OpenTelemetry for GenAI

Tracing Token Costs, Tool Calls, and RAG Latency

Trex Team

Published by NobleTrex Press

For permissions and other inquiries, write to:

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

Contents

Introduction

Generative AI applications operate as highly distributed systems. The integration of large language models into production environments requires orchestrating multi-stage retrieval systems, vector databases, external tool executions, and complex prompt assembly pipelines. Consequently, evaluating system performance and operational economics requires precise, deterministic observability. Traditional monitoring approaches, often reliant on unstructured logging, are insufficient for tracking multi-step agentic workflows, correlating vector search latency with final inference outcomes, or attributing hardware consumption and token usage to specific tenants and business features.

This book addresses these technical requirements by detailing the implementation of OpenTelemetry for generative AI workloads. OpenTelemetry provides a standardized, vendor-neutral framework for generating, propagating, and exporting telemetry data. By applying this architecture to AI-driven systems, engineering teams can establish a unified signal model utilizing traces, metrics, logs, and baggage. This approach ensures that telemetry data remains portable and avoids vendor lock-in, while providing the structural rigor necessary to trace complex, asynchronous model interactions, fan-out operations, and retry mechanisms.

The text centers on the practical application of GenAI semantic con-

ventions. Establishing a canonical telemetry model is critical for ensuring compatibility across analytics backends. The following chapters detail the specific span taxonomy required to represent model operations, retrieval-augmented generation (RAG) pipelines, and autonomous agent behavior. Because these semantic conventions are in active development, the material also establishes explicit strategies for managing schema drift, handling compatibility constraints, and designing internal telemetry contracts. This ensures that analytical queries and production dashboards remain stable across version migrations.

Economic and performance variables are treated as core observability dimensions. A primary focus is the exact attribution of token costs and the precise decomposition of pipeline latency. The material demonstrates how to construct model invocation spans for both buffered and streaming requests, attach detailed token usage metrics, and propagate immutable business context using OpenTelemetry Baggage. Furthermore, it addresses the technical separation of model inference duration from downstream dependency execution, such as database lookups, deterministic post-processing, and custom function executions within agentic tool calls.

Scaling observability requires transitioning from application-level instrumentation to infrastructure-level control. The latter portion of this book examines the OpenTelemetry Collector as the primary mechanism for data routing, policy enforcement, and cost normalization. It details production pipeline topologies utilizing receivers, processors, and exporters to manage high-volume telemetry and mixed-sensitivity workloads. Methods for redacting sensitive prompt content, implementing head and tail sampling strategies to contain high-cardinality traces, and applying data retention tradeoffs are examined to prevent prohibitive backend storage costs.

Ultimately, this book provides the technical specifications required to

operate generative AI systems efficiently and securely. By systematically correlating traces, metrics, and logs, organizations can identify latency bottlenecks resulting from RAG context inflation, isolate discrete tool execution failures, and maintain strict token cost attribution. The resulting observability infrastructure enables data-driven optimization, secure payload handling, and exact fault isolation for production AI systems.

Chapter 1

OpenTelemetry Architecture for GenAI Systems

A GenAI request rarely behaves like a single RPC: it fans out into model calls, retrieval steps, middleware hops, asynchronous continuations, and business-specific routing decisions. This chapter gives readers the architectural lens for seeing that distributed execution clearly, so later chapters on inference, tools, RAG, collectors, and privacy can build on a shared understanding of signal production, causality, propagation, and business attribution.

1.1. The OpenTelemetry Signal Model in AI Workloads

A single user question can trigger an API gateway, prompt assembly, retrieval, reranking, model routing, inference, moderation, tool execution, and post-processing before the answer ever reaches the screen. If you only collect a few application logs and a coarse latency metric, you can tell that something happened, but not which part of the path consumed time, which tenant drove the cost, which branch retried, or where the business context was lost. OpenTelemetry gives you a vendor-neutral way to describe that execution path with a small set of complementary signals: traces, metrics, logs, and baggage.

A practical baseline helps anchor the model early. Consider a gateway service that forwards a chat request to an orchestrator, which then invokes a retriever, a model provider, and a tool executor. A minimal OpenTelemetry setup usually emits all four signal families from that path:

```
export OTEL_SERVICE_NAME=chat-orchestrator
export OTEL_EXPORTER_OTLP_ENDPOINT=http://otel-collector:4318

python app.py
```

That configuration does not create observability by itself; it only defines where the service identifies itself and where exported telemetry should go. The important architectural point is that instrumentation libraries produce signals inside the process, SDKs shape and batch them, collectors transform and route them, and backends index and query them. If you confuse those layers, you end up asking the application code to solve storage problems or expecting the backend to recover context that was never emitted.

Signals are different views of the same workload.

Traces describe causality. They answer *what happened*, *in what order*, and *which operations belong together*. In GenAI systems, traces are the only signal that can reconstruct the execution path across orchestration, retrieval, inference, and tool use. A trace is therefore not just a timer around the model call; it is the structural record of a distributed request.

Metrics describe aggregate behavior. They answer *how often*, *how much*, and *how it trends over time*. For AI workloads, metrics are the right place for request rate, latency histograms, token counts, cache hit ratios, retry rates, error ratios, queue depth, and GPU or CPU saturation. You do not need a separate trace for every request if a metric can already tell you that p95 model latency jumped after a routing change.

Logs describe discrete facts. They answer *what was observed at a point in time*. Logs are appropriate for validation failures, moderation decisions, fallback triggers, prompt template selection, policy denials, and exceptional states that do not justify their own metric. A good log records a small, queryable fact; a bad log tries to carry the entire execution story and then becomes an inefficient trace surrogate.

Baggage carries request-scoped business metadata. It answers *which logical request, tenant, cohort, or policy context is this execution part of?* Unlike span attributes, baggage is intended to travel with execution so downstream services can make consistent attribution or policy decisions. That makes it useful for tenant IDs, subscription tier, experiment cohort, prompt version, or region classification. It is not a payload store, and it is not a substitute for semantic design.

OpenTelemetry treats these signals as complementary rather than competing. Traces give structure, metrics give scale, logs give detail, and baggage gives propagation-safe business context. The architectural discipline is to emit each fact at the narrowest signal that can represent it well.

Signal production is not signal export.

The OpenTelemetry API is where your code records work. The SDK is where those recordings are sampled, aggregated, batched, and transformed. Exporters and collectors move the resulting data to backend systems. That separation matters in GenAI systems because the same service may need to emit high-frequency metrics, selective traces, and sparse diagnostic logs with different retention and privacy rules.

You usually instrument three points in the stack:

- Application code and libraries, where spans, metric instruments, log records, and baggage values are created or read.
- The SDK layer, where sampling, aggregation, and processors decide what is retained and exported.
- The collector pipeline, where telemetry can be normalized, filtered, enriched, redacted, or routed to multiple backends.

That division is especially important for AI workloads because the application often knows the prompt class, tenant, and routing decision, while the collector often owns export policy, sampling strategy, and destination-specific shaping. If you push all of that into the app, every service becomes a telemetry ETL engine. If you push it too far downstream, you lose the context that makes the data useful.

GenAI systems need all four signals because no single one is sufficient.

A model gateway can emit a latency metric, but that metric does not show whether the delay came from retrieval, serialization, provider queuing, or output post-processing. A trace can show those steps, but it is a poor place to answer fleet-wide questions such as *how many tokens did premium tenants consume this hour?* That question belongs to

metrics. A log can record a fallback or refusal, but it cannot replace the request structure that traces preserve. Baggage can tell downstream components which tenant or experiment cohort they are serving, but it should not be treated as a general query substrate.

The most common observability failure in AI systems comes from overloading one signal with the job of another. Teams often start with logs, then add a few traces around the model call, and later discover that they still cannot answer basic operational questions:

- Which stage consumed the latency?
- Which branch caused the cost spike?
- Which tenant saw a degraded path?
- Which requests were routed to a fallback model?
- Which responses were reconstructed after asynchronous work?

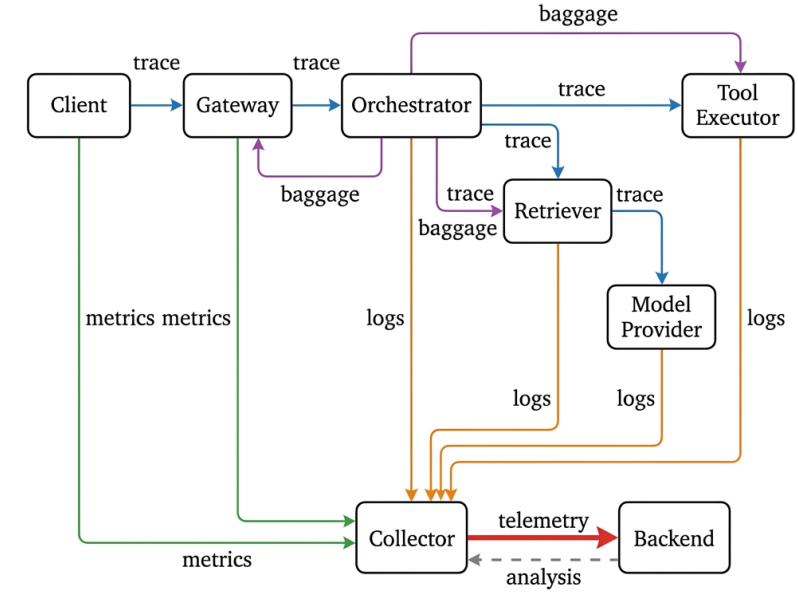
Each of those questions points to a different signal. Traces answer the first two. Metrics answer the cost and trend questions. Logs answer the fallback and refusal details. Baggage makes the tenant and cohort visible as execution flows through multiple services.

A layered architecture keeps the responsibilities clear.

The same user request often traverses several services and execution modes, and each layer should emit the signal it naturally owns. The client or frontend usually contributes coarse request and session identifiers. The gateway sees ingress and routing decisions. The orchestrator sees prompt assembly, chain selection, and downstream orchestration. Retrievers and embedding workers see retrieval latency and corpus selection. Model providers see inference latency, token usage, safety outcomes, and failure modes. Tool executors see external API

latency and side effects. The collector receives all of it and turns it into something queryable.

The diagram below captures the role separation without binding it to any vendor-specific product.



The exact topology varies by architecture, but the signal roles do not. Traces move along the request path. Metrics aggregate operational behavior. Logs capture discrete facts. Baggage carries compact business context. A service may emit more than one signal at a single boundary,

but it should do so because each signal adds distinct analytical value.

Traces explain latency decomposition and causal structure.

In a GenAI request, the user-facing response time is almost always a composition of multiple stages. Prompt construction may be cheap while retrieval is slow. A model call may be fast while streaming post-processing is expensive. A retry may make a request appear slow even though the first attempt failed quickly and the second attempt succeeded after backoff. Only traces can preserve that structure in a form you can read after the fact.

Traces also distinguish accountable execution from incidental work. If a retriever fans out to multiple vector partitions, the trace can preserve that fan-out. If the orchestrator calls a safety filter before and after generation, the trace can show both checks. If a callback resumes after the original HTTP response has started streaming, the trace can still record the resumed work when it is modeled correctly. Those relationships are important because they determine whether latency is attributed to the user-facing path, a background continuation, or a separate workflow.

The useful mental model is that a trace is the execution graph for one logical unit of work. In GenAI systems, that unit is often broader than a single RPC. It may include prompt construction, retrieval, inference, moderation, and tool use. If you try to force all of that into logs or metrics, you lose the graph structure that makes distributed work intelligible.

Metrics explain scale, cost, and distribution.

A trace can tell you that one request used 8,200 input tokens and 1,100 output tokens. A metric can tell you that the service spent the last hour serving 12,000 requests with a p95 of 1.8 seconds and a 17 percent retry rate. That distinction is central to GenAI operations. Trace-level detail

is too expensive to query for fleet-wide trend analysis, and metrics are too aggregated to explain one bad user experience.

The most useful AI metrics tend to fall into a few families:

- **Latency distributions**: overall request latency, model latency, retrieval latency, tool latency, queue latency.
- **Volume counters**: request count, token count, embedding count, retriever query count, tool invocation count.
- **Quality and reliability ratios**: error rate, fallback rate, moderation rejection rate, cache hit rate, retry rate.
- **Capacity indicators**: worker utilization, queue depth, provider quota consumption, GPU saturation.

These metrics complement traces rather than replace them. If p95 latency rises, you use traces to find which stage changed. If cost rises, you use traces and logs to determine whether the increase came from longer prompts, more retrieval fragments, or a model routing change. If a quality metric drops, you use logs and traces to locate the failure mode.

Logs preserve discrete facts that would otherwise disappear.

Logs are most valuable when you need a durable record of something specific that happened once, not a statistical summary. In GenAI systems, that often means a policy action, a fallback decision, a validation error, a prompt template choice, or a tool result summary. A log line can say that a request was routed from one model family to another, that a tool call timed out, or that a moderation rule blocked a completion. That is enough detail for diagnosis without turning the log stream into a replayable transcript.

There is a common failure mode here: teams put request lineage into logs because it feels convenient, then later discover that lineage is better expressed in traces and baggage. Logs do not provide a stable causal model. They are not naturally queryable as a request graph, and they are usually not the right home for data that must survive multiple hops in a uniform way. Use logs for moments, not for structure.

This also matters for privacy and cost. Once you decide to log prompt text, output text, or tool payloads, you have accepted a significantly higher storage, redaction, and access-control burden. A log should usually record that a prompt was assembled from a template, not the full prompt body unless there is a narrow, reviewed diagnostic need.

Baggage carries business context across boundaries.

Many AI systems need to keep track of business dimensions that are not purely technical. Tenant identity, experiment cohort, pricing tier, routing policy, compliance region, and request class are all examples of context that may need to travel with the execution path. Baggage is the signal designed for that job.

The distinction from span attributes is subtle but important. Span attributes describe one operation instance. Baggage describes contextual state associated with the broader request or workflow. If you need the same tenant ID to be visible in an orchestrator, a retriever, and a tool executor, baggage provides a propagation-safe carrier. If you only need a field on one span for local analysis, span attributes are the better choice.

Treat baggage as compact and controlled. It is suitable for a few short keys with bounded cardinality. It is not suitable for prompts, embeddings, large routing tables, or arbitrary JSON blobs. Large baggage increases propagation overhead and makes accidental disclosure more likely. In practice, you want baggage to support joins and policy decisions, not to mirror your entire application state.

Collector pipelines separate emission from storage policy.

Once telemetry leaves the process, the collector can normalize names, filter records, redact sensitive fields, batch exports, and route data to specialized backends. That separation is valuable in AI systems because different signal types often have different retention and access rules. For example, traces may be sampled aggressively while metrics are kept long term. Logs may require privacy filtering before export. Baggage may be copied into selected spans or logs only when a business analysis requires it.

The collector also lets you apply policy consistently. If every service tries to redact prompt content independently, you multiply the chance of inconsistency. If the collector owns the redaction step, you can enforce one rule across the fleet. The same logic applies to attribute normalization, tenant routing, and backend fan-out.

That boundary is worth preserving even when auto-instrumentation is available. Auto-instrumentation reduces effort, but it does not eliminate the need to decide which signals should survive, where they should go, and under what privacy constraints. In an AI platform, the collector is often where observability policy becomes enforceable.

Breadth and precision have to be balanced deliberately.

If you instrument only the model call, you miss the surrounding work that actually dominates the user experience. If you instrument every helper function, you create trace noise, storage pressure, and analysis fatigue. The same trade-off applies to metrics, logs, and baggage. Too little breadth leaves blind spots. Too much detail overwhelms the operator and makes expensive queries routine.

A disciplined GenAI observability model usually follows a few rules:

- Emit traces for each materially distinct stage that affects latency,

failure, or accountability.

- Emit metrics for recurring properties that you want to trend or alert on.
- Emit logs for infrequent but diagnostic facts.
- Carry only compact business context in baggage.
- Keep raw content out of the default path unless policy explicitly allows it.

Those rules are simple, but they prevent most of the common anti-patterns. They also set up the rest of the architecture cleanly: tracing can focus on causality, metric design can focus on aggregation, logs can focus on incidents, and baggage can focus on end-to-end attribution. In practice, that separation is what makes a multi-service GenAI system understandable instead of merely measurable.

1.2. Span Lifecycles, Causality, and Request Boundaries

A trace can be present and still be misleading. A model call may appear to own the full latency even though retrieval was the slow stage. A retry may look like a second user request. A streamed response may end before the work that computed the final tokens has actually completed. In GenAI systems, those mistakes do not just distort dashboards; they break the causal record that you rely on for cost attribution, quality debugging, and incident analysis.

The discipline here is to treat spans as the structural units of execution, not as convenient timers. A span has a start, an end, a parent, optional links, events, status, and exception information. The lifecycle and relationship decisions you make at creation time determine

whether the trace remains readable once requests fan out, retry, resume asynchronously, or continue after the first byte of output has already been sent.

A practical implementation pattern helps anchor the rules. Suppose your orchestration service receives a chat request, performs retrieval, calls a model provider, and then launches a deferred moderation pass after the response stream begins. The instrumentation should create one root application span for the user request, child spans for prompt assembly and retrieval, a client span for the model provider call, and a linked span for the deferred moderation work if it no longer fits the same synchronous parent-child timing. In Python with OpenTelemetry, the structural decisions look like this:

```
with tracer.start_as_current_span("chat.request") as root:
    with tracer.start_as_current_span("prompt.build"):
        build_prompt()

    with tracer.start_as_current_span("rag.retrieve"):
        docs = retrieve()

    with tracer.start_as_current_span("model.invoke"):
        for token in stream_model():
            root.add_event("token.chunk")
            yield token

    ctx = context.get_current()
    schedule_moderation(ctx)
```

The code is not important because of its syntax. It is important because it shows three lifecycle rules at once: the root span owns the accountable user request, child spans model direct sub-operations, and events record milestones inside a long-running span without exploding the span count. The deferred moderation step may need to continue with the captured context, but whether it remains a child or becomes a linked span depends on how you want to represent causality and accountability.

Pick one root span for one accountable request.

The root span should represent the user-facing unit of work that the system is prepared to explain. For a chat application, that is usually the end-to-end request from the moment the gateway accepts it until the system can account for the final response or terminal failure. Do not create competing root spans for prompt construction, retrieval, and model invocation as though they were separate transactions; they belong to one logical request unless you have intentionally split them into independently accountable workflows.

That choice matters because root-span selection controls how the trace is read. If you start a new root at the model call, you erase the application work that led to the call. If you start one root per retry, you turn a single user request into several unrelated operations. If you start one root per callback in an async chain, you lose the continuity that makes the path interpretable.

The root span should also reflect the user boundary, not the internal service boundary. An ingress gateway may propagate the trace, but the application root often belongs to the orchestrator or request handler that owns the end-to-end story. Once the request crosses a boundary you do not control, the same trace can continue through child spans, but the root should remain the original accountable unit.

Use child spans for direct sub-operations.

Child spans are the default relationship when work is a direct part of the same causal path and occurs within the same accountable request. Prompt assembly, retrieval, reranking, model invocation, safety checks, and tool calls are all natural candidates when they are immediate components of the response path.

This rule is useful because it avoids overcomplication. If retrieval is one logical step, create one retrieval span and put the internal mechanics in events or attributes if you need them. If a model call includes request serialization, provider latency, and output streaming, make that

one client span unless a more detailed decomposition has clear operational value. The trace should answer questions cleanly, not become a transcript of every function invocation.

Child spans also preserve timing relationships that matter for analysis. If retrieval begins before the model call, the trace shows that. If a preflight moderation step overlaps with prompt assembly, you can represent that overlap as separate spans under the same root, rather than pretending the work was strictly sequential when it was not. In GenAI systems, this distinction often reveals where concurrency helps and where it only hides latency.

Use events for milestones, not for fake spans.

Events capture important moments inside a span without turning each moment into a separate span. That makes them ideal for streamed generation, token checkpoints, partial retries, cache decisions, fallback transitions, and internal phase markers. When a model is streaming tokens, you do not usually want one span per token. You want a span for the generation operation and events for the significant milestones inside it.

The difference is operational, not stylistic. A span should represent a meaningful operation with its own start and end. An event should represent a point in time that helps you understand what happened during that operation. If you record token boundaries as events, you can correlate output pacing with provider behavior without drowning the trace in thousands of tiny spans. If you emit a separate span for each token, you make the trace expensive to store, expensive to render, and hard to read.

Events also work well for state changes that alter interpretation but do not change ownership. A tool call may emit an event when it sends the request, another when the first byte arrives, and another when the response is parsed. A moderation span may emit an event when a thresh-

old is crossed. A reranker may emit an event when the candidate set changes. Those are all moments inside an operation, not operations that deserve their own tree branches.

Record exceptions and status deliberately.

Failures are often what you need to inspect after the fact, so the trace has to preserve them explicitly. A span should record exceptions when the failure is part of the observable behavior of the operation, and it should set status in a way that reflects the terminal outcome. In GenAI systems, that usually means distinguishing between retriable provider failures, user-facing failures, policy denials, and benign fallbacks.

Do not collapse everything into a generic error. A retrieval miss is not the same as a provider timeout. A safety rejection is not the same as a parsing failure. A retry that succeeds on the second attempt should not erase the first failure; it should preserve the exception or event that explains why the retry happened. That separation is important because one trace can then answer both *did the request succeed?* and *why was it slow or expensive?*

Use exceptions to preserve diagnostic detail, but keep them scoped. A span should carry the exception that belongs to that operation, not every lower-level stack trace it can possibly observe. If a model provider call fails and the orchestrator retries, the provider span should capture the provider failure, and the orchestration span should capture the retry decision. The trace stays intelligible because each level owns its own failure story.

Model retries as part of the same request, not as new requests.

Retries are one of the easiest ways to break causal interpretation. If you create a separate top-level trace for each retry, the dashboard may show several apparently independent requests when there was only one user

interaction. That distorts error rates, latency analysis, and cost accounting. Instead, keep the retry inside the original request trace and make the retry visible through span structure, events, or child spans.

The appropriate shape depends on what you want to learn. If the retry logic is simple, one model-provider span with a retry event may be enough. If each attempt has materially different latency or error behavior, create a child span for each attempt beneath the same orchestration span. That gives you a precise picture of backoff timing, provider instability, and fallback behavior without inventing separate requests.

The key principle is that the user asked once, so you should explain one request. A retry is not a new user intent. It is a continuation of the same intent under a different execution attempt.

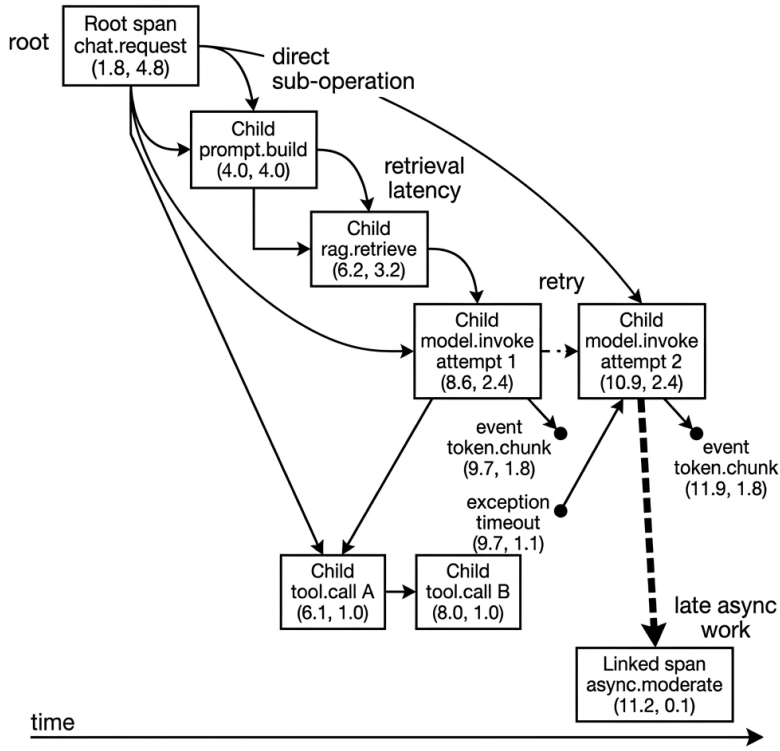
Use links when the causality is real but the parent is not.

A child span implies direct parentage in the same execution chain. A link says something weaker but often more accurate: this span is causally related to one or more prior spans, but it does not fit a strict single-parent model. That distinction matters in GenAI systems because fan-out, batching, deferred work, and resumed workflows are common rather than exceptional.

Imagine a retriever that fans out to several vector partitions, or a post-processing job that combines results from multiple asynchronous tool calls. No single parent span may be truthful for the work that happens next. A link lets you preserve the causal association without pretending the work executed synchronously under one parent. The same applies when a background moderation job is resumed after the initial response has already begun streaming. The moderation span may belong to the same user request, but it is no longer a simple child in the timing sense.

Links are also useful when a span depends on multiple upstream inputs.

A reranking stage may consume candidates from several retrievers, or a summarizer may combine results from multiple tool executions. In that case, a single parent-child chain would be misleading. A span with links can point back to all the contributing spans and preserve the many-to-one relationship that actually existed.



The diagram illustrates a useful balance. The root span owns the request. Prompt assembly and retrieval are direct children. The first model attempt records a timeout exception and an event for streamed tokens before the retry. The second attempt stays within the same re-

quest. A deferred moderation pass becomes a linked span because it no longer fits a strict synchronous child relationship, even though it still belongs to the same logical workflow.

Boundaries should follow accountability, not convenience.

A request boundary is not just a technical delimiter; it is the point at which you decide what one team or one trace is responsible for explaining. In GenAI systems, that decision can be subtle. A gateway may accept an HTTP request, but the orchestration service may own the full workflow. A tool executor may complete after the user sees a response, but the tool work may still be part of the same accountable operation. A workflow engine may checkpoint a job and resume it minutes later, but the resumed work may still belong to the original user interaction or may represent a separate background task.

You should choose boundaries using four questions. Is the work directly on the user path? Does it influence the same latency budget? Does it share the same success or failure outcome? Do you want operators to reason about it as part of the original request? If the answer is yes, keep it in the same trace and preserve the causal edges. If the answer is no, start a new trace and link it only when needed for analysis.

This is where over-modeling and under-modeling both become expensive. Over-modeling every callback as its own trace fragment makes incident response harder. Under-modeling a background task as part of the foreground request makes latency and accountability look better than they are. The boundary should be precise enough to explain the system but broad enough to remain readable.

Streaming work needs special care.

Streaming responses are a frequent source of lifecycle mistakes. The user may receive tokens incrementally while the model is still generating, the safety filter may still be analyzing, or a post-processor may

still be formatting the final answer. If you end the span when the first token is emitted, you lose the real completion point. If you leave the span open after the accountable output is done, you blur latency and complicate export timing.

The right rule is simple: end the span when the operation that the span represents is actually complete. If the span covers the generation operation, it should run until generation is finished, not until streaming starts. If the span covers the end-to-end user request, it should run until the response is complete enough for the system to claim it has fulfilled the request. If a post-response job is still running, decide whether it belongs to the same accountable path or should become a linked continuation.

That lifecycle choice also affects instrumentation around flushing and export. Long-lived or streaming spans may remain open while token events accumulate. You want the SDK and processors to preserve those events without forcing premature closure. At the same time, you should not depend on a flush to rescue work that never belonged in the span structure in the first place.

Async continuations preserve causality only if you carry intent, not just code.

Background callbacks, task queues, and workflow resumptions are common in GenAI pipelines. They are also where traces most easily fragment. A task that resumes later is not automatically a new request, but it is also not always a straightforward child of the synchronous span. The right representation depends on whether you are tracking the same accountable request or a related but separately scheduled operation.

If the resumed work still belongs to the original user request, preserve the causal connection using the appropriate linked or continued span pattern. If the resumed work is operationally separate, start a new trace and add a link back to the original only if that link helps later

analysis. In both cases, avoid the temptation to create a detached span with no connection at all. Detached work is cheap to implement and expensive to explain.

This is one reason trace modeling in GenAI systems needs more care than simply wrapping a function decorator around the model call. The meaningful unit is the request as it evolves across execution modes. A good trace keeps that evolution visible, even when the execution becomes concurrent, asynchronous, or partially deferred.

The practical outcome is a trace that tells the truth about how the system behaved: where time went, where failures occurred, which steps were retried, and which pieces belonged to the original user intent. Once that structure is stable, the transport mechanics can move it across services without changing the story it tells.

1.3. Context Propagation Across Services and Execution Modes

A trace can be correctly modeled and still fail at the first network hop. The gateway may create the right root span, the orchestrator may create sound child spans, and the retriever may emit useful events, yet the trace fractures into unrelated islands once the request crosses HTTP, enters a queue, or resumes inside an async callback. In GenAI systems, that fracture destroys the connection between the user request and the work that actually produced the answer, which in turn makes latency, cost, and failure analysis unreliable.

Propagation is the mechanism that keeps the causal story intact while execution moves. The active in-process context, the serialized wire form of that context, and the runtime machinery that restores it are three different concerns. If you blur them together, you end up either dropping context at boundaries or leaking it across unrelated tasks.

If you keep them separate, you can move a request across services, threads, queues, and workflow engines without losing the parent-child intent established in the span model.

A practical starting point is a simple inbound HTTP request. The application receives trace headers, extracts the context, continues the current span, and injects context into downstream calls. In OpenTelemetry, that is explicit rather than magical:

```
from opentelemetry import trace, propagate

tracer = trace.get_tracer(__name__)

def handle_request(headers):
    ctx = propagate.extract(headers)
    with tracer.start_as_current_span(
        "chat.request",
        context=ctx,
    ) as span:
        downstream_headers = {}
        propagate.inject(downstream_headers)
        call_orchestrator(downstream_headers)
```

That pattern looks small because it is meant to be invisible in the business logic. The important work happens around the handler boundary. Extraction reconstructs the current context from the incoming carrier. Injection writes context to the outgoing carrier. The code in the middle remains focused on the request itself. In a mature deployment, middleware or interceptors usually own these steps so your application code does not become a propagation framework.

Context is the in-process carrier, not the wire format.

OpenTelemetry context is an immutable, execution-scoped container for values that need to move with a request. In practice, that means trace state, baggage, and any other context-local data that instrumentation needs to find later. Because it is immutable, a write produces a new context rather than mutating a shared object. That property is important in async runtimes and concurrent handlers, where accidental

mutation can leak one request's state into another.

The wire representation is separate. On the wire, trace context is usually carried as headers, message metadata, or similar carrier fields. W3C Trace Context standardizes the interoperable pieces that matter most for distributed traces: `traceparent` and `tracestate`. A compliant participant must forward those fields, even if it does not interpret or mutate the vendor-specific parts. That requirement is what keeps a multi-service trace from collapsing when one participant is only partially instrumented.

The runtime mechanism is yet another layer. A language may use `contextvars`, thread-local storage, coroutine-local storage, `context.Context`, or its own native execution model to hold the active context. OpenTelemetry libraries generally integrate with the dominant mechanism in each ecosystem so that the logical request follows the actual call flow. The moment work escapes that flow, you must restore the context explicitly.

Propagators define how context crosses a boundary.

A propagator knows how to read and write contextual data to a carrier. The API is intentionally narrow: `Extract` consumes a carrier and reconstructs context; `Inject` serializes context back into a carrier. That symmetry is what makes propagation portable across HTTP, RPC, queues, and workflow metadata.

You usually want a composite propagator in GenAI services. Trace context alone preserves observability continuity, but baggage often carries the business identifiers that make the trace analytically useful later. Composite propagation lets you combine those concerns without inventing a custom wire format for each service. The practical rule is simple: inject and extract the same standardized context fields everywhere you can, and keep any vendor-specific or framework-specific additions small and deliberate.

The carrier itself varies by transport. HTTP uses headers. gRPC uses metadata. Message queues use message headers or envelope fields. Workflow engines often use checkpoint state or execution metadata. The propagation logic does not change because the carrier changed; only the adapter does. That is why propagation belongs in the integration layer, not in the business rule that decides whether a prompt should be reranked or a tool should be called.

HTTP and RPC are the easiest case, but they still fail at boundaries.

Synchronous request/response paths are the most straightforward propagation path because the downstream call happens while the caller still owns the execution. The gateway extracts incoming context, starts or continues the local span, and injects outgoing context on the request to the orchestrator. The orchestrator does the same for the retriever, model router, and tool executor. If every hop honors the carrier, the trace remains one connected graph rather than a set of unrelated service-local traces.

Even here, failures are common. An ingress proxy may strip trace headers. A service may start a new root span instead of continuing the extracted one. An interceptor may inject on the client side but never extract on the server side. A load balancer or API gateway may terminate one protocol and originate another without preserving the metadata. In GenAI systems, where one user request frequently crosses several internal services, a single dropped header can make the entire path look disconnected.

The safest operational pattern is to propagate at the edge of the transport, not in the code that interprets the payload. Let the middleware extract on ingress and inject on egress. Let business code assume the context is already active. That separation keeps the request handler readable and makes the propagation policy reusable across routes and

services.

Queues need a deliberate decision about continuity.

Deferred work is common in AI platforms. You may enqueue embedding jobs, moderation scans, evaluation tasks, or document enrichment steps. Queues break the immediate caller-callee relationship, so the question is no longer whether to propagate context, but how much continuity you want to preserve.

If the queued task is still part of the same accountable user request, continue the trace across the queue by carrying the relevant context in message metadata. If the task is operationally separate, start a new trace and optionally link back to the original for analysis. That distinction matters because queue latency can be hours, not milliseconds. Pretending that a long-lived background job is a direct child of the original request can distort latency graphs and make the user path look artificially long.

Use links when the queue consumer participates in the same outcome but not in the same synchronous path. Use direct continuation when the job is still plainly part of the original interaction. The decision should follow accountability and interpretation, not convenience. If an analyst reading the trace would reasonably ask, “was this work on the user path or a later side effect?”, then the structure should make the answer obvious.

Workflow engines checkpoint state, so context must survive resume points.

Workflow engines and orchestration frameworks add another kind of boundary: execution may pause, serialize state, and resume later on a different worker. That is not merely transport; it is state migration. If the workflow engine does not persist the current trace context or an equivalent correlation handle, resumed work starts life detached from

the request that triggered it.

The correct boundary behavior depends on the engine's semantics. Some engines can preserve an execution context across checkpoints and restore it automatically when the workflow resumes. Others require you to store a context snapshot in the workflow metadata and rehydrate it explicitly. The important part is that resumed execution should either continue the original trace or deliberately create a linked span that explains the relationship. Silent re-entry with a fresh trace is the wrong default because it erases the causal bridge the workflow engine was supposed to preserve.

This is especially relevant in GenAI systems that use long-running planners, tool supervisors, or evaluation pipelines. A single user request may trigger a workflow that spans multiple scheduling cycles. If you only propagate the latest local context at each cycle, the trace will reflect execution fragments rather than the one logical unit of work the platform actually performed.

Streaming handlers keep the request open while output moves.

Token streaming introduces a subtle propagation problem because the response can begin before the work is complete. In a server-sent event or websocket path, the first chunk may leave the process while the model is still generating or while the post-processor is still deciding whether to redact or reframe later output. The trace context must remain active for as long as the accountable work is active, not just until the first byte is written.

That requirement affects both propagation and span lifetime. If your streaming callback runs on a different coroutine, thread, or event-loop turn, the callback needs the restored context before it creates child spans or events. If the callback escapes the instrumented boundary, the trace can lose its parentage even though the output still belongs to

the same request. The same issue appears in language runtimes that hand off streaming callbacks to framework-managed executors.

A good operational rule is to treat the initial response stream as part of the same request context until the stream closes or the system has otherwise handed responsibility to a separate workflow. If a later moderation or archival task continues after the stream closes, then decide explicitly whether that task is a linked continuation or a new accountable unit.

Async runtimes require explicit restoration at callback boundaries.

Many propagation failures are not network failures at all; they are runtime boundary failures. A coroutine resumes without the right context. A thread pool callback executes after the parent scope has ended. A promise or future callback runs on a scheduler that does not inherit the current context automatically. In each case, the call sequence is still correct, but the execution context is wrong.

The fix is to capture and restore context around the boundary where execution changes hands. In practice, that means the middleware or framework adapter should snapshot the active context before scheduling the async operation, then restore it before the callback creates new spans or emits events. If you skip that step, spans created in the callback become detached even though the code appears to be running "under" the original request.

Different ecosystems expose different helpers for this, but the decision criteria are the same. Ask whether the runtime propagates context automatically across the boundary you are using. If it does not, you must restore it manually. If you rely on ambient behavior, write a test that proves the callback still sees the expected parent context after scheduling, suspension, or resume.

Propagation must be shaped by interpretation, not just mechanics.

Not every downstream unit should be forced into strict parent-child continuity. Some units are part of the same user request and should remain in the same trace. Others are related but better represented as linked work. The difference is not academic. It determines whether a reader interprets the downstream step as synchronous continuation, deferred side effect, or separately accountable background activity.

Use strict continuity when the downstream step directly contributes to the same latency budget and the same user-visible outcome. Use links when the work shares a causal relationship but no longer fits a single parent-child timing model. Use a fresh trace when the task is clearly separate but you still want to preserve a diagnostic connection. That decision tree keeps traces truthful in systems that mix synchronous calls, retries, queues, and long-lived workflow executions.

The cost of getting this wrong is visible immediately. If you continue every task indiscriminately, your traces become too long and too linear. If you start new traces too eagerly, your observability fragments and you lose the ability to reconstruct the request path. Propagation is therefore not just a transport concern; it is how you preserve the meaning of the structure designed in the span model.

Middleware owns most propagation, and that is where the policy belongs.

You should not scatter propagation calls through business logic unless you have no alternative. Middleware, interceptors, server filters, client wrappers, queue adapters, and workflow hooks are the right places to extract, inject, restore, and, when necessary, redact context. That placement keeps the propagation policy close to the boundary it governs and away from the code that implements product behavior.

This also helps with correctness across languages. Java frameworks often expose server and client interceptors. Python services commonly use middleware, decorators, or framework hooks. JavaScript and TypeScript runtimes rely heavily on async context helpers and request lifecycle middleware. Go typically wraps `context.Context` through call chains and transport adapters. The surface area differs, but the architectural rule does not: make the transport layer responsible for preserving context, and make the application layer consume it.

That separation keeps the GenAI request path coherent even when the execution mode changes three or four times within one interaction. The user sees one answer, the trace shows one causal story, and the platform can still cross HTTP, queue, stream, and resume boundaries without inventing a new observability model at each hop.

1.4. Baggage as the Backbone of Business Attribution

Two requests can look identical in the trace viewer and still belong to very different business realities. One may come from a premium tenant on an experimental prompt version, routed to an expensive model path under a feature flag. The other may be a baseline request with the same latency and token count but a different contract, cohort, or region. If you lose that business context after the first hop, the telemetry stays technically correct and analytically useless. Baggage solves that problem by carrying compact request-scoped metadata alongside execution so the rest of the system can attribute behavior consistently.

A useful implementation pattern makes the role concrete. You create a short, bounded set of keys at the edge, inject them into outgoing carriers, and extract them at each service boundary. In OpenTelemetry Python, the shape is straightforward:

```
from opentelemetry import baggage, context, propagate

ctx = baggage.set_baggage("tenant_id", "acme", context.Context())
ctx = baggage.set_baggage("tier", "premium", ctx)
ctx = baggage.set_baggage("prompt_ver", "v12", ctx)

headers = {}
propagate.inject(headers, context=ctx)

incoming = propagate.extract(headers)
tenant = baggage.get_baggage("tenant_id", context=incoming)
tier = baggage.get_baggage("tier", context=incoming)
```

That snippet shows the basic discipline. You do not treat baggage as a telemetry store. You treat it as execution-scoped business context that survives transport boundaries. The value appears later when spans, metrics, and logs can all be joined against the same tenant, cohort, or routing decision without each service inventing its own ad hoc metadata path.

Baggage is context, not a payload cache.

The most important distinction is that baggage exists to accompany execution, not to hold rich application state. It is designed for compact key-value metadata that downstream components may need for correlation, attribution, or policy checks. Typical examples include tenant or account identifiers, subscription tier, experiment cohort, prompt version, workflow version, model routing decision, request class, session class, cost center, and compliance region.

That scope is deliberate. If you try to use baggage for prompts, tool outputs, embeddings, or large JSON structures, you defeat its purpose and create operational problems at the same time. Large baggage increases propagation overhead, raises the chance of header bloat, and makes privacy control harder because the data now travels through every hop by default. Baggage should answer questions like *which business slice is this request part of?* not *what was the full content of the request?*

This is where baggage differs from span attributes. Span attributes describe one operation instance. Baggage describes request-scoped business context that may span many operations. If only one span needs the value, put it on the span. If several downstream participants need it, propagate it as baggage first and optionally copy it into selected spans later.

Propagation works because baggage follows the same execution path as the trace.

Baggage is independent of tracing in the sense that it is its own signal, but it becomes valuable because it moves with the same request context that trace propagation uses. In practice, you usually propagate baggage together with trace context through a composite propagator. That is the right default for GenAI systems because trace context preserves causal continuity while baggage preserves business meaning.

The relationship is subtle but essential. The trace tells you that a request went from gateway to orchestrator to retriever to model provider. The baggage tells you that all of that work belonged to tenant `acme`, tier `premium`, cohort `exp_17`, and prompt version `v12`. Without baggage, you can still diagnose the path. With baggage, you can group, filter, and attribute the path correctly across every downstream signal.

This is especially useful in systems where the actual business decision is made once at the edge and then consumed everywhere else. A model router may need the tenant tier. A retriever may need the region. A tool executor may need a policy class. A collector or backend may later need to join the same metadata against cost analysis or experiment results. Baggage provides one compact carrier for all of that, which keeps the rest of the schema cleaner.

Use baggage to preserve joins, not to duplicate every attribute.

A common mistake is to copy baggage into every span attribute automatically. That seems convenient because it makes the data visible in the trace viewer, but it quickly creates schema clutter and cardinality pressure. Some baggage values belong on selected spans, some belong only in the propagated context, and some should never be promoted at all.

A good rule is to distinguish between three uses:

- *Propagation-only fields*: values that downstream code needs for policy or routing but that do not need to appear on every span.
- *Selective attribution fields*: values that you copy onto a small number of spans or logs because they materially improve analysis.
- *Ephemeral request context*: values that are useful in-process but should not travel beyond their local scope.

For GenAI workloads, tenant ID and experiment cohort often belong in the first or second category. A routing decision may belong in the second because it explains why a request took a given path. A raw prompt fragment belongs in neither by default. If you treat baggage as a blind mirror of application state, you end up with data that is difficult to query, difficult to secure, and expensive to store.

The selective-copy rule also protects the analytical shape of the system. Trace attributes should remain meaningful operation descriptors, not a pile of every business label that happened to pass through the request.

Bound baggage by size, count, and stability.

Propagation formats have limits. Practical baggage carriers must stay small enough to survive headers, message envelopes, and intermediary systems. Platforms may drop members when limits are exceeded,

and they must not propagate partial members. That means a large or unstable baggage set is not just inefficient; it can become semantically inconsistent when some fields disappear and others survive.

That constraint leads to a useful design discipline. Keep the key set small, the values short, and the semantics stable over the life of the request. If a value changes during execution, ask whether you actually need baggage or a local span attribute. If the value is large, ask whether you need an identifier rather than the content itself. If the value is high-cardinality and not needed for downstream joins, do not propagate it.

The practical consequence is easy to overlook. A baggage design that looks fine in one service can fail when the request crosses a proxy, queue, or workflow engine that imposes stricter limits. When that happens, partial propagation is worse than no propagation because downstream systems may infer an incomplete business context. The right approach is to choose a compact set of fields that can survive the entire route without special handling.

Treat baggage as immutable business context.

Baggage should not drift as the request moves. The OpenTelemetry baggage container is immutable, which matches the broader context model and prevents accidental cross-request mutation. When you need a different value, you create a new context rather than editing shared state in place.

That immutability is not a convenience detail. It protects you from subtle bugs in async runtimes, worker pools, and callback-heavy orchestration code. A mutable business context can be overwritten by a concurrent request, refreshed midway through a retry, or partially updated by a helper that only sees one stage of the workflow. Once that happens, your attribution becomes inconsistent and difficult to debug.

The safer model is to establish business context at the edge, propagate

it unchanged, and only create a new derived context when the business semantics genuinely change. If a user switches tenants or a workflow crosses into a differently accountable unit, that is a new context decision. If the system simply advances to the next step in the same request, the baggage should remain stable.

Copy baggage into spans only when it improves analysis.

Some SDKs or instrumentation libraries provide processors that copy baggage into span attributes at span start. That can be useful, but only if you keep the set bounded and the rationale explicit. Copying baggage into spans makes selected business dimensions easier to query alongside trace structure, which is useful for tenant-level analysis, experiment comparison, or routing audits.

You should do that selectively. A few short keys on a few spans can materially improve observability. A blanket copy of every baggage item into every span adds noise and can create a privacy problem because span storage is often more broadly accessible than the original business context should be. The copy step should therefore be a deliberate promotion from propagation metadata to queryable telemetry, not an automatic cloning of all request context.

In GenAI systems, the best candidates for promotion are usually the dimensions that you need in every cost, latency, or quality report: tenant, cohort, prompt version, route, and region. Even then, promotion should happen at a bounded set of spans rather than everywhere. The tracing schema stays manageable when span attributes remain focused on the operation itself and baggage carries the cross-cutting business dimensions.

Use baggage to make multi-signal attribution work.

Baggage earns its place when traces, metrics, and logs all need the same business slice. Suppose a premium tenant experiences slow retrieval

and higher token usage after a prompt rollout. Traces show the slow stages. Metrics show the aggregated latency and cost trend. Logs show the fallback and routing decisions. Baggage supplies the tenant, cohort, and prompt-version labels that let you join all three signals without relying on each service to reconstruct the same identifiers independently.

That joinability is the backbone of downstream analysis. A metrics dashboard can break out request rate by tenant. A trace query can filter for an experiment cohort. A log search can isolate a region-specific policy denial. The same baggage keys make those views line up. Without them, you can still inspect each signal in isolation, but you lose the ability to explain why one slice of traffic behaved differently from another.

This matters even more in GenAI platforms because business context often determines the operational path. The same user request may route to different models, different retrieval corpora, different prompt templates, or different moderation policies depending on tenant, experiment, or region. Baggage is the lightweight mechanism that keeps that decision context available all the way down the path.

Keep the field set small and explicit.

A disciplined baggage strategy begins with an explicit allowlist. You decide which keys are allowed to propagate, which keys may be promoted to span attributes, and which keys must never leave the process. That allowlist should be short enough that you can audit it by inspection. If you need a long schema, you are probably trying to turn baggage into a general-purpose metadata bus.

The best fields are ones that are:

- short,
- stable over the request lifetime,

- useful across multiple downstream services,
- low risk when propagated, and
- valuable for joins or policy checks.

Tenant ID, cohort, route class, and compliance region fit that profile. Free-form user input, generated text, or large routing blobs do not. If you are uncertain whether a field belongs in baggage, ask whether downstream code needs the value for a routing decision, attribution join, or policy enforcement. If the answer is no, keep it local or record it on a span or log where it belongs.

Do not confuse baggage with resource attributes or metric labels.

Baggage travels with the request. Resource attributes describe the emitting process or service. Metric labels are dimensions on aggregate measurements. Span attributes describe a particular operation instance. Those are different homes, and GenAI observability gets messy when you blur them.

A model service may have a resource attribute for its version, environment, or cluster. That is not baggage because it describes the emitter. A metric may have labels for model family, tenant tier, or route class. That is not baggage either; it is a dimension on the aggregate signal. A span attribute may record the number of output tokens or the chosen routing decision. That is an operation-level fact. Baggage is the cross-boundary context that may inform all three, but it is not equivalent to any of them.

The practical benefit of keeping those homes separate is schema discipline. Resource attributes stay stable and service-oriented. Metric labels stay bounded and aggregation-friendly. Span attributes stay operation-specific. Baggage carries the compact business context that

stitches them together. That separation keeps later analyses sharper and avoids turning one signal into a dumping ground for the others.

Respect privacy and policy boundaries.

Because baggage propagates broadly, it deserves the same care you would give to any request-scoped business metadata that can cross multiple services. If a key identifies a user, a tenant, a billing relationship, or a regulated region, decide intentionally whether it is allowed to propagate and whether it may appear in selected spans or logs. The propagation path is not a safe place for data you are unwilling to expose downstream.

The risk is not only leakage; it is also overexposure. Once business context is copied into too many places, access control becomes harder to reason about. A field that was safe in a narrow internal context may become visible in a shared telemetry backend. Baggage helps attribution only if you keep the field set narrow, the values short, and the downstream promotion rules explicit.

That is why baggage works best when treated as a policy surface as much as a technical one. You define it once, propagate it consistently, and limit where it is materialized. Then tenant-level cost attribution, experiment analysis, routing audits, and compliance tracing can all share the same business context without contaminating the rest of the telemetry model.

Chapter 2

Semantic Conventions for GenAI Observability

GenAI systems fail in ways that are expensive, subtle, and distributed: token spikes appear in one service, retrieval latency in another, and tool execution failures somewhere else entirely. This chapter argues that without disciplined semantic conventions, those failures become untraceable folklore; with them, observability becomes portable, evolvable, and operationally trustworthy even while the GenAI specification itself is still changing.

2.1. Why Semantic Conventions Matter More Than Custom Field Names

Two teams can ship the same GenAI service and still produce telemetry that cannot be combined. One team records `llm.model`, another emits `model_name`, and a third hides provider details inside a JSON

blob. Each service remains debuggable in isolation, yet the organization loses the ability to ask one reliable question across the fleet: which model family is driving latency, token burn, and error rate? That failure is not a logging annoyance; it is a schema problem. OpenTelemetry semantic conventions exist to solve exactly this class of problem by turning names and attributes into a shared analytical contract rather than a local implementation habit.[1]

A concrete example of the problem. Suppose you instrument a single model gateway and see three different payload shapes across services:

```
POST /v1/chat
span.name="chat request"
attributes:
  llm.model="gpt-4.1"
  latency_ms=842
  tokens_in=1294
  tokens_out=311

POST /v1/chat
span.name="chat request"
attributes:
  model_name="gpt-4.1"
  request.latency=842
  input_tokens=1294
  output_tokens=311

POST /v1/chat
span.name="chat request"
attributes:
  provider_meta={
    "model":"gpt-4.1",
    "latency_ms":842,
    "usage":{"in":1294,"out":311}
  }
```

Each service can answer its own debugging questions. None of them

[1]OpenTelemetry defines semantic conventions as a common naming scheme for operations and data so telemetry can be standardized across codebases, libraries, and platforms.

can be queried together without per-service translation logic. A dashboard built on the first schema silently misses the second and third. A token-cost aggregation job becomes a tangle of field aliases. An alert for vendor-specific failures becomes impossible to trust because the error dimension is not normalized. The issue is not that the services emit too little data; it is that they emit incompatible meaning.

Semantic conventions are contracts, not naming style. Custom field names feel harmless because they work immediately inside the code that produced them. The cost appears later, when telemetry leaves that codebase. Semantic conventions supply the stable vocabulary that downstream systems depend on: dashboards, correlation rules, alert expressions, ETL jobs, warehouse models, cost reports, and vendor-agnostic analysis. Once you treat those names as a contract, you stop optimizing for local convenience and start optimizing for interoperability.

That distinction matters because observability only pays off when signals can be recombined. A span attribute is useful not because the application can print it, but because another service, a collector, or an analytics backend can interpret it the same way. OpenTelemetry's naming guidance is explicit about this direction: reuse existing conventions where possible, avoid inventing new names under reserved namespaces, and treat the semantic model as the shared interface between producers and consumers of telemetry. If you create local synonyms for already-defined concepts, you get short-term freedom and long-term translation debt.

Portability across backends depends on shared meaning. A backend can ingest arbitrary key-value pairs, but ingestion is not interpretation. If one team sends `provider`, another sends `llm.vendor`, and a third sends `model_host`, the storage engine may happily store all

three. The analytical layer, however, now needs a translation matrix before it can answer even basic questions. That translation is fragile for three reasons.

- First, it is easy to forget a field. A newly added service may emit a variant nobody mapped.
- Second, it is easy to misread a field. Two names may look related but encode different concepts, such as requested model versus actually served model.
- Third, it is expensive to maintain. Every schema rewrite propagates into saved queries, alert rules, notebooks, and data pipelines.

Semantic conventions compress that translation surface. If every service uses the same canonical attribute for the same concept, the backend can aggregate directly instead of normalizing ad hoc vocabulary first. The result is portability: you can move analysis between vendors, storage systems, and collector topologies without rewriting your telemetry model at each hop.

Consistency across libraries and teams is a scaling requirement. A single engineering team can coordinate naming by social agreement. A larger organization cannot. Once you have multiple SDKs, auto-instrumentation packages, language runtimes, and platform teams, field naming becomes a coordination problem. One library may prefer dotted names, another snake case, and a third may nest structured payloads because its local framework makes that easy.

Semantic conventions give you a boundary against that drift. They let you instrumentation layer answer one question repeatedly: what is the domain concept, and what is the standard field for it? For GenAI

telemetry, that includes operation type, provider identity, model identifier, token counts, tool calls, retrieval targets, and exception categories. If each team invents its own version of those fields, then the fleet becomes semantically fragmented even if the wire format is perfectly healthy.

This is why semantic conventions are especially important in GenAI observability. The same workload often crosses multiple teams: product code triggers a model call, a platform service handles routing, a retrieval service fetches context, a tool backend executes side effects, and a vendor API returns the response. Each layer may be owned by a different group with different coding style preferences. Without standard names, the trace looks connected syntactically but disconnected analytically.

Stable dashboards depend on durable field identity. Telemetry consumers do not merely query live data. They accumulate assumptions. A dashboard titled *Model Latency by Provider* may power on-call response, quarterly capacity planning, vendor cost review, and regression tracking. If the underlying field names change, the dashboard does not just need a cosmetic update; it loses historical continuity. A saved alert that watches `error.type` or a query that filters on a canonical model attribute becomes brittle when the same concept is renamed locally in one service.

Semantic conventions protect those assets. They let you build dashboards against stable field identity rather than implementation detail. That stability matters because observability artifacts outlive the code that produced them. A service can be refactored in a week. A business-critical dashboard may remain in use for years. If the dashboard depends on local names, every refactor becomes a schema migration event. If it depends on shared conventions, the telemetry producer can evolve internally while preserving the analytical surface.

The same logic applies to derived metrics. If token usage is calculated from several span attributes, those attributes must mean the same thing everywhere. Otherwise, a metric built from mixed schemas becomes numerically plausible but analytically wrong.

Cardinality is a schema design problem. Custom fields are often introduced because they seem more precise. In practice, precision without normalization can create cardinality explosions. A team may choose to emit per-request identifiers, raw prompt text fragments, full endpoint URIs, user IDs, or model deployment aliases inside fields that later become grouping keys. That makes the data look richly descriptive but breaks aggregation.

Semantic conventions help you separate dimensions that are analytically useful from values that are merely unique. A well-chosen convention uses low-cardinality fields for grouping and keeps high-cardinality details in places where they will not poison queries. The distinction is especially important for GenAI systems, where conversational identifiers, prompt variants, tool arguments, and retrieval document references can become enormous if treated as metric dimensions.

The practical rule is simple: if a value must be aggregated across services, standardize it; if it is only meaningful inside one service, consider whether it belongs in a span event, log record, or suppressed entirely. The schema itself should push you away from accidental high-cardinality dimensions.

Ad hoc naming hides vendor lock-in. Teams sometimes justify custom telemetry fields by saying the names are only internal. That argument overlooks how quickly internal names become external dependencies. As soon as dashboards, alert rules, or warehouse models rely on them, the schema is no longer internal. It is an interface.

Custom naming also creates hidden lock-in to the first backend that happens to support it well. If your schema mirrors one vendor's terminology, changing backends later becomes a semantic migration, not a storage migration. Semantic conventions reduce that risk because they define a common contract independent of one observability stack. You can still extend the model, but the center of gravity stays in a portable vocabulary.

This does not mean that every backend must expose every field identically. It means that the producer should emit a standard meaning, and the consumer should preserve that meaning even if its storage model differs. When a backend needs translation, translation belongs in a controlled layer, not in every application call site.

Use standard fields when the concept is shared. A useful decision rule is to ask whether the field represents a domain concept that other services will need to understand. If the answer is yes, prefer the standard semantic field. Model identity, operation type, provider identity, token counts, exception category, and execution duration are all shared concepts in GenAI observability. They are not local implementation details.

Use a custom extension only when the concept is genuinely organization-specific or not yet covered by the ecosystem. Even then, keep it in a namespaced, explicit extension area rather than repurposing a standard field with overloaded meaning. Overloading is dangerous because it makes queries look correct while changing what the data actually means.

For example, a field named `model` should not sometimes mean the requested model, sometimes the serving backend, and sometimes the finetuned deployment alias. Those are different analytical concepts. If you need all three, keep them separate. Semantic discipline is what

allows post-hoc questions like “Which backend actually served the request?” and “Which model did the caller ask for?” to be answered without guesswork.

Use custom fields sparingly, and only for local concerns. Not every piece of telemetry needs a global name. A field that helps one service debug an implementation detail may never need fleet-wide agreement. Internal cache keys, routing hints, experiment group labels, and local retry reasons often fall into this category. If they will never be aggregated across services, a custom field is acceptable.

The mistake is to treat local usefulness as evidence of global importance. Many teams begin by adding a field because it helps a single incident. Later, analysts discover that similar fields exist under five different names across the fleet. At that point the field is no longer local; it has become a fragmented convention.

A good extension policy keeps custom fields narrow and clearly scoped. If the field describes a concept that is likely to appear in other services, promote it to a shared organizational vocabulary or map it to an established semantic convention. If it describes an implementation detail that only matters inside one service, keep it private and avoid designing queries around it.

Do not encode structured meaning in opaque blobs. The worst compromise is often the most tempting one: put everything in a JSON blob and call it flexible. That approach appears to preserve future optionality, but it destroys queryability. A backend can store the blob, but it cannot aggregate across nested keys without extra extraction logic. Cross-service joins become brittle because the structure is not canonicalized.

Structured data is valuable only when the structure itself is standard-

ized. If you need nested values, define them explicitly and consistently. If a provider response includes token usage, surface the token counts as named attributes with stable semantics. If a tool call has a target and duration, represent those as queryable fields rather than embedding them inside a serialized payload.

Opaque blobs are appropriate only for data that is inherently unstructured and rarely queried. For operational telemetry, they usually hide more than they help.

Semantic conventions also shape collector policy. Collectors are not just plumbing. They are where schema discipline either survives or degrades. If upstream services emit inconsistent field names, the collector must either normalize them or pass the inconsistency downstream. Normalization is far easier when the source schema already aligns with semantic conventions. The collector can then enrich, filter, or transform without first guessing whether `model_name` and `llm.model` mean the same thing.

That matters operationally because collector rules become simpler, safer, and cheaper to maintain. A translation processor that reconciles a few intentional extensions is manageable. A processor that reverse-engineers arbitrary service-specific naming is not. Standard conventions reduce collector complexity and lower the risk of schema drift during fleet-wide processing.

This is one reason semantic conventions are not merely documentation. They are an enforcement surface. Once instrumentation emits a shared vocabulary, downstream systems can validate it, enrich it, and route it with far less ambiguity.

The decision criteria are analytical reuse, stability, privacy, and scope. When you decide whether to use a standard semantic

field, extend it, or suppress the data, four questions usually settle the matter.

- Is the concept reused across services? If yes, prefer the standard field.
- Is the field likely to stay stable over time? If the answer is no, do not let dashboards depend on a local ad hoc name.
- Does the field expose sensitive content or unnecessary detail? If yes, avoid emitting it by default and consider whether it belongs in a redacted or opt-in path.
- Is the field describing a domain concept or a local implementation detail? Domain concepts belong in shared conventions. Local details usually do not.

These criteria force you to think about the field as part of an organizational model, not just a code-level convenience. That shift is the real value of semantic conventions: they turn telemetry design into a contract decision.

A standard schema makes later GenAI analysis possible. GenAI observability becomes hard when every service invents its own vocabulary for the same few ideas. The same prompt can flow through a gateway, a retrieval layer, an embedding job, a model invocation, and one or more tools. If each hop uses different names for provider, model, latency, or failure category, later analysis collapses into manual reconciliation.

A shared semantic model avoids that collapse. It lets you compare latency across services, attribute cost to the right model family, join tool execution to the correct request, and separate transport failures from

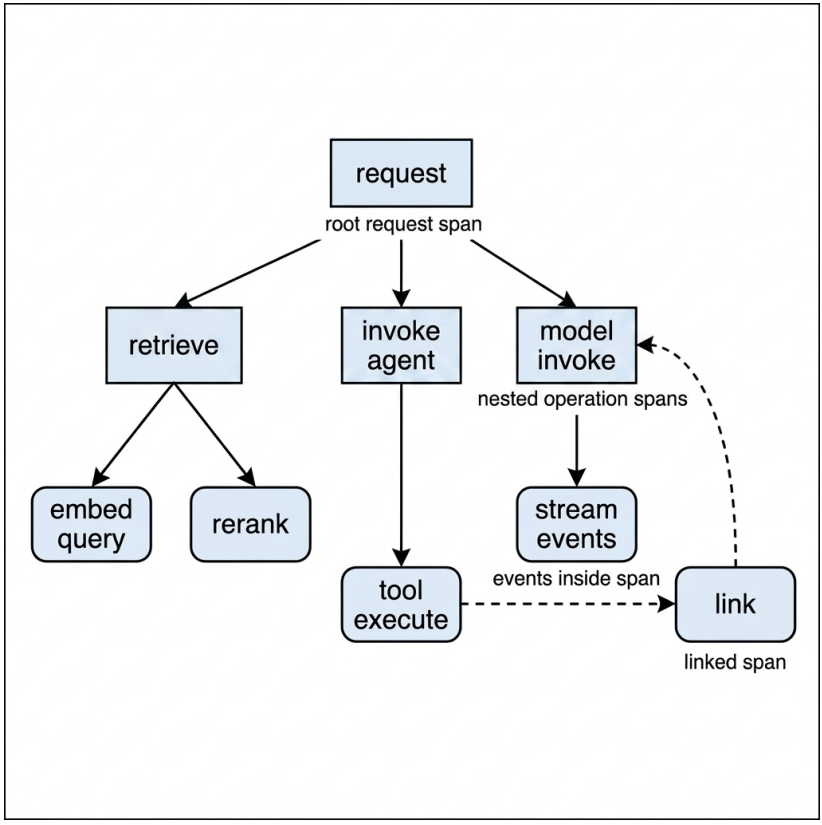
model failures. It also gives your dashboards a longer life, because they depend on meaning rather than the local shape of one codebase.

The point is not to make telemetry rigid. It is to make it interpretable after it leaves the service that produced it. Once that contract is in place, GenAI instrumentation can evolve without turning every query, alert, and report into a schema archaeology exercise.

2.2. The GenAI Span Taxonomy

A slow GenAI request rarely has a single cause. The user sees one delay, but the trace often hides several distinct operations: a retrieval fan-out, a reranking pass, an embedding call, a model invocation, and sometimes a tool call nested inside orchestration code. If you collapse all of that into one undifferentiated "LLM span," you lose the ability to assign latency, cost, ownership, and failure modes to the right part of the system. The span taxonomy exists to prevent that collapse. It gives each kind of GenAI work a place in the trace model so you can reason about the system as a sequence of accountable operations rather than a monolith.

A practical taxonomy sketch. The following hierarchy captures the basic shape of a single GenAI trace. The root request span represents the user interaction or API entry point. Beneath it, you place the operations that do real work: retrieval, embedding, reranking, model invocation, agent planning, and tool execution. Events record meaningful milestones inside a span, while links connect causally related work that does not fit a strict parent-child tree.



This is not a decorative tree. It is a decision framework. Once you decide where each operation belongs, the trace tells a coherent story about where time, cost, and risk accumulate.

Start with the root request, then decompose the work. The root span marks the end-user request or the service entry point. It should cover the full interactive unit of work, but it should not absorb every sub-operation into an unstructured blob. The request span answers a narrow set of questions: how long did the user-facing operation take, how often did it fail, and which major phases contributed to the

total latency?

Everything else belongs below it. If the request invokes a retriever, call out retrieval. If retrieval performs an embedding request, represent that separately when the embedding computation has meaningful latency or distinct failure behavior. If the orchestrator consults a model, create a model span. If the model response triggers a tool call, represent the tool execution as its own span when that tool work is externally significant. This decomposition keeps each unit of work visible without making the trace unreadable.

The crucial idea is that a GenAI trace is usually a composition of heterogeneous operations. A request can be user-facing, while the children are system-facing. That distinction matters because the user experience depends on the sum of all children, not just the model call.

Model spans represent a concrete inference or generation operation. Model spans are the core unit of GenAI observability, but they are not the whole model. A model span should represent one invocation of an LLM, embedding model, image model, or similar inference endpoint. It exists when you need to measure latency, token usage, error behavior, or provider-specific differences for that inference operation.

For chat or completion workloads, the model span is where you measure prompt processing and response generation as one logical unit. For embeddings, the span captures the vectorization call, not the downstream index lookup or retrieval logic. For streaming responses, the span may remain open while incremental output arrives, and the stream itself can be represented by events inside the span when the incremental chunks are part of a single attributable model call.

Do not use a model span as a bucket for everything that happens around the model. Prompt formatting, policy checks, cache lookup, and downstream post-processing may belong elsewhere. If those steps

are independently meaningful, make them their own spans or events. If they are trivial bookkeeping, keep them as attributes or internal implementation details.

Agent spans represent orchestration, not inference. Agent behavior deserves its own place in the taxonomy because it is not the same thing as model invocation. An agent span represents planning, routing, loop control, memory access, and coordination across tools or models. The agent decides what to do; the model span records the inference that helped it decide or respond.

That distinction prevents a common observability error: attributing orchestration latency to the model. In many agentic systems, the model itself is only one stage in a longer control loop. The agent may query state, select tools, inspect previous actions, and decide whether to call the model again. Those steps are analytically important because they explain why a request took 2 seconds instead of 200 milliseconds. If you bury them inside a single model span, you lose the ability to distinguish control-plane overhead from inference latency.

Agent spans are also the right place for low-frequency but important orchestration events: branch selection, retry decisions, tool-choice decisions, and termination conditions. Keep the span focused on orchestration work that matters across requests. Do not turn every internal function call into an agent span unless it changes the latency or failure model in a way you will actually analyze.

Retrieval and reranking deserve first-class spans when they carry their own latency. Retrieval is one of the easiest operations to hide and one of the easiest to misdiagnose. A user blames "the model" when the actual delay came from a vector search, a document fetch, or a reranking step. If retrieval is part of your request path and its latency can vary independently, give it its own span.

That span should represent the retrieval operation as a unit of work,

not a vague “RAG” label. A retrieval span may contain child operations such as an embedding query, a vector database lookup, a lexical search, or a reranker call. If a retrieval stage fans out across multiple sources, you may need one parent retrieval span with several child spans beneath it, or a set of linked spans when the work is parallel but not strictly nested.

Reranking often deserves separate treatment because its cost profile differs from the initial search. It may be CPU-heavy, model-assisted, or external-service backed. If reranking meaningfully changes total latency or has its own failure behavior, model it explicitly. That gives you a clean way to compare search time to ranking time and to see whether the bottleneck lies in candidate generation or candidate ordering.

Embedding calls are separate operations when they can fail or vary independently. Embedding generation often appears to be a small helper call, but in production it is frequently a distinct external dependency with its own latency, quotas, and failure rates. If your request path includes on-the-fly query embedding, represent that call explicitly when it is not trivial.

Embedding spans are especially useful in two cases. First, they make retrieval latency explainable, because the retrieval path often begins with an embedding request before any vector search occurs. Second, they expose cost drivers that are otherwise invisible if the embedding stage is folded into a generic retrieval span. If the embedding service has a different provider, model, or quota than the chat model, that distinction belongs in the trace.

If embeddings are precomputed offline and not part of the live request path, you usually do not need to inflate every user trace with those jobs. Separate batch traces or background-job spans are more appropriate. The taxonomy should reflect user-facing accountability, not every upstream artifact in the data pipeline.

Events record meaningful milestones inside a span. Not every important moment deserves a child span. Events are the right tool when the work is part of a larger operation and does not need its own latency accounting. Streaming token chunks, retry notices, cache hits, tool-selection decisions, and guardrail outcomes are all good candidates for events if they happen inside an existing span and do not justify a separate child.

Use events when you want to preserve chronological detail without fragmenting the trace. For example, a model span may emit events for "first token received," "tool call requested," or "response truncated." Those markers help you reason about progress and control flow while keeping the span hierarchy manageable.

Do not force everything into events. If a step can fail independently, has a separately meaningful duration, or is owned by another service, a child span is usually better. Events do not give you their own end-to-end latency semantics, and that limitation matters when you are trying to answer where the time went.

Exceptions belong to the operation that failed. Failures should live as close as possible to the span that experienced them. If the model request times out, record the failure on the model span. If the retrieval service rejects the query, attach the exception to the retrieval span. If the tool call fails because a downstream API returned an error, record that on the tool span.

This placement matters because GenAI failures are often domain-specific. A model can be healthy while a tool backend is down. A retrieval request can fail while inference succeeds. An orchestrator can recover from one model failure and retry with another provider. If you flatten those into one generic failure on the root request, you lose the ability to distinguish provider outage from orchestration logic from application policy.

Keep exception metadata low-cardinality and normalized. The taxonomy is about structure, not verbose error text. You want to know which operation failed and in what category, not turn every unique message into an analytical dimension.

Links capture causality when nesting is the wrong shape. Trace trees are useful, but not every causal relationship is a parent-child relationship. Some GenAI systems fork work across services, queues, or background workers. A tool call may happen asynchronously. A retrieval refresh may be initiated by one request but complete during another. In those cases, a span link is often better than forcing a misleading parent-child edge.

Links preserve the causal association without pretending the work is strictly nested in time. That distinction is important when the work is parallel, deferred, or fan-out driven. You still keep the main trace understandable, but you avoid encoding an artificial hierarchy just to fit a diagram.

Use links sparingly and deliberately. They solve a real problem, but they also make the trace harder to scan if you overuse them. Prefer a tree when the work is genuinely nested. Use links when causality survives but strict containment does not.

The boundary rules are operational, not stylistic. The hardest taxonomy decisions appear at the boundaries. You need a span when the operation has independently meaningful latency, a distinct failure mode, a separate owner, or a different cost profile. You do not need a span for every helper function, parameter transform, or response formatter.

That rule set is simple enough to memorize and strong enough to survive real systems. A retrieval step that fans out across documents should be a span because you will want to time it. A prompt template substitution usually does not deserve one because it is cheap, local, and

not a meaningful failure domain. A tool invocation should be a span when the tool is external, slow, or operationally important. A tiny in-process callback should usually remain an event or attribute.

The same logic applies to streamed responses. If the stream is part of one model invocation, emit events or attributes inside the model span. If the streamed segments are separate service calls with their own latency and error handling, model them as separate spans. The taxonomy should follow accountability, not implementation trivia.

Avoid the oversized "chat request" span. One of the most common anti-patterns is the single giant span that hides everything. It may have a nice name, but it defeats the purpose of tracing. If your only GenAI span is "chat request," you cannot tell whether the delay came from retrieval, inference, tool execution, or orchestration. You also cannot identify which layer failed when the request breaks.

That oversimplification often happens because the first implementation is easy. The service starts with one API call, so one span feels natural. As the system evolves, the span taxonomy must evolve too. When you add retrieval, a tool, or an agent loop, the trace model has to make room for those operations explicitly. Otherwise, the observability surface lags behind the architecture.

The opposite error is over-fragmentation. If you create a span for every trivial callback, token, or local helper, the trace becomes noisy and unreadable. Analysts then stop trusting it. The right taxonomy is the one that reflects meaningful operational units, not every line of code.

A concrete modeling flow keeps the hierarchy honest. Consider a user prompt that asks for a grounded answer. The request enters an API service, which creates the root span. The service starts an agent span because it must decide whether to retrieve context, ask a model, or call a tool. The agent launches a retrieval span. Inside retrieval, an embedding request is emitted as a child span when the live

query must be vectorized. A reranking step follows as another child span because it has its own latency and can fail independently. The agent then emits a model span for the generation step. During generation, the model emits stream events as tokens arrive, and a tool call is launched as a separate span when the model asks for external data. If the tool result returns asynchronously, a link can connect the background work to the originating request without pretending it was strictly nested.

That single flow shows why the taxonomy matters. The request span gives you the total. The agent span explains the orchestration. The retrieval and embedding spans isolate context-building cost. The model span captures inference. The tool span exposes external dependency behavior. Events preserve fine-grained progress without drowning the trace in extra spans.

Use the taxonomy to preserve analytical questions. A good span taxonomy is not measured by elegance; it is measured by the questions it keeps answerable. You want to know which phase dominates latency, which operation fails most often, which provider or tool drives cost, and which orchestration path causes retries. If the trace shape makes those questions easy to answer, the taxonomy is doing its job.

That is the reason the model, agent, retrieval, embedding, and tool layers must stay distinct. They correspond to different accountability domains. Some are inference-heavy, some are orchestration-heavy, and some are data-access heavy. Once you draw those boundaries clearly, later work on token accounting, tool tracing, and retrieval latency becomes a matter of enriching a well-formed structure rather than reconstructing meaning from a flat log of custom names.

2.3. Development Status, Stability Boundaries, and Migration Rules

A GenAI rollout can look healthy right up to the point where dashboards stop agreeing with each other. The deployment succeeds, traffic flows, and span volume stays steady, but half the saved queries go blank because one library started emitting a newer semantic field set while another stayed on the older one. That is the operational cost of working with a convention set that is still in development. The schema is useful enough to standardize on, but not yet frozen enough to ignore migration discipline.

Begin with the opt-in mechanism, not with hope. When you upgrade an instrumentation package that emits GenAI semantic conventions, do not assume the emitted field set will drift automatically in a compatible way. OpenTelemetry explicitly treats the GenAI conventions as development-status, and existing instrumentations that emitted v1.36.0 or earlier are not supposed to change their defaults silently. The control point is the environment variable `OTEL_SEMCONV_STABILITY_OPT_IN`, which lets you choose the convention family and category to emit. For GenAI, the documented opt-in value `gen_ai_latest_experimental` selects the latest experimental GenAI semantic conventions supported by the instrumentation without also emitting the older shape.

A minimal deployment fragment looks like this:

```
export OTEL_SEMCONV_STABILITY_OPT_IN=gen_ai_latest_experimental

java -javaagent:/otel/opentelemetry-javaagent.jar \
  -Dotel.service.name=api-gateway \
  -jar app.jar
```

That single switch carries more weight than it appears to. It tells the SDK and instrumentation layer which schema family to emit, and it

gives you an explicit migration boundary. Without it, you risk a mixed fleet in which some services still produce old names and others emit the newer experimental ones. With it, you can stage the change intentionally and know which schema is present at each hop.

Separate stability domains before you reason about compatibility. Not all OpenTelemetry names move at the same speed. Stable tracing concepts, stable resource fields, and stable general semantic conventions are not the same thing as a development-status GenAI schema. You should treat those domains differently when you plan upgrades.

A span itself may remain structurally valid while its attributes evolve. That distinction matters. The transport, context propagation, and trace identity may work exactly as before, but the analytical meaning of the emitted attributes can still shift. If you assume that a trace being well-formed implies the GenAI fields are stable, you will miss the real failure mode: the backend ingests the spans, but the dashboard logic no longer recognizes the fields.

This is why compatibility notes need to mention more than the SDK version. You also need to know which instrumentation package generated the spans, whether an auto-instrumentation agent rewrote the fields, and whether a collector processor normalized them on the way in. A configuration that looks identical at the application layer can still emit different schemas depending on the collector and language SDK in use.

Treat schema drift as a fleet property, not a local bug. Schema drift is rarely introduced by one service alone. It appears when a fleet contains several release trains at once: a few services upgraded early, some stayed pinned, a collector transform rewrote part of the stream, and saved dashboards still assume the old names. The resulting telemetry is not broken in a syntactic sense; it is fragmented in a semantic

sense.

The dangerous part is that this fragmentation can be asymmetric. One service might expose both old and new dimensions for a while, another might emit only the new schema, and a third might map the new fields into a backend alias. That mixture can produce partial query matches that look plausible but are incomplete. A dashboard may show the right trend line while silently undercounting the services that moved first.

Plan for drift as if it were a deployment topology issue. You are not managing a single binary compatibility boundary; you are managing a schema transition across applications, collectors, storage, and queries. The correct response is not to trust that "additive" changes are safe by default. The correct response is to define exactly which fields are canonical, which are transitional, and which are deprecated.

Use a migration matrix before you roll anything to production. The simplest way to stay honest is to enumerate what each layer knows about the schema. A migration matrix is useful because it forces you to separate emission, transformation, storage, and query expectations.

Layer	Old schema	New schema	Dual-read	Dual-write
Application A	yes	no	no	no
Application B	yes	yes	yes	yes
Collector	map only	map only	yes	yes
Warehouse view	yes	yes	yes	yes
Dashboard	yes	partial	yes	no

You do not need a formal tool to do this; a spreadsheet is enough. What matters is that you can answer four questions for every layer: which schema it emits, which schema it can read, whether it can tolerate dual writes, and whether it depends on a canonicalized view.

If a dashboard depends directly on raw field names, it should be treated as fragile during migration. If a warehouse view normalizes both old and new names into one analytical model, it can absorb schema drift

more safely. If a collector is responsible for translation, you need to validate that translation against real traces before you trust the rollout.

Understand where old and new fields can coexist safely. During a transition, you may need dual emission or dual interpretation. That is sometimes acceptable, but only when the meaning of the fields is clear and the duplication is temporary. Dual emission works best when one side is explicitly deprecated and the other is canonical. It works poorly when both fields are treated as equally authoritative.

A safe coexistence strategy has three properties. First, the old and new fields must describe the same domain concept. Second, the downstream system must know which one to prefer. Third, there must be a retirement date or retirement condition. Without that third property, temporary coexistence becomes permanent clutter.

Do not let dual emission leak into forever semantics. If you leave both names in place indefinitely, you multiply cardinality, complicate queries, and make every analyst remember which version to use. That is exactly the failure semantic conventions are meant to prevent.

Validate with representative traces, not synthetic assumptions. A schema migration can pass unit tests and still break observability in practice. The reason is that dashboards and queries depend on real trace shape, not just on the presence of attributes. You need representative traces that cover the common code paths: successful model calls, timeout paths, streaming responses, tool failures, retrieval fan-out, and retry behavior.

A practical validation loop looks like this:

```
1. Enable the new semconv mode in staging.
2. Capture traces from common and failure paths.
3. Diff span names and attribute keys.
4. Check collector translations and views.
5. Run saved queries against the staged data.
6. Compare counts, latency percentiles, and error rates.
7. Promote only after the schema matches expectations.
```

That process is deliberately repetitive because it catches different classes of failure. A key diff can reveal a renamed field. A query check can reveal a dashboard that still points at the old name. A percentile comparison can reveal that part of the fleet is double-counting or dropping records during a transformation.

Synthetic traces are useful for smoke tests, but they are not enough. The real risk lies in interactions between instrumentation, retries, streaming, and collector transforms. Those interactions only show up in representative traffic.

Keep the compatibility boundary explicit in code and configuration. If you build internal wrappers around instrumentation, expose the semconv choice as a deliberate configuration point. Do not bury it in defaults that only one team knows how to change. When a migration boundary is explicit, platform teams can flip a narrow switch and know what changed. When it is implicit, the same release can alter both semantics and alert behavior without a visible control plane.

A concise configuration policy helps:

```
# Canonical schema choice for GenAI telemetry.
OTEL_SEMCONV_STABILITY_OPT_IN=gen_ai_latest_experimental

# Keep the service name stable across schema transitions.
OTEL_SERVICE_NAME=api-gateway

# Record the active semconv version in deployment notes.
GENAI_SEMCONV_PROFILE=latest-experimental
```

The extra metadata variable is not an OpenTelemetry requirement; it is an operational convenience. It gives you a deployment-time record of what schema mode a service was expected to use. That helps during incident review when two teams disagree about what the service was emitting.

Do not let collector transforms become a black box. Collector-side translation can make migrations survivable, but only if you treat

the transform as versioned code. A collector processor that maps one GenAI schema to another is part of the contract, not a throwaway filter. You should test it with real traces, pin its behavior to a known semconv mapping, and document what it does for fields it cannot translate.

The risk is subtle. A collector can successfully preserve some fields while dropping or rewriting others. That can make a dashboard look mostly correct while quietly erasing the dimensions needed for cost attribution or error analysis. If the transform is partial, analysts need to know which attributes survived and which were normalized away.

The right pattern is to make translation transparent: emit the original field set when possible, expose the normalized view downstream, and document the mapping at the collector boundary. If you cannot preserve a field, say so explicitly rather than letting the absence masquerade as intentional emptiness.

Version notes belong with the schema, not in tribal memory. Because GenAI semantic conventions are in active development, release notes matter. A service owner should be able to answer, for any given deployment, which schema family was emitted, which SDK or instrumentation package produced it, and whether the collector or backend applied any compatibility mappings.

That information should live near the deployment artifact, not only in a meeting note or chat thread. If you cannot reconstruct the active schema from configuration, the next migration will be slower and riskier than it needs to be. The practical standard is simple: make the semconv mode visible in deployment manifests, pin instrumentation versions when necessary, and record the translation path in the system's runbook.

Treat dashboard resilience as a migration requirement. If a dashboard cannot survive a semconv change, the dashboard depends too directly on the raw producer schema. During migration, you want a

stable analytical layer that can read both the transitional and canonical forms. That layer may be a warehouse view, a saved query abstraction, or a backend mapping rule. What matters is that the dashboard points at the stable layer, not at raw service-specific names.

This is especially important for alerting. An alert tied to a transient schema can go silent when a service upgrades or can fire incorrectly if a field is renamed but still present in another form. You should validate alerts against both schemas during the transition and retire the old query only when the new view has proven stable over real traffic.

The same advice applies to cost and quality metrics. If token counts, tool calls, or retrieval latency are derived from attributes that are still moving, the derived metrics need a schema-normalized input layer. Otherwise, you will spend more time explaining discrepancies than using the metrics.

Plan the migration as a staged control problem. The operational sequence is straightforward if you treat it as a control problem rather than a documentation update.

First, identify the schema boundary and the services that cross it.

Second, decide whether you will dual-write, dual-read, or translate at the collector.

Third, stage the change in a non-production environment with representative traces.

Fourth, compare queries, alerts, and rollups against the old baseline.

Fifth, roll out gradually, watching for mixed-schema behavior.

Sixth, retire the old path only after the fleet and the analytics layer agree on the new contract.

That sequence reduces surprise because every phase has a known ver-

ification point. You are not asking whether the new schema is "better" in the abstract. You are asking whether the new schema can replace the old one without breaking the operational system that depends on it.

Expect the transition plan to evolve. Development-status conventions are temporary by design. The migration guidance is also temporary and can change again before the GenAI conventions become stable. That is not a flaw; it is the normal cost of working in a fast-moving schema space. What you can rely on is the discipline of explicit opt-in, pinned versions where needed, and documented translation boundaries.

If you keep the schema boundary explicit and treat migration as a controlled rollout, the observability surface can evolve with the GenAI stack instead of lagging behind it. The traces stay useful, the dashboards keep their meaning, and analysts can compare behavior across releases without reconstructing history from renamed fields.

2.4. Designing Internal Telemetry Contracts Around Evolving Semconv

The practical challenge is not just that GenAI semantic conventions change; it is that every layer above them assumes they will not. Application code wants a stable helper API. Dashboards want durable field names. Analysts want query continuity. If you let development-status conventions leak directly into all of those places, every schema change becomes a fleet-wide coordination event. The better pattern is to define an internal telemetry contract that absorbs semantic convention churn at the boundary and keeps the rest of the organization on stable ground.

Start with a canonical internal model. You need one place

where GenAI meaning is expressed in organization-stable terms, even if the emitted OpenTelemetry schema changes underneath it. That model should include the concepts your analysts actually reason about: provider, requested model, served model, operation type, token usage, tool name, retrieval target, latency phase, and normalized error type. Those are analytical dimensions, not raw wire names.

A minimal internal model can look like this:

```
type GenAIRecord struct {
    Provider        string
    RequestedModel  string
    ServedModel     string
    Operation       string
    LatencyPhase    string
    InputTokens     int64
    OutputTokens    int64
    ToolName        string
    RetrievalTarget string
    ErrorClass      string
    SchemaProfile   string
}
```

This structure is deliberately not the OpenTelemetry schema. It is your contract with the rest of the organization. It lets you keep one stable analytical vocabulary while mapping multiple semconv versions into and out of it. When the upstream schema evolves, you update the adapter, not every dashboard and service.

Separate emission from meaning. The most common mistake is to treat the telemetry payload as the contract. That works until the schema changes. Instead, make the application emit a deliberately small, stable set of internal concepts through a helper library, then translate those concepts into the current semconv shape at the edge.

The internal helper should answer questions such as: what operation is this, what model was requested, what provider routed the call, and what failure class occurred? The helper should not expose raw official field names to every call site. That keeps the application code insulated

from semconv churn and prevents each team from inventing slightly different mappings for the same idea.

A clean pattern is:

```
app code -> internal helper -> semconv adapter -> span
```

The helper owns meaning. The adapter owns compatibility. The span is just the transport vehicle.

Use adapters as translation boundaries, not as business logic dumps. An adapter layer should convert between the internal model and the active semconv version. It should not contain business decisions such as retry policy, prompt policy, or tool selection. Those belong in application logic. The adapter's job is narrow: map fields, normalize enums, attach required attributes, and preserve version context.

That narrow scope matters because adapters are where drift usually hides. If one service starts using the adapter to infer providers from hostnames, while another injects provider identity explicitly, you get inconsistent telemetry again. Keep the adapter deterministic and boring. If it cannot map a concept cleanly, surface that gap in validation instead of inventing a convenient guess.

Record the schema profile explicitly. You cannot translate or query safely unless you know which schema a record was produced under. For that reason, the emitted telemetry should carry version context. In OpenTelemetry terms, the schema URL and instrumentation semconv mode are the most useful pieces of metadata; in an internal model, you can also record a schema profile name that your organization controls.

That profile lets you answer operational questions later: Was this trace emitted under the old GenAI field set or the latest experimental one? Which collector transform applied? Did the query view normalize the

record already, or are you reading raw spans? Without that context, a mixed fleet becomes hard to interpret and nearly impossible to backfill correctly.

A practical deployment might propagate the mode in both the environment and the emitted record:

```
OTEL_SEMCONV_STABILITY_OPT_IN=gen_ai_latest_experimental
GENAI_SCHEMA_PROFILE=latest-experimental
```

The environment variable guides emission. The internal profile field guides downstream interpretation. Those are related but not interchangeable.

Keep internal names out of the OpenTelemetry namespace. If you need an organization-specific field that does not yet exist in semconv, place it in your own namespace instead of minting a custom attribute under an OpenTelemetry prefix. That avoids collisions with current and future conventions and makes the ownership boundary obvious.

For example, do this:

```
acme.genai.request_tier=gold
acme.genai.prompt_policy=safe
```

Do not create unofficial fields under reserved namespaces just because they look convenient. Reserved naming rules are part of what makes the ecosystem interoperable. If your internal dimension later becomes generally useful, you can translate it into a standard field when one exists. Until then, keep it clearly local.

Centralize field construction in a shared library. If every service hand-assembles span attributes, the fleet will drift even if everyone is trying to follow the same guidance. Centralize semantic field construction in a shared helper library so that provider names, model identifiers, token counts, and error classes are normalized in one place.

That library should expose intent-oriented functions rather than raw attribute setters. For example, a caller should say "record model invocation" or "record retrieval phase" instead of populating five string keys manually. The library can then map to the active semconv version, enforce allowed values, and attach compatibility metadata.

This is where the organizational contract becomes enforceable. If you allow each team to write directly against raw attributes, you have no practical control over schema drift. If you control field construction centrally, you can update mappings once and propagate the change safely.

Translate at the collector when fleet heterogeneity is unavoidable. Application wrappers are the best place to keep meaning stable, but they are not always sufficient. In a mixed fleet, some services will upgrade sooner than others, and some will run older SDKs for longer than you would like. Collector-side translation gives you a second compatibility boundary. It can normalize multiple emitted shapes into one downstream analytical view.

That boundary is especially useful when one backend must support both old and new traces during a migration window. A collector processor can map the raw span attributes into a canonical set before storage or before export to the analytics backend. The output then becomes easier to query, even if the fleet is still transitioning.

The trade-off is that collector transforms can hide schema differences if you do not document them. That is why the transform needs a versioned mapping table and an explicit retirement plan. The collector should not become a black box that silently rewrites meaning. It should be a controlled translation layer with known coverage.

Design query views around stable organizational aliases. Raw telemetry fields should not be the only way analysts interact with the data. Build a query view or materialized mapping that exposes

organization-stable aliases for the dimensions you care about most. Those aliases can point to different raw fields depending on the schema profile, but the analyst sees one consistent model.

For example, a warehouse view may unify several semconv variants into:

- `genai.provider`
- `genai.requested_model`
- `genai.served_model`
- `genai.operation`
- `genai.error_class`

That approach protects dashboards from raw field churn. It also lets you define business-facing aliases, such as `customer_facing_model` or `billing_provider`, without polluting the instrumentation layer with local terminology. Keep the query layer stable and the instrumentation layer flexible.

Preserve analytical intent when mapping between schemas. A semconv migration is not just a rename exercise. A correct mapping must preserve what analysts mean by the field, not merely copy string values into new keys. That distinction matters for concepts like requested model versus served model, provider identity versus API gateway identity, and operation name versus phase label.

If the upstream schema does not encode a concept directly, do not manufacture it by guessing. For example, if you only observe a gateway hostname, that is not automatically a provider name. If you only observe a routed deployment alias, that is not necessarily the model family. The adapter should preserve uncertainty when the source data is ambiguous. It is better to record “unknown” or “unresolved” than

to emit a plausible but incorrect label that will contaminate cost and quality analysis.

That principle is important in GenAI observability because provider routing, proxy layers, and model aliases can all differ from the actual serving backend. Your internal contract should distinguish these concepts explicitly so that future queries do not collapse them into one field.

Use low-cardinality normalized values. Internal contracts are only helpful if they remain query-friendly. Normalize values that become dimensions: operation names, provider identifiers, error classes, and latency phases should come from a controlled vocabulary. Do not let raw exception messages or request IDs leak into dimension fields.

A good normalized error class might distinguish `timeout`, `rate_limit`, `model_error`, `invalid_request`, and `upstream_unavailable`. Those categories are coarse enough for aggregation and fine enough for operational investigation. Raw stack traces, provider payloads, and free-form messages belong elsewhere, if they are captured at all.

The same rule applies to model identifiers. Use the canonical model family or deployment alias that analysts actually query, not an unbounded string from the request body. If you need both, separate them into requested and served fields. That separation keeps rollups stable and prevents accidental cardinality spikes.

Treat version support as a compatibility matrix. Once you define an internal contract, document which semconv profiles it can ingest and which downstream views it feeds. The matrix should capture at least three states: old schema supported, latest experimental schema supported, and mixed-mode translation supported. If a library or collector cannot handle one of those states, say so explicitly.

A simple compatibility matrix might read:

```
Internal view      Old GenAI    Latest exp.    Mixed fleet
raw spans             yes           yes           no
collector view        yes           yes           yes
warehouse view        yes           yes           yes
saved dashboard       yes           yes           yes
```

That table forces you to be honest about what is actually safe. A service may emit both schemas during rollout, but your direct raw-span queries may still be unsafe until the collector view is normalized. The matrix becomes part of the operational contract, not just documentation.

Make migrations reversible where possible. If you cannot roll back a semconv migration, you are taking unnecessary risk. Reversibility does not mean the old schema will remain forever. It means you can move back to a known-good state if a field mapping or collector transform breaks an important query.

The practical tactic is to keep the old mapping available until you have validated the new one on real traffic. That may mean dual-write for a period, or it may mean preserving both query paths behind a view. The key is that your adapters and views should not assume a one-way move. GenAI telemetry changes often enough that reversibility pays for itself quickly.

Validate the contract with representative workloads. A contract is only durable if it survives the actual traces your system emits. Test the full chain: instrumentation helper, emitted span attributes, collector transform, storage view, and dashboard query. Include streaming responses, retries, tool calls, retrieval fan-out, and error paths. Those are the places where schema assumptions break.

A practical test harness can emit a trace and then print the normalized output:

```
genai_span(
    provider="acme-proxy",
```

```
    requested_model="gpt-4.1",
    operation="chat",
    input_tokens=1294,
    output_tokens=311,
    error_class="none",
)
```

Expected normalized output:

```
genai.provider=acme-proxy
genai.requested_model=gpt-4.1
genai.operation=chat
genai.input_tokens=1294
genai.output_tokens=311
genai.error_class=none
schema_profile=latest-experimental
```

That kind of test is not decorative. It confirms that the contract still means what you think it means after translation. If a collector mapping drops a field or merges two concepts, you will see it immediately.

Apply the same contract discipline to dashboards and alerts. Dashboards are consumers of the contract, not the source of truth. If a panel references raw span keys directly, you will need to rewrite it whenever the semconv changes. Instead, point dashboards at the stable query view or alias layer. Alerts should follow the same rule.

This keeps your operational posture sane during schema churn. Analysts can keep their mental model fixed while the instrumentation layer evolves. The backend can migrate from one semconv profile to another, and the saved query surface stays intact. If you later need to expose a newly standardized field, you can add it behind the stable view without forcing every chart to change at once.

Choose the right adaptation layer for the right kind of change. Not every schema change belongs in the same place. If the change is local to one service, handle it in the application helper. If it affects every service but the meaning is stable, centralize it in the shared library. If

the fleet is mixed and the backend must normalize multiple versions, use the collector. If analysts need a durable interface over time, add a query view.

Those layers are complementary, not redundant. The helper keeps producers aligned. The collector keeps transport consistent. The query view keeps consumers insulated. When all three layers are present, a semconv update becomes a controlled translation exercise instead of a wholesale rewrite of the observability estate.

The result is an observability system that can absorb schema change without making every team move in lockstep. The telemetry stays interpretable, the dashboards stay durable, and the internal contract gives you room to adopt richer GenAI semantics as they mature without turning every release into a schema migration ceremony.

Chapter 3

Tracing Inference and Attributing Token Costs

Inference is where GenAI systems turn architecture into spend, latency, and user-visible behavior. This chapter shows how to instrument that path so traces answer not just what happened, but which model call consumed the tokens, which business action caused the spend, and why the final cost picture is often wrong unless streaming and late-bound usage are modeled carefully.

3.1. Model Invocation Spans for Chat, Completion, and Agent Calls

A model call is often the most expensive operation in the request path, and it is usually the one that shapes the user-visible outcome. If you trace it only as a generic HTTP client request, you lose the very boundary that determines latency, token spend, and success semantics. The

canonical unit of analysis is the inference span: one span per logical attempt to obtain a model-generated response or tool request, regardless of whether the call comes from a chat endpoint, a text completion endpoint, or an agent planner that delegates to a provider model.

A minimal instrumentation shape makes the boundary concrete. You start the span when the application has committed to a specific inference attempt and has enough context to identify the operation, the model, and the parent business action. You end it when the provider has returned the terminal result for that attempt, including a final streaming frame if the API is streaming. In OpenTelemetry GenAI conventions, this span is the client-side inference span, and its recommended name combines the operation and model identity, such as `chat gpt-4.1` or `generate_text claude-3.7`. The point of that naming pattern is not aesthetics; it keeps aggregation stable while preserving the model dimension that drives both latency and cost.

```
from openai import OpenAI
from opentelemetry import trace

tracer = trace.get_tracer(__name__)
client = OpenAI()

with tracer.start_as_current_span("chat gpt-4.1") as span:
    span.set_attribute("gen_ai.operation.name", "chat")
    span.set_attribute("gen_ai.request.model", "gpt-4.1")
    span.set_attribute("gen_ai.system", "openai")
    span.set_attribute("tenant.id", tenant_id)
    resp = client.responses.create(
        model="gpt-4.1",
        input="Summarize the incident in one paragraph."
    )
    span.set_attribute("gen_ai.response.finish_reason", "stop")
```

That example is intentionally small. The important detail is that the span belongs to the logical inference attempt, not to the SDK call site. The application handler may do input validation, prompt assembly, safety gating, and policy checks before it reaches the model. Those are real work, but they are not the inference span. They belong in the

parent request or orchestration span because they answer a different question: why did the application choose this prompt and this model at this moment? The inference span answers a narrower one: what happened while the provider generated the model output?

Where the span begins. Start the inference span after you have selected the model and request mode, but before the outbound provider call leaves your process. That timing matters because model choice is part of the observable decision, while socket setup, serializer overhead, and retry scheduling are implementation details. If you begin the span inside a low-level SDK wrapper, you risk attributing orchestration latency to the provider. If you begin it too early, you contaminate the span with prompt construction and validation work that should be separately measurable.

For chat and completion workloads, the span usually begins once the request payload is complete and the application has resolved the exact provider invocation. For agent workloads, the same rule applies, but the caller is usually a planner or executor span that has already decided which subtask requires inference. That parent span should stay open across the control-flow decision, while the child inference span should capture only the model attempt itself. This distinction lets you answer whether a long request spent time reasoning about the next action or whether the model itself was slow.

The child relationship is the critical causal link. The inference span should be a descendant of the application operation that made the decision to call the model, not merely a sibling of a transport span that happened to emit the HTTP request. In practice, that means your tracing context should flow from the request handler into orchestration code, then into the provider SDK wrapper, and finally into the transport layer. The transport layer may create its own internal span, but that span should remain subordinate to the inference span unless you need it for debugging a specific network failure mode. Most of the time,

you do not want the transport span to become the primary analytical unit, because it hides the semantics of the model call behind HTTP mechanics.

Where the span ends. End the inference span when the provider has produced the authoritative terminal state for the attempt. For a buffered response, that is the point where the complete response and final usage are available. For a streamed response, that is the final chunk or terminal event that closes the stream, not the first token emitted to the client. A model that begins streaming quickly but completes slowly still occupies the inference span until the provider has closed the logical attempt. If the client disconnects, the inference span should end with cancellation status or an error status that records the true termination condition. Do not mark the span successful merely because some content was delivered before the failure.

This end boundary becomes especially important when the application retries the call. A single user-visible request may contain several inference attempts: the first times out, the second succeeds, or a fallback model replaces the primary one after a policy check. Each attempt deserves its own child inference span. Collapsing all attempts into one span erases the retry cost, distorts latency percentiles, and makes provider failover look cheaper than it is. If you need a logical umbrella for the user request, use the parent orchestration span to group the attempts. That span can carry request identifiers, tenant identity, experiment cohort, and feature flags; the child spans carry the individual model attempts.

Canonical attributes. The inference span should carry normalized attributes that describe the model invocation in a provider-neutral form. At minimum, include the operation name, provider or system, request model, and a stable status or finish reason. The operation name should distinguish the broad invocation type, such as chat, text generation, embedding, or agent-directed inference. The provider or system

attribute identifies the vendor or serving stack. The request model attribute records the exact model target, and it should be the same value the provider used for routing whenever possible.

The span is also the right place for invocation metadata that affects interpretation of the result. Include conversation or thread identifiers if the application uses them, because they establish whether the call belongs to a new session or a continuation of prior context. Include the response format or output mode when it materially changes semantics, such as structured output versus free-form text. Include the tool-call intent only when the model is asked to emit a tool request; the tool execution itself belongs to a separate span in the later tool-tracing layer. Keep the invocation span focused on the model attempt so that analysis does not have to disentangle model generation from downstream execution.

You should prefer stable, normalized keys over provider-specific display fields. A model label shown in a dashboard may differ from the internal identifier used for billing, routing, or fine-tune selection. If you store only the display name, you will create broken joins later. Preserve provider-native identifiers in extension attributes when you need lossless analysis, but keep the standard GenAI attributes authoritative for aggregation. That pattern gives you a clean analytical layer without discarding the details needed for debugging rare provider-specific behavior.

Agent calls are still inference calls. Agent systems complicate the control flow but not the core boundary. When a planner decides to query a model for reasoning, summarization, or action selection, that model call is still an inference span. What changes is the parentage. The agent workflow span should represent the higher-level unit of work, such as `plan`, `invoke`, or `execute`, and each model call should sit beneath it as a child inference span. If the planner makes three separate model calls while decomposing a task, you should see three

separate inference spans. That structure tells you whether the cost came from a single expensive generation or from repeated planning iterations.

This separation also protects your analysis when the agent alternates between reasoning and acting. The agent span can contain the business context: which user action initiated the workflow, which policy bucket applied, which experiment was active, and which goal the agent pursued. The inference span then captures the provider-specific attempt that served that workflow. If you later add tool execution spans, those will sit alongside the inference spans under the same workflow parent, not inside the inference span itself. That hierarchy keeps the causal story intact without blending model time, tool time, and orchestration time into one opaque block.

Retries, failover, and transport spans. Retries are the easiest place to destroy trace fidelity. If the SDK retries a failed call internally and you only observe the final success, you miss the cost of the failed attempt. The better pattern is to model each attempt explicitly. The outer request or workflow span remains stable. Each attempt gets its own inference span, and each transport attempt, if you need it at all, remains nested beneath the corresponding inference span. This gives you an honest accounting of latency and failure rate across providers and models.

Provider failover should follow the same logic. If the primary model is unavailable and the application falls back to a secondary model, the fallback inference must be represented as a separate child span with its own request model and system attributes. Do not mutate the original span to make it look as if the fallback was the intended target all along. That would corrupt both routing analytics and cost attribution. Instead, record the fallback decision on the orchestration span, then emit a new inference span for the alternate provider call. The trace then shows exactly where the application spent time and money.

Transport-level spans only deserve first-class visibility when network behavior itself is operationally important. A provider SDK may expose an HTTP client span, a DNS lookup span, or a retry policy span. Those can be useful for debugging outages, but they are not the canonical analytical unit for model usage. Keep them subordinate unless you have a concrete reason to study network latency separately from inference latency. The model invocation span should remain the semantic anchor.

What not to put on the span. Avoid attaching mutable business state that changes during the request lifecycle. The inference span is not a dumping ground for the full prompt, the full response, every intermediate prompt rewrite, or every internal planning artifact. Those may belong in logs, events, or redacted payload storage depending on your privacy policy, but they do not belong as unconstrained attributes on the span. Excessive attribute churn makes queries brittle and increases storage cost without improving the definition of the invocation boundary.

Avoid mixing usage accounting into the span shape before the usage is authoritative. The span can certainly carry token counts later, but the boundary must remain independent of whether usage arrives synchronously, in a terminal stream event, or through a reconciliation pass. Likewise, do not encode business revenue metadata on the inference span unless that metadata is truly part of the invocation contract. Tenant, feature, and experiment identifiers are usually enough; order value and subscription tier belong closer to the request context or the business event stream.

A practical lineage model. A useful mental model is a three-tier chain: request span, orchestration span, inference span. The request span captures ingress into your service boundary and carries the external request identity. The orchestration span captures the application's decision-making, such as prompt selection, policy enforcement, and agent planning. The inference span captures one attempt to obtain a

model-generated result. This chain is what preserves causality. You can ask why a user action triggered a given model call, what decision logic led to that model choice, and how expensive that specific attempt was.

That lineage becomes the basis for all later analysis. Once the invocation span is stable, token usage can attach to a known unit of work, retries can be counted honestly, and provider failover can be compared across models. The model call stops looking like an incidental network transaction and becomes what it actually is: the semantic and economic center of the application path.

3.2. Token Usage as a First-Class Observability Dimension

Token counts are easy to dismiss when you first see them in a provider payload. They look like incidental metadata, useful for billing after the fact but not worth the rigor of a real telemetry schema. That attitude fails quickly in production. The difference between a 2,000-token prompt and a 20,000-token prompt is not a cosmetic difference, and neither is the difference between output tokens that were freshly generated and input tokens that were read from cache. If you do not model those categories explicitly, you cannot tell whether cost growth comes from user behavior, prompt bloat, poor cache reuse, or a regression in the model path itself.

The practical rule is simple: treat token usage as a first-class measurement, not a blob. Put it on the inference span, preserve the categories the provider exposes, and normalize those categories into stable fields that can be aggregated across tenants, models, and experiments. OpenTelemetry's GenAI conventions follow that model by attaching token usage directly to spans and distinguishing input, output, cache-read,

and cache-creation counts. In the same ecosystem, OpenInference conventions reinforce the same design choice: usage belongs in telemetry, not only in logs or ad hoc billing records.

```
from opentelemetry import trace

with tracer.start_as_current_span("chat gpt-4.1") as span:
    span.set_attribute("gen_ai.operation.name", "chat")
    span.set_attribute("gen_ai.request.model", "gpt-4.1")
    span.set_attribute("gen_ai.usage.input_tokens", 812)
    span.set_attribute("gen_ai.usage.output_tokens", 176)
    span.set_attribute("gen_ai.usage.cache_read.input_tokens", 384)
    span.set_attribute("gen_ai.usage.cache_creation.input_tokens", 0)
    span.set_attribute("tenant.id", tenant_id)
```

That shape gives you a durable record of what the model consumed on that attempt. The counts may originate from the final response payload, a terminal stream event, or a later reconciliation call, but the span is still the right attachment point because it is the unit that already captures the invocation boundary. Once the counts are on the span, you can build metrics, dashboards, and cost models without depending on provider-specific response schemas.

Use categories, not a single number. A single `tokens` field is too coarse for operational analysis. Prompt or input tokens answer a different question from completion or output tokens. Input growth usually points to prompt engineering, conversation history, retrieval overhead, or context-window pressure. Output growth usually points to user intent, generation style, or model behavior. Cached input tokens reveal whether prior context was reused economically. Cache-creation tokens tell you when the application paid to seed future reuse. Total tokens are useful as a denominator for throughput, unit cost, and saturation analysis, but they do not explain where the spend came from.

This is why the normalized fields matter. If you collapse all categories into one total too early, you lose the ability to tell whether a product change increased prompt size, whether an agent repeated the same con-

text across subcalls, or whether a caching strategy is actually reducing billed input. Keep the categories separate on the span and derive totals only when you need them for presentation or rate calculations. That preserves analytical fidelity while still allowing simple dashboards to show total spend.

Some providers expose more detail than the canonical fields. You may see reasoning tokens, audio tokens, multiple choice variants, or service-specific subcounts. Keep those as provider-native extension attributes when they matter to your analysis, but do not force them into the standard fields unless the semantics truly align. The normalized fields should remain stable across providers so that your metrics do not break when you switch vendors or SDK versions. Provider-specific detail should increase precision, not redefine the schema.

Attach usage to the attempt, not to the request. An easy mistake is to attach token usage to the outer API request span instead of the individual inference span. That seems harmless until retries, fallback models, or multiple candidate generations enter the picture. A single request may produce several inference attempts, and each attempt may have a different token profile. If you pin all usage to the request span, you smear those attempts together and make attribution impossible.

Keep the outer request span for business context and lifecycle boundaries. Put usage on the child inference span that represents the specific model attempt. That lets you analyze the cost of the primary attempt separately from the cost of a retry or fallback. It also keeps per-attempt totals honest in agent workflows, where a planner may issue repeated calls while decomposing a task. If the planner makes three inference attempts, you should see three separate usage records, even if they all belong to one end-user request.

That distinction also matters for idempotency and replay. If you replay a request for debugging or recovery, the new attempt should produce

its own usage record rather than mutate the historical span. Observability data should reflect what happened at runtime, not what a later analysis tool inferred should have happened. Preserving attempt-level granularity is what makes cost analysis and incident review trustworthy.

Preserve absence, zero, and unknown as different states. Token usage fields are only useful if you preserve their semantics precisely. An absent field does not mean zero. It means the provider did not supply the count, or the instrumentation did not yet know it. A zero value means the provider explicitly reported none of that category. An unknown value means you know the category exists, but the provider or SDK did not expose it in a usable form.

Those distinctions matter in aggregation. If a streaming API delivers usage only at the end of the stream, the intermediate span state should not fabricate zeros. If a provider does not report cached input separately, do not backfill it with the difference between total input and visible prompt length unless that inference is explicitly valid in your system. Otherwise you are manufacturing precision that the source data does not support. The cost of being honest about uncertainty is far lower than the cost of building dashboards that quietly lie.

The same rule applies to derived totals. If you compute `gen_ai.usage.input_tokens` by summing prompt and cache-read counts, do it only when the provider defines those categories in a way that makes the sum authoritative. If a provider already supplies a total input count, prefer that total and keep the subcategories as supporting dimensions. That avoids drift caused by rounding, delayed reconciliation, or hidden token classes that the provider included but your local sum did not.

Model the fields so they aggregate cleanly. Token usage becomes operationally useful only when it can be grouped across stable

dimensions. You typically want to slice by model, endpoint, tenant, feature, experiment, prompt version, service tier, or request class. Those are the dimensions that answer who spent the tokens, on what path, and under which configuration. The usage fields themselves answer how many tokens were consumed. Together they let you ask useful questions such as whether a new prompt version increased input size, whether one tenant's workload uses more cached context, or whether a particular experiment cohort produces longer completions.

The schema choice is a trade-off. Rich dimensionality improves analysis, but every extra label increases storage cost and cardinality pressure. You do not want to copy every possible business attribute onto every token metric. Pick dimensions that are stable enough to support long-lived dashboards and cost reports. Tenant and model usually qualify. Ephemeral values such as request IDs, user IDs, or raw prompt hashes usually do not belong on aggregate metrics because they explode cardinality and make rollups expensive or impossible.

A practical pattern is to keep the span richly annotated and the metric stream more selective. The span can carry the exact request model, endpoint name, tenant, feature, and experiment cohort. A derived metric can emit token counters grouped by a smaller set of stable labels, such as model, tenant class, and endpoint family. That gives you the right balance: detailed evidence remains available for trace drill-down, while metrics stay compact enough for fast dashboards and efficient storage.

Normalize across provider schemas without erasing meaning. Provider APIs often differ in how they describe token usage. One service may report prompt and completion tokens. Another may split input into cached and uncached portions. A third may include reasoning or audio categories. Some providers expose final totals in the response payload; others expose them in a stream terminator or a separate usage endpoint. The telemetry layer should normalize those differ-

ences into a common vocabulary without pretending the sources were identical.

The canonical fields give you that vocabulary. If the provider exposes total input usage, populate `gen_ai.usage.input_tokens`. If it also exposes cached input usage, populate `gen_ai.usage.cache_read.input_tokens`. If it exposes cache creation, populate `gen_ai.usage.cache_creation.input_tokens`. If the provider splits output into multiple subcategories that are meaningful for your organization, record them as extension attributes or derived metrics, but do not sacrifice the normalized base fields. That way, a dashboard written against the common schema keeps working even when the provider schema changes.

This normalization also protects you from language drift. Different SDKs use different terms for nearly the same idea: prompt versus input, completion versus output, cache hits versus cached reads. Pick one canonical model for internal analysis and map every provider payload into it at the boundary. If you keep the original provider fields only in raw logs, you make every analyst learn each vendor's naming quirks. If you normalize up front, the observability layer becomes portable and queryable.

Choose the right attachment point for streaming and late usage. Streaming complicates usage capture because the model may start sending content before the final counts are available. The safe approach is to keep the inference span open through the full lifecycle of the stream and attach the authoritative usage values only when they arrive. If the provider emits usage in a terminal chunk or completion event, set the usage attributes there before closing the span. If a later reconciliation endpoint provides better totals, update the analytical store from that source while preserving the original span's attempt identity.

Do not fill the gap with speculative estimates unless you explicitly separate estimates from authoritative values. Estimated token counts can be useful for near-real-time dashboards, but they should live in a different field or metric series from finalized usage. A production dashboard that mixes estimates and final counts will oscillate and create false anomalies. The more disciplined pattern is to keep the span as the source of truth for the attempt, then derive a finalized metric in a post-processing step when the provider data is complete.

This matters even more for multi-candidate generation. Some APIs can return several alternatives or best-of candidates, and the billed usage may exceed the visible answer size. The user sees one answer, but the provider may have generated more than one candidate behind the scenes. If you only count visible text length, you will systematically understate true token usage. The span must therefore reflect the provider's accounting, not the apparent length of the returned content.

Treat token usage as a control variable, not just a billing artifact. Once token counts are on spans and normalized into metrics, they become a control variable for operational decisions. A higher input-token count can explain latency inflation, queue pressure, or model selection changes. A higher output-token count can explain longer streaming duration or increased downstream bandwidth. A rise in cached input can indicate better reuse and lower marginal cost. A drop in cache creation may reveal that your application stopped reusing context effectively, even if user-facing latency stayed flat.

This is where the observability dimension becomes more than accounting. You can compute tokens per successful request, output tokens per minute, input tokens by tenant, cache hit ratio by endpoint family, or completion tokens per experiment cohort. Each of those views answers a different operational question. None of them is available if token usage remains buried inside raw response JSON or collapsed into a single opaque total.

You can also relate token usage to SLOs and error budgets. A small increase in average tokens may be acceptable if it buys a large increase in answer quality. A large increase in tokens with no product gain may justify prompt refactoring or routing changes. The telemetry does not make the decision for you, but it provides the evidence needed to make the decision without guessing.

Avoid the common anti-patterns. Several patterns reliably produce misleading data. Storing token usage only in logs is one of them. Logs are good for forensics, but they are poor for aggregation and expensive to query at scale. If the only durable record of usage is buried in free-form logs, you will end up with ad hoc scripts and inconsistent totals. Put the usage on the span and, if needed, also forward it to logs for debugging.

Another anti-pattern is mixing per-attempt and per-request totals. A request with retries may double or triple its spend, but a dashboard that records only the final successful attempt will undercount actual usage. A related mistake is summing cached tokens into prompt tokens without preserving the cache category. That destroys the ability to study cache behavior and makes prompt growth look worse than it is. Keep the categories separate and derive combined views only when the analysis explicitly calls for them.

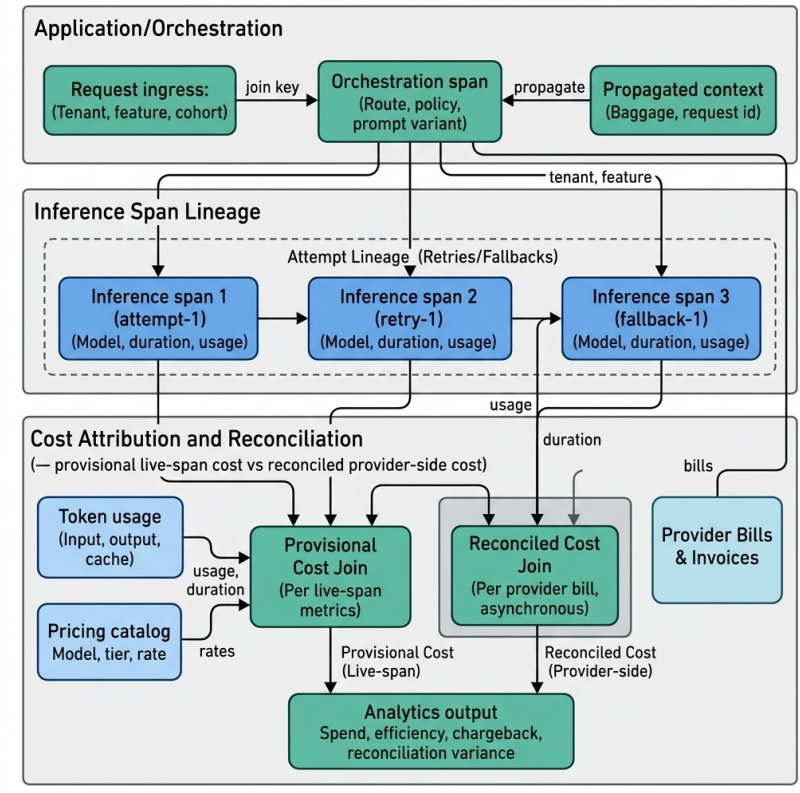
Finally, do not treat provider-estimated usage as stable if the provider documentation says it may arrive later or be reconciled asynchronously. Some systems expose one number at stream time and a more authoritative number later. If you copy the early number into final dashboards without revisiting it, you will create systematic error. Capture the source and timing of each usage value so that later reconciliation can overwrite provisional counts cleanly.

The practical outcome is a telemetry schema that does not merely record that a model call happened. It records what it consumed, which

categories of consumption mattered, and which business slices those counts belong to. That makes token usage queryable in the same way you query latency or errors: by model, by tenant, by feature, by experiment, and by time.

3.3. Joining Duration, Usage, and Business Context for Cost Analysis

A cost spike rarely has a single cause. Finance sees spend; product sees traffic; engineering sees traces. The useful explanation emerges only when you join those views onto the same inference attempts and the same business context. A new assistant feature may not increase request count at all, yet still double spend because it lengthens prompts, increases retries, or shifts traffic onto a larger model. The telemetry already contains the evidence, but you only get a trustworthy answer if you correlate duration, token usage, request lineage, and propagated business metadata before you apply any pricing logic.



The core separation is conceptual as much as technical. Telemetry capture records what happened: the span hierarchy, the token counts, the timing, and the propagated attributes. Analytical attribution decides which business entity should absorb the spend: a tenant, a feature, an experiment cohort, a workflow, or a fallback path. Pricing computation applies a rate table and any provider-specific billing rules. OpenTelemetry gives you the first layer directly and the second layer indirectly by preserving stable join keys. It does not compute the chargeback amount for you, and that is a feature, not a gap, because pricing policy belongs to finance and platform governance rather than instru-

mentation code.

Use the span lineage as the primary join path. The first join you should make is not between cost and user ID, but between cost and the span lineage that explains why the model was called. The request span identifies the external transaction. The orchestration span identifies the application decision that led to inference. The inference span identifies the concrete model attempt. When you keep those layers distinct, you can allocate spend to the business operation that actually consumed tokens rather than to whichever library emitted the HTTP request.

That distinction is crucial for agent workflows. A planner may invoke the model repeatedly while decomposing a task, and each attempt may produce a different token profile and latency profile. If you collapse those attempts into one record, you lose the ability to answer whether the cost came from a single difficult turn or from repeated internal reasoning. Keep the inference spans separate, then roll them up under the workflow span only after you have retained the attempt-level data. The workflow span is the right boundary for user-visible features; the inference span is the right boundary for token consumption.

The same logic applies to retries and failover. If the primary model fails and the application falls back to another provider, the fallback span should remain a separate inference attempt with its own model identity and usage. You can still attribute both attempts to the same request or workflow, but you should not erase the distinction between the primary and fallback paths. Otherwise, a failed first attempt disappears from cost analysis, and the remaining spend appears cheaper than the real operational cost.

Propagate business context explicitly. Token usage alone cannot tell you who should pay for a request. You need the tenant, feature, experiment, workflow, or entitlement context that existed when

the inference decision was made. OpenTelemetry baggage and other propagated context give you that bridge. If you set tenant and experiment metadata at ingress and carry it through the request, the inference span can inherit those values and keep them attached to the cost record. That is the cleanest way to connect a token count to a business slice without relying on logs or out-of-band joins.

Use context propagation intentionally. A tenant identifier should be stable, low-cardinality, and present wherever the billing question demands it. An experiment cohort should be present when you need to compare prompt variants or routing policies. A feature flag or workflow name should tell you which user-facing path generated the spend. Chapter 1 covers baggage mechanics; here the important point is that the business metadata must travel with the request long enough to reach the inference span. If asynchronous boundaries drop that context, the later join becomes ambiguous or impossible.

Avoid propagating mutable user-facing state that changes mid-request. You want the business frame that explains the inference decision, not a grab bag of all application fields. Use a small, stable set of dimensions. If you need more detail, keep it in the orchestration span or in structured logs that can be joined later by request or trace identifier. The cost record should be compact enough to query across long time ranges without exploding cardinality.

Join duration and usage, but do not confuse them. Duration and token usage are related, yet they answer different questions. A long span may be network-bound, queued, rate-limited, or waiting on a downstream service before the model call even starts. A high-token span may finish quickly if the model is efficient or if the response is streamed with little delay. If you conflate the two, you will misdiagnose performance and spend.

A better analytical pattern is to retain both fields and compute derived

indicators only after the join. Tokens per second can reveal generation efficiency. Milliseconds per token can reveal throughput limits. Cost per successful request can reveal whether a feature is economically sustainable. Latency inflation associated with longer prompts can reveal prompt growth that users never see directly. Each derived measure depends on the same raw span, but each answers a different operational question.

When you compute these ratios, preserve the underlying distributions. Averages alone can hide the tail behavior that drives user pain and budget overruns. One feature path may have average latency that looks acceptable while still producing a long tail of expensive retries. Another may have moderate token usage but terrible variance because it triggers tool loops or repeated orchestration. Keep duration and usage joined at the attempt level so you can examine medians, percentiles, and outliers separately.

Treat provider-side reconciliation as an authoritative but separate source. Live traces are the starting point for attribution, not always the final billing truth. Provider reporting systems may include categories that do not appear in a single response payload: cache creation, cache reads, service tier adjustments, web search, code execution, or conversation carryover. Realtime and multi-turn APIs can also shift when cost is incurred, because earlier outputs may become later inputs. That means a live span may capture the attempt cleanly while a provider usage report later supplies a more complete accounting.

Use the live span for operational analysis and the provider reconciliation feed for financial finalization. The join key should be stable across both: request identifier, conversation identifier, tenant, API key, model, or another provider-supported correlation field. If you have to choose between immediacy and completeness, keep both records but label them clearly. Provisional cost supports dashboards and alerts; reconciled cost supports chargeback and finance. Mixing the two in

one field creates numbers that are neither timely nor trustworthy.

This separation also helps when provider semantics evolve. A provider may revise how it counts cache read tokens or how it reports multi-choice generation. If your internal cost model depends only on the final span payload, you will miss those changes. If you keep the raw usage data and the reconciled usage data distinct, you can compare them and detect systematic drift. That is how you avoid quietly absorbing billing changes into your application analytics.

Choose the right attribution dimensions. Not every dimension deserves to be part of the cost analysis. The tempting approach is to attach every interesting label to every metric series. That works for a week and then collapses under cardinality. You need a deliberate policy for which dimensions are stable enough to survive aggregation and which should stay in traces or logs.

The usual high-value dimensions are tenant, model, endpoint family, feature, experiment cohort, and service tier. Those dimensions explain most of the variation in spend and are stable enough to support dashboards and chargeback reports. Dimensions such as raw request ID, user ID, prompt hash, or session ID are usually too granular for metrics, although they may still be useful in traces for debugging. If you need to compare prompt versions, use an explicit prompt-version label rather than the literal prompt text. If you need to compare experiments, use a bounded cohort identifier rather than the full feature-flag payload.

The choice depends on the analytical goal. Finance needs stable dimensions for chargeback. Product needs feature and experiment dimensions for adoption analysis. Operations needs model, service tier, and latency-related dimensions for efficiency and saturation. You rarely need all of those on the same aggregate metric series, and you almost never want every one of them as a free-form label. Keep the tracing model richer than the metric model, and derive the smaller metric view

from the richer trace data when possible.

Build the join in layers. A practical cost pipeline has three stages. First, capture telemetry at the inference span with duration, model identity, token usage, and business context attached. Second, derive a normalized analytical record with one row per inference attempt, preserving the request lineage and the stable context fields. Third, join that record with a pricing catalog or rate table that lives outside the telemetry system. The final result is a cost fact table or metric series that product, finance, and operations can query.

That layering gives you flexibility. You can change pricing rules without re-instrumenting the application. You can change attribution policy without touching provider SDK code. You can even keep multiple cost views side by side: a live operational estimate, a reconciled financial total, and a model-comparison benchmark. Each view uses the same trace identity and the same usage fields, but each applies a different business rule after the join.

If you are building this in a warehouse, the relational shape is straightforward: join the inference span table on trace or span identifiers, join business context on propagated tenant or workflow fields, and join pricing data on model and service tier. If you are building it in a metrics backend, emit low-cardinality counters or histograms from the span processor and enrich them with the same stable dimensions. The important part is not the storage engine; it is preserving a reliable mapping from inference attempt to business owner.

Watch the failure modes. The most common mistake is to use a mutable display name as the billing key. Model aliases change, routing layers rename them, and dashboards drift. Use the canonical request model or provider model identifier instead. A second mistake is to drop baggage across async boundaries, which severs the tenant or experiment context from the inference attempt. A third is to attribute

fallback calls to the primary model because the request started there. That hides the real cost of failover and makes primary-model performance look worse than it is.

Another subtle failure is to mix revenue metadata with execution metadata. The cost analysis layer should know what the model did and which business slice consumed it; it should not also try to infer the pricing contract, discount policy, or subscription state unless those are explicitly part of the attribution rule. Keep subscription logic in finance or billing services. Keep execution facts in telemetry. Join them intentionally at analysis time.

Be equally careful with multi-turn conversations. Earlier outputs can become later inputs, which means a per-request view may understate the true cumulative cost of a conversation. If you need conversation-level economics, roll up the per-attempt records using a conversation or thread identifier, but keep the attempt-level rows intact. That lets you answer both questions: what did this request cost, and what did the whole conversation cost across turns?

Use the joined data for operational decisions, not just reporting. Once you have the duration-usage-context join, the analysis becomes actionable. You can compare cost per successful request across features and decide which workflows need prompt reduction. You can identify tenants whose spend is dominated by cache misses and investigate whether their usage pattern or prompt design is responsible. You can compare experiment cohorts and see whether a new routing policy reduced latency at the cost of higher token consumption, or vice versa.

The same data supports fairness decisions. A premium workflow may legitimately use more tokens if it buys better answer quality. A free-tier workflow may need tighter guardrails. A tenant with unusually large prompts may be a candidate for prompt compression, context trimming, or retrieval redesign. The joined cost record gives you evi-

dence for those decisions without forcing you to reverse-engineer them from logs.

That is the real payoff of treating inference telemetry as a joinable economic record. The span carries the causal story, the usage fields carry the consumption story, and the propagated business metadata carries the ownership story. When you keep those three together, cost analysis stops being a retrospective guess and becomes a reproducible property of the model path itself.

3.4. Modeling Streaming Responses, Partial Results, and Late-Bound Usage

A streaming model call looks simple from the user's perspective: tokens appear quickly, the interface stays responsive, and the answer fills in over time. Under the hood, though, the observability problem gets harder, not easier. The first byte may arrive before the final usage totals exist, a client may disconnect before generation finishes, and the provider may report token accounting only in the terminal event or in a later reconciliation path. If you mark the span as complete when the first token arrives, or if you overwrite final usage with a speculative estimate, your telemetry becomes internally consistent but operationally false.

```
with tracer.start_as_current_span("chat gpt-4.1") as span:
    span.set_attribute("gen_ai.operation.name", "chat")
    span.set_attribute("gen_ai.request.model", "gpt-4.1")
    span.add_event("stream.start")

    for chunk in stream:
        if chunk.text:
            span.add_event(
                "stream.chunk",
                {"gen_ai.output.delta_chars": len(chunk.text)}
            )
        if chunk.usage is not None:
            span.set_attribute(
```

```
                "gen_ai.usage.input_tokens",
                chunk.usage.input_tokens,
            )
            span.set_attribute(
                "gen_ai.usage.output_tokens",
                chunk.usage.output_tokens,
            )

    span.set_attribute("gen_ai.response.finish_reason", finish_reason
     )
```

That pattern is deliberately conservative. The span remains open for the full lifecycle of the attempt, chunk arrivals are recorded as events rather than as span mutations, and usage is attached only when the provider has made it authoritative. If the provider sends final usage in the last chunk, you capture it there. If the provider emits a separate terminal callback or reconciliation record, you update the analytical store from that source rather than from a guessed intermediate value. The model call is then represented as a real attempt, not as a partial guess wrapped in a trace.

Keep the span alive until the attempt truly ends. The defining rule for streaming inference is that the span should represent the lifecycle of the attempt, not the lifecycle of the first visible token. That distinction sounds obvious, but it is easy to violate when SDKs expose streaming as a sequence of callbacks or async iterators. If the application forwards chunks to the client as they arrive, the temptation is to close the span when the response body starts flowing. Doing so hides the tail of the generation and undercounts the latency for any work that happens after the first token.

A correct span stays open until the model has terminated the attempt. For a buffered stream, that means the final chunk or event. For a canceled stream, that means the cancellation point, with status that reflects the termination condition. For a timeout, that means the timeout boundary, not the moment the first partial content was emitted. The

span end time should match the actual end of the attempt, because that is the only point where latency, finish reason, and usage can be interpreted together without ambiguity.

This is also where agent systems and orchestration loops matter. A planner may open a streaming inference span, consume partial output, and then decide to issue a tool request or a second model call. Those are separate operations, and the trace should show that separation. Do not reuse the same span across distinct inference attempts just because they happen in the same user-visible turn. Each attempt gets its own lifecycle, its own terminal state, and its own usage record.

Record intermediate events instead of mutating the final story. Chunk arrival is valuable information, but it is not final information. The right model is to emit events for notable milestones, not to overwrite the span with provisional state every time a fragment arrives. Event-based modeling preserves the timeline without pretending that the stream is already complete. You can record the first chunk, a tool-call argument delta, a partial text milestone, or a moderation interruption as individual events on the same inference span.

That event stream becomes especially useful when you debug user experience problems. If the first token arrives quickly but the remaining tokens stall, the span events tell you whether the stall happened before or after the first emission. If a tool-call argument was streamed incrementally and then truncated, the events can show how far the assembly progressed. If the application is buffering chunks before forwarding them, the events reveal whether the latency belongs to the provider, the SDK, or your own relay code. The span remains the durable container; the events explain the sequence inside it.

Use events sparingly and intentionally. A token-by-token event for every single character often adds more noise than value, especially in high-volume systems. Prefer semantic milestones: stream start, first

chunk, tool-call delta, usage finalization, cancellation, and stream end. That keeps the trace readable while still preserving the information needed for forensic analysis and latency decomposition.

Treat usage as late-bound state. Usage accounting is the hardest part of streaming telemetry because it may appear after the content has already been forwarded. OpenAI's streaming patterns can deliver usage in the final chunk when usage reporting is enabled, and that terminal chunk may have empty choices because its purpose is accounting rather than content. Anthropic streaming can surface usage earlier in the event sequence, before all content blocks have finished arriving. Either way, the lesson is the same: the moment you see the first token is not the moment you know the final token totals.

The instrumentation rule follows directly. Do not finalize usage attributes until the provider has delivered authoritative totals. If the provider exposes usage in the terminal event, set the attributes there before ending the span. If the provider only supplies provisional counts during the stream and a more complete report later, mark the early values as estimates and overwrite them with the reconciled totals in your analytical store. Never mix the two without labeling them, because dashboard consumers will assume that the number shown is final.

This distinction matters for more than billing. Late-bound usage also affects efficiency analysis, cache accounting, and comparison across models. A model that appears cheap during the first part of a stream may turn out to be expensive if it generates a long tail of output tokens. A conversation that appears stable at the request level may accumulate higher cumulative input because earlier outputs re-enter the next turn. The usage record must therefore reflect the full attempt, not the first available snapshot.

Preserve provisional and authoritative values separately. If you must expose near-real-time token counts before the provider has

finalized them, keep the provisional and authoritative paths separate. Use distinct fields or distinct metric series. A provisional estimate is useful for dashboards that need immediate feedback, but it should never silently replace final usage. The simplest discipline is to attach authoritative counts to the span only when they are known, and to emit provisional metrics under a clearly labeled series such as `estimated` or `inflight`.

That separation gives you a clean reconciliation story. The operational dashboard can display estimated usage during the stream, while the nightly aggregation job can replace those estimates with final values from the provider's terminal payload or usage API. If the estimate diverges from the final total, you can measure the drift and decide whether the estimation strategy is good enough. If you collapse both values into one field, you lose the ability to detect and correct bias.

You should also preserve the source of the value. A usage record derived from the terminal chunk is not the same as one derived from a separate usage endpoint or billing export. The first is close to the request path and useful for fast analysis; the second is closer to finance truth and useful for reconciliation. Keeping source and timing attached to the record makes later analysis much more reliable.

Handle cancellation and partial success honestly. Streaming introduces a common failure mode: the user sees something useful, but the generation did not complete. A client may disconnect, a proxy may time out, the server may cancel due to budget enforcement, or the provider may abort due to a moderation or transport error. If you mark those attempts as successful because a partial answer was delivered, you hide the fact that the model never reached a clean terminal state.

The span status should tell the truth about the attempt. If the generation was canceled, record cancellation or error status, even if the user received partial content. If the provider timed out after emitting

chunks, mark the span accordingly and preserve the partial output as a traced event or stored artifact if your policy allows it. If the application stopped the stream because a budget guardrail fired, record that policy decision explicitly. The trace should show that the response was partial, not successful, unless the provider itself reported a clean finish.

That honesty is important for cost analysis as well. Partial success still consumed tokens, and those tokens still belong in the usage record. If a canceled generation produced 400 input tokens and 120 output tokens before aborting, the cost analysis should count those tokens. What changes is the outcome label, not the accounting. Distinguishing partial completion from full success lets you study whether expensive cancellations correlate with particular tenants, prompts, or model paths.

Model the stream as a sequence of states. A streaming inference attempt moves through a small number of semantically meaningful states: started, emitting, finalized, canceled, or failed. You do not need to represent every transport-level packet; you do need to represent state transitions that change the meaning of the attempt. The state model helps you decide which attributes are safe to set at which point.

At stream start, the span should already contain the invocation metadata: model, provider, request mode, and business context. During emission, you can add events for chunk arrival or partial tool-call assembly. At finalization, you set authoritative usage and finish reason. At cancellation or failure, you set span status and capture the exception or cancellation cause. That sequence ensures that later readers can reconstruct what was known when, rather than mistaking provisional state for final truth.

This state model also simplifies downstream joins. A warehouse job that aggregates only finalized spans can exclude canceled attempts from success-rate metrics while still including their usage in spend to-

tals. A real-time dashboard can show inflight spans separately from finalized spans. An incident reviewer can inspect only the failed or canceled spans without wading through completed requests. The trace is richer because it preserves state transitions, not because it stores every low-level detail.

Keep partial results useful without overexposing content. A streaming trace often contains enough information to be useful without storing the full response text. A chunk arrival event can record timing, length, and structural metadata without preserving raw content. A tool-call delta event can record that arguments were extended, without storing the entire argument body. This is an important boundary when tracing must coexist with privacy policy and storage limits.

The practical implication is that you should separate content capture from lifecycle capture. You can faithfully represent the stream's progress with a small set of events and final attributes even if you do not retain the payload itself. That keeps the observability data useful for latency, cancellation, and usage analysis while reducing the risk that every partial token becomes a long-lived data artifact. Chapter 7 covers content capture policy in depth; here the key point is that stream telemetry does not require raw content to remain analytically valuable.

If you do store partial results, make the retention policy explicit. Partial outputs may be helpful during debugging, but they also carry the same privacy and security concerns as full outputs. The trace should make it possible to understand the lifecycle either way. What changes is the payload retention decision, not the definition of the inference attempt.

Separate provider truth from local approximation. Different providers expose streaming usage at different times. Some place it in the terminal event. Some surface it earlier. Some expose a later reconciliation path. Your instrumentation should accept that variability rather than forcing every provider into the same immediate shape. The

local trace should capture the attempt as observed in real time, then your analytics pipeline should reconcile it to the most authoritative source available.

That means you often need two clocks, so to speak: the request-time trace and the post-hoc accounting record. The request-time trace tells you how the user experience unfolded and how long the attempt lasted. The accounting record tells you what the provider ultimately charged or reported. If you conflate those into one record too early, you lose both the operational truth and the financial truth. If you keep them linked by trace and request identifiers, you get both.

The distinction matters when provider behavior changes across SDK versions. A streaming client that used to expose usage in the last chunk may later move that data into a separate callback or a finalized response object. Your schema should tolerate that evolution by treating usage as late-bound state attached to the same attempt, not as a property of a particular callback. That keeps the analytical model stable even when the SDK surface changes.

Finalize spans on the true terminal condition. The final rule is operationally simple and easy to violate: end the span when the attempt ends, not when the first answer fragment appears. If the generation completes cleanly, record the final usage and finish reason, then close the span. If the attempt is canceled, timed out, or failed, record that termination state and close the span at the actual terminal event. If usage arrives late, update the authoritative record before closing or reconcile it immediately after with a linked accounting record.

That discipline is what makes streaming telemetry trustworthy. It lets you compute latency, usage, and success rate from the same attempt without inventing data or hiding failure. It also preserves the analytical continuity with the earlier span model: one inference attempt, one parent request or workflow, one set of usage values, one terminal outcome.

When the lifecycle outlasts the first token, the span must outlast it too.

Chapter 4

Observing Agentic Workflows and Tool Execution

Agentic systems fail in more interesting ways than plain prompt-response applications: the model may be fast while the agent is slow, the answer may look successful while hidden retries exhausted budgets, and a tool may return stale or partial data without ever surfacing as an explicit error. This chapter teaches readers how to represent those realities in OpenTelemetry so traces remain analytically useful under loops, fan-out, retries, and mixed ownership boundaries.

4.1. Modeling Execute Tool Spans and Agent Causal Chains

A production agent often looks simple from the outside: a user asks a question, the model responds, and somewhere in the middle a tool fetches data or performs work. The trace is usually less simple. A single answer may involve an orchestration span, one or more model inference spans, one or more tool requests, downstream service calls, and occasional async continuations that finish long after the original request thread has moved on. If you model that path as a flat list of spans, you lose the causal structure that explains why the agent behaved correctly, why it slowed down, or why it appeared to "change its mind."

The span model you want is not just a convention for naming spans. It is a causal map. The planner decides, the model proposes, the runtime executes, and the downstream services complete work. Each of those layers has a distinct observability job, and conflating them produces traces that are technically valid but analytically useless.

A practical starting point is to anchor the agent run with a top-level workflow span, then place inference and tool execution where they belong. With current OpenTelemetry GenAI conventions, the tool boundary is represented with an `execute_tool` operation, while a broader coordinated run can use an `invoke_workflow` boundary when the agent is acting as an orchestrator over multiple steps or sub-operations. Because those conventions are still in development, isolate the span names and attributes behind an internal wrapper so you can translate field names later without rewriting every call site. That gives you a stable application API even if the semantic layer evolves. For a direct implementation, the orchestration boundary can look like this:

```
with tracer.start_as_current_span(
    "invoke_workflow",
    kind=SpanKind.INTERNAL,
```

```
) as wf:
    wf.set_attribute("gen_ai.operation.name", "invoke_workflow")
    wf.set_attribute("gen_ai.request.model", "planner-v2")

    with tracer.start_as_current_span(
        "execute_tool search_customer",
        kind=SpanKind.CLIENT,
    ) as tool:
        tool.set_attribute("gen_ai.operation.name", "execute_tool")
        tool.set_attribute("gen_ai.tool.name", "search_customer")
        tool.set_attribute("workflow.step.id", "step-3")
        result = search_customer(query)
```

The code is short, but the structure matters. The workflow span marks the agent boundary. The tool span marks the concrete unit of external or semantically significant work. The downstream call, if instrumented, becomes a child of the tool span rather than a peer of the model span or an anonymous internal event. That hierarchy preserves the difference between deciding to do work and doing it.

Four execution layers

The clearest way to reason about agent traces is to separate four layers that are often collapsed in logs and dashboards:

- the user request boundary,
- the planner or orchestrator step,
- the model inference that emits a tool decision,
- the actual tool execution work.

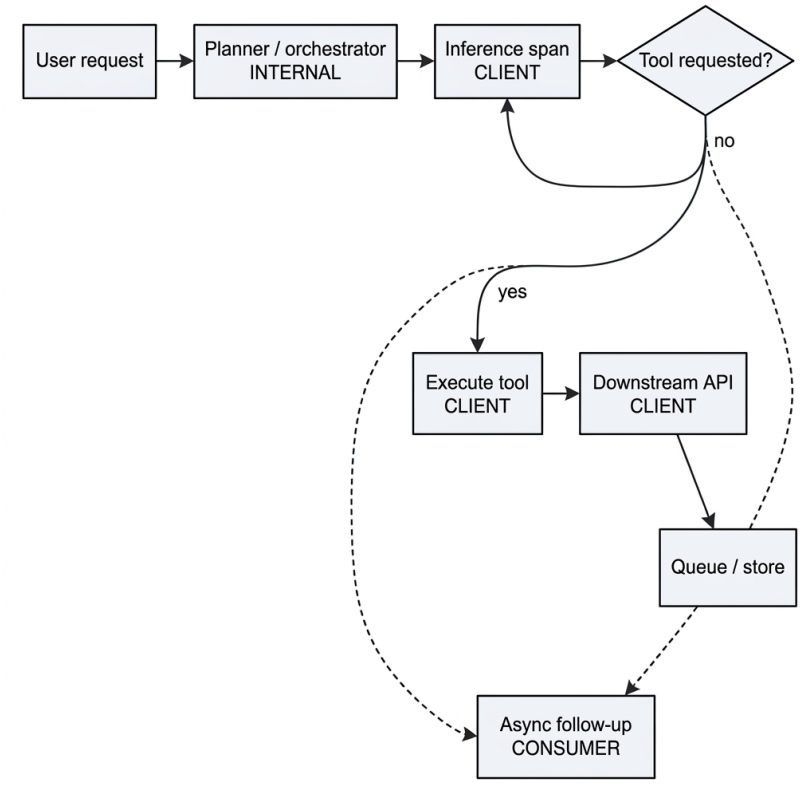
Each layer has a different responsibility. The user request boundary is the externally visible transaction. The orchestrator decides which internal path to take. The model produces a candidate action, often as a tool request or a response plan. The tool execution layer performs concrete work against an API, database, queue, filesystem, or internal

service. If you keep those layers distinct, you can answer questions like: Was the delay in planning, inference, marshaling, or the tool itself? Did the model ask for a tool but the orchestrator reject or defer it? Did the tool succeed but the final synthesis fail?

This layering also clarifies span kinds. The orchestrator is usually `INTERNAL` because it represents local decision logic. A remote model call is typically `CLIENT`. A tool that calls an external dependency can itself be a `CLIENT` span if it owns the remote boundary, or `INTERNAL` if it is only a local wrapper around already-instrumented calls. The key is consistency: the same conceptual boundary should not be modeled as a client span in one service and a random internal span in another.

A layered trace model

The causal structure becomes easier to see when you picture the flow explicitly. The figure below shows a user request entering an orchestrator, the orchestrator invoking inference, the inference producing a tool request, and the tool execution continuing into downstream work. The async branch is linked rather than nested to show that the causal relationship exists even when the timing does not fit a strict parent-child chain.



The diagram intentionally avoids pretending that all causality is a tree. The solid arrows represent nested execution that can be modeled as parent-child spans. The dashed arrow represents a continuation that is caused by the earlier decision but cannot be faithfully represented as a synchronous child because the work resumes later. That distinction is critical in agent systems, where queueing, retries, deferred callbacks, and workflow resumes are normal rather than exceptional.

Tool spans are semantic boundaries, not mere wrappers

An `execute_tool` span should represent the logical act of executing a

tool, not just the existence of a helper function. That distinction prevents a common anti-pattern: instrumenting every local function as if it were a tool and then drowning the trace in noise. If the function merely validates JSON, converts units, or formats a prompt fragment, keep it as an event or a small internal span only when it helps explain latency or control flow. Reserve tool spans for externally visible operations, expensive transformations with user impact, or application-owned actions that matter to incident analysis.

A useful test is whether you would want the span even if the implementation changed. If the operation has a stable business identity—for example, entitlement lookup, inventory search, pricing quote generation, or policy evaluation—it deserves its own span. If it is an incidental helper that exists only because the code is structured that way, it probably does not.

That naming discipline matters because later analysis depends on stable queries. You want to be able to ask, "How often did `execute_tool search_customer` exceed 500 ms?" or "Which traces included a fallback from `execute_tool retrieve_docs` to `execute_tool web_search`?" If every team invents its own naming style, the trace backend becomes impossible to aggregate.

Causality is not the same as nesting

Parent-child span structure works well when work is synchronous and locally owned. It breaks down when the execution path crosses an async boundary, fans out across multiple workers, or resumes in a different trace context. In those cases, a span link is often the right representation. A link says, "this span was caused by that context," without pretending the earlier span was still the active parent at the moment this work ran.

This matters for at least three agent patterns.

First, a planner may issue a tool call that is queued and executed later by a worker. The worker span should often be a new root with a link to the planner or tool-request span. If you force it to be a child, you misstate timing and risk breaking the trace when context is no longer live.

Second, a fan-out retrieval step may dispatch several candidate lookups in parallel. Each retrieval span can be a child of the orchestrator while the final synthesis step links to all of them, because the synthesis is causally dependent on multiple results rather than any single linear parent.

Third, a deferred continuation such as human approval, webhook callback, or scheduled retry should preserve the original context through links. That keeps the causal chain visible without inventing an impossible synchronous ancestor.

The rule of thumb is simple: use parent-child when the parent actively contains the work in time; use links when the work is merely caused by the parent or depends on multiple contexts. Links should be attached at span creation whenever possible, because samplers can consider creation-time links, while links added later may not influence the sampling decision.

How the model, planner, and tool interact

A good agent trace separates what the model proposed from what the runtime actually did. The model may produce a tool call suggestion, but the orchestrator can still reject it, normalize it, enrich it, or route it through a policy gate. Do not attach tool metadata only to the model span, because that erases the runtime decision that turned a suggestion into an actual execution. The model span should capture inference behavior; the tool span should capture execution behavior.

This separation becomes especially valuable when the model is not the

bottleneck. A trace might show a short inference span, a long tool span, and an even longer orchestration span that handles validation, retries, policy checks, and aggregation. Without the explicit boundary between inference and execution, all of that collapses into an opaque "model latency" story that leads engineers in the wrong direction.

The trace should also record the decision path at the orchestration layer. Useful attributes are the selected tool name, the planner step identifier, the workflow step number, the selection confidence or policy reason if that value is available, and any routing outcome such as "allowed," "suppressed," or "deferred." Keep those attributes compact. They are there to explain control flow, not to store the full prompt or the full tool payload. That material belongs to token accounting, content capture policy, or an externalized blob store, depending on your telemetry design.

A single tool call is the baseline

The simplest case is a planner that emits one tool request and then synthesizes the answer from the result. In that path, the trace should form a clean chain:

workflow → orchestrator → inference → execute_tool → downstream client spans.

That chain is the baseline because it gives you a directly readable story. The workflow span measures the user-visible transaction. The inference span measures the model decision. The tool span measures the logical tool operation. The downstream spans measure the concrete dependencies inside the tool.

If the tool is itself implemented on top of auto-instrumented HTTP, database, or RPC clients, let those spans remain children of the tool span. That makes it easy to compute inclusive versus exclusive time. The inclusive duration of the tool span tells you how long the tool took

end to end. The child spans reveal whether the time went into network calls, database work, serialization, or CPU-bound post-processing. The same logic applies to the orchestrator span: the difference between its wall-clock duration and the durations of its children gives you the orchestration overhead that is otherwise invisible.

Iterative loops need stable identities

Real agents rarely make one decision and stop. A ReAct-style loop may alternate between inference and tool execution several times before returning a final answer. In that case, resist the temptation to put every turn into a single long span. A monolithic span hides the boundary between attempts and removes the possibility of attributing latency to individual loop iterations.

Instead, model each iteration with its own inference span and, when appropriate, its own `execute_tool` span. Use a stable workflow identifier and a step identifier so the sequence can be reconstructed later. That gives you a trace shape like:

$$\text{workflow} \rightarrow \text{plan}_1 \rightarrow \text{tool}_1 \rightarrow \text{plan}_2 \rightarrow \text{tool}_2 \rightarrow \cdots$$

The step identifiers do not have to be user-visible. They only need to be stable enough for diagnostics. A workflow that loops three times before success is not the same operationally as a workflow that loops once. If you collapse those cases into a generic "agent span," you lose the ability to ask why the agent became expensive or slow.

At the same time, do not explode the trace into dozens of trivial helper spans. The right granularity is the one that answers operational questions without producing query fatigue. For a typical agent, planner turns, model invocations, tool executions, and long-lived state transitions are the natural units. Tiny helper methods are not.

Tool-to-tool chains and mixed ownership

A tool can call other tools or internal services. For example, a business tool may query a search service, then fetch a record from a database, then call a policy engine. The instrumentation question is not whether those child calls should exist—they should—but where to draw the ownership line.

If the application-owned tool is the semantic boundary that the user cares about, wrap the tool logic in an `execute_tool` span and let the downstream client spans hang beneath it. That way the tool span represents the unit of intent, while the child spans represent implementation detail. If, by contrast, the tool is merely a thin wrapper around a remote service with its own strong tracing, let the remote service own the business boundary and make the wrapper span lighter.

Mixed ownership is where over-instrumentation and under-instrumentation both cause trouble. Over-instrumentation duplicates the same dependency work at several layers. Under-instrumentation leaves a gap in the trace just where the business logic actually happens. The right balance is to instrument the logical tool boundary once, then allow existing HTTP/DB/RPC instrumentation to explain the internals.

Fan-out needs links, not fake serial chains

Fan-out is a common agent tactic when the system evaluates several retrieval sources, candidate tools, or hypotheses before synthesizing a result. The causal reality is that one planner step causes several parallel branches. The incorrect trace shape is a serial chain that makes those branches appear to depend on each other in time. That shape inflates latency attribution and hides concurrency.

Represent each parallel branch as its own child span when the work is truly concurrent under the same live context. Then link the final synthesis span to the branch spans if the synthesis depends on multiple results. The final synthesis span should not pretend to be the child of

only one branch, because that would misrepresent the causal graph. If the branches cross process boundaries or are replayed from a queue, use links more aggressively so the trace preserves the dependency even when a tree would be false.

This is one of the reasons span links exist. They let you model causality beyond the simple tree without resorting to custom metadata hacks. For agent systems, that is not an edge case. It is the normal shape of real work.

What to preserve, and what to leave out

A useful trace preserves stable parentage, semantic names, a step identity, and enough compact attributes to reconstruct the decision path. It does not need the full prompt, full tool payload, or every local helper method. Those belong elsewhere: token usage and inference accounting belong with model telemetry; sensitive content capture belongs behind explicit policy; low-level implementation details belong in logs or debug traces when truly necessary.

That division of responsibility protects both readability and query performance. A trace backend that receives a span for every helper function quickly becomes hard to search and expensive to store. A trace backend that only sees one giant agent span cannot answer any meaningful operational question. The model here sits between those extremes: one span per semantically meaningful boundary, with links where causality is real but synchronous nesting is not.

Anti-patterns that break the causal story

Several mistakes show up repeatedly in agent telemetry:

- Sibling spans with no causal linkage. If the model, tool, and downstream work appear as unrelated peers, the trace cannot explain how the request unfolded.

- Every tool operation modeled as a generic internal span. That destroys the distinction between orchestration and externally meaningful execution.
- Tool metadata attached only to the model span. That hides the actual runtime boundary and makes later latency analysis misleading.
- Asynchronous work forced into a synchronous child chain. That misstates timing and often breaks trace continuity.
- Excessive micro-spans for trivial helpers. That makes the trace noisy without improving diagnosis.

Each of these mistakes has the same root cause: the span tree was chosen for convenience rather than for causality. Agent observability is better when the trace reflects the execution semantics of the system, not the incidental structure of the code.

The practical goal is not to make every trace look elegant. It is to make the important questions answerable. Which tool did the planner request? Which request actually ran? Was the work synchronous or deferred? Did one inference lead to several parallel retrieval paths? Did the runtime choose to execute a tool, or did it merely consider doing so? Once the trace answers those questions cleanly, the rest of the observability stack can build on top of it without guessing.

4.2. Instrumenting Application-Owned Tools and Workflow Boundaries

A model SDK often gives you excellent visibility into inference, yet the work that matters to the business usually happens one layer away: entitlement checks, quote generation, policy evaluation, internal search

adapters, spreadsheet parsing, approval flows, and other application-owned tools. If you do not instrument those boundaries yourself, the trace tells you that the agent asked for a tool and then went quiet. You cannot see whether the delay came from domain logic, an internal API, queue wait, serialization, or a hidden retry. You can only see that something happened somewhere.

That gap is exactly where manual instrumentation pays for itself. The goal is not to create more spans indiscriminately. The goal is to mark the semantic boundary that the user cares about: the operation that transforms an agent decision into real work. When you do that well, the trace becomes a readable operational graph instead of a set of disconnected library events.

A practical manual pattern looks like this:

```
with tracer.start_as_current_span(
    "invoke_workflow",
    kind=SpanKind.INTERNAL,
) as wf:
    wf.set_attribute("gen_ai.operation.name", "invoke_workflow")
    wf.set_attribute("workflow.id", workflow_id)
    wf.set_attribute("tenant.id", tenant_id)

    with tracer.start_as_current_span(
        "execute_tool quote_order",
        kind=SpanKind.CLIENT,
    ) as tool:
        tool.set_attribute("gen_ai.operation.name", "execute_tool")
        tool.set_attribute("gen_ai.tool.name", "quote_order")
        tool.set_attribute("workflow.step.id", "step-2")
        tool.set_attribute("tool.input.shape", "order+customer")
        quote = quote_order(order, customer)
```

That pattern is intentionally small. It demonstrates the important part: the workflow span captures orchestration, the tool span captures the logical tool boundary, and the downstream clients inside `quote_order` can inherit the active context automatically if they are already instrumented. Once that structure exists, the rest of the trace can explain itself.

Decide what deserves its own span

The first decision is boundary selection. A span should represent a unit of work that matters operationally or semantically, not every function call. In application-owned tooling, the strongest candidates are externally visible tool invocations, expensive deterministic transformations, multi-step business workflows, and handoffs to internal services. Those are the places where latency, failures, and retries affect user experience.

Keep trivial validation, in-process mapping, and short helper logic inside the surrounding span as events or attributes. If you split every helper into a separate span, the trace becomes expensive to store and hard to query. More important, the hierarchy becomes misleading because the reader starts to interpret implementation detail as business structure. A tiny JSON field normalization helper is not a first-class tool. An entitlement service that can block a purchase is.

A useful test is ownership. If the operation has a stable business identity and an operator would reasonably ask, "How long did that step take, and did it fail?", it deserves its own span. If the operation only exists because the code is organized a certain way, it usually does not.

Use a stable naming contract

Span names matter more than teams often expect. If one team emits `searchCustomer`, another emits `search_customer`, and a third emits `customer-search`, your backend now needs normalization logic before it can answer a simple question. Pick one naming convention and keep it stable across versions.

For application-owned tools, prefer a span name that reflects the operation class and the logical tool name. The common pattern is `execute_tool {tool.name}` for the tool boundary and `invoke_workflow` or `invoke_agent` for orchestration. That keeps the

name readable while preserving enough structure for aggregation. Put the more detailed identifiers in attributes. This separation gives you stable queries without forcing the name field to carry every detail.

That distinction is especially useful when a tool evolves. If you later split `quote_order` into precheck, pricing, and approval substeps, the top-level tool span can remain `execute_tool quote_order` while the children explain the internal decomposition. Dashboards and alerts stay intact because the boundary remained stable.

Manual spans are a context problem first

The most common instrumentation failure is not span creation. It is context loss. You start a workflow span in the orchestrator, dispatch work to a helper, and the helper creates an unrelated span because the active context was not propagated. The result looks like the code ran, but the trace says the tool appeared out of nowhere.

You prevent that by extracting the current context at the orchestration boundary and passing it explicitly into any function, worker, callback, or queue payload that will continue the operation. In synchronous code, the current context is usually enough. In asynchronous code, thread pools, task queues, or callback-based frameworks, you must carry it across the boundary yourself.

The same rule applies to baggage. Baggage is useful for small cross-process metadata such as a workflow identifier, tenant identifier, or experiment bucket. Keep it compact and stable. Do not use baggage as a dump for prompt text, tool payloads, or large state blobs. Baggage propagates widely and is intentionally immutable in the context container, which makes it a good fit for correlation data and a poor fit for operational payloads.

A simple guideline helps: if the value is needed to join traces, keep it small enough to travel everywhere. If the value is needed to debug the

business logic, keep it in the span attributes or external storage under an explicit policy.

Attach safe, high-value attributes

A custom tool span should tell you what the tool was, what kind of work it did, and enough about the invocation shape to be queryable later. It should not duplicate the full prompt or raw input by default. Useful attributes typically include the tool name, workflow step id, tenant id, routing outcome, input shape, data source class, and a coarse result class such as `success`, `empty`, or `policy_blocked`.

The reason to favor shape metadata over raw payloads is twofold. First, shape is usually what you need for aggregation. Second, raw payloads often contain sensitive or high-cardinality fields that make telemetry expensive and risky. A span attribute such as `tool.input.shape=order+customer` helps you answer operational questions without exposing the order body.

If you need the exact input for debugging, store it outside the span and emit a reference identifier. That keeps the trace lean and lets you apply a different access policy to the payload store. The same approach works for large tool outputs. Put the summary on the span and externalize the full content if you truly need to keep it.

Instrument the workflow edge, not only the inner dependency

When you own the tool code, it is tempting to rely on downstream auto-instrumentation and call the job done. That approach misses the logical boundary the user actually cares about. Suppose a tool calls an internal HTTP API, a database, and a rules engine. Auto-instrumentation will show the HTTP and database spans, but it will not tell you that these calls were collectively part of `policy_evaluation` or `quote_order`.

The tool span is the semantic wrapper around those internals. It explains why the work happened and groups the child calls into a business-visible unit. The downstream spans still matter because they decompose the cost. Without the wrapper, you can measure dependency latency but not tool latency. With it, you get both.

This is where the boundary model becomes practical:

workflow span → orchestration INTERNAL spans → `execute_tool` → auto-instru

That order is the one you want to preserve. It keeps orchestration visible, tool execution visible, and dependency work visible without mixing their responsibilities.

Record lifecycle milestones as events

Not every significant moment deserves a new span. Many tool operations have a few milestones that are useful to see but too small to justify additional spans. In those cases, use span events. For example, you might record `tool.request.built`, `tool.call.sent`, `tool.response.received`, and `tool.result.validated`. Those events give you a coarse timeline inside the span without fragmenting it.

Events are especially useful when a tool has an internal serialization or validation phase that can dominate latency on large inputs. If the tool spends most of its time parsing a spreadsheet or normalizing a configuration payload, the events let you separate parsing, remote I/O, and business logic without turning each step into a parent of the next. That keeps the trace readable while still making the bottleneck visible.

Keep the event names stable and low cardinality. A good event name behaves like a protocol marker. A bad one behaves like a log line with timestamps removed.

Propagate context into nested clients

If the application-owned tool invokes HTTP, RPC, database, or queue clients that already support OpenTelemetry, let them inherit the active context so their spans become children of the tool span. That produces the correct hierarchy for downstream timing decomposition. The tool span shows the logical operation; the child client spans show the implementation details.

This pattern is easiest when the tool runs synchronously in the same process. In that case, the active span context is already current when the nested client executes. In asynchronous code, you need to make the propagation explicit. Carry the context into the worker, then start the client calls under that context inside the worker process or coroutine.

If the downstream client library is not instrumented, your tool span becomes even more important. It is the last reliable measurement boundary you have. You can still attach events and coarse attributes to it, but you lose the lower-level decomposition until the client is instrumented or wrapped.

Handle async work without pretending it is synchronous

Many agents schedule work outside the original request thread. A human approval may be queued, a batch lookup may run in a worker pool, or a callback may arrive minutes later. You should not force those paths into one synchronous span tree.

Use a child span only when the parent actively contains the work in time. Use a link when the later work is causally related but runs after a break in execution. If the continuation is a worker that receives a queued message, propagate the trace context in the payload when feasible and add the original context as a link on the resumed span. If the resumed work represents a fresh execution lifetime, make it a new root with a link rather than a fake child.

This distinction matters for both correctness and analysis. A fake child span under the original request suggests the user waited on work that actually happened later. A linked continuation tells the truth: the workflow caused the work, but the work was not synchronous with the parent span.

A small example helps. Suppose `quote_order` submits a pricing request to a queue, then a worker processes it and writes the result back. The request span should end when the queue handoff finishes. The worker span should start when the message is consumed, with a link to the enqueuing context. The queue wait is then visible as its own operational gap instead of being hidden inside a misleading parent-child chain.

Keep workflow identifiers separate from business correlation identifiers

A workflow step id, a deduplication key, a tenant id, and a customer id all serve different purposes. Do not collapse them into one generic correlation field. The workflow id helps you reconstruct execution order. The business id ties the action back to a user or account. The deduplication key identifies idempotent retries. The tenant id helps with partitioning, filtering, and access control.

If you overload these identifiers, later analysis becomes ambiguous. An operator asking whether a tool retried will not want to infer retry behavior from a customer id. A security reviewer looking for cross-tenant leakage will not want to parse workflow ids to find partition boundaries.

Use the span attributes for the identifiers that matter to the operational query, and keep them semantically distinct. That discipline also reduces cardinality problems because each identifier has a known purpose and a known lifecycle.

Choose instrumentation depth by operational value

The right amount of manual instrumentation depends on ownership, latency, criticality, and existing coverage. A high-latency tool that gates revenue deserves richer telemetry than a short helper that formats a string. A tool that already sits under complete downstream auto-instrumentation may need only a wrapper span and a handful of attributes. A tool that performs business logic with no downstream calls may need more explicit lifecycle events because there is no child span structure to explain it.

There is a cost to every added span. More spans mean more storage, more export traffic, more sampling pressure, and more query complexity. If you instrument every helper, you spend observability budget on noise. If you instrument only the outer request, you spend it on ignorance. The best outcome is usually a sparse set of semantically rich spans that bracket the work the business actually cares about.

A rough heuristic helps:

- instrument the boundary if the operation can fail independently,
- instrument the boundary if the operation can dominate latency,
- instrument the boundary if the operation crosses ownership or process limits,
- keep it inside the parent span if the work is trivial and local.

Avoid the common failure modes

Several anti-patterns show up again and again in application-owned telemetry:

- Wrapping every helper function with a span. The trace becomes noisy and expensive.

- Recording raw tool arguments by default. That creates privacy and compliance risk.

- Losing context at thread-pool or queue boundaries. The trace fragments into orphan spans.

- Giving every team its own span naming style. Queries become brittle and dashboards drift.

- Relying only on downstream client spans. You lose the logical business boundary.

Each of these errors is fixable once you decide that the tool boundary is a first-class operation. That decision changes how you write the code. You create the boundary span before the tool work begins, propagate the context explicitly where needed, and close the span on both success and failure paths. If the tool has asynchronous branches, you add links rather than pretending the tree is linear.

Keep source telemetry clean before enrichment

Application-owned spans should be trustworthy at the source. Downstream enrichment can add derived attributes, metadata, or aggregation labels, but it cannot repair a missing causal boundary cleanly after the fact. If the tool span is absent, the collector cannot invent the business operation. If the context was dropped before the queue handoff, the collector cannot reconstruct the true parentage with certainty.

That is why this instrumentation work belongs at the application edge. You want the trace to describe the actual execution semantics before any pipeline logic touches it. Once the source telemetry is correct, later chapters can add collector-side enrichment, privacy controls, and policy-aware filtering without guessing at the original structure.

4.3. Decomposing Agent Latency Across Model Time, Tool Time, and Orchestration Overhead

A slow agent is rarely slow for one reason. A model may generate a tool request quickly, then a search service stalls, then an orchestration loop waits on validation, then a deterministic formatter burns time stitching results together. If you measure only the end-to-end request, you know the user waited, but you do not know what to fix. The wrong diagnosis is expensive: teams tune prompts when they should tune caches, or they optimize retrieval when the real problem is queue wait.

The useful unit of analysis is the latency bucket. Model time belongs to inference spans such as chat, generate, or completion calls. Tool time belongs to the `execute_tool` boundary and its downstream children. Orchestration overhead belongs to the internal work that binds those steps together: planning, policy checks, marshaling, retries, waiting, aggregation, and response assembly. Once those buckets are explicit in the trace, you can reason about bottlenecks instead of guessing.

A direct instrumentation pattern makes the decomposition visible early. The workflow span marks the request lifetime, the model span captures inference, the tool span wraps logical tool execution, and nested client spans or events expose the internal cost structure.

```
with tracer.start_as_current_span(
    "invoke_workflow",
    kind=SpanKind.INTERNAL,
) as wf:
    wf.set_attribute("workflow.id", workflow_id)

    with tracer.start_as_current_span(
        "chat.completion",
        kind=SpanKind.CLIENT,
    ) as model:
        model.set_attribute("gen_ai.operation.name",
                            "generate")
        response = llm.generate(messages)
```

```
    with tracer.start_as_current_span(
        "execute_tool search_docs",
        kind=SpanKind.CLIENT,
    ) as tool:
        tool.set_attribute("gen_ai.operation.name",
                           "execute_tool")
        tool.set_attribute("gen_ai.tool.name", "search_docs")
        docs = search_docs(query)
```

That sketch is not the full answer, but it gives you the structure required for the full answer. The model span tells you how long the inference took. The tool span tells you how long the logical tool step took. The workflow span gives you the total wall-clock duration of the request. Everything else is decomposition.

Separate wall-clock latency from work latency

The first analytical mistake is to treat one total duration as if it described all the work. It does not. The workflow span's inclusive duration gives you the user-visible latency: from request arrival to final response emission. That is the number that matters for the SLO. The child spans let you partition that wall clock into pieces that are meaningful for diagnosis.

The distinction between inclusive and exclusive time is the core of latency attribution. The inclusive duration of a workflow span includes all nested model calls, tool executions, and orchestration logic. The exclusive duration of an orchestration span excludes its children and therefore exposes the overhead that is not spent in model inference or downstream dependencies. If a planner span is long but its children are short, the problem is likely the planner itself, not the model or the tool.

This becomes especially important in agents because orchestration logic is often invisible in conventional service graphs. You may see a long endpoint latency and a few quick downstream calls, then con-

clude the upstream model is slow. The trace may instead reveal that the model finished in under a second and the remaining time went into reranking, validation, schema conversion, or response stitching. The model was not the bottleneck; the runtime was.

Use a latency taxonomy that maps to span structure

A useful taxonomy has six buckets:

- model inference duration,
- planner or orchestration overhead,
- tool selection and marshaling cost,
- downstream dependency latency inside tools,
- workflow idle or queue wait time,
- deterministic post-processing before response emission.

Each bucket maps naturally to a span type or timing boundary. Model inference duration belongs in the model span. Tool selection and marshaling usually belong in the orchestration span or a short tool-preparation child span. Downstream dependency latency appears in the tool span's children, such as HTTP, database, or RPC spans. Queue wait may not belong to any active CPU span at all; it often appears as a gap between a producer span and a later consumer span linked by context. Deterministic post-processing belongs in internal spans or events around formatting, ranking, filtering, or synthesis.

The taxonomy matters because it prevents double counting. If a tool span wraps a database query and an HTTP call, and those child spans are instrumented separately, you should not add all durations together and treat the result as elapsed wall-clock time. Parallel work overlaps. A fan-out can have cumulative child duration greater than the actual

request latency. The relevant value for user experience is the critical path, not the sum of all branches.

Critical path and cumulative work are different quantities

Fan-out is where naive latency calculations fail most often. Suppose an agent fans out to three retrieval sources in parallel, then waits for the slowest result before synthesizing the answer. The wall-clock contribution of that retrieval phase is the duration of the slowest branch, not the sum of all three branches. The cumulative work, however, is the sum of all three because all three services consumed time and resources.

You need both views.

The critical path tells you what the user felt. The cumulative work tells you what the system paid. If you want to reduce p95 latency, you optimize the critical path. If you want to reduce cost or capacity pressure, you may optimize cumulative work. An agent can have a fast critical path and still be operationally expensive because it burns compute in many parallel branches or repeated loops.

That distinction is why child durations can exceed wall-clock duration in concurrent systems. Overlapping work is real work, but it is not serial latency. If the trace backend or dashboard sums every child span as if they were serial, the analysis is wrong. You need to interpret the tree as a partial order, not a linear timeline.

Decompose one-call, one-tool requests first

The simplest useful case is a single model call followed by one tool call. This pattern gives you a baseline decomposition with three regions: model time, tool time, and orchestration overhead around them. In a healthy implementation, the model span is usually easy to inspect because the library generates it. The tool span is the one you add manually when the tool is application-owned. The orchestration span sits above both and captures policy, routing, and response assembly.

For this pattern, query the workflow duration, the model duration, and the tool duration separately. If the workflow is 12 seconds, the model is 800 ms, and the tool is 9 seconds, the model is not the problem. If the tool child spans show 8.5 seconds of database wait and 100 ms of local transformation, the tool itself may still be a thin wrapper over a slow dependency. If the tool is short and the orchestration span is long, the issue is likely in planning, serialization, or post-processing.

That basic triage is often enough to change priorities. You stop spending time on prompt engineering when the evidence says the dominant cost is in search queries, cache misses, or internal service locality.

Interpret alternating inference-tool loops as repeated budgets

Agents that iterate through plan, tool, and synthesize loops need a slightly different lens. Each loop consumes a separate budget, and each budget may have a different dominant cost. One loop may spend most of its time in inference, another in tools, another in validation. If you collapse all of them into one long span, you lose the ability to identify which turn is pathological.

Model each turn as a separate inference span and, when the model requests a tool, a separate `execute_tool` span. The orchestration span can remain the stable container for the whole run, but the inner spans should preserve turn boundaries. Then you can answer questions like: Which turn caused the latency spike? Did the second tool call take longer because the first result changed the query shape? Did the model loop because the tool output was invalid?

The answer often matters more than the total runtime. A three-turn agent that spends 300 ms, 5 s, and 200 ms is not the same as one that spends 2 s, 2 s, and 2 s. The mean is the same order of magnitude, but the tuning strategy is different. One suggests a downstream bottleneck. The other suggests repeated orchestration or model hesitation.

Account for orchestration overhead explicitly

Orchestration overhead is the category most likely to be lost if you do not instrument it directly. It includes planner logic, state updates, schema checks, tool routing, retry coordination, result aggregation, and response formatting. In many systems, this is the work that makes the agent "feel slow" even when the model and tool spans look reasonable.

The key measurement is exclusive orchestration time. If you already have nested model and tool spans, the orchestration span's wall-clock duration minus child durations reveals the time spent in local control flow. That number is actionable. It tells you whether the runtime is spending too much time in serialization, framework callbacks, event-loop coordination, or post-processing.

In practice, this bucket is where framework design choices show up. A heavily layered callback system can add tens or hundreds of milliseconds per turn. A state machine that reloads context on every step can create avoidable I/O. A serializer that copies large payloads between components can dominate the critical path. None of those are model problems. They are orchestration problems, and the trace must isolate them if you want to fix them.

Queue wait and deferred work deserve their own treatment

Not all agent work happens on the original request thread. Some tools enqueue jobs, some workflows wait for approval, and some callbacks resume later in a different worker. If you only look for synchronous child spans, you will miss the waiting time and misattribute the delay.

Treat queue wait as a separate latency component. It is often visible as the gap between the enqueue span and the consumer span. If the consumer runs in a different trace, use a span link to preserve causality. Then the latency analysis can distinguish compute time from idle time.

This matters because a slow agent may not be compute-bound at all. It may simply be waiting in a saturated queue or blocked on a workflow engine.

You should also distinguish queue wait from tool time. A tool that executes instantly after dequeue is not the same as a tool that spent 6 seconds in business logic. The user experienced both as delay, but only one indicates an inefficient tool implementation. The other may indicate resource saturation or topology problems.

Be careful with deterministic post-processing

Deterministic post-processing is easy to ignore because it often looks small until it is not. Agents frequently normalize text, convert schemas, rank results, deduplicate citations, format tables, or apply policy filters before emitting a response. Those operations can consume measurable time, especially when the tool output is large.

If you do not instrument post-processing, that time falls into an unattributed tail at the end of the workflow span. The result is a misleading story: the model was fast, the tool was fast, and the system still took 4 seconds. In reality, 3 seconds may have disappeared into result assembly. Internal spans around formatting, validation, or filtering make that visible.

Post-processing spans should remain coarse. Do not instrument every string operation. Instrument the phases that can dominate latency or fail independently: parsing, normalization, ranking, deduplication, and final rendering. Those are the boundaries that matter to the SLO and to incident response.

Turn decomposition into decision support

Once the buckets are explicit, the trace can guide optimization. If inference dominates, look at model routing, prompt inflation, streaming strategy, or context window pressure. If tool time dominates, focus on

cache design, query shape, rate limits, service locality, or fan-out strategy. If orchestration dominates, inspect framework overhead, state persistence, serialization, or planner loops. If queue wait dominates, examine worker saturation, batch sizing, or scheduling policy.

This is where agent telemetry differs from ordinary service telemetry. The same end-to-end latency number can lead to very different fixes depending on which bucket dominates. A 12-second request with 10 seconds in retrieval is a search problem. A 12-second request with 10 seconds in orchestration is a runtime problem. A 12-second request with 10 seconds in repeated inference is a planning problem. The trace should make those cases separable.

Use consistent queries and aggregation rules

Latency decomposition only works if your backend computes it consistently. The same span structure should feed dashboard panels, alert rules, and ad hoc investigation queries. If one query sums child durations while another uses critical path, the numbers will not match and confidence erodes quickly.

For user-facing SLOs, aggregate at the workflow span and use critical-path timing for the user-visible response. For capacity or cost analysis, aggregate cumulative child durations and per-component counts. For debugging, look at the per-span timeline and the gaps between spans. Each view answers a different question. Do not force one number to do all three jobs.

It also helps to keep your internal abstractions stable even if GenAI semantic conventions evolve. The exact field names for model spans or tool spans may change, but your dashboards should query your own normalized attribute set where possible. That protects the latency analysis from version churn in the upstream conventions.

Common mistakes distort the picture

Several errors recur in production systems:

- attributing all delay to the model span;
- measuring only aggregate endpoint latency;
- adding nested child durations as if they were serial;
- ignoring orchestration overhead because it is local code;
- treating queue wait as tool execution;
- optimizing the fastest component because it is the only one visible.

Each mistake changes the optimization target. If the tool time is actually the bottleneck, prompt tuning gives you the illusion of progress while the user still waits. If orchestration overhead is the issue, caching the model response does nothing. If fan-out is parallel, summing child spans exaggerates latency and leads you to chase the wrong service.

The cure is structural. Make the workflow boundary explicit. Make the model boundary explicit. Make the tool boundary explicit. Then let the trace answer the question the dashboard usually cannot: where the time actually went.

4.4. Representing Failures, Retries, and Idempotent Tool Telemetry

A trace that ends in `OK` can still describe a bad execution. An agent may answer correctly after three tool retries, a timeout, a provider fallback, and a compensating cleanup step that hid a stale intermediate result. If you only record the final success, you erase the operational story that

explains latency spikes, budget burn, and fragile behavior. The trace then becomes polite rather than truthful.

The task here is to preserve failure semantics without turning every trace into an unreadable chain of red spans. You want enough structure to answer incident questions, but not so much noise that operators stop trusting the view. The right model separates attempt-level failure from overall request success, records unhandled exceptions accurately, and uses stable identifiers so retries can be distinguished from duplicate side effects.

A minimal retry pattern shows the shape clearly:

```
for attempt in range(1, max_attempts + 1):
    with tracer.start_as_current_span(
        f"execute_tool quote_order attempt {attempt}",
        kind=SpanKind.CLIENT,
    ) as span:
        span.set_attribute("gen_ai.operation.name",
                           "execute_tool")
        span.set_attribute("gen_ai.tool.name", "quote_order")
        span.set_attribute("tool_call_id", tool_call_id)
        span.set_attribute("idempotency_key", idem_key)
        span.set_attribute("attempt_number", attempt)
        try:
            result = quote_order(order)
            break
        except TimeoutError as exc:
            span.set_status(Status(StatusCode.ERROR))
            span.set_attribute("error.type", "TimeoutError")
            span.record_exception(exc)
```

That pattern encodes the central rule: mark the failed attempt as failed, not the whole logical operation if the later attempt succeeds. The final wrapper span remains unset for status when the operation eventually completes successfully. That distinction is essential for reliability analysis and aligns with OpenTelemetry guidance on error recording.

Separate failure classes before you instrument them

Not every bad outcome deserves the same telemetry. Deterministic

tool errors, transient dependency failures, timeouts, cancellations, invalid tool outputs, planner mis-specification, duplicate execution, and compensating actions after partial completion all need different representations.

A deterministic error usually indicates that the tool was called with inputs that will not work until the logic changes. A timeout may indicate a slow dependency or a bad timeout budget. A cancellation often means the work was abandoned intentionally because the user moved on or a higher-priority request superseded it. Invalid tool output can mean the tool returned a malformed payload, but it can also mean the agent requested the wrong shape. Duplicate execution may be a correct retry or a business bug, depending on side effects. Compensating actions are not failures by themselves; they are recovery steps after a partial effect.

You should not collapse those into one generic `error` attribute. The incident response question is not merely "did it fail?" but "how did it fail, and did we recover correctly?" The trace must preserve that distinction.

Record failure at the attempt level, not only at the wrapper level

The safest default is one span per attempt. If the first attempt times out and the second succeeds, the first span gets error status and the second span remains successful. The enclosing logical operation can stay successful if the overall workflow completed successfully. That keeps the attempt history visible without forcing the final state to inherit a failure that no longer exists.

This matters because a successful response can hide poor execution health. A user does not care that the first attempt timed out if the final answer arrives quickly enough, but the operator very much cares when the request needed multiple tries. Attempt-level spans make retry pres-

sure visible, which helps you distinguish a healthy fallback from an unstable dependency.

Avoid the opposite mistake as well: do not collapse all retries into one long span. A single span that runs across three attempts makes the trace look clean, but it destroys the ability to answer how many attempts actually occurred, which attempt failed, and what the retry policy did. When the span is long and ambiguous, incident reviews become guesswork.

Set status carefully and keep success truthful

OpenTelemetry error guidance is strict in the way that helps reliability analysis: if the operation ultimately succeeds, do not mark the final successful operation span as failed merely because intermediate attempts failed and were retried. Mark the failed attempt spans with error status. Leave the final logical span unset for status if no error remains. Reserve error status for spans that truly ended in failure.

That rule keeps success semantics honest. A workflow span may finish with no error even though it contains failed child attempts, because the workflow as a whole succeeded. A final retry span may be successful even when earlier siblings were not. If you stamp the parent with error simply because there was turbulence along the way, you make dashboards noisy and alerts less meaningful.

Unhandled exceptions should be captured as exception events. Include `exception.type` and `exception.message`, and add a stack trace when it is appropriate for your environment. Handled exceptions do not need to be duplicated as exception events if they are part of an expected retry path and do not leave the operation failed. That keeps the event stream focused on truly abnormal endings rather than on every local catch block.

Use stable identifiers to separate retries from duplicates

Retries and duplicates can look similar in raw logs. A second call to the same tool with the same arguments might be a correct retry, an idempotent reissue, or an accidental duplicate. You need identifiers that survive attempt boundaries and tell you which case you are looking at.

The most useful identifiers are `tool_call_id`, `idempotency_key`, and `attempt_number`. Keep them attached to the attempt span. `tool_call_id` identifies the logical invocation. `attempt_number` distinguishes the retries. `idempotency_key` tells you whether repeated execution should have been safe. Together they let you reconstruct the execution history without inspecting raw tool payloads.

This separation is especially valuable when a tool has side effects. If attempt 1 times out after the side effect was committed, and attempt 2 replays the same action, the telemetry must let you see whether the replay was safe or dangerous. A duplicate action can be acceptable in an idempotent flow and catastrophic in a non-idempotent one. The span should carry that operational difference.

Represent compensating actions as first-class work

A compensating action is not a cosmetic cleanup step. It is evidence that the system had to reverse or neutralize a partial effect. If a tool inserted a stale record, invalidated a quote, reverted a workflow state, or suppressed an old result after a late response, that work should appear in telemetry as its own span or as a clearly named child under the original operation.

Do not hide compensation inside an unremarkable success path. If the system had to roll back or invalidate something, that fact matters for reliability analysis. A request that “succeeded” only after cleanup is not the same as a request that executed cleanly the first time. The compensating span lets you count those events and measure how often recovery logic runs.

Compensating work also helps explain cost. In some systems, the expense of recovery rivals the expense of the original operation. If you omit the cleanup step, the trace understates both latency and operational load.

Timeouts need cause attribution

A timeout is often reported as a generic error, but the underlying cause matters. Did the tool itself stall? Did the queue delay exceed the budget before the tool even ran? Did the remote service respond too slowly? Did the planner set an unrealistically short budget?

Capture the timeout as an attempt failure and annotate the cause as precisely as the implementation can support. If the timeout happened while waiting on the downstream dependency, the child span should end in error and the parent tool span should reflect that the tool failed because a dependency exceeded its window. If the timeout happened before the task was dequeued, that is queue wait, not tool execution latency. If the tool was cancelled because the agent loop abandoned the path, record cancellation rather than timeout.

This distinction is not pedantic. It changes the tuning target. A queue timeout leads you to scale workers or adjust scheduling. A dependency timeout leads you to fix the dependency or widen the budget. A planner timeout leads you to improve orchestration behavior.

Handled errors and retries should not pollute the success path

A healthy retry policy catches expected transient failures and continues. That does not mean those failures should disappear. It means they should live in the failed attempt spans, not in the final successful operation span. If you add exception events to the final span for every handled failure, you make the final success look dirty even when the recovery was correct.

Use the attempt spans to preserve the story. The parent workflow span can carry a summary attribute such as `retry.count` or `tool.attempts`, but avoid overstating the error state. A single summary event on the parent may help with readability, especially in traces with many attempts, but the summary should point to the attempt history rather than replace it.

This is the balance that keeps traces readable and analytically safe. The final success stays clean. The retries remain visible. The recovery logic becomes part of the observed behavior rather than a hidden implementation detail.

Model retries as a policy, not only as a loop

A retry loop in code is an implementation. A retry policy is the operational intent. Your telemetry should reflect both. The span attributes should make the policy visible enough to explain the behavior: maximum attempts, backoff class, whether the retry is idempotent, and whether the retry is caused by a transient dependency or a semantic mismatch.

Those details matter because different retry policies produce different failure signatures. An aggressive retry policy may increase success rate while inflating latency and cost. A conservative policy may fail faster but surface upstream instability sooner. The trace should let you see which policy was in effect when the request ran.

If your system supports different retry modes for different tool classes, record that choice. A read-only retrieval tool and a side-effecting approval tool should not share the same retry assumptions. The telemetry should show that distinction explicitly, because operational risk changes with side effects.

Use tail sampling to preserve rare failure-rich traces

Failure-rich traces are often the most valuable traces and also the least

common. Head sampling can drop them before the collector has a chance to see the retries, the fallback, or the cleanup path. Tail sampling is usually a better fit when you want to preserve rare long-running or failure-heavy agent traces, because the collector can wait for the outcome before deciding what to keep.

That benefit comes with a constraint: all spans for a given trace must reach the same collector instance for the sampling decision to be correct. If your architecture shards trace traffic in a way that splits a trace across collectors, the sampling policy may make incomplete decisions. For agent systems with multi-step workflows and async continuations, that routing requirement matters.

Tail sampling is not a universal answer. It introduces collector complexity and operational coupling. But for incidents where retries, fallbacks, and eventual success matter most, it preserves the evidence that head sampling often discards.

Distinguish eventual success from healthy execution

A useful answer is not always a healthy execution. An agent that retries three times, falls back to a secondary service, and then compensates a stale result may produce the right user response, but the trace still describes a fragile path. You want the trace to make that fragility visible.

That is why the final successful span should remain clean while the history of attempts, failures, and recoveries stays intact in sibling or child spans. If the parent workflow span shows no error and the child attempts show the retries, you can tell that the user saw success while the system worked hard to achieve it. That is the truth the reliability team needs.

The trace model should therefore answer four questions at once: what the user received, what the model decided, what the tools attempted, and what the system had to do to recover. If those answers remain

distinct, the telemetry supports incident response instead of obscuring it.

Chapter 5

RAG and Embedding Pipeline Performance Analysis

RAG systems often look slow, expensive, or unreliable for reasons that are invisible if engineers trace only the final inference call. This chapter shows how to turn the retrieval pipeline into a first-class observability surface, exposing where embeddings stall, search broadens, rerankers bottleneck, and context assembly silently inflates downstream token cost.

5.1. Embedding Generation as a Distinct Performance Domain

A RAG request can look deceptively simple in traces: a query arrives, the answer model runs, and the latency budget appears to be domi-

nated by generation. That view collapses the most expensive hidden work into one bucket and obscures the real bottleneck. Query-time embedding, offline corpus indexing, periodic re-embedding, and repair jobs all have different resource profiles, queueing behavior, and failure modes. If you trace them as a generic prelude to retrieval, you lose the very signals that explain why a system is slow, stale, or unexpectedly costly.

A concrete instrumentation pattern

A useful starting point is to separate online query embeddings from offline indexing embeddings at the span boundary and to record lineage explicitly. For request-time embedding, make the embedding span a child of the user request and attach the embedding model, input kind, batch size, token count, and vector dimension as attributes. For indexing, create a worker span for the queue dequeue, then a batch-embedding span for the actual model call, and link that span back to the ingest job or document-processing span rather than forcing an artificial parent-child relationship. The distinction keeps request latency clean while preserving enough provenance to diagnose reindex backlogs.

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

def embed_query(text, model_name, dim):
    with tracer.start_as_current_span("embed.query") as span:
        span.set_attribute("genai.operation", "embed")
        span.set_attribute("genai.input.kind", "query")
        span.set_attribute("genai.model.name", model_name)
        span.set_attribute("genai.embedding.dimensions", dim)
        span.set_attribute("genai.request.input_count", 1)
        span.set_attribute("genai.request.tokens", len(text.split()))
        return call_embedding_api([text], model_name, dim)

def embed_batch(docs, model_name, dim, ingest_span_ctx):
    links = [trace.Link(ingest_span_ctx)]
    with tracer.start_as_current_span(
        "embed.batch",
```

```
        links=links,
    ) as span:
        span.set_attribute("genai.operation", "embed")
        span.set_attribute("genai.input.kind", "document")
        span.set_attribute("genai.model.name", model_name)
        span.set_attribute("genai.embedding.dimensions", dim)
        span.set_attribute("genai.request.input_count", len(docs))
        span.set_attribute("genai.batch.size", len(docs))
        return call_embedding_api(docs, model_name, dim)
```

That shape is intentionally different from an inference span. Embedding spans represent an encoding step, not a conversation turn or a completion. If you reuse generic model-inference conventions, you will misclassify throughput limits, retry behavior, and token accounting. GenAI semantic conventions are still evolving, so record the version or opt-in mode used by your SDK and collector pipeline when you rely on experimental embedding attributes. Stable tracing practice matters more than any single attribute name: the instrumentation must remain intelligible when libraries change defaults or when a collector upgrades its schema mapping.

Two execution domains, two observability stories

Online query embeddings and offline corpus embeddings solve different operational problems. Query embeddings live on the critical path of the user request. Their latency contributes directly to time-to-first-answer, and their tail behavior is often determined by downstream service contention, transient throttling, or local serialization overhead. Offline embeddings belong to a throughput-oriented ingestion pipeline. Their main risks are queue growth, batch inefficiency, reprocessing lag, and index freshness drift.

That difference drives how you model them. A query embedding should usually be a normal child span under the request span, because the user is waiting and the causal path is linear. The useful question is whether the embedding service, tokenizer, or local CPU saturation delayed the request. An indexing embedding should usually be modeled

as a worker span with explicit queue-delay measurement and a link to the upstream ingest or document-change event. The useful question is whether the corpus is falling behind, whether batch size is oscillating, or whether a reindex job is monopolizing capacity.

A single trace tree cannot always express both concerns cleanly. For asynchronous workloads, trace links are often more truthful than parentage because the job that triggered the work may have finished long before the embedding batch runs. Use the link to preserve lineage, and use metrics to describe the queue itself. That combination gives you both causality and operational state.

The signals that actually explain embedding cost

Embedding generation has a resource envelope that differs from retrieval and differs again from answer generation. You care about model-call latency, but you also care about queue delay, batch fill efficiency, payload size, tokenization overhead, and any normalization or serialization cost on the client side. A request can be fast at the model boundary and still be expensive because the worker spent most of its time waiting for a batch to fill or because tiny batches wasted throughput.

Capture at least four classes of signals:

- **Latency decomposition:** queue wait, preprocessing time, model RPC time, postprocessing time, and write-to-index time.
- **Throughput shape:** requests per second, documents per batch, batch-size distribution, and backlog depth.
- **Correctness and compatibility:** input validation failures, empty-input rejection, token-limit violations, retry exhaustion, and model-version mismatches.
- **Capacity and cost:** tokens processed, output dimensions, se-

rialized payload size, and any provider-side rate-limit or quota events.

This breakdown is more useful than a single "embedding latency" metric because the same end-to-end number can arise from very different causes. If queue delay dominates, you need more workers or looser batch thresholds. If model RPC dominates, you need better concurrency, a different embedding model, or a different service tier. If post-processing dominates, your own code is the bottleneck.

Batching is a performance decision, not a logging detail

Embedding pipelines almost always batch, but batching has a nontrivial operating envelope. Larger batches usually improve throughput by amortizing request overhead, yet they also increase queue delay and can worsen tail latency for online traffic. Smaller batches reduce waiting time but may underutilize the embedding service and increase per-item overhead. The right choice depends on whether the path is synchronous or asynchronous and on whether your dominant constraint is user latency or corpus freshness.

The batch-size distribution belongs in metrics, not just logs, because oscillation is a common failure mode. When traffic is bursty, a worker queue can alternately overfill and drain, producing a sawtooth pattern in batch size. That pattern often looks like instability in the embedding service when the real culprit is the batcher. Record batch size at span start and queue age at dequeue so you can distinguish a slow provider from an overly conservative batching policy.

For offline indexing, batch aggregation should be visible as a stage in the trace. The batch span should begin after the queue dequeue and end after the vector write or upsert succeeds. If your pipeline performs document chunking upstream, record the fan-out count as well. The number of chunks per source document is often the hidden multiplier

that turns a modest ingestion rate into an indexing backlog.

Token limits and input validation belong in the observability model

Embedding APIs enforce request-shape constraints. Inputs cannot be empty, and model token limits bound the maximum amount of text you can send in one call. Those are not merely application errors; they are performance signals because they drive chunking policy, batch fragmentation, and retry behavior. A high rate of token-limit failures usually means your chunker is too permissive or your preprocessor is collapsing too much content into a single embedding input. A high rate of empty-input failures usually means upstream filtering is producing dead chunks or malformed records.

Record validation failures with structured error classes rather than a generic failure counter. Then you can distinguish malformed source data from provider throttling and from transient transport failures. When the embedding worker retries, include the retry count and the backoff strategy as span attributes. A slow request that succeeds after three retries is operationally very different from a slow request that simply used a large batch.

Vector dimensionality is part of the runtime footprint

Embedding output dimensionality affects storage, network transfer, index memory, and approximate-nearest-neighbor behavior. Lower dimensions reduce the size of each vector and usually reduce the cost of shipping and storing embeddings, but they can also reduce recall quality if you compress too aggressively. The dimensionality decision is therefore not a static model detail; it is part of your performance contract.

Treat dimension as a first-class attribute on both query and corpus embedding spans. That makes it possible to correlate search quality

regressions with a dimensionality change during model migration or cost optimization. If you run mixed-dimension corpora during a partial reindex, record the dimension used for each document so retrieval anomalies can be tied to the correct embedding version. A vector store that accepts multiple dimensions or multiple indexes may otherwise conceal a heterogeneous corpus behind a deceptively stable API.

Normalization changes how you interpret scores

Many embedding systems produce L2-normalized vectors by default. When vectors are normalized, cosine similarity, Euclidean distance ranking, and dot-product ranking can induce the same ordering under common retrieval setups. That matters for observability because the backend score you emit does not always mean what analysts think it means. If the vectors are normalized, a score change may reflect the geometry of the candidate set rather than a new embedding malfunction.

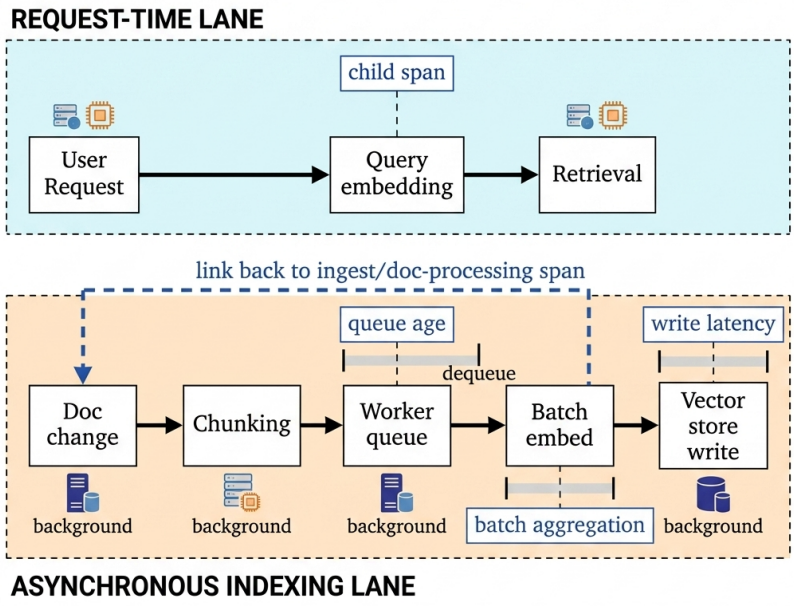
Do not log raw similarity scores without also recording whether the embedding output was normalized and which distance metric the downstream index used. Otherwise a search regression can be misdiagnosed as an embedding regression when the actual change occurred in the index configuration or the ranking function. For mixed backends, normalization metadata is also a compatibility guardrail: if one path assumes normalized vectors and another does not, score distributions become incomparable.

How to trace the offline indexing path

Offline indexing is where embedding observability often breaks down, because the work is decoupled from the request that caused it. A practical flow looks like this: a document-change event enters an ingest pipeline, the document is chunked, chunks are queued, workers dequeue batches, embedding calls run, and the resulting vectors are written to the vector store. Each step should expose its own span or event

boundary.

A compact architecture diagram clarifies the separation between request-time and batch-time work.



In the synchronous lane, the query embedding span should sit directly under the request span. In the asynchronous lane, the queue and batch spans tell you where freshness is being lost. If the queue age grows while the batch model RPC time stays flat, the issue is capacity or batching policy. If the vector write latency grows, the bottleneck is probably

the index or network path, not the embedding model.

For batch indexing, preserve lineage by linking the batch span to the upstream ingest or document-update span. Do not try to force the document event itself to remain an active parent if the actual embedding work runs minutes later. Trace links make the relationship explicit without falsifying execution order.

Metrics complement traces; they do not replace them

A trace tells you why one batch was slow. Metrics tell you whether that slowness is systemic. For embedding pipelines, the most useful counters and histograms are the ones that capture rate, batching, backlog, and error class. Track embeddings per second, documents per batch, batch wait time, model RPC latency, vector-write latency, token counts, and success or failure by reason code. If you operate multiple embedding models or dimensions, label the metrics with model identity and output dimensionality.

Cardinality control matters here. Model name and deployment version are usually safe labels. Document ID, user ID, and raw source path are not. If you need drill-down, use traces or logs for the specific request and keep the metrics aggregate. That boundary prevents the telemetry system itself from becoming the bottleneck.

A separate freshness metric is often valuable for corpus indexing: age of the oldest unembedded change, or lag between source modification and vector-store availability. That number is the simplest way to answer whether retrieval quality is stale because embedding generation is behind. It is also the metric most likely to reveal backlog problems before users do.

Retry and throttling behavior need explicit visibility

Embedding services can throttle just like any other remote model API. The important diagnostic difference is that throttling affects not only

latency but also queue dynamics. If workers retry aggressively, they may occupy threads that could have been processing fresh work. If they retry too timidly, transient capacity spikes become user-visible failures.

Record retry counts, backoff durations, and provider rate-limit responses separately from generic transport errors. For offline indexing, a wave of throttling can cause the backlog to expand even if the input rate is unchanged. For online query embedding, the same event appears as a tail-latency spike on the request path. The right remediation differs: offline jobs can often be slowed or rescheduled, while online paths may need a faster fallback model or a stricter concurrency cap.

Correctness problems often masquerade as performance problems

Embedding pipelines are frequently blamed for slow retrieval when the actual issue is semantic drift or heterogeneous indexing. If documents are partially re-embedded after a model upgrade, old and new vectors may coexist in the same index and produce unstable similarity behavior. If chunking changes, the number of embeddings per source document changes as well, which alters both corpus density and retrieval recall. Neither symptom is visible if you only measure answer latency.

Record the embedding model version, chunking policy version, and index build epoch with each write. That provenance makes it possible to correlate quality anomalies with a specific reindex window. It also helps during rollback: if a new embedding model improves average latency but degrades recall on a specific document class, you can identify exactly which subset to reprocess.

Deciding how much lineage to preserve

There is a real trade-off between full trace fidelity and telemetry cost. One span per chunk gives you precise visibility into outliers and failure

localization, but it can explode span volume during large indexing jobs. One span per batch reduces overhead but hides which chunk inside the batch triggered retries or token-limit errors. The right balance depends on whether your main problem is index freshness, per-document debugging, or spend control.

A practical compromise is to emit batch spans always, per-chunk spans selectively, and structured events for chunk-level validation failures. That keeps the common path cheap while preserving enough detail to investigate the rare path. Sampling should be applied carefully: if you sample away the slowest batches, you lose exactly the evidence you need to understand backlog formation. Prefer deterministic retention of error spans and high-latency tails over uniform sampling alone.

Why this domain must stay separate from retrieval and inference

Embedding generation precedes retrieval, but it is not retrieval. It can fail, throttle, and queue independently of search. It can also dominate cost even when retrieval and generation appear healthy. Folding it into a generic “LLM” bucket hides the operational differences that matter most: whether the user is waiting on a query embedding, whether the corpus is stale because indexing is behind, whether a model upgrade changed vector geometry, and whether queue growth is due to batch policy or provider limits.

When you model embedding generation as its own performance domain, you get clearer ownership of latency, throughput, and correctness. You can trace request-time embeddings without contaminating answer latency analysis, track offline indexing without falsifying request causality, and correlate vector dimensions, normalization, batching, and model version with actual system behavior. That separation gives the rest of the pipeline a trustworthy foundation.

5.2. Tracing Retrieval, Vector Search, and Reranking

A retrieval stack can fail in three very different ways: it can return too slowly, it can return the wrong candidates, or it can return a plausible candidate set and then spend most of its time fixing that set with reranking. If you compress all of that into one "search" span, you erase the signal that tells you whether to tune the ANN index, relax a metadata filter, change the hybrid fusion policy, or cap reranker fan-out. The right tracing model treats retrieval as a chain of decisions, each with its own latency, cardinality, and failure surface.

A practical span layout

A useful starting point is to make the retriever orchestrator the parent span and then model each major retrieval decision as a child or sibling span according to execution topology. A semantic branch, a lexical branch, a merge step, and a reranker call should all be visible separately. When a managed vector service or external rerank API is involved, preserve the remote boundary with the correct client span kind rather than pretending the work happened in-process.

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

def retrieve(query):
    with tracer.start_as_current_span("retrieval.orchestrate") as
     span:
        span.set_attribute("retrieval.mode", "hybrid")

        with tracer.start_as_current_span("retrieval.semantic") as
     sem:
            sem.set_attribute("retrieval.branch", "semantic")
            sem.set_attribute("retrieval.k", 50)
            sem_hits = vector_search(query)

        with tracer.start_as_current_span("retrieval.lexical") as lex
     :
            lex.set_attribute("retrieval.branch", "lexical")
```

```
            lex.set_attribute("retrieval.k", 50)
            lex_hits = bm25_search(query)

        with tracer.start_as_current_span("retrieval.merge") as merge
    :
            merge.set_attribute("retrieval.semantic.count", len(
    sem_hits))
            merge.set_attribute("retrieval.lexical.count", len(
    lex_hits))
            candidates = merge_and_dedup(sem_hits, lex_hits)

        with tracer.start_as_current_span("retrieval.rerank") as rr:
            rr.set_attribute("retrieval.candidate.count", len(
    candidates))
            rr.set_attribute("retrieval.top_k", 10)
            top = rerank(query, candidates[:50])

        return top[:10]
```

The example is intentionally simple, but the structural choice matters. If semantic and lexical search run concurrently, their spans should be siblings under the orchestrator, not a nested sequence that implies dependence. If the reranker is external, let its span carry the network boundary and annotate the candidate count there. Once that structure is in place, the trace can explain not just that retrieval was slow, but where the time and recall went.

Model the pipeline as staged selection

Retrieval is not a single lookup. It is a sequence of selection steps that progressively reduce uncertainty. First, the system transforms the query into one or more search intents. Then each branch generates candidates. Then the candidates are filtered, merged, deduplicated, possibly rescored, and finally truncated to the top-k evidence set that the prompt assembler will consume. Each step can improve quality while also increasing latency or backend load.

The first observability rule is to expose the cardinality at every stage. Record how many candidates each branch produced, how many survived metadata filters, how many duplicates were removed, how many

reached reranking, and how many were returned. Those numbers often explain regressions better than latency alone. If answer quality drops after a search change, the culprit is frequently not the score function but a candidate-set shrinkage that was invisible because no one recorded it.

The second rule is to distinguish *decision latency* from *execution latency*. A query rewrite can add 5–10 ms without changing any external service. A reranker can add 100 ms while improving precision. A metadata filter can be effectively free when it is indexed, or painfully expensive when it requires post-filtering a large candidate set. Put those stages on separate spans so the timing breakdown can answer the operational question you actually have.

Vector search needs backend-aware tracing

Vector lookup is often treated as a database call, but operationally it is a specialized approximate search with its own index family and tuning parameters. If no dedicated vector-search semantic convention exists in your stack, use the database client span conventions as a fallback and attach the vector-specific attributes you need for analysis. At minimum, record the backend system, the collection or index name, the query mode, and whether the search was exact or approximate. For ANN systems, record the index family and the parameters that materially affect recall and latency.

That distinction is especially important for HNSW-like indexes. Search breadth and construction parameters trade latency against recall and memory. If you change one of those knobs, the trace should let you correlate the shift in tail latency with any change in candidate quality. Without that metadata, a slower or faster vector call can be misread as a service regression when the real cause is a deliberate index setting. The same principle applies to other index families such as inverted-file or product-quantized structures: the backend may hide the tuning, but

your trace should not.

When the vector search runs against a managed service, use a client span that makes the remote boundary explicit. If the service provides server-side spans, keep both sides. The client span tells you what your application paid for; the server span tells you how the backend spent the time. When you only have the client span, attach enough contextual attributes to recover the search shape later: query vector dimension, top-k, filter count, and any fallback mode.

Metadata filtering is not free

Metadata filters often look cheap because they are expressed as a clause attached to the search query. In practice, their cost depends on index support, selectivity, and whether the filter executes before or after candidate generation. A highly selective filter can reduce downstream cost by shrinking the candidate set early. A low-selectivity filter can add little overhead. A badly placed filter can force the backend to scan a broad candidate set and then discard most of it.

Trace the filter as its own stage when it materially affects candidate count or latency. Record the filter shape in structured form: number of predicates, indexed versus unindexed fields, and whether the filter was pushed down to the search backend or applied in the application. If you only keep the final result count, you lose the evidence that explains why a search that returned ten documents still took twice as long as expected. Candidate-set shrinkage is especially useful for diagnosing quality issues after policy changes, because a new filter can silently remove the document class that used to rescue borderline queries.

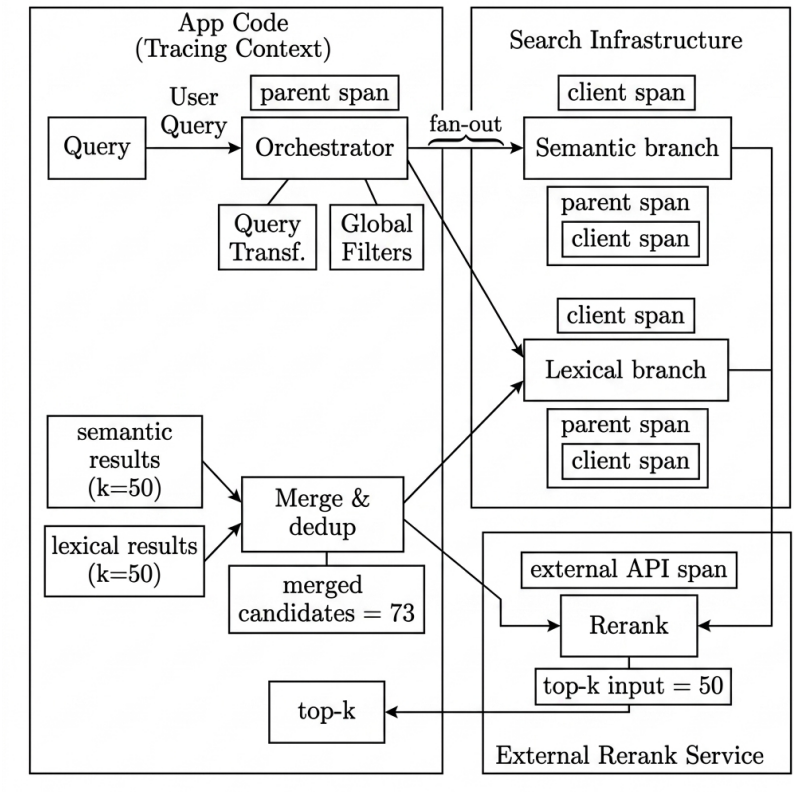
Hybrid search needs explicit branch concurrency

Hybrid retrieval usually mixes semantic search with lexical or sparse search, then fuses the results. That architecture is attractive because it recovers from the weaknesses of either branch alone, but it complicates

tracing because the branches are concurrent and may have different service boundaries. Treat the branches as siblings under a common orchestration span. If the lexical branch calls a search engine and the semantic branch calls a vector service, each branch should carry its own client boundary and its own timing breakdown.

The merge step deserves its own span as well. Merging is not a passive concatenation; it is a ranking decision that can change the top-k evidence set even when neither branch degrades. Record the input counts from each branch, the deduplication policy, the fusion method, and the final candidate count before reranking. If the merged set is smaller than expected, you may have a duplicate-heavy corpus or an over-aggressive normalization step. If it is larger than expected, the fusion policy may be letting too many low-confidence candidates through, which increases reranker load.

A compact diagram makes the control boundaries and span relationships concrete.



The diagram encodes a simple but important point: concurrency is visible at the orchestration level, not hidden inside a monolithic search span. If one branch slows down, the orchestrator span can still tell you whether the request was gated by the slow branch or whether the merge waited for a branch that returned a bloated candidate set. That is the difference between debugging a pipeline and guessing at a black box.

Reranking is a separate compute stage

Cross-encoder rerankers are a different class of work from candidate generation. They score query-document pairs jointly, which usually

yields better precision but prevents precomputing document-side embeddings for the rerank step itself. That architectural fact should show up in your traces. A rerank span should carry the candidate count, the selected top-k, the reranker model identity, and any truncation policy applied before scoring. If you batch rerank pairs, record the batch size and the number of tokens or bytes processed per batch.

Reranking is the stage most likely to collapse throughput when candidate generation expands unexpectedly. A search pipeline that is healthy at 20 candidates can become expensive at 200 candidates even if the vector search itself barely changes. The reranker span makes that blow-up visible. If reranking is external, model the remote call boundary directly. If it is in-process, still separate the CPU-bound scoring span from the retrieval spans so you do not misattribute reranker cost to vector lookup.

Do not treat reranker scores as absolute truth across model versions. A score distribution can shift when you change the reranker, alter candidate ordering, or adjust input truncation. Capture the reranker version and the candidate window that actually reached it. If the reranker consumes only the first 50 candidates after merge, the trace should make that cutoff explicit; otherwise a recall regression can be hidden behind a seemingly healthy rerank score.

Trace causality, not just call structure

The hardest observability problem in retrieval is fan-out. Search branches are often parallel, speculative, or fall back to alternate services when one branch degrades. A strict parent-child tree can therefore lie about the runtime structure. Use span links when work is logically related but not strictly nested, especially across asynchronous retries, fan-out merges, and backend jobs. Use sibling spans when branches execute concurrently under one coordinator. Reserve nested spans for actual containment of work.

This distinction matters operationally. Suppose a lexical branch times out, and the orchestrator falls back to semantic search alone. If the timeout is represented only as an error log, you lose the fact that answer quality may have dropped because the candidate set was intentionally degraded. If the timeout is a visible branch span with fallback metadata, you can group incidents by fallback presence and separate backend instability from search quality issues. The same principle applies to speculative searches, cached candidate reuse, and reranking that is skipped because the candidate set was too small to justify the cost.

What to measure alongside traces

Traces are necessary but not sufficient. Retrieval diagnostics improve when you pair them with stage-specific metrics: branch latency histograms, candidate counts, reranker invocation rates, merge dedup ratios, filter selectivity, and fallback frequency. For ANN-backed systems, recall-oriented offline evaluation and trace-oriented online timing answer different questions; both matter. An offline recall metric tells you whether an index configuration is capable of finding the right items. The online trace tells you whether the request path spent its time in vector lookup, filtering, or reranking.

This is also where top-k becomes a performance variable rather than a static parameter. A larger k can improve evidence coverage, but it also raises rerank cost and can inflate the prompt later. Record the chosen top-k at each stage, not just the final answer count. If your system exposes managed retrieval knobs such as result count limits or ranking options, tie those settings to the trace so a quality regression can be correlated with the exact retrieval policy that produced it.

Common failure modes that traces should expose

Monolithic “retrieve_context” spans are the most common anti-pattern. They tell you almost nothing about whether the problem was

search latency, branch imbalance, or reranking overhead. Another common mistake is to bury reranking in application logs because it is "just scoring." That hides the cost center that often grows fastest when candidate generation is widened. A third mistake is to treat metadata filtering as free because it occurs inside the search backend; in reality, it may be the dominant source of candidate loss or latency.

A more subtle failure mode appears during service migrations. A team might move from an in-process vector library to a managed search backend and preserve the same logical trace name. The result looks consistent until latency shifts, backend span detail disappears, and the team can no longer tell whether the regression came from ANN tuning, network round-trips, or a changed filter execution plan. Preserve the boundary in your tracing model when the implementation boundary changes. That extra structure pays for itself the first time you need to localize a regression.

Versioning and compatibility affect the trace shape

Retrieval instrumentation is only as stable as the components beneath it. Managed vector services expose different timing breakdowns, result metadata, and error classes than in-process libraries. Some search stacks return branch-level scores; others only expose the final result set. OpenTelemetry conventions for database and client spans give you a portable fallback, but the attributes you choose for vector-specific details should be versioned in your instrumentation notes or deployment metadata. That is especially important when the retrieval stack mixes lexical search, vector search, and external reranking APIs with different telemetry maturity.

If you change the ANN implementation, reranker model, or hybrid fusion policy, keep the trace names stable and change the attributes or version labels instead. Stable names let dashboards remain useful across upgrades. Versioned attributes let you compare behaviors with-

out corrupting historical data. In practice, that means the retriever orchestrator should survive implementation churn even when the underlying services do not.

What good retrieval tracing buys you

Once retrieval is split into query transformation, candidate generation, filtering, merge, and reranking, the system becomes diagnosable in the way an engineer needs. You can tell whether a latency regression came from the semantic backend, the lexical branch, or the reranker. You can tell whether a quality regression came from a smaller candidate set, from over-filtering, or from a fallback path that quietly removed one branch. You can also see when throughput collapses because the reranker is now scoring too many candidates after a retrieval policy change.

That granularity is what turns search from an opaque service call into a staged decision pipeline. Each decision leaves a different trace signature, and each signature points to a different operational remedy.

5.3. Correlating Retrieval with Inference Outcomes

A traced retrieval pipeline can still look healthy while the system fails in ways users actually feel. The candidates may arrive on time, the reranker may behave, and the vector index may never miss a deadline, yet answers remain vague, expensive, or simply wrong. That gap is the reason correlation matters: retrieval telemetry becomes operationally useful only when you connect it to the downstream model span, the request lineage, and the business outcome that the user experiences.

Start with one request lineage A practical way to ground the analysis is to propagate a stable request identifier through retrieval and inference, then join the stages in your trace backend or analytics layer. Keep the retrieval spans focused on candidate generation and ranking, and attach the lineage key to the inference span rather than trying to infer causality from timestamps alone. A simple OpenTelemetry-style pattern is enough to make the point:

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

def handle_request(req):
    rid = req.headers["x-request-id"]
    with tracer.start_as_current_span("rag.request") as span:
        span.set_attribute("request.id", rid)

        with tracer.start_as_current_span("retrieval") as rspan:
            rspan.set_attribute("retrieval.top_k", 10)
            docs = retrieve_context(req.query)

        with tracer.start_as_current_span("inference") as ispan:
            ispan.set_attribute("request.id", rid)
            ispan.set_attribute("context.count", len(docs))
            ans = generate_answer(req.query, docs)

        return ans
```

That minimal join key does two things. First, it lets you compare retrieval features against answer outcomes without guessing which retrieval run fed which model call. Second, it preserves the distinction between stages, so you can tell whether a high-latency answer was delayed by retrieval, by prompt construction, or by the model itself. The point is not the exact attribute names; the point is that the same request must be visible across the stage boundary.

Correlate the features that matter Not every retrieval attribute deserves equal analytical weight. The features that most often explain downstream behavior are the ones that change what the model actually

sees: candidate count before and after filtering, top-k selected, source diversity, retrieval freshness, reranker score spread, fallback-path activation, and average chunk size. Those are the variables that connect search behavior to generation behavior.

Candidate counts tell you whether the model saw a rich evidence set or a heavily pruned one. If filtering reduced the candidate set from 80 to 6, a good answer may still be possible, but the retrieval stage has already constrained the model. Freshness tells you whether the evidence set reflects the current corpus or stale content from a previous indexing cycle. Diversity tells you whether the top chunks all came from the same document or the same section, which often produces repetitive or brittle answers. Reranker score spread is useful because a narrow spread suggests ambiguity, while a sharp separation suggests the reranker had a strong preference and the model likely received a cleaner signal.

Fallback-path activation is especially important in incident review. If the semantic branch timed out and the system silently fell back to lexical-only retrieval, the answer may still be fluent but materially less grounded. That is not an inference bug. It is a retrieval degradation that only becomes visible when the fallback flag follows the request into downstream analysis.

Join retrieval outcomes to model outcomes, not just model latency A common mistake is to stop at latency correlation. That tells you whether retrieval made the answer slower, but not whether it changed the answer. You need to compare retrieval signals against both performance and quality indicators. Useful downstream measures include prompt token growth, completion latency, retry count, finish reason, user-visible quality labels, and any business KPI you already trust, such as task completion or answer acceptance.

This is where the RAG framing matters. The original retrieval-augmented generation design makes retrieval quality part of generation quality, not a separate concern. If retrieval quality drops, generation can still appear healthy at the span level while the answer quality falls. That is why a retrieval configuration that improves offline recall or ranking metrics is not automatically better in production. Offline metrics such as recall@k or NDCG describe the candidate set; they do not guarantee that the selected chunks were actually used well by the model. Long-context behavior compounds the problem: evidence placed in the middle of a large prompt can be underused even when it was retrieved correctly.

The practical implication is that you should group requests by answer outcome and compare their retrieval signatures. If poor answers cluster around low diversity, small candidate sets, or stale retrieval, you have a retrieval problem. If they cluster around long prompt windows, unusually large token counts, or repeated truncation events, the issue may be prompt assembly. If they cluster around stable retrieval but high model retry rates or unusual finish reasons, the issue may sit in inference. The analytical mistake is to assume that all poor answers originate where the complaint is loudest.

Use ordered metrics, not only binary success flags Retrieval quality is order-sensitive. The highest-ranked chunk usually matters more than the tenth candidate, and a reranker may improve precision by moving one key passage upward rather than by increasing the total candidate count. That makes order-aware metrics such as NDCG especially useful for top-k analysis, because they reward relevant items appearing earlier in the ranked list. When only a few chunks survive into the prompt, ranking position matters more than raw hit rate.

Yet even order-aware offline metrics are only one piece of the picture. A retrieval set can score well and still fail in production if the prompt as-

sembler truncates the relevant passage, duplicates irrelevant content, or places the strongest evidence in a part of the context the model underuses. That is why correlation should include the final prompt shape: number of retrieved chunks rendered, bytes added by templating, truncation count, and any evidence dropped after ranking. Those features bridge the retrieval stage to the inference stage and explain why the same ranking quality can produce different answer quality after assembly.

Separate causality from coincidence When answer quality changes, you need to decide whether retrieval, retrieval execution, or inference caused the change. A good decision rule is to inspect the first stage where the request cohort diverges from baseline.

If the cohort shows lower freshness, fewer candidates, or more fallback activations, investigate retrieval design or execution first. If the cohort shows stable retrieval but longer prompts and higher token usage, investigate prompt assembly. If the cohort shows stable retrieval and stable prompt size but higher completion latency or more retries, investigate the model stage. This ordering is not arbitrary; it follows the causal chain from evidence availability to evidence consumption.

The same rule applies to cost. A retrieval change can lower latency while increasing token spend if it returns larger or more redundant context. A reranker change can improve answer quality while increasing both retrieval and inference cost if it pushes more candidate text into the prompt. A model change can make inference faster while leaving retrieval-related cost untouched. Cost analysis only becomes meaningful when retrieval telemetry is joined to downstream token usage and model latency.

A practical cohort analysis flow A simple workflow is usually enough to localize the problem. Identify a cohort of poor answers

or slow answers. Group those requests by retrieval fallback presence, candidate-set shrinkage, reranker spread, and retrieval freshness. Then compare prompt token count, completion latency, retry rate, and user feedback between the cohort and a healthy baseline.

The sequence matters. Start with retrieval shape because it is often the easiest way to see whether the model received enough evidence. If the poor-answer cohort shows smaller candidate sets and lower freshness, you have an upstream retrieval issue. If the poor-answer cohort has normal retrieval but much larger prompts, you are likely dealing with context inflation rather than search quality. If the poor-answer cohort has both healthy retrieval and healthy prompt shape, the model or downstream application logic becomes the more likely source of failure.

That analysis also handles regressions that are not obvious from averages. For example, a retrieval tuning change may improve the median request but hurt a minority cohort that depends on one rare document type. If you do not preserve request lineage and cohort labels, the overall metrics may still look acceptable. The problem surfaces only when you slice by retrieval branch, fallback status, or source diversity.

Tie business context to telemetry carefully Request lineage alone does not tell you whether a poor answer matters. You need the business context that turns technical behavior into an operational decision. In practice, that means adding governed business identifiers or cohort labels, not raw user data, so you can distinguish one document class, one tenant, or one task type from another. This is the boundary between observability and analytics governance: the trace records the mechanism, while the business dimension says why the mechanism matters.

Use baggage or another controlled propagation mechanism for stable

business labels when they must cross service boundaries. Avoid throwing those labels into ad hoc span attributes if they will create high-cardinality noise or privacy problems. The same discipline from general tracing applies here: the wrong identifier in the wrong field makes the telemetry harder to query and harder to trust. When you need tenant-level or workflow-level analysis, attach only the label necessary to group the cohort and keep the sensitive detail in governed storage.

Interpret retrieval metrics in the context of prompt assembly Retrieval telemetry is easy to overread if you ignore how the prompt is built. A retrieval configuration that improves recall@k can still worsen answer quality if the prompt assembler expands each chunk aggressively or inserts them in an order the model does not use effectively. Long-context evidence shows that more context is not monotonically better; relevant evidence can land in a position that the model underweights, especially in long prompts with many chunks. That makes the retrieval-to-inference join essential.

When a request has high token usage and mediocre answer quality, ask whether the retrieval stage produced too much context or whether the assembler over-expanded the context after retrieval. Count the chunks selected, the chunks rendered, and the bytes added by metadata and template scaffolding. If those values differ materially from the retrieval output, the quality problem may have been created after search finished. A traced retrieval pipeline that ignores prompt size can still misattribute the failure to retrieval because both stages happen before model generation.

Watch for model-version interactions A retrieval cohort that performs well under one model can degrade under another, even if the retrieval spans are identical. That happens because different models vary in context sensitivity, truncation behavior, and tolerance for noisy

evidence. If you swap the inference model, preserve the retrieval lineage and compare cohorts by model version rather than pooling them. Otherwise you may blame retrieval for a change that came from the model's own consumption pattern.

This is also why reranker scores should not be treated as an absolute truth across versions. A score distribution that looked healthy with one reranker or one model may not retain the same meaning after an upgrade. Use the scores as comparative signals within a fixed configuration, not as universal ground truth. The stable join key is the request lineage; the stable interpretation comes from version labels on both retrieval and inference stages.

Anti-patterns that break correlation Several habits make correlation useless even when the underlying traces are present. The first is to correlate by loose timestamps instead of request lineage. Timestamps drift under load, and concurrent traffic makes the join ambiguous. The second is to treat reranker scores as if they were comparable across model versions or candidate windows without recording the configuration that produced them. The third is to ignore retrieval fallback paths during incident review, which hides the very event that changed the downstream answer set.

A fourth anti-pattern is to attach business identifiers as ungoverned span attributes. That creates privacy risk and high-cardinality clutter at the same time. A fifth is to stop at retrieval metrics such as recall@k or NDCG and assume they predict user quality directly. They do not. They predict candidate ranking quality, which is only one factor in the final answer. If the prompt assembler drops the relevant chunk, or if the model underuses it, the retrieval metric can improve while the user experience worsens.

A useful decision table for incident response When an incident shows up as slow answers, low-quality answers, or rising cost, the first question is where the observed delta appears.

Observed delta	Likely first area to inspect	Why
Higher retrieval latency, stable prompt size	Retrieval execution	The model sees similar evidence, but search took longer.
Lower answer quality, smaller candidate set	Retrieval design or filtering	The model received less or narrower evidence.
Higher token usage, stable retrieval quality	Prompt assembly	The evidence was expanded or duplicated before inference.
Higher completion latency, stable retrieval and prompt size	Inference model	The downstream model likely changed behavior.
Quality drop with fallback activation	Retrieval execution or dependency failure	A degraded branch changed the evidence set.

The table is deliberately operational rather than theoretical. It gives you a first-pass localization rule before you dig into the full trace. The important point is that the retrieval stage by itself does not answer the business question; it only becomes useful once its outputs are joined to the model span and the request cohort.

Keep the correlation model versioned Telemetry correlation is not static. Retrieval policies change, rerankers change, prompt templates change, and inference models change. If you do not record which retrieval configuration produced a request, your historical comparisons will mix incompatible systems. Version labels on the retrieval policy, reranker model, and prompt assembly logic let you compare cohorts across deployments without treating structural changes as noise.

That versioning discipline also supports rollback. When a retrieval change improves offline ranking but degrades live answer quality, you can isolate the cohort by configuration, compare it to the previous version, and determine whether the issue came from candidate selection,

ordering, or prompt consumption. The same trace that explains a regression also becomes the audit trail that proves which stage changed.

5.4. Prompt Assembly and Context Inflation Diagnostics

Retrieval can be fast and still produce expensive answers. The hidden cost often appears after ranking finishes, when the application expands chunks, injects metadata, renders templates, and serializes the final message list for the model API. That handoff is easy to miss because no single stage looks broken. Retrieval spans report healthy latency, the model span looks ordinary, and the only visible symptom is a rising token bill or a slower-than-expected completion. Prompt assembly deserves its own diagnostic surface because it transforms a retrieved evidence set into the actual workload the model consumes.

Instrument the assembly boundary explicitly

A practical tracing pattern is to place one span around the selection of evidence and a second span around the construction of the final prompt. That boundary lets you separate search quality from context inflation. The first span should report what the retriever selected; the second should report what the model actually received. If those numbers diverge, the application layer is creating additional cost that retrieval telemetry alone cannot explain.

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

def build_prompt(query, chunks):
    with tracer.start_as_current_span("prompt.assemble") as span:
        span.set_attribute("rag.selected_chunks", len(chunks))
        span.set_attribute(
            "rag.raw_tokens",
            sum(len(c["text"].split()) for c in chunks)
        )
```

```
        rendered = render_template(query, chunks)
        span.set_attribute("rag.rendered_bytes", len(rendered))
        span.set_attribute("rag.rendered_tokens", estimate_tokens(
            rendered
        ))

        return rendered
```

That example is deliberately modest. The important point is not the token estimator itself; it is that you record both the retrieved shape and the rendered shape. If the rendered prompt is much larger than the retrieved text, the inflation came from templating, metadata, role framing, citations, or duplication. If the rendered prompt is smaller, the system compressed aggressively, and you need to understand what evidence was removed before inference began.

The handoff between retrieval and inference changes the cost model

Retrieval produces candidates. Prompt assembly turns those candidates into a serialized context window. Those are different cost domains. Retrieval latency is usually driven by search, filtering, merge, and reranking. Prompt assembly latency is usually driven by string processing, token estimation, deduplication, truncation, and API serialization. The two domains can move in opposite directions. You can make retrieval faster while making answers slower by sending more evidence into a larger prompt. You can also improve ranking while increasing cost if the assembly layer expands every chunk with metadata and safety text.

That is why prompt assembly must be observed at the same granularity as retrieval. Record selected-versus-rendered chunk counts, raw retrieved characters or tokens, retained characters or tokens after compression, duplication rate, overlap rate, citation yield, and the ratio of answer tokens to context tokens. Those values explain whether the ap-

plication is using retrieval efficiently or merely inflating the context window with extra structure.

Chunk expansion is often the first source of inflation

Chunking and prompt assembly interact in ways that are easy to underestimate. A retriever may return ten concise passages, but the assembly layer may restore surrounding overlap, add document headers, insert citations, and repeat the same metadata on every chunk. The result is a prompt that is much larger than the evidence set suggests.

Track chunk expansion as a separate step from retrieval. Record how many chunks were selected, how many were expanded, and how many remained after deduplication or compression. If your chunker restores neighboring text to maintain continuity, expose the expansion factor. If your prompt builder reattaches titles, timestamps, or source identifiers to every chunk, treat that as a measurable amplification factor rather than a harmless annotation. The more structured the template, the more likely you are to pay a hidden token tax for convenience.

The same applies to overlap. Overlap helps local coherence during retrieval, but it creates redundant text when multiple adjacent chunks are assembled together. If you do not measure overlap rate, you can end up with a prompt in which large portions of the retrieved context repeat nearly the same sentences. That kind of redundancy can improve human readability while degrading model efficiency.

Template logic creates invisible growth

Prompt templates are not neutral wrappers. A system prompt, role framing, citation preamble, safety instruction, and answer-format instruction can dominate the cost of a small retrieved context. When the retrieval set is modest, scaffolding can exceed evidence. When the retrieval set is large, repeated scaffolding can multiply across every chunk. Both cases deserve measurement.

Count the bytes and estimated tokens contributed by each template segment. Separate the static scaffold from the dynamic evidence. If a citation block is inserted for every chunk, measure the per-chunk overhead. If the template adds long policy instructions or output schemas, measure that cost once per request. That breakdown makes it obvious when a formatting decision, rather than a retrieval decision, is responsible for the jump in prompt size.

Template logic can also create brittle performance cliffs. For example, a small change in citation formatting can add enough characters to push a request over a model's effective context threshold, which then triggers truncation or fallback behavior. If you only observe the final model call, you will misread the symptom as an inference problem. The actual cause may be that the prompt crossed a serialization boundary created by your own application code.

Message framing is a distinct inflation layer

Many chat-oriented model APIs accept an ordered list of messages, not one flat string. That framing introduces its own overhead. Every message has a role label, separators, and serialization syntax, and each chunk may be wrapped in a separate message or embedded inside a shared message body. The choice matters.

If you place each retrieved chunk in its own message, you preserve provenance and make per-chunk auditing easier, but you also pay repeated framing overhead. If you concatenate all evidence into one message, you reduce framing cost but lose fine-grained structure and sometimes make truncation behavior harder to predict. Neither choice is inherently right. You should record how many messages the final prompt contains, how many carry evidence, and how much framing overhead the serializer adds.

This is one of the places where the trace should reflect the model API shape, not just the application's internal representation. A prompt

with five chunks may look small in memory and large after message serialization. If your trace records only pre-serialization text length, you miss the cost introduced by role packaging and message boundaries.

Token estimation is a control signal, not a cosmetic metric

You need token estimation before model invocation because prompt size affects latency, cost, and truncation. That estimator should run at the assembly boundary, not after the completion returns. It should tell you whether the prompt is nearing a context limit, whether a truncation policy will activate, and whether the selected evidence fits the budget you intended.

Record both estimated and provider-reported token counts when available. The difference between them often reveals estimator drift, tokenizer-version mismatch, or serialization overhead that your local estimator did not model. If a request repeatedly underestimates size, you will see surprise truncation, unpredictable cost, or poor answer quality. If it repeatedly overestimates size, you may be discarding evidence too early and paying for unnecessary compression.

Token estimation is also useful for capacity planning. An increase in average prompt tokens can raise latency even if retrieval remains unchanged, because the model now processes more input on every request. That is why you should correlate prompt token counts with answer latency and cost at the request level. A strong relationship between the two usually means the inflation occurred in assembly, not in search.

More context is not always better

A retrieval configuration can improve recall@k or NDCG and still harm answer quality if the relevant evidence is buried in the middle of a long prompt. Long-context behavior is not monotonic. The model may pay more attention to the beginning and end of the prompt than to the mid-

dle, especially when the prompt contains many retrieved chunks and a large scaffold. That means the best-ranked evidence is not always the best-used evidence.

The operational consequence is simple: you should record chunk position in the final prompt. A chunk that appears early may have a different contribution than the same chunk placed halfway through a long context. If your prompt builder always appends new evidence at the end, then a ranking policy change can alter answer quality without changing the retrieved set itself. If your assembler groups sources by document or by branch, the positional distribution can become just as important as the content distribution.

This is the point where retrieval metrics need prompt-awareness. An offline ranking improvement may look excellent until you observe that the highest-value chunk now lands in the middle of a long, noisy prompt and gets underused. The retrieval stage did its job; the assembly stage changed the effective evidence quality seen by the model.

Deduplication and compression are quality controls

Prompt inflation is not always caused by deliberate expansion. It can also come from repeated content. Hybrid retrieval branches often produce overlapping results. A reranker may select multiple passages from the same source. A document with internal chunk overlap may reintroduce the same text several times. Without deduplication, the prompt can waste tokens on near-duplicates that add little new information.

Measure duplication rate before the final prompt is emitted. If two or more chunks share substantial overlap, record how much of the rendered context is redundant. If your builder compresses content, record how many tokens or characters were removed and what evidence was lost. That data helps you decide whether compression improves efficiency or quietly strips away useful nuance.

Compression trades fidelity for cost. That trade-off is worth it when the model needs less evidence to answer correctly, but it can hurt when the answer depends on fine distinctions or exact wording. The right metric is not compressed size alone; it is answer quality per retained token. That is the ratio that tells you whether compression is doing productive work.

Citation yield exposes whether the context is being used

Prompt assembly often includes citations or source markers. Those markers are useful only if they correspond to evidence the model can plausibly use. Citation yield measures how much of the assembled context survives into answer-relevant text, whether through explicit references or through improved grounding. If you insert many citations but the answer ignores them, the assembly layer is adding structure without improving usage.

Track citation count, citation placement, and whether cited chunks were actually retained after truncation. If the prompt includes sources that were later removed or compressed beyond recognition, the citation scheme becomes decorative. If the model frequently cites sources that were not near the prompt edges or were buried deep in the evidence block, you may have a positional problem rather than a retrieval problem.

This is also where answer-token-to-context-token ratio becomes valuable. A very low ratio means the model consumed a lot of input for a small output, which can be legitimate for complex tasks but often signals context inflation. A stable ratio with rising cost may suggest that the prompt got longer without improving the answer. A falling ratio with stable quality may indicate that the assembly layer is becoming less efficient.

The operational flow should be measurable end to end

A practical prompt-assembly pipeline looks like this: retrieved chunks arrive, the assembly layer deduplicates them, restores overlap if needed, injects metadata and instructions, estimates token usage, truncates low-priority content if necessary, serializes the messages, and sends the final payload to inference. Each step can increase, preserve, or reduce context size. Each step should leave a trace or metric artifact.

That means you should record the following at minimum: raw retrieved tokens, retained tokens after deduplication, retained tokens after compression, final serialized tokens, number of evidence messages, truncation count, and the order in which chunks appear. If the assembler also reorders chunks by source type or confidence, record that too. A request with a good retrieval set can still become a poor answer if the assembly flow changes the order or removes the critical passage.

Use trace data to distinguish prompt problems from model problems

When a bad answer arrives, prompt assembly is often blamed too quickly or not blamed at all. The trace should tell you which is more likely. If the prompt is much larger than expected, the failure is probably in assembly. If the prompt size is stable but answer quality falls after a model change, the failure is probably in inference. If both prompt size and model behavior are stable, the issue may lie in retrieval quality or business context.

That distinction is easiest to make when you correlate prompt metrics with downstream outcomes. Compare slow and fast requests, good and bad answers, and low-cost and high-cost cohorts. Look for the first point at which the cohorts diverge. If divergence begins after prompt assembly, do not spend time tuning retrieval scores. If divergence begins before assembly, do not blame the model for a context it never received. The value of prompt telemetry is precisely that it turns an

invisible handoff into a diagnosable stage boundary.

Trade-offs should be explicit, not accidental

Every assembly policy makes a trade-off. Larger evidence sets improve coverage but increase latency and cost. Aggressive truncation keeps prompts short but risks dropping the key passage. More detailed metadata helps auditability but inflates every chunk. One-message prompts reduce framing overhead but obscure provenance. Per-chunk messages improve traceability but add serialization cost.

You should choose among those options based on the operational question you need to answer. If the main concern is user-facing latency, favor smaller contexts and cheaper templates. If the main concern is answer quality, preserve more evidence and record more lineage. If the main concern is privacy, reduce content capture and move sensitive detail into governed telemetry rather than prompt text. The right decision is rarely the same for every workload.

The important habit is to make the cost visible before it reaches inference. Once you can see how retrieval output becomes prompt input, you can explain why a system that looked healthy at search time still spent too many tokens, too much latency, or too much budget on the final answer.

Chapter 6

Collector-Centric Production Pipelines for AI Telemetry

Most teams first encounter the Collector as a forwarding hop, then discover too late that GenAI telemetry is where policy, privacy, cost, and scale collide. This chapter explores that collision directly, showing how Collector pipelines become the place where raw spans are turned into production-grade observability assets without forcing every service team to reimplement the same controls.

6.1. Structuring Collector Pipelines for High-Volume AI Telemetry

When a GenAI platform grows beyond a handful of services, telemetry volume stops behaving like ordinary microservice observability. Model

gateways emit short request spans, retrieval tiers emit bursty fan-out traces, tool runners emit nested child spans, and streaming inference often produces long-lived spans with late-arriving updates. If every service ships directly to every backend, you lose control over trust boundaries, backpressure, and policy enforcement. The Collector exists to absorb that complexity and turn it into an explicit pipeline design.

A useful operational model starts with a simple rule: the Collector is not a passive relay. It is a programmable telemetry execution environment made of receivers, processors, exporters, connectors, and extensions. Receivers ingest telemetry over OTLP or other protocols. Processors modify, filter, sample, or protect the data. Exporters hand off to backends. Connectors bridge signals or pipelines. Extensions provide operational support such as authentication, health checks, or storage. The production question is not whether you use a Collector, but where you place each control point and how much state you allow it to hold.

```
receivers:
  otlp:
    protocols:
      grpc:
      http:

processors:
  memory_limiter:
    check_interval: 1s
    limit_mib: 512
    spike_limit_mib: 128
  batch:
    timeout: 5s
    send_batch_size: 1024

exporters:
  otlp/gateway:
    endpoint: gateway-collector:4317
    tls:
      insecure: false

service:
  pipelines:
    traces:
      receivers: [otlp]
```

```
    processors: [memory_limiter, batch]
    exporters: [otlp/gateway]
  metrics:
    receivers: [otlp]
    processors: [memory_limiter, batch]
    exporters: [otlp/gateway]
  logs:
    receivers: [otlp]
    processors: [memory_limiter, batch]
    exporters: [otlp/gateway]
```

That configuration is intentionally small, but it already reveals the operating model. A local edge Collector protects the application from export stalls, groups telemetry into manageable batches, and forwards all three signals to a gateway layer. The edge instance is usually close to the workload and should stay mechanically simple. The gateway Collector becomes the policy-rich control point where redaction, routing, enrichment, and backend-specific shaping happen.

Pipeline anatomy and execution order

A Collector configuration is a graph of components, but each pipeline executes as a linear chain. The order matters. A receiver produces telemetry in a pipeline, processors consume it in sequence, and the final exporter writes the resulting data. If you place a processor too early, later processors may never see the original attributes. If you place it too late, the data may already have been exported or grouped in a way that makes the intended policy ineffective.

That sequencing becomes especially important for AI telemetry because the data is heterogeneous. A prompt-bearing span may need redaction before any fan-out to secondary exporters. A resource processor may add tenant and deployment metadata before routing decisions depend on it. A batch processor should usually sit late in the chain so that it groups already-normalized data rather than amplifying malformed records. The Collector does not infer intent; you encode intent by ordering the processors.

The same processor type can appear in multiple pipelines, but each pipeline receives its own instance. That isolation prevents accidental state sharing across traces, metrics, and logs, yet it also means you pay for duplicated memory and CPU. In a high-volume GenAI deployment, that detail matters. If you create three nearly identical pipelines because one signal needs an extra filter, you have also created three separate processor graphs with separate queues, timers, and buffers. The topology should reflect governance boundaries, not merely convenience.

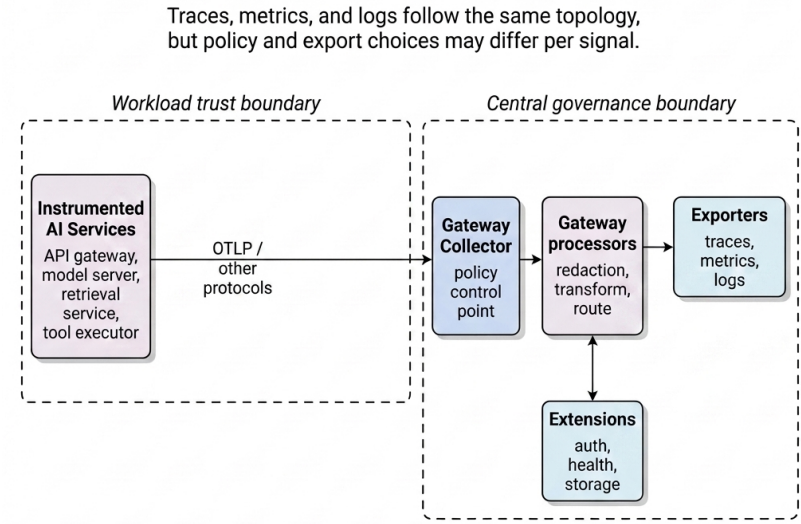
Deployment layers and trust boundaries

A single Collector layer is rarely enough once AI services become multi-tenant, cross-region, or compliance-sensitive. The most common production patterns are sidecar, node-level agent, gateway, and tiered Collector layouts. Each answers a different operational problem.

A sidecar Collector offers the tightest coupling to a service instance. It is useful when you need local buffering, local mTLS termination, or per-pod isolation for sensitive request data. A node-level agent reduces operational overhead and centralizes host-level resource limits. A gateway Collector gives you a strong governance boundary: it becomes the first central place where organization-wide policies can be enforced. A tiered design combines them, usually with edge instances near workloads and a smaller number of gateways that perform shared policy and export fan-out.

That layered model is often the right fit for GenAI. Prompt content, tool metadata, retrieval context, and model-specific attributes often need one set of handling rules at the edge and another set at the gateway. Edge collection can minimize local egress and protect the application from exporter latency. Gateway collection can enforce uniform retention class, tenant isolation, and backend selection. The trade-off is that every extra layer introduces another queue, another failure do-

main, and another place where misordered processors can alter data semantics.



The topology above emphasizes control boundaries rather than backend branding. That is deliberate. A backend can change without altering the architecture, but the Collector placement, processor order, and trust boundaries define the actual operational model. In AI systems, the gateway is often the durable decision point: it is where you decide what to retain, what to route, what to sample, and what to discard.

Receivers as ingress boundaries

Receivers define how telemetry enters the system. For most GenAI services, OTLP is the default ingress path because it gives you one transport model for traces, metrics, and logs, with both gRPC and HTTP variants available. The receiver is also where protocol conversion begins. If one language SDK emits over gRPC and another over HTTP, the Collector normalizes that difference before any later processor sees the data.

That normalization matters because AI platforms are rarely homogeneous. A model serving layer may use one language and one SDK, an agent runtime another, and an embedding service a third. If each component exports to a different backend protocol, you have multiplied your ingestion surface area for no observability gain. The receiver concentrates that complexity. It is usually best to keep receiver logic small: accept the data, authenticate if needed, enforce transport limits, and hand off to processors.

Receivers are also where you first encounter overload. Long-running spans from streaming generation may arrive with large attribute payloads or late events. Bursty retrieval traces may arrive in tight clusters. The receiver should be configured with the expectation that the workload is spiky, not smooth. In practice, that means pairing ingress with memory protection early in the pipeline and keeping the receiver itself free of business rules.

Processors as the policy engine

Processors are where the Collector becomes useful as a governance layer. A memory limiter protects the Collector from collapse when an exporter slows down. A batch processor improves throughput and reduces downstream request overhead. An attributes or resource processor can attach environment, region, service tier, or tenant metadata. A transform processor can clean up schema differences or compute derived values. A filter processor can discard telemetry that fails policy.

These are not interchangeable tools; each addresses a different control point.

For GenAI telemetry, the policy engine has to balance fidelity against scale. Token-heavy traces can be expensive to retain. Prompt and response content may be subject to deletion or masking rules. High-cardinality attributes such as user IDs, session IDs, or agent step names can make backends unusable if left unchecked. The processor chain should therefore perform the cheapest safe operations first. Add stable metadata early, redact before export fan-out, and batch only after the records are in their intended shape.

Processor order also determines what can be recovered later. If a redaction processor removes an attribute before a transform processor runs, the transform cannot use that attribute to derive a safer replacement. If a transform mutates an identifier before a routing decision, you may destroy the very key used to separate sensitive traffic from routine traffic. The right pattern is to preserve original semantics long enough to apply policy, then minimize aggressively before the export boundary.

Connectors and signal bridging

Connectors are often misunderstood because they are neither simple processors nor ordinary exporters. They bridge pipelines and sometimes signals. A connector can take telemetry from one pipeline and emit derived telemetry into another. The classic example is converting spans into metrics, but the broader design value is structural: connectors let you fan out policy results without making every downstream consumer parse the original signal.

In AI observability, connectors are useful when you want to aggregate operational indicators while preserving trace fidelity elsewhere. For example, you might derive request rate, error rate, or latency histograms from traces, then export the traces to a forensic backend and the metrics to a dashboard backend. The connector becomes a controlled

bridge between detailed and summarized telemetry. That summary path should never replace the original path; it complements it.

Connectors also create a temptation to over-abstract. If you do not understand which signal is entering and which signal is leaving, you can easily build loops, duplicate events, or lose attribution. Treat connectors as explicit dataflow edges. Draw them on paper before you configure them.

Exporters and destination shaping

Exporters are the final step in a pipeline, but they are not interchangeable plumbing. Every backend has different limits for payload size, batching behavior, retry semantics, and accepted field sets. One exporter may tolerate large trace batches; another may reject them. One may store structured attributes efficiently; another may penalize cardinality. The Collector helps you shape telemetry to each destination without requiring application changes.

That flexibility is valuable in GenAI environments because the same telemetry often needs multiple destinies. A low-retention operational backend may receive broad, sanitized traces for live debugging. A restricted forensic store may receive redacted content-bearing records with tighter access controls. A metrics backend may receive derived counters and latency histograms from a connector path. The exporter boundary is where you align telemetry shape with backend contract.

Extensions and operational support

Extensions do not move telemetry, but they make the Collector viable in production. Health checks let orchestration systems determine whether the process is ready. Authentication and TLS extensions help terminate or validate transport. Persistent storage can back certain queueing or sampling behaviors. In a high-volume AI platform, operational support is not optional, because the Collector itself becomes part

of the critical path.

The practical rule is to keep extensions narrowly scoped. They should support the pipeline, not define policy. When teams overload extensions with business logic, they blur responsibilities and make failure behavior harder to reason about. The Collector should remain transparent: data comes in, processing rules apply, and data leaves according to an explicit chain.

Why GenAI workloads stress the topology differently

Conventional service telemetry usually has short spans, modest attributes, and predictable volume. GenAI telemetry is different in three ways. First, spans can be long-lived and hierarchical, especially when agents call tools or retrieval systems inside an inference request. Second, payload-adjacent metadata is richer, with token counts, prompt classes, model identifiers, cache-hit indicators, and tool-call labels competing for space. Third, bursts are common. A retrieval fan-out or multi-agent workflow can create a trace explosion even when user traffic looks modest at the edge.

Those properties force you to think about topology before policy. A naive single-layer Collector may work during development but fail under load because it lacks isolation between ingestion, transformation, and export. A layered design gives you room to absorb burstiness near the workload while enforcing uniform policy at the gateway. It also gives you a clean place to add or remove exporters without touching instrumentation.

Operational trade-offs and failure modes

The strongest argument for centralized gateways is governance. You can apply one redaction rule, one retention policy, one routing table, and one export contract. The strongest argument against overcentralization is blast radius. If a single gateway stalls, it can delay

telemetry for many services. That is why edge buffering and memory limits matter. They protect the application from telemetry outages and give you time to shed load safely.

There is also a subtle observability trade-off: the more processing you push into the Collector, the less immediately obvious the telemetry path becomes to application owners. A bad transform rule can silently erase attributes. A misordered batch processor can delay incident visibility. A connector can create misleading derived metrics if its source spans are incomplete. These failures are usually not catastrophic, but they are operationally expensive because they distort trust in the data.

For that reason, configuration should stay declarative and testable. Keep processor chains short enough that you can reason about them during incident review. Separate concerns by pipeline where the traffic class or trust boundary differs. Use edge Collectors for buffering and transport normalization, gateways for policy and export fan-out, and connectors only where a signal conversion adds clear analytical value.

A working mental model for later pipeline design

The most useful vocabulary is also the simplest. Receivers define ingress. Processors define policy. Connectors define structural relationships between pipelines or signals. Exporters define destination contracts. Extensions define the operational envelope around the Collector itself. Once you adopt that model, the rest of the chapter becomes easier to reason about because every later decision can be located at one of those control points.

For high-volume AI telemetry, that matters more than any individual processor choice. A GenAI platform becomes governable when you can say where telemetry enters, where it is normalized, where it is protected, and where it leaves. The Collector gives you those boundaries, and the topology you choose determines whether they remain crisp under load or dissolve into an opaque forwarding path.

6.2. Implementing Redaction, Enrichment, and Token-Cost Normalization in the Pipeline

A GenAI team usually discovers the need for pipeline policy after the first incident, not before it. The spans are already flowing, the dashboards are already useful, and then someone finds prompt fragments, customer identifiers, or incompatible token fields distributed across multiple backends. At that point, instrumentation changes are too slow and too disruptive to be the only fix. The Collector becomes the place where you enforce content policy, add organizational context, and reshape token data into a stable analytic form before the telemetry reaches long-term storage.

The key is to separate three responsibilities that often travel together but must remain distinct. Redaction removes or masks data that should not persist in its original form. Enrichment adds controlled metadata that gives the telemetry organizational meaning. Normalization maps inconsistent model- or SDK-specific fields into a common shape so queries and alerts do not depend on vendor quirks. Once you treat those as separate passes, the pipeline becomes easier to reason about and much safer to operate.

Start with a policy-shaped processor chain

A practical configuration usually begins with a narrow receiver, then a memory guard, then redaction, enrichment, and normalization, followed by batching and export. You want the collector to reject unsafe content before any fan-out, because every additional branch multiplies the risk of leakage. You also want stable metadata inserted before routing or aggregation logic so that downstream rules can rely on it.

```
receivers:
  otlp:
    protocols:
      grpc:
      http:
```

```
processors:
  memory_limiter:
    check_interval: 1s
    limit_mib: 768
    spike_limit_mib: 192

  redaction:
    allow_all_keys: false
    allowed_keys:
      - service.name
      - service.version
      - deployment.environment
      - cloud.region
      - genai.model.name
      - genai.operation.name
      - genai.usage.input_tokens
      - genai.usage.output_tokens
      - genai.usage.total_tokens
    blocked_values:
      - pattern: '(?i)api[_-]?key'
      - pattern: '(?i)secret'
      - pattern: '(?i)password'

  attributes/enrich:
    actions:
      - key: tenant.class
        value: enterprise
        action: insert
      - key: compliance.domain
        value: regulated
        action: insert
      - key: deployment.ring
        value: prod
        action: upsert

  transform/tokens:
    trace_statements:
      - context: span
        statements:
          - set(attributes["genai.usage.total_tokens"],
                attributes["genai.usage.input_tokens"] +
                attributes["genai.usage.output_tokens"])
            where attributes["genai.usage.total_tokens"] == nil

  batch:
    timeout: 5s
    send_batch_size: 1024
```

```
exporters:
  otlp/gateway:
    endpoint: gateway-collector:4317
    tls:
      insecure: false

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors:
        [memory_limiter, redaction, attributes/enrich,
         transform/tokens, batch]
      exporters: [otlp/gateway]
```

That example is intentionally conservative. It uses an allow-list posture for the redaction processor, inserts only stable organizational fields, and computes a derived token total only when the source does not already provide one. This is the shape you want for production policy: deterministic, inspectable, and easy to test.

Redaction is a gate, not a cosmetic filter

The most important property of the redaction processor is that it can fail closed. When you do not allow all keys, attributes outside the allow-list are removed before blocked-value masking runs. That ordering matters. A sensitive attribute may disappear entirely instead of being masked, which is exactly what you want for many retention policies, but it also affects auditability. If downstream teams expect to see “masked prompt” markers or placeholder values, they may be surprised when the key is absent instead.

That behavior leads to a useful design rule: choose between removal and masking intentionally. Removal minimizes the retained surface area. Masking preserves the existence of the field, which can help with troubleshooting or schema validation. If a field is known to be risky and unnecessary, remove it. If the field is important for query shape or incident triage, mask it only after you have confirmed that the value

itself can safely remain in the record.

The allow-list should contain only attributes that are both operationally useful and safe to retain. Stable model identifiers, deployment metadata, and aggregate token counts are usually fine. Prompt text, raw tool arguments, user identifiers, and free-form retrieval snippets usually are not. Do not let the presence of a redaction processor tempt you into keeping more than you need.

Use ignored keys sparingly

Most redaction implementations provide an escape hatch for ignored keys or patterns. That option is useful when you know an attribute is safe and you want to bypass blocking logic for a small set of fields. It is also dangerous because it can quietly exempt data from the very controls meant to protect it. Treat ignored keys as a last-mile exception mechanism, not as a convenience for broad policy relaxation.

A good rule is to reserve ignored patterns for structurally safe attributes whose content is already bounded, normalized, and reviewed. Examples include a vendor-generated model label or a known-safe deployment tag. Do not use ignored keys to rescue fields that are merely inconvenient to redact, such as JSON blobs or user-controlled labels. If a field is hard to classify, it belongs in a separate review path, not in an exception list.

Understand what the redaction processor reports

Redaction processors often emit audit attributes only when they actually perform an action. Zero-value counters or empty audit markers are commonly omitted. That means absence of an audit field is not evidence that nothing happened. It may simply mean the processor had no matching keys or values to remove.

For operational monitoring, that distinction matters. If you are validating the pipeline, inspect the preserved records themselves and, where

available, the processor's debug logs or metrics rather than assuming a missing audit key means a clean pass. This is especially important in AI systems where a single prompt field may be nested, repeated, or serialized differently across services. A redaction rule that works on one shape of span may do nothing on another, and the absence of an audit marker will not tell you which case occurred.

Enrichment should add stable organizational context

Enrichment is not a dumping ground for every context source you can reach. It is the controlled insertion of durable metadata that makes the telemetry easier to route, query, and govern. Good enrichment fields answer questions like: Which tenant class produced this span? Which region handled it? Which deployment ring was active? Which compliance domain applies? Which internal product surface emitted the request?

Use the attributes processor for straightforward enrichment because it gives you deterministic insert, update, upsert, delete, and hash actions. When the value is known and stable, insert it directly. When the field may already exist in some pipelines, upsert it. When you need a pseudonymous key rather than the original value, hash it. That simplicity is a strength: you can inspect the configuration and predict the outcome without evaluating a more complex transform language.

```
processors:
  attributes/enrich:
    include:
      match_type: strict
      services: [inference-gateway, rag-service]
    actions:
      - key: tenant.class
        value: enterprise
        action: insert
      - key: compliance.domain
        value: regulated
        action: insert
      - key: user.hash
        action: hash
      - key: deployment.ring
```

```
        value: prod
        action: upsert
```

Keep enrichment close to resource-level facts when possible. Resource attributes such as environment, region, cluster, or service name are stable and low-cardinality, which makes them ideal for downstream aggregation. Avoid injecting volatile request-scoped values unless they are required for policy or incident response. A field that changes on every span can explode cardinality and make the backend more expensive without adding meaningful operational value.

Normalize token dimensions without pretending to price

Token accounting is where many GenAI telemetry pipelines drift into confusion. Different SDKs, model providers, and wrappers may name the same concept differently. One source emits “prompt” and “completion” tokens. Another uses “input” and “output”. A third includes “cached” tokens or a total count only when the provider returns it. The Collector can unify those names, but it should not become the source of billing truth unless pricing is guaranteed to remain external and stable.

Normalize the dimensions you need for querying, alerting, and cost estimation. Keep the derivation simple. Map provider-specific fields into a standard set such as input tokens, output tokens, cached tokens, and total tokens. When total tokens are missing, compute them from the components if that is semantically valid for the model and SDK involved. When cached tokens exist, preserve them separately because they often influence cost or latency analysis. Do not collapse all token categories into a single number unless you are sure the backend will never need the breakdown.

```
processors:
  transform/tokens:
    trace_statements:
      - context: span
        statements:
          - set(attributes["genai.usage.input_tokens"],
```

```
            attributes["llm.prompt_tokens"])
          where attributes["genai.usage.input_tokens"] == nil
        - set(attributes["genai.usage.output_tokens"],
            attributes["llm.completion_tokens"])
          where attributes["genai.usage.output_tokens"] == nil
        - set(attributes["genai.usage.cached_tokens"],
            attributes["llm.cached_tokens"])
          where attributes["genai.usage.cached_tokens"] == nil
        - set(attributes["genai.usage.total_tokens"],
            attributes["genai.usage.input_tokens"] +
            attributes["genai.usage.output_tokens"])
          where attributes["genai.usage.total_tokens"] == nil
```

That pattern makes the Collector a schema harmonizer, not a billing engine. If price depends on model family, region, customer contract, or promotion, keep the pricing catalog outside the pipeline. Feed the normalized usage dimensions into a downstream system that owns the pricing rules. The Collector should tell you how many tokens were used and which model handled them; it should not hard-code a changing commercial policy.

Preserve provenance when you mutate fields

Whenever you rewrite a field, ask whether the original value still matters. If a transform maps provider-specific token names to a common schema, the original names may be useful during migration or incident review. If you replace a raw customer identifier with a hash, it may be valuable to retain a separate marker that says which hashing policy was used and when it was applied.

This is where the boundary between content minimization and content mutation becomes practical. Minimization removes data that should not persist. Mutation changes a retained value into another form. The two are not equivalent. If you delete a prompt, you lose it completely. If you hash an identifier, you retain a stable join key but lose reversibility. Decide which property you need before you configure the processor. A common mistake is to overwrite a field in place and then later discover that no one can tell which version of the field was original and which

version was derived.

A simple provenance strategy is to keep derived fields under the normalized namespace and leave the source fields either removed or explicitly renamed. For example, map provider-specific usage fields into "genai.usage.*" and retain a separate "genai.normalization.source" attribute only if you need to track migration provenance. That gives you enough structure to debug without keeping more sensitive data than necessary.

Choose the cheapest safe transformation

The cost of a pipeline is not only CPU. It is also complexity, memory pressure, and the probability that a misconfigured rule silently changes meaning. An attributes processor is usually cheaper and easier to validate than a full transform rule. Use it for direct enrichment, deterministic hashing, and obvious field renames. Reserve transform logic for cases that need conditional computation, arithmetic, or cross-field normalization.

For example, computing a total token count from input and output tokens is a genuine transform because it depends on multiple fields and a fallback condition. In contrast, inserting "deployment.ring=prod" is a plain attribute operation. If you push too much simple work into transform statements, you make the pipeline harder to read and easier to break. If you try to use only the attributes processor, you will quickly run out of expressive power for arithmetic or conditional normalization. The right split is usually obvious once you frame the requirement in terms of deterministic data movement rather than generalized scripting.

Validate fail-closed behavior explicitly

Because disallowed keys are removed before masked-value checks, validation must inspect the output shape rather than assume a value was

masked. This is easiest to test with a few representative spans: one containing a prompt field, one containing a safe model field, and one containing a nested customer identifier. After the pipeline runs, verify that the prompt field is absent when policy requires removal, that the model field survives with the expected value, and that any hashed or renamed identifiers preserve the downstream join semantics you intended.

A quick manual validation path is often enough to catch the most dangerous errors.

```
otelcol-contrib --config=collector.yaml
```

```
2026-05-14T10:00:00Z INFO   Starting otelcol-contrib
2026-05-14T10:00:01Z INFO   Memory limiter enabled
2026-05-14T10:00:01Z INFO   Processor redaction active
2026-05-14T10:00:01Z INFO   Processor transform/tokens active
2026-05-14T10:00:02Z INFO   Exporting 1024 spans to gateway-collector:4317
```

The exact log format will vary, but the operational question does not: do the output records show only the attributes you intended to preserve, and do the derived token fields match the source spans? If the answer is no, inspect the processor order first. In many cases, the bug is not in the rule itself but in the stage at which it runs.

Avoid the common failure modes

The most damaging anti-pattern is regex-only redaction for structured payloads. If prompts or tool arguments are serialized JSON, a regex can miss nested values or falsely match safe text. Prefer field-aware removal when the collector can parse the structure, and only fall back to pattern matching for truly flat attributes.

The second failure mode is volatile enrichment. If you attach request-specific labels that change on every call, you will inflate cardinality and make the backend harder to query. Keep enrichment stable unless the field is explicitly needed for routing or incident review.

The third failure mode is silent overwriting. If you mutate a field without recording the normalization policy, later users cannot tell whether they are looking at source data or derived data. That problem is especially painful for token and cost fields, where a number can be mathematically valid yet semantically misleading.

The fourth failure mode is treating Collector-derived cost estimates as authoritative accounting. That is usually wrong because pricing depends on external catalogs, discounts, region-specific rules, or negotiated contracts. The Collector can normalize usage. It should not become your financial system of record.

Use the pipeline as an enforcement layer, not a rewrite system

The Collector works best when you use it for uniform, reversible, and auditable policy decisions. Redaction enforces retention and privacy rules. Enrichment adds stable organizational context. Normalization aligns token dimensions across vendors and SDKs. Together they make raw GenAI telemetry usable without forcing every application team to reimplement the same policy logic.

That division of labor keeps instrumentation focused on semantic truth and keeps the Collector focused on mechanical enforcement. Once you preserve that boundary, the telemetry pipeline becomes easier to operate under load, easier to audit during incidents, and easier to evolve as models, vendors, and naming conventions change.

6.3. Routing and Batching AI Telemetry for Mixed-Sensitivity Workloads

As soon as GenAI telemetry becomes useful, it becomes uneven. A small fraction of traces carry prompt fragments or tool arguments that

you cannot retain broadly, a different fraction belongs to high-value incidents that you want to keep longer, and the majority are routine requests that matter only for latency, error rate, and cost trend analysis. If you send all of that to one destination with one retention rule and one batching policy, you either overspend, under-protect, or drown the signals you actually need. The Collector gives you the mechanics to separate those classes without changing application code.

The routing problem is not just about privacy. It is about controlling where different telemetry classes land, how quickly they arrive, and how much pressure they place on the pipeline. In practice, you are deciding which traces get broad export, which get restricted retention, which get duplicated for incident analysis, and which get dropped early because they add noise but no operational value. Batching sits inside that same decision space because it changes memory use, flush latency, and retry behavior. You want to batch only after you have made the major routing and filtering decisions, or you end up paying to package data that will be discarded or re-sent elsewhere.

Define telemetry classes before you define routes

Routing works best when you classify traffic explicitly. In AI systems, the useful classes are usually routine operational spans, high-sensitivity content-bearing spans, high-cardinality experimental traffic, and premium forensic traces. Routine spans are the everyday request records that support SLOs, error analysis, and latency tracking. High-sensitivity spans include prompt-bearing or tool-bearing records that require restricted handling. Experimental traffic often comes from tests, canaries, or new prompt flows with unstable attributes. Forensic traces are the small subset you want to preserve with maximal detail for incident review.

The class label can come from several places. You can use resource attributes when the distinction is deployment-wide, such as environment

or tenant tier. You can promote baggage-derived values upstream when a request has already been tagged by trusted application code. You can route on span names, model family, or an explicit sensitivity attribute added by the instrumentation layer. The important point is that the Collector should not infer the class from brittle heuristics if a stable signal already exists.

```
receivers:
  otlp:
    protocols:
      grpc:
      http:

processors:
  memory_limiter:
    check_interval: 1s
    limit_mib: 1024
    spike_limit_mib: 256

  redaction:
    allow_all_keys: false
    allowed_keys:
      - service.name
      - service.version
      - deployment.environment
      - tenant.class
      - genai.model.name
      - genai.sensitivity
      - genai.usage.total_tokens

  routing:
    from_attribute: genai.sensitivity
    table:
      - value: public
        exporters: [otlp/public]
      - value: restricted
        exporters: [otlp/restricted]
      - value: forensic
        exporters: [otlp/restricted, otlp/archive]

  batch:
    timeout: 5s
    send_batch_size: 1024

exporters:
  otlp/public:
```

```
    endpoint: public-backend:4317
    tls:
      insecure: false
  otlp/restricted:
    endpoint: restricted-backend:4317
    tls:
      insecure: false
  otlp/archive:
    endpoint: archive-backend:4317
    tls:
      insecure: false

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors:
        [memory_limiter, redaction, routing, batch]
      exporters:
        [otlp/public, otlp/restricted, otlp/archive]
```

That example is deliberately simple. It shows the design pattern you want: classify early, redact before any export fan-out, route by a stable attribute, and batch only after the routing decision has been made. The actual routing component may vary by Collector distribution, but the reasoning does not.

Split by pipeline when policy diverges materially

A single pipeline can fan out to multiple exporters, but there is a point where one pipeline becomes too blunt. If sensitive traces need stricter redaction, shorter retention, and more conservative batching than routine traces, split them into separate pipelines. Because processors are instantiated per pipeline, you can apply stronger controls to one path without paying the same operational cost on every path.

That isolation is valuable in mixed-sensitivity environments. A public operational path can preserve enough detail for dashboards and live debugging. A restricted path can retain more context but send it only to a backend with tighter access controls. A forensic path can duplicate

a small set of traces to a deeper store for incident response. You get different retention classes without duplicating instrumentation logic.

The trade-off is complexity. More pipelines mean more configuration surface, more exporter instances, and more failure modes to test. Use separate pipelines only when the classes differ in a way that affects policy, retention, or backend contract. If the only difference is a label or dashboard view, one normalized pipeline with downstream filters is usually simpler.

Fan out from one normalized path when the data shape is uniform

If telemetry has already been normalized and redacted to a common shape, one pipeline can safely fan out to multiple exporters. That is often the best choice for routine spans and low-risk operational metrics. You avoid duplicated processing, keep the configuration smaller, and preserve a single place where policy changes are applied.

Fan-out works best when each destination expects the same sanitized record but uses it differently. For example, one backend may serve low-latency dashboards, another may provide long-retention search, and a third may store compact aggregates. If the exporter contracts are compatible, you can split after redaction and normalization and let the destinations diverge downstream. If they are not compatible, split earlier.

Avoid the temptation to fan out before redaction. Once a trace reaches more than one exporter path, every branch becomes part of your trust boundary. The safest order is still: classify, redact, normalize, then fan out.

Batch late unless you need backpressure earlier

Batching improves throughput because it reduces exporter overhead and amortizes network cost. It also increases memory use and can de-

lay visibility because telemetry waits in buffers before being sent. For AI workloads, that trade-off is sharper than usual because spans can be large and uneven. Prompt-bearing spans, retrieval-heavy traces, and tool-rich agent spans can produce batches with substantial attribute payloads.

You usually want batching late in the pipeline, after major drop and routing decisions. That way, the Collector does not spend effort batching records that will be discarded or diverted. Late batching also keeps the batch content aligned with the final export destination, which matters when one backend tolerates larger payloads and another prefers smaller ones.

There are cases where earlier batching makes sense. If the Collector is under severe export pressure and you need to relieve network churn before routing, an early batch can stabilize throughput. But early batching should be an exception, not the default. It can obscure latency spikes and make routing decisions harder to inspect because the record has already been grouped with others that may follow a different policy path.

Treat batching as a latency control, not just a throughput trick

Batching affects more than export efficiency. It also affects how quickly an incident appears in a backend. If you set the timeout too high, you delay the first useful record from a failing request. If you set the batch size too low, you increase exporter churn and waste bandwidth. The right setting depends on the signal class.

For routine traffic, you can often tolerate a slightly larger batch and a modest timeout because the main goal is efficient ingestion. For forensic traffic, keep the timeout shorter so that critical traces appear quickly. For high-cardinality experimental traffic, a smaller batch often reduces the blast radius of bursts and makes retry behavior easier

to reason about. The operational question is not whether batching is good; it is which latency you are willing to trade for which efficiency gain.

Large GenAI spans also create a subtle risk: a single oversized payload can trigger retries or backend rejections, especially when the exporter has strict request limits. If you know some requests carry unusually large attributes, keep the batch size conservative and rely on routing to send those spans to a backend that can absorb them.

Use tail sampling as a routing-adjacent control

Tail sampling is not a privacy boundary. Sensitive data must already be removed before any branch that could export it. What tail sampling gives you is selective retention of whole traces based on post-hoc properties such as latency, status code, span count, or trace attributes. That makes it a routing-adjacent control: it decides which traces are worth keeping, but it does not decide whether the trace was safe to keep in the first place.

This distinction matters in mixed-sensitivity workloads because a trace may contain both harmless operational metadata and a prompt fragment. You cannot depend on tail sampling to hide the fragment. You must redact first, then sample. Once you respect that ordering, tail sampling becomes useful for keeping rare high-value traces while discarding the bulk of ordinary traffic.

The Collector-side complication is late spans. Tail-sampling processors maintain decision state while they wait for traces to complete. In GenAI systems, some spans arrive late because tool execution, retrieval, or streaming generation stretches the trace over time. If one branch of the pipeline has already finalized a decision and another assumes full-trace completeness, you can get inconsistent behavior. The safest design is to keep tail sampling after all processors that require original context and to ensure that all spans for a trace reach the same

collector instance.

Preserve trace affinity when decisions depend on the full trace

Tail sampling only works if the Collector sees enough of the trace to make a sound decision. That means trace affinity matters. If spans for the same trace are split across collectors without a shared decision path, one collector may drop a trace that another collector would have retained. In a mixed-sensitivity environment, that inconsistency is more than an operational nuisance; it can distort forensic analysis.

You can reduce the risk by keeping trace-local traffic on the same collector instance, using a tiered design that routes all spans for a trace through the same gateway, or limiting tail sampling to classes that do not depend on extremely late arrivals. If you cannot guarantee affinity, prefer simpler probabilistic or attribute-based routing rules that do not require full-trace reconstruction.

Separate retention classes from routing classes

A route sends telemetry to a destination. A retention class determines how long that destination should keep it and who can query it. You want those two concerns related but not merged. A trace can be routed to a restricted backend, yet the backend may still have multiple retention tiers. Likewise, a public operational stream may be routed broadly but contain only already-sanitized summaries.

This separation lets you design policies that match business needs. For example, routine request traces may go to a standard observability backend with short retention. Restricted traces may go to a limited-access store with a longer window for incident analysis. Forensic traces may go to an archive backend that is queried rarely but preserved longer. The Collector makes the routing decision; the backend or adjacent storage policy makes the retention decision.

If you collapse the two, you lose flexibility. A retention change then requires a routing change, or vice versa. Keep them separate so you can revise one without destabilizing the other.

Use per-tenant routing only when the tenant signal is stable

Per-tenant routing is attractive in multi-tenant AI platforms because it makes cost attribution and data isolation obvious. It also raises cardinality and operability costs quickly if the tenant label is unstable, derived too late, or overloaded with business meaning. Route only on tenant markers that are durable and trusted. Do not route on free-form user identifiers or high-cardinality request labels unless you have already bounded them upstream.

Stable tenant routing is most useful when different tenants have different egress destinations, different retention contracts, or different compliance constraints. In that case, the Collector can enforce the boundary mechanically. If the only reason to route by tenant is dashboard convenience, a downstream filter is usually simpler and less risky.

Watch for exporter-specific failure behavior

Different exporters do not fail the same way. Some reject oversized batches quickly. Some retry aggressively. Some tolerate partial outages and queue internally. When you mix routing and batching, those differences become visible under load. A backend that accepts large payloads may look healthy while a stricter backend starts dropping or delaying data from the same pipeline.

You need to size batches with exporter behavior in mind. If one route goes to a low-latency operational backend and another to a slower archive backend, do not assume one batch policy suits both. Either separate the pipelines or make the batch settings conservative enough to satisfy the stricter destination. The wrong assumption here is expensive because it usually shows up only during traffic spikes, when large

GenAI traces arrive together.

Prefer dropping early over exporting broadly and filtering later

If a class of telemetry is not useful outside a small trust boundary, do not ship it everywhere and rely on downstream access control to hide it. Filtering later still consumes network, storage, and operational capacity. Early dropping or early diversion is cheaper and safer. That principle is especially important for prompt-bearing traces and high-volume experimental traffic.

Early dropping is also easier to test. You can inspect the output of the redaction and routing chain and confirm that unsafe data never leaves the restricted path. A downstream filter cannot give you the same assurance because the unsafe data has already crossed a broader boundary. The Collector should be the point where policy becomes irreversible.

Keep the routing tree readable

The biggest practical failure mode is an opaque pipeline that tries to do everything at once. If one configuration block mixes redaction, sensitivity tagging, tenant routing, tail sampling, exporter fan-out, and batch tuning, nobody can reason about it during an incident. The Collector should make policy explicit. Split by class when the policies differ, keep routing conditions simple and stable, and batch late unless you have a clear reason not to.

For mixed-sensitivity GenAI workloads, the design goal is not maximal cleverness. It is predictable data movement. You want to know, for any given trace, which path it will take, where it will be retained, how quickly it will arrive, and what parts of it survive the trip. Once you can answer those questions from the pipeline itself, the Collector becomes a reliable control plane rather than just another hop in the telemetry

path.

6.4. Choosing Between Collector Policy and In-Application Instrumentation Logic

Once you have a Collector that can redact, enrich, route, sample, and reshape telemetry, the next mistake is easy to make: you start treating the pipeline as if it can recover every observability decision after the fact. That usually fails in GenAI systems because some telemetry meaning exists only at the moment the application creates it. The Collector can enforce policy and standardize shape, but it cannot reconstruct lost intent, causal order, or model-specific semantics that never left the process in the first place.

The boundary matters because different kinds of observability work live at different layers. Some concerns are cross-cutting and uniform, so the Collector should own them. Others depend on request-local context or application intent, so the instrumentation must own them. If you place the wrong responsibility in the wrong layer, you get brittle transforms, ambiguous telemetry, or a pipeline that silently hides defects in the code that emitted the data.

Lead with a simple boundary rule

A practical rule is to ask whether the decision requires business or request intent. If it does, instrument it in code. If it is mechanically enforceable across teams, centralize it in the Collector. That rule is blunt, but it works because it separates semantic authority from operational enforcement.

You can see the distinction in a compact configuration. The Collector is a good place for retention-class routing and uniform redaction, but not for inventing model operation names or reconstructing tool-call

semantics after the span has been emitted.

```
processors:
  attributes/enrich:
    actions:
      - key: deployment.ring
        value: prod
        action: insert
      - key: compliance.domain
        value: regulated
        action: insert

  redaction:
    allow_all_keys: false
    allowed_keys:
      - service.name
      - service.version
      - genai.operation.name
      - genai.model.name
      - genai.usage.total_tokens

  routing:
    from_attribute: genai.sensitivity
    table:
      - value: public
        exporters: [otlp/public]
      - value: restricted
        exporters: [otlp/restricted]
```

That configuration is appropriate because it applies the same policy to every service that emits into the pipeline. It does not depend on local prompt structure, retry intent, or the exact order of tool invocations. Those details belong in the application.

Keep causal structure in the application

Span creation, parent-child relationships, and precise start and end boundaries belong in code. So do retry intent, error semantics, and any attribute whose meaning depends on what the application was trying to do at the moment the request was handled. If you wait until the Collector to invent those relationships, you are guessing from partial evidence.

GenAI telemetry makes this especially important. A model invocation span might contain nested tool calls, retrieval steps, streaming chunks, or fallback attempts. The application knows which call initiated which child span, which retry was user-visible, and which failure is part of normal control flow. The Collector does not. It can filter or route the spans, but it cannot infer the precise decision tree that produced them.

That is why model-specific operation names also belong in instrumentation. The GenAI semantic conventions encourage standardized names, but the application is the right place to emit them correctly because it understands which operation was actually performed. If you flatten that meaning downstream, you risk turning a semantically rich trace into a generic one that looks consistent but answers the wrong questions.

Use the Collector for uniform policies, not for semantic invention

Collector policy is strongest when the rule is cross-cutting, uniform, and mechanically testable. Redaction is the classic case. So is organization-wide enrichment, retention-class routing, schema cleanup, and export fan-out. These are the kinds of rules you want to change centrally without forcing every service team to redeploy.

The attributes processor is usually the safer choice for simple enrichment because its action set is narrow and easy to audit. You can insert a deployment ring, update a compliance tag, hash a stable identifier, or delete an obsolete attribute with very little ambiguity. The transform processor is more expressive, which is useful, but that expressiveness is also where the risk begins. OTTL can centralize normalization and conditional logic, but it can also become a substitute for domain-correct instrumentation if you let it absorb too much meaning.

The practical test is whether the rule would still be valid if every language team implemented it independently. If the answer is yes, the

Collector is a strong candidate. If the answer depends on the exact request flow or model behavior, keep the rule in the application.

Prefer application ownership for token provenance

Token accounting is a good example of a boundary that looks simple but is not. The Collector can normalize token dimensions, rename provider-specific fields, and compute derived totals when the inputs are stable. It cannot, by itself, know whether a token count came from the final model response, an intermediate cached path, a retried request, or a vendor-specific wrapper that already adjusted the numbers.

That provenance matters. If you want trustworthy analytics, the application should emit the original accounting fields and the Collector should only harmonize their names and structure. In other words, the app owns the semantics of the count, and the Collector owns the shape. If you reverse that order, you invite subtle errors where a mathematically correct number is semantically wrong.

A useful division is this: let the application emit prompt, completion, cached, and total token usage with any provider-specific nuance intact. Let the Collector rename those fields into a common schema and drop fields that should not persist. Do not ask the Collector to rediscover how the number was produced.

Handle privacy tags carefully

Privacy classification often looks like a Collector problem because it affects retention and access control. In reality, it is often shared. The application should emit the most authoritative privacy signal it can, because it knows the request context, user action, and content source. The Collector can validate, enforce, or override that signal according to policy.

That shared model is safer than trying to infer privacy entirely downstream. A prompt segment may be sensitive because of user content,

or because a tool returned data from a restricted source, or because a retrieval hit included customer records. The application sees those distinctions first. The Collector can enforce the final policy, but it should not be the only place where privacy meaning appears.

A good pattern is to emit a sensitivity tag in code, then have the Collector treat that tag as an input to redaction and routing. If the application and pipeline disagree, decide explicitly which layer is authoritative and which layer validates. Ambiguity here leads to false confidence, especially when a record looks sanitized but still contains meaningful context in another field.

Treat schema migration differently from semantic migration

Schema cleanup is a Collector-strength task when the change is syntactic. If one SDK emits "llm.prompt_tokens" and another emits "genai.usage.input_tokens," a transform can normalize the field names. That is a schema problem, not a semantic problem. The values already mean the same thing.

Semantic migration is different. If a field changes meaning across SDK versions, model vendors, or request flows, you should fix the emitter, not paper over the difference downstream. A Collector transform can hide the inconsistency long enough for dashboards to keep working, but it also postpones the real correction. You eventually end up with a pipeline full of compatibility hacks that no one trusts.

The decision signal is whether the field still means the same thing after the rename. If yes, transform it centrally. If no, revise the instrumentation and update the semantic contract. The Collector is a good place to bridge versions temporarily, but not to freeze an incorrect abstraction into the telemetry path.

Avoid reconstructing request-local intent downstream

There are several observability details the Collector cannot reliably re-

cover. Exact prompt segmentation is one of them. Tool-call semantics are another. Retry intent is a third. These are all request-local decisions whose meaning depends on where the application drew a boundary or took an action.

For example, a model call may contain multiple prompt fragments assembled from system instructions, user input, retrieval context, and tool output. The application knows which fragment came from which source. If you emit only a flattened prompt string and hope the Collector can later separate it into semantic parts, you have already lost information. The same is true for tool invocations: a downstream transform can preserve the presence of a tool call, but it cannot recreate the causal reason the tool was invoked.

The operational consequence is straightforward. Put the semantic split points in code. Emit spans and attributes that already reflect the structure you want to analyze. Then let the Collector enforce policy and standardize the result. If you do not preserve request-local intent at the source, you will eventually overfit the pipeline to one application and break another.

Use a decision matrix when the boundary is ambiguous

Some responsibilities are not obviously one-layer or the other. Those cases deserve a small decision matrix rather than a rule of thumb. Ask four questions: who knows the truth, who needs to react fastest, who must change independently, and what is the cost of getting it wrong?

If the application knows the truth and the cost of error is high, keep the logic in code. If the policy is cross-cutting and the cost of change is high, move it to the Collector. If the pipeline and the application both need to participate, define one as authoritative and the other as validating. That last category is common in GenAI systems where the app emits a sensitivity tag and the Collector enforces organization policy from that tag.

A compact way to reason about this is to distinguish semantic authority from operational control. The application often owns authority. The Collector often owns control. You want those roles aligned but not collapsed into one another.

Responsibility	Best layer	Why
Span parentage and timing	Application	Requires request-local intent
Prompt redaction	Collector	Uniform, enforceable policy
Token field normalization	Collector	Schema cleanup across SDKs
Token provenance	Application	Must reflect true accounting source
Retry intent	Application	Depends on control flow
Tenant enrichment	Collector or app	Use the stable source of truth
Model-specific operation names	Application	Semantic correctness at emission
Retention-class routing	Collector	Central policy enforcement

That table is intentionally asymmetric. Some rows are absolute, others depend on which layer owns the reliable source of truth. That is normal. The goal is not to centralize everything; the goal is to centralize the parts that benefit from uniform enforcement without destroying semantic precision.

Prefer Collector validation over Collector invention

When both layers must know about a rule, let the application emit the authoritative value and let the Collector validate or normalize it. That pattern keeps the pipeline honest. It also reduces the chance that a transform silently compensates for a bug in the instrumentation.

A concrete example is semantic-convention migration. If a language SDK has not yet adopted a new field name, the Collector can bridge old and new names for downstream consumers. But that bridge should be temporary and visible. Once the application is updated, the transform should shrink or disappear. If it remains forever, it becomes technical debt hidden inside the observability stack.

The same logic applies to privacy tags. If the application says a request is restricted, the Collector can enforce stricter routing and shorter retention. If the pipeline later discovers content that contradicts the tag, it should fail closed. Validation is easier to reason about than reconstruction because it preserves a single source of truth.

Watch for the two main failure modes

The first failure mode is over-centralization. Teams move business logic into the Collector because it is easier to change centrally, then they discover that the pipeline has become a disguised application layer. At that point, debugging gets harder because important decisions happen outside the code path that generated the request. You also make the pipeline brittle because it now depends on assumptions that only the application can guarantee.

The second failure mode is under-centralization. Teams leave every policy in application code, so each service reimplements the same redaction, enrichment, and schema cleanup rules. That creates drift, inconsistent retention, and a large review burden whenever policy changes. It also makes compliance harder because you no longer have a single place where policy is enforced.

The best boundary avoids both extremes. Keep semantic truth close to the request, where only the application can know it. Keep cross-cutting enforcement in the Collector, where you can govern it uniformly.

Use the Collector as a policy engine, not a truth engine

That distinction is the core of the control boundary. The Collector should decide what to keep, where to send it, and how to reshape it for downstream systems. It should not pretend to know what the request meant when the application issued it. The application should emit the semantic facts that only it can know, including exact operation names, causal structure, retry intent, and token provenance.

Once you keep those roles separate, the rest of the chapter becomes easier to apply. Collector rules stay auditable because they are mechanical and centralized. Application instrumentation stays semantically precise because it owns the meaning of the trace. The pipeline then becomes a place where policy is enforced cleanly, not a place where meaning is guessed after the fact.

Chapter 7

Privacy, Scale, and Cross-Signal Operational Debugging

The final challenge in GenAI observability is not generating more telemetry, but deciding what must be seen, what must never be stored, and how to preserve enough structure for fast diagnosis under production pressure. This chapter turns that tension into a method: constrain sensitive payload capture, budget telemetry volume intentionally, correlate signals into a repeatable triage flow, and design queries that survive schema change instead of breaking during the next upgrade.

7.1. Capturing Prompts and Responses Without Creating a Data Liability

When a GenAI system finally exposes enough of its prompt and response path to make hallucinations, jailbreaks, and retrieval failures debuggable, the observability stack often inherits the very data the application was supposed to protect. A trace that contains the full customer prompt, intermediate tool arguments, retrieved documents, and model output is diagnostically rich, but it is also a repository of regulated content, secrets, and internal business material. The operational task is not to choose between visibility and safety; it is to decide which representation of content belongs in telemetry at all, and under what controls. OpenTelemetry's GenAI semantic conventions explicitly allow content capture, but they frame it as appropriate only when volume is manageable and privacy or storage obligations are satisfied. That makes metadata-first capture the default and raw content the exception. [**?**, **?**]

A practical pattern starts with a narrow capture policy at the application boundary. You can record token counts, model identifiers, finish reasons, request and response hashes, and selected structured fields while suppressing the full payload unless a request matches a predefined diagnostic class. A minimal OpenTelemetry span in Python illustrates the shape of that policy:

```
from opentelemetry import trace

tracer = trace.get_tracer(__name__)

with tracer.start_as_current_span("genai.infer") as span:
    span.set_attribute("genai.request.model", "gpt-4.1")
    span.set_attribute("genai.usage.input_tokens", 842)
    span.set_attribute("genai.usage.output_tokens", 211)
    span.set_attribute("genai.response.finish_reason", "stop")

    if should_capture_content(user_id, route):
        span.set_attribute("genai.prompt.preview", prompt[:256])
```

```
        span.set_attribute("genai.response.preview", output[:256])
    else:
        span.set_attribute("genai.prompt.sha256", prompt_hash)
        span.set_attribute("genai.response.sha256", output_hash)
```

That pattern is deliberately asymmetric. The application decides whether to emit content, but it does so with bounded previews or opaque references rather than defaulting to full text. The span still carries enough structure to explain cost, truncation, and route behavior, and the raw material remains outside telemetry unless a specific policy allows it. The GenAI semantic conventions include fields for token usage, request and response data, and optional content capture, but the documentation also warns that content should be recorded only when the surrounding storage and privacy posture can support it. [**?**, **?**]

Choose the content form before you choose the storage path. The most common mistake is to treat all GenAI content as if it were a single artifact. In practice, four representations have very different risk profiles. Full-payload buffering preserves exact reproducibility and is useful for severe jailbreak analysis, audit review, and prompt assembly defects, but it creates the largest retention burden. Streaming chunk capture exposes partial responses as they are produced, which helps explain truncation, tool interleaving, and safety filters, but it introduces ordering and reconstruction problems. Selective field extraction retains only message roles, function arguments, retrieval citations, or system-instruction deltas, which is often enough to debug prompt construction without storing the whole conversation. Out-of-band content storage keeps the content in a separate governed store and retains only a trace-linked reference in telemetry, which is the safest pattern when compliance or access segregation matters more than inline readability.

The choice should follow the debugging question. If you need to explain why the model ignored a system instruction, structured extrac-

tion of the message array is usually sufficient. If you need to understand why a streaming response was cut off after a tool call, chunk-level capture with sequence numbers and timestamps is more useful. If you need a regulator-acceptable audit trail, a separate content repository with a trace identifier and access log is more defensible than span attributes. If you need only to correlate a later incident with the exact prompt version, a hash and versioned template identifier are enough.

Make minimization the default, not an afterthought. Minimization means you retain only the smallest content fragment that still supports the operational task. For prompts, that might be the system prompt ID, the user-visible template name, the variable values that were interpolated, and a truncated preview of the final assembled text. For responses, it might be the first error line, the tool invocation plan, or a redacted excerpt around the failure boundary. For retrieval, it might be document IDs, chunk offsets, and similarity scores rather than document bodies.

You need to distinguish between diagnostic fidelity and semantic completeness. A trace attribute that says "the prompt was long" is less useful than a normalized prompt-length bucket and a hash of the template version. A response preview that includes a PII-heavy customer name is worse than a redacted excerpt that preserves the sentence boundary and the completion error. When you preserve enough structure to answer "what happened?" without keeping "what was said?" verbatim, you reduce the odds that your observability backend becomes a secondary data warehouse.

Redact early, and redact where replication is still cheap. Redaction is most effective before fan-out. If you redact only after a collector has already exported content to multiple backends, the damage is done: caches, queues, backups, and downstream indexes may all have seen the original payload. Inline application-side suppression is therefore the first line of defense for known-sensitive paths, especially

for prompts that may contain account numbers, personal data, secrets, or regulated text. You can then apply collector-side redaction and deletion rules to catch anything the application missed.

Collector transforms are useful because they centralize policy. A single allowlist can strip raw message content while preserving model name, token counts, finish reason, and request timing. A single delete rule can remove fields that are legally or operationally off-limits. A single truncation rule can cap previews to a fixed length so that a prompt, response, or tool argument cannot silently expand the storage footprint. OpenTelemetry collector guidance positions the Collector as the practical place to transform telemetry before export, and the common specification allows attribute-length and attribute-count limits that give you additional hard bounds. [**?**, **?**, **?**]

A representative transform pipeline looks like this:

```
processors:
  transform/genai_redact:
    error_mode: ignore
    trace_statements:
      - context: span
        statements:
          - delete_key(attributes, "genai.prompt")
          - delete_key(attributes, "genai.response")
          - keep_keys(attributes,
              ["genai.request.model",
               "genai.usage.input_tokens",
               "genai.usage.output_tokens",
               "genai.response.finish_reason"])
```

This kind of rule is more durable than relying on every service team to remember a local masking library. It also gives you a single policy surface for auditing and change control. If the organization later decides that short previews are permitted for a subset of routes, you update the allowlist once rather than patching dozens of services.

Hashing and token-preserving transforms solve different problems. A non-reversible hash is a reference mechanism, not a

content substitute. It tells you whether two prompts or responses are identical, and it lets you join telemetry to an external store or offline dataset without carrying the content itself. Hashes are useful for deduplication, template drift detection, and forensic correlation. They are not a privacy panacea if the underlying content space is small or guessable. A fixed system prompt, a common refusal template, or a short classification answer can be brute-forced if you treat a hash as confidentiality.

Token-preserving transforms sit in a different category. You may need to keep the rough token sequence length, sentence count, or role structure so that latency, truncation, or prompt-compression bugs remain intelligible. In that case, you can replace content with placeholders of the same approximate size, normalize variable text into stable tokens, or preserve only delimiters and section boundaries. These transforms support debugging of assembly logic without preserving the original business meaning. They are especially useful when you need to compare a successful prompt path and a failing one while avoiding the original payload.

Reversible masking belongs in a narrower class. It can be appropriate when the observability store is not the system of record and a separate access-controlled vault holds the decryption material. That pattern is acceptable only when the organization has a clear operational need to recover content later and a well-defined approval workflow. If the policy requirement is genuine minimization, then reversible masking is the wrong tool.

Separate debug telemetry from content archives. Observability is for runtime diagnosis; archives are for governance, training review, legal response, or offline evaluation. Those uses overlap, but they are not interchangeable. A trace backend should not silently become the long-term home for raw customer conversations, retrieved documents, or model-generated outputs. If you need durable content

retention, move that responsibility into a separate content store with explicit ownership, lifecycle controls, and access auditing.

The join between telemetry and content should be by reference, not by duplication. A trace can carry a content object ID, a versioned prompt template name, a retrieval session ID, or an immutable audit key. A log line can record the same key with the active span and request outcome. The content store can then enforce its own retention and access policy while telemetry retains only the link necessary for diagnosis. This separation is especially important for streaming systems, where chunk events can otherwise accumulate into an accidental transcript service.

Retention classes need to reflect content sensitivity, not just signal type. You should not apply a single retention period to all telemetry that happens to involve GenAI. Metadata-only spans, token usage metrics, and route-level error logs usually support longer retention because they are less sensitive and more useful for trend analysis. Payload-bearing events, content previews, and tool argument captures need shorter retention and tighter access review. In practice, you end up with at least three classes: long-lived operational metadata, medium-lived debug content, and short-lived break-glass evidence.

The distinction matters during incident response. If a prompt content event is needed to debug a production refusal pattern, keeping it for a few days may be enough. If the same event contains regulated data, the right answer may be to retain only the redacted derivative and a short-lived access-controlled copy for the incident team. If the event contains secrets or credential material, immediate deletion or suppression may be the only acceptable outcome. The policy should be explicit about when content can move from temporary diagnostic retention into permanent storage, and who can authorize that move.

Access domains should mirror the content boundary. A backend that stores both metadata and raw payloads needs more than row-

level filters. You need separate access domains for operators, developers, compliance staff, and break-glass responders. Ordinary service owners often need latency, token, and failure attributes, but not the underlying prompt text. Security and privacy reviewers may need sampled access to content, but only through audited workflows. Incident commanders may need temporary elevated access to a small set of traces, but that access should expire automatically.

Role-based access control alone is not enough if every role can query the same index. You also need field-level controls, separate indexes or buckets for payload-bearing records, and audit logs that record who viewed which trace-linked content and why. If a backend cannot enforce that separation, the safer design is to keep content out of the backend entirely and expose only opaque references.

Use selective capture for the failure modes that actually need content. Raw content is justified only when it materially improves the root-cause loop. Typical examples include prompt-template assembly bugs, safety-policy regressions, malformed retrieval context, tool-call serialization errors, and jailbreak attempts that depend on exact wording. In those cases, capture a narrow slice around the failure, not the entire conversational history. A system prompt, the last user turn, the tool plan, and the first 200 characters of the model output are often enough to reconstruct the defect.

Content capture is much harder to justify for success paths, routine completions, or repetitive classification tasks. For those cases, token counts, finish reasons, model IDs, and high-level outcome labels usually suffice. If you retain every successful prompt because one failure might someday need it, you will over-retain by orders of magnitude and obscure the small set of traces that deserve closer review.

```
{
  "trace_id": "7f3c5c4c2d1e4f38a9c8c5f8c8a3a21b",
  "genai.request.model": "gpt-4.1",
  "genai.usage.input_tokens": 1280,
```

```
  "genai.usage.output_tokens": 186,
  "genai.prompt.template": "support_answer_v12",
  "genai.prompt.sha256": "b3a1...f9d2",
  "genai.response.finish_reason": "stop",
  "content.ref": "cs://case/2026-05-14/18422"
}
```

That record is enough to connect the inference span to the governed content store without placing the transcript directly in the trace backend.

Guard against late redaction and accidental propagation. Redaction loses value once the original payload has already replicated. Exporters, retry queues, dead-letter paths, and backend-side indexing can all create additional copies. If you redact in the application but then forward the raw payload into logs, metrics exemplars, or diagnostic events, you have not reduced the data liability; you have merely fragmented it. The same risk appears when a collector performs a transformation after fan-out. At that point, one exporter may already have persisted the raw field.

You should test the full data path, not just the code path that emits the span. Verify what reaches the collector, what survives processor chains, what lands in each exporter, and what the backend indexes for search. For GenAI payloads, the dangerous failure mode is often not the primary trace record but the secondary artifact generated by export, enrichment, or replay tooling.

Avoid reversible identifiers as accidental content proxies. A hash or opaque reference can become a backdoor if it is too easy to resolve, too broadly distributed, or too stable across contexts. If you embed a customer-specific prompt identifier in every trace, and that identifier is globally resolvable in internal systems, you may have created a new join key to sensitive material. Similarly, if you use the same stable identifier across tenants, environments, and retention classes, you make correlation easy for operators and attackers alike.

Prefer scoped identifiers that are meaningful only within a bounded access domain. Use environment-specific namespaces, tenant scoping, and expiring references where feasible. When you need cross-system correlation, carry the minimum reference necessary and keep the resolution service behind stronger controls than the telemetry store itself.

Treat streaming chunks as first-class records, not as a concatenated transcript. Streaming model output often tempts teams to reconstruct a complete transcript in the collector or backend. That is convenient for human reading, but it hides timing, partial failures, and tool interleaving. A better pattern is to treat each chunk as a record with sequence number, arrival time, role, and a redaction status. The collector can then drop or truncate content according to policy while preserving the fact that the output progressed in steps.

If you must reconstruct a transcript for a controlled workflow, keep the reconstruction in the governed content store, not in the observability backend. The trace can link to the reconstructed object, but the backend should remain a diagnostic index, not a transcript archive. That separation becomes especially important when chunks contain partial secrets, unescaped JSON, or intermediate tool arguments that were never intended for long-term retention.

Use a decision rule that is operational, not emotional. The right question is rarely whether content is "useful." Most content is useful to someone. The practical question is whether the content is needed to answer a bounded operational question faster or more accurately than structured metadata can. If the answer is yes, ask whether the needed content can be reduced to a snippet, a reference, or a transformed representation. If not, ask whether the retention can be confined to a separate governed store with short-lived access and an explicit reason code. If neither answer is satisfactory, capture no content.

That decision rule keeps the system honest. It allows you to debug

prompt assembly failures, explain harmful outputs, and support audit review without turning telemetry into a shadow data store. More importantly, it forces each payload-bearing field to justify itself against a concrete incident path rather than against a vague fear that "we might need it later."

7.2. Sampling and Cardinality Control for Expensive AI Traces

A GenAI deployment can become observability-expensive long before the model itself becomes expensive. Once you instrument inference spans, retrieval hops, tool calls, retries, and streamed token events, the backend cost often rises faster than token spend, query latency degrades, and dashboards turn sluggish because every useful question is attached to a dimension that explodes in cardinality. Sampling is the mechanism that keeps that system tractable, but for AI workloads it must preserve the traces that explain latency, token usage, and failure modes. The goal is not fewer traces in the abstract; the goal is a smaller, better-chosen trace set that still answers the operational questions you actually ask. OpenTelemetry makes the main architectural split explicit: head sampling happens in the SDK before spans are exported, while tail sampling happens after spans complete, typically in the Collector. [**?**, **?**]

A practical policy often starts with a small SDK sampler and a more selective Collector policy:

```
{
  "sampler": "parentbased_traceidratio",
  "arg": 0.05,
  "tail_rules": [
    {"name": "errors", "status": "ERROR"},
    {"name": "slow", "latency_ms": 2000},
    {"name": "costly", "output_tokens": 2000}
  ]
}
```

That simple split reflects the central trade-off. Head sampling keeps overhead low and is easy to reason about at request start, but it cannot see late-arriving evidence such as long completion time, a streamed model error, or a token count that only appears after the final chunk. Tail sampling can preserve those high-value traces, but it requires buffering and additional Collector resources while the decision window stays open. OpenTelemetry's sampling guidance recommends ratio-based sampling primarily for root spans in combination with parent-based sampling, which fits this pattern well because it preserves trace continuity without forcing every service to make an independent retention decision. The current TraceIdRatioBased behavior remains preserved until at least January 1, 2027, so policy work should be written with that compatibility boundary in mind. [**?**]

Treat sampling as a placement problem, not just a percentage. A single global sample rate rarely matches GenAI traffic. An interactive chat route, a batch evaluation job, and an agentic tool-execution path place different pressure on the tracing backend and produce different debugging value. A route that emits short, successful completions can often tolerate a lower head-sample rate, while a route that frequently triggers safety filters or tool failures may need near-complete retention for a small subset of requests. Remote sampling is useful precisely because it can vary strategy by endpoint, service, or span name, letting you keep expensive paths visible without paying full freight on ordinary ones. That is a better fit for GenAI than a blanket rate that either overspends or misses the very traces you need. [**?**]

The placement question also applies to where you make the decision. In-process head sampling is cheap and immediate, but it can only use information available at span start. Tail sampling centralized in the Collector can key on duration, status, token counts, exception payloads, and span-event patterns, but the decision arrives after buffering and

transport. The operational rule is simple: decide early when broad coverage matters, decide late when the value signal appears only after the trace has unfolded.

Separate broad coverage from selective retention. GenAI workloads usually need two retention layers. The first is broad, low-cost coverage for trend detection and coarse correlation. The second is selective retention for traces that carry diagnostic or financial significance. A typical implementation keeps metadata-rich, low-volume spans for most requests, then retains a much smaller set of traces that satisfy one of a few predicates: error status, latency above a threshold, unusually high token usage, unusual model routing, or a safety-related event. Tail sampling is the natural mechanism for that second layer because it can inspect completed traces and keep the ones that exceed your rules. The Collector documentation treats sampling-related processors as a standard part of the observability pipeline, and tail sampling is the one that most directly matches AI outlier retention. [**?**, **?**]

That layered model also solves a practical incident-response problem. If you sample away all successful traces, you may still detect a spike in failures, but you lose the baseline needed to compare prompt length, retrieval depth, or model routing before and after the regression. If you keep only success traces, you miss the rare failure that matters. Keeping both a slim always-on cohort and a selectively retained high-signal cohort gives you the statistical floor and the forensic ceiling.

Match the retention policy to the signal that arrives late. GenAI traces are expensive in part because the important attributes are often late. Token usage may only be known after the response completes. A streaming response may reveal truncation or refusal only after several chunks. A retrieval pipeline may emit the relevant span event after a vector search, a rerank, and a reassembly step. If you rely only on head sampling, you make the retention decision before those signals exist. That is acceptable for route coverage but weak for cost

and failure analysis.

Tail sampling makes sense whenever the useful predicate depends on completion. Examples include large output token counts, long latency, repeated retries, model fallback, or an exception only visible in the final span status. If the workload is highly bursty, tail sampling also lets you prioritize rare but high-value traces during overload, because the decision logic can favor error and outlier cohorts even when the traffic spike would otherwise overwhelm a fixed head rate. The cost of that selectivity is memory, buffering latency, and more complex tuning. You need enough Collector headroom to hold unfinished traces until the decision window closes, and you need a decision wait time that is long enough to observe the relevant terminal signals but short enough to avoid excessive backlog. [**?**]

Preserve correlation when you sample. Sampling only works operationally if sampled traces remain joinable to logs and metrics. That requires consistent trace context propagation, stable correlation identifiers in logs, and a metrics model that does not depend on every trace being present. This is where the earlier signal-correlation foundation matters: the sampled trace is the detailed specimen, not the only record of the event. If you are using exemplars, trace IDs, or log correlation fields, preserve those identifiers even when the trace itself is dropped so that aggregate signals still point to representative or high-value cases. The operational target is not a perfectly complete trace corpus; it is a corpus that can still anchor your triage workflow. W3C Trace Context remains the standard propagation mechanism for that cross-service linkage. [**?**, **?**]

A common failure mode is to downsample traces and then accidentally downsample the surrounding evidence with them. That makes a bad trace untraceable, because neither the logs nor the metrics know how to point back to it. Keep the join keys even when you discard the bulk payload.

Budget expensive attributes separately from trace volume. Sampling controls how many traces you keep. Attribute limits control how large each kept trace can become. For GenAI workloads, those are independent cost levers. A trace that contains a long prompt preview, a multi-chunk streamed output, a retrieved document list, and a detailed exception payload can be individually expensive even if the sample rate is low. OpenTelemetry's common specification supports attribute-count and attribute-length limits, which are the right place to cap runaway prompt previews, overly verbose tool arguments, or large retrieval snippets. [**?**]

Use these limits as a second guardrail rather than as a substitute for sampling. If a prompt preview is 20,000 characters long, cutting the sample rate from 5% to 1% does not solve the per-trace storage problem. Truncation, field deletion, or reference substitution does. For AI traces, you should think in two budgets: how many traces per minute you are willing to store, and how many bytes each retained trace may carry. Both budgets must stay bounded.

Control cardinality before it contaminates every query. Sampling reduces volume; cardinality control reduces shape complexity. Teams often confuse the two and assume that a lower trace rate will automatically fix slow queries. It will not if each retained trace still carries high-cardinality attributes such as user IDs, request IDs, prompt hashes, document IDs, experiment IDs, and model-version overrides. Those values may be useful for one-off debugging, but they are disastrous as primary grouping dimensions because they explode the number of unique combinations the backend must index and aggregate.

A practical classification helps here. Analytical cardinality is acceptable when the value space is bounded and the field supports population analysis, such as route, model family, tenant class, or coarse latency bucket. Identity cardinality is dangerous when the field names a single request, user, prompt, or document. Identity values belong in traces

or logs as lookup keys, not as metric labels and not as default span attributes that every backend index must ingest. Prometheus guidance on label discipline is the same idea in a different vocabulary: keep metrics low-cardinality and stable, and move high-cardinality detail into traces or logs. [**?**, **?**]

Bucket, hash, or externalize when exact identity is unnecessary. If you do not need exact identity, you usually should not store it as exact identity. Bucket model versions into families, bucket latency into coarse ranges, and hash prompt or document identifiers only when equality comparison is enough. A hash is useful when you need deduplication or joinability, but it still has cardinality cost if you use it as a grouping label. If the backend is meant for operational trend analysis, a stable bucket or family label is usually better than an opaque unique value.

There is an important distinction between using a hashed field as an investigative reference and using it as a primary dimension. As a reference, the hash helps you correlate two records or link a span to an external store. As a dimension, it creates almost as many unique buckets as the raw value. The same caution applies to prompt hashes, retrieved-document IDs, and per-request model override strings. Keep them out of charts and alerts unless you have a very specific, bounded use case.

Use metrics for accounting, traces for explanation. One of the easiest ways to overload a trace backend is to force it to answer accounting questions. If you want to know whether tenant A consumed more tokens than tenant B over the last day, aggregate that in metrics. If you want to know why a particular request consumed more tokens, inspect the trace. If you want to know whether a model rollout increased average completion cost, use a metric derived from all requests, not a sampled trace population pretending to be the complete population.

This split matters because sampled traces are excellent for causality but

weak for financial truth. If you compute spend directly from sampled traces, you will undercount unless you apply a defensible expansion method and understand its error bars. For cost attribution, keep an unsampled or separately aggregated accounting path for token totals, then use traces to explain outliers, regressions, and anomalies. That keeps the backend fast and preserves the meaning of the numbers you present to product and finance stakeholders. GenAI semantic conventions already expose token counts and model identifiers in the structure of spans and metrics, which makes this division practical. [**?**, **?**]

Reduce event chatter before you sample traces. Streaming systems can produce a large number of span events or log-like emissions for a single request. If each generated token, chunk boundary, or tool state transition becomes a separate event, trace size balloons even when the trace is sampled at a low rate. The more verbose the stream, the less meaningful head sampling becomes, because a tiny fraction of requests may still generate enormous retained traces. In practice, you often need event dropping, event coalescing, or summarized state changes before you look at trace sampling.

A useful rule is to retain transitions, not every heartbeat. Keep the event that marks the first tool call, the event that captures the retry, the event that records truncation, and the event that records finish reason. Drop repetitive chunk chatter unless the stream itself is the thing you are debugging. That preserves the operational narrative without turning each trace into a transcript of every token boundary.

Choose head sampling when breadth matters more than completeness. Head sampling is the right default when your main goal is broad coverage at low cost. It is deterministic at request start, cheap to implement, and stable across a distributed system because parent-based propagation keeps downstream spans aligned with the root decision. It works well for trend analysis, general reliability monitoring, and environments where most traces are ordinary and

you only need a representative slice.

Its weakness is obvious in AI systems: the interesting part often comes later. If the failure signal is a long completion, a late token burst, a tool timeout, or an upstream retrieval stall, a head decision cannot see it. That makes head sampling a coverage tool rather than a precision tool. Use it to keep the observability budget bounded; do not rely on it alone to retain rare but expensive incidents.

Choose tail sampling when outliers drive the investigation. Tail sampling pays for itself when the traces you care about are the traces that finish badly, slowly, or expensively. It lets you retain error traces, long traces, traces with large token counts, and traces that triggered fallback logic. That is especially useful for GenAI because the rare bad request often carries more diagnostic value than ten ordinary requests. It is also useful for selective retention on specific endpoints, where a small number of routes need aggressive observability and the rest do not.

The trade-off is operational complexity. You need a Collector that can buffer completed traces, a decision policy that is explicit and auditable, and enough memory to survive traffic bursts. You also need to tune the waiting period carefully. If the decision window is too short, late signals will be missed. If it is too long, the Collector becomes a holding area rather than an efficient filter. Tail sampling is a precision instrument; you do not use it to replace an entire observability strategy.

Roll policy changes out as if they were production code. Sampling and cardinality rules are not bookkeeping settings. They change what your operators can see. Before tightening a policy, inventory the dimensions in use, classify them as required or optional, and identify which dashboards, saved searches, and alerts depend on them. Then test the reduced-retention policy against real incident workflows. Ask whether you can still answer the same questions: which tenant re-

gressed, which model route failed, whether token spend rose because of prompt growth or because of a fallback path, and whether a slow request was an isolated outlier or part of a broader trend.

That rollout discipline prevents two common errors. The first is sampling away all successful traces and then losing the baseline needed to interpret failures. The second is retaining too much identity detail and then discovering that the backend cost problem simply moved from ingest to query. A good policy keeps the accounting path intact, preserves enough exemplars for forensic work, and constrains the dimensions that would otherwise make every query expensive. When you keep those three constraints in view, sampling stops being a blunt cost lever and becomes a controlled part of GenAI operations.

7.3. Correlating Traces, Metrics, and Logs for Triage and Optimization

When a GenAI incident hits production, the painful part is rarely the absence of telemetry. More often, you have too much of the wrong kind in the wrong place: a percentile latency graph shows a regression, a few traces show elevated token counts, and logs contain scattered provider errors, but none of those signals alone explain whether the root cause is prompt inflation, retrieval slowdown, a route change, or a tool failure. The practical answer is to make the three core signals work as one workflow. Metrics tell you that something changed, traces explain how the request unfolded, and logs provide the local evidence that is too verbose or too ephemeral to keep in spans. OpenTelemetry and W3C Trace Context give you the standards basis for that linkage: trace identity propagates across services, logs can carry trace and span identifiers, and baggage can carry only the small, safe business context you need to slice the data without exploding cardinality.

A minimal log correlation pattern looks like this:

```
{
  "ts": "2026-05-14T15:22:41Z",
  "level": "error",
  "msg": "provider timeout",
  "trace_id": "7f3c5c4c2d1e4f38a9c8c5f8c8a3a21b",
  "span_id": "4d2b1a0c9f8e3b77",
  "tenant": "acme",
  "route": "chat.send",
  "model": "gpt-4.1"
}
```

That record is small, but it changes how you investigate an incident. You can jump from a metric anomaly to representative traces, and from a trace to the exact log lines that explain the failure mode. In some language SDKs, trace context fields such as TraceId, SpanId, and TraceFlags are populated automatically in logs from the active span context, which reduces the risk that correlation breaks because an application forgets to inject the identifiers manually.

Assign each signal a distinct operational job. If you let all three signals do the same work, you will duplicate data and still miss the answer. Metrics should remain aggregate and bounded. They answer questions such as whether error rate rose, whether p95 latency shifted, whether token usage accelerated, and whether retrieval hit rate fell below the expected band. Traces should explain structure and causality. They answer which branch of an agent plan executed, where retries occurred, whether a tool call stalled, and how long each stage of the request path consumed. Logs should preserve the narrow narrative and edge evidence: provider-specific errors, parser failures, content-filter rejections, truncation notices, serialization exceptions, and internal state transitions that would be too noisy for spans.

This division is what keeps observability scalable. Metrics give you population-level truth with low storage cost. Traces give you per-request detail, but only for the subset you retain. Logs give you exact

text and local state, but only when you can correlate them back to the trace. When you follow that split, you avoid the anti-pattern of copying the same prompt, response, or error payload into all three signals.

Use metrics as the change detector. The most efficient triage starts with a signal that changed, not with a trace that you hope is representative. For GenAI systems, that signal is often a dashboard or alert tied to one of a few behaviors: token spend per request, completion latency, retrieval latency, tool timeout rate, refusal rate, or completion error rate. Those metrics are the system's early warning layer. They tell you which route, tenant, model family, or environment deserves inspection.

Once you know the change signal, segment it along stable dimensions that have bounded cardinality: route, model family, deployment version, tenant class, or coarse prompt category. Avoid slicing on identity values such as user ID, prompt hash, or document ID in the metric layer. If you need those values, move to traces or logs. The point of the metric layer is to tell you where the regression lives, not to preserve every request identity that happened to be affected.

A practical triage rule is to begin with a rate or percentile and then compare cohorts. If p95 latency rose only for one route, look for route-specific traces. If token spend grew only for one tenant or experiment cohort, look for prompt changes or fallback behavior in that cohort. If retrieval latency rose without a matching token increase, the issue is probably before generation rather than inside the completion model.

Use traces as the structural explanation. A trace tells you how a request was assembled, routed, retried, and completed. For GenAI systems that structure matters more than in ordinary RPC systems, because the interesting latency often sits in multiple stages: prompt assembly, retrieval, reranking, model selection, generation, tool execution, and post-processing. A single long request can hide several dif-

ferent bottlenecks. The trace is the only signal that can show whether the user spent most of the budget in retrieval, in a tool call, or in the completion itself.

When you inspect a suspicious trace, look for the stage where time accumulated and for the branch that differs from the normal path. Did the agent route to a fallback model? Did the retriever return fewer chunks than expected, forcing a second query? Did a tool retry twice before completing? Did token growth precede a longer completion delay? Those are structural questions, and traces answer them better than metrics or logs alone.

Traces also give you a place to attach safe context from baggage. The baggage model is useful for low-cardinality routing hints such as tenant class, experiment cohort, prompt template version, or session type. That context lets you filter traces and logs without turning those same values into high-cardinality metric labels. The caution is straightforward: baggage is for small, safe values only. Do not use it for large mutable objects or sensitive content. The standards guidance warns that baggage can leak broadly to unintended recipients, so keep it bounded and intentional.

Use logs as the local proof. Once the trace identifies the suspect span, logs usually supply the immediate cause. They tell you which provider timed out, which parser failed, whether a safety filter cut the response, and what exact exception text occurred at the edge of the failure. That detail is often too ephemeral or too verbose for a span attribute, but it is indispensable when you need to confirm the mechanism.

The most useful pattern is not “trace first, logs second” in a rigid sense, but a reversible pivot. Start from a metric anomaly, find a representative trace, then jump to the logs attached to the trace and active span context. Or start from a distinctive log signature, such as a provider

timeout or a malformed tool schema, and locate the affected traces by trace ID or span ID. The important part is that the identifiers survive the hop. Without them, logs become isolated text and traces become isolated structure.

If your language runtime populates trace identifiers automatically in log records, enable that path and keep it consistent across services. If it does not, inject the identifiers in the application logger and verify the output format. A correlation field that is present in one service and missing in another will break the incident workflow as soon as you cross a boundary.

Build a repeatable triage flow. A disciplined workflow saves more time than a heroic one-off search. The sequence should be consistent enough that on-call engineers can run it under pressure.

- Start with the alerting metric and identify the affected cohort.
- Split by route, model family, tenant class, or deployment version.
- Inspect representative traces or exemplars for each cohort.
- Pivot from the suspicious trace to the correlated log lines.
- Confirm whether the failure is isolated, systemic, or route-specific.
- Validate the fix by watching the same metric band after rollout.

That sequence works because each step narrows the search space without discarding context. The metric says "something changed." The trace says "where in the path it changed." The log says "why this particular request failed." For GenAI systems, exemplars are especially useful because they give you a concrete trace anchor for a metric spike instead of forcing you to search an entire trace corpus blindly. The exact exemplar behavior depends on the backend, but the design goal

is always the same: make a metric point to a sample trace that is worth inspecting.

Treat baggage as routing context, not as storage. Baggage is valuable when you need a few small values to follow the request across the system. It is not a place to stash arbitrary context. In GenAI operations, the best baggage fields are coarse and stable: tenant class, experiment bucket, route name, prompt template ID, session type, or deployment cohort. These fields let you segment metrics and traces without exploding cardinality. They also make it possible to carry business context into downstream services that might not otherwise know which cohort a request belongs to.

You should keep baggage sparse for two reasons. First, the propagation cost grows with every field. Second, the privacy and leakage risk grows with every field, because baggage can be forwarded more widely than you expect. If you need a value primarily for diagnosis and it is not safe to propagate broadly, put it in a span attribute or a log field instead, and keep baggage reserved for the minimal routing hints that make correlation work.

Use metric labels only when aggregation survives scale. A metric label is a promise that the value space stays bounded enough for your backend to aggregate it cheaply. That promise is easy to violate in AI systems. User IDs, prompt hashes, document IDs, per-request experiment names, and ad hoc model override strings create immediate cardinality pressure. They also make dashboards noisy, because every new value becomes a new series.

The safe label set is small and stable: route, model family, deployment version, tenant class, environment, and perhaps a coarse completion outcome. If you need exact request identity, use traces and logs. If you need exact business context, use baggage only for the small values that are safe to propagate. Keep metrics for trend detection and SLOs, not

for forensic identity.

This split also matters for token cost analysis. Aggregate token metrics can tell you that spend rose, while trace-level token attributes can explain which request or route caused the rise. If you try to encode per-request token detail as metric labels, you will turn a useful accounting signal into a storage problem.

Understand the placement trade-off for each datum. A single datum can reasonably live in different signals depending on its role. Ask four questions: Does it need to be aggregated? Does it need per-request identity? Does it need to propagate across services? Does it need a long retention horizon?

If the answer is aggregate-only, put it in metrics. If the answer is per-request diagnostic and joinable, put it in traces or logs. If the answer is small and safe routing context, baggage may be appropriate. If the value is sensitive or large, keep it out of baggage and keep it out of metric labels. For example, tenant class belongs naturally in baggage or a low-cardinality label; the exact user prompt does not. Route name is fine as a label; full document text is not. Finish reason can live in spans and metrics; raw provider error payloads belong in logs or controlled content telemetry, not in labels.

This is the same judgment you apply elsewhere in the chapter, but here it is framed by signal behavior rather than by privacy or sampling alone. The right placement reduces cost, preserves joinability, and keeps the backend usable.

Use correlated signals to separate prompt regressions from infrastructure regressions. A common GenAI failure is ambiguous from any single signal. Suppose latency rises and token spend rises at the same time. That could mean a longer prompt template, a retrieval change that adds context, a model fallback that uses a slower endpoint, or a tool path that now retries. If the metric only shows the

aggregate change, you need traces to inspect path shape and logs to inspect the failure mechanism.

A prompt regression usually appears as larger input tokens, higher prompt assembly latency, or more retrieval context in the trace, with logs showing little or no provider error. An infrastructure regression usually appears as stable prompt size but longer retrieval, slower provider calls, or retry-related log messages. A route regression may show a different model family or endpoint in baggage or trace attributes. Using the signals together lets you distinguish those cases quickly instead of treating every slowdown as a model problem.

That distinction matters for remediation. Prompt regressions often require template changes or retrieval tuning. Infrastructure regressions often require routing, capacity, or timeout changes. If you only inspect one signal, you may fix the wrong layer.

Close the loop with optimization, not just incident response. Correlated signals are not only for firefighting. They are the basis for controlled optimization. If you revise a prompt template, watch whether the trace-level token counts fall, whether the completion latency narrows, and whether the refusal rate changes for the same cohort. If you change a retriever, watch retrieval latency, chunk count, and downstream generation cost. If you alter model routing, compare the old and new route cohorts in the same metric band and then inspect the traces that still look anomalous.

That workflow is what turns observability into engineering feedback. Metrics tell you whether the system got better. Traces tell you which part got better or worse. Logs tell you whether the improvement was clean or whether a new edge failure appeared. With stable trace context, bounded baggage, and disciplined metric labels, you can move from page to root cause to validated fix without losing the connection between the signals that described the problem.

7.4. Building Durable Dashboards and Queries Across Version Changes

A GenAI observability stack usually breaks in a quieter way than an application does. The traces still arrive, the Collector still exports, and the backend still stores data, but a dashboard panel goes blank because an attribute was renamed, a query silently filters the wrong field, or a saved search assumes a payload layout that no longer exists. The failure is subtle because the instrumentation is still "working" while the operational contract has drifted. Durable dashboards treat version change as a design constraint, not as an upgrade nuisance. OpenTelemetry's telemetry schemas exist specifically to manage evolution between producers and consumers, and the GenAI semantic conventions are still evolving in places, including fields that are marked development or recommended and event bodies that may move toward more structured attributes over time.

A practical way to absorb that drift is to normalize early and query late:

```
processors:
  transform/house_schema:
    error_mode: ignore
    trace_statements:
      - context: span
        statements:
          - set(attributes["ai.model"],
              attributes["genai.request.model"])
          - set(attributes["ai.in_tokens"],
              attributes["genai.usage.input_tokens"])
          - set(attributes["ai.out_tokens"],
              attributes["genai.usage.output_tokens"])
          - delete_key(attributes, "genai.request.model")
          - delete_key(attributes, "genai.usage.input_tokens")
          - delete_key(attributes, "genai.usage.output_tokens")
```

That pattern gives you a stable internal contract even when upstream names change. The Collector can translate old and new producer fields into a house schema before export, and your dashboards can bind to

the house names instead of the raw semconv names. That separation is the core design principle: emitted telemetry may change, but the query surface should change far less often. OpenTelemetry explicitly documents telemetry schema support for transforming received data from a producer schema version to the consumer schema expected at query or dashboard time, which is exactly the mechanism you need when service teams upgrade at different speeds.

Separate emitted schema, transport normalization, and query aliases. The first mistake teams make is to treat the raw span as the dashboard contract. That works until the instrumentation library changes a field name, the semantic convention graduates a field, or a backend normalizes data differently than the upstream service. You need three layers instead.

The emitted schema is what services write. That layer changes as SDKs, semconv packages, and application code evolve. The transport or normalization layer is the Collector or ingestion tier that rewrites fields into a stable internal vocabulary. The query layer is the set of dashboards, alerts, recorded queries, and runbooks that should depend on the stable vocabulary rather than on the raw upstream names. If you blur those layers, you push upgrade risk directly into every operational asset.

This separation also gives you a migration boundary. Services can upgrade semconv versions independently, the Collector can dual-map old and new names during the transition, and dashboards can remain pinned to the house schema until you intentionally update them. That is much safer than rewriting queries every time a library release lands.

Normalize to semantic concepts, not serialized payload shapes. GenAI semconv evolution is not limited to renaming. Some event body fields are expected to move toward more structured attributes, and some current representations are only transitional. If

you build a saved search around the exact JSON shape of a serialized message array, you are binding your operational tooling to a layout that may not survive the next instrumentation revision. Durable queries should target the semantic concept: input token count, output token count, model name, finish reason, tool call name, retrieval query latency, or response truncation status.

That distinction matters most when payload data is partially captured. A dashboard that reads "the second array element in a JSON string" is brittle. A dashboard that reads "the normalized prompt template name" is stable. A panel that depends on a raw event body will also be difficult to migrate across collector pipelines, because different exporters and backends may index bodies differently or not at all. Prefer a query path that survives structure changes by design.

Use version-pinned dashboards as migration scaffolding. Version pinning is not the same as freezing the system forever. It is a way to make the migration explicit. When you know a dashboard assumes a particular semconv generation or collector mapping, annotate it with that assumption and keep a side-by-side version for the next schema. That lets you compare old and new outputs during rollout without forcing a hard cutover.

A practical migration often uses three artifacts: a legacy panel, a compatibility panel, and a target panel. The legacy panel confirms what the old schema still reports. The compatibility panel reads the house schema or dual-mapped fields. The target panel is what you want after the migration settles. During the transition, keep all three visible to the people who own the service and the on-call workflow. Once the compatibility panel matches the target panel for a representative period, you can retire the legacy view.

This pattern matters because schema drift often reveals itself only in edge cases. A field may exist in development or recommended form

in one release and move or disappear in another. A panel may look correct for the median cohort but fail for one route or model family. Side-by-side validation catches those mismatches before they become an incident.

Design a house schema with explicit ownership. A house schema is a locally governed vocabulary that absorbs upstream churn. It should be small, stable, and operationally meaningful. For GenAI observability, a sensible house schema might include model family, route, token in, token out, finish reason, response status, retry count, retrieval latency, tool latency, and request cohort. That is enough to power the majority of dashboards and alerts without exposing every raw semconv field.

The house schema should not become a dumping ground for every upstream attribute. Keep the vocabulary narrow and document the mapping rules. If the instrumentation changes from one semconv name to another, update the Collector mapping once. If a backend changes its indexing model, update the normalization layer once. Dashboards and alerts should not need to know which service emitted which semconv version. They only need the house fields that answer the operational question.

A compact mapping table is often enough to make ownership clear:

House field	Source examples	Use
`ai.model`	`genai.request.model`, `llm.request.model`	Route and cohort segmentation
`ai.in_tokens`	`genai.usage.input_tokens`	Cost and prompt growth analysis
`ai.out_tokens`	`genai.usage.output_tokens`	Completion cost analysis
`ai.finish_reason`	`genai.response.finish_reason`	Truncation and refusal analysis
`ai.retry_count`	service-local retry metadata	Failure and latency analysis

The exact field names will vary by estate, but the principle stays constant: normalize the operational semantics, not the implementation details.

Expect GenAI semconv churn and plan for it. GenAI seman-

tic conventions are still moving. Some fields are still marked development or recommended, and some event-body representations are transitional. That is not a problem if your dashboards query through a compatibility layer; it is a problem if they depend on raw names or payload shapes. Writers should assume at least three kinds of churn: renamed fields, reclassified stability levels, and structural changes in event bodies or message fields.

The safe response is to treat raw semconv names as input, not as your final contract. The Collector can translate old names into stable aliases, backfill new names from old data when possible, and drop transient fields that do not belong in long-lived dashboards. When a backend supports schema-aware query translation, that can help too, but you should not rely on vendor-specific normalization as your only line of defense. The durable boundary lives in your own pipeline.

Keep queries stable by querying derived dimensions, not raw payloads. Saved searches and dashboard queries should prefer fields that the normalization layer computes explicitly. If a backend query hits a derived field such as `ai.model` or `ai.in_tokens`, the query survives changes in upstream field names. If it hits a raw payload path, it will break the moment the body structure shifts.

That advice is especially important for token and cost analysis. Token counts are the most common GenAI dashboard metric, but they are also one of the easiest fields to rename or relocate as semconv packages evolve. A query that aggregates on a stable house field can continue to work while the underlying instrumentation migrates from one semconv family to another. If you also keep the original raw attribute in a compatibility panel for a release or two, you can validate that the normalized field still matches the source before you remove the old path.

For metrics derived from traces, use recording rules or backend trans-

forms that target the house schema rather than the emitted schema. That keeps the aggregation rules stable even if the instrumented service changes its vocabulary.

Use dual-write or dual-read windows deliberately. Migration failures often come from moving too fast. If you rename a field in the service and remove the old name immediately, every dashboard that still reads the old field breaks at once. A safer strategy is dual-write or dual-read.

Dual-write means the producer or Collector emits both the old and new names for a controlled period. That gives you time to compare outputs and update queries. Dual-read means the consumer query accepts either field and prefers the new one when present. Dual-read is often better for dashboards because it lets old and new services coexist without a long overlap in emitted telemetry volume. Dual-write is often better when the query language lacks a clean coalescing function or when you need to validate exact equivalence during rollout.

Whichever form you choose, define the end date. Compatibility windows that never end become permanent technical debt, and the house schema loses its meaning if every panel still depends on old aliases. The point is to make migration observable, not indefinite.

Sequence migrations from low-risk fields to high-risk joins. Not all fields are equally sensitive. A model-family label is easier to change than a field used as a join key across traces, metrics, and logs. The safest sequence is to migrate low-risk descriptive fields first, then high-value aggregations, then cross-signal join keys, and only then retire the old schema path.

Start with a field such as model family or route alias. Validate that the new house field matches the old semantics. Then move on to token counts or finish reasons, which feed important dashboards but are still relatively local. Leave correlation keys, trace identifiers, and log join

fields for last, because breaking those has the highest operational cost. If a migration affects metrics, traces, and logs together, verify each signal independently and then verify the joins across signals.

This sequencing also helps with mixed estates. Not every service will upgrade at the same time, so the normalization layer has to tolerate old and new fields simultaneously. A central mapping layer lets you survive that overlap without forcing a lockstep release across the fleet.

Test queries against sample data from both schemas. A query that works against one schema version may fail on the next because missing fields collapse the result set or because the backend interprets a renamed field as a different dimension. The only reliable defense is to test saved queries against sample data from both the old and new schemas before you ship the migration.

You do not need a full production clone to do this well. A representative set of traces and metrics from the legacy schema and the target schema is enough to surface the common failures: missing fields, wrong grouping, broken filters, and accidental cardinality growth. If you can, test both the raw emitted data and the normalized house schema. That tells you whether the mapping layer is doing the right thing and whether the query still behaves correctly after translation.

A simple validation checklist helps here: the panel returns data, the aggregate value matches within expected tolerance, the top cohorts are unchanged, and the alert threshold still fires under the same conditions. If any of those fail, the migration is not ready.

Annotate dashboards and runbooks with schema assumptions. A dashboard without schema assumptions invites future confusion. If a panel expects the house schema as of a specific mapping version, state that in the panel description or in the runbook that references it. Include the producer semconv family, the Collector transform version, and any known field aliases the query accepts. That metadata

tells later operators whether a blank panel means an outage or just a schema mismatch.

This annotation matters most in mixed environments where some services are still emitting older fields. A well-annotated runbook lets the on-call engineer answer a practical question quickly: is the signal absent, or did the query drift away from the data? That distinction can save a lot of time during an incident.

Avoid the common failure modes. The most damaging anti-pattern is hard-coding unstable development-status field names into alerts. The alert may work today and fail after the next semconv update. A close second is migrating traces first and leaving metrics and logs behind, because that breaks the cross-signal joins that make triage efficient. Another failure is trusting backend vendor auto-mapping as if it were a durable contract. Backends often normalize fields, but that behavior is not a substitute for your own house schema.

You should also avoid using serialized payload structures as dashboard sources when a stable semantic field exists. If the backend changes how it indexes message arrays or complex attributes, your panels will break even though the underlying data still exists. Likewise, do not assume that a field that is "recommended" today will have the same shape or name forever. Recommended is not the same thing as frozen.

Treat schema translation as part of the observability architecture. The most resilient estates do not ask every dashboard author to chase semconv changes. They centralize the burden in the Collector or ingestion layer, expose a house schema, and version the translation rules alongside instrumentation changes. That gives you a stable contract for dashboards, queries, recording rules, and detectors while still allowing the producer side to evolve.

The payoff is operational, not cosmetic. When a semconv upgrade lands, you update translation rules, validate against sample data, and

leave the dashboards intact. When a new GenAI attribute appears in the spec, you decide whether it belongs in the house schema or whether it should remain a raw input only. When a backend changes its query behavior, you adjust the query layer against stable aliases instead of rewriting every service. That discipline keeps observability assets usable through change, which is the real requirement in a mixed and evolving GenAI estate.