible because retrieval decisions and prompt construction appear as inspectable artifacts.

#### Wrong documents retrieved

This is the cleanest failure class. The candidate set in `retrieval.documents` simply does not contain the evidence needed to answer the question. Common causes include a misrouted corpus, poor chunking that splits the decisive fact away from its context, stale indexes, and overly aggressive query rewriting. In traces, you will often see high-scoring documents that are topically similar yet operationally irrelevant. The tell is absence: the needed identifier, source family, or answer-bearing text never appears in the retrieved set.

When this happens, inspect whether the retrieved documents share misleading lexical overlap. Stale policy pages, release notes for the wrong product tier, or reference documents from another tenant frequently look convincing until you inspect `document.metadata`. If your traces do not carry metadata rich enough to distinguish those cases, add it. Retrieval without provenance is much harder to debug than retrieval with modest but precise source descriptors.

### **Right documents retrieved, wrong documents assembled**

This failure class is more subtle and more common than many teams expect. The retriever returns the decisive chunk, but reranking, thresholding, duplicate suppression, or context budgeting removes it before the model sees it. A trace may show the relevant `document.id` in `retrieval.documents`, followed by a final prompt context that contains only broader background chunks.

Two mechanisms cause this repeatedly.

First, reranking can optimize for topical similarity rather than answer-bearing specificity. A chunk explaining the product family may outrank a terse but decisive policy statement because the former overlaps with more query terms. Second, token-budget logic can punish exactly the kind of chunk you need. If the context assembler preserves long explanatory passages and trims short, high-value excerpts late in the pipeline, the answer degrades while retrieval metrics still look acceptable.

When you diagnose this class, compare the content and order of retrieved candidates with the exact text inserted into the final prompt. If the decisive chunk was shortened, inspect whether truncation removed a qualifier such as a date, exception clause, or regional condition. Losing ten words at the end of a chunk can turn a correct answer into a wrong one.

**Right context provided, model still misses it**

Sometimes the trace shows that the final prompt included the needed evidence, but the answer still contradicts or ignores it. That is not a retrieval failure, and treating it as one leads to wasted tuning. The evidence path was intact; the problem is now in generation or synthesis.

You can still use retrieval traces to establish this cleanly. Confirm that the decisive chunk survived all earlier stages, that it remained readable in the final prompt, and that its source was not drowned in irrelevant surrounding context. If those conditions hold, move the investigation to prompt framing, model behavior, or answer evaluation. The value of retrieval tracing here is negative proof: it lets you stop blaming the retriever once the retriever has done its job.

**Prompt-readable context matters**

When retrieved content is used later for evaluation, especially faithfulness-style analysis, format the context as readable text with clear separators between documents. Raw JSON dumps are poor diagnostic artifacts and poor evaluator inputs. They make it harder for humans to inspect evidence flow and harder for evaluators to reason over the actual supporting passages. Even outside formal evaluation, readable context reduces debugging friction. You can compare what the retriever returned against what the generator received without mentally decoding nested structures.

A compact assembly pattern works well in practice: preserve document identifiers, include a short metadata header when relevant, and separate documents with explicit delimiters. That keeps provenance visible while making the context intelligible to both the model and the engineer inspecting the trace.

### **5.2.2. Operational trade-offs in retrieval observability**

Retrieval debugging benefits from more metadata, more candidate visibility, and more prompt transparency, but each of those adds cost or governance pressure.

#### **Recall versus token budget**

A broader candidate set increases the chance that the decisive chunk appears somewhere in `retrieval.documents`, but it also increases the burden on reranking and prompt assembly. If your traces show that useful chunks often rank below a hard cutoff, expand the candidate set before generation or improve reranking. If traces show that candidate recall is already strong but final context is noisy, do not increase retrieval breadth further. The trace evidence tells you where additional tokens help and where they only add distraction.

#### **Rich metadata versus privacy and storage**

Document content is diagnostically powerful, but metadata often provides the higher leverage per byte stored. Tenant, source collection, document version, timestamp, and chunk offsets can explain many failures without storing every passage in full. Where privacy constraints force redaction, preserve enough structural metadata to reconstruct provenance and selection decisions. A trace with redacted content but intact `document.id`, `document.metadata`, and rank ordering is still far more useful than a trace that only says “retrieval happened.”

#### **Auto-instrumentation versus custom boundaries**

Some framework integrations emit useful retrieval spans automatically, while others capture only coarse model invocations. If your traces do not expose candidate-level retrieval data, reranking outputs, or the final assembled context, you may need to add custom spans or attributes around those boundaries. Add instrumentation where



decisions are made, not at arbitrary call sites. A single well-placed span around context assembly is usually more useful than five low-level spans around string concatenation.

### **A practical debugging loop**

When a RAG answer looks wrong, run the investigation in a fixed order:

- Inspect the user question and any rewritten retrieval query.
- Inspect the `RETRIEVER` span and `retrieval.documents`.
- Check `document.id`, `document.score`, and `document.metadata` for provenance, rank, staleness, and tenant correctness.
- Compare the retrieved candidate set with the selected context actually inserted into the prompt.
- Determine whether evidence was absent, dropped, truncated, or preserved but later ignored.
- Only then change retrieval settings, reranking, context assembly, or generation logic.

That order prevents a common failure of diagnosis: tweaking prompts to compensate for a retrieval system that never surfaces the right evidence, or rebuilding retrieval when the real bug is a context assembler that silently discards the decisive chunk. Once you read `RETRIEVER` spans and final prompt context as linked evidence rather than isolated artifacts, retrieval stops being an opaque prelude to generation and becomes a debuggable causal chain.

## 5.3. Diagnosing Grounding, Relevance, and Hallucination Failures

Teams often label any bad answer as a hallucination because that label is fast and emotionally satisfying. It is also operationally weak. A wrong answer can come from missing evidence, irrelevant evidence, context that was present but ignored, prompt framing that encouraged overconfident synthesis, or a genuine model limitation even when the trace is otherwise healthy. If you collapse all of those into one diagnosis, you will change the wrong subsystem. Phoenix traces give you the execution record; evaluators give you structured judgment about what that record produced. Used together, they let you localize quality failures instead of merely naming them.

A concrete evaluator workflow makes that distinction easier to internalize. Install the Phoenix evaluation package and set up a judge model:

```
pip install "arize-phoenix-evals>=3"
```

```
from phoenix.evals import LLM, evaluate_dataframe
from phoenix.evals.metrics import (
    DocumentRelevanceEvaluator,
    FaithfulnessEvaluator,
)

judge = LLM(
    provider="openai",
    model="gpt-4o-mini",
)
```

Now create a small dataframe that mirrors what you already see in traces: user input, retrieved context, and model output. Keep the context readable. Concatenate document passages into plain text with clear separators rather than dumping raw JSON. Faithfulness depends on whether the output is grounded in the supplied context, not whether the answer happens to be true in the world. If your context field is unreadable or mapped incorrectly, the evaluator can only produce noisy

judgments.

```
import pandas as pd

df = pd.DataFrame(
    [
        {
            "input": "What is the refund window for annual plans?",
            "context": (
                "[doc-77]\n"
                "Refunds are available within 30 days "
                "for annual plans.\n\n"
                "[doc-41]\n"
                "Annual plans include priority support."
            ),
            "output": (
                "Annual plans can be refunded within "
                "30 days of purchase."
            ),
        },
        {
            "input": "What is the refund window for annual plans?",
            "context": (
                "[doc-77]\n"
                "Refunds are available within 30 days "
                "for annual plans.\n\n"
                "[doc-41]\n"
                "Annual plans include priority support."
            ),
            "output": (
                "Annual plans can be refunded within "
                "30 days and enterprise plans have a "
                "60-day exception."
            ),
        },
    ]
)
```

You can run faithfulness and document relevance separately:

```
faithfulness = FaithfulnessEvaluator(judge)
doc_relevance = DocumentRelevanceEvaluator(judge)

results = evaluate_dataframe(
    dataframe=df,
    evaluators=[faithfulness, doc_relevance],
)
```

This simple setup already exposes a key debugging principle: relevance and faithfulness are different questions. A retrieved document can be relevant to the user's query while the answer still adds unsupported details. Conversely, retrieval can be weak or partially irrelevant while the model still produces a correct-looking answer from prior knowledge. Phoenix traces tell you what evidence was available and how it moved through the pipeline. Evaluators help you judge what the answer did with that evidence.

### **Define the failure in terms of evidence state**

You get a stronger diagnosis if you classify answer failures by the state of available evidence before using any evaluator score. That keeps the trace as the primary artifact and uses evaluation to sharpen, not replace, reasoning.

A practical evidence-state taxonomy has five categories:

- No relevant evidence was retrieved.
- Relevant evidence was retrieved but omitted from final context.
- Relevant evidence was present in context but ignored in generation.
- Evidence was present but conflicting, partial, or ambiguous.
- Evidence was adequate, but the answer failed during synthesis, framing, or post-processing.

These categories are operationally different. The first two point toward retrieval or context assembly. The third points toward grounding failure during generation. The fourth often requires both trace inspection and human review because the answer may not be strictly unsupported; it may be an overconfident resolution of ambiguity. The fifth points to-

ward prompt design, reasoning quality, or output shaping rather than evidence acquisition.

If you do not classify evidence state first, evaluator outputs can tempt you into overly broad conclusions. A low faithfulness result may suggest unsupported claims, but it does not tell you whether the unsupported claim arose because the evidence was missing, because the context field was malformed, or because the model embellished a well-supported answer.

### **Use document relevance to isolate retrieval quality**

`DocumentRelevanceEvaluator` helps answer a narrow question: did the retrieved context actually relate to the user's question? That is useful because retrieval quality often fails before generation quality can even be assessed. If the answer is wrong and the supporting documents are irrelevant, there is little value in debating model grounding behavior until retrieval is fixed.

In practice, document relevance becomes especially useful when traces show many plausible candidates and you need to separate topical overlap from answer-bearing evidence. A policy overview can be relevant in a loose sense while still failing to contain the one clause that answers the question. Relevance evaluation therefore works best alongside trace inspection of individual retrieved passages, not as a replacement for it.

A useful pattern is to align the evaluator with the same context artifact you inspected in the trace. If the retriever span exposes multiple documents and your final prompt context includes only a subset, evaluate the subset that actually reached the model. That helps distinguish retrieval-stage relevance from assembly-stage selection. When document relevance looks good but the answer still fails, move the investigation to grounding and synthesis rather than broad retrieval tuning.

### **Use faithfulness to test grounding, not world truth**

Faithfulness is often misunderstood because teams expect it to answer “is this true?” The actual question is narrower and more actionable: is the output grounded in the supplied context? That distinction matters most in RAG systems.

Consider two answers to the same query. The first says, “Annual plans can be refunded within 30 days of purchase.” The second says, “Annual plans can be refunded within 30 days and enterprise plans have a 60-day exception.” If the supplied context only mentions the 30-day rule, the first answer is faithful and the second is not, even if the 60-day exception happens to be true elsewhere in the product documentation or in the model’s training data. From a debugging standpoint, the second answer is unsupported by the evidence path for that request.

This makes faithfulness especially useful after you have already confirmed that retrieval and context assembly were functioning. A low faithfulness result in that situation points to model behavior: the system had evidence, but generation stepped beyond it. A low faithfulness result with poor retrieval evidence is less informative, because the model may have been forced to answer without enough support.

### **Field mapping errors can create false diagnoses**

Evaluator pipelines are only as good as their input mapping. Faithfulness requires input, output, and context. If you bind the wrong field into context, or if you pass a machine-oriented structure that obscures the supporting text, you can create false negatives or false positives.

Raw JSON is a common source of trouble. Suppose your trace contains retrieved documents as nested objects with metadata, offsets, and scores, and you pass that raw serialized blob into the evaluator. A human can sometimes decode it with patience; a judge model often performs worse because the actual evidence is buried in syntax noise. The

remedy is simple: transform the context into readable text with explicit separators, preserve document identifiers where useful, and evaluate the same assembled context the model actually received.

You should also watch for silent schema drift. If an upstream transformation renames the answer column, swaps user input with rewritten query text, or drops the final prompt context in favor of a larger candidate set, evaluation results may degrade for reasons unrelated to model quality. When evaluator output looks inconsistent with what you saw in the trace, inspect the dataframe mapping before you change prompts or models.

### **A bad answer is not always a hallucination**

The term hallucination has some value as shorthand for unsupported content, but it becomes damaging when it suppresses fault isolation. Several neighboring failure classes are easy to confuse:

- **Missing evidence:** the answer is unsupported because the evidence path never contained the needed material.
- **Weak grounding:** relevant evidence exists, but the model adds claims that are not justified by it.
- **Synthesis failure:** the answer draws from evidence but combines it incorrectly, drops qualifiers, or overcompresses.
- **Prompt framing error:** the prompt encourages decisiveness, speculation, or answer shape that exceeds the evidence.
- **Model limitation:** the evidence is present and clear, the prompt is reasonable, and the model still fails to interpret or compose the answer correctly.

These are not merely academic distinctions. They map to different owners and different fixes. Missing evidence belongs with retrieval

or indexing. Weak grounding often belongs with prompting, model choice, or guardrails. Synthesis failures may need better output instructions or decomposition. Prompt framing errors sit in orchestration. Genuine model limitations may justify escalation to a more capable model or a narrower task design.

### **Trace evidence should lead the evaluator, not follow it**

A robust debugging loop starts with the trace, uses evaluators to sharpen the diagnosis, and then returns to the trace for confirmation. Reverse that order and you risk chasing the evaluator's failure mode rather than the application's failure mode.

Suppose a query returns a factually correct answer that fails faithfulness. That is not a contradiction. It means the retrieved context did not support the answer, even if the model knew the fact from elsewhere. In a closed-book or grounded-answer setting, this is exactly the failure you care about. The trace will usually confirm one of two stories: either retrieval failed to supply the decisive evidence, or generation ignored the instruction to stay within evidence. Without the trace, you cannot tell which.

Similarly, suppose document relevance scores are low but the answer is correct. That can happen when the model relies on prior knowledge or memorized patterns. If your product requirement is grounded assistance rather than open-world answering, you should still treat that as a problem. The trace-and-evaluator combination helps you distinguish lucky correctness from reliable evidence-backed behavior.

### **Compare answer claims against source passages**

Scalar evaluator outputs are useful for triage, but they are not sufficient for root-cause analysis. Once an answer is flagged, compare its claims directly with the retrieved passages and final prompt context. Look for three recurring mismatches:



- the answer introduces entities, dates, thresholds, or exceptions absent from the context;
- the answer collapses a conditional statement into an unconditional one;
- the answer resolves ambiguity without acknowledging conflict in the sources.

These mismatch patterns often reveal where the problem belongs. Introduced entities and unsupported exceptions usually indicate weak grounding. Lost conditions often indicate synthesis compression or truncation. Over-resolved ambiguity may reflect prompt framing that rewards certainty over evidence fidelity.

This is also where lightweight human review remains valuable. Evaluators can surface suspicious rows efficiently, but nuanced evidence conflicts and domain-specific wording still benefit from an engineer or analyst reading the trace-linked context directly.

### **Span-level scope matters**

Phoenix evaluations are most useful when you evaluate the output at the right scope. Faithfulness is especially suited to span-level LLM outputs in RAG pipelines because the question is whether a particular generated answer is grounded in the specific context that reached that model invocation. If you evaluate only a final user-visible response after multiple transformations, you may hide where unsupported content was introduced. A post-processing span may paraphrase a faithful intermediate answer into an unfaithful final one, or vice versa.

Where your tracing model exposes intermediate LLM spans, treat those as evaluation candidates. That lets you detect whether the first unsupported claim appears during retrieval-aware synthesis, later summarization, or a tool-augmented rewrite. The closer the evaluation scope is to the decision boundary, the more actionable the result becomes.

### **A practical diagnostic flow**

When a user reports a bad answer, use a fixed sequence:

- Inspect the trace and classify the evidence state.
- Verify what retrieval returned and what context the model actually received.
- Compare the answer against the supplied context, not against external world knowledge.
- Run document relevance when retrieval quality is in doubt.
- Run faithfulness when you need to test whether the output is grounded in the supplied context.
- Check evaluator field mapping if results conflict with obvious trace evidence.
- Assign the likely owner: retrieval, prompt construction, generation, post-processing, or model selection.

That sequence keeps the diagnosis honest. You do not excuse unsupported claims simply because the answer sounds plausible, and you do not blame the model for evidence it never received. Over time, this approach changes how you talk about failures. “Hallucination” stops being a catch-all accusation and becomes one possible outcome of a narrower reasoning process grounded in traces, evidence state, and evaluator judgment.

## **5.4. Building Reusable Debugging Playbooks**

A team usually realizes it needs playbooks after the third or fourth incident that feels oddly familiar. One engineer remembers that a similar

failure last month turned out to be a missing retriever span. Another remembers that the answer looked ungrounded only because prompt inputs were redacted in production. A third knows that the real issue was session context not propagating across turns. If that knowledge lives only in memory, debugging quality varies with who is on call. A reusable playbook converts those scattered lessons into a stable inspection path.

The useful unit of standardization is not a vague checklist like “inspect the trace.” You need a playbook that encodes a symptom, the likely span kinds, the mandatory evidence to collect, the branch points that separate neighboring failure classes, and the conditions that justify escalation. Phoenix is well suited to this because the trace is already structured around spans, sessions, and attributes. The playbook becomes a repeatable query over that structure.

A small amount of trace stamping early in the request path makes later playbooks much stronger. In Python, OpenInference context managers let you attach stable debugging metadata that Phoenix can later filter on:

```
from phoenix.otel import (
    register,
    using_session,
    using_metadata,
    using_tag,
    using_attributes,
)

register(
    project_name="support-bot",
    auto_instrument=True,
)

with using_session("session-42"):
    with using_tag("incident-class:bad-answer"):
        with using_metadata({
            "env": "prod",
            "release": "2026-03-15",
        }):
            with using_attributes({
```

```
"deployment.region": "us-east-1",
}):
    pass
```

This is not decorative metadata. It gives you consistent pivots for incident correlation, environment scoping, and reproduction tracking. If a bad-answer incident clusters around one release or one region, you can see that quickly. If a session-level failure spans many traces, `session.id` lets you group them rather than treating each request as independent.

### **Define what a playbook must contain**

A useful debugging playbook needs a fixed schema. If each team writes prose notes in a different style, the result is still ad hoc. At minimum, capture these fields:

- incident symptom visible to users or alerts,
- expected span graph and required span kinds,
- mandatory evidence fields to inspect,
- branch points for likely failure classes,
- comparison trace criteria,
- evaluator checks, if applicable,
- escalation thresholds,
- environment and version context,
- privacy and redaction settings.

The version and privacy fields deserve explicit treatment. Phoenix and OpenInference release frequently enough that import boundaries and emitted span shapes can drift. A playbook should record the

package versions involved in a reproduced investigation, especially when behavior depends on which package emitted which span attributes. Privacy controls are equally important. Settings such as `OPENINFERENCE_HIDE_INPUTS` and `OPENINFERENCE_HIDE_LLM_TOOLS` directly change what evidence is available during incident response. If a prompt body or tool schema is missing, you need to know whether that absence is an instrumentation defect or an expected result of redaction policy.

### **Start every recurring incident with stable checkpoints**

The strongest playbooks encode stable checkpoints that apply across many incident classes. Those checkpoints prevent teams from skipping foundational checks and jumping too early to root-cause theories. A compact baseline works well:

- Verify trace completeness.
- Verify expected span kinds are present.
- Verify session grouping when the symptom may span turns.
- Verify redaction settings for the environment.
- Inspect the earliest suspicious span, not only the final bad output.
- Run trace-linked evaluators if the failure concerns answer quality.

These checkpoints are intentionally general. They create consistency without forcing every incident through the same narrow path. Trace completeness alone catches a surprising number of failures in investigation quality. If the retriever span is missing, if tool execution is untraced, or if a parent-child relationship is broken by context propagation issues, the playbook should say so immediately rather than allowing the investigator to build conclusions from partial evidence.

### 5.4.1. A playbook template for bad final answers

The most common incident class is the bad final answer: plausible wording, incorrect substance. This is where the earlier diagnostic methods become operational. Instead of asking a senior engineer to reconstruct the method each time, write it down as a branching workflow.

#### Template structure

A practical template for “bad final answer” might include:

- Confirm the request trace is complete.
- Confirm expected top-level span kinds: CHAIN, LLM, and, where applicable, RETRIEVER or TOOL.
- Identify the first suspicious span by comparing the failing trace against a known-good trace of the same request class.
- Branch into one of three diagnostic paths:
  - retrieval evidence missing or weak,
  - evidence present but answer ungrounded,
  - answer grounded but malformed or incomplete.

You can encode that template in a runbook document or ticket form. The point is that every investigator follows the same decision boundary. A retrieval branch requires inspection of RETRIEVER spans, `retrieval.documents`, provenance metadata, and final context assembly. A grounding branch requires answer-versus-context comparison and usually evaluator runs. A malformed-but-grounded branch points toward prompt framing, output parsing, or post-processing.

## Tie the branch to evaluators

Best practice is to pair each recurring incident class with one or more evaluators so the playbook can mature from manual diagnosis into measurable regression testing. For bad-answer incidents in RAG systems, two evaluators are especially useful: document relevance and faithfulness. Start with manual trace inspection to classify the failure, then run the evaluator on the same artifacts.

```
from phoenix.evals import LLM, evaluate_dataframe
from phoenix.evals.metrics import (
    DocumentRelevanceEvaluator,
    FaithfulnessEvaluator,
)

judge = LLM(
    provider="openai",
    model="gpt-4o-mini",
)

evaluators = [
    DocumentRelevanceEvaluator(judge),
    FaithfulnessEvaluator(judge),
]

results = evaluate_dataframe(
    dataframe=dataframe,
    evaluators=evaluators,
)
```

When the team repeatedly confirms that one failure mode corresponds to a stable evaluator pattern, promote that pattern into a regression check. That does not replace trace inspection; it lets you detect recurrences earlier and at larger scale.

## Record the output of the investigation, not only the fix

A reusable playbook should capture what evidence proved the diagnosis. For a bad-answer incident, record the suspicious span ID, the exact branch taken, whether relevance or faithfulness was low, whether the context was redacted, and which remediation fixed the issue. Future

investigators benefit more from “relevant document was present in `retrieval.documents` but dropped during assembly” than from “fixed prompt.” The former is reusable evidence; the latter is a shallow conclusion.

### 5.4.2. Session-level playbooks for conversational failures

Single-trace playbooks are not enough for conversational systems. Many agent failures only make sense when you group requests by session and inspect recurrence across turns. Forgetting prior context, repeated clarification loops, and gradual token or latency growth are all session-shaped failures.

#### Use session IDs deliberately

If you do not propagate a stable session identifier, conversation-level debugging becomes a manual search problem. With session stamping in place, a playbook for “agent forgot prior turn context” can start from the session rather than the final answer. The first checks should ask:

- Did all related traces carry the same `session.id`?
- Did the failure begin after a particular turn or tool result?
- Did prompt size, latency, or tool count change sharply before the forgetting behavior appeared?
- Did the state passed into the relevant LLM or AGENT span omit earlier entities or constraints?

This kind of playbook often benefits from lightweight session metrics such as number of turns, average latency by turn, and repeated-tool patterns. You do not need elaborate analytics to make this valuable. Even a consistent manual review order reduces false diagnoses. Teams



frequently blame memory or model limitations when the real issue is a broken propagation path or a session reset that happened mid-conversation.

### **Add annotations to resolved sessions**

Phoenix supports grouping traces into sessions and adding annotations. Use that capability to mark resolved incident sessions with a short diagnostic label and remediation owner. Over time, those annotated sessions become a library of exemplars: known-good, known-bad, and known-fixed. A playbook is stronger when it can point investigators to two or three representative sessions rather than only abstract instructions.

### **Escalate when recurrence crosses a threshold**

Session-level playbooks should include explicit escalation conditions. For example, if the same forgetting pattern appears in three sessions after a release, escalate from ad hoc debugging to a release-blocking defect. If the issue is isolated to a single tenant or region, route it differently. A playbook without escalation criteria helps with local consistency but does not improve operational response.

### **5.4.3. Make privacy and version context first-class evidence**

Many failed investigations are caused not by missing skill but by missing context about the observability surface itself. A playbook should therefore treat instrumentation boundaries, privacy controls, and package versions as part of the incident record.

### **Record redaction settings**

If your production environment uses `OPENINFERENCE_HIDE_INPUTS` or `OPENINFERENCE_HIDE_LLM_TOOLS`, the playbook must state that up front. Otherwise, an investigator may waste time searching for prompt

bodies or tool schemas that were intentionally removed. A compact incident template can capture this directly:

```
Incident metadata
- environment: prod
- package versions:
  arize-phoenix-otel=...
  arize-phoenix-evals=...
- redaction:
  OPENINFERENCE_HIDE_INPUTS=true
  OPENINFERENCE_HIDE_LLM_TOOLS=false
- trace completeness: pass/fail
- session linked: yes/no
```

This block is not executable code; it is a documentation pattern. Keep it near the top of the playbook record so every investigator sees the evidence constraints before interpreting absent fields.

### **Record import and version boundaries**

Because Phoenix and OpenInference evolve quickly, a playbook should note which imports and package versions were active during reproduction. This matters whenever trace structure, auto-instrumentation behavior, or available context managers differ by package line. If a recurring incident only reproduces under one deployment image or one pinned environment, you need that fact in the record. Without it, teams can spend days comparing traces that were emitted under different semantics.

### **Add instrumentation only when the playbook proves a gap**

Not every hard investigation justifies more spans. A good playbook helps you tell the difference between a one-off ambiguous case and a persistent instrumentation blind spot. If multiple incidents fail at the same decision boundary because the trace omits the key state, add custom instrumentation there. If the trace was adequate and the difficulty came from evaluator ambiguity or domain complexity, stronger human review guidance may be enough.

This distinction keeps traces readable. Playbooks should reduce investigation cost without encouraging instrumentation sprawl.

### **Convert repeated diagnoses into tests**

A mature debugging culture does not stop at documented runbooks. Once a playbook reliably identifies a recurring failure mode, bind it to an evaluator workflow and a stored dataset slice. For example, if you repeatedly diagnose unsupported embellishments in a RAG answer path, preserve the representative prompts, contexts, and outputs, then use document relevance plus faithfulness as an automated regression check during release validation. Manual playbooks and automated checks should reinforce each other. The playbook explains the reasoning path; the regression test catches recurrences before users do.

### **Keep the playbook opinionated but not brittle**

A playbook should narrow the search space, not eliminate judgment. If it is too loose, every investigator improvises and the team learns nothing cumulative. If it is too rigid, it fails whenever the framework shape changes or a new provider introduces a different trace pattern. The right balance is to standardize checkpoints, evidence collection, and escalation while leaving room for expert branching when the trace shows something genuinely novel.

That balance is easiest to maintain when every playbook names its stable checkpoints explicitly: trace completeness, span-kind expectations, retrieval evidence inspection, grounding checks, tool-use checks, and session-level recurrence analysis. Those checkpoints hold across frameworks, provider changes, and many product iterations. What changes is the local interpretation and the exact remediation. When your team can open an incident, stamp it with session and environment metadata, follow a known inspection path, attach evaluator results, and archive the resolved trace as an exemplar, debugging stops being a craft practiced by a few individuals and becomes an operational

capability the whole group can rely on.



## Chapter 6

# Evaluations, Datasets, and Experiments as a Quality System

*A traced failure tells you what happened; an evaluation system tells you whether it was acceptable, why it failed, and whether a proposed fix actually improves the system. This chapter builds that progression deliberately, moving from Phoenix's evaluation primitives to evaluator design choices, then to the operational boundary between live trace scoring and controlled dataset experiments, and finally to regression-oriented datasets and comparative experiments that support responsible iteration.*

## 6.1. Evaluation Primitives in Phoenix

You can inspect a stack of failed runs, recognize that something is off, and still have no durable answer to the engineering question that matters: was that a one-off anomaly, or a repeatable defect class you can measure, monitor, and prevent? Raw traces are excellent at telling you what happened during one execution. They do not, by themselves, tell you whether that execution was acceptable against a defined quality bar. Phoenix draws a hard line between those two concerns. Observation lives in traces and spans. Judgment lives in evaluations.

A useful way to start is with a concrete scoring pass over structured records. Phoenix supports running evaluations on arbitrary datasets, including data in a dataframe, using built-in evaluators or custom ones. The mechanics are straightforward:

```
from phoenix.evals import LLM, evaluate_dataframe
from phoenix.evals.metrics import (
    DocumentRelevanceEvaluator,
)

# Configure the judge model according to your environment.
llm = LLM()

# Your dataframe would typically contain columns such as:
# input, output, reference, and retrieved_document.
relevance = DocumentRelevanceEvaluator(llm=llm)

# Evaluate rows and return structured results.
results = evaluate_dataframe(
    dataframe=df,
    evaluators=[relevance],
)
```

That snippet is intentionally modest. It does not solve evaluator selection, scoring policy, or governance. It does establish the central primitive: an evaluator consumes explicit inputs and produces structured output that can be attached to a concrete unit of work. That unit might be a dataset row, an experiment result, a trace, or a span derived from

a live execution. Once you make the judgment explicit and persistent, you can aggregate it, slice it, compare it, and treat it as part of the system rather than as a note in an incident document.

### **Observation versus judgment**

Phoenix is built around a distinction that teams often blur during their first deployment. Telemetry describes execution. A trace records a root operation and the spans underneath it. Those spans capture model calls, retrieval steps, tool invocations, latency, branching behavior, and attributes emitted by your instrumentation. None of that is a quality judgment. A retrieval span with ten returned documents is an observation. A model span with a 7 s latency is an observation. A tool call carrying malformed JSON is an observation.

Judgment begins when you evaluate those observations against a criterion. “Were the retrieved documents relevant to the question?” is a judgment. “Did the answer faithfully use the retrieved context?” is a judgment. “Did the tool response conform to the required schema?” is a judgment. The criterion can be deterministic, model-based, or human-applied; the important part is that it is bounded, named, and attached to a specific entity.

That separation matters operationally because the same trace can support several independent judgments. A run might be slow but correct. It might be fast but hallucinated. It might produce a good final answer while taking an invalid branch through a tool call that happened not to affect the result. If you collapse all of that into a single informal label such as “bad run,” you destroy the information needed for debugging and later regression testing.

### **What Phoenix evaluates**

The entity being judged determines both the evaluator design and the meaning of the result. Phoenix supports evaluations on production



traces, experiment results, and arbitrary datasets. In practice, you should think in terms of several common targets.

- **Trace-level evaluations** attach a judgment to the whole observed execution. Use them when the criterion is inherently end-to-end: overall answer correctness, user task success, refusal appropriateness, or aggregate policy compliance. A trace-level label answers a broad question, but it usually cannot localize the failure mechanism on its own.
- **Span-level evaluations** attach a judgment to a sub-operation inside the trace. This is where Phoenix becomes especially valuable for LLM systems. Many defects that users perceive as “the model answered badly” actually originate in a narrower component: retrieval returned irrelevant documents, reranking surfaced the wrong passages, context assembly truncated a needed snippet, a tool was selected incorrectly, or a tool response was mishandled. Evaluating the relevant span lets you score closer to the true failure boundary.
- **Record- or row-level evaluations** apply to structured offline data. A dataset row may represent a prompt-reference pair, a retrieval example, a conversation turn, or an experiment input/output pair. This is the natural unit for benchmark-style evaluation and controlled comparisons.
- **Output-level evaluations** judge the generated artifact directly: a final answer string, a JSON object, a citation set, or a tool call payload. These often coexist with trace-linked context, because the output by itself rarely explains why it was good or bad.
- **Human annotations** are also evaluation primitives when they are attached to a specific execution or record with explicit semantics. A reviewer note that says “citation missing” without a stable

label or rubric has weak downstream value. The same review, captured as a categorical annotation with optional rationale and provenance, becomes something you can filter, compare, and later promote into a dataset.

### **What an evaluation contains**

A useful evaluation payload carries more than a single number. Phoenix’s evaluation model supports structured signals such as scores, labels, and explanatory information. You should treat these as separate fields with distinct semantics rather than interchangeable ways of saying “good” or “bad.”

**Categorical labels** are best when the outcome space is discrete and operationally meaningful: `relevant/irrelevant`, `tool_selected_correctly/incorrectly`, `policy_violation/clean`. Labels are easy to group and slice, and they age better than free-form reviewer comments.

**Scalar scores** are useful when the judgment admits gradation. A faithfulness score or relevance score can express degrees of quality that matter for thresholding or trend analysis. Scores need a declared scale. A 0--1 confidence-style metric should never be mixed casually with a 1--5 rubric score in the same chart.

**Pass/fail outcomes** are the simplest primitive and often the strongest when a hard boundary matters. Schema validity, exact match, regex conformance, or presence of mandatory fields all naturally collapse to binary results.

**Rationales and explanations** preserve why the evaluator reached the result. For LLM judges, a rationale is often essential for debugging evaluator behavior and for adjudicating disagreement with humans. For code checks, the explanation may simply be the failed rule or offending field.

**Confidence indicators** can help with triage, especially for semantic judges, but they are easy to misuse. Treat evaluator confidence as metadata about the judging process, not as proof that the score is correct.

**Provenance metadata** tells you who or what produced the judgment: a code evaluator, an LLM-based evaluator, or a human labeler; which prompt version was used; which model acted as judge; and when the score was recorded. Without provenance, your metrics become hard to audit and harder to compare over time.

### **Why trace-linked evaluation changes the debugging loop**

Detached spreadsheets and benchmark CSVs can tell you that quality is low. They usually cannot tell you why. Trace-linked evaluation closes that gap by preserving the execution context around the judgment. If a faithfulness evaluator marks an answer as unsupported, you can inspect the exact prompt, retrieved documents, token usage, intermediate tool calls, and branch-specific behavior associated with that score. That is the difference between a metric and an actionable defect.

This is particularly important for retrieval-augmented systems. Suppose the final answer contains a factual error. A trace-level “incorrect” label tells you that the user-visible result failed. A span-level evaluation on the retrieval step may reveal that the retrieved documents were irrelevant. Another span-level evaluation on the answer generation step may show that the model ignored the one correct document it did receive. Those are different engineering problems with different fixes. One pushes you toward retrieval tuning or indexing changes; the other points toward prompt design, context formatting, or model selection.

OpenTelemetry’s span model makes this decomposition practical. A trace contains a root span and nested spans for sub-operations. As long as identity propagates across your system boundaries, Phoenix can associate judgments with the right execution context rather than forcing everything into one opaque run-level score.

### Three signal origins

Phoenix treats evaluation signals as coming from three broad sources: llm, code, and human. That vocabulary is worth adopting early because it keeps the measurement system explicit.

A code evaluator expresses a deterministic rule. If the output must match a pattern, contain a required field, or satisfy a business constraint, code is usually the strongest instrument. It is cheap, reproducible, and easy to audit.

An LLM evaluator expresses a semantic judgment where deterministic rules are too narrow. Relevance, faithfulness, answer quality, and tool-use quality often fit this category. Phoenix provides built-in evaluators and APIs for binding evaluators into workflows. In Python, the documented identifiers include `bind_evaluator`, `create_evaluator`, `create_classifier`, `evaluate_dataframe`, and `async_evaluate_dataframe`. The details of when to choose each signal origin belong to evaluator design, but the primitive itself is already clear: each origin still produces a structured judgment attached to a concrete target.

A human evaluation supplies the highest-context form of judgment and often serves as calibration or arbitration. The key is to capture it with the same discipline as automated results. Free-text review comments are useful during incident response; they are weak as long-term quality signals unless they are normalized into stable labels, scores, or adjudicated outcomes.

### A minimal custom evaluator pattern

When the built-ins do not express the rubric you need, you can bind your own evaluator. The exact scoring rubric matters more than the mechanics, but the mechanics should still be explicit:

```
from phoenix.evals import bind_evaluator
```

```
def required_citation_check(output: str):
    passed = "[1]" in output or "[source]" in output
    return {
        "label": "pass" if passed else "fail",
        "score": 1.0 if passed else 0.0,
        "explanation": (
            "Found citation marker"
            if passed else
            "Missing required citation marker"
        ),
    }

citation_evaluator = bind_evaluator(
    required_citation_check
)
```

This example stays deliberately simple. The return structure shown here is illustrative of the kind of bounded payload you want: explicit label, explicit score, explicit explanation. The important design lesson is that a custom evaluator should emit stable fields with clear semantics. If you instead return ad hoc text such as “seems okay” or vary the label set over time, you make later aggregation and filtering unreliable.

### **The operational flow**

The practical workflow is repeatable.

First, inspect the execution and identify the quality question. Avoid vague prompts such as “Was this run good?” Ask a bounded question: “Were the retrieved documents relevant to the user query?” or “Did the tool invocation include all required arguments?”

Second, choose the correct entity to judge. If the problem is localized to retrieval, score the retrieval span. If the question is end-to-end success, score the trace. If you are comparing prompt variants offline, score dataset rows or experiment outputs.

Third, define the output semantics before you run the evaluator at scale. Decide whether the result is binary, categorical, scalar, or some combination. Declare the range, the meaning of each label, and any threshold

that turns a score into a gate.

Fourth, persist provenance with the result. Record whether the evaluator was code-, LLM-, or human-based, and preserve the evaluator version or prompt tag when available. Phoenix’s evaluator prompt management and versioning are especially useful here because judge prompts evolve, and you need to know which version measured which runs.

Fifth, aggregate only after the semantics are stable. Once many traces or rows carry the same kind of structured judgment, you can compute rates, segment by workflow path, correlate with latency or retrieval behavior, and identify recurring failure classes.

### **Anti-patterns that poison the quality signal**

Several mistakes appear early and then become expensive to unwind.

- **Scoring without a rubric.** If different engineers use the same label to mean different things, your metric becomes an argument disguised as data. Write the rubric down, even for internal evaluators.
- **Mixing incompatible scales.** A binary pass rate, a 0--1 relevance score, and a 1--5 human quality rating cannot be averaged into a single “quality” number without careful normalization and a good reason.
- **Attaching broad labels to narrow reviews.** If a reviewer only checked retrieval relevance, do not record an overall trace failure. Label the retrieval span. Broad labels create false certainty about the locus of failure.
- **Dropping explanations and provenance.** A score without its producing evaluator, model, prompt version, or rationale is difficult to audit and often impossible to compare later.

- **Using labels that cannot be grouped.** Reviewer-specific text such as “kind of wrong” and “mostly okay” may be meaningful in the moment but is nearly useless for dashboards, dataset curation, or experiment slicing.

### Version sensitivity and compatibility discipline

Phoenix has evolved quickly, and evaluation workflows can differ across release lineages. That matters most when you encode evaluator entry points, prompt-managed judges, dataset attachment workflows, or UI-based annotation practices into team procedures. Keep your code examples pinned to documented APIs such as `pip install "arize-phoenix-evals>=3"`, from `phoenix.evals` import `LLM`, `evaluate_dataframe`, and from `phoenix.evals` import `bind_evaluator`. When a capability is available but the exact API shape is not stable across versions, describe the behavior in prose and preserve the version context in your internal runbooks.

The same discipline applies to human annotation workflows. If your team relies on UI-driven annotation or conversion from traces into datasets, verify the current behavior in the deployed Phoenix version before treating it as a stable process guarantee. Evaluation only works as a system primitive when engineers trust that the meaning and storage of a score remain consistent.

Once you start treating judgments as structured, trace-linked artifacts rather than informal impressions, every later quality workflow becomes more rigorous. A bad run is no longer just an anecdote. It is a trace or span with explicit labels, scores, rationales, and provenance that can be queried, curated, and replayed under controlled conditions. That shift is what turns observability data into a usable quality substrate.

## 6.2. Choosing Between LLM Judges, Code-Based Checks, and Human Review

Once you start attaching structured judgments to traces, spans, and outputs, a harder question appears immediately: which kind of judge should produce those judgments? Teams often fail here not because they skipped evaluation, but because they chose the wrong instrument for the failure mode. A tool-call argument mismatch judged by an LLM is usually wasteful. A nuanced answer-faithfulness decision reduced to exact string matching is usually misleading. A policy-sensitive edge case delegated entirely to automation is often reckless. Phoenix supports all three signal origins—code, llm, and human—because no single evaluator class is reliable across every quality dimension.

A compact example makes the boundary concrete. Suppose your application calls a weather tool and then writes a natural-language answer to the user. You might enforce the tool contract with deterministic checks and score the answer quality with a semantic judge:

```
from phoenix.evals import LLM, evaluate_dataframe
from phoenix.evals.metrics import MatchesRegex

llm = LLM()

tool_name_check = MatchesRegex(
    pattern=r'"tool_name"\s*:\s*"get_weather"'
)

args_check = MatchesRegex(
    pattern=r'"city"\s*:\s*"["]+"'
)

results = evaluate_dataframe(
    dataframe=df,
    evaluators=[tool_name_check, args_check],
)
```

That example covers only the deterministic part of the problem. It can verify that the tool call contains the required shape or fields, but it



cannot tell you whether the final answer was helpful, whether it over-claimed certainty, or whether it grounded its claims in tool output. For those questions you need a different evaluator class. This is the central design principle: choose the evaluator by the semantics of the judgment, not by convenience or habit.

### **Use code when the requirement is objective**

Code-based checks are most reliable when the requirement is objective, deterministic, and machine-checkable. That is not a narrow corner case; many high-value production checks fall into this category. Schema validation, exact argument presence, citation-marker presence, output field extraction, regex conformance, forbidden pattern detection, latency thresholds, token-budget limits, and business-rule compliance all benefit from code evaluators.

Phoenix officially supports code evaluators such as `exact_match` and `MatchesRegex(pattern=...)`. Their value is not subtlety. Their value is reproducibility. Given the same input, they return the same result. They are cheap to run, easy to parallelize, straightforward to audit, and well suited to hard gates in CI or release review.

The most important design question for a code evaluator is whether the rule actually expresses the requirement. If your real concern is “does this answer correctly explain the policy exception,” a regex for the phrase “policy” is a poor substitute. If your real concern is “did the tool call include the `account_id` field and format the date correctly,” code is ideal. Deterministic evaluators fail when teams use them to approximate semantics they cannot directly measure.

### **Code checks as invariants, not as proxies for meaning**

A productive mental model is to treat code evaluators as invariants around the system. They enforce properties that must always hold, regardless of model behavior. This is especially powerful in tool-using

and structured-output applications.

For a tool call, you can check:

- the selected tool name matches an allowed set,
- required arguments are present,
- argument formats match expected patterns,
- mutually exclusive fields are not emitted together,
- output JSON parses and satisfies a schema,
- tool usage stays within policy or cost limits.

For a text answer, you can still use code where the requirement is crisp:

- exact match on a canonical label,
- citation token or URL presence,
- prohibited phrases or content classes,
- maximum response length,
- mandatory disclaimer presence.

These checks are narrow by design. That narrowness is a strength when you need auditability. It becomes a weakness when teams mistake a passing invariant for overall quality. An answer can satisfy every formatting rule and still be unhelpful or false.

### **Use LLM judges when the question is semantic**

LLM judges are appropriate when the quality dimension is semantic and cannot be reduced cleanly to deterministic rules.

Correctness without an exact reference, faithfulness to retrieved context, conciseness, tool-use quality, answer relevance, explanation quality, or nuanced response quality all fit this category. Phoenix supports LLM-based evaluators, and the documented Python APIs include `LLM`, `evaluate_dataframe`, `async_evaluate_dataframe`, `create_evaluator`, `create_classifier`, and `bind_evaluator`.

A practical pattern is to bind a rubric-driven evaluator rather than ask for an unconstrained impression. Research on G-EVAL supports rubric-based and structured prompting as a better route to human alignment than free-form scoring. The operational lesson is simple: if you want an LLM judge to behave consistently, specify the criterion, define the scale, and constrain the output format.

For example, a faithfulness evaluator should not merely ask, “Is this a good answer?” It should ask whether the answer is supported by the provided context, instruct the judge to ignore irrelevant style preferences, and return a bounded label or score with a rationale. Structure reduces variance and makes disagreements easier to interpret.

### **Why LLM judges drift**

LLM judges offer broad semantic coverage, but they introduce measurement risk. Bias, prompt sensitivity, and domain mismatch remain real problems. A judge may prefer verbose answers, penalize terse but correct ones, overvalue polished tone, or fail on domain-heavy factual distinctions it does not actually understand. If the prompt wording changes, the scores may shift. If the underlying judge model changes, the threshold for “good enough” may shift again.

That creates a measurement hazard familiar from experiments: if the instrument changes while you compare variants, the comparison is no longer clean. Version-pin the evaluator prompt and the judge model whenever you intend to compare runs over time. Phoenix’s prompt-managed evaluator workflows and versioned prompts are op-

rationally important here because they preserve the measurement artifact rather than treating the judge as an invisible implementation detail.

The right way to think about LLM judges is not “cheap human replacement.” Think of them as scalable semantic triage with known failure modes. They can screen large volumes, highlight likely defects, and provide consistent first-pass coverage, but they still need calibration.

### **Calibrate LLM judges before trusting them**

A small human-labeled calibration set is often enough to reveal whether a proposed LLM judge is usable. Start with a deliberately mixed sample: easy passes, obvious failures, borderline cases, and examples from high-risk slices. Have humans label them with an explicit rubric. Run the LLM evaluator on the same set. Then inspect agreement and disagreement patterns.

You do not need a perfect judge to gain value. You need to know where it is trustworthy and where it is brittle. A judge that aligns well on routine relevance checks but fails on domain-specific compliance questions can still be useful if you restrict its scope. The anti-pattern is deploying an LLM judge globally because it appears plausible on a few hand-picked examples.

A second calibration pass should test sensitivity to evaluator changes. If a minor prompt edit flips many scores, the evaluator is unstable. If swapping the judge model shifts labels materially, comparisons across time require stricter version control. Stable evaluator design matters as much as application design when you use evaluation to make shipping decisions.

### **Use humans where ambiguity or risk dominates**

Human review remains the strongest evaluator for high-stakes, ambiguous, policy-sensitive, or calibration-target cases. It is also indis-

pensable when you need domain expertise that neither deterministic checks nor generic LLM judges can reliably supply. Legal interpretation, safety adjudication, medical nuance, and adversarial edge cases typically fall here.

Human review is expensive, so the design problem is not whether to use it, but where to spend it. Use people where the cost of a wrong automated judgment is high or where the automated evaluator itself needs supervision. That usually means three roles.

**Adjudication:** humans resolve disagreements among evaluators, or between an evaluator and stakeholder expectations.

**Calibration:** humans create the reference set you use to estimate whether an LLM judge is aligned enough for production use.

**Escalation:** humans review a targeted slice of traces or outputs selected by risk, ambiguity, novelty, or disagreement concentration.

A human review process also needs explicit agreement and disagreement handling. If two reviewers disagree, define whether you use majority vote, expert override, or adjudication by a senior reviewer. Without a disagreement policy, “human-labeled data” often hides inconsistent standards.

### **Pairwise preference versus absolute scoring**

Many teams default to absolute scores because they seem more scientific. Often the real decision is comparative: which variant is better? Pairwise preference evaluation is often preferable when your deployment question is relative rather than absolute. If two prompts both produce acceptable answers but one is consistently clearer and more grounded, a pairwise judge can be more sensitive than an absolute rubric with a wide middle band.

This matters especially in prompt and model iteration. Absolute labels

such as `pass/fail` are ideal for hard constraints. Pairwise preference is stronger when you are ranking candidate outputs from two variants on the same input. It reduces some of the ambiguity of assigning an isolated score and better matches the engineering decision you actually need to make. The trade-off is that pairwise methods answer a narrower question. They tell you which output is better, not necessarily whether either one is acceptable for release.

### **A practical decision matrix**

You can reduce evaluator selection to a few operational questions.

**Is the requirement machine-checkable?** If yes, start with code. Deterministic checks are the default for objective rules.

**Does the judgment depend on meaning rather than form?** If yes, consider an LLM judge, ideally with a rubric and structured output.

**Would a wrong evaluator decision create material business, safety, or governance risk?** If yes, reserve human review for that slice, even if automation handles lower-risk traffic.

**Do you need reproducibility across long time horizons?** Favor code where possible, and version-pin LLM evaluators when not.

**Is throughput the dominant constraint?** Code is cheapest and fastest, LLM judges are slower and costlier, humans are slowest and most expensive.

**Is nuance the dominant constraint?** Humans and well-designed LLM judges outperform code on open-ended semantic quality.

That trade-off can be expressed compactly:

| Criterion                | Code       | LLM judge       | Human review |
|--------------------------|------------|-----------------|--------------|
| <b>Cost per judgment</b> | Low        | Medium          | High         |
| <b>Latency</b>           | Low        | Medium          | High         |
| <b>Semantic coverage</b> | Low        | Medium/High     | High         |
| <b>Variance</b>          | Low        | Medium/High     | Medium       |
| <b>Auditability</b>      | High       | Medium          | Medium/High  |
| <b>Calibration need</b>  | Low        | High            | Medium       |
| <b>Best use</b>          | Invariants | Semantic triage | Adjudication |

The table is not a scoring formula. It is a control aid. If you find yourself choosing human review for a regex problem or using regex to judge nuanced factual grounding, the mismatch should be obvious.

### Layer the evaluators instead of choosing one

Advanced teams rarely deploy only one evaluator class. They compose them. A layered system is usually the most reliable pattern: code gates for crisp failures, LLM judges for scalable semantic coverage, and humans for adjudication and calibration.

For a tool-using assistant, the layers often look like this:

- Run code checks on tool selection, argument presence, schema, and policy invariants.
- Run an LLM judge on final-answer helpfulness, faithfulness, and tool-response handling quality.
- Escalate harmful, ambiguous, or evaluator-disagreement cases to human review.

This layering is not just about quality; it is also about cost control. Deterministic checks absorb the easy, high-confidence failures cheaply. LLM judges focus on the semantic middle where automation still helps. Humans spend time only where their judgment adds unique value.

### Failure modes in evaluator design

Several evaluator mistakes appear repeatedly.

**Using one evaluator for every dimension.** A single LLM judge that emits one overall score is convenient but often useless for debugging. Split quality dimensions when the fixes differ.

**Replacing hard constraints with soft judges.** If a field must exist, test for the field. Do not ask an LLM whether the output “seems complete.”

**Trusting an uncalibrated LLM judge.** A plausible prompt is not evidence of reliability. Check alignment against human labels on a small calibration set.

**Changing evaluator prompts mid-comparison.** If the evaluator itself changes prompt or model, comparisons become invalid unless version-pinned.

**Overusing human review.** Sending all traffic to human reviewers wastes scarce expertise and slows iteration. Triage first.

**Underusing human review.** If no human ever inspects high-risk or disagreement-heavy slices, automation drifts unchecked.

### **Operational guidance in Phoenix**

Phoenix’s unified treatment of evaluations makes composition practical. You can attach deterministic and semantic results to the same traces, spans, dataset rows, or experiment outputs, then compare where they agree and where they diverge. That is often the fastest way to identify a weak evaluator. If code checks pass but the semantic judge fails, the system may satisfy formal constraints while still disappointing users. If the LLM judge passes but humans consistently reject the result, your judge rubric or model may be misaligned.

Keep evaluator provenance with every score. Record whether the result came from `code`, `llm`, or `human`; preserve the relevant prompt or model version for LLM judges; and keep human review policy stable



enough that agreement statistics remain interpretable. Once these signals accumulate on traces and datasets, you can promote recurring failures into regression examples and run controlled comparisons without changing the measuring instrument underneath the experiment.

Good evaluator design is mostly about respecting the boundary between what can be checked deterministically, what can be judged semantically at scale, and what still requires accountable human interpretation. When that boundary is explicit, Phoenix stops being just a place to store scores and becomes a disciplined measurement system for real engineering decisions.

### **6.3. Deciding Between Online Trace-Linked Evals and Offline Dataset Evals**

A common failure in mature AI teams is not the absence of metrics, but the misuse of the wrong metric for the wrong decision. You collect live evaluation scores on production traces, see a drop in faithfulness, and compare that number directly to a benchmark result from a curated dataset run last month. Both are “evaluation results,” but they answer different questions under different assumptions. Phoenix supports both modes explicitly: evaluations can run on production traces, experiment results, or arbitrary datasets. The practical challenge is to keep those modes separate long enough that each remains interpretable, then connect them deliberately in a quality workflow.

A simple Python setup for experiment work makes the offline side concrete. Phoenix’s documented experiment tooling starts with the client package:

```
from phoenix.client import Client

client = Client()
```

That initialization is not the whole workflow, but it anchors the mental model. Offline work begins from a stable collection of examples and proceeds through a controlled sequence such as dataset → task → evaluator → run experiment, with support for adding more evaluations afterward via `evaluate_experiment`. Online work begins somewhere else entirely: an observed execution arrives with its trace context, and you attach judgments to what actually happened in live traffic.

These two modes serve complementary purposes. Online trace-linked evaluation answers, “What is failing in today’s traffic, under today’s traffic mix, infrastructure behavior, retrieval corpus, and provider conditions?” Offline dataset evaluation answers, “Did prompt B beat prompt A on the same examples, judged by the same evaluator?” Confusing those questions leads directly to bad release decisions.

**What makes an evaluation online** An online or nearline evaluation is attached to observed executions from production, staging, or another live system. The defining property is not timing alone. The defining property is that the evaluated entity comes from real runtime behavior. The trace contains prompts, retrieved context, tool calls, latency, branching, metadata, and any intermediate steps your instrumentation emits. That makes online evals rich in context and strong for diagnosis.

The strength of online evals is ecological validity. You are not approximating usage; you are watching actual usage. If retrieval relevance drops after an index update, a trace-linked evaluator can surface the problem quickly. If a tool provider starts returning malformed payloads, span-level evaluations can show where the execution degrades. If one user cohort triggers a failure mode that your curated benchmark never included, online signals reveal it.

This realism comes with statistical instability. Live traffic shifts.

Prompt distributions change by hour and by feature rollout. User intent changes with seasonality, marketing campaigns, or enterprise onboarding. Retrieval corpora drift as documents are added or deleted. External tools fail intermittently. Provider behavior changes outside your release cycle. Any aggregate online score therefore mixes product quality with traffic composition and operational state.

That is why online scores are excellent for monitoring and discovery but weak as clean baselines for fair comparison.

**Trace context is the main reason online evals matter** Online trace-linked evals are best for production realism, failure discovery, and context-rich debugging because traces contain intermediate steps and metadata. A low answer-correctness score on its own may only tell you that the user-visible output failed. The surrounding trace can show whether the problem began in retrieval, truncation, tool misuse, prompt branching, or a timeout-induced fallback path.

This context is especially important when your system is not a single model call. A retrieval-augmented assistant, an agent with tools, or a multi-step orchestration graph often fails far away from the final output. When you attach an evaluation directly to a span in the live trace, you score closer to the actual fault boundary. That makes online evals the natural source for discovering new defect classes.

A useful operational pattern is to inspect a poor live trace, identify the failing span, attach or run a targeted evaluator against that span, and then search for similar failures across traffic. That process converts anecdotal debugging into measurable defect discovery.

**Sampling changes what your online metrics mean** Open-Telemetry sampling introduces an important trade-off: lower cost and lower overhead versus potentially incomplete online evaluation cover-

age. If you retain only a subset of traces, then the failures you observe come from that subset, not from the full stream of executions. That may be acceptable, but you need to interpret the resulting metrics honestly.

If sampling is unbiased and stable, sampled online evaluations can still reveal directional changes and recurring classes of defects. If sampling is uneven across endpoints, user cohorts, latency bands, or error conditions, the observed failure rate can diverge substantially from the true failure rate. In other words, the score is not wrong; the population it describes is narrower than the full system.

This matters when teams treat online dashboards as release gates. A rising pass rate on sampled traces can mean the system improved, but it can also mean that traffic changed or that the sampled set no longer covers the hardest cases. Keep the distinction between “observed in retained traces” and “true across all traffic” explicit in your internal reporting.

**What makes an evaluation offline** Offline dataset evals are best for controlled comparisons because the same examples can be replayed across variants. The defining property is control. The inputs are fixed. The rubric is fixed. The evaluator suite is fixed, or at least version-pinned. You can run prompt A and prompt B against the same dataset, then compare like with like.

That control is why offline evaluation is the right place for comparative engineering decisions: prompt revisions, model swaps, retrieval-policy changes, reranking strategies, orchestration refactors, or tool-selection logic updates. If your question is whether a change helped, you need the traffic distribution held steady long enough to isolate the effect of the change.

The experiment structure formalizes this. In Phoenix, the official work-

flow is dataset → task → evaluator → run experiment, with the ability to add more evaluations later via `evaluate_experiment`. That sequencing matters because it separates the candidate system behavior from the measuring instrument. You choose the examples, run the variants, then apply a stable evaluator suite.

Offline evaluation therefore answers a narrower but more trustworthy question: under controlled conditions, how did these variants differ?

**Offline control is powerful, but it can go stale** A curated dataset is never the same thing as production. Even a well-designed benchmark reflects a deliberate sample, not the full mess of live traffic. That is a strength for comparability, but it creates a stale-benchmark risk. A system can improve on the dataset while degrading in the field if the dataset underrepresents fresh user intents, new document types, changing tool behaviors, or emerging edge cases.

This is where teams often slip into benchmark theater. They optimize what the dataset measures, celebrate score gains, and then miss a new failure class already visible in live traces. The fix is not to distrust offline evals; it is to keep them anchored to reality by feeding them with representative and failure-rich examples mined from online behavior.

Industry convention around experimentation follows the same logic: an experiment is a versioned result over a fixed dataset, useful for side-by-side comparison. That is exactly why experiments are strong decision tools and weak production mirrors.

**Choose the mode by the decision you need to make** You can usually decide between online and offline evaluation by asking four questions.

- Are you trying to monitor current behavior or compare candidate

variants? Use online evals for monitoring current behavior. Use offline evals for comparing variants fairly.

- Do you need immediacy or repeatability? Online evals give immediacy. Offline evals give repeatability.
- Is traffic diversity the signal or the confounder? For failure discovery, traffic diversity is valuable because it surfaces long-tail conditions. For A/B-style comparison, traffic diversity is a confounder unless tightly controlled.
- Are you making a release decision or a debugging decision? Debugging usually starts online because the live trace contains the causally relevant context. Release gating usually depends on offline experiments because they provide a stable basis for comparison.

These questions create a clean control boundary. Online evaluation asks, “What is happening now?” Offline evaluation asks, “What happens if we change the system under matched conditions?”

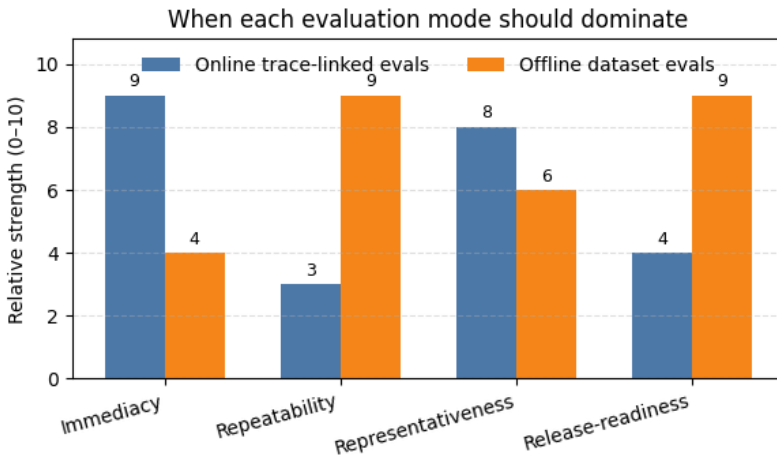


**Figure 6.1:** *Relative strengths of online trace-linked evaluation and offline dataset evaluation across common engineering decision criteria.*

**A practical loop from live failure to controlled test** A mature quality system does not choose one mode and ignore the other. It uses each mode where it is strongest. A practical loop often begins with a bad live trace. An online evaluator flags low faithfulness or poor tool handling. You inspect the trace, confirm the defect, and determine whether it represents a recurring class rather than an isolated anomaly.

If it does, promote that case into a dataset example. Preserve the relevant inputs, outputs, retrieval context, expected behavior, and any labels or rationales needed for later judging. Then run offline experiments on that curated example, plus other related cases, to test whether a proposed fix actually works under controlled conditions. If the candidate variant passes the offline threshold, roll it out and return to online monitoring to confirm that the real-world failure rate actually drops.

This is the cleanest way to connect discovery and validation. Online evals discover and localize. Offline evals validate and compare. Together they create a feedback loop that is both realistic and experimentally disciplined.

**Release gates should not rely on one mode alone** A robust release gate usually combines exploratory online monitoring with an offline experiment threshold. If you rely only on online metrics, you risk confounding system quality with traffic mix shifts, partial trace retention, and temporary provider behavior. If you rely only on offline metrics, you risk shipping a change that wins on the benchmark but fails under live conditions your dataset missed.

A practical release pattern looks like this:

- Monitor live traces with targeted online evaluators to surface current failure classes.
- Curate representative and high-severity failures into a maintained dataset.
- Run offline experiments on the same dataset across the old and new variants.
- Require the candidate to satisfy predefined offline quality thresholds before rollout.
- After rollout, verify impact with online trace-linked monitoring on the relevant traffic slices.

This approach respects the strengths of both modes without treating either as universal truth.



**Anti-patterns that collapse the boundary** Several mistakes repeatedly erase the distinction between online and offline evaluation.

- Comparing online scores directly to benchmark scores. The populations differ, the traffic conditions differ, and often the evaluator context differs. The numbers may both be meaningful, but they are not directly interchangeable.
- Treating sampled online traces as a statistically complete view. If trace retention is partial, online metrics describe retained traces, not necessarily total traffic.
- Using offline datasets as if they were production mirrors. A curated dataset is a controlled test asset, not a faithful reproduction of all live behavior.
- Tuning directly to a static eval set. If teams optimize prompts and logic specifically to satisfy a frozen benchmark, the system can become benchmark-savvy while growing brittle in production.
- Shipping based only on live anecdotes. Seeing fewer bad traces after a small change does not establish causal improvement unless the comparison conditions are controlled.
- Ignoring evaluator drift. If the evaluator prompt or judge model changes between runs, even offline comparisons can become unstable.

**Version discipline matters in both modes** Evaluator stability matters for online and offline workflows, but the reason differs. In online monitoring, you want temporal continuity: a score trend should reflect system behavior more than evaluator changes. In offline comparison, you want experimental fairness: prompt A and prompt B should be judged by the same evaluator version. If you alter the judge while altering the system under test, you no longer know what changed.

Keep evaluator provenance attached to both trace-linked and dataset-linked results. Preserve the evaluator type, prompt or model version for LLM judges, and the policy used for any human adjudication. The same discipline that keeps experiments trustworthy also keeps production dashboards interpretable.

**A concise rule of thumb** When you need to know what is breaking in real traffic, evaluate the trace you actually observed. When you need to know whether one variant is better than another, replay the same dataset under the same evaluator suite. If a live issue matters enough to fix, it usually matters enough to promote into an offline regression asset. That boundary keeps production monitoring honest, experiments fair, and improvement work tied to the failures users actually experience.

### 6.4. Curating Regression Datasets from Real Failures and Critical Scenarios

A regression dataset becomes valuable only when it protects you against the kinds of breakage your system actually produces. Many teams collect examples that are easy to export, easy to label, or easy to score, then wonder why their benchmark improves while production incidents continue. The problem is rarely the evaluation tooling. The problem is dataset curation without a protection goal. A release-protection dataset should preserve failure modes, retain hard cases, and stay anchored to the traffic and workflows that matter most.

Phoenix is especially strong here because it closes the loop between live failures and controlled testing. Phoenix supports adding spans or groups of spans from traced applications directly into datasets from

the UI, enabling a direct failure-to-regression-example workflow. That matters operationally. When you inspect a hallucinated answer, a bad retrieval step, or a malformed tool interaction in a trace, you do not need to retype the case into a spreadsheet and hope the details survive. You can promote the observed execution into a reusable test asset.

Dataset maintenance also needs stable identity. Phoenix dataset workflows support `example_id_key` for stable IDs and diff-based updates, which helps keep regression suites clean over time rather than turning them into append-only dumps of near-duplicates. Stable IDs are what let you say, with confidence, that this item represents the same historical failure across multiple dataset versions.

**Start with the failure you want to prevent** Before you collect rows, define what the dataset is supposed to guard against. A useful regression set is not a random sample of usage and not a vanity benchmark assembled from clean happy-path prompts. It is a deliberately shaped collection of examples chosen to reveal regressions in the behaviors you care about.

In practice, that means naming failure classes explicitly. Examples include retrieval misses, hallucinations under missing context, prompt brittleness for paraphrased requests, unsafe formatting, domain-specific reasoning errors, long-context degradation, tool misuse, truncation side effects, and policy violations. If you cannot state the failure class, you will not know whether the dataset covers it or whether a new example adds protection rather than noise.

This is also where critical scenarios enter. Real production failures should anchor the dataset because they best reflect traffic, integration context, and user language. Alongside them, keep business-important, safety-relevant, and historically brittle scenarios even if they are rare. A case that occurs only a few times a month may still deserve perma-

ment residence in the regression set if its consequences are severe or if it captures a known structural weakness.

**Lead with a concrete curation pattern** A practical regression suite often includes several slices with distinct jobs. Organize them intentionally rather than blending everything into one undifferentiated pool:

- *Real failures*: incidents mined from traces and support escalations.
- *Golden happy paths*: representative cases that must keep working after refactors.
- *Edge cases*: adversarial phrasing, long inputs, odd formatting, sparse context, and rare entity combinations.
- *Policy/safety*: refusal boundaries, escalation triggers, and prohibited-content handling.
- *Critical customer journeys*: workflows tied to revenue, compliance, or key user commitments.

That organization helps you resist a common failure mode: easy, high-volume cases drowning out severe but infrequent defects. A dataset is not a poll. It is a test instrument. If a slice matters disproportionately to your business or risk profile, the dataset should reflect that.

**Preserve the right fields when promoting a trace** When you turn a trace or span into a regression example, preserve enough context to make the example interpretable later. A retrieval-related hallucination is a good example. Suppose a user asks a benefits question, retrieval surfaces irrelevant policy fragments, and the assistant

answers confidently with a fabricated exception. If you save only the user prompt and the bad answer, you lose the causal context that made the failure informative.

Instead, keep the example with the retrieved context and the expected grounded behavior. The expected result does not always need to be a full gold answer. It may be enough to encode a bounded expectation such as “must refuse unsupported claim,” “must cite retrieved policy language,” or “must answer only from supplied documents.” That is often more robust than forcing a single canonical reference string onto an open-ended task.

A well-preserved example should usually include:

- the user input or task prompt,
- the system or application context that materially shaped the response,
- retrieved documents or tool outputs when they are relevant,
- the observed output,
- expected behavior, reference output, or evaluation rubric,
- failure mode label,
- source trace or incident provenance.

Regression datasets should preserve provenance, such as source trace or incident, failure mode label, and expected behavior or adjudication status. That provenance is not bookkeeping overhead. It is what lets you decide later whether a case should be retired, re-labeled, or moved into a different protection slice.

**Metadata is part of dataset quality** Treat metadata fields as first-class design elements. Without them, you will struggle to segment experiment results or rebalance the dataset over time. A useful row schema usually includes operational metadata beyond the task content itself.

Typical fields include:

- severity,
- scenario\_family,
- source\_trace\_id,
- incident\_id,
- slice,
- has\_human\_reference,
- expected\_behavior\_type,
- adjudication\_status,
- synthetic.

These fields enable later filtering such as “show only severe retrieval failures with human-reviewed references” or “compare variants on critical customer journeys only.” If you omit the metadata now, you will rebuild it later under pressure when an experiment regresses on a high-value slice and nobody can isolate the cause.

**Stable IDs keep the suite maintainable** As the suite grows, stable identity becomes non-negotiable. Phoenix dataset maintenance supports `example_id_key` for stable IDs and diff-based updates. Use

that mechanism early. If you rely on positional row order or append-only ingestion, you will accumulate duplicates, lose lineage, and make it difficult to answer simple questions such as whether a specific historical failure is still in the benchmark.

A stable ID should represent the conceptual example, not a transient export artifact. If a row's wording changes because you corrected a typo or added provenance metadata, it should usually keep the same ID. If the failure mode changes materially or the example now encodes a different expectation, create a new ID and retire the old one deliberately.

This discipline also helps when multiple engineers curate the same regression suite. Diff-based updates prevent one cleanup pass from silently duplicating or erasing another engineer's protected case.

### **Sampling should preserve failures, not mirror traffic**

A release-protection dataset is usually not supposed to match production frequency. If your traffic contains 80% easy greetings and 0.5% critical compliance requests, a representative sample may produce a useless benchmark with excellent aggregate scores and almost no release protection. For regression testing, you often need failure-preserving sampling rather than traffic-mirroring sampling.

That means deliberately overweighting certain examples:

- severe incidents,
- historically brittle scenarios,
- cases with repeated evaluator disagreement,
- boundary-condition inputs,
- business-critical workflows.

This does not mean abandoning representative traffic entirely. Keep a slice of realistic everyday cases so that broad usability regressions remain visible. But do not let volume determine importance. High-frequency easy cases should not dilute the visibility of rare, damaging failures.

A useful operating model is to separate a *discovery set* from a *release-gate set*. The discovery set can be broader and more exploratory, including newly observed cases and synthetic expansions. The release-gate set should be smaller, stable, and intentionally curated for protection value.

**Labeling discipline matters more than row count** A small, sharply labeled regression set is often more useful than a large ambiguous one. If labels are inconsistent, evaluators will disagree for reasons that have nothing to do with model quality. If the expected behavior is under-specified, experiment results become arguments about interpretation rather than signals about regression.

Write rubrics for the hard slices. Decide when a case requires exact factual correctness, when a refusal is acceptable, when a partial answer counts as failure, and when style is irrelevant. Track adjudication status for disputed items. A borderline case with unresolved disagreement may still belong in the dataset, but it should be marked as such rather than pretending to be clean ground truth.

Human review is especially important for calibration and for policy-sensitive examples. If a high-risk item is judged by multiple humans and produces disagreement, preserve that fact. It can inform evaluator design and may justify special handling in experiment interpretation.

**Synthetic data should expand coverage, not replace reality** Synthetic generation is useful for expanding coverage around real fail-



ures, but synthetic cases should be marked and not allowed to dominate the set. Once you identify a real defect class, synthetic examples can help you probe nearby variants: paraphrases, different personas, different document orderings, adversarial wording, or alternate tool arguments. This is often the fastest way to turn one production incident into a family of regression guards.

Promptfoo's `promptfoo generate dataset` CLI and its persona and instruction options are useful references for controlled scenario expansion. Use tools like that as amplifiers, not anchors. The anchor should remain the observed production case that proved the failure matters.

Always label synthetic origin explicitly. Synthetic examples are valuable because they broaden the local neighborhood of a real problem. They become dangerous when they overwhelm the suite and shift the benchmark toward artificially regular language or contrived edge cases that no longer represent customer behavior.

**Keep the hard examples even when they hurt the score** One of the most damaging curation habits is removing difficult examples because they make aggregate metrics look worse. If an example captures a real failure mode, a policy boundary, or a critical customer journey, a low score is often a feature of the benchmark, not a flaw in it. The dataset exists to expose risk, not to advertise progress.

You should retire examples only for principled reasons: the product no longer supports that behavior, the underlying source material is obsolete, the case is a duplicate, or the labeling rubric was fundamentally wrong. Do not remove a case merely because it is stubbornly hard. Persistent difficulty is often exactly why it belongs in the protected set.

This requires some political discipline. Teams naturally prefer benchmarks that trend upward. A serious regression dataset often resists that desire by preserving the examples that encode institutional mem-

ory of painful incidents.

**Refresh the suite without erasing history** Datasets decay. Corpora change, product scope expands, user language evolves, and yesterday's odd edge case can become today's common workflow. Periodic pruning and rebalancing are necessary so the regression set does not overfit to last month's issues only. The challenge is to refresh the dataset without erasing the history that made it protective.

A good maintenance pattern keeps three layers:

- *Stable core set*: long-lived, high-value cases that should rarely change.
- *Rotating challenge set*: newer edge cases and emerging patterns from recent traffic.
- *Incident backfill set*: examples added directly from support incidents, outages, or postmortems.

The stable core preserves continuity in experiment comparisons. The rotating challenge set keeps the benchmark current. The incident backfill set ensures painful failures become durable protections instead of forgotten anecdotes.

**A minimal operational playbook** A disciplined regression-curation loop is usually simple:

1. Mine live traces and incident reports for concrete failures.
2. Promote high-value spans or runs into dataset examples with provenance.
3. Label each example with slice, severity, and expected behavior.

4. Expand important failure families with a limited amount of marked synthetic variation.
5. Maintain stable IDs with `example_id_key` so updates are differential rather than append-only.
6. Periodically rebalance slices to preserve both realism and protection value.

The mechanics are not the difficult part. The difficult part is deciding that a regression suite is a safety device, not a leaderboard. Once you curate around that principle, the dataset becomes a durable memory of what your system must not break again, and every future experiment inherits that memory instead of starting from a blank slate.

## 6.5. Running Comparative Experiments for Prompts, Models, and Application Variants

Once you have a curated dataset and a stable evaluator suite, you can stop arguing from anecdotes and start making controlled engineering decisions. That shift matters because prompt changes, model swaps, retrieval-policy tweaks, and orchestration refactors often produce mixed effects. A variant can improve average answer quality while increasing severe failures on a small but critical slice. Another can reduce hallucinations but add latency or cost that makes it impractical in production. Comparative experiments exist to resolve those trade-offs under shared conditions rather than by intuition.

Phoenix supports both UI and SDK workflows for running the same dataset through different tasks, prompts, or models. The Python client setup is minimal:

```
from phoenix.client import Client

client = Client()
```

The installation path documented for experiment-oriented Python work is also straightforward:

```
pip install arize-phoenix-client \
  arize-phoenix-otel \
  openinference-instrumentation-openai \
  openai \
  datasets \
  duckdb \
  pandas
```

That setup only gives you the control surface. A useful experiment still depends on design discipline. The operative question is not “which variant got the highest score,” but “did the comparison isolate the change you care about, under fixed inputs and a stable measuring instrument?”

### **What makes an experiment valid**

A comparative experiment is credible only when the compared variants face the same task under the same conditions. In practical terms, that means a fixed dataset, fixed evaluator versions, clear scenario slices, and an explicit optimization direction. If any of those move during the comparison, your interpretation becomes fragile.

The fixed dataset requirement is the easiest to understand and the easiest to violate. If prompt A is evaluated on one set of inputs and prompt B on another, any score difference may simply reflect sample composition. Use the same examples for all variants, including the same hard cases, the same failure-derived regression items, and the same slice labels.

The fixed evaluator requirement is equally important and often overlooked. Phoenix’s server-side LLM evaluators are versioned and tag-driven, which is operationally important because experiment comparisons need a stable measurement instrument. If you modify the judge prompt, switch the judge model, or change a threshold midway, you are

no longer comparing application variants alone. You are also changing the ruler.

The explicit optimization direction requirement prevents a quieter class of mistakes. Before you run the experiment, decide what “better” means. Is the primary goal higher faithfulness? Lower severe-failure rate? Better pairwise preference on executive-summary style? Lower cost at equal quality? Without a declared objective, teams tend to cherry-pick whichever metric favors the preferred variant.

### **Compare one change at a time when possible**

The cleanest experiments change a single variable: one prompt revision, one model substitution, one retrieval policy difference, or one application code path. Single-variable comparisons make it easier to attribute outcomes. If the new prompt improves answer relevance but harms citation behavior, you know where to look. If you simultaneously change the model, prompt, retriever, and temperature, you may still find a winner, but you lose explanatory power.

There are legitimate reasons to compare bundled variants. A migration from one model family to another may require prompt restructuring and different sampling parameters. A retrieval redesign may alter chunking, reranking, and answer formatting together. When you run multivariate comparisons, document the bundle explicitly and accept that a “win” tells you the package is better, not which component inside the package caused the gain.

This is where detailed experiment metadata becomes part of the result. Log the application variant, model version, prompt version tag, evaluator version, and dataset version alongside every run. Without that provenance, later re-analysis becomes guesswork.

### **Use Phoenix experiments as controlled replay**

The operational shape of Phoenix experiments is simple: take a fixed

dataset, run one or more candidate tasks or variants over it, then evaluate the resulting outputs consistently. Official Phoenix experiment workflows support running the same dataset through different tasks, prompts, or models, and Phoenix recommends reviewing failure modes after an experiment, then defining evaluators for those issues and re-running so the experiment produces annotation columns.

That recommendation is more powerful than it first appears. Early experiments often reveal a defect that your original evaluator suite did not measure directly. Perhaps a prompt revision improved average helpfulness but started omitting citations in a subset of long-context cases. Instead of treating that as a one-off observation, codify it as an evaluator and rerun the experiment. The result becomes richer and more reusable because the newly discovered failure mode is now a first-class annotation column rather than a note in someone's notebook.

### **A practical comparison pattern**

Suppose you are evaluating two prompt variants for a retrieval-augmented assistant:

- *Prompt A*: concise answer-first style.
- *Prompt B*: more explicit grounding instructions and citation reminders.

A disciplined experiment would keep the dataset fixed, preserve the same retrieval context for each example, run both prompts through the same application logic, and evaluate both outputs with the same evaluator versions. Then inspect not only the aggregate score but also scenario slices such as long-context queries, ambiguous policy questions, and prior hallucination regressions.

It is entirely plausible for Prompt B to win overall while regressing on a slice such as short factual answers, perhaps because the stronger

grounding instructions make those answers unnecessarily verbose. If you only look at the average score, you may ship a variant that harms an important workflow even though the global number improved. Slice analysis is not optional; it is part of what makes the experiment informative.

### **Scenario slices should be designed, not improvised**

Every serious experiment needs segment-level interpretation. Aggregate metrics hide too much. The relevant segments usually come directly from the dataset metadata you curated earlier: severity, scenario family, critical customer journey, safety or policy class, retrieval-dependent cases, human-reference availability, and synthetic versus real origin.

This is where dataset curation and experiment design reinforce each other. If the dataset lacks slice metadata, you cannot answer questions such as:

- Did the new model improve only easy cases?
- Did a prompt revision hurt compliance-sensitive scenarios?
- Did retrieval changes help long documents but hurt short, direct questions?
- Did the variant fail disproportionately on incident-derived regressions?

A useful experiment review therefore starts at the aggregate level, then moves immediately to scenario slices and finally to per-example disagreements. The aggregate metric is a summary, not the decision.

### **Quality alone is rarely the full objective**

Model upgrades are a good example of why multi-metric evaluation matters. A new model may deliver better semantic quality while in-

creasing latency and cost enough to violate service constraints. Another may improve average performance but produce more tail-latency outliers on complex prompts. If you judge only on quality, you can select a variant that is technically stronger and operationally worse.

For model comparisons, track at least three families of outcomes:

- *Quality metrics*: correctness, faithfulness, preference, pass rate, severe-failure rate.
- *Efficiency metrics*: latency, token usage, throughput, provider cost.
- *Risk metrics*: safety failures, policy regressions, tool misuse, evaluator disagreement concentration.

The decision rule should be explicit. For example: adopt the new model only if it reduces severe-failure rate, preserves policy pass rate, and keeps median latency within an acceptable bound. That kind of rule is less glamorous than a single quality score, but it reflects how real systems are shipped.

### **Pairwise evaluation is often the right design**

When the actual question is comparative, pairwise preference can be a better evaluation design than independent scalar scoring. This is especially true when both outputs are plausible but differ in style, emphasis, or structure. Independent scalar scores often compress both candidates into the same broad band: both seem “good,” or both seem “acceptable.” A pairwise judge can answer the sharper engineering question: which one is better on this example?

Pairwise evaluation is particularly useful for prompt iteration. Imagine two responses to a customer-support question. Both are factually grounded and policy-compliant, but one is clearer and more action-



oriented. A pointwise 1--5 score may not separate them consistently. A pairwise preference judgment often will.

The trade-off is interpretability. Pairwise results are ideal for ranking variants but less suited to absolute release gates unless you also maintain pointwise checks for hard constraints. In practice, pairwise preference works well alongside deterministic guards and pointwise semantic evaluators.

### **Evaluator versioning is part of experimental rigor**

Phoenix's versioned, tag-driven server-side LLM evaluators help solve one of the most common experiment mistakes: silently changing the measuring instrument. If the evaluator prompt evolves or the judge model changes, the same candidate output may receive a different score for reasons unrelated to the application variant. That makes comparisons unstable.

For every experiment result, preserve at least:

- dataset version,
- evaluator version,
- evaluator prompt tag,
- judge model version when applicable,
- application variant identifier,
- prompt version,
- model version,
- relevant runtime configuration.

This is not bureaucratic overhead. It is what turns an experiment result into something you can reproduce, audit, and compare later. It

also makes failure investigations faster. If a release candidate suddenly “improves” after you updated the evaluator prompt tag, the provenance tells you the comparison is no longer apples-to-apples.

### **Interpret disagreements before declaring a winner**

A strong experiment review always drills into disagreements between variants. The most informative examples are often not the obvious failures but the cases where the variants diverge. If prompt A and prompt B produce materially different outputs for the same input, inspect those examples directly. They tell you what the system is actually trading off.

Disagreement review often reveals hidden failure patterns:

- the new prompt improves groundedness but over-refuses,
- the new model is better on common cases but worse on domain jargon,
- a retrieval change helps long documents but truncates short snippets needed for precise answers,
- an orchestration refactor improves latency but skips a valuable verification step.

Per-example inspection is also where you detect judge pathology. If the evaluator clearly prefers a verbose but less correct answer, you may need a better rubric or a different evaluation design.

### **Common experiment failures**

Several mistakes repeatedly produce misleading comparisons.

*Changing evaluator prompts during comparison.* If evaluator versions differ, the score delta may reflect measurement drift rather than system improvement.

*Comparing on different samples.* Without a fixed dataset, sample composition can dominate the result.

*Relying on a single aggregate score.* Important regressions often hide inside scenario slices or severe-failure counts.

*Ignoring optimization direction.* A higher average score is not enough if latency, cost, or policy failures also worsened.

*Overfitting to judge-friendly outputs.* Prompt wording can become tailored to the evaluator rather than to user value, especially with LLM judges.

*Failing to add evaluators after discovering a new failure mode.* If an experiment reveals a recurring defect and you do not encode it into the evaluator suite, the next experiment will miss it again.

### **A compact operating loop**

A reliable comparative workflow usually looks like this:

1. Define the hypothesis and the decision rule.
2. Freeze the dataset version and evaluator versions.
3. Run all candidate variants on the same dataset.
4. Compare aggregate metrics, then slice metrics, then per-example disagreements.
5. Review newly discovered failure modes and encode them as evaluators when possible.
6. Re-run the experiment with the expanded evaluator suite so the new signal becomes part of the official result.

This loop is what turns experiments into a durable engineering method rather than a one-time bake-off.

### **Experiments should produce decisions, not just dashboards**

The strongest Phoenix experiments are not the ones with the prettiest summary charts. They are the ones where you can state, in operational terms, why one variant wins. Perhaps Prompt B is chosen because it improves groundedness on the fixed regression set, preserves policy behavior on the safety slice, and incurs no meaningful latency penalty. Perhaps the new model is rejected because its quality gain is outweighed by higher cost and worse tail latency on critical workflows. Perhaps a retrieval change is promoted only for a subset of scenarios where it clearly dominates.

When experiments are designed around fixed data, stable evaluators, slice-aware interpretation, and explicit trade-offs, they stop being score-reporting exercises and become the decision engine for prompts, models, and application variants.



## Chapter 7

# Iterative Improvement and Programmatic Phoenix Workflows

*Many teams stop at observability: they can inspect a bad run, perhaps even score it, but they still improve the system through intuition and scattered manual fixes. This chapter shows how Phoenix becomes the backbone of an iterative quality program, where every important change can be traced from observed failure to evaluator design, experiment evidence, and automated operational policy.*

### 7.1. Closing the Loop from Observation to Action

A mature Phoenix workflow begins when a bad run stops being treated as an isolated annoyance and starts being treated as reusable evidence. You can usually spot the operational smell quickly: one trace shows a

hallucinated answer, another reveals a missed retrieval, a third exposes a tool call with malformed arguments, and each incident gets a local patch. The system may improve for that exact input, yet the team still lacks a way to say what kind of failure occurred, how often it occurs, and whether a proposed fix actually reduces it. Trace inspection explains a run. Iterative improvement explains a class of runs.

A concrete workflow helps fix that distinction early. Phoenix exposes traces and spans as analyzable data rather than dashboard-only artifacts, so you can move from anecdotal debugging into repeatable triage. In Python, a typical first step is to retrieve spans associated with suspicious executions and inspect them as a dataframe for recurring patterns such as retrieval emptiness, tool errors, or schema violations.

```
from phoenix.client import Client

client = Client()

# Retrieve spans for offline analysis.
# Exact filtering options depend on your project setup.
spans_df = client.spans.get_spans_dataframe()

print(spans_df.columns)
print(spans_df.head(5))
```

```
Index(['trace_id', 'span_id', 'name', 'span_kind',
      'start_time', 'end_time', 'status_code', ...],
      dtype='object')
```

That code does not improve anything by itself. Its value is that it changes the unit of work from “one frustrating trace” to “a population of executions that can be grouped, labeled, and measured.” Once you can query runs programmatically, you can stop relying on memory and screenshots. You can ask better questions: which failures recur, which layers produce them, and which signals distinguish them from healthy runs.

## From bad traces to failure hypotheses

A bad trace is evidence, not a final diagnosis. If a user receives an unsupported claim, the underlying defect may live in retrieval, context assembly, generation behavior, tool orchestration, or downstream formatting. Traces matter because they let you localize the failure boundary before you change anything. When a retrieval span shows no relevant documents despite their presence in the corpus, you are looking at a retrieval recall problem. When retrieval is strong but the generation span ignores the evidence, you are looking at a grounding or instruction-following problem. When the model chooses an unnecessary tool or alternates unpredictably between tools for similar inputs, you are looking at tool selection instability rather than prompt weakness in the narrow sense.

This distinction changes how you intervene. A prompt edit can sometimes mask a retrieval defect for a handful of examples, but it rarely fixes low recall as a class. Likewise, adding brittle post-processing to repair malformed JSON may hide a deeper schema-compliance issue in the model or tool contract. Phoenix traces are especially useful here because they capture not only model calls but also retrieval, tool use, timing, and custom logic. That breadth lets you decompose a user-visible failure into its contributing stages.

You need a failure hypothesis that is specific enough to test. “The app is unreliable” is not a hypothesis. “The app produces unsupported claims when relevant retrieval evidence is present but appears late in the context” is a hypothesis. “Tool invocation fails because the planner alternates between two semantically similar tools under long-context pressure” is a hypothesis. A useful hypothesis names the symptom, the probable layer, and the recurring condition.

### **Failure classes as operating vocabulary**

Once you have several investigated traces, the next move is to convert them into explicit failure classes. This is the point where debugging



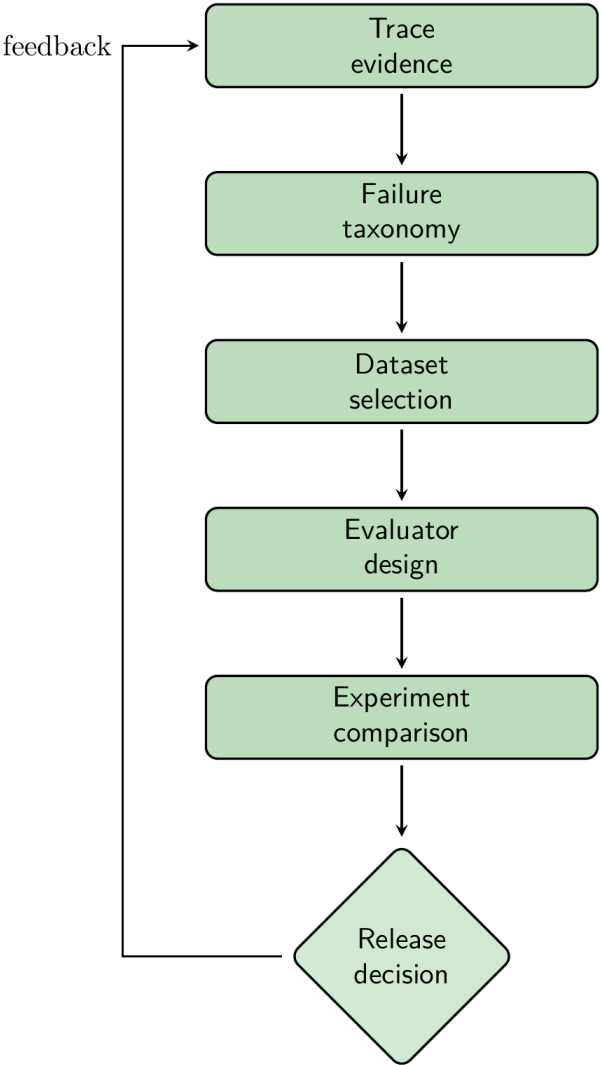
becomes quality engineering. A backlog of miscellaneous examples is hard to evaluate, hard to assign, and hard to automate. A failure taxonomy gives the team stable language and stable buckets.

Good failure classes are operationally meaningful rather than purely descriptive. They should tell you what kind of remediation to consider and what kind of evaluator could plausibly detect the issue. Common classes in LLM systems include:

- unsupported claims despite strong retrieval evidence,
- low retrieval recall despite relevant corpus content,
- instruction drift under long context,
- tool selection instability,
- schema noncompliance in structured output,
- latency-quality regressions introduced by orchestration changes.

These categories are not universal, and they should not be too fine-grained at first. If you create twenty narrowly phrased labels before you have measurement discipline, the taxonomy collapses into case notes. If you create one giant category such as “bad answer,” you hide the mechanisms that matter. Start with classes that reflect distinct causal structures and distinct evaluation strategies.

A useful test is whether two traces assigned to the same class would likely benefit from the same family of fixes. Unsupported claims with good retrieval evidence may call for grounding instructions, citation constraints, or answer synthesis changes. Low retrieval recall may call for indexing, chunking, query construction, or ranking changes. These are different intervention spaces, so they deserve different labels.



**Evaluators should emerge from observed failures**

Only after the failure class is clear should you design an evaluator. This sequencing matters because evaluator quality depends on what you are trying to detect. If you design metrics before you understand the defect class, you usually end up with broad, weak signals that blur several mechanisms together.

Phoenix supports multiple sources of judgment: human, LLM, and code-based annotation and evaluation. That flexibility is not just a feature list; it reflects the fact that different failure classes require different detection strategies. Some defects are deterministic. Schema noncompliance, missing required fields, invalid JSON, or a tool argument outside an allowed range are best handled by code checks. Some defects are semantic and contextual. Unsupported claims, answer completeness, or whether the response followed a nuanced instruction often require an LLM judge or human review. Some classes benefit from hybrid scoring, where deterministic checks gate obvious failures and an LLM evaluator handles the semantic remainder.

The design question is not merely “can I score this?” but “can I score this reliably enough to drive decisions?” A strong evaluator for iterative improvement has three properties.

First, it aligns with the failure class. An evaluator for unsupported claims should inspect the relationship between answer content and supplied evidence, not just answer fluency. Second, it is stable enough across examples that score movement means something. If tiny wording differences in the judge prompt swing outcomes unpredictably, you do not yet have a release-quality signal. Third, it is sensitive to the change you plan to make. If you are testing retrieval improvements, an evaluator that mostly measures style will not help you.

Phoenix evaluation runs are themselves traced and sent to a dedicated project, which is operationally valuable. When an evaluator behaves strangely, you can inspect the evaluator trace rather than treating the

judge as a black box. You can review the evaluator input, the prompt sent to the judge model, the resulting score, intermediate reasoning if present, and timing. That visibility is often what separates a trustworthy evaluator from an expensive mystery.

### **Baselines turn opinions into comparisons**

A proposed fix has no engineering meaning without a baseline. Teams often say a prompt, retrieval policy, or orchestration change “looks better” because they reviewed three memorable failures and saw improvement. That is useful for ideation and weak for decision-making. You need an initial measurement on a dataset that reflects the failure class you care about.

This dataset does not have to start large. It does have to be deliberate. For a newly discovered failure class, a focused dataset of representative examples is often better than a sprawling benchmark assembled from convenience samples. If the class is low retrieval recall, your dataset should include queries where relevant corpus content exists and can be verified. If the class is schema noncompliance, your dataset should cover the range of structures that actually matter in production. The goal is not to simulate the whole world immediately. The goal is to create a stable testbed where candidate changes can be compared against the same evidence.

Phoenix annotations are the connective tissue here. Annotations can target observability artifacts such as spans and documents, and preserved annotations propagate into datasets. That means a trace you investigated today can become tomorrow’s evaluation example with its quality judgment attached. This is the crucial shift from reactive debugging to cumulative learning: the expensive human attention spent understanding a failure is not lost after the incident closes.

### **Assembling a focused experiment loop**

Once the failure class, evaluator, and dataset exist, you can run an actual comparison rather than a narrative argument. Phoenix exposes experiment execution programmatically through the client. The central pattern is simple: define a task, define evaluators, run them on a dataset, and compare candidate behavior against a baseline.

```
from phoenix.client import Client

client = Client()

# dataset, task, and evaluators are prepared separately
# according to your application and failure class.
result = client.experiments.run_experiment(
    dataset=dataset,
    task=task,
    evaluators=[evaluator_a, evaluator_b],
)
```

The exact task and evaluator definitions depend on your application, and it is better to leave them explicit in your own codebase than to hide them behind generic wrappers. What matters operationally is that the experiment uses a fixed dataset, a stable evaluator suite, and a clearly identified candidate change. Without those controls, you cannot interpret score movement.

For version-sensitive workflows, keep a brief compatibility check in mind. Current Phoenix client surfaces for datasets, experiments, prompts, and trace-linked curation have evolved, and some CLI and dataset-linkage features depend on contemporary client versions. When you operationalize this loop, verify the package version that provides the specific capability you depend on, especially if you preserve span associations from traces into datasets.

### **Sampling strategy determines what you learn**

The dataset should be shaped by the failure class, not by convenience. There are three common sources of examples, and each serves a different purpose.

First, incident-derived examples come from traces you already know are problematic. They are the fastest way to bootstrap a regression set and are especially effective for newly discovered defects. Second, contrastive examples pair failures with nearby successes. These help you avoid building evaluators that simply flag all difficult cases. Third, background samples represent ordinary traffic and protect you from overfitting to spectacular failures while degrading typical behavior.

A healthy experiment set usually mixes all three, but not in equal proportions. If you are trying to reduce a known high-cost defect, deliberately oversample that defect class during development. Just do not mistake that focused dataset for a general performance benchmark. Keep the label on the dataset honest: this is a regression set for class X, not a universal quality scorecard.

When source traces are curated into datasets, preserving linkage back to the original spans strengthens the chain of evidence. If an experiment shows improvement, you can still inspect the underlying production-like traces that motivated the test. If the improvement disappears under broader sampling, you can identify whether the problem lies in dataset composition, evaluator sensitivity, or the candidate change itself.

### **Reviewing metric movement without fooling yourself**

Interpreting experiment results requires more discipline than reading a leaderboard. A metric shift is only useful when you know what else was supposed to move. Before running the experiment, state the expected pattern. A retrieval intervention might improve grounding and completeness while increasing latency. A stricter output instruction might improve schema compliance while hurting answer richness. These trade-offs should be predicted, not discovered accidentally after the fact.

Avoid aggregate scores early in the loop. A single blended metric can

conceal the fact that one failure class improved while another worsened sharply. Review metrics by failure class and by slice. If your unsupported-claims rate falls but tool-selection errors rise, you need to know that before shipping. Phoenix supports the evidence-gathering primitives for this style of analysis, but the discipline comes from how you structure the loop, not from any single dashboard.

Evaluator traces also deserve routine review. Phoenix runtime behavior for evaluations includes retrying temporary failures, distinguishing permanent errors, and adjusting concurrency dynamically during larger runs. That makes large-scale evaluation more operationally tractable, but it does not remove the need to inspect evaluator failures as first-class artifacts. If scores become erratic, the fault may be provider-side throttling, ambiguous evaluator prompts, or an evaluator that is poorly matched to the failure class.

### **Common anti-patterns**

Several anti-patterns appear repeatedly in teams that have observability but not yet an improvement loop.

One is patching before localizing. If you change prompts, retrieval, and output parsing at the same time, you may accidentally improve the observed example while learning nothing reusable. Another is building evaluators from anecdotes. A judge designed around three memorable failures often measures resemblance to those failures rather than the underlying class. A third is using one broad metric to cover heterogeneous defects. “Answer quality” is too blunt if your actual problems are retrieval recall, citation grounding, and schema compliance.

A fourth anti-pattern is declaring victory on tiny handpicked samples. Small, focused regression sets are legitimate when they are labeled as such and used for the right purpose. They become dangerous when teams present them as proof of general improvement. Finally, avoid uncontrolled drift in prompt references or evaluator definitions.

Phoenix prompt management supports retrieval by mutable selectors such as name or tag and by immutable version identifiers. For development, mutable references are convenient. For production-like experiments and stable baselines, immutable version IDs are much safer because they prevent silent benchmark drift.

### **Handing off from investigation to team process**

The closed loop is complete when a single investigated failure becomes part of a managed quality system. The progression is straightforward: inspect traces, identify a recurring defect, assign it to a failure class, design or refine an evaluator, curate a dataset, compare candidate fixes, and decide whether to promote, revise, or discard the change. Each step narrows ambiguity. Each step also converts fleeting debugging effort into an asset the team can reuse.

That operating model gives you a compact vocabulary for engineering discussions. Instead of saying “the app looked weird on a customer call,” you can say “this is an unsupported-claim class with strong retrieval evidence; we need a grounding evaluator and a focused regression set before changing the prompt.” Instead of saying “the new version seems faster and mostly fine,” you can say “latency improved, schema compliance held, but tool-selection instability regressed on the failure slice that motivated the change.” Those sentences carry different organizational weight because they are attached to trace evidence, explicit categories, and controlled comparison. Once you work this way consistently, improvement stops being a sequence of clever fixes and becomes a system of record for what the application has learned.



## 7.2. Operating Prompt and Application Iteration as an Engineering Discipline

Teams often reach for prompt edits first because prompts are cheap to change and easy to discuss. That convenience is useful and dangerous. A weak answer can come from prompt wording, but it can also come from poor retrieval recall, irrelevant context assembly, unstable tool descriptions, planner errors, model choice, or a downstream validator that rejects good outputs. If you treat every defect as a prompt problem, you bias the system toward superficial fixes and lose the ability to reason about where the application is actually failing.

A disciplined iteration process starts by treating prompts as one intervention layer among several. Phoenix prompt management is valuable here not because it makes prompt editing easier, but because it makes prompt changes explicit, versioned, and testable. You can retrieve prompts by selector during development, pin immutable versions for production-like evaluation, and keep prompt artifacts visible in a shared workflow rather than buried inside notebooks or ad hoc source edits.

A practical selector policy makes that discipline concrete. In TypeScript, you can retrieve a prompt by name when you want development convenience, or by immutable version identifier when you need controlled behavior.

```
import { getPrompt } from "@arizeai/phoenix-client/prompts";

// Development convenience: latest version by name
const devPrompt = await getPrompt({
  prompt: { name: "support-answer" }
});

// Deployment-safe: immutable version
const prodPrompt = await getPrompt({
  prompt: { versionId: "pv_1234567890abcdef" }
});
```

The distinction matters operationally. A name-based lookup is a moving target. It is useful when you are iterating quickly and intentionally want the latest artifact. A version identifier is immutable and non-deletable, which makes it the right reference for controlled experiments, staged rollouts, and any environment where you need reproducibility. A tag can sit between those two extremes when you want controlled but movable references such as `staging` or `candidate`. The engineering policy is straightforward: use latest-by-name for local development, a controlled tag for shared pre-release environments, and immutable `versionId` references for production-like decisions.

### **Choose the intervention layer before you choose the fix**

The most important design decision in iteration is not the wording of the candidate prompt. It is the choice of intervention layer. When a trace shows poor behavior, ask which part of the application most plausibly caused it and which part can change that causal mechanism with the least collateral risk.

Useful intervention layers include:

- prompt wording and prompt structure,
- retrieval depth, ranking, and chunk selection,
- model choice and generation parameters,
- tool availability, descriptions, and argument schemas,
- planner or executor logic,
- context shaping, compression, and truncation policy,
- downstream validation and guardrail logic.

These layers differ along four operational axes: speed of change, expected impact, observability, and regression risk. Prompt edits are

fast and easy to compare, but they can hide deeper defects. Retrieval changes may have larger quality impact for grounding failures, but they can also increase latency and pollute context. Tool description changes may stabilize selection behavior, but only if the tool boundary is the actual source of confusion. A model swap may improve instruction following, yet it may also change latency, cost, and output style so much that prompt results are no longer comparable.

This is why failure classes from the earlier operating loop matter. If the failure class is “unsupported claims despite strong retrieval evidence,” you should examine prompt grounding, answer synthesis, citation constraints, and context placement before touching indexing. If the class is “low retrieval recall despite relevant corpus content,” prompt edits are usually secondary. If the class is “schema noncompliance under otherwise good reasoning,” output constraints, structured generation strategy, and deterministic validators are more promising than broad answer-style rewrites.

### **Write hypotheses as testable movement patterns**

A candidate change should do more than name a tweak. It should encode a falsifiable hypothesis about mechanism and expected metric movement. You need three fields at minimum:

- the target failure class,
- the proposed causal mechanism,
- the expected improvements and expected trade-offs.

That framing is what converts prompt work from craft to engineering. Consider three examples.

If you add stricter formatting instructions to the system prompt, your hypothesis might be: “This will improve schema compliance for extraction tasks because the model receives clearer structural constraints,

but completeness may decrease on ambiguous inputs.” If you increase the retrieval set size, your hypothesis might be: “This will improve factual grounding on sparse queries by increasing recall, but latency will rise and answer quality may fall on dense topics due to distractor passages.” If you rewrite tool descriptions to make tool boundaries more distinct, your hypothesis might be: “This will reduce tool selection instability without changing answer quality for no-tool cases.”

These are useful hypotheses because they predict multi-metric behavior. A change that improves one score while harming a related operational metric is not necessarily a failure; it may be an expected trade-off. But you need the trade-off stated in advance or you will rationalize any result after the run.

### Version prompts as application artifacts

Prompt text should be managed like code or configuration with operational consequences, not like disposable notebook state. Phoenix prompt management supports prompt creation, versioning, retrieval by selector, and conversion into provider SDK payloads. In Python, prompt creation uses a named prompt with an explicit version object.

```
from phoenix.client import Client
from phoenix.client.types import PromptVersion

client = Client()

client.prompts.create(
    name="support-answer",
    version=PromptVersion(
        template="Answer using only provided context."
    ),
)
```

That minimal example matters for two reasons. First, the prompt name becomes a durable artifact identifier that other environments can reference. Second, prompt versions create a history of changes that can be compared and audited. Prompt names are constrained to lowercase

alphanumeric characters plus hyphens and underscores, which is a useful discipline by itself because it pushes teams toward stable naming rather than ad hoc labels.

If you use TypeScript-heavy application code, Phoenix also supports prompt artifact workflows in `@arizeai/phoenix-client/prompts` using `createPrompt`, `promptVersion`, `getPrompt`, and `toSDK`. The exact provider payload generation depends on your runtime integration, but the important design principle is stable indirection: store prompt content as a managed artifact, retrieve it deliberately, and feed the chosen version into the provider SDK without silently mixing prompt revisions across environments.

### **Separate exploratory work from controlled evaluation**

Iteration has two rhythms, and they should not be confused. Early exploratory work is broad. You may try several prompt structures, retrieval settings, or tool descriptions quickly to discover promising directions. Controlled evaluation is narrower. You isolate one candidate change, run it against a stable dataset, and interpret the result against a fixed evaluator suite.

The failure mode is obvious: teams often jump from exploration directly to shipping because one variant “looked better” on a handful of examples. That skips the point where the change is stabilized enough to compare. To avoid that, define a promotion boundary between exploratory artifacts and evaluable artifacts. An exploratory prompt can live under a development name lookup. A candidate prompt intended for comparison should be versioned and pinned. The same logic applies to retrieval and orchestration settings: if you change chunking, ranking, and prompt wording simultaneously, you have created a new system but no explanation for why it behaves differently.

Controlled deltas are essential when the intervention layer is not the prompt. If you are testing retrieval changes, keep the prompt fixed.

If you are testing prompt structure, keep the retrieval policy fixed unless the prompt explicitly depends on new context formatting. If you change evaluator prompts at the same time as application prompts, you contaminate the comparison. You may still need joint redesign later, but you should not pretend the result isolates one variable.

### **Prompt work is only one part of system optimization**

A prompt can compensate for some weaknesses elsewhere in the stack. It can encourage grounded answering, discourage unnecessary tool use, or clarify output format. But prompt changes have limits because they operate through model behavior rather than through hard system constraints.

Retrieval settings, by contrast, alter the evidence available to the model. If grounding fails because the right document never reaches the context window, a clever prompt can only partially compensate. Tool orchestration changes affect whether the model can even execute the right action path. Guardrails and validators control what outputs are accepted or rejected after generation. Each layer has a different failure surface.

You should choose the layer that best matches the failure mechanism while minimizing side effects. Prompt changes are usually the fastest to try and the easiest to revert. Retrieval changes often produce larger gains for factuality and completeness but introduce cost and latency trade-offs. Model changes can improve broad capability but make regression analysis harder because many behaviors shift at once. Guardrails give you deterministic enforcement but may reduce flexibility or increase false rejections. System optimization therefore asks a sharper question than “what improves the score?” It asks “what intervention most directly addresses the failure class, produces interpretable movement, and preserves operability?”

### **Runtime prompt retrieval requires resilience design**

Externalized prompt management introduces a real runtime dependency. When the application fetches prompts from Phoenix at runtime, you gain consistency, central review, and environment portability, but you also depend on network availability and service responsiveness. Phoenix documentation explicitly warns about this and recommends caching and fallback mechanisms.

Treat prompt retrieval like any other configuration dependency with operational value. Cache the resolved prompt version locally in memory or on disk. Keep a fallback copy of the last known good prompt for the environment. Decide whether the application should fail closed, fail open, or degrade gracefully when prompt retrieval is unavailable. For a user-facing assistant, graceful degradation with a cached immutable version is usually the right default. For a high-risk workflow where prompt changes carry compliance meaning, you may want stricter startup validation and explicit refusal to run on an unknown prompt state.

A practical policy is simple:

- local development may fetch latest-by-name on each run,
- shared staging should resolve a tag or candidate version and cache it,
- production should resolve a pinned `versionId` and cache it with a known fallback.

This policy gives you the flexibility of centralized prompt management without accepting uncontrolled runtime drift.

### **Use Phoenix CLI and artifacts to support reviewable changes**

Prompt engineering becomes more disciplined when prompt artifacts are visible outside the application codebase. With

@arizeai/phoenix-cli 0.4.0 and later, Phoenix provides terminal workflows for listing and viewing prompts as well as operating on datasets and experiments. That matters because it enables a review practice closer to standard software engineering: people can inspect the artifact history, compare what changed, and align experiment results with named prompt versions rather than with copied text fragments in chat logs.

```
px prompts
px prompt support-answer
px datasets
```

These commands do not replace application-side evaluation, but they support a healthier workflow. A prompt candidate should be identifiable, discussable, and easy to locate when an experiment result is reviewed later. The same applies to datasets and experiments. When artifacts are named and queryable, iteration becomes traceable rather than anecdotal.

### **Evaluate interventions by expected trade-offs**

Quality work in LLM systems is almost always multi-objective. A change that improves one metric can still be a net regression. You need explicit decision signals.

For prompt changes, evaluate not just semantic quality but also schema compliance, tool invocation behavior, refusal rates where relevant, and latency if the prompt materially increases token count. For retrieval changes, evaluate recall-oriented quality signals alongside context pollution, latency, and model distraction effects. For model changes, evaluate cost and response variance in addition to task quality. For guardrail changes, evaluate false accept and false reject behavior, not just total blocked outputs.

Two trade-offs recur often. The first is factuality versus flexibility. A prompt or guardrail that sharply restricts claims may improve ground-



ing but reduce useful synthesis. The second is recall versus noise. Adding more retrieved material can improve evidence coverage while making the answer less focused. Those are engineering trade-offs, not philosophical ones. You should state them before running the experiment and inspect the slices where harm is most likely.

This is where the failure-class discipline pays off. If the motivating issue is unsupported claims, you may accept a small reduction in stylistic richness to gain stronger grounding. If the motivating issue is missed relevant evidence, you may accept modest latency growth to improve retrieval recall. The key is that the cost is intentional and measured, not accidental.

### **Anti-patterns that look productive and are not**

Several habits create motion without learning. One is overfitting to a benchmark dataset by repeatedly tuning prompts until the examples look perfect. If the dataset is small or unrepresentative, you have optimized to the benchmark rather than to the application. Another is collapsing multiple objectives into one aggregate score too early. A higher blended score can hide catastrophic failure in a critical slice such as tool misuse or invalid structured output.

Another common mistake is treating model swaps as interchangeable with prompt refinement. A different model may improve results, but it changes the optimization landscape so broadly that prompt conclusions may no longer transfer. Handle model changes as a separate intervention class unless you are explicitly redesigning the whole stack. Also avoid changing evaluator prompts and application prompts in the same experimental round. If both move, you cannot tell whether the system improved or the judge changed its standard.

A final anti-pattern is running production-like environments against mutable prompt references. Name-based lookup is convenient, but it undermines reproducibility when a new prompt version silently

changes behavior. If a candidate is important enough to benchmark or release, it is important enough to pin.

### **A repeatable proposal-review-test-promote cadence**

A disciplined iteration cycle is compact. Start with a named failure class and a small set of implicated traces. Propose one primary intervention layer. Write the hypothesis in terms of expected metric movement and expected side effects. Version the candidate artifact, whether it is a prompt revision, a retrieval configuration, or a tool description update. Evaluate against a stable dataset and evaluator suite. Review not just the headline score but the motivating failure slice and the operational trade-offs. Promote only the variants whose behavior you can explain.

That cadence creates a useful form of technical accountability. Every candidate change has a reason, a measured effect, and a durable artifact trail. Prompt iteration remains fast, but it stops being improvisation. It becomes one controlled instrument in a larger application design practice, where changes are chosen because they match the failure mechanism and survive comparison against the system behaviors that actually matter.

## **7.3. Automating Trace, Evaluation, and Annotation Workflows with Phoenix Clients and APIs**

Manual inspection works until trace volume outruns human attention. At that point, the main problem is no longer visibility. It is repeatability. You may already know how to inspect a bad run, identify a failure class, and compare candidate fixes. The operational challenge is turning those steps into software that runs every day without depending on whoever remembers the query, the filter, or the interpretation logic.

Phoenix gives you a useful boundary for that transition. It records traces, evaluator runs, and annotations in forms that you can inspect interactively and also query or drive programmatically. That means you can move from “I found a bad example” to “a scheduled job retrieves suspicious runs, enriches them, evaluates them, and updates the working dataset for the team.” The implementation surface is not one monolithic automation API. It is a set of composable client capabilities for reading traces, writing quality signals, and running experiments in a controlled way.

A small Python job captures the basic shape. You connect a client, fetch spans into a dataframe, apply your own triage logic, and hand the resulting examples into the next stage of your workflow.

```
from phoenix.client import Client

client = Client()

spans_df = client.spans.get_spans_dataframe()

failed = spans_df[
    spans_df["status_code"].notna()
]

print(f"retrieved={len(spans_df)}")
print(f"candidate_failures={len(failed)}")
```

```
retrieved=1842
candidate_failures=137
```

That code is intentionally modest. It demonstrates the most important pattern in automation work: Phoenix provides the observability record, and your code layers decision logic on top of it. The triage filter can be crude at first—non-null error status, unusually high latency, missing retrieval results, or spans tagged by an evaluator outcome. What matters is that the filter becomes executable and repeatable. Once that happens, you can run it nightly, compare volumes over time, and feed the results into datasets or review queues.

## Read paths come first

You should build automation in the same order that a human investigator works. Start with read paths. If you cannot reliably fetch the runs you care about, every write-side operation becomes fragile.

In Phoenix, a common read path is span-oriented rather than trace-oriented because spans often correspond to the actual unit you want to reason about: retrieval calls, tool invocations, generation steps, or evaluator runs. Dataframe access is especially practical in Python because it lets you use standard filtering, grouping, and aggregation machinery without inventing an intermediate representation.

A healthy retrieval job answers four questions explicitly:

- What time window is the job responsible for?
- Which project or environment does it query?
- Which risk profile or failure heuristic selects candidates?
- How does it avoid double-processing the same evidence?

The last point is where many notebook scripts fail. They fetch “recent failures,” print a few rows, and stop there. A real automation job needs stable selection criteria and some notion of idempotency. If you rerun yesterday’s batch, you should know whether the job is reprocessing the same spans intentionally or because your filter drifted.

Phoenix client configuration supports this operationalization cleanly through environment variables. The officially documented variables include `PHOENIX_BASE_URL`, `PHOENIX_API_KEY`, and `PHOENIX_CLIENT_HEADERS`. That lets you separate code from deployment-specific connection settings.

```
export PHOENIX_BASE_URL=http://localhost:6006
export PHOENIX_API_KEY=your_api_key
```

With those in place, the same job can run locally, in a scheduled container, or inside CI with environment-appropriate credentials and routing.

### **Write paths need provenance**

Once a job can retrieve candidate runs, the next temptation is to write labels back immediately. That is useful and risky. Labels without provenance become technical debt very quickly. If you persist a judgment such as “hallucination” or “schema failure,” you should also know where it came from: a human reviewer, an LLM evaluator, or deterministic code.

Phoenix annotations are designed around exactly that distinction. The platform supports annotator kinds such as Human, LLM, and Code, and annotation targets include spans and documents, with related SDK surfaces also covering traces and sessions. That model matters because it prevents your quality history from collapsing into an undifferentiated bag of scores. A human adjudication has different authority than an automated heuristic. A code-based schema validator is different again.

Even when the exact write call is abstracted behind your internal tooling, preserve these metadata concepts in your own pipeline design. Store or forward, at minimum, the target artifact, annotator kind, label name, score or class value, and the origin of the run that produced it. If you later promote annotations into datasets, that provenance determines whether the signal can support benchmark comparison, reviewer assignment, or only exploratory triage.

A practical pattern is to split write-side automation into two lanes. The first lane writes low-risk machine-generated annotations that are directly traceable to deterministic checks, such as invalid JSON, missing required fields, or timeout events. The second lane queues ambiguous cases for human or LLM review rather than asserting ground truth

prematurely. That distinction keeps your quality store useful instead of polluted.

### **Programmatic experiments close the loop**

Once you can retrieve candidate failures and preserve their judgments, you can automate the expensive part of iteration: recurring experiments. Phoenix supports programmatic experiment execution in Python through `client.experiments.run_experiment(...)`. In TypeScript, the documented experiment surface includes `createDataset`, `asExperimentEvaluator`, `runExperiment`, and the `EvaluatorParams` type from `@arizeai/phoenix-client`.

A TypeScript example is especially useful when you want the evaluation job to live near application code or CI logic rather than in a Python notebook.

```
import {
  createDataset,
  asExperimentEvaluator,
  runExperiment
} from "@arizeai/phoenix-client";

const dataset = await createDataset({
  name: "schema-regression"
});

const evaluators = [
  asExperimentEvaluator(async (params) => {
    const passed = params.output.includes("{}");
    return {
      score: passed ? 1 : 0,
      label: passed ? "validish" : "invalidish"
    };
  })
];

const result = await runExperiment({
  dataset,
  task: async (example) => example,
  evaluators
});
```

The example is deliberately simple and the evaluator is only illustrative of wiring, not a production-quality schema validator. The real lesson is structural. Automation jobs can create or update a dataset, wrap evaluators into the experiment runtime, and run the full comparison step without a human opening the Phoenix UI. That is the bridge from ad hoc benchmarking to an internal quality service.

For larger evaluation runs, Phoenix’s evaluation runtime handles several operational concerns that you would otherwise have to build yourself: retries for provider throttling, differentiation between temporary and permanent failures, and dynamic concurrency adjustment. Those behaviors are not minor conveniences. They determine whether a nightly evaluation job finishes reliably or oscillates between partial success and noisy failure. Even with that support, you still need job-level observability, timeout budgets, and retry boundaries in your own scheduler.

### **Evaluator runs are observable too**

A common mistake in automation design is to treat evaluator execution as a hidden implementation detail. That makes debugging painful when scores drift, a provider changes behavior, or a judge prompt becomes brittle. Phoenix avoids that trap by tracing evaluator runs automatically. The resulting traces expose evaluator inputs, prompts, reasoning, final scores, and execution timing.

That design enables a more sophisticated automation pattern: build calibration workflows for the evaluators themselves. If an LLM-based judge suddenly flags too many healthy examples, you can inspect evaluator traces the same way you inspect application traces. If evaluation latency spikes, you can separate provider throttling from prompt inflation or oversized example payloads. If one evaluator becomes noisy on a specific slice, you can compare its trace history against human annotations and decide whether to re-prompt, replace, or downgrade that

evaluator’s authority.

Operationally, this means your quality pipeline should treat evaluator failures as first-class events. Do not just record the aggregate score table. Persist references to the experiment run, evaluator identity, and any exceptions or partial failures. A job that returns “quality unchanged” while half the evaluators silently errored is worse than no job at all because it creates false confidence.

### **Offline export supports hybrid workflows**

Not every analysis belongs inside Phoenix. Sometimes you need to export traces with annotations, compute additional metrics in a separate environment, or retain evidence as CI artifacts. Phoenix CLI supports this path with trace export that includes annotations.

```
px traces export --include-annotations
```

That command is useful in three common situations. First, you may need offline feature engineering or statistical analysis that is easier in a data warehouse or notebook environment. Second, you may want CI jobs to retain the exact evidence bundle associated with a release candidate, including both trace structure and attached quality judgments. Third, you may be migrating or reconciling quality data across environments and need a transport format that preserves annotation context.

The important design choice is to avoid creating a shadow quality system that diverges from Phoenix. If you export data for offline processing, define a round-trip policy. Decide which derived metrics remain external, which ones should be written back as annotations or summaries, and how you maintain lineage between the exported artifact and the original trace identifiers. Without that discipline, two competing truths emerge: Phoenix says one thing, your offline analytics say another, and nobody knows which system of record to trust.

### **A representative automation pipeline**



A recurring quality job usually has five stages.

First, retrieve a bounded slice of traces or spans. Bound it by time and scope so that retries are safe. Second, score or classify the slice using a mix of deterministic checks and evaluator runs. Third, persist results in forms that support later action: annotations for localized judgments, datasets for curated regression examples, and reports for human review. Fourth, compare the resulting distribution against a baseline or threshold. Fifth, emit a compact artifact for downstream systems, such as a JSON report, a saved dataframe, or a CI summary.

In Python, you might prototype the first and fourth steps with dataframe operations and hand the dataset curation step to your internal tooling. In TypeScript, you might keep the experiment runner close to application code and deploy it inside the same CI environment that builds the service. Neither language choice is inherently better. The important distinction is between one-off analysis code and hardened jobs. Prototypes can tolerate manual inspection and loose failure semantics. Scheduled jobs need strong boundaries around input scope, retry behavior, logging, and artifact persistence.

### **Integration concerns shape reliability**

Once you automate Phoenix workflows, several systems concerns become more important than the core client calls.

Authentication should be externalized through environment variables or secret management rather than embedded in notebooks. Pagination and batch sizing matter when trace volume grows, even if an initial prototype only touches a few hundred examples. Retry policy needs to distinguish Phoenix client transport issues from evaluator-provider throttling. Idempotency matters whenever a job can rerun on the same time window. Provenance matters whenever a label can influence later experiments.

You also need a clear contract between job code and selection logic. If your filter for “suspicious runs” is defined informally in application code, dataset composition will drift invisibly over time. That drift breaks comparability. Encode the selection rule in code, version it, and log enough metadata to reconstruct why an example entered the batch.

Keep a practical boundary around undocumented interfaces. The current client and CLI surfaces are well suited to many workflows, but exact REST endpoint paths and full request schemas were not established in the retrieved material. If you need raw API integrations beyond the documented client packages, verify them from the current API reference before hard-coding them into automation. Otherwise, prefer the supported client interfaces and exported artifacts.

### **Notebook convenience versus service discipline**

Ad hoc notebooks are excellent for discovering what to automate. They are poor containers for production quality policy. The difference is not style. It is reproducibility. A notebook usually mixes retrieval, heuristic filtering, inline judgments, and presentation in one mutable document. A service or batch job separates those concerns, pins dependency versions, emits logs, and leaves an auditable trail.

A good migration path is gradual. Start with a notebook to discover the right dataframe filters and evaluator mix. Move stable selection logic into a script. Add configuration through environment variables. Add structured outputs and run metadata. Only then embed the workflow into a scheduler, internal platform job, or CI pipeline. If you jump directly from dashboard work to a fully automated gate without this intermediate hardening, you will automate ambiguity.

### **Anti-patterns that automation amplifies**

Automation multiplies both good design and bad design. If your selection filter is unstable, a scheduled job turns that instability into noisy

metrics every night. If your annotations lack provenance, batch labeling can contaminate datasets at scale. If your evaluator is poorly calibrated, automatic experiment runs turn one judge's bias into an institutional decision signal.

Another common anti-pattern is coupling business logic too tightly to undocumented behavior. If you depend on implicit ordering, hidden fields, or inferred status conventions instead of the documented client surface, an innocuous library upgrade can quietly break the pipeline. Resist the urge to treat exported tables or JSON blobs as fixed schemas unless you own the compatibility contract.

The healthiest automation pipelines remain modest in ambition and strict in record-keeping. Retrieve only what you can justify, label only what you can explain, and run only the experiments whose outputs you are prepared to interpret.

Once you encode the loop into software, Phoenix stops being only a place where engineers inspect traces after something goes wrong. It becomes a substrate for repeatable quality operations: jobs that find failures, preserve judgments, compare variants, and feed those results back into the system with enough structure that the organization can trust the evidence.

## **7.4. Designing CI and Release Gates with Phoenix Quality Signals**

A quality signal becomes strategically important when it changes what you are willing to ship. Many teams reach a frustrating stage where they can diagnose failures well, run thoughtful experiments, and still release regressions because none of that evidence has been turned into an explicit control. The build passes, the prompt version is tagged, and only after deployment does someone notice that groundedness slipped,

a tool-use path became unstable, or latency rose enough to change user behavior. The technical gap is not observability. It is policy.

Phoenix provides the raw materials for that policy: curated datasets, evaluators, experiment runs, annotation history, and traceability back to production spans. Those pieces do not form a turnkey CI framework by themselves. You have to compose them into release rules. That composition is where most of the engineering judgment lives. A good gate blocks meaningful regressions, tolerates evaluator noise, and preserves delivery speed for low-risk changes. A bad gate creates false failures, hides important slices inside aggregate scores, or teaches the organization to distrust its own quality process.

A practical gate starts with a small, explicit contract. You define a release candidate, evaluate it against a fixed dataset and evaluator suite, compare the result to a known baseline, and classify the outcome as pass, warn, review, or fail. You can express the mechanics programmatically with Phoenix experiment execution and keep the policy decision in your CI logic.

```
from phoenix.client import Client

client = Client()

result = client.experiments.run_experiment(
    dataset=release_dataset,
    task=candidate_task,
    evaluators=[helpfulness_eval, groundedness_eval],
)
```

That call is only the measurement step. The gate policy sits around it. You decide what counts as acceptable movement, which metrics are binding, and what evidence is preserved when the candidate fails. That distinction matters because the same experiment result can support different delivery policies depending on risk. A customer-support assistant that must remain grounded against provided evidence deserves a stricter gate on unsupported claims than an internal brainstorming

tool. A reversible prompt-only change can tolerate more warning-only signals than a model swap that affects cost, latency, and output variance all at once.

### **Informative metrics versus gating metrics**

Not every Phoenix-derived signal should block a merge or release. Some metrics are exploratory. They help you learn about the system, but they are not stable enough or important enough to act as hard controls. Others are reliable enough and aligned enough with user harm that they can support automated policy.

A useful classification has four levels:

- hard blockers: failing these prevents promotion,
- soft warnings: failing these requires visibility but not immediate rejection,
- review triggers: failing these routes the candidate to a human decision,
- post-deployment monitors: these do not block release but must be watched after promotion.

Groundedness on a gold dataset derived from annotated production failures often belongs in the hard-blocker or review-trigger class for retrieval-based applications. A broad helpfulness score from an LLM judge may be better as a review trigger than a blocker if judge variance is nontrivial. Evaluation runtime failure rate is often a blocker once the pipeline matures, because a quality report built on partial evaluator failure is not trustworthy. Trend metrics such as style preference or verbosity usually belong in the warning class unless the application has strict brand or compliance requirements.

This classification keeps you from overloading the gate with every available signal. It also forces a more useful question: what evidence do you trust enough to stop delivery? If the answer is “all of it,” your gate will become brittle. If the answer is “almost none of it,” the gate becomes ceremonial.

### **Use fixed suites and known baselines**

A gate without a stable baseline is only a ritual. The release candidate must be compared against something fixed: a prior prompt version, a currently deployed model-and-config combination, or a named baseline task implementation. The dataset and evaluator suite must also be fixed for the duration of the decision window. If you change the dataset while assessing the candidate, you are changing both the ruler and the measurement.

Phoenix annotations are especially valuable here because they preserve labels and scores across the improvement loop. A trace investigated in production can become part of a curated dataset, and that label history can remain attached as the example moves into benchmarking and release review. This continuity is what allows regression detection against prior behavior rather than anecdotal inspection. You are not asking whether the new candidate looks good in the abstract. You are asking whether it preserves or improves behavior on evidence you have already determined matters.

This is also where slice design matters. A single monolithic dataset is rarely enough. Keep a general regression suite for broad sanity checks, but add focused slices for failure classes that carry high user cost: unsupported claims with strong evidence, schema violations, unstable tool selection, or latency-sensitive queries. A release can pass the general suite and still fail a critical slice. If the gate only sees the aggregate score, you ship a regression while congratulating yourself on the average.

## Thresholds should match risk and signal quality

Choosing thresholds is a policy problem, not just a statistical one. You want thresholds that are strict enough to catch real regressions and permissive enough to avoid constant false alarms. The right tolerance depends on evaluator stability, sample size, business criticality, and reversibility of the change.

For high-confidence deterministic checks, you can usually be strict. If structured output must be valid JSON and the code-based validator says it is not, there is little reason to tolerate regression. For LLM-judged semantic metrics such as groundedness or helpfulness, you usually need tolerances and review bands rather than brittle binary cutoffs. A small drop may warrant manual inspection instead of automatic rejection, especially if the candidate offers gains elsewhere such as latency or tool reliability.

A practical rule set often looks like this:

- fail if a hard-blocker metric drops beyond a fixed tolerance,
- fail if evaluator execution failure exceeds a ceiling,
- warn if a secondary metric regresses within a review band,
- require review if trade-offs cross a predefined threshold,
- pass only if all blocker metrics hold and no review trigger fires.

This style resembles scenario-weighted readiness frameworks used in broader AI quality practice, where correctness, safety, latency, cost, and groundedness are treated as distinct dimensions rather than merged into one opaque number. Phoenix gives you the experiment and trace primitives to implement that pattern even though it does not present a single built-in readiness harness abstraction.

### **Multi-metric gates are more honest than single scores**

A single aggregate score is attractive because it is easy to compare and easy to put in CI output. It is also the fastest way to hide dangerous regressions. If your application has more than one meaningful failure mode, you should expect more than one gating dimension.

For many LLM systems, a minimal release scorecard includes:

- one or more semantic quality metrics,
- one or more safety or policy-alignment metrics,
- latency distribution, often including p95,
- evaluator runtime health,
- cost or token-use indicators when relevant.

Latency belongs in the gate because user-perceived quality is partly temporal. A response that is more grounded but dramatically slower may not be operationally acceptable. Evaluator runtime health belongs in the gate because a candidate should not pass on the strength of missing evidence. Cost can be warning-only or review-triggered depending on whether the change is economically reversible.

Multi-metric does not mean endlessly many metrics. It means you define a small set of decision-relevant dimensions and keep them visible separately. If groundedness drops while helpfulness rises, the gate should show that tension explicitly rather than averaging it away.

### **Pre-merge, pre-release, and staged rollout gates serve different jobs**

A common mistake is to force every quality decision into one binary CI check. Release governance works better when you match the gate to the delivery stage.



Pre-merge checks should be fast and selective. They are good for small regression suites, deterministic validations, evaluator smoke tests, and critical failure slices that catch obvious breakage. They should not require a full expensive benchmark run for every minor edit unless the system is safety critical and the volume is manageable.

Pre-release validation can be broader. This is the natural place for richer Phoenix experiments over curated datasets with multiple evaluators and more complete latency or cost accounting. Because release cadence is slower than commit cadence, you can afford more compute and more human review here.

Staged rollout review sits between offline validation and full production exposure. At this stage, Phoenix traceability becomes especially useful. You can sample newly observed traces, apply evaluators or human review to them, and compare candidate behavior to the baseline under production-like conditions before broader promotion. This is often where rare slices emerge that the curated dataset underrepresents.

These layers should share vocabulary and baseline references even if they differ in depth. Otherwise you end up with one standard for merge checks, another for release, and a third for production triage, with no coherent governance story.

### **Gate debugging requires visibility into the evaluators**

When a candidate fails a quality threshold, teams often ask the wrong first question: “Is the release blocked?” The better first question is “Do we trust the signal and understand the failure?” Phoenix evaluator tracing is critical here because it exposes the evaluator prompt, inputs, reasoning, outputs, and timing. That makes gate debugging feasible.

Suppose groundedness drops on a release candidate. Without evaluator traces, you only see a number. With evaluator traces, you can in-

spect whether the judge prompt over-penalized paraphrase, whether retrieval context was malformed for a subset of examples, whether provider throttling increased evaluator instability, or whether the candidate actually generated unsupported claims. That difference determines whether you should fix the application, recalibrate the evaluator, or rerun the job.

Treat evaluator health as part of release health. If a gate fails because the evaluator pipeline itself is unstable, that is not the same operational event as a confirmed application regression. The policy response may still be to block promotion, but the remediation path is different. Record that distinction in CI output and in your run artifacts.

### **Artifacts should support rapid triage**

A blocked release is only useful if the evidence arrives with it. Phoenix CLI provides building blocks for that workflow, including terminal-driven datasets and experiments, plus exportable traces with annotations. A practical CI failure path can preserve experiment summaries and export the underlying traces with their annotations for immediate inspection.

```
px traces export --include-annotations
```

That exported artifact can be attached to the CI job, archived with the build, or forwarded into an incident or review system. The value is speed. When the gate fails, reviewers should not have to reconstruct the dataset, guess which examples moved, or search manually for the motivating traces. They should have the failing slice, the associated annotations, and enough metadata to inspect the evaluator run and candidate configuration.

This is also where prompt and dataset versioning meet release governance. If the candidate prompt version, dataset identity, evaluator suite, and exported evidence are all recorded together, exceptions be-

come auditable rather than informal. If you grant a waiver, you can later explain exactly what evidence was accepted and why.

### **Design the exception path before you need it**

No gate is perfect. There will be cases where a candidate should ship despite a warning, or where a blocker should be overridden because the measured regression is irrelevant to the deployment context. If you do not design the exception path up front, overrides happen through private messages and institutional memory.

A healthy exception process has three properties. It names the owner authorized to approve the exception. It records the rationale in relation to the specific metrics and artifacts that failed. It defines follow-up obligations, such as increased monitoring, rollback criteria, or a required patch window. That turns an override into a deliberate risk acceptance rather than a quiet erosion of standards.

Exceptions should be rare and visible. If they become routine, the gate is either poorly calibrated or politically unsupported. Both are serious system design problems.

### **Anti-patterns that undermine trust**

Several anti-patterns repeatedly weaken release gates.

One is gating on aggregate scores that conceal catastrophic performance on rare but critical slices. Another is changing datasets during an active release cycle, which destroys baseline comparability. A third is ignoring sample size or confidence considerations and making hard decisions on tiny, noisy samples. A fourth is relying on one LLM judge as the sole source of truth for a high-stakes decision. Even a well-crafted evaluator can drift, bias toward its own prompt framing, or mis-handle edge cases.

Another subtle anti-pattern is turning every metric into a blocker. That

usually produces alert fatigue, manual bypass habits, and quiet cynicism about the whole process. A gate should concentrate authority in the few signals that actually map to user harm or operational instability. Everything else should inform review, not dominate it.

### **A workable release checklist**

A compact checklist often performs better than an elaborate scoring formula. Before promoting a candidate prompt version or application revision, verify that:

- the experiment used a fixed, named dataset suite,
- blocker metrics held within tolerance,
- p95 latency stayed within acceptable bounds,
- evaluator failure rate remained below a defined ceiling,
- no critical failure slice regressed beyond its threshold,
- the candidate artifacts and evidence bundle were archived,
- any exception was explicitly approved and recorded.

This checklist style is blunt by design. It keeps the release decision legible to engineers, reviewers, and operators. It also reflects a practical truth: governance fails when only the most specialized people can understand the gate.

When Phoenix quality signals are wired into delivery this way, experimentation stops being a side activity and becomes part of the release contract. Candidates are promoted not because they feel better or because a few examples looked strong in local testing, but because the organization can point to specific datasets, evaluator outcomes, trace-linked evidence, and operational thresholds that justify the decision to ship.



## Chapter 8

# Self-Hosting Phoenix for Secure, Scalable Team Use

*A self-hosted Phoenix deployment changes the character of observability work: prompts, retrieval payloads, tool inputs, evaluator outputs, and user-linked trace metadata become durable operational data that must be run like any other internal platform. This chapter is about making that transition deliberately—choosing deployment and storage patterns that match team scale, then enforcing isolation, governance, and upgrade discipline so Phoenix remains useful rather than becoming a fragile side system.*

## 8.1. Choosing a Phoenix Deployment Mode from Workstation to Kubernetes

Phoenix often starts as a personal debugging surface. You run it beside an application, inspect a few traces, and close it when the immediate issue is resolved. That mode works until traces become shared operational evidence rather than private scratch data. The moment another engineer needs to inspect the same run, a prompt author wants to review failures, or a team decides that trace history should survive a laptop reboot, deployment stops being a convenience choice. You are no longer deciding how to launch a tool; you are deciding where an internal observability service belongs in your operating model.

A minimal local launch makes that distinction easy to miss because it is almost frictionless:

```
phoenix serve
```

A container launch is nearly as simple:

```
docker run \  
-p 6006:6006 \  
-p 4317:4317 \  
-i -t \  
arizephoenix/phoenix:latest
```

Both commands are valid starting points, but they imply very different habits once you care about persistence, shared access, and operational ownership. The deployment mode you choose should match five constraints: data sensitivity, expected concurrency, persistence requirements, reliability expectations, and the operational substrate your team already knows how to run well. If you ignore any one of those, Phoenix may still start successfully, but it will be misaligned with the way your organization actually uses trace data.

**Start with the operating question, not the packaging format.**

Teams often ask whether they should use the terminal, Docker, Compose, or Kubernetes as if those were purely ergonomic alternatives. They are not. They encode assumptions about who owns the runtime, how state survives, how configuration is managed, and how failures are recovered. A local CLI session assumes one operator, local trust, and low recovery requirements. A shared container on a single host assumes a lightly managed internal service. A Compose-style deployment usually signals a team sandbox with explicit service wiring and some level of persistent storage. Kubernetes assumes Phoenix is joining a broader internal platform with secrets, ingress, scheduled backups, policy controls, and repeatable deployment workflows.

That distinction matters because Phoenix has documented self-hosting paths across terminal, Docker, Kubernetes via kustomize, Helm, and additional hosted infrastructure patterns. The existence of many packaging options does not mean they are interchangeable. The practical dividing line is usually the storage and service model around Phoenix rather than the command used to start it. Local development and single-user operation map naturally to lightweight local storage and a workstation process. Production, multi-user, and highly available operation map to externalized, durable infrastructure and stronger runtime controls.

### **8.1.1. Local CLI mode: fast feedback, weak operational guarantees**

Local CLI operation is the right answer more often than teams admit. When you are iterating on instrumentation, validating trace shapes, or debugging a narrow problem on your own machine, the shortest path is usually best. A workstation-local Phoenix instance has almost zero coordination cost. You do not need a shared host, a cluster namespace, or a deployment ticket. You can restart freely, change environment variables ad hoc, and throw the whole environment away after the ex-



periment.

That speed has real technical value. Instrumentation work is exploratory. During early debugging, you want to answer questions such as whether spans are emitted at the right boundaries, whether retrieval metadata is present, or whether prompt and response payloads are being captured in a useful form. A personal Phoenix instance keeps that feedback loop tight.

**Where local mode fits.** Use local CLI mode when all of the following are true:

- One person is the primary consumer of the traces.
- The data is non-sensitive or already constrained to a developer-safe environment.
- Losing the stored traces is acceptable.
- The runtime only needs to exist for hours or days, not as a standing team service.
- You are still shaping instrumentation and do not yet need durable review workflows.

**Where it breaks.** Local mode fails gradually, which makes it dangerous. The first failure mode is ambiguous persistence. A developer may assume traces are durable because they survived a process restart, while another treats the instance as disposable. That mismatch creates false confidence around incident evidence and historical debugging. The second failure mode is collaboration. Screenshots and ad hoc exports do not scale into team debugging, especially when multiple people need to pivot through the same trace set. The third is operational drift: workstation defaults become de facto standards, and later someone tries to reproduce them in a shared environment without ever formalizing storage, secrets, or network exposure.

You should also treat laptop-local usage as a poor proxy for production privacy posture. A workstation environment often has broad outbound network access, weak secret discipline, and little separation between personal and team assets. Those shortcuts are tolerable for local experimentation but become liabilities if copied into a shared deployment.

**Decision signal.** If your main requirement is personal velocity and you can tolerate low durability, stay local. The moment you hear “can you keep that running for the team?” you have crossed into a different deployment class.

### **8.1.2. Single-container deployment: shared access with minimal platform overhead**

A single container on a shared host is usually the first serious step beyond local use. It preserves much of Docker’s operational simplicity while giving the team a stable endpoint and a runtime that is not tied to one engineer’s laptop. This model works well for a small group that wants a central debugging console without taking on full cluster operations.

The important design question is not whether Docker can run Phoenix; it clearly can. The real question is whether the host itself is being treated as durable internal infrastructure or as a disposable sandbox. That answer changes almost every operational decision around it.

A typical production-minded container launch begins to externalize configuration:

```
docker run \  
-p 6006:6006 \  
-p 4317:4317 \  
-e PHOENIX_ENABLE_AUTH=true \  
-e PHOENIX_SECRET=replace-with-strong-secret \  
-e PHOENIX_SQL_DATABASE_URL=\  
postgresql://phoenix:secret@db:5432/phoenix \  
-i -t \  

```

```
arizephoenix/phoenix:latest
```

This example is intentionally plain. It demonstrates the actual control surface you use in self-hosted Phoenix: environment variables. It also reveals the central operational shift from local mode. Once you inject authentication settings and a database URL, Phoenix is no longer “just a container”; it is a service with state, credentials, and a dependency graph.

**When single-container deployment is the right fit.** Choose this mode when you need a shared endpoint for a small team, have one host that the team or platform group can manage, and do not yet need the scheduling, policy, or rollout mechanics of Kubernetes. It is a strong fit for an internal lab, a small applied ML team, or a departmental observability service where uptime matters but cluster-level automation would be disproportionate.

**Host ownership matters more than the image choice.** A single-container Phoenix deployment inherits every property of its host. If the host is manually administered, backups are informal, and container upgrades are done by shell history, then Phoenix will be fragile regardless of how polished the container image is. If the host is patched, monitored, backed up, and integrated into internal networking standards, the same single-container pattern can be perfectly adequate for moderate team use.

This is why “Docker versus Kubernetes” is often the wrong framing. The more meaningful comparison is “single managed host versus managed cluster.” If your organization already runs trusted internal services on a hardened VM or bare-metal node, Phoenix can fit there cleanly.

**Network exposure and trust boundaries.** A single-container deployment should not be treated as safely exposed just because it sits

inside the corporate network. Internal networks are still large trust domains. If Phoenix is carrying prompts, retrieved documents, user metadata, or evaluation outputs, you should decide explicitly whether it belongs on a private subnet, behind a reverse proxy, or within a restricted VPN-only segment. Public exposure without upstream controls is a clear anti-pattern. The convenience of binding to host ports does not remove the need for ingress policy, TLS termination, and secret handling.

**Persistence is the real fork in the road.** A single container can be either disposable or durable. Disposable means the instance is there for temporary shared work and can be rebuilt with little consequence. Durable means users expect trace history, authentication, and service identity to survive host restarts and upgrades. The difference should be made explicit in your runbook, because the operating habits diverge immediately. Disposable environments favor quick rebuilds and short retention. Durable ones require an external database or, at minimum, well-defined persistent storage and backup procedures.

### 8.1.3. Compose-style environments: the practical team sandbox

Compose-style deployment occupies an important middle ground even when Phoenix documentation does not elevate it as a separate product surface. Teams use it because it maps neatly to the way they think: one file, multiple services, explicit environment injection, and persistent volumes where needed. In practice, this is often the most understandable way to stand up a small shared Phoenix environment with a database and predictable network wiring.

A representative layout might look like this:

```
services:
  phoenix:
    image: arizephoenix/phoenix:latest
```

```
ports:
- "6006:6006"
- "4317:4317"
environment:
PHOENIX_ENABLE_AUTH: "true"
PHOENIX_SECRET: "replace-with-strong-secret"
PHOENIX_SQL_DATABASE_URL: >-
postgresql://phoenix:secret@postgres:5432/phoenix
depends_on:
- postgres

postgres:
image: postgres:16
environment:
POSTGRES_USER: phoenix
POSTGRES_PASSWORD: secret
POSTGRES_DB: phoenix
```

This is not about Compose as a brand preference. It is about expressing a small service topology clearly: Phoenix, a PostgreSQL backend, and the configuration that binds them together. For many teams, that is enough structure to move from “someone’s debug container” to “a shared internal service.”

**Why this mode works well for small teams.** Compose-style environments make hidden decisions visible. You can see which variables are injected, where the database lives, and which ports are exposed. That visibility is valuable when a team is still converging on ownership. It encourages discussion about whether storage is local to the host or external, whether the stack is meant to persist, and who is responsible for restarting or upgrading it.

This mode is especially useful when your team wants shared Phoenix access but does not have a cluster platform or does not want to wait for one. It gives you enough structure to practice good habits: named services, explicit database configuration, isolated service networks, and a path to externalize secrets later.

**Its limits are organizational, not merely technical.** Compose-

style deployment becomes awkward when multiple teams need different lifecycles, when upgrades must be standardized across environments, or when secret and ingress management need to conform to central platform policy. It also tends to centralize too much responsibility in one host owner. That is acceptable for a team sandbox, but less so when Phoenix becomes a cross-team dependency.

You should resist the temptation to use Compose as a forever platform simply because it worked for the first few users. It works best as deliberate small-scale infrastructure: stable enough to share, simple enough to reason about, and easy to retire once requirements exceed the host-based model.

**Decision signal.** Choose a Compose-style deployment when you want a shared service with explicit service wiring, likely including PostgreSQL, but you still want the full stack to fit inside a small team's operational envelope.

#### **8.1.4. Kubernetes deployment: platform integration, not automatic virtue**

Kubernetes becomes justified when Phoenix needs to behave like any other internal service your platform team already operates: declarative deployment, secret separation, controlled ingress, reproducible promotion between environments, integration with managed PostgreSQL, and consistent backup and monitoring practices. Phoenix supports Kubernetes deployment paths through kustomize and Helm, which gives you two operational styles: direct manifest control or packaged chart-based installation.

The official entry points are straightforward:

```
kubectl apply -k kustomize/base
```

```
CHART_URL=oci://registry-1.docker.io/\
```

```
arizephoenix/phoenix-helm

helm install phoenix \
  $CHART_URL \
  --version $CHART_VERSION
```

Those commands are easy to copy, but Kubernetes only pays for itself when the surrounding platform features matter to you. A cluster does not make Phoenix more correct by default. It gives you a better place to express controls you already know you need.

**What Kubernetes buys you.** The strongest reasons to choose Kubernetes are consistency and integration. You can manage secrets through native secret objects or external secret operators, place Phoenix behind standard ingress controllers, apply network policy, attach persistent volumes or managed databases, and align upgrades with normal release engineering. If your organization already runs internal tooling this way, Phoenix benefits from joining that substrate instead of inventing a one-off host pattern.

Kubernetes also improves repeatability across environments. If you need separate nonproduction and production Phoenix instances, or multiple instances for different trust zones, manifests and Helm values become a practical way to standardize deployment without handcrafting each environment.

**What Kubernetes does not solve for you.** Kubernetes does not eliminate the need to decide whether Phoenix is disposable or durable. It does not remove database planning, retention policy decisions, or access-control design. It also does not justify itself for a team of three people who only need a shared debugging UI twice a week. If your platform does not already have cluster expertise, using Kubernetes for Phoenix alone can be expensive in attention, not just in infrastructure cost.

Premature cluster adoption creates a subtle anti-pattern: teams inherit

the operational weight of Kubernetes without actually needing its controls. The result is often a fragile deployment that is formally sophisticated but practically under-owned.

**A useful threshold test.** Kubernetes is a strong fit when several of these are true:

- Multiple teams will depend on the service.
- You need standardized secrets, ingress, and network controls.
- The environment must be reproducible across dev, staging, and production.
- Phoenix should coexist with managed PostgreSQL and existing backup regimes.
- A platform team already operates the cluster and expects services to integrate with it.

If most of those are false, a simpler host-based deployment is often the better engineering choice.

**Choose the simplest mode that satisfies your real constraints.**

A practical selection flow is more reliable than arguing abstractly about deployment sophistication:

- Classify the trace data. If prompts, retrieved documents, or meta-data are sensitive, rule out casual workstation or broadly exposed host patterns.
- Estimate concurrency. One engineer debugging alone is fundamentally different from a shared team console with continuous ingestion.



- Decide whether the environment is disposable or durable. If users expect trace history to survive failures and upgrades, treat it as service infrastructure immediately.
- Align with your standard operations substrate. If your team already manages internal services on a hardened host, use that. If you already run everything through Kubernetes, Phoenix should likely join it.
- Pick the least complex deployment mode that still satisfies privacy, durability, and operability needs.

This flow deliberately favors under-complexity rather than over-engineering. You can move from local to container to Compose-style to Kubernetes as requirements accumulate. The reverse path is much harder because people get attached to platform choices even when those choices exceed the problem.

**Common anti-patterns.** Three mistakes appear repeatedly.

First, teams promote local defaults into shared infrastructure without ever naming the shift. That produces unclear persistence, weak secret handling, and no real ownership boundary.

Second, teams expose Phoenix too broadly because it feels like an internal developer tool rather than a system carrying rich application traces. Internal does not mean low-risk.

Third, teams choose Kubernetes for organizational fashion rather than operational need. If the cluster gives you no meaningful advantage beyond perceived modernity, it is probably the wrong first deployment mode.

The most durable Phoenix deployments are rarely the most elaborate ones. They are the ones whose runtime, storage expectations, and access assumptions match the way the team actually intends to use trace

data day after day.

## 8.2. Designing Storage for Durable, Multi-User Phoenix Operations

Teams usually discover the storage problem only after Phoenix becomes useful. A workstation instance that felt perfectly adequate for one engineer starts receiving traces from several services, other people begin querying older runs, and someone eventually asks a question that exposes the gap: “Where is this data actually stored, and how would we recover it if the host disappears?” At that point, storage is no longer an implementation detail. It determines whether Phoenix is a convenient viewer or a dependable internal system.

A concrete configuration makes the boundary clear. For local, single-user use, Phoenix can run with its default storage posture. For shared and durable use, you should point it at PostgreSQL explicitly:

```
export PHOENIX_SQL_DATABASE_URL=\
postgres://phoenix:secret@db:5432/phoenix

phoenix serve
```

Phoenix supports exactly two documented backends for self-hosted storage: SQLite and PostgreSQL. That narrow backend surface is helpful because it forces a clean decision. SQLite is the default, no-setup path and is appropriate for local or single-user storage. PostgreSQL is the production backend for durable, multi-user operation. Everything else in your deployment design follows from that choice: concurrency behavior, backup mechanics, upgrade handling, retention, and the size of the failure domain.

**Storage changes the meaning of availability.** A running Phoenix process does not mean Phoenix is operationally trustworthy. You can

have a healthy web UI and a reachable ingestion endpoint while still lacking recoverability, backup coverage, or acceptable multi-user behavior. Observability systems are particularly vulnerable to this illusion because they often begin life as debugging utilities. Teams underestimate the value of the stored data until they need historical traces to understand a regression, compare prompt changes, or audit a production incident.

That is why storage deserves its own decision process. The question is not merely whether Phoenix can save traces somewhere. The question is whether the chosen storage mode matches your expected write rate, read concurrency, retention window, and incident tolerance.

### **8.2.1. From convenient local storage to production persistence**

SQLite is attractive for the same reason local Phoenix is attractive: it minimizes setup. You do not provision a separate database service, rotate another credential, or coordinate with a platform team. For initial instrumentation work, those are real advantages. The storage is local, the mental model is simple, and the cost of experimentation stays low.

**Why SQLite works early.** SQLite is a good fit when you need a single-user environment for exploratory trace inspection, schema familiarity, or short-lived testing. It keeps the operational surface small enough that you can focus on what matters most in the beginning: whether spans, attributes, retrieval metadata, and evaluation outputs are arriving in the right shape.

For a solo workflow, that simplicity is often worth more than theoretical scalability. You do not need network connectivity to an external database, and you avoid the failure modes that come from provisioning mistakes or misconfigured credentials. If you are validating instrumen-

tation on a laptop or a disposable development host, SQLite is often the right choice precisely because it is boring.

**Why SQLite stops being enough.** The same properties that make SQLite convenient also make it brittle as team infrastructure. Once Phoenix becomes shared, you need to think about simultaneous readers, continuous ingestion, longer retention, and recovery after host loss. Embedded local storage ties the application lifecycle too closely to the host lifecycle. If the machine fails, is rebuilt, or is repurposed, your trace history may disappear with it unless you have already designed around that risk.

The issue is not that SQLite is inherently bad technology. The issue is mismatch. A backend optimized for low-friction local use is not the right foundation for a service expected to support multiple people, retain history, and survive routine infrastructure events. Treating it as such usually leads to vague backup stories, lock contention under heavier use, and uncertainty about which traces can be trusted to still exist tomorrow.

**The transition is a maturity move, not a feature toggle.** Moving from SQLite to PostgreSQL should not be framed as “switching databases” in the abstract. You are changing the operating model of Phoenix. With PostgreSQL, storage becomes a managed service boundary rather than a file attached to an application process. That lets you separate concerns: Phoenix operators manage the application, while database operators or managed database services handle replication, backups, patching, and high availability according to your standards.

### 8.2.2. What PostgreSQL changes in practice

Production Phoenix should use PostgreSQL, configured either through `PHOENIX_SQL_DATABASE_URL` or the split `PHOENIX_POSTGRES_*` envi-

environment variables. The URL form is usually the clearest way to express the connection in containerized or orchestrated deployments:

```
export PHOENIX_SQL_DATABASE_URL=\
postgres://phoenix:secret@postgres:5432/phoenix
export PHOENIX_DEFAULT_RETENTION_POLICY_DAYS=30

phoenix serve
```

This example does two useful things. It externalizes the storage backend, and it places retention next to connectivity, where it belongs operationally. Once traces live in a durable database, retention becomes a first-class cost and governance control rather than an afterthought.

**Concurrency and correctness.** Phoenix is not just a read-heavy dashboard. It ingests spans and trace metadata while people query, filter, and inspect those records. In a team environment, this creates a mixed workload: concurrent writes from instrumented applications and concurrent reads from engineers, analysts, or evaluators. PostgreSQL is the right backend for that pattern because it is designed to support shared transactional workloads with predictable durability semantics and mature operational tooling.

For Phoenix specifically, this matters less as a benchmark contest and more as a correctness issue. Multi-user systems need stable behavior when several clients are active at once. You do not want trace ingestion to become erratic simply because someone is running a broad query in the UI, and you do not want read access patterns to be constrained by the limitations of workstation-local storage.

**Recoverability and backup posture.** The moment Phoenix becomes a team service, you need a real answer to two recovery questions:

- What is the recovery point objective for trace data?
- How do you restore the service after host or environment loss?

A local file or engine-level named volume can preserve data across process restarts, but that is not the same thing as a backup strategy. Docker named volumes are useful persistence primitives. They survive container recreation and help avoid accidental data loss during routine restarts. They do not, by themselves, give you off-host backups, tested restores, point-in-time recovery, or cross-host failover. Treat them as building blocks, not as evidence that durability is solved.

That distinction is especially important in team sandboxes. A Compose stack backed by a named volume may be perfectly reasonable for local durability on a shared host, but it still leaves open the most important production questions: where are backups stored, how often are they taken, who validates restores, and how much data loss is acceptable?

**Schema migration and operational trust.** Persistent databases introduce another responsibility: schema change management. Phoenix upgrades can carry migrations, and durable storage means those migrations now operate on data that multiple users depend on. That makes backup timing, restore testing, and version coordination part of storage design, not just application deployment. A storage design is incomplete if it explains where data lives but not how schema changes are protected by backup and rollback discipline.

### 8.2.3. Storage patterns that map cleanly to Phoenix use

You can think of Phoenix storage in three practical stages.

**Stage 1: local SQLite for development.** Use local default storage when the environment is personal, low-risk, and disposable. This stage optimizes for speed of setup and freedom to experiment. Do not normalize it as team infrastructure. Document it as local development only.

**Stage 2: Docker volume-backed persistence for small shared**

**work.** A single-host environment with Docker volumes can provide local durability across restarts. This is useful for short-lived team sandboxes or shared evaluation sessions where trace history should survive container replacement but does not yet justify a full production database service.

A Compose-style stack often looks like this:

```
services:
  phoenix:
    image: arizephoenix/phoenix:latest
    environment:
      PHOENIX_SQL_DATABASE_URL: >-
      postgresql://phoenix:secret@postgres:5432/phoenix
      PHOENIX_DEFAULT_RETENTION_POLICY_DAYS: "7"
    depends_on:
      - postgres

  postgres:
    image: postgres:16
    environment:
      POSTGRES_USER: phoenix
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: phoenix
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

This is a sensible small-team pattern because it externalizes storage from the Phoenix container and introduces explicit retention. It is still host-scoped infrastructure. If the host fails and the volume is not backed up elsewhere, you have durability only within that host's lifetime.

### Stage 3: managed or platform PostgreSQL for durable team

**use.** When Phoenix becomes a standing service, move the database to a managed PostgreSQL instance or an internally operated HA PostgreSQL service. At that point, Phoenix and its database should fail independently, be backed up according to platform policy, and follow

your normal restore and patching procedures. In Kubernetes environments, this usually means Phoenix connects to a PostgreSQL service that already participates in your backup and monitoring ecosystem rather than embedding database lifecycle inside the same application release process.

This stage is where durable multi-user expectations become realistic. You can define retention intentionally, align database backups with business recovery objectives, and avoid coupling trace durability to the lifecycle of a single node.

**A useful rule of thumb.** If you would be uncomfortable losing a week of traces, you are already beyond the point where implicit local persistence is enough.

#### 8.2.4. Retention is part of storage design

Storage design is not only about where bytes live; it is also about how long they remain valuable and acceptable to keep. Phoenix 9.0 and greater can automatically purge traces by time or trace count. At the deployment level, `PHOENIX_DEFAULT_RETENTION_POLICY_DAYS` sets the default time-based retention policy, and 0 means indefinite retention.

You should treat this as both a capacity control and a governance control. Teams often postpone retention decisions because storage initially feels cheap. That works until query performance degrades, backups inflate, or the data turns out to be more sensitive than the original debugging use case suggested.

A practical policy split often looks like this:



| Environment | Typical purpose                          | Default retention                          |
|-------------|------------------------------------------|--------------------------------------------|
| Sandbox     | Short-lived debugging, prompt iteration  | 7 days                                     |
| Shared dev  | Team debugging and evaluation review     | 14–30 days                                 |
| Production  | Incident analysis, operational forensics | Policy-driven; avoid “infinite” by default |

A corresponding configuration for a sandbox environment is straightforward:

```
export PHOENIX_DEFAULT_RETENTION_POLICY_DAYS=7
phoenix serve
```

And for environments where policy truly requires indefinite retention:

```
export PHOENIX_DEFAULT_RETENTION_POLICY_DAYS=0
phoenix serve
```

Use indefinite retention only when you can justify the storage growth, the backup cost, and the data-governance implications. “We did not want to think about deletion yet” is not a production policy.

**Retention and backup must agree.** A common operational mistake is to define a short logical retention policy while keeping long-running backups or snapshots indefinitely. That creates a mismatch between application behavior and actual recoverable data. If you purge traces after 30 days but your backup system can restore six months of history, your effective retention is six months. Coordinate Phoenix retention with database backup retention so the data lifecycle is coherent.

### 8.2.5. Separate application ownership from database ownership

Phoenix operators do not need to become full-time database administrators, but they do need to make the ownership boundary explicit. In mature environments, the split usually looks like this:

- Phoenix owners define connection settings, retention defaults, upgrade timing, and service-level expectations.
- Platform or database owners provide PostgreSQL availability, backups, patching, monitoring, and restore procedures.
- Security or governance stakeholders define retention classes and acceptable handling for trace data.

This separation prevents a common failure mode in internal tooling deployments: the application team assumes the database is covered by platform policy, while the platform team assumes the application data is disposable because it originated as a debugging tool. Put the assumptions in writing before adoption expands.

**Phoenix-specific sizing intuition.** Trace storage is not ordinary low-cardinality metrics data. Prompts, retrieved context, tool inputs, outputs, and evaluation artifacts can be semantically dense and sometimes large. That means storage growth depends not only on trace count but also on payload richness. Two services generating the same number of spans may create very different storage footprints if one captures compact metadata and the other stores long prompt and retrieval bodies. Forecast growth using real representative traces, not abstract event counts.

**Anti-patterns to avoid.** Three storage mistakes recur often enough to name directly.

Running shared team usage on SQLite indefinitely is the first. It usually starts as pragmatism and ends as silent fragility.

Treating named volumes as a complete durability strategy is the second. They are useful, but they do not replace backup, restore testing, or off-host recovery planning.

Ignoring retention until the database is already bloated is the third.

Trace systems accumulate value quickly, but they also accumulate cost and risk quickly if you never decide what deserves to persist.

Storage is where Phoenix stops being merely available and starts becoming dependable. Once traces matter to more than one person and survive longer than one debugging session, the right database, retention posture, and recovery plan determine whether the service can keep its promises under normal use and under failure.

### 8.3. Structuring Topology, Isolation, and Horizontal Scale

A single shared Phoenix instance is often the right first answer and the wrong long-term assumption. Once multiple teams send traces, different environments produce materially different data, or production debugging requires tighter controls than experimentation, the hard question is no longer how to deploy Phoenix. It is how many Phoenix instances you should run, which data belongs together, and where you want failure domains to stop.

A concrete Kubernetes rollout pattern captures the operational shape of a mature deployment. Phoenix’s official production path on Kubernetes assumes PostgreSQL plus persistent storage, and upgrades should gate application startup on schema migration:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: phoenix
spec:
  replicas: 2
  strategy:
    rollingUpdate:
      maxSurge: 0
      maxUnavailable: 1
  template:
    spec:
```

```
initContainers:
  - name: migrate
    image: arizephoenix/phoenix:latest
    command: ["phoenix","db","migrate"]
containers:
  - name: phoenix
    image: arizephoenix/phoenix:latest
    command: ["phoenix","serve"]
```

That example is useful even before you care about Kubernetes syntax. It reveals three topology facts that matter regardless of packaging: the database is shared state, application replicas are not fully independent of it, and rollout behavior becomes part of system design once more than one replica or one environment exists. Horizontal scale in Phoenix is therefore not just “add more pods.” It is bounded by shared storage semantics, migration sequencing, and the trust boundaries around the data you ingest.

**Start with isolation goals, not replica counts.** Teams often ask about scale too early and isolation too late. For Phoenix, the cleaner order is the reverse. Decide first whether the same instance should hold development traces and production traces, whether different teams can share operational visibility safely, and whether one upgrade or incident should affect everyone at once. Only then does replica count become meaningful.

That ordering matters because many operational problems that look like scale problems are actually topology problems. If one team’s high-volume development ingestion makes production trace queries sluggish, the first fix is often environment separation rather than more app replicas. If two business units require different retention or different administrative ownership, the issue is not CPU saturation; it is an ownership boundary that should be reflected in deployment boundaries.

### 8.3.1. When one Phoenix instance is still enough

One instance remains a good design when trust assumptions are aligned, data sensitivity is similar across users, and the expected write and read load is modest. A small engineering group sharing one nonproduction environment often fits this pattern well. Operationally, the advantages are obvious: one service to patch, one backup policy, one ingress path, and a single place to search traces.

**Use one instance when the operational story is simple.** A shared instance is usually sufficient when these conditions hold:

- The same team owns most instrumented services.
- Development and production data are not mixed, or production ingestion is absent.
- Retention and access expectations are broadly uniform.
- A single maintenance window or outage is acceptable for all users.
- Trace volume is low enough that one database-backed deployment remains easy to operate.

This is the best place to start because it minimizes administrative sprawl. Every new Phoenix instance creates more upgrades, more secrets, more backup scopes, and more chances for configuration drift. If one shared instance genuinely fits your trust and workload model, preserve that simplicity.

**Breaking points appear as policy mismatches.** The first signs that one instance is no longer enough are rarely raw throughput alarms. More often, you see policy tension. Production teams do not want experimental users browsing sensitive traces. One group needs longer

retention than another. A platform team wants stricter network controls for workloads that can reach user-configurable providers or URLs. Or a rollout that should have been routine becomes risky because too many consumers depend on the same runtime.

Those are topology triggers. They indicate that the blast radius of a single Phoenix instance has outgrown the shared assumptions that once made it attractive.

#### 8.3.2. Common topology patterns

You can organize Phoenix instances along several axes. The right one depends on what you are trying to isolate: operational lifecycle, data sensitivity, administrative ownership, or failure impact.

**One instance per environment.** This is the most common first expansion. Separate development from production once trace sensitivity, reliability expectations, or operational ownership diverge. The main benefit is clean lifecycle separation. You can restart, upgrade, or experiment in development without creating production risk. You also avoid the anti-pattern of using one Phoenix instance as a universal sink for every environment.

The drawback is duplication. You now have at least two ingress points, two deployment objects, two sets of credentials, and potentially two retention policies. That cost is usually worth paying because environment separation maps directly to how teams already think about software risk.

**One instance per team or business unit.** Split by team when ownership is the dominant concern. This works well when groups operate independently, have materially different trace volumes, or need freedom to evolve instrumentation and retention without negotiating every change with unrelated users. It also reduces noisy-neighbor effects. A

high-ingest experimentation team will not distort the experience of a low-volume but operationally critical team.

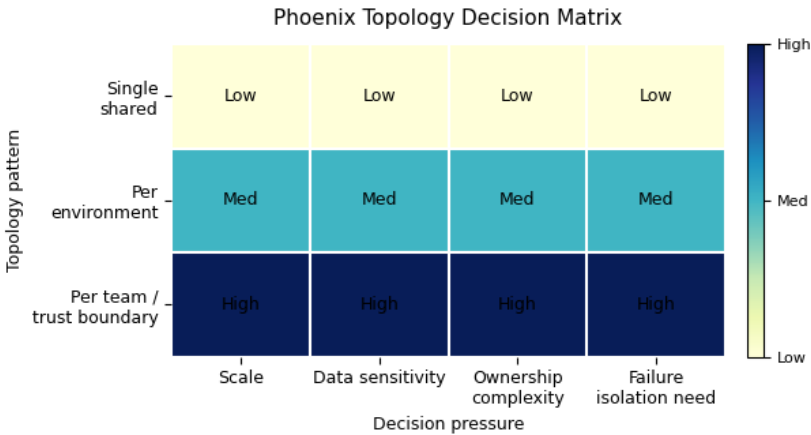
The downside is administrative multiplication. Separate team instances require consistent provisioning and upgrade workflows, or you will accumulate many slightly different Phoenix environments that nobody fully understands. Team-based splitting therefore only works well once deployment automation is strong enough to make each additional instance boring.

**One instance per trust boundary.** Choose trust-boundary separation when the central issue is not convenience but risk containment. This applies when contractual, regulatory, or internal security requirements prevent broad sharing of trace data. It also applies when different instances must follow materially different egress policies. Phoenix operators should pay close attention to this because isolation is not just a namespace decision. User-configurable providers or URLs can create outbound-call risk, so network policies, firewalls, or proxies are part of the trust boundary, not optional decorations around it.

Trust-boundary splitting is the cleanest way to ensure that traces, credentials, and network behavior obey the same security assumptions. It is also the most expensive topology because the separation is usually enforced at multiple layers: network, ingress, secrets, operational ownership, and often database isolation.

**Multiple app replicas behind one logical instance.** Horizontal scale within one instance is different from multi-instance topology. Replicas can improve availability and absorb moderate request concurrency, but they do not create isolation by themselves. All replicas still depend on the same shared database and the same surrounding controls. If your problem is mixed trust levels or conflicting retention, adding replicas solves nothing. If your problem is brief pod loss during node maintenance, replicas help.

That distinction matters because many teams overestimate what horizontal replication buys them. Phoenix’s app container may be operationally close to stateless, but the service as a whole is not. The shared database, migration ordering, and outbound access controls remain centralized constraints.



### 8.3.3. Implementation progression that keeps topology manageable

The cleanest topology evolution is incremental. Start with a shared nonproduction instance. Add a separate production instance when trace sensitivity, uptime expectations, or operational review practices diverge from development. Split further by team or trust boundary only when concrete policy or blast-radius concerns appear. That progression keeps the number of instances coupled to real constraints rather than to architectural fashion.

**Stage 1: one shared nonproduction instance.** This stage optimizes for speed of onboarding. Instrument services, validate trace quality, and establish operating habits. If you are using Kubernetes,



keep the deployment simple and database-backed. A single replica is often enough at first, because the important question is not pod count but whether the instance becomes a useful common debugging surface.

**Stage 2: separate production from development.** This is usually the first mandatory split. Production traces often carry more sensitive context, stricter retention expectations, and different availability requirements. Keeping production isolated also lets you run looser experimentation in development without risking the operational evidence you depend on during incidents.

At this point, standardize deployment inputs aggressively: image source, database configuration, secret handling, ingress policy, and rollout mechanics. If every environment is handcrafted, the second instance is where operational entropy begins.

**Stage 3: split by ownership or trust only when justified.** Additional instances should exist for a reason you can name in one sentence. “We need separate production and development” is a good reason. “Finance traces cannot share an observability boundary with general product teams” is another. “Our organization likes micro-platforms” is not.

When you do split further, automate provisioning at the same time. The danger is not the extra instance itself; it is the manual uniqueness that tends to accumulate around each one.

**Shared databases versus fully separated backends.** You may be tempted to share a PostgreSQL service across multiple logical Phoenix instances while keeping data separated at the application level. This can be operationally efficient, but it also couples availability, maintenance windows, and database-level incidents. If your isolation goal is blast-radius reduction, sharing too much of the persistence layer may defeat the point. If your main goal is administrative convenience and the trust boundary remains common, shared database infrastructure can be reasonable. Decide

which property matters more: efficiency or independence.

#### 8.3.4. Horizontal scale and rollout sequencing

Horizontal scaling matters, but not in the simplistic web-tier sense. Phoenix replicas can improve resilience against node loss and support concurrent users, yet the main operational hazards are rollout coordination and migration behavior.

**Why migrations gate scaling.** Phoenix recommends running `phoenix db migrate` in an `initContainer` before `phoenix serve` starts. Init containers run to completion before the main application container starts, which makes them the right place to gate schema advancement. Without that gate, a long migration during a rolling update can cause liveness probe failures and trigger `CrashLoopBackoff` while the application tries to start against an incomplete schema state.

A more complete deployment sketch looks like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: phoenix
spec:
  replicas: 2
  strategy:
    rollingUpdate:
      maxSurge: 0
      maxUnavailable: 1
  template:
    spec:
      initContainers:
        - name: migrate
          image: arizephoenix/phoenix:latest
          command: ["phoenix", "db", "migrate"]
      containers:
        - name: phoenix
          image: arizephoenix/phoenix:latest
          command: ["phoenix", "serve"]
```

The rolling update settings matter here. Using `maxSurge: 0` and `maxUnavailable: 1` rolls one pod at a time and reduces concurrent migration noise. Phoenix documents that concurrent migration attempts from multiple replicas are safe, but one pod may fail transiently and retry while another completes the migration. Safe does not always mean quiet. If you want cleaner rollouts, serialize them or pre-run migrations with a Job.

**A common failure mode.** Consider a two-replica deployment pointed at a large PostgreSQL database during a major upgrade. Both new pods start. Each init container tries to run `phoenix db migrate`. One succeeds first; the other may fail or block, then retry. If liveness and readiness timings are too aggressive, the app container may never get a calm startup window. Operators often misread this as an application crash when the real issue is rollout choreography around shared schema state.

**When more replicas help.** Additional Phoenix replicas are useful when you need better availability during node disruptions, expect multiple active UI users, or want more stable service capacity during restarts and rolling changes. They are less useful when the bottleneck is database throughput, migration duration, or outbound network control. If those are your binding constraints, scaling the app layer alone yields little benefit.

**Egress control is part of scale design.** As adoption grows, more users and integrations increase the number of outbound operations Phoenix may mediate. Because user-configurable providers or URLs can trigger external calls, horizontal growth can amplify outbound risk as well as inbound load. That is why network policies, firewalls, or proxies belong in topology planning. They are not merely security add-ons; they define whether scaling preserves your trust assumptions.

**Operational heuristics that work well.** A few rules keep Phoenix

topology sane as usage grows:

- Split by environment before you split by team.
- Split by trust boundary before you split for cosmetic organizational symmetry.
- Add app replicas for availability, not as a substitute for isolation.
- Treat database migrations as rollout events, not background details.
- Standardize deployment automation before multiplying instances.

These heuristics favor clarity over theoretical elegance. A topology that operators can reason about during an incident is usually better than one that maximizes abstraction.

**Anti-patterns to avoid.** Using one Phoenix instance as a universal sink across development, staging, and production is the most common topology mistake. It looks efficient until retention, access, and outage expectations diverge.

Creating many per-team instances without automation is the second. The administrative cost compounds quickly, and upgrades become a scavenger hunt across snowflake environments.

Assuming horizontal scaling solves all future growth is the third. In Phoenix, database behavior, migration coordination, and trust boundaries dominate much of the real complexity. App replicas matter, but they are only one piece of the system.

A well-structured Phoenix topology makes instance count, isolation boundaries, and rollout behavior predictable enough that growth feels incremental rather than architectural. When those decisions line up

with ownership, sensitivity, and failure domains, the system scales in a way that remains understandable under load and during change.

## 8.4. Operating Authentication, RBAC, and Trace Data Governance

Trace data becomes sensitive faster than most teams expect. A single Phoenix trace may contain raw user prompts, retrieved passages from internal documents, tool inputs carrying account identifiers, model outputs with business logic implications, and metadata rich enough to reconstruct user intent. Once that data is centralized, you are not operating a harmless debugging dashboard. You are operating a governed internal data system that happens to present itself as observability tooling.

A secure baseline starts with explicit authentication, durable secret handling, and a finite retention posture. In Kubernetes, that usually means keeping signing material and database credentials out of inline manifests and injecting them through secrets:

```
apiVersion: v1
kind: Secret
metadata:
  name: phoenix-secrets
type: Opaque
stringData:
  PHOENIX_SECRET: "replace-with-strong-secret"
  PHOENIX_API_KEY: "replace-with-api-key"
  PHOENIX_SQL_DATABASE_URL: >-
    postgresql://phoenix:secret@postgres:5432/phoenix
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: phoenix
spec:
  template:
    spec:
      containers:
```

```
- name: phoenix
  image: arizephoenix/phoenix:latest
  env:
    - name: PHOENIX_ENABLE_AUTH
      value: "true"
    - name: PHOENIX_DEFAULT_RETENTION_POLICY_DAYS
      value: "30"
    - name: PHOENIX_TELEMETRY_ENABLED
      value: "false"
    - name: PHOENIX_SECRET
      valueFrom:
        secretKeyRef:
          name: phoenix-secrets
          key: PHOENIX_SECRET
    - name: PHOENIX_API_KEY
      valueFrom:
        secretKeyRef:
          name: phoenix-secrets
          key: PHOENIX_API_KEY
    - name: PHOENIX_SQL_DATABASE_URL
      valueFrom:
        secretKeyRef:
          name: phoenix-secrets
          key: PHOENIX_SQL_DATABASE_URL
```

This configuration captures the minimum governance posture you should expect in a shared environment. Authentication is enabled with `PHOENIX_ENABLE_AUTH=true`. A strong `PHOENIX_SECRET` signs JWTs and must be treated like any other sensitive signing key. The deployment-wide default retention is explicit rather than accidental. UI analytics are disabled with `PHOENIX_TELEMETRY_ENABLED=false` when your privacy posture requires it. None of these controls are exotic. They are the price of admitting that traces are governed data, not disposable developer residue.

**Treat trace data as semantically dense internal data.** The usual mental model for logs is too weak here. Conventional logs often record technical symptoms: status codes, latencies, stack traces. Phoenix traces can instead capture intent, context, and decision inputs. In LLM and retrieval systems, that difference is substantial. A prompt may contain private user text. Retrieved context may quote proprietary knowl-

edge bases. Tool invocations may reveal identifiers, financial state, or workflow transitions. Intermediate spans may preserve exactly the information a motivated insider would want to search.

That is why governance starts with classification, even if the classification is simple. Decide whether Phoenix traces from a given environment are development exhaust, internal confidential data, or restricted operational records. The category determines everything downstream: who may log in, what roles they need, how long traces remain, whether redaction is required before ingestion, and what network boundaries must exist around the service.

#### 8.4.1. Authentication defines the first trust boundary

Authentication is not merely a feature toggle for the UI. It establishes whether Phoenix participates in your identity perimeter or sits outside it as a sidecar web app with informal credentials. In any shared deployment, choose the former deliberately.

**Enable auth early, not after adoption.** Phoenix enables authentication with `PHOENIX_ENABLE_AUTH=true` and a strong `PHOENIX_SECRET`. If you postpone auth until after users and integrations have already formed habits, you will end up retrofitting identity under load, which is precisely when exceptions and shortcuts multiply. Turn it on before the instance becomes widely shared.

A simple container launch still needs explicit secrets:

```
docker run \
  -p 6006:6006 \
  -p 4317:4317 \
  -e PHOENIX_ENABLE_AUTH=true \
  -e PHOENIX_SECRET=replace-with-strong-secret \
  -e PHOENIX_DEFAULT_RETENTION_POLICY_DAYS=30 \
  -e PHOENIX_TELEMETRY_ENABLED=false \
  -i -t \
  arizephoenix/phenix:latest
```

The example is operationally valid, but it also illustrates an important boundary. Inline environment variables are fine for demonstrating mechanics; they are not a good long-term secret-management practice. Store `PHOENIX_SECRET` in a secret manager or Kubernetes Secret and inject it at runtime. Signing keys belong in the same class of protected material as database passwords and API credentials.

**Understand what the signing key protects.** `PHOENIX_SECRET` signs JWTs. If it is weak, exposed in source control, copied into shell history, or shared broadly among operators, you have weakened the entire authentication boundary. Rotate it with the same seriousness you would apply to other authentication secrets, and make sure the operational team knows who owns that rotation.

**API clients need governed authentication too.** Phoenix supports `PHOENIX_API_KEY` for API clients and trace ingesters. That matters because governance cannot stop at the browser. If automation, SDK-based ingestion, or other service clients can write or read operational data, they must authenticate under the same governance model as human users. Phoenix tooling can automatically attach the authorization header when configured, which is useful operationally because it reduces the temptation to handcraft ad hoc header logic in every client integration.

You should still distinguish user identity from service identity in your runbooks. A human troubleshooting a production trace and a pipeline ingesting spans should not share credentials or lifecycle assumptions, even if both authenticate through supported Phoenix mechanisms.

#### **8.4.2. RBAC turns authentication into enforceable access**

Authentication answers who is present. RBAC answers what they may do. Phoenix defines three roles: `admin`, `member`, and `viewer`. That



model is intentionally compact, which makes it usable in real operations but requires you to be disciplined about assignment.

**Role semantics are operational, not abstract.** Phoenix role controls apply to both UI and APIs. That point matters because teams often think of roles only in terms of the web interface, then forget that automation and programmatic access need equivalent restrictions.

The role model is straightforward:

| Role   | Operational meaning                                                 |
|--------|---------------------------------------------------------------------|
| admin  | Can manage users, system API keys, and secrets                      |
| member | Can create operational entities and work actively within the system |
| viewer | Read-only access                                                    |

This simplicity is a strength if you use it well. It forces you to think in terms of actual responsibilities instead of inventing brittle role hierarchies.

**Assign the minimum role that supports the task.** Use `viewer` for broad debugging visibility when users need to inspect traces but not change operational state. Use `member` for engineers or analysts who create and manage operational artifacts. Reserve `admin` for the small group that must manage users, system API keys, and secrets.

The most common RBAC failure is giving too many people `admin` because the deployment starts small and informal. That shortcut is easy to justify in week one and hard to unwind after adoption grows. If someone only needs to inspect traces or run normal analysis workflows, `viewer` or `member` is the better default.

**Automation should have explicit role design.** Because role controls apply to APIs as well as the UI, service-account design matters. Ask which automated jobs need write access, which only need read access, and whether any automation genuinely needs administrative power. Most do not. A trace ingester authenticated with

PHOENIX\_API\_KEY rarely needs privileges equivalent to a human administrator.

That boundary becomes especially important when multiple teams share an instance. Without clear service-account design, credentials intended for ingestion can quietly become general-purpose access tokens.

**RBAC is necessary but not complete.** Phoenix’s three-role model is sufficient for many internal deployments, but it does not remove the need for surrounding controls. Network placement, secret handling, environment separation, and upstream identity policy still matter. If a production Phoenix instance should only be reachable through a private ingress path or internal gateway, RBAC does not substitute for that boundary. It sits inside it.

#### 8.4.3. Retention, minimization, and the lifecycle of trace data

Retention is a governance control, not just a storage optimization. If you keep traces indefinitely by default, you are making a policy choice whether you intended to or not. Phoenix 9.0 and greater supports project retention policies and a deployment-wide default via PHOENIX\_DEFAULT\_RETENTION\_POLICY\_DAYS. Use that capability to express how long data should exist, not merely how long storage happens to remain cheap.

**Set a default that matches risk, not convenience.** A deployment-wide default gives you a baseline. That baseline should be shorter in sandbox and development environments and more deliberate in production. A simple governance matrix is often enough to keep teams honest:

| Environment | Auth/RBAC posture                        | Default retention                | Notes                               |
|-------------|------------------------------------------|----------------------------------|-------------------------------------|
| Sandbox     | Auth enabled, mostly viewer/member       | 7 days                           | Fast iteration, disposable traces   |
| Shared dev  | Auth enabled, limited admins             | 14–30 days                       | Team debugging and review           |
| Production  | Strict role assignment, audited API keys | Policy-driven, finite by default | Sensitive traces, incident use only |

A corresponding deployment default is simple to express:

```
export PHOENIX_DEFAULT_RETENTION_POLICY_DAYS=30
phoenix serve
```

If you set it to 0, retention becomes indefinite. That may be acceptable in narrowly justified cases, but you should assume it increases governance and storage burden immediately.

**Minimize before ingestion when possible.** Retention only answers how long the data stays. It does not help if you never should have stored the data at all. Sensitive prompts, raw secrets, regulated identifiers, or retrieved documents with restricted distribution should be redacted or excluded before they reach Phoenix when your policies require it. The safest trace field is the one never ingested.

This is where application teams and Phoenix operators must coordinate. Phoenix can enforce retention and access boundaries, but it

cannot infer your redaction rules. Make the tracing contract explicit: which fields are allowed, which are transformed, and which are forbidden.

**Retention and backup must agree.** If Phoenix purges data after 30 days but your database backups persist for six months, your effective retention is six months. Align backup retention with Phoenix retention or document the mismatch clearly. Governance fails quietly when application-level deletion and infrastructure-level recoverability tell different stories.

#### **8.4.4. Network controls and privacy posture complete the governance envelope**

Identity and retention are necessary, but they do not close the loop by themselves. Phoenix can mediate external traffic through user-configurable providers or URLs. That creates a network-governance problem: even if access is authenticated and authorized, unrestricted outbound access can still violate policy or create an SSRF-style risk surface.

**Constrain egress as part of the platform, not as a user habit.** Use network policies, firewalls, or proxies to restrict outbound behavior. In Kubernetes or similar environments, route egress through approved paths instead of assuming application-layer provider allowlists are sufficient. Application settings can help shape behavior, but they do not replace infrastructure-enforced network boundaries.

A deployment may therefore combine auth, finite retention, and proxy-based egress controls:

```
export PHOENIX_ENABLE_AUTH=true
export PHOENIX_SECRET=replace-with-strong-secret
export PHOENIX_DEFAULT_RETENTION_POLICY_DAYS=30
export PHOENIX_TELEMETRY_ENABLED=false
export HTTP_PROXY=http://proxy.internal:8080
```

```
export HTTPS_PROXY=http://proxy.internal:8080
phoenix serve
```

The exact proxy topology belongs to your platform design, but the principle is fixed: if Phoenix can reach outbound destinations on behalf of users or workflows, that path needs the same policy attention as inbound access.

**Disable analytics when your privacy posture requires it.** Phoenix collects basic UI analytics by default. If that conflicts with your internal privacy or air-gapped posture, set `PHOENIX_TELEMETRY_ENABLED=false`. This is not merely a cosmetic setting. In regulated or tightly controlled environments, you should know exactly which telemetry leaves the system and why.

**A representative anti-pattern.** A particularly dangerous configuration looks secure at first glance: authentication enabled, admins assigned, and a private database in place. But the deployment also has unrestricted outbound access, indefinite retention, and broad read access for convenience. In a regulated environment, that is not a mature governance posture. It creates too much opportunity for sensitive traces to persist too long and to interact with networks more broadly than policy intended.

#### 8.4.5. Operational habits that keep governance real

Governance fails less often from missing features than from weak habits. A few operational disciplines matter disproportionately.

**Review role assignments periodically.** Small teams drift toward privilege accumulation. Revisit who holds `admin`, which automation uses API keys, and whether any users only need `viewer`. Because Phoenix role controls affect both UI and APIs, stale role assignments can have broader impact than people realize.

**Treat secrets as rotating credentials, not deployment constants.** Store `PHOENIX_SECRET`, database credentials, and `PHOENIX_API_KEY` in secret infrastructure rather than in checked-in manifests or copied shell snippets. Rotate them on a schedule or when ownership changes.

**Make retention defaults visible.** If users do not know the retention window, they will assume one. Put the chosen retention class in deployment documentation and incident runbooks so nobody mistakes sandbox history for durable operational evidence.

**Audit what gets traced.** When teams change prompting, retrieval, or tool integrations, review whether newly captured fields alter the data classification of Phoenix itself. Governance is not a one-time setup task. It shifts whenever the application begins emitting richer trace payloads.

A Phoenix deployment becomes compatible with enterprise expectations when the data is classified deliberately, authentication is enabled from the beginning, roles are assigned by task rather than convenience, retention is finite by default, and network behavior is constrained by infrastructure rather than trust alone. At that point, the system stops being merely accessible and becomes governable enough to carry the operational truth of real AI applications.

## 8.5. Migrating and Operating Phoenix Under Current Version Boundaries

A self-hosted Phoenix deployment usually fails slowly before it fails loudly. The UI still loads, ingestion still appears to work, and the team assumes the platform is healthy. Underneath that calm surface, the runbooks may still contain old commands, the deployment image may have drifted ahead of the documented operating model, and the

database schema may already have crossed a boundary that simple rollback cannot undo. That is why release-aware operation matters. Once Phoenix becomes shared infrastructure, version changes are not just application updates. They are coordinated changes to binaries, schemas, automation, and operator expectations.

A practical upgrade workflow starts with the two commands that remain central to self-hosted operation:

```
phoenix db migrate
phoenix serve
```

Those commands are deceptively simple. They encode a strong operational truth: migrations and runtime startup are separate concerns, and you should treat them separately during upgrades. Phoenix documents migrations as primarily associated with major version upgrades. New major versions may automatically run migrations on startup, which means a version bump can be a schema event rather than a mere image replacement. If you do not account for that distinction, the first surprise usually arrives during rollback.

**Treat version changes as compatibility boundaries.** The most important habit is to stop thinking of a Phoenix upgrade as “pull a newer image and restart the container.” That model works only when no schema change matters, no automation depends on changed behavior, and no clients have encoded stale assumptions. In practice, Phoenix version changes can affect operational entry points, schema state, and the timing of rollout orchestration. Once the database schema changes, your compatibility surface is larger than the application binary alone.

This is especially important in team environments because Phoenix stores durable observability data. A schema-changing major upgrade affects the correctness and supportability of the deployment, not just whether a pod becomes Ready.

### 8.5.1. The upgrade unit is application plus schema

The cleanest way to operate Phoenix is to define the upgrade unit explicitly: target version, schema migration, deployment artifact, and validation procedure. That framing avoids a common failure mode in internal tooling teams, where the image tag changes in one repository while the runbook and the database posture remain implicit.

A simple noninteractive runbook sequence often starts like this:

```
export PHOENIX_SQL_DATABASE_URL=\
postgres://phoenix:secret@postgres:5432/phoenix

phoenix db migrate
phoenix serve
```

That example is intentionally plain because the operational meaning matters more than the shell syntax. First, establish a database connection to the target deployment database. Next, run the schema migration deliberately. Only after migration succeeds should you start or roll the application. In Kubernetes, you may enforce that with an `initContainer` or a pre-upgrade `Job`. On a single host, you may perform it as an explicit maintenance step. The principle stays the same: serialize schema advancement before broad application rollout.

**Why major versions deserve special caution.** Phoenix documents migrations as primarily associated with major version upgrades. That means you should read a major version bump as an instruction to plan a schema event. Even if startup can trigger migrations automatically, do not rely on that behavior as your only operating model. Automatic migration on startup is convenient for simple cases, but in shared production environments it can obscure timing, prolong pod initialization, and complicate incident diagnosis when something stalls.

The better posture is intentionality: know when a major-version migration will occur, know how long it might take on your data volume,



and know what validation gates you will apply before exposing the upgraded application broadly.

**Rollback is not symmetric.** Phoenix explicitly does not auto-downgrade schemas when rolling back to an older version. Downgrade requires manual down migrations or database tooling. This creates a hard operational boundary: image rollback alone is not a full rollback strategy. If a newer Phoenix binary migrates the schema and you later restore only the previous container image, you may end up with an older application pointed at a newer schema state. That is not a rollback. It is an unsupported compatibility experiment.

The practical consequence is severe but simple: always define rollback in database terms as well as application terms. If the upgrade crosses a schema boundary, your rollback plan must include either a tested database restore path or a controlled manual downgrade strategy. If you do not have one, be honest and call the operation forward-only.

### **8.5.2. A release-aware upgrade flow that scales beyond one operator**

Once more than one person depends on Phoenix, ad hoc upgrades are no longer acceptable. You need a repeatable sequence that protects data, tests assumptions, and updates the surrounding automation and documentation.

**1. Inventory what actually runs.** Start by identifying the current Phoenix image or package version, the deployment method in use, the database backend, and any automation that interacts with Phoenix. Include CI jobs, ingestion scripts, operational runbooks, and dashboards that assume specific endpoints or behaviors. The hidden risk is rarely the main server process. It is the small automation surrounding it that still encodes an older operating model.

**2. Pin the target version intentionally.** Do not upgrade to “latest” in production simply because it is available. Choose a specific target image or chart version and test that exact artifact in a nonproduction environment. This matters especially for major versions because the artifact and the schema state become coupled during migration.

**3. Verify backups and restore path before migration.** A database backup that has never been restored is only a hopeful file. Before a schema-affecting upgrade, confirm that backups are current, identify the restore procedure, and make sure operators know the recovery time and recovery point assumptions. If the database is managed by a platform team, get explicit confirmation of the restore process rather than assuming it exists because the service is labeled “managed.”

**4. Run migrations deliberately.** Use `phoenix db migrate` as an explicit control point. Measure how long it takes against representative data, especially if trace volume has grown substantially. If you run on Kubernetes, gate startup with the migration step or execute it as a controlled pre-upgrade Job.

**5. Roll application instances after schema success.** Only after migration completes should you roll the main `phoenix serve` instances. If you operate multiple replicas, roll conservatively enough that a slow migration or an unexpected startup delay does not create unnecessary turbulence.

**6. Smoke-test the real behaviors.** Validate more than page load. Confirm ingestion, trace browsing, authentication, and any critical operational workflows your team actually uses. A Phoenix deployment is not healthy just because the login screen renders.

**7. Update internal examples immediately.** After a successful upgrade, update the runbooks, deployment templates, and internal docs that operators copy from. Stale examples are one of the most persistent

sources of long-term drift.

### 8.5.3. Migration timing, index strategy, and write availability

Large Phoenix deployments backed by PostgreSQL introduce a more subtle upgrade problem: migration-induced interruption. Some schema changes require index creation on large tables, and the method you choose affects both migration time and write availability.

Phoenix provides `PHOENIX_MIGRATE_INDEX_CONCURRENTLY=true` for large PostgreSQL tables. This uses `CREATE INDEX CONCURRENTLY`, which PostgreSQL documents as a way to avoid blocking inserts, updates, and deletes during index builds. That can be valuable when you need to preserve write activity while upgrading. The trade-off is that concurrent index creation is slower and more operationally complex than standard index builds.

A migration-oriented environment might therefore look like this:

```
export PHOENIX_SQL_DATABASE_URL=\
postgres://phoenix:secret@postgres:5432/phoenix
export PHOENIX_MIGRATE_INDEX_CONCURRENTLY=true

phoenix db migrate
```

**Understand the trade-off clearly.** Default index creation is usually faster, but it may block writes while the index is built. Concurrent index creation reduces ingestion interruption, which is attractive for high-ingest environments or upgrades where you want the old version to keep accepting writes during the migration window. The cost is time. Phoenix notes that concurrent index mode is roughly 2–3x slower. On very large tables, that difference can turn a short maintenance task into a long-running operation.

This is not a tuning flag you enable blindly. It is a deliberate choice

between upgrade duration and write availability.

**When concurrent index mode is worth it.** Use `PHOENIX_MIGRATE_INDEX_CONCURRENTLY=true` when write continuity matters more than minimizing migration wall-clock time. That is common when a production instance receives steady ingestion and a brief write block would cause operational pain. You accept a longer migration because the workload remains more available.

**When the default path is better.** If you can schedule a maintenance window, quiesce ingestion, or tolerate a brief interruption, the default migration path may be simpler and faster. Simpler operations are often safer if the service impact is acceptable.

**When neither option is comfortable.** For very large PostgreSQL tables, Phoenix notes that you may need to pre-create indexes manually before the main upgrade. That advice reflects operational reality: once data volume grows enough, even supported migration mechanisms may need to be staged rather than executed monolithically during the same maintenance window.

#### 8.5.4. Kubernetes and multi-replica rollout boundaries

Cluster deployments add another layer of version discipline because multiple replicas interact with shared schema state. Even if Phoenix can migrate automatically on startup, that does not mean you should let every new pod discover the migration boundary independently.

A conservative Kubernetes pattern keeps migration and serving separate:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: phoenix
spec:
  replicas: 2
```

```
strategy:
  rollingUpdate:
    maxSurge: 0
    maxUnavailable: 1
template:
  spec:
    initContainers:
      - name: migrate
        image: arizephoenix/phoenix:latest
        command: ["phoenix", "db", "migrate"]
    containers:
      - name: phoenix
        image: arizephoenix/phoenix:latest
        command: ["phoenix", "serve"]
```

This pattern keeps the migration as a startup gate and limits rollout concurrency. That reduces noise during upgrade and makes failure modes easier to reason about.

**Why rollout sequencing matters.** If multiple replicas start against a major-version migration at the same time, they may compete to advance schema state. Phoenix documents that concurrent migration attempts are safe, but one replica may fail transiently and retry. Safe is not the same as operationally quiet. During an incident, repeated init-container retries can look like generalized failure when the real condition is simply migration serialization.

That is why one-pod-at-a-time rollouts, manual pre-run migrations, or pre-upgrade Jobs are often cleaner than relying entirely on implicit startup behavior.

**Readiness is not the whole story.** A pod becoming Ready does not guarantee that the full deployment is sane under the new version boundary. In multi-replica environments, validate that all replicas converge, the migration did not leave partial operational side effects, and the shared database remains healthy under actual ingestion and query patterns.

### 8.5.5. Automation drift is a versioned interface problem

Operational drift usually hides in automation long after the main deployment has been upgraded. A script that starts Phoenix, a CI job that runs health checks, or an internal operations page with copied commands can quietly preserve outdated assumptions for months.

**Treat internal runbooks as part of the interface.** If engineers copy commands from an internal wiki more often than they consult upstream docs, the wiki is the real operational interface. Update it with the same seriousness as the deployment manifests. Otherwise, you will keep reintroducing outdated commands and workflows every time someone provisions a new environment or responds to an incident.

**Test automation against staged upgrades.** Before promoting a new version, exercise the surrounding jobs: deployment automation, backup hooks, smoke tests, and any scripts that depend on the Phoenix process model. You do not need an exhaustive certification program, but you do need to know that the scripts your operators trust still map to the current Phoenix surface.

**Prefer supported interfaces over accidental ones.** When teams are in a hurry, they sometimes rely on whatever internal behavior appears reachable rather than on the documented CLI and deployment mechanisms. That shortcut eventually becomes upgrade pain. Favor the verified operational commands—`phoenix db migrate` and `phoenix serve`—and documented configuration keys rather than shell wrappers or internal conventions that nobody can justify later.

### 8.5.6. A compact operational checklist

For production-minded teams, a short checklist is often more useful than a long essay when the maintenance window begins:

- Identify whether the target release crosses a major-version boundary.
- Pin the exact target artifact.
- Verify current backups and restore procedure.
- Stage the upgrade in nonproduction with representative data.
- Decide whether index concurrency is needed:

```
export PHOENIX_MIGRATE_INDEX_CONCURRENTLY=true
```

- Run `phoenix db migrate` deliberately.
- Roll `phoenix serve` only after schema success.
- Validate ingestion, UI access, and core workflows.
- Update runbooks and automation references immediately.

**Anti-patterns worth eliminating permanently.** Copying old blog-post commands into production runbooks is the first. If the provenance of an operational command is unclear, do not trust it.

Assuming container-image rollback is a sufficient recovery plan is the second. After a major-version migration, it is not.

Mixing old and new operational habits in the same environment is the third. When half the team follows current migration procedure and the other half treats Phoenix as a stateless container restart, the next outage will be harder than it needed to be.

A stable self-hosted Phoenix deployment is not one that never changes. It is one whose operators understand where the hard version boundaries are, separate schema change from application rollout, and keep their automation as current as the binaries they run.



