Build Your Own Observability Platform: Design Document

Overview

This document outlines the design of a unified observability platform that ingests, stores, and correlates logs, metrics, and traces. The key architectural challenge is creating a single system that efficiently handles the distinct data shapes and query patterns of these three 'pillars' while enabling powerful cross-signal correlation to debug complex distributed systems.

> This guide is meant to help you understand the big picture before diving into each milestone. Refer back to it whenever you need context on how components connect.

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4, Milestone 5

1. Context and Problem Statement

Modern software systems, particularly microservices architectures, have evolved into intricate distributed networks. A single user request might cascade through dozens of independent services, each running across ephemeral containers spread over multiple data centers or cloud regions. While this decoupling enables agility and scalability, it creates a fundamental challenge for engineers: **system visibility becomes fragmented**. When a performance issue or failure occurs, the root cause is no longer isolated to a single log file or server metric; it's buried somewhere within the complex interactions of the system.

Traditional monitoring tools—designed for monolithic or simpler architectures—struggle in this environment. They provide **metrics** (like CPU usage or request rate), **logs** (textual records of events), and sometimes **traces** (the journey of a request through services), but these three "pillars" of observability are typically stored and queried in isolation. An engineer debugging a slow API endpoint might start with a dashboard showing high latency (metrics), then jump to a trace viewer to find the slow span, then switch to a log aggregator to search for error messages from that service, manually copying identifiers like `trace_id` between tools. This context-switching is slow, error-prone, and often fails to reveal the subtle, emergent behaviors that cause distributed failures.

This project addresses the core need for **unified observability**—a platform that doesn't just collect logs, metrics, and traces, but intrinsically correlates them. It aims to build a coherent system where a slow metric automatically links to exemplar traces, where logs are indexed by their originating trace, and where a single query can join data across all three signal types. The goal is to transform raw telemetry data into actionable understanding, enabling engineers to debug complex systems with the same ease as they once debugged a single process.

Real-World Analogy: The Air Traffic Control Center

Imagine a busy airspace, like the one over a major airport. This is your microservices architecture. Each aircraft represents a service instance—some are passenger jets (core user-facing services), others are cargo planes (background workers), and some are small private planes (utility services). Visibility into this airspace is provided by several independent systems:

- **Radar Systems (Metrics):** These provide a real-time, quantitative view: altitude, speed, bearing, and transponder codes for each blip on the screen. They tell you "what" is happening at an aggregate level (e.g., "sector 5 is getting congested") but not "why" (e.g., "plane N123 is circling because of a landing gear warning light").
- **Radio Communications (Logs):** Air traffic controllers and pilots exchange verbal messages. These are unstructured or semi-structured narratives: "Tower, N123, we have a warning light on panel 3, requesting hold pattern." This data is rich with context but is only useful if you are tuned to the right frequency (service) and can piece together the conversation (trace) from fragmented, asynchronous messages.
- **Flight Plans (Traces):** Before takeoff, each flight files a plan detailing its intended route, waypoints, and estimated times. This is the *structured narrative* of a request's journey. When a plane deviates or is delayed, comparing its actual radar track (metrics) and radio chatter (logs) against its flight plan (trace) is the fastest way to understand the anomaly.

An air traffic controller's effectiveness depends entirely on a **unified control room** that overlays radar data, radio audio channels, and flight plan information onto a single situational display. When a problem occurs—like a plane entering restricted airspace—the controller doesn't juggle three separate screens. They see the radar blip, click it, and instantly hear the related radio traffic and see the filed flight plan. This correlation is what turns data into understanding and enables decisive action.

Our observability platform aims to be that unified control room for software systems. It must ingest the radar feeds (metrics), radio recordings (logs), and flight plans (traces), then build and maintain the indexes that link them together, providing a single pane of glass for system understanding.

The Core Problem: Debugging in the Dark

Let's make this concrete with a classic debugging scenario in a microservices e-commerce platform. A user reports that "Checkout is slow."

1. **The Metric Blind Alley:** You check your dashboard. The `checkout_service.request.duration.p99` metric is spiking. This confirms the symptom but gives no clue about the cause. Is it the database? The payment service? A new deployment? The metric alone is a dead end.
2. **The Log Haystack:** You open your log aggregator and search for `ERROR` or `WARN` in the `checkout_service` from the last 15 minutes. You get thousands of results. Some are related, most are not. You see a cryptic error: "Failed to acquire inventory lock." Which user session did this affect? What was the full context?

Without a `trace_id` or `user_id` readily searchable, you're left sifting through noise.

3. **The Trace Odyssey:** You open your distributed tracing tool and filter for slow traces in the `checkout_service`. You find a trace that took 12 seconds. The trace view shows a deep tree of spans. You see that a call to the `inventory_service` took 8 of those seconds. Progress! But why? You need to see the logs from that specific `inventory_service` instance for that specific request to understand the error or delay.
4. **The Manual Correlation Grind:** You copy the `trace_id` (a 32-character hexadecimal string) from the trace viewer. You switch back to your log aggregator, paste the `trace_id` into the search bar, and hope the logs were emitted with that field attached. If you're lucky, you find the "Failed to acquire inventory lock" log line. Now you need to understand *why* the lock was contended. You might need to look at metrics for the inventory database, or logs from other concurrent traces. Each hop requires manual, error-prone copying of IDs and mental context switching between different tools' query languages, time pickers, and UIs.

This process is what we call **"debugging in the dark."** The data exists, but the connections between data points—the **correlation**—are missing or require heroic manual effort to reconstruct. The cognitive load is immense, and time-to-resolution stretches from minutes to hours or days. In a production incident, this delay directly impacts user experience and business outcomes.

> **Design Insight:** The primary value of a unified observability platform is not in storing more data, but in **creating and preserving context**. The relationships between signals (a log belongs to a trace, a metric spike is exemplified by a trace) are first-class citizens that must be captured at ingestion and made queryable.

Existing Approaches and Their Trade-offs

Before designing our own system, it's critical to survey the landscape and understand the trade-offs of existing solutions. There are three primary architectural patterns for observability today, each with distinct advantages and drawbacks for a learning or bespoke implementation.

Decision: Evaluating Foundational Observability Architectures

- **Context:** We need to choose a starting point and set of standards for our platform that balances interoperability, learning value, and implementation complexity.
- **Options Considered:**
  1. **Commercial SaaS Platforms (e.g., Datadog, New Relic, Splunk):** Integrated, full-featured platforms offered as a service.
  2. **DIY Point Solutions (e.g., Prometheus for metrics, ELK for logs, Jaeger for traces):** A assemblage of best-in-class, open-source tools for each pillar, stitched together.
  3. **OpenTelemetry-Based Unification:** Using the OpenTelemetry standards and collector as the foundation, building custom storage and query layers on top.
- **Decision:** We will build an **OpenTelemetry-Based Unification** platform.
- **Rationale:** Commercial SaaS is a black box unsuitable for learning. DIY point solutions teach the individual tools but leave the hardest problem—correlation—unaddressed. OpenTelemetry provides a vendor-neutral, standard data model (OTLP) and semantic conventions that solve the initial data unification problem, allowing us to focus our learning efforts on the core architectural challenge: building correlated storage and query layers. It also ensures our platform can ingest data from a vast ecosystem of instrumented applications.
- **Consequences:** We must implement the OTLP protocol receivers and adhere to OpenTelemetry semantic conventions. Our internal data model will mirror the OTLP protobuf definitions. We gain interoperability but accept some initial complexity in understanding the OTLP schema.

The following table summarizes the key trade-offs:

Approach	Pros	Cons	Why Not Chosen for This Project
Commercial SaaS (e.g., Datadog)	· **Fully Integrated:** Logs, metrics, traces, and profiling in one UI with automatic correlation. · **Zero Operations:** No infrastructure to manage. · **Advanced Features:** Machine learning, sophisticated alerting, out-of-the-box dashboards.	· **High Cost:** Can be prohibitively expensive at scale. · **Vendor Lock-in:** Data format, APIs, and query language are proprietary. · **Black Box:** Limited ability to customize storage, query engines, or data processing pipelines.	The goal of this project is to **learn and build**, not to consume a service. The proprietary nature and operational simplicity obscure the underlying architectural principles we aim to master.
DIY Point Solutions (Prometheus + ELK + Jaeger)	· **Best-of-Breed:** Each tool is highly optimized for its specific signal type. · **Open Source:** Full control and visibility into each component. · **Community Support:** Large ecosystems and extensive documentation.	· **Correlation is Bolted-On:** Linking traces to logs and metrics is manual, fragile, and often incomplete. Requires careful instrumentation and sidecar configurations. · **Operational Overhead:** Running and scaling three distinct complex systems. · **Inconsistent APIs:** Each tool has its own query language, data model, and storage engine.	This approach teaches you how to operate three excellent tools, but it **sidesteps the fundamental architectural problem** of designing a system where correlation is intrinsic. The "glue" between these systems is often an afterthought, which is the very challenge we want to tackle head-on.
OpenTelemetry-Based Unification	· **Unified Data Model:** OTLP provides a single schema for logs, metrics, and traces with built-in correlation fields (`trace_id`, `span_id`, exemplars). · **Vendor-Neutral:** Avoids lock-in; data can be sent to any OTLP-compatible backend. · **Focus on Core Challenge:** Lets us concentrate on the storage, indexing, and query problems, not on defining a data model.	· **Immature Storage Backends:** The OpenTelemetry Collector is primarily a pipeline; mature, unified long-term storage backends are still emerging. · **Implementation Complexity:** Requires building significant components from scratch (storage engines, query planner).	**This is our chosen approach.** It provides the perfect foundation for a learning project: a robust, standard data ingestion layer that forces us to solve the interesting problems of multi-signal storage and correlated query execution. The "cons" are precisely the learning opportunities we seek.

⚠️ **Pitfall: Underestimating Data Model Complexity** A common mistake when starting an observability platform is to design simplistic, independent schemas for logs, metrics, and traces. For example, storing logs as plain text files, metrics as plain `(timestamp, value)` tuples, and traces as JSON documents. This makes later correlation nearly impossible. The decision to adopt the OpenTelemetry data model from the start is critical. It forces us to think about **resources**, **attributes**, and **correlation IDs** as fundamental concepts, preventing a costly redesign later.

⚠️ **Pitfall: Ignoring Cardinality in Early Design** Another early mistake is not considering the explosive effect of high-cardinality dimensions (like unique user IDs, trace IDs, or request IDs) on metric systems. A design that works for 100 services will collapse under 10,000. While our initial implementation won't handle web-scale, the storage designs for metrics and the cross-signal index must be crafted with cardinality constraints in mind, using techniques like indexing choices and selective aggregation.

The path forward, therefore, is clear. We will build a unified observability platform using OpenTelemetry as our lingua franca for telemetry data. The subsequent sections of this document detail the architecture, data models, and component designs that will transform this standardized stream of correlated data into a powerful tool for understanding complex systems.

> **Milestone(s):** Milestone 1, Milestone 2, Milestone 3, Milestone 4, Milestone 5

2. Goals and Non-Goals

Before diving into architecture, it's crucial to define what success looks like for this platform and, equally important, what's explicitly out of scope. Building an observability platform is a massive undertaking—commercial solutions like Datadog or Splunk represent thousands of engineering years. This project is a learning exercise to understand the core principles, not to build a production-ready competitor. Therefore, we must be ruthlessly specific about our **Goals** (the essential capabilities we *must* implement to demonstrate understanding) and **Non-Goals** (features we explicitly choose *not* to build to stay focused and complete the project in a reasonable timeframe).

Think of this as a construction blueprint for a **learning skyscraper**. The Goals are the steel beams and concrete pillars that define the core structural integrity—without them, the building collapses. The Non-Goals are the marble facades, rooftop gardens, and underground parking garages—desirable, but we can build a functional, instructive structure without them. This clarity prevents "feature creep" and ensures we invest energy where it yields the most learning value.

Must Achieve (Goals)

Our goals are derived directly from the five milestones' acceptance criteria. They represent the **minimum viable product (MVP)** for a unified observability platform that demonstrates the key architectural concepts. Each goal is a concrete, testable capability.

> **Critical Insight:** A successful implementation of these goals will result in a functional system where you can ingest telemetry from a sample application, query across logs, metrics, and traces to debug an issue, and receive an alert when something goes wrong. It won't be pretty, scalable to petabytes, or enterprise-grade—but it will *work* and teach you *how*.

We categorize goals into **Functional** (what the system does) and **Non-Functional** (how well it does it, within our learning constraints).

Functional Goals

Goal ID	Description	Derived From Milestone	Key Success Metric
F1	**Unified Data Correlation**	Milestone 1	A user can query all logs for a given `trace_id` and see metric exemplars linked to trace IDs.
F2	**Multi-Signal Ingestion**	Milestone 2	The system accepts logs, metrics, and traces via the OTLP/gRPC protocol and normalizes them to a common model.
F3	**Specialized Storage**	Milestone 3	Logs are full-text searchable, metrics are queryable as time series, and traces can be reconstructed into full trees.
F4	**Cross-Signal Query Interface**	Milestone 4	A single query API can return correlated data (e.g., "show me logs and traces for slow requests from service X").
F5	**Multi-Signal Alerting**	Milestone 5	An alert can be defined using conditions across signals (e.g., "high error rate AND increased latency") and trigger notifications.

Let's expand on each functional goal with concrete, testable behaviors.

F1: Unified Data Correlation

- **Shared Resource Attributes:** Every piece of telemetry (log line, metric data point, span) must be tagged with a common set of resource attributes (`service.name`, `service.version`, `service.instance.id`, `deployment.environment`). This answers the question "Who sent this?"
- **Trace-Log Linking:** A log record emitted during the execution of a span must include the span's `trace_id` and `span_id` as attributes. The storage and query systems must allow efficient retrieval of logs by `trace_id`.
- **Metric-Trace Linking (Exemplars):** A histogram or summary metric data point can include an **exemplar**—a reference to a specific trace (`trace_id`) that represents an observed measurement (e.g., a particularly slow request that contributed to the 99th percentile bucket).
- **Compatibility:** The attribute naming and semantics should follow OpenTelemetry Semantic Conventions where practical, to ensure the platform can ingest data from standard instrumentation libraries.

F2: Multi-Signal Ingestion

- **OTLP Receiver:** The primary ingestion endpoint must implement the OpenTelemetry Protocol (OTLP) over gRPC, as defined by the OTLP specification. It must handle `ExportTraceServiceRequest`, `ExportMetricsServiceRequest`, and `ExportLogsServiceRequest` messages.
- **Batch Processing:** Incoming data must be buffered in configurable-sized batches (e.g., 1000 spans, 5MB) before being written to storage to amortize I/O overhead.
- **Normalization:** The pipeline must transform incoming data into the platform's unified internal data model, ensuring consistent timestamp formats, attribute naming, and value types (e.g., converting all strings to UTF-8, normalizing numeric types).
- **Backpressure:** When the ingestion pipeline is saturated (e.g., memory buffers are full, storage is slow), it must signal backpressure to clients. For gRPC, this means returning an appropriate error code (like `UNAVAILABLE` or `RESOURCE_EXHAUSTED`) to cause clients to retry with exponential backoff, preventing the pipeline from collapsing under load.

F3: Specialized Storage

- **Log Storage:** Must support efficient full-text search over log body fields and structured attribute filtering. For a dataset of 1 million log records, a simple keyword search should return results within 500ms on commodity hardware (e.g., a laptop with an SSD). This implies the use of an inverted index.
- **Metric Storage:** Must handle time-series data with potentially high cardinality (many unique time series, defined by combinations of metric name and attributes). It must support configurable retention policies (e.g., keep raw data for 7 days, keep 1-hour downsampled averages for 30 days) and automatic cleanup of expired data.
- **Trace Storage:** Must store individual spans and be able to efficiently reconstruct the complete directed acyclic graph (DAG) of spans that form a trace. It must support queries by `trace_id`, attributes, and duration.
- **Cross-Signal Index:** A secondary index must exist to enable efficient cross-signal queries. At minimum, it should map `trace_id` to the storage locations (e.g., document IDs) of related log records and metric exemplars.

F4: Cross-Signal Query Interface

- **Unified Query Language:** A simple, JSON- or Protobuf-based query language must allow filtering by time range, resource attributes (e.g., `service.name="frontend"`), and signal-specific attributes (e.g., `log.severity="ERROR"`, `span.duration > 200ms`).
- **Correlation Queries:** The API must support queries that join data across storage backends in a single request. For example: `find traces where duration > 1s AND service.name="api"` and, for each resulting trace, fetch the associated error logs.
- **Pagination:** Results must be paginated using a cursor-based mechanism to handle large result sets without overwhelming the client or server.
- **Aggregations:** For metrics, the query API must support aggregations like `rate()`, `sum()`, `avg()`, and percentile calculations (e.g., `p99()`).

F5: Multi-Signal Alerting

- **Rule Definition:** Users must be able to define alert rules that evaluate conditions against the query engine. Rules should support thresholds (e.g., `error_rate > 5%`) and simple boolean logic across signals (e.g., `high_error_rate AND increased_latency`).
- **Anomaly Detection (Basic):** The system must include a simple statistical anomaly detector, such as detecting when a metric deviates significantly (e.g., > 3 standard deviations) from a rolling historical baseline (e.g., the average of the last 1 hour).
- **Notification Routing:** Alerts must be routable to different channels (e.g., stdout, a webhook to a service like Slack) based on configurable routing rules (e.g., severity, team).
- **Deduplication:** Repeated evaluations of the same alert condition for the same underlying issue (identified by a fingerprint of the alert labels) should not generate duplicate notifications. A common approach is to require the condition to be true for a "for" duration before firing, and then suppress further notifications until the alert resolves.

Non-Functional Goals (Quality Attributes)

Goal ID	Description	Rationale & Target
NF1	**Educational Clarity**	The primary purpose is learning. Code should be well-structured, commented, and avoid unnecessary optimization that obfuscates core concepts.
NF2	**Functional Completeness Over Scale**	We prioritize implementing all core features over handling massive scale. The system should work end-to-end with a single instance and moderate load (100s of requests/sec, GBs of data).
NF3	**Operational Simplicity**	The system should be easy to run locally for development and testing. A single binary or a simple `docker-compose` setup is ideal.
NF4	**Performance Thresholds**	Meet the specific performance criteria called out in the acceptance criteria (e.g., 10k events/sec ingestion, 500ms log search on 1M records) as proof of correct algorithm and data structure choices, not as production SLAs.
NF5	**Resilience to Naïve Failures**	The system should not lose data or crash due to simple mistakes like a malformed batch or a full disk. It should implement basic resilience patterns (backpressure, retry queues, graceful shutdown).

Will Not Do (Non-Goals)

Explicitly stating what we will **not** build is a powerful tool to manage scope and set clear expectations. These are features that, while valuable in a commercial product, would add disproportionate complexity for our learning objectives. Choosing these non-goals allows us to focus our limited time on the architectural fundamentals.

> **Warning:** The temptation to add "just one more feature" is strong. Refer to this list when that temptation arises. Each item here represents a potential multi-month project in itself.

Non-Goal	Why It's Out of Scope	What We Do Instead
Production-Grade Scalability & High Availability	Building a horizontally scalable, multi-tenant, fault-tolerant cluster is a massive distributed systems challenge that would dwarf the core observability lessons.	Our system is designed to run as a single instance. We'll use simple, single-node storage engines (like local BoltDB, Badger, or plain files with indexes). We focus on the *algorithms* of indexing and query, not their distributed coordination.
Advanced User Interface (UI)	Building a rich, interactive UI like Jaeger or Grafana is a major front-end project that distracts from the backend architecture we're learning.	We will provide a **command-line interface (CLI)** and a **simple REST/GRPC API** for all operations. For visualization, we can output JSON that can be piped to `jq` or plotted with simple scripts.
Machine Learning for Root Cause Analysis	ML-based anomaly detection and root cause identification is a cutting-edge research area requiring large datasets and specialized expertise.	We implement **simple statistical baselines** (rolling average, standard deviation) for anomaly detection. Root cause analysis is left to the human operator using our correlation tools.
Long-Term Storage Tiering & Archival	Implementing automated data migration from hot to cold storage (e.g., SSD to S3 to Glacier) involves complex lifecycle management and is more of a storage systems problem.	We implement configurable retention with **deletion**. Data older than the retention period is simply deleted.
Advanced Sampling Strategies	Intelligent head- or tail-based sampling of traces is crucial for production cost control but adds significant complexity to the ingestion and storage layers.	We ingest **everything** sent to us. For learning, this is simpler and ensures we have complete data for debugging our own platform.
Custom Instrumentation Agents/SDKs	Writing client libraries for various languages to generate telemetry is a separate domain from building the backend platform.	We rely entirely on **OpenTelemetry** as the instrumentation standard. Our platform accepts OTLP, and users instrument their apps using the official OpenTelemetry SDKs.
Real-time Streaming Analytics	Complex event processing or continuous streaming queries (like Apache Flink) is a different paradigm from store-then-query.	Our query engine operates on **stored data**. Alerts are evaluated on a periodic polling basis (e.g., every 30 seconds).
Advanced Security (RBAC, Encryption, Audit Logs)	Full-featured security is critical for multi-tenant SaaS but adds enormous complexity for authentication, authorization, and encryption at rest/in transit.	Our platform assumes a **trusted environment** (e.g., running inside a secure network). We may add basic API key authentication if needed for integration demos, but not full RBAC.
Support for Legacy Protocols	Supporting every vendor's proprietary protocol (e.g., StatsD, Zipkin v1, Splunk HEC) turns the ingestion pipeline into a compatibility layer, obscuring the core design.	We support **OTLP as the primary protocol**. If needed, the OpenTelemetry Collector can be run as a sidecar to translate legacy formats into OTLP for our platform.

Architecture Decision Record: Scoping for Educational Value

> **Decision: Prioritize End-to-End Completeness of Core Concepts Over Production Features**
>
> - **Context:** We are building a learning platform to understand observability architecture, not a commercial product. Engineering time and cognitive load are limited resources.
> - **Options Considered:**
>   1. **Build a Minimal but Complete System:** Implement the five pillars (unified model, ingestion, storage, query, alerting) with simple, single-node components that work together.
>   2. **Build a Scalable but Narrow System:** Deep-dive into one area (e.g., building a distributed time-series database) at the expense of not implementing other pillars like logs or traces.
>   3. **Build a Feature-Rich but Shallow System:** Implement many features (UI, multiple protocols, sampling) but only as superficial, non-integrated components.
> - **Decision:** Choose Option 1. We will build a complete, integrated pipeline from ingestion to alerting for all three signals, using simple but educationally illustrative implementations.
> - **Rationale:** The deepest learning comes from understanding how the pieces fit together—how a trace ID flows from a span to a log to a metric exemplar, and how a query can stitch them back. A narrow deep-dive (Option 2) misses these integration insights. A shallow system (Option 3) teaches API design but not the hard problems of stateful storage and efficient query execution. Our chosen path maximizes architectural learning per unit of implementation time.
> - **Consequences:** The resulting platform will be functional for small-scale, non-critical use (e.g., personal projects, internal dev tools) and will perfectly demonstrate architectural principles. It will not be suitable for production workloads without significant re-engineering for scale and reliability—which is an excellent learning exercise for the *next* project.

Option	Pros	Cons	Chosen?
Minimal but Complete	Teaches system integration, end-to-end data flow, and core algorithms for all three signals. Achievable in a bounded time.	Lacks production-grade scale, robustness, and polish.	**Yes**
Scalable but Narrow	Allows deep understanding of one complex subproblem (e.g., distributed consensus in a TSDB).	Fails to teach the unified observability vision and correlation across signals.	No
Feature-Rich but Shallow	Demonstrates a wide API surface and many features.	Risk of becoming a "kitchen sink" of copy-pasted code without deep understanding of underlying mechanics.	No

Common Pitfalls in Defining Scope

⚠️ Pitfall: Underestimating the Ingestion Pipeline

- **The Mistake:** Treating the ingestion pipeline as a trivial "receive and forward" component, leading to a system that cannot handle even moderate load, loses data on restart, or blocks clients indefinitely.
- **Why It's Wrong:** The ingestion pipeline is the front door to your system. If it's unreliable or inefficient, nothing else matters. Real observability platforms spend enormous effort here.
- **How to Avoid:** Design the ingestion pipeline as a first-class component with its own ADRs for batching, buffering, backpressure, and durability (e.g., a Write-Ahead Log). Implement it before you build fancy storage engines.

⚠️ Pitfall: Over-Engineering Storage on Day One

- **The Mistake:** Starting by trying to build a distributed, compressed, columnar time-series database because "that's what Prometheus does."
- **Why It's Wrong:** You'll get bogged down in distributed systems problems before you even have a working query. The learning goal is to understand *why* time-series data needs special treatment, not to replicate a decade of Prometheus development.
- **How to Avoid:** Start with the simplest possible storage that meets the functional goal (e.g., store metrics as JSON files per series, or use a simple key-value store). Get the entire pipeline working first. *Then* iterate on storage efficiency as a later optimization exercise.

⚠️ Pitfall: Neglecting the Cross-Signal Index

- **The Mistake:** Storing logs, metrics, and traces in three separate silos with no way to efficiently find relationships between them, making the "unified" platform no better than three separate tools.
- **Why It's Wrong:** The entire value proposition of a unified platform is correlation. Without a dedicated index to map `trace_id` to related data, cross-signal queries become slow, full-table scans that are unusable at any scale.
- **How to Avoid:** Design the correlation index (often called a **reverse index** or **correlation store**) as a core component from the beginning. Every time you write a log with a `trace_id` or a metric with an exemplar, you must also write an entry to this index.

⚠️ Pitfall: Building a UI Instead of a Robust API

- **The Mistake:** Spending weeks building a React frontend to visualize traces before having a stable, queryable trace storage backend.
- **Why It's Wrong:** The UI depends entirely on the backend API. If the API is unstable or inefficient, the UI will be broken and frustrating. Furthermore, a CLI/API-first approach forces you to design a clean, machine-readable interface, which is a valuable skill.
- **How to Avoid:** Commit to an **API-first development process**. Define the query API's Protobuf/JSON schemas first. Use `curl` or simple Go scripts to test your backend. Only consider a basic UI after the entire backend system is functional.

By adhering to these Goals and respecting these Non-Goals, you will construct a platform that is both **instructive** and **functional**. You will grapple with the real architectural tensions of observability—schema design, ingestion performance, storage specialization, and query efficiency—without being overwhelmed by the infinite complexity of a commercial-grade system.

---

3. High-Level Architecture

> **Milestone(s):** Milestone 1 (Unified Data Model), Milestone 2 (Data Ingestion Pipeline), Milestone 3 (Multi-Signal Storage), Milestone 4 (Unified Query Interface), Milestone 5 (Alerting & Anomaly Detection)

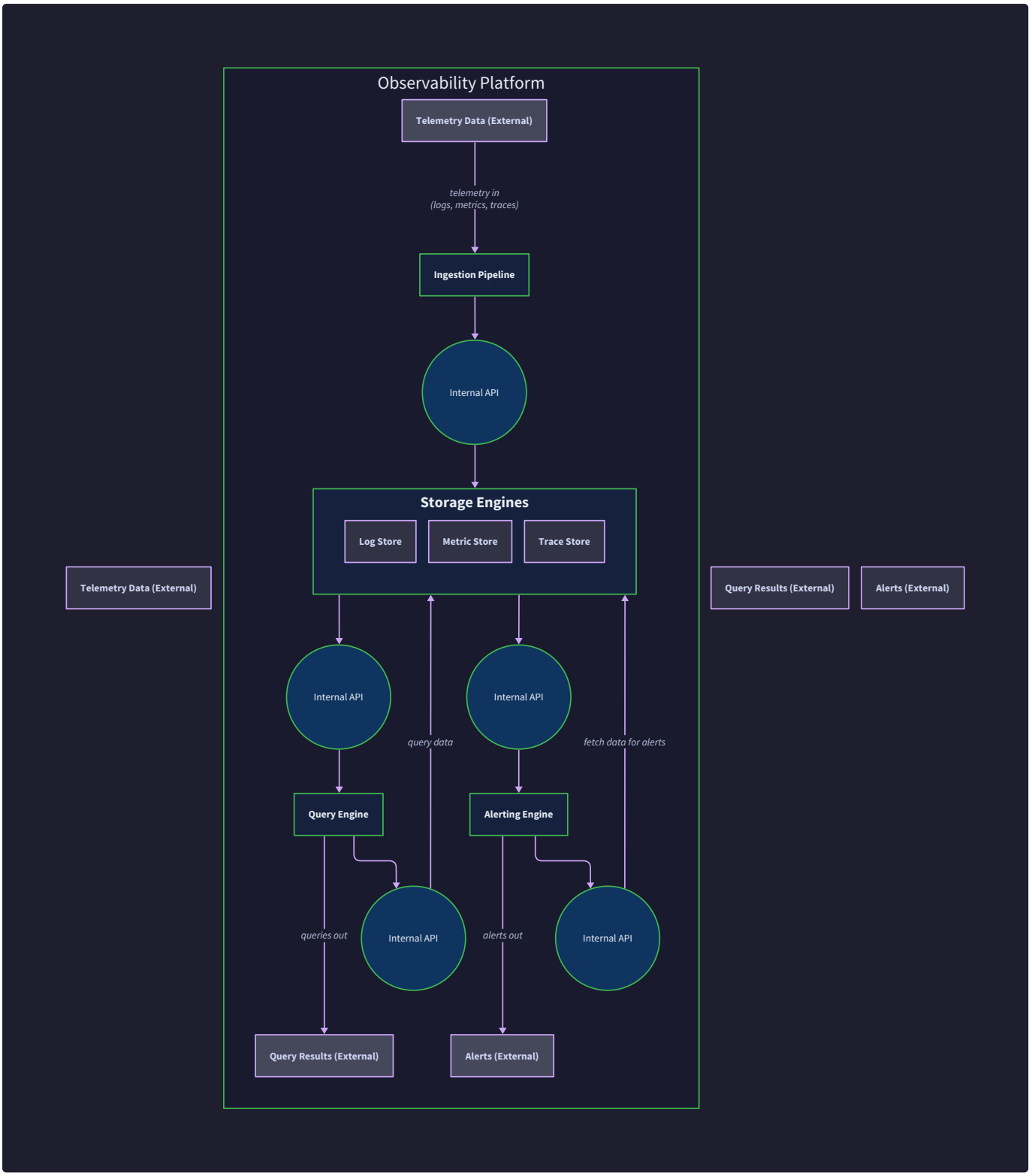
This section presents the architectural blueprint of the observability platform. We'll begin by establishing an intuitive mental model to frame the system's purpose and flow, then translate that model into concrete software components and their interactions. Finally, we'll provide a recommended Go module structure to organize the codebase effectively, separating concerns and establishing clear boundaries.

The Telemetry Refinery: A Mental Model

Imagine a massive oil refinery on the coast. Tanker ships (your applications) continuously arrive, each carrying a different type of crude raw material: one ship carries a light, volatile liquid (traces), another a thicker, continuous stream (metrics), and a third a chunky mixture of solid logs and text (logs). These raw materials cannot be used effectively in their delivered state. The refinery's job is to:

1. **Dock and Unload (Ingestion):** Provide specialized berths (`OTLP_GRPC_PORT`) where each tanker can safely and efficiently offload its cargo. The dock has traffic signals (backpressure) to tell arriving ships to slow down or wait if the internal processing is congested.
2. **Separate and Refine (Processing):** Pipe each raw material to a dedicated processing unit. The light trace liquid is distilled into individual spans and relationships. The metric stream is cleaned and concentrated. The chunky log mixture is broken down, with valuable keywords and fields extracted.
3. **Store in Specialized Tanks (Storage):** Pump the refined products into vast, purpose-built storage tanks. The volatile traces go into a tank designed for quick retrieval of complex relationships (a graph archive). The continuous metrics go into a tank built for compressing and aggregating time-series data. The processed logs go into a warehouse with a sophisticated indexing system for text search.
4. **Central Catalog (Cross-Signal Index):** As materials are stored, an automated system creates a central catalog. It notes that a particular barrel of trace liquid (a `trace_id`) is related to specific batches of logs and metric samples. This catalog doesn't store the data itself but points to its location in the specialized tanks.
5. **Control Tower and Distribution (Query & Alerting):** The refinery's control tower (`QueryEngine`) accepts orders from customers. A customer might ask, "Find all log debris related to the trace shipment `abc123` and show me the metric pressure readings from that same time." The tower consults the central catalog, dispatches forklifts to the relevant tanks to fetch the data, and assembles a unified report. Separate watchtowers (`AlertingEngine`) constantly monitor gauge readings (metrics) and sensor patterns, sounding alarms and routing notifications when predefined conditions are met.

This **Telemetry Refinery** model emphasizes the core architectural principle: *specialization of function*. Logs, metrics, and traces have fundamentally different shapes, write patterns, and query patterns. Attempting to store them in one monolithic database leads to severe inefficiency. Instead, we accept their differences, build optimized pathways and storage for each, and invest in a robust correlation layer (the catalog and control tower) to reunite them when needed for holistic understanding.



The diagram above visualizes this model as software components. Data flows from left to right: raw telemetry enters the **Ingestion Pipeline**, is processed and stored in specialized **Storage Engines**, and can later be retrieved via the **Query Engine**. The **Alerting Engine** operates in a loop, periodically querying the stored data to detect issues.

Core Component Responsibilities

Component	Responsibility & Mental Model Analog	Key Data it Owns	Key Operations
Ingestion Pipeline	The loading dock and initial processing line. Accepts deliveries, checks manifests, and routes goods to the correct warehouse.	In-flight batches in memory buffers.	`ReceiveOTLP(request)`, batch buffering, normalization, applying backpressure.
Storage Engines (Log, Metric, Trace)	Specialized warehouses. Each is architecturally optimized for its cargo type (full-text search, time-series compression, graph relations).	All persisted telemetry data for its signal type on disk.	Write, index, query, compact, and enforce retention policies.
Cross-Signal Index	The central catalog or ledger. A secondary index that maps correlation keys (like `trace_id`) to data locations across warehouses.	Mapping tables (e.g., `trace_id` → list of log record IDs, metric exemplar timestamps).	Update on ingest, query for correlations.
Query Engine	The control tower and dispatch office. Accepts complex orders, breaks them down, fetches from warehouses, and assembles the final report.	Query execution plans, pagination cursors.	`ExecuteQuery(query)`, planning, fan-out to storage engines, result merging.
Alerting Engine	The network of watchtowers and alarm systems. Continuously scans the horizon (data) for smoke (anomalies) according to predefined rules.	`AlertRule` definitions, alert state (firing, resolved).	`EvaluateRules()`, anomaly detection, notification routing, deduplication.

Key Architectural Decisions and Data Flow

> **Decision: Specialized Storage with Unified Query Front-End**
>
> - **Context**: We must handle three data types with conflicting optimal storage layouts: append-only logs for text search, regular time-series for compression, and tree-structured traces for graph traversal.
> - **Options Considered**:
>   1. **Monolithic Unified Store**: A single database (e.g., extended time-series DB) attempting to store all signal types.
>   2. **Polyglot Persistence**: Separate, best-of-breed storage engines for each signal, glued by a query layer.
> - **Decision**: Polyglot Persistence (Specialized Storage Engines).
> - **Rationale**: A monolithic store forces one data model to accommodate all, leading to severe trade-offs (e.g., poor text search performance in a TSDB, or inefficient compression for metrics in a document store). Polyglot persistence allows us to use optimal data structures (inverted indexes, Gorilla compression, span trees) for each signal. The complexity of coordination is moved to the query layer, which is a known and solvable problem.
> - **Consequences**: **Enables** high performance and cost-effective storage for each signal type. **Introduces** operational complexity (managing multiple storage systems), and requires a sophisticated query engine to perform cross-signal joins.

The flow of data through the system follows two primary pathways:

1. **The Ingestion Path**: `Application` → (`OTLP/gRPC`) → `IngestionPipeline` (Receiver → Batcher → Normalizer) → `Storage Engines` & `Cross-Signal Index`.
2. **The Query Path**: `User/AlertingEngine` → (`HTTP/gRPC`) → `QueryEngine` (Parser → Planner → Executor) → `Cross-Signal Index` & `Storage Engines` → Merged Results.

The **Cross-Signal Index** is updated synchronously during ingestion to ensure correlations are immediately queryable. The **Alerting Engine** is a background process that periodically invokes the `QueryEngine` to evaluate conditions.

Recommended File & Module Structure

A well-organized codebase mirrors the architecture, making components clear, boundaries explicit, and dependencies manageable. Below is a recommended structure for a Go project implementing our Telemetry Refinery.

```
observability-platform/
├── cmd/
│   ├── ingestion/          # Entry point for the ingestion pipeline service
│   │   └── main.go
│   ├── query/              # Entry point for the query API service
│   │   └── main.go
│   └── alerting/           # Entry point for the alerting daemon service
│       └── main.go
├── internal/               # Private application code, not importable by others
│   ├── ingestion/
│   │   ├── receiver/
│   │   │   ├── otlp_grpc.go    # Implements `ReceiveOTLP`
│   │   │   └── otlp_http.go
│   │   ├── batcher/
│   │   │   └── batcher.go      # Manages buffers up to `MAX_BATCH_SIZE`
│   │   ├── normalizer/
│   │   │   └── normalizer.go   # Applies unified data model schema
│   │   └── pipeline.go         # Orchestrates receiver, batcher, normalizer
│   ├── storage/
│   │   ├── logstore/
│   │   │   ├── index/          # Full-text and field inverted index
│   │   │   ├── wal.go          # Write-Ahead Log
│   │   │   ├── segment.go      # Immutable data segment
│   │   │   └── engine.go       # Main log storage API
│   │   ├── metricstore/
│   │   │   ├── compression/    # Gorilla-style compression
│   │   │   ├── retention.go    # Enforces `DEFAULT_RETENTION_DAYS`
│   │   │   └── engine.go       # Main metric storage API
│   │   ├── tracestore/
│   │   │   ├── span_tree.go    # Logic for reconstructing trace trees
│   │   │   └── engine.go       # Main trace storage API
│   │   └── correlation/
│   │       └── index.go        # `Cross-Signal Index` implementation
│   ├── query/
│   │   ├── language/
│   │   │   └── parser.go       # Parses query string to AST
│   │   ├── planner/
│   │   │   └── planner.go      # Creates execution plan from AST
│   │   ├── executor/
│   │   │   └── executor.go     # Implements `ExecuteQuery`, fans out to storage
│   │   └── api/
│   │       └── server.go       # HTTP/gRPC API layer
│   ├── alerting/
│   │   ├── rule/
│   │   │   └── engine.go       # Implements `EvaluateRules`
│   │   ├── detector/
│   │   │   └── anomaly.go      # Statistical baseline anomaly detection
│   │   ├── notifier/
│   │   │   └── router.go       # Handles notification channels
│   │   └── dedupe/
│   │       └── dedupe.go       # Alert deduplication logic
│   └── pkg/
│       ├── models/             # Shared data structures (DO NOT put business logic here)
│       │   ├── resource.go     # `ResourceAttributes` struct
│       │   ├── log.go          # `LogRecord` struct
│       │   ├── trace.go        # `Span` struct
│       │   ├── metric.go       # `MetricPoint` and `Exemplar` structs
│       │   └── alert.go        # `AlertRule` struct
│       └── timeutil/           # Shared utilities (e.g., timestamp conversions)
├── api/                        # Protobuf/gRPC service definitions (.proto files)
│   └── v1/
├── configs/                    # Configuration files (YAML/JSON)
├── deployments/                # Dockerfiles, Kubernetes manifests
├── go.mod
├── go.sum
└── README.md
```

Rationale for Structure

- **`cmd/` per Service**: We anticipate three long-running processes: ingestion, query, and alerting. This follows the standard Go convention of one directory per binary. This allows independent scaling and deployment (e.g., you might run multiple ingestion workers).
- **`internal/` by Component**: The `internal` directory houses the core business logic, organized by architectural component (`ingestion/`, `storage/`, etc.). This prevents external packages from importing our internal logic, enforcing clean boundaries. The `pkg/models` subdirectory is an exception for widely shared data structures that have no logic.
- **Clear Dependencies**: Dependencies should flow in one direction: `ingestion` → `storage` and `models`. `query` → `storage` and `models`. `alerting` → `query`, `models`. `storage` engines only depend on `models`. This acyclic graph is crucial for maintainability.

- **Separated API Definitions**: The `api/` directory contains Protobuf files, cleanly separating the interface definition from its Go implementation. This makes it easier to generate client libraries for other languages.

Common Pitfalls in Project Structure

⚠️ Pitfall: The God Package

- **Description**: Putting all types and functions in a single `common` or `utils` package, or directly in `internal`. This creates a tangled web of dependencies and makes it impossible to reason about what parts of the system use what.
- **Why it's Wrong**: It violates separation of concerns, leads to import cycles, and makes testing difficult. Changes in a "common" package can have unpredictable side effects across the entire codebase.
- **How to Fix**: Adhere strictly to the component-based structure above. If a piece of logic is truly needed by multiple components, evaluate if it belongs in a shared, minimal utility package (`internal/pkg/`) or if the component boundaries need redesign.

⚠️ Pitfall: Business Logic in Data Models

- **Description**: Adding methods with complex business logic (e.g., `LogRecord.Index()`) to the simple data structure definitions in `pkg/models`.
- **Why it's Wrong**: It couples the pure data representation with specific storage or processing concerns. This makes the models harder to serialize/deserialize and breaks the single-responsibility principle.
- **How to Fix**: Keep `pkg/models` as **dumb data holders**. All logic that operates on these models (indexing, validation, transformation) should reside in the component packages (e.g., `internal/storage/logstore/index/`).

⚠️ Pitfall: Ignoring Internal Package Visibility

- **Description**: Making all subdirectories under `internal` export their main types (using uppercase names) and then importing `github.com/yourproject/internal/ingestion` from another project.
- **Why it's Wrong**: The Go compiler's `internal` directory rule prevents this, causing build failures. More importantly, it breaks encapsulation. Your platform should be interacted with via its defined APIs (gRPC/HTTP), not by directly importing internal libraries.
- **How to Fix**: Respect the `internal` mechanism. If you need to provide a library for others (e.g., a client SDK), place it in a separate, versioned module at the repository root or in a `client/` directory, outside `internal`.

Implementation Guidance

This section provides concrete, actionable steps to start implementing the high-level architecture in Go.

A. Technology Recommendations Table

Component	Simple Option (For Learning/Early Milestones)	Advanced Option (For Production/Scale)
Transport (Ingestion)	HTTP/JSON over standard `net/http`	**gRPC with Protocol Buffers** (OTLP standard)
Transport (Query API)	HTTP REST + JSON (`net/http`)	gRPC-Web + HTTP/JSON for browser compatibility
Internal Communication	Direct function calls / in-memory channels	gRPC for inter-service communication if components are split
Log Storage Index	In-memory map for field indexes, simple full-text tokenization	Bleve or Tantivy (Rust) bindings for production-grade full-text search
Metric Storage	In-memory map of slices per series, periodic flush to disk	Custom columnar format with Gorilla compression, or embed Prometheus TSDB
Trace Storage	Store spans as JSON/Protobuf in key-value store by `trace_id`	Native graph database (e.g., Neo4j) or custom adjacency list format
Cross-Signal Index	In-memory `map[string][]string` (`trace_id` → record IDs)	Persistent key-value store (e.g., BadgerDB) with range queries
Alerting Evaluation	Periodic goroutine scanning in-memory data	Time-series database with continuous query support (e.g., PromQL)

For this project, we recommend starting with the **Simple Options** to validate the architecture and data flow, then iterating towards the **Advanced Options** as you hit scaling limits.

B. Recommended Starter Code and Module Layout

To begin, create the foundational module structure and a minimal, runnable ingestion point.

1. Initialize the Go Module:

```bash
mkdir observability-platform

cd observability-platform

go mod init github.com/yourusername/observability-platform
```

2. Create the Core Data Models (`internal/pkg/models/`): Create the directory and the following files. These are *complete* starter definitions.

```go
// internal/pkg/models/resource.go

package models


// ResourceAttributes identifies the source of telemetry data.

type ResourceAttributes struct {

        ServiceName    string `json:"service_name"`

        ServiceVersion string `json:"service_version"`

        InstanceID     string `json:"instance_id"`

        Environment    string `json:"environment"`

}
```

```go
// internal/pkg/models/log.go

package models


import "time"


// LogRecord represents a single log entry.

type LogRecord struct {

        Timestamp  int64             `json:"timestamp"`   // Unix nanoseconds

        Body       string            `json:"body"`        // The raw log message

        Severity   string            `json:"severity"`    // e.g., "INFO", "ERROR"

        TraceID    string            `json:"trace_id"`    // Optional: for correlation

        SpanID     string            `json:"span_id"`     // Optional: for correlation

        Resource   ResourceAttributes `json:"resource"`

        Attributes map[string]string `json:"attributes"` // Key-value pairs

}


// Time returns the timestamp as a Go time.Time.

func (l *LogRecord) Time() time.Time {

        return time.Unix(0, l.Timestamp)

}
```

```go
// internal/pkg/models/trace.go

package models

import "time"

// Span represents a single operation within a trace.
type Span struct {
	TraceID      string             `json:"trace_id"`
	SpanID       string             `json:"span_id"`
	ParentSpanID string             `json:"parent_span_id"` // Empty for root spans
	Name         string             `json:"name"`
	StartTime    int64              `json:"start_time"` // Unix nanoseconds
	EndTime      int64              `json:"end_time"`   // Unix nanoseconds
	Resource     ResourceAttributes `json:"resource"`
	Attributes   map[string]string  `json:"attributes"`
}

// Duration returns the span's duration.
func (s *Span) Duration() time.Duration {
	return time.Duration(s.EndTime - s.StartTime)
}
```

GO

```go
// internal/pkg/models/metric.go

package models

// Exemplar links a metric sample to a specific trace.
type Exemplar struct {
	TraceID   string  `json:"trace_id"`
	Value     float64 `json:"value"`
	Timestamp int64   `json:"timestamp"` // Unix nanoseconds
}

// MetricPoint represents a single measurement at a point in time.
type MetricPoint struct {
	Name      string             `json:"name"`
	Value     float64            `json:"value"`
	Timestamp int64              `json:"timestamp"` // Unix nanoseconds
	Exemplar  *Exemplar          `json:"exemplar,omitempty"` // Optional
	Resource  ResourceAttributes `json:"resource"`
	Attributes map[string]string `json:"attributes"`
}
```

GO

```go
// internal/pkg/models/alert.go

package models


// AlertRule defines a condition that triggers an alert.

type AlertRule struct {

        Name       string            `json:"name"`

        Condition  string            `json:"condition"`     // e.g., "error_rate > 0.05"

        ForDuration string           `json:"for_duration"`  // e.g., "5m"

        Severity   string            `json:"severity"`      // e.g., "critical", "warning"

        Labels     map[string]string `json:"labels"`        // Attached to fired alerts

        Annotations map[string]string `json:"annotations"`  // Descriptive text

}
```

3. Create a Minimal Ingestion Service Skeleton (`cmd/ingestion/main.go`): This is a complete, runnable HTTP server that accepts JSON logs as a starting point. It demonstrates the ingestion flow.

GO

```go
// cmd/ingestion/main.go

package main


import (

	"encoding/json"

	"fmt"

	"log"

	"net/http"

	"sync"


	"github.com/yourusername/observability-platform/internal/pkg/models"

)


// batchBuffer holds incoming records before they are processed.

type batchBuffer struct {

	mu      sync.Mutex

	logs    []models.LogRecord

	maxSize int

}


func newBatchBuffer(maxSize int) *batchBuffer {

	return &batchBuffer{

		logs:    make([]models.LogRecord, 0, maxSize),

		maxSize: maxSize,

	}

}


func (b *batchBuffer) add(log models.LogRecord) bool {

	b.mu.Lock()

	defer b.mu.Unlock()

	if len(b.logs) >= b.maxSize {

		return false // Buffer full, applying backpressure

	}

	b.logs = append(b.logs, log)

	return true

}


func (b *batchBuffer) flush() []models.LogRecord {

	b.mu.Lock()

	defer b.mu.Unlock()

	flushed := b.logs

	b.logs = make([]models.LogRecord, 0, b.maxSize)

	return flushed
```

```
}

func main() {
	const maxBatchSize = 1000 // MAX_BATCH_SIZE
	buffer := newBatchBuffer(maxBatchSize)

	http.HandleFunc("/ingest/log", func(w http.ResponseWriter, r *http.Request) {
		if r.Method != http.MethodPost {
			http.Error(w, "Method not allowed", http.StatusMethodNotAllowed)
			return
		}

		var logRec models.LogRecord
		if err := json.NewDecoder(r.Body).Decode(&logRec); err != nil {
			http.Error(w, "Bad request", http.StatusBadRequest)
			return
		}

		// TODO: Normalize timestamp, resource attributes here.

		if ok := buffer.add(logRec); !ok {
			// Backpressure: buffer full
			http.Error(w, "Ingestion too busy, try again later", http.StatusTooManyRequests)
			return
		}

		w.WriteHeader(http.StatusAccepted)
		fmt.Fprintf(w, "Log accepted")
	})

	// Start a background goroutine to process batches.
	go func() {
		for {
			// In a real implementation, this would be triggered by size or time.
			// For now, we just log and clear periodically.
			batchedLogs := buffer.flush()
			if len(batchedLogs) > 0 {
				log.Printf("Processing batch of %d logs", len(batchedLogs))
				// TODO: Send batch to storage engine and cross-signal index.
				for _, l := range batchedLogs {
					_ = l // Placeholder for processing
				}
			}
		}
```

```
        }()

        log.Println("Starting ingestion server on :8080")
        if err := http.ListenAndServe(":8080", nil); err != nil {
                log.Fatal(err)
        }
}
```

4. Core Component Skeleton (`internal/ingestion/pipeline.go`): This is the skeleton for the main ingestion orchestration logic. The learner will fill in the TODOs.

```go
// internal/ingestion/pipeline.go

package ingestion


import (

        "context"

        "fmt"


        "github.com/yourusername/observability-platform/internal/pkg/models"

)


// Pipeline orchestrates the ingestion flow: receive, batch, normalize, store.

type Pipeline struct {

        // TODO: Add fields for components (receiver, batcher, normalizer, storage clients)

}


// NewPipeline creates a new ingestion pipeline.

func NewPipeline() *Pipeline {

        return &Pipeline{

                // TODO: Initialize all sub-components with configuration.

        }

}


// ProcessLogBatch is the core method for handling a batch of logs.

// This is called by the receiver after batching.

func (p *Pipeline) ProcessLogBatch(ctx context.Context, logs []models.LogRecord) error {

        // TODO 1: Normalize timestamps (ensure they are in nanoseconds, UTC).

        // TODO 2: Enrich or validate resource attributes (ensure service_name is present).

        // TODO 3: For each log, extract correlation IDs (trace_id, span_id) for indexing.

        // TODO 4: Send the normalized batch to the Log Storage Engine.

        // TODO 5: Update the Cross-Signal Index with the log IDs and their trace IDs.

        // TODO 6: Handle any errors from storage, potentially implementing retries.

        fmt.Printf("Processing batch of %d logs\n", len(logs)) // Placeholder

        return nil

}


// ProcessMetricBatch and ProcessTraceBatch would have similar skeletons.
```

C. Language-Specific Hints (Go)

- **Concurrency**: Use `sync.Mutex` or `sync.RWMutex` for protecting in-memory structures like the `batchBuffer`. For pipeline stages, consider using channels (`chan []models.LogRecord`) to pass data between goroutines, which provides natural backpressure.
- **Time**: Always store timestamps as **int64 Unix nanoseconds** (the OpenTelemetry standard). Use `time.Unix(0, nanoseconds)` to convert to `time.Time`. Use `time.Now().UnixNano()` to get the current timestamp.
- **Error Handling**: In the ingestion pipeline, distinguish between fatal errors (malformed data, disk full) and transient errors (network timeouts). For transient errors, implement a retry queue with exponential backoff.
- **Configuration**: Use a struct to hold configuration constants like `MAX_BATCH_SIZE` and `DEFAULT_RETENTION_DAYS`. Consider using a library like `github.com/spf13/viper` for reading from config files and environment variables.

- **Testing**: Use Go's built-in `testing` package. For component integration tests, you may need to spin up temporary storage engines. Use `testify/assert` for clearer test assertions.

D. Milestone Checkpoint: After Setting Up Structure

What to do:

1. Create the directory structure as outlined.
2. Copy the model definitions (`internal/pkg/models/*.go`) and the minimal ingestion server (`cmd/ingestion/main.go`).
3. Run `go mod tidy` to fetch dependencies.
4. Start the server: `go run ./cmd/ingestion/main.go`.
5. In a separate terminal, send a test log using `curl`:

```bash
curl -X POST http://localhost:8080/ingest/log \
  -H "Content-Type: application/json" \
  -d '{
    "timestamp": 1681234567890000000,
    "body": "User login failed",
    "severity": "ERROR",
    "trace_id": "abc123",
    "span_id": "def456",
    "resource": {
      "service_name": "auth-service",
      "service_version": "v1.2.3",
      "instance_id": "host-123",
      "environment": "staging"
    },
    "attributes": {"user_id": "user42", "error_code": "invalid_credential"}
  }'
```

Expected Output:

- The server should start without errors.
- The `curl` command should return `Log accepted`.
- The server logs should periodically show: `Processing batch of X logs`.

Signs Something is Wrong:

- **Import errors**: Check your `go.mod` module name and ensure the import paths in your `.go` files match.
- **`curl` returns 400**: Your JSON payload might be malformed. Check the exact structure against the `LogRecord` model.
- **No batch processing logs**: The background goroutine in `main.go` is simplistic. It only flushes when you manually trigger or after a fixed interval. You may need to send more logs or modify the flush logic to trigger on batch size.

This foundational setup validates your build environment, module structure, and the most basic ingestion path. The next steps involve replacing the placeholder processing logic with real storage engines and the cross-signal index.

> **Milestone(s):** Milestone 1 (Unified Data Model)

4. Data Model

At the heart of our observability platform lies a unified data model. This is the single most critical architectural decision, as it determines how effectively we can correlate disparate signals. Without a cohesive model, logs, metrics, and traces remain in isolated silos, forcing engineers to perform manual, error-prone joins during debugging. Our model is heavily inspired by OpenTelemetry's semantic conventions, providing a vendor-neutral foundation while ensuring all telemetry shares common attributes that answer the fundamental question: **"Who sent this data, in what context, and how is it related to other data?"**

Think of the unified data model as a **universal identification card for every piece of telemetry data**. Every log line, metric sample, and trace span carries this "ID card" that includes its origin (service, host), context (environment, version), and relationship to other data (trace and span IDs). When a problem occurs, we don't have to guess which service version in the staging environment is misbehaving; the data tells us directly. Similarly, we can follow a `trace_id` like a breadcrumb trail through logs and metrics, reconstructing the complete story of a failed request.

Unified Resource Model: Who Sent This?

The **Resource Model** is the cornerstone of unification. It defines the immutable, coarse-grained identity of the source producing telemetry: the service, its deployment instance, and the environment it runs in. All signals (logs, metrics, traces) emitted from the same source share the same resource attributes. This is analogous to the **return address on an envelope**—no matter what the letter inside says (a log message, a metric value, or a span record), we always know where it came from.

Without a consistent resource model, you might receive an error log from "service-foo" but not know which of its 50 pods in which data center sent it, or whether it's from the `prod` or `staging` environment. Our model solves this by attaching a standardized set of resource attributes to every telemetry signal at the point of creation (by the instrumentation SDK).

The core `ResourceAttributes` struct is defined as follows:

Field Name	Type	Description & Semantic Convention
`service_name`	`string`	**Required.** The logical name of the service (e.g., `"checkout-service"`). This is the primary grouping attribute for all observability data. Follows OpenTelemetry's `service.name` attribute.
`service_version`	`string`	**Recommended.** The version string of the service (e.g., `"v2.1.0"`, `"git-abc123"`). Critical for identifying deployments and correlating issues with specific releases. Maps to `service.version`.
`instance_id`	`string`	**Recommended.** A unique identifier for the specific instance of the service, typically the hostname, pod ID, or container ID (e.g., `"pod-checkout-abc-123"`). Enables pinpointing problems to a specific faulty node. Maps to `service.instance.id`.
`environment`	`string`	**Recommended.** The deployment environment (e.g., `"production"`, `"staging"`, `"development"`). Essential for filtering and separating data from different environments. Maps to `deployment.environment`.

> **Design Insight:** We treat resource attributes as immutable and attached at the source. The ingestion pipeline should never modify these, only propagate them. This ensures the origin of data remains trustworthy throughout its lifecycle.

Architecture Decision Record: Adopting OpenTelemetry Semantic Conventions for Resources

> **Decision: Use OpenTelemetry semantic conventions for the core resource attribute names.**
>
> - **Context:** We need a standardized set of attribute names for service identification that is widely recognized, toolable, and future-proof. Inventing our own names would create friction with existing instrumentation libraries and limit interoperability.
> - **Options Considered:**
>   1. **Invent custom attribute names** (e.g., `app`, `version`, `host`).
>   2. **Adopt another standard** (e.g., Elastic Common Schema (ECS) or prior vendor-specific conventions).
>   3. **Adopt OpenTelemetry semantic conventions.**
> - **Decision:** Option 3 – Adopt OpenTelemetry semantic conventions.
> - **Rationale:** OpenTelemetry is becoming the de facto standard for cloud-native instrumentation. Its semantic conventions are comprehensive, well-documented, and backed by a broad community. Using them ensures maximum compatibility with the OpenTelemetry SDKs that applications will likely use to emit data. It also aligns with our use of OTLP as the primary ingestion protocol.
> - **Consequences:**
>   - **Positive:** Out-of-the-box compatibility with OpenTelemetry-instrumented applications. Easier to explain and document using established terminology. Future-proof as the standard evolves.
>   - **Negative:** May require a translation layer for data from older systems using different conventions (handled in the normalization step of the ingestion pipeline). Some attribute names are longer (e.g., `service.instance.id` vs `host`), slightly increasing payload size.

Option	Pros	Cons	Chosen?
Custom names	Short, simple, full control	Incompatible with ecosystem, requires custom instrumentation	No
Other standard (e.g., ECS)	Mature, used in logging stack	Less focused on tracing/metrics correlation, different community	No
OpenTelemetry semantics	**Ecosystem standard, comprehensive, forward-looking**	**Longer names, newer standard (but rapidly adopted)**	**Yes**

In practice, resource attributes can be extended with custom key-value pairs (e.g., `team="payments"`, `cluster="us-east-1"`). However, the four core fields above are treated as first-class citizens in our system for indexing and query optimization.

Signal-Specific Data Structures

While resource attributes provide common origin information, each telemetry signal type has a unique internal structure optimized for its purpose. The **Span**, **LogRecord**, and **MetricPoint** are the primary data structures. Think of them as **specialized forms attached to the universal ID card**. A span is a detailed travelogue of a request's path, a log record is a timestamped diary entry, and a metric point is a numerical gauge reading. Their structures reflect these distinct roles.

Span: The Request's Journey

A `Span` represents a single, named, and timed operation within a trace, such as a database query or an HTTP handler. Spans are the building blocks of distributed traces, forming a directed acyclic graph (tree) via parent-child relationships. The span structure captures the **"what, when, and where"** of an operation.

Field Name	Type	Description & Semantic Convention
`trace_id`	`string`	**Required.** A 16-byte array identifier, usually represented as a 32-character hex string (e.g., `"4bf92f3577b34da6a3ce929d0e0e4736"`). Uniquely identifies the entire trace tree this span belongs to. All spans sharing the same `trace_id` are part of the same request flow.
`span_id`	`string`	**Required.** An 8-byte array identifier, usually a 16-character hex string (e.g., `"00f067aa0ba902b7"`). Uniquely identifies this specific span within its trace.
`parent_span_id`	`string`	**Optional.** The `span_id` of this span's parent. If empty, this span is a root span (the first operation in a trace). This field establishes the parent-child links that allow reconstruction of the trace tree.
`name`	`string`	**Required.** A human-readable, concise name for the operation (e.g., `"GET /api/users"`, `"orders_db.query"`). Should be low-cardinality (same name for all instances of the same operation).
`start_time`	`int64`	**Required.** The start time of the operation, expressed as nanoseconds since the Unix epoch (UTC).
`end_time`	`int64`	**Required.** The end time of the operation (nanoseconds since Unix epoch). Duration is implicitly `end_time - start_time`.
`resource`	`ResourceAttributes`	**Required.** The resource attributes identifying the service instance that produced this span.
`attributes`	`map[string]string`	**Optional.** Key-value pairs providing additional context about the specific operation. These are high-cardinality details (e.g., `http.status_code="500"`, `db.statement="SELECT * FROM users"`, `error="true"`). Used for filtering and analysis.

Example Span Walkthrough: Consider a user login request. A root span `"POST /login"` is created with a new `trace_id`. It calls the `"auth_db.validate"` service, creating a child span with `parent_span_id` pointing to the root span's `span_id`. Both spans share the same `trace_id`. The child span might have attributes like `db.user="admin"` and `duration_ms=45`. The resource attributes tell us these spans came from the `"auth-service"`, version `"v1.2"`, running in the `"production"` environment on pod `"auth-pod-xyz"`.

LogRecord: The Timestamped Event

A `LogRecord` represents a discrete, timestamped event emitted by a service—typically a line of text from application logs, but could also be structured JSON. Logs provide the **narrative detail and context** that metrics and traces often lack, such as error messages, stack traces, or business events.

Field Name	Type	Description & Semantic Convention
`timestamp`	`int64`	**Required.** The time the log event occurred, in nanoseconds since the Unix epoch (UTC).
`body`	`string`	**Required.** The primary content of the log, which can be plain text (e.g., `"Failed to connect to database: connection refused"`) or a structured JSON string. This field is the target for full-text search.
`severity`	`string`	**Recommended.** The severity level of the log event. We recommend using the OpenTelemetry standard values: `"TRACE"`, `"DEBUG"`, `"INFO"`, `"WARN"`, `"ERROR"`, `"FATAL"`.
`trace_id`	`string`	**Optional.** The `trace_id` of the active trace when the log was emitted. This is the primary link that allows logs to be correlated to a specific trace.
`span_id`	`string`	**Optional.** The `span_id` of the active span when the log was emitted. Provides even more precise correlation within a trace.
`resource`	`ResourceAttributes`	**Required.** The resource attributes of the service instance that produced this log.
`attributes`	`map[string]string`	**Optional.** Additional key-value pairs providing structured context. These can be parsed from structured log JSON (e.g., `error.code="ECONNREFUSED"`, `http.route="/api/login"`) or added by the logging library. Used for field-level filtering (as opposed to full-text search on `body`).

> **Key Insight:** The `trace_id` and `span_id` fields in a `LogRecord` are **optional but critical**. When present, they transform a standalone log entry into a correlated piece of a larger story. Modern logging libraries can be configured to automatically inject these IDs from the current OpenTelemetry context.

MetricPoint: The Numerical Measurement

A `MetricPoint` represents a single sample of a metric—a numerical measurement taken at a specific point in time. Metrics are aggregated over time to reveal trends, patterns, and anomalies. They answer questions like **"how many?"** and **"how slow?"** at a system level.

Field Name	Type	Description & Semantic Convention
`name`	`string`	**Required.** The name of the metric (e.g., `"http.server.request.duration"`, `"queue.size"`). Should follow a naming convention (like OpenTelemetry's metric name rules) to ensure consistency.
`value`	`float64`	**Required.** The numerical value of the measurement. Interpretation depends on metric type: for a `Gauge`, it's a current value; for a `Counter`, it's a cumulative total; for a `Histogram`, it might be a bucket count or sum.
`timestamp`	`int64`	**Required.** The time the metric was measured, in nanoseconds since the Unix epoch (UTC).
`exemplar`	`Exemplar`	**Optional.** A reference to a specific trace that exemplifies this metric point (see next section). This is a powerful correlation mechanism.
`resource`	`ResourceAttributes`	**Required.** The resource attributes of the service instance that produced this metric.
`attributes`	`map[string]string`	**Optional.** Key-value pairs that define the metric's dimensions, creating unique time series. For example, the metric `http.server.request.duration` might have attributes `method="GET"`, `route="/api/users"`, `status_code="200"`. **Caution:** High cardinality (too many unique attribute combinations) can cause storage performance issues.

Understanding Metric Types: While our `MetricPoint` struct is a simplified representation, in practice metrics come in types (Counter, Gauge, Histogram, etc.). For our initial model, we treat `value` as a float, but a full implementation would need a `type` field and potentially additional fields (like bucket boundaries for histograms). The `attributes` field is what leads to **cardinality**—each unique combination of metric `name` and attribute values creates a separate time series.

Exemplar: The Metric's Representative Sample

An `Exemplar` is a special structure attached to a metric data point that provides a direct link to a specific trace that is representative of that measurement. Think of it as a **"see example" footnote** on a metric chart. When you see a spike in latency on a dashboard, you can click through to an exemplar that takes you directly to a trace of a slow request that occurred at that moment.

Field Name	Type	Description
`trace_id`	`string`	**Required.** The `trace_id` of a trace that is a representative example for this metric point.
`value`	`float64`	The actual measured value from the exemplar trace (e.g., the duration of the specific slow request). Often matches the metric point's `value` or is a sample from its distribution.
`timestamp`	`int64`	The timestamp of the exemplar event (usually matches the metric point's timestamp).

Exemplars are typically attached to histogram buckets or summary metrics. For example, a `http.server.request.duration` histogram bucket for `[100ms, 200ms]` might include an exemplar with a `trace_id` of a request that took 150ms. This creates a bridge from aggregated metrics (which lose individual request details) back to specific traces for deep inspection.

Correlation Mechanisms: The Glue

The true power of a unified observability platform emerges from its ability to **correlate across signals**. We implement two primary correlation mechanisms: **Trace-Log correlation** via shared identifiers, and **Metric-Trace correlation** via exemplars. These mechanisms turn separate data streams into an interconnected web, enabling workflows like "show me the logs for this slow trace" or "find traces that contributed to this error rate spike."

1. Trace-Log Correlation: Following the Breadcrumb Trail

Mental Model: Imagine a detective following a suspect (`trace_id`) through a city. Logs are security camera footage (`LogRecords`) from different buildings. If the suspect's ID (`trace_id`) appears in the camera's metadata, the detective can instantly retrieve all footage of that suspect's path. Without that ID, the detective must manually sift through hours of footage from all buildings, hoping to spot the suspect visually.

Mechanism: Correlation is achieved by embedding the active `trace_id` and `span_id` in `LogRecord` fields. When an application is instrumented with OpenTelemetry, the logging library can be configured to automatically inject the current context. This results in logs that are "tagged" with the trace and span they belong to.

How it works:

1. A request enters the system, and a new trace is started with a unique `trace_id`.
2. As the request propagates through services, each span carries this `trace_id`.
3. Any log message emitted within the context of a span (e.g., inside an HTTP handler function) captures the current `trace_id` and `span_id` and includes them in the `LogRecord`.
4. When storing logs, the platform indexes the `trace_id` field (e.g., in the **Cross-Signal Index**).

5. To find logs for a given trace, the query engine looks up the `trace_id` in the log index or cross-signal index, retrieving all matching `LogRecords`.

Example: A failed `/checkout` request with `trace_id="t1"` generates spans in the `frontend`, `cart`, and `payment` services. Each service also logs error messages. All these logs will have `trace_id="t1"`. By querying for `trace_id="t1"`, an engineer gets a unified, chronological view of logs across all three services, painting a complete picture of the failure.

2. Metric-Trace Correlation via Exemplars: From Spike to Sample

Mental Model: Consider a weather report showing an average daily temperature chart (metric). An exemplar is a footnote on a particularly hot day that says, "For example, on July 10th at 2 PM, the temperature was 102°F (see detailed weather log entry #4711)." It connects the aggregated trend to a concrete, inspectable example.

Mechanism: Exemplars are `Exemplar` structures attached to `MetricPoint`s, containing a `trace_id`. They are typically sampled from low-latency, high-cardinality data (like spans) and attached to aggregated metrics (like histograms) where individual request details are otherwise lost.

How it works:

1. As the system processes requests (traces), it also updates relevant metrics (e.g., a latency histogram).
2. Periodically, the system selects a representative trace (e.g., the slowest request in a batch) and attaches its `trace_id` and measured value as an `Exemplar` to the corresponding metric data point (e.g., to the histogram bucket that request's duration fell into).
3. The metric storage retains these exemplars alongside the metric data.
4. When viewing a metric chart that shows an anomaly (e.g., a latency spike), the UI can fetch the exemplars for the data points around the spike and offer a "View Example Trace" link.

Example: The `http.server.request.duration` histogram for the `checkout-service` shows a spike in the `500-1000ms` bucket at 10:05 AM. That bucket's metric point includes an exemplar with `trace_id="t2"` and `value=750`. The engineer clicks on the exemplar and is taken to the full trace `t2`, revealing that the slowness was due to a particular database query in the `payments` service.

Common Pitfalls in Data Modeling

⚠️ Pitfall: Inconsistent `trace_id` Format or Propagation

- **Description:** Different services or logging libraries generate `trace_id` in different formats (e.g., 16-byte array vs 32-char hex string) or fail to propagate it via context headers, breaking correlation.
- **Why it's wrong:** Logs and traces won't link correctly. Correlation queries will return partial or no results, defeating the purpose of unification.
- **How to fix:** Enforce a standard format (32-character lowercase hex string) at the ingestion boundary via normalization. Use OpenTelemetry SDKs and context propagation libraries (like W3C TraceContext) consistently across all services.

⚠️ Pitfall: High Cardinality Explosion in Metric Attributes

- **Description:** Adding unbounded, high-cardinality attributes to metrics (e.g., `user_id="12345"`, `request_id="abc-def"`) creates a unique time series for each value, overwhelming storage and query systems.
- **Why it's wrong:** Storage costs skyrocket, query performance plummets, and the system may become unusable. This is a classic observability system breaker.
- **How to fix:** Carefully design metric attributes to be low-to-moderate cardinality. Use dimensions that group requests meaningfully (e.g., `http.route`, `status_code`, `error_type`). Reserve high-cardinality details for traces and logs. Implement cardinality limits and monitoring in the metric ingestion path.

⚠️ Pitfall: Missing Resource Attributes or Incorrect Values

- **Description:** Services are deployed without properly configured resource attributes (e.g., `environment` defaults to `"unknown"`, `instance_id` is not unique), or attributes have wrong values (staging service reporting as `production`).
- **Why it's wrong:** Filtering and grouping by environment, service version, or instance becomes impossible or misleading. You cannot reliably isolate problems.
- **How to fix:** Make resource attribute configuration mandatory and automated in deployment pipelines (e.g., injected via environment variables or pod metadata). Validate attributes at ingestion and tag data with warnings if required fields are missing.

⚠️ Pitfall: Not Indexing Correlation Fields

- **Description:** Storing `trace_id` in `LogRecord` but not creating a database index on that field.
- **Why it's wrong:** Queries for "logs by trace_id" will require a full table/collection scan, which is unacceptably slow at scale, making the correlation feature useless in production.
- **How to fix:** Ensure the storage engines for logs and traces have dedicated indexes on `trace_id`. The **Cross-Signal Index** is specifically designed for this purpose.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Data Model Structs	Plain Go structs with `json` tags.	Protocol Buffers definitions (`.proto` files) for strict schema enforcement and efficient serialization across language boundaries.
Correlation ID Generation	Use OpenTelemetry SDK's `go.opentelemetry.io/otel/trace` package to generate and propagate W3C TraceContext-compliant IDs.	Implement a custom, high-performance ID generator (e.g., based on Snowflake or UUIDv4) if OTel SDK overhead is a concern (rare).
Exemplar Sampling	Simple random sampling: attach an exemplar to every Nth metric point, or when value exceeds a threshold.	Adaptive sampling based on rate of change or outlier detection; reservoir sampling to ensure fair representation.

B. Recommended File/Module Structure

Place the core data model definitions in an `internal/models/` package. This keeps them isolated and importable by all other components (ingestion, storage, query).

```
project-root/
  internal/
    models/                   # Unified data model definitions
      resource.go            # ResourceAttributes struct and helpers
      log.go                 # LogRecord struct
      span.go                # Span struct
      metric.go              # MetricPoint and Exemplar structs
      common.go              # Constants (severity levels, etc.)
    ingestion/               # Milestone 2
    storage/                 # Milestone 3
    query/                   # Milestone 4
    alerting/                # Milestone 5
```

C. Infrastructure Starter Code (Complete)

Here is the complete, ready-to-use data model definition in Go. This should be placed in the respective files under `internal/models/`.

File: `internal/models/resource.go`

```go
package models

// ResourceAttributes identifies the source of telemetry data.
// Follows OpenTelemetry semantic conventions.
type ResourceAttributes struct {
        ServiceName    string `json:"service_name"`    // service.name
        ServiceVersion string `json:"service_version"` // service.version
        InstanceID     string `json:"instance_id"`     // service.instance.id
        Environment    string `json:"environment"`     // deployment.environment
        // Custom attributes can be added as a map, but core fields are promoted for efficiency.
        // OtherAttributes map[string]string `json:"other_attributes,omitempty"`
}

// IsValid performs basic validation on required fields.
func (r *ResourceAttributes) IsValid() bool {
        return r.ServiceName != ""
}
```

File: `internal/models/log.go`

```go
package models

// LogSeverity represents the severity level of a log record.
// Values align with OpenTelemetry's LogRecord.SeverityNumber.
type LogSeverity string

const (
	SeverityTrace  LogSeverity = "TRACE"
	SeverityDebug  LogSeverity = "DEBUG"
	SeverityInfo   LogSeverity = "INFO"
	SeverityWarn   LogSeverity = "WARN"
	SeverityError  LogSeverity = "ERROR"
	SeverityFatal  LogSeverity = "FATAL"
	SeverityUnspecified LogSeverity = "UNSPECIFIED"
)

// LogRecord represents a single log entry.
type LogRecord struct {
	Timestamp  int64                `json:"timestamp"`   // nanoseconds since Unix epoch
	Body       string               `json:"body"`        // the log message, can be plain text or JSON string
	Severity   LogSeverity          `json:"severity"`    // severity level
	TraceID    string               `json:"trace_id,omitempty"`    // optional correlation to trace
	SpanID     string               `json:"span_id,omitempty"`     // optional correlation to span
	Resource   ResourceAttributes   `json:"resource"`    // who produced this log
	Attributes map[string]string    `json:"attributes,omitempty"` // additional structured context
}

// HasTraceContext returns true if the log record is associated with a trace.
func (lr *LogRecord) HasTraceContext() bool {
	return lr.TraceID != ""
}
```

File: `internal/models/span.go`

```go
package models

// Span represents a single operation within a trace.
type Span struct {
	TraceID      string                `json:"trace_id"`              // 32-character hex string
	SpanID       string                `json:"span_id"`               // 16-character hex string
	ParentSpanID string                `json:"parent_span_id,omitempty"` // optional, 16-character hex string
	Name         string                `json:"name"`                  // low-cardinality operation name
	StartTime    int64                 `json:"start_time"`            // nanoseconds since Unix epoch
	EndTime      int64                 `json:"end_time"`              // nanoseconds since Unix epoch
	Resource     ResourceAttributes    `json:"resource"`              // who produced this span
	Attributes   map[string]string     `json:"attributes,omitempty"`  // high-cardinality context
}

// Duration returns the span's duration in nanoseconds.
func (s *Span) Duration() int64 {
	return s.EndTime - s.StartTime
}

// IsRoot returns true if the span has no parent (empty ParentSpanID).
func (s *Span) IsRoot() bool {
	return s.ParentSpanID == ""
}
```

File: `internal/models/metric.go`

```go
package models

// Exemplar links a metric measurement to a specific trace.
type Exemplar struct {
	TraceID   string  `json:"trace_id"`   // 32-character hex string
	Value     float64 `json:"value"`      // the measured value from the exemplar trace
	Timestamp int64   `json:"timestamp"`  // nanoseconds since Unix epoch
}

// MetricPoint represents a single sample of a metric.
type MetricPoint struct {
	Name       string                `json:"name"`                // metric name, e.g., "http.server.request.duration"
	Value      float64               `json:"value"`               // the numerical value
	Timestamp  int64                 `json:"timestamp"`           // nanoseconds since Unix epoch
	Exemplar   *Exemplar             `json:"exemplar,omitempty"`  // optional link to a trace
	Resource   ResourceAttributes    `json:"resource"`            // who produced this metric
	Attributes map[string]string     `json:"attributes,omitempty"` // dimensions/labels
}

// HasExemplar returns true if the metric point includes an exemplar.
func (mp *MetricPoint) HasExemplar() bool {
	return mp.Exemplar != nil
}
```

File: `internal/models/common.go`

```go
package models

// Constants used across the data model.
const (
	// TraceIDLength is the length of a hex-encoded trace ID (16 bytes -> 32 chars).
	TraceIDLength = 32
	// SpanIDLength is the length of a hex-encoded span ID (8 bytes -> 16 chars).
	SpanIDLength = 16
)
```

D. Core Logic Skeleton Code

For the **normalization component** (part of Milestone 2 ingestion pipeline), which will ensure incoming data adheres to our unified model. This is where you would validate and standardize fields like `trace_id` format.

File: `internal/ingestion/normalizer.go`

```go
package ingestion

import (
	"errors"
	"strings"
	"your-project/internal/models"
)

// Normalizer ensures incoming telemetry conforms to the unified data model.
type Normalizer struct {
	// config might include options like strict mode, default environment, etc.
}

// NormalizeLog takes a LogRecord, validates, and standardizes its fields.
// Returns an error if the log is invalid.
func (n *Normalizer) NormalizeLog(log *models.LogRecord) error {
	// TODO 1: Validate required fields: Timestamp, Body, Resource.ServiceName
	// TODO 2: If TraceID is present, normalize its format (ensure 32-char lowercase hex, pad if needed)
	// TODO 3: If SpanID is present, normalize its format (ensure 16-char lowercase hex)
	// TODO 4: Ensure Severity is one of the allowed constants; default to SeverityUnspecified if empty
	// TODO 5: (Optional) Parse log Body if it's JSON and merge top-level fields into Attributes map
	// TODO 6: Trim whitespace from string fields to avoid storage of padded strings
	// Hint: Use `strings.ToLower(strings.TrimSpace(log.TraceID))` for normalization
	return errors.New("not implemented")
}

// NormalizeSpan validates and normalizes a Span.
func (n *Normalizer) NormalizeSpan(span *models.Span) error {
	// TODO 1: Validate required fields: TraceID, SpanID, Name, StartTime, EndTime, Resource.ServiceName
	// TODO 2: Normalize TraceID, SpanID, ParentSpanID to lowercase hex
	// TODO 3: Ensure StartTime <= EndTime (allow zero-length spans)
	// TODO 4: If ParentSpanID is empty string, set it to empty string (not "0000000000000000")
	// TODO 5: Validate that TraceID and SpanID are not equal (a span cannot be its own trace)
	return errors.New("not implemented")
}

// NormalizeMetric validates and normalizes a MetricPoint.
func (n *Normalizer) NormalizeMetric(metric *models.MetricPoint) error {
	// TODO 1: Validate required fields: Name, Value, Timestamp, Resource.ServiceName
	// TODO 2: If Exemplar is present, validate its TraceID format and that Exemplar.Timestamp is not far from MetricPoint.Timestamp
	// TODO 3: Check for extreme values (e.g., Value is NaN or Inf) and handle or reject
	// TODO 4: Warn or drop attributes with extremely high cardinality (e.g., keys like "user_id") - this is a policy decision
	return errors.New("not implemented")
```

```
}
```

E. Language-Specific Hints (Go)

- Use `time.Unix(0, timestamp).UTC()` to convert nanosecond timestamps to Go `time.Time`.
- For hex string validation/normalization, use `encoding/hex` package functions like `hex.DecodeString`.
- Consider using `sync.Pool` for reusing `map[string]string` attribute maps in hot paths to reduce garbage collection pressure.
- Use `json.RawMessage` for the `LogRecord.Body` if you want to defer parsing of structured logs.

F. Milestone Checkpoint After implementing the data model (Milestone 1), you should be able to run a simple test that creates correlated data and verifies the relationships.

Test Command:

```bash
go test ./internal/models/... -v
```

Expected Output: Tests should pass, demonstrating:

- Structs can be marshaled to JSON and unmarshaled back.
- Validation functions correctly identify invalid data (e.g., missing `service_name`).
- Correlation helper functions work (e.g., `LogRecord.HasTraceContext()` returns true when `trace_id` is set).

Manual Verification:

1. Write a small Go program that creates:
   - A `ResourceAttributes` for service `"test-svc"`.
   - A `Span` with a generated `trace_id`.
   - A `LogRecord` with the same `trace_id`.
   - A `MetricPoint` with an `Exemplar` containing that `trace_id`.
2. Print these structs as JSON. Observe that the `trace_id` appears consistently across all three signals.
3. This validates that the data model can represent correlated telemetry.

Signs of Trouble:

- JSON marshaling fails due to private fields (ensure fields are exported).
- `trace_id` hex strings have inconsistent length (must be 32 chars for trace, 16 for span).
- The `exemplar` field is missing when it should be present (check pointer usage).

> **Milestone(s):** Milestone 2 (Data Ingestion Pipeline)

5. Component: Ingestion Pipeline

The Ingestion Pipeline is the **front door** of the observability platform. It is responsible for receiving telemetry data from potentially thousands of distributed services, performing initial validation and transformation, and routing it efficiently to the appropriate storage backends. This component must be resilient, high-throughput, and impose minimal overhead on the services it observes. Its design is critical for platform reliability; a bottleneck here means telemetry data loss and blind spots in system visibility.

Mental Model: A High-Capacity Loading Dock

Imagine the observability platform as a massive warehouse complex. The Ingestion Pipeline is the **loading dock** where delivery trucks (your applications) arrive 24/7 to unload crates of telemetry (logs, metrics, traces).

- **Trucks (Applications):** Each service instance is a truck. They arrive at scheduled intervals or when full, delivering their cargo.
- **Dock Workers (Receivers):** These workers greet the trucks, check the shipping manifests (protocol headers), and begin unloading the cargo crates (data batches).
- **Conveyor Belts (Internal Channels):** Once unloaded, crates are placed on fast-moving conveyor belts (buffered channels in Go). These belts transport crates to the next stage without blocking the dock workers.
- **Quality Inspection & Labeling (Normalizer):** A station on the conveyor belt where workers open each crate, check for damage (validation), repackage items into standard containers (normalization), and attach consistent warehouse labels (resource attributes).
- **Sorting Machinery (Router):** Automated machinery reads the labels on the standardized crates and routes them onto different conveyor lines headed for the correct specialized warehouse: the Document Warehouse (Log Storage), the Time-Series Vault (Metric Storage), or the Graph Archive (Trace Storage).
- **Traffic Lights (Backpressure):** Sensors monitor the speed of the conveyor belts and the backlog at the sorting machinery. If things are moving too slowly, they turn on a red light at the loading dock entrance, signaling trucks to wait before attempting to unload.

This mental model clarifies the pipeline's decoupled, staged architecture. Each stage has a single responsibility, and buffers (conveyor belts) between stages prevent a slow downstream stage (like a jam at sorting) from immediately halting upstream work (unloading trucks). The traffic light system (backpressure) protects the entire internal system from being overwhelmed by external load.

ADR: Choosing OTLP/gRPC as the Primary Protocol

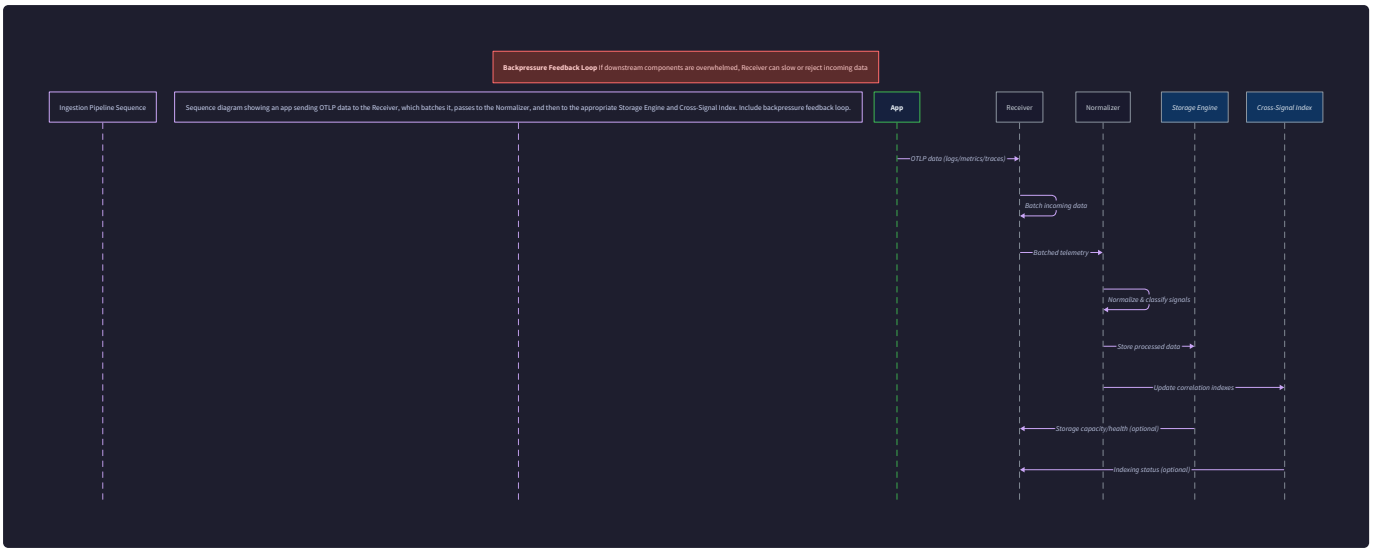
> **Decision: Standardize on OTLP/gRPC for Primary Telemetry Ingestion**
>
> - **Context:** We need a unified, efficient, and widely-adopted protocol for receiving logs, metrics, and traces from instrumented applications. The protocol choice impacts client instrumentation complexity, network efficiency, and our ability to integrate with the broader observability ecosystem.
> - **Options Considered:**
>   1. **Vendor-Specific Protocols (e.g., Datadog Agent Forwarder, Splunk HEC):** Protocols designed for specific commercial backends.
>   2. **Raw HTTP/JSON with Custom Schema:** A simple REST API where clients POST JSON payloads.
>   3. **OpenTelemetry Protocol (OTLP) over gRPC:** The CNCF-standardized protocol for telemetry data.
> - **Decision:** Implement OTLP over gRPC as the primary, first-class ingestion protocol. We will support OTLP over HTTP/protobuf as a secondary option for environments where gRPC is problematic.
> - **Rationale:** OTLP is an open standard that provides a unified, efficient, and forward-compatible wire format for all three telemetry signals. Using gRPC provides performance benefits (binary protobuf encoding, multiplexed streams, connection pooling) which are critical for high-throughput ingestion. Adopting the standard maximizes interoperability; applications instrumented with OpenTelemetry SDKs can send data directly to our platform with zero or minimal configuration changes. This reduces vendor lock-in for both us and our users.
> - **Consequences:**
>   - **Positive:** Immediate compatibility with the OpenTelemetry ecosystem. Efficient binary transport reduces network overhead and CPU usage for serialization. Strongly-typed .proto definitions serve as canonical API documentation.
>   - **Negative:** Requires clients to have gRPC support (though OTel SDKs provide this). Slightly more complex to debug than plain HTTP/JSON. We must still handle payload validation and sanitization.

Protocol Options Comparison

Option	Pros	Cons	Decision
Vendor-Specific Protocols	Highly optimized for that vendor's backend.	Creates lock-in. Requires running vendor's agent/proxy. Fragments the ecosystem.	❌ Rejected
Raw HTTP/JSON	Universally accessible, easy to debug with `curl`. Simple to implement initially.	Inefficient (verbose encoding, no compression). No standard schema leads to fragility. Requires us to design and maintain API semantics.	❌ Rejected as primary
OTLP over gRPC	**CNCF Standard.** Unified schema for all signals. High performance (binary Protobuf, HTTP/2). Built-in streaming and flow control. Future-proof.	Requires gRPC client libraries. Binary format requires tooling for debugging.	✅ **Chosen as Primary**
OTLP over HTTP/Protobuf	Standard OTLP schema. Easier firewall traversal than gRPC for some users. Still reasonably efficient.	Less performant than gRPC (no multiplexing, often text-based protobuf).	✅ **Supported as Secondary**

Component Design & Data Flow

The pipeline is composed of several sequential stages, each communicating via bounded buffers (channels in Go). This design follows the **pipeline pattern**, promoting separation of concerns, independent scalability per stage, and resilience through buffering.



1. OTLP Receiver This is the network-facing component. It listens for incoming gRPC (and optionally HTTP) connections on port `OTLP_GRPC_PORT` (4317). Its responsibilities are:

- Accepting connections and streaming requests.
- Performing protocol-level validation (e.g., checking gRPC metadata, message size limits).
- Deserializing the incoming Protocol Buffer payloads into internal in-memory representations (`LogRecord` , `Span` , `MetricPoint`).
- Applying **backpressure** by rejecting or throttling new requests when internal buffers are full.
- Placing the deserialized data onto the first internal batch channel for processing.

2. Batch Aggregator This stage receives individual telemetry items and groups them into batches for more efficient processing. Batching amortizes the overhead of channel communication, storage writes, and indexing across many items.

- It maintains a buffer per signal type (log, metric, trace) or per tenant.
- Batches are flushed based on two criteria: a **size limit** (`MAX_BATCH_SIZE`) or a **time window** (e.g., every 100ms).
- The output is a `Batch` structure containing a slice of items of the same signal type and their common `ResourceAttributes` .

3. Normalizer & Validator This is a critical data hygiene stage. It ensures all data conforms to the platform's unified data model before it hits storage. It processes entire batches for efficiency.

- **Validation:** Checks for required fields (e.g., `trace_id` length must be `TraceIDLength`), valid timestamps, and enum values (e.g., `LogSeverity` must be a known constant).
- **Normalization:** Standardizes field formats (e.g., ensures all timestamps are in nanoseconds since Unix epoch, converts `service.name` resource attribute to a standard key).
- **Enrichment:** Can add missing default attributes (e.g., `environment=production` if not provided) based on the resource model.
- It calls the `Normalizer.NormalizeLog` , `Normalizer.NormalizeSpan` , and `Normalizer.NormalizeMetric` methods.

4. Router & Dispatcher The final stage inspects the normalized batch and decides where to send it. It fans out batches to the appropriate storage engine's ingestion interface.

- Log batches are sent to the **Log Storage** engine.
- Metric batches are sent to the **Metric Storage** engine.
- Trace batches are sent to the **Trace Storage** engine.
- Additionally, for any batch containing items with a `trace_id` , it sends information to the **Cross-Signal Correlation Index** to update the mapping from `trace_id` to the location of related logs and spans.

Key Data Structures

Batch Container

Field Name	Type	Description
`signal_type`	`string`	One of: `"log"` , `"metric"` , `"trace"` . Determines which storage engine receives the batch.
`items`	`[]interface{}`	A slice of normalized items. In practice, a type-safe wrapper like `[]*LogRecord` would be used per batch type.
`resource`	`ResourceAttributes`	The common resource attributes for all items in this batch. Extracted during aggregation.
`received_at`	`int64`	Timestamp (nanoseconds) when the batch was created by the aggregator. Used for internal latency monitoring.

Internal Channel & Buffer Configuration These are not structs but critical configuration constants that govern pipeline behavior.

Constant	Typical Value	Description
`INGEST_CHANNEL_BUFFER`	1000	Size of the buffered channel between the Receiver and the Batch Aggregator. Once full, the Receiver must apply backpressure.
`BATCH_CHANNEL_BUFFER`	100	Size of the channel holding ready-to-process batches between the Aggregator and the Normalizer.
`FLUSH_INTERVAL_MS`	100	Maximum time (milliseconds) the Batch Aggregator will hold an incomplete batch before flushing it.
`MAX_BATCH_SIZE`	1000	Maximum number of telemetry items in a single batch. Prevents overly large memory allocations.

Backpressure Strategy

Backpressure is essential to prevent the pipeline from consuming unbounded memory during traffic spikes or downstream outages. Our strategy is **progressive** and **communicative**.

1. **Internal Buffering (First Line of Defense):** Each channel between pipeline stages has a fixed capacity (`INGEST_CHANNEL_BUFFER`, `BATCH_CHANNEL_BUFFER`). When a channel is full, the goroutine trying to send to it will block. This naturally slows down the preceding stage.
2. **Receiver-Level Throttling (Second Line):** The OTLP Receiver monitors the fullness of its output channel. When the channel is above a high watermark (e.g., 90% full), it begins to reject new incoming gRPC requests with a **`RESOURCE_EXHAUSTED`** status code (Google APIs) or **`UNAVAILABLE`** (gRPC). This signals to the client SDK that the server is temporarily overloaded. Well-behaved SDKs will implement exponential backoff and retry.
3. **Downstream Circuit Breakers (Third Line):** If a storage engine (e.g., Log Storage) becomes slow or fails entirely, the Router should detect this (via timeouts or error returns). It can then open a **circuit breaker** for that specific engine, failing fast for new batches destined for it and returning an error to the Normalizer stage. This prevents queues from forming for a broken backend.

Backpressure Response Table

Pipeline State	Receiver Action	Client SDK Expected Behavior
Healthy (buffers < 80%)	Accept all requests.	Send data normally.
Loaded (buffers 80-95%)	Continue accepting, but log warnings.	May see slightly increased latency.
Saturated (buffers > 95%)	Reject new requests with `UNAVAILABLE`.	Should back off and retry with exponential delay.
Downstream Failure (e.g., storage offline)	Reject requests. Circuit breaker may be open.	Back off and retry. May eventually drop data based on SDK configuration.

Common Pitfalls

⚡ Pitfall: Blocking on an Unbounded Channel

- **Mistake:** Using an unbuffered channel or a channel with a massive buffer between stages. An unbuffered channel tightly couples stage speeds, causing the whole pipeline to run at the speed of the slowest stage. A huge buffer hides problems and leads to uncontrolled memory growth.
- **Why it's wrong:** It destroys the resilience benefits of the pipeline pattern. A temporary slowdown in storage causes the Receiver to block, which then causes client requests to block, potentially causing application timeouts.
- **Fix:** Use **bounded buffered channels** with sizes tuned to absorb small bursts (e.g., 100-1000 items). Combine this with explicit backpressure at the Receiver when the buffer is full.

⚡ Pitfall: Ignoring Tenant or Service Isolation in Batches

- **Mistake:** Batching telemetry items solely based on signal type and timing, without considering their source (`ResourceAttributes`). This mixes data from different services or tenants into a single batch.
- **Why it's wrong:** It breaks logical isolation. If a batch fails during processing (e.g., normalization error), it fails for all services in that batch. It also makes per-service/tenant rate limiting and quality-of-service harder to implement later.
- **Fix:** The Batch Aggregator should use a composite key for its buffers, such as `(tenant_id, service_name, signal_type)`. This ensures batches are homogenous from a resource perspective.

⚡ Pitfall: Performing Heavy Processing in the Receiver

- **Mistake:** Putting normalization, validation, or enrichment logic directly in the gRPC request handler.
- **Why it's wrong:** It ties up the gRPC worker goroutines, limiting the server's ability to accept new connections and requests. It makes the response time to the client dependent on the cost of processing.
- **Fix:** The Receiver's job is **protocol deserialization and flow control only**. It should decode the payload and immediately place the raw items onto a channel for asynchronous processing by the next stage.

⚡ Pitfall: Silent Data Loss on Normalization Failure

- **Mistake:** When `Normalizer.NormalizeLog` (or similar) returns an error for an item, simply logging the error and discarding the entire batch.
- **Why it's wrong:** A single malformed log record from one buggy service should not cause the loss of hundreds of valid records from other healthy services in the same batch.
- **Fix:** Implement **per-item error handling** within batch processing. Valid items proceed; invalid items are moved to a "dead letter queue" (a separate storage area) for later inspection and debugging, and a counter is incremented.

Implementation Guidance

A. Technology Recommendations

Component	Simple Option	Advanced Option
gRPC Server Framework	Standard `google.golang.org/grpc`	With connection pooling & advanced interceptors
Protocol Buffers	`google.golang.org/protobuf` (v2 API)	With `protoc` code generation for OTLP
Internal Concurrency	Go channels & goroutines	Pattern like "worker pool" for the Normalizer stage
Backpressure Signaling	Channel capacity & `grpc.Status` codes	Adaptive rate limiting based on queue length
Metrics (for pipeline itself)	Prometheus Go client library	Exported to the platform's own Metric Storage

B. Recommended File/Module Structure

```
project-root/
├── api/                          # Protobuf definitions and generated code
│   └── otlp/                     # OTLP .proto files (copied from opentelemetry-proto)
├── internal/
│   ├── ingestion/
│   │   ├── pipeline.go           # Main pipeline orchestration logic
│   │   ├── receiver.go           # OTLP gRPC/HTTP server implementation
│   │   ├── batcher.go            # Batch aggregation logic
│   │   ├── normalizer.go         # Validation & normalization logic
│   │   ├── router.go             # Routing to storage engines
│   │   ├── backpressure.go       # Backpressure monitoring & control
│   │   └── models/               # Internal batch models
│   │       └── batch.go
│   └── telemetry/                # Core data models (from Milestone 1)
│       ├── resource.go           # ResourceAttributes
│       ├── log.go                # LogRecord
│       ├── span.go               # Span
│       └── metric.go             # MetricPoint, Exemplar
└── cmd/
    └── server/
        └── main.go               # Server entry point, starts pipeline
```

C. Infrastructure Starter Code

1. OTLP gRPC Server Setup (Complete) This is a complete, minimal gRPC server setup using the official OTLP proto definitions. Place this in `cmd/server/main.go` or a dedicated `internal/ingestion/server.go`.

GO

```go
// This is a complete starter for the gRPC server. You need to generate the protobuf code first.
// To generate: protoc --go_out=. --go-grpc_out=. api/otlp/*.proto
package main

import (
    "log"
    "net"
    "os"
    "os/signal"
    "syscall"

    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/status"

    otlpcollector "your-project/api/otlp" // Generated package
)

const (
    OTLP_GRPC_PORT = ":4317"
)

func main() {
    lis, err := net.Listen("tcp", OTLP_GRPC_PORT)
    if err != nil {
        log.Fatalf("failed to listen: %v", err)
    }

    // Create the gRPC server with options for size limits and keepalive.
    s := grpc.NewServer(
        grpc.MaxRecvMsgSize(10*1024*1024), // 10MB max message size
    )

    // Register our service implementation.
    svc := &otlpService{}
    otlpcollector.RegisterLogsServiceServer(s, svc)
    otlpcollector.RegisterMetricsServiceServer(s, svc)
    otlpcollector.RegisterTraceServiceServer(s, svc)

    log.Printf("Starting OTLP gRPC server on %s", OTLP_GRPC_PORT)

    // Graceful shutdown handling.
    go func() {
        sigChan := make(chan os.Signal, 1)
        signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)
```

```
        <-sigChan
        log.Println("Shutdown signal received, gracefully stopping gRPC server...")
        s.GracefulStop()
    }()

    if err := s.Serve(lis); err != nil {
        log.Fatalf("failed to serve: %v", err)
    }
}

// otlpService is a stub that implements the OTLP gRPC interfaces.
// You will fill in the Export methods to forward data to the pipeline.
type otlpService struct {
    otlpcollector.UnimplementedLogsServiceServer
    otlpcollector.UnimplementedMetricsServiceServer
    otlpcollector.UnimplementedTraceServiceServer
}

func (s *otlpService) Export(ctx context.Context, req *otlpcollector.ExportLogsServiceRequest) (*otlpcollector.ExportLogsServiceResponse,
error) {
    // TODO 1: Extract ResourceLogs from req
    // TODO 2: Convert each log record to internal LogRecord model
    // TODO 3: Send to the ingestion pipeline's input channel (apply backpressure here)
    // TODO 4: Return appropriate gRPC status on error (e.g., UNAVAILABLE if pipeline is full)
    return &otlpcollector.ExportLogsServiceResponse{}, nil
}
// Similar Export methods for Metrics and Traces...
```

2. Thread-Safe Batch Buffer (Complete) This helper manages in-memory batching with flush triggers. Place in `internal/ingestion/batcher.go`.

```go
package ingestion

import (
    "sync"
    "time"
    "your-project/internal/telemetry"
)

type batchBuffer struct {
    mu          sync.Mutex
    items       []interface{} // Will be typed per signal in practice
    resource    *telemetry.ResourceAttributes
    signalType  string
    lastFlush   time.Time
    maxSize     int
    flushInterval time.Duration
}

func newBatchBuffer(signalType string, maxSize int, flushInterval time.Duration) *batchBuffer {
    return &batchBuffer{
        signalType:    signalType,
        maxSize:       maxSize,
        flushInterval: flushInterval,
        lastFlush:     time.Now(),
        items:         make([]interface{}, 0, maxSize),
    }
}

func (b *batchBuffer) add(item interface{}, resource *telemetry.ResourceAttributes) bool {
    b.mu.Lock()
    defer b.mu.Unlock()

    // Set resource on first item, or verify it matches subsequent items.
    if b.resource == nil {
        b.resource = resource
    } else if !resourcesEqual(b.resource, resource) {
        // In a real implementation, you'd handle multi-resource batches differently.
        return false // Indicate mismatch, caller should flush and retry.
    }

    b.items = append(b.items, item)

    // Check if we should flush due to size.
    if len(b.items) >= b.maxSize {
```

```
        return true
    }
    return false
}

func (b *batchBuffer) shouldFlushOnTime() bool {
    b.mu.Lock()
    defer b.mu.Unlock()
    return time.Since(b.lastFlush) >= b.flushInterval && len(b.items) > 0
}

func (b *batchBuffer) take() ([]interface{}, *telemetry.ResourceAttributes) {
    b.mu.Lock()
    defer b.mu.Unlock()
    items := b.items
    resource := b.resource
    // Reset the buffer.
    b.items = make([]interface{}, 0, b.maxSize)
    b.resource = nil
    b.lastFlush = time.Now()
    return items, resource
}

func resourcesEqual(r1, r2 *telemetry.ResourceAttributes) bool {
    // Simplified equality check. Real implementation would compare all fields.
    return r1.ServiceName == r2.ServiceName && r1.Environment == r2.Environment
}
```

D. Core Logic Skeleton Code

1. Pipeline Orchestrator (`pipeline.go`)

```go
package ingestion

import (
    "context"
    "log"
    "your-project/internal/telemetry"
)

// Pipeline orchestrates the stages of ingestion.
type Pipeline struct {
    logChan       chan *telemetry.LogRecord
    metricChan    chan *telemetry.MetricPoint
    traceChan     chan *telemetry.Span
    batchChan     chan *Batch
    // ... other channels and components
    cancelFunc    context.CancelFunc
}

func NewPipeline() *Pipeline {
    // TODO 1: Initialize all channels with their buffer sizes (e.g., make(chan *telemetry.LogRecord, INGEST_CHANNEL_BUFFER))
    // TODO 2: Create instances of Batcher, Normalizer, Router
    // TODO 3: Return a new Pipeline struct with these components
}

func (p *Pipeline) Start(ctx context.Context) {
    ctx, cancel := context.WithCancel(ctx)
    p.cancelFunc = cancel

    // TODO 4: Start goroutines for each pipeline stage, passing the context.
    //   - go p.batcher.run(ctx, p.logChan, p.metricChan, p.traceChan, p.batchChan)
    //   - go p.normalizer.run(ctx, p.batchChan, p.normalizedBatchChan)
    //   - go p.router.run(ctx, p.normalizedBatchChan)
    // TODO 5: Log that the pipeline has started.
}

func (p *Pipeline) Stop() {
    // TODO 6: Call cancelFunc to signal all goroutines to stop.
    // TODO 7: Wait for all goroutines to exit (using sync.WaitGroup).
    // TODO 8: Close all channels safely (only after producers are stopped).
}

// ReceiveLog is called by the OTLP receiver to inject a log into the pipeline.
func (p *Pipeline) ReceiveLog(log *telemetry.LogRecord) error {
    select {
```

```
    case p.logChan <- log:
        return nil
    default:
        // Channel is full. Apply backpressure.
        return ErrPipelineFull
    }
}
// Similar methods for ReceiveMetric and ReceiveSpan.
```

2. Normalizer Implementation (`normalizer.go`)

```go
package ingestion


import "your-project/internal/telemetry"


type Normalizer struct {

    // May contain configuration for default attributes, etc.

}


func (n *Normalizer) NormalizeLog(log *telemetry.LogRecord) error {

    // TODO 1: Validate required fields: timestamp > 0, non-empty body.

    // TODO 2: Normalize timestamp: ensure it's in nanoseconds. If in milliseconds, multiply by 1e6.

    // TODO 3: Normalize severity: convert from string/int to the LogSeverity enum constant (e.g., map "ERROR" -> SeverityError).

    // TODO 4: Validate trace_id and span_id: if present, length must be TraceIDLength (32) and SpanIDLength (16) characters (hex).

    // TODO 5: Normalize resource attributes: ensure required fields (service_name, environment) exist, add defaults if not.

    // TODO 6: Standardize attribute keys: e.g., convert 'http.status_code' to 'http.status_code' (lowercase, dot separator) per
OpenTelemetry semantic conventions.

    // TODO 7: Return a validation error if any critical check fails (e.g., invalid trace_id format).

    return nil

}


func (n *Normalizer) NormalizeSpan(span *telemetry.Span) error {

    // TODO 1: Validate required fields: trace_id, span_id, name, start_time, end_time.

    // TODO 2: Validate ID lengths (TraceIDLength, SpanIDLength).

    // TODO 3: Ensure start_time <= end_time. Calculate and set duration if not present.

    // TODO 4: Normalize parent_span_id: if empty, it's a root span.

    // TODO 5: Normalize resource attributes (same as for logs).

    // TODO 6: Standardize span attributes (e.g., 'http.method' -> 'http.method').

    // TODO 7: Ensure the span's resource.IsValid() returns true.

    return nil

}


func (n *Normalizer) NormalizeMetric(metric *telemetry.MetricPoint) error {

    // TODO 1: Validate required fields: name, value, timestamp.

    // TODO 2: Normalize timestamp (to nanoseconds).

    // TODO 3: If exemplar is present, validate its trace_id length and timestamp.

    // TODO 4: Normalize resource attributes.

    // TODO 5: Standardize metric attributes (e.g., 'http.server.port').

    // TODO 6: For certain metric names (e.g., latencies), ensure value is non-negative.

    return nil

}
```

E. Language-Specific Hints (Go)

- **Concurrency:** Use `select` with a `default` clause on channel sends to implement non-blocking writes for backpressure detection. Always use a `context.Context` to propagate cancellation through your goroutines.

- **Protobuf:** Use the `google.golang.org/protobuf/encoding/protojson` package if you need to debug OTLP payloads by converting them to JSON.
- **Error Handling:** In gRPC methods, always return errors wrapped with `status.Error(codes.Code, message)` to provide meaningful status codes to clients.
- **Performance:** Pre-allocate slices with known capacities (e.g., `make([]*LogRecord, 0, MAX_BATCH_SIZE)`) to avoid repeated allocations during batching.
- **Testing:** Use the `testing` package with goroutines to simulate concurrent load. The `github.com/stretchr/testify/assert` package is helpful for assertions.

F. Milestone 2 Checkpoint

After implementing the core ingestion pipeline:

1. **Run the gRPC server:** `go run cmd/server/main.go`. It should start without error, listening on port 4317.
2. **Send a test OTLP trace:** Use a simple Go program with the OpenTelemetry Go SDK or a tool like `grpcurl` to send a protobuf payload. A quick test can be done by generating a mock `ExportTraceServiceRequest` in a test file.
3. **Verify pipeline flow:** Add debug logs in each stage (Receiver, Batcher, Normalizer, Router). You should see the log messages in order as a trace passes through.
4. **Test backpressure:** Modify your test client to send data as fast as possible in a loop. Monitor the pipeline's memory usage. You should see the Receiver start rejecting requests (log a message or metric) once the internal buffer is full, and the client should receive `UNAVAILABLE` errors.
5. **Expected Output:** No data will be stored yet (storage engines are not built), but the Router should log where it would send each batch. The key success criteria is that the pipeline **accepts data, processes it through all stages without panics, and applies backpressure under load**.

Signs of Trouble:

- **Server crashes on startup:** Likely a missing protobuf dependency or port conflict.
- **Data disappears, no debug logs after Receiver:** The channel between Receiver and Batcher is probably unbuffered or the Batcher goroutine isn't started.
- **Memory grows indefinitely:** The internal channel buffers might be too large, or a downstream stage is blocked, causing infinite queueing. Check for deadlocks.
- **Client hangs forever:** The gRPC server might be blocking on a full channel without a timeout. Ensure you use `select` with `ctx.Done()` or a timeout.

6. Component: Storage Engines

> **Milestone(s):** Milestone 3 (Multi-Signal Storage)

Mental Model: Specialized Warehouses

Imagine a massive industrial complex with three distinct, specialized warehouses built side-by-side, each with a unique internal organization and machinery tailored to a specific type of cargo. A central **master index** sits at the crossroads, acting as a directory that tells you which items are stored in which warehouse and how they're related.

- **The Document Warehouse (Log Storage):** This building is designed for storing millions of semi-structured text documents (log records). It has a massive, automated shelving system (inverted index) that allows you to find any document containing specific words or phrases in seconds. Each box is stamped with metadata tags (resource and span attributes) and a barcode (unique ID) for quick location. Before a box is shelved, its details are recorded in an unerasable logbook (Write-Ahead Log) to ensure nothing gets lost in transit.
- **The Time-Series Vault (Metric Storage):** This is a climate-controlled, high-security vault. Inside, countless tiny meters are constantly ticking, recording measurements like temperature, pressure, and flow (counter, gauge, histogram values). The data is stored in dense, compressed blocks organized by time, not by content. The vault has automated machinery for condensing older data into summaries (downsampling) and securely shredding data past its expiration date (retention policies).
- **The Graph Archive (Trace Storage):** This facility stores complex 3D models of interconnected nodes (spans). It doesn't just store the individual pieces; it preserves the blueprint of how they fit together into a tree (trace). The archive is optimized for two tasks: rapidly storing individual components as they arrive, and later efficiently retrieving and reassembling an entire model (trace tree) on demand, following the parent-child pointers (`parent_span_id`) left on each component.

The **Cross-Signal Index** is the master index at the complex's entrance. It's a giant, constantly updated ledger that records: "Trace `T123abc` has related log entries stored in Document Warehouse at shelf L-45, L-46, and L-47, and a representative sample (exemplar) stored in Time-Series Vault at meter block M-889." This index doesn't store the actual data; it stores pointers, enabling the Query Engine (our universal librarian) to efficiently gather related items from all three warehouses in a single operation.

This **polyglot persistence** approach—using different storage engines for different data shapes and access patterns—is fundamental. Trying to force trace trees into a time-series database or log text into a graph store would be like trying to store liquid in a bookshelf: inefficient and messy. By specializing, each engine can apply optimizations (compression, indexing, layout) that would be impossible in a one-size-fits-all database.

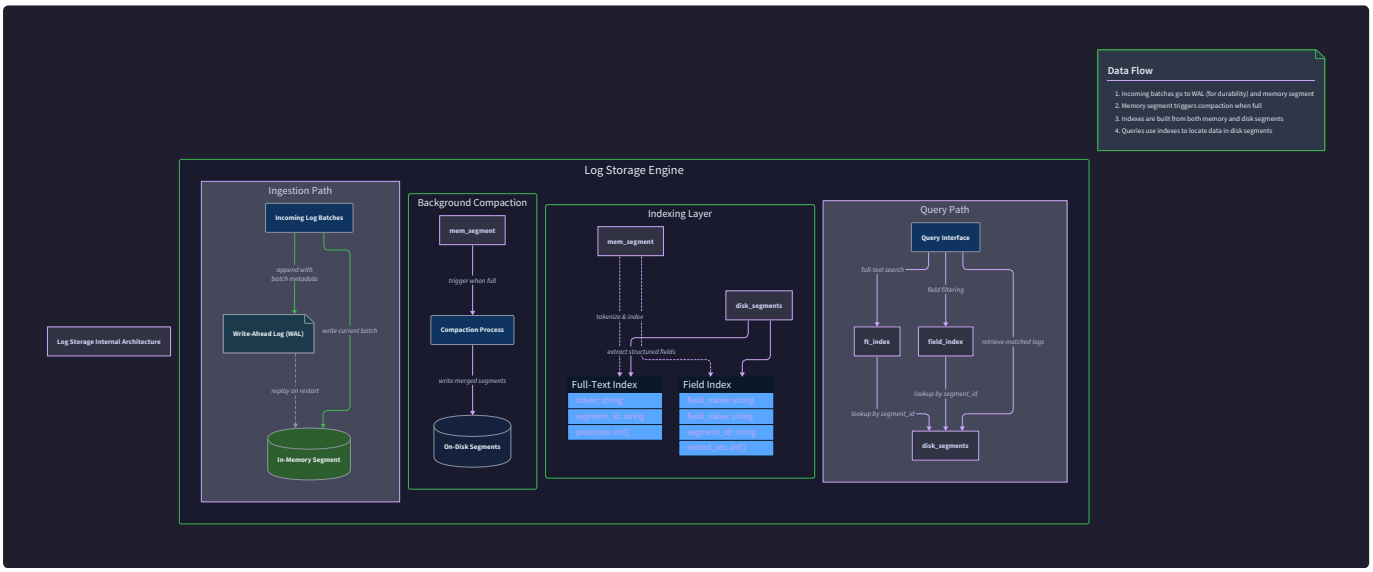
Log Storage: Full-Text Search with Inverted Index

The Log Storage Engine's primary job is to enable **interactive debugging** by allowing engineers to quickly search through massive volumes of log data using free-text keywords and structured filters (e.g., `service_name:"api-gateway" AND severity:ERROR`). The core challenge is balancing write throughput for ingestion

with low-latency read performance for search.

Internal Architecture & Data Flow

The design follows a **segment-based architecture**, common in search engines like Apache Lucene.



1. **Write-Ahead Log (WAL):** Every incoming `LogRecord` batch is first appended sequentially to a persistent WAL file. This is a crash-recovery mechanism. If the process crashes after acknowledging a write but before indexing it, the WAL can be replayed on startup to recover all data. The WAL entry contains the raw, serialized `LogRecord`.
2. **In-Memory Segment (MemTable):** `LogRecord` data is simultaneously added to an in-memory buffer structure, often called a memtable. This buffer is organized for fast writes and is the primary source for read queries while data is fresh. It holds data in a format ready for searching (e.g., maps of terms to document IDs).
3. **Field Index:** For structured fields we frequently filter on (`service_name`, `severity`, `trace_id`, `environment`), we build an inverted index. Think of it as a lookup table: for the key `service_name=api-gateway`, it stores a list of all `LogRecord` IDs that have that value. This allows the query "find all logs from api-gateway" to skip scanning all records.
4. **Full-Text Index:** For the unstructured `body` field, we build a separate inverted index. The `body` text is tokenized (split into words, normalized). For each unique token (e.g., "connection", "timeout"), the index stores a list of `LogRecord` IDs containing that token. This enables sub-second keyword searches across millions of logs.
5. **Compaction to Immutable Segment:** When the in-memory segment reaches a configured size (e.g., 10,000 records or 10MB), it is "frozen," marked as read-only, and flushed to disk as an **immutable segment** file. This process serializes both the raw log data and the in-memory indexes to a compact, read-optimized format. New writes then go to a new, empty memtable.
6. **Segment Merging:** Over time, many small immutable segments accumulate. A background process periodically merges smaller segments into larger ones. During merging, it can also apply log retention policies, deleting data older than `DEFAULT_RETENTION_DAYS`. This keeps the number of segments to search manageable.

Data Structures

Name	Type	Description
`LogStorageEngine`	Component	The main orchestrator managing WAL, memtables, segments, and queries.
`WriteAheadLog`	Interface	Append-only log for durability. Methods: `Append(record *LogRecord) (offset int64, err error)`, `ReadFrom(offset int64) ([]*LogRecord, error)`, `Truncate(offset int64) error`.
`InMemorySegment`	Struct (Memtable)	Holds recent logs in memory. Fields: `maxSize int`, `logs map[string]*LogRecord` (keyed by ID), `fieldIndex map[string]map[string][]string` (e.g., `fieldIndex["service_name"]["api-gateway"] = [id1, id2]`), `fullTextIndex map[string][]string` (e.g., `fullTextIndex["timeout"] = [id1, id3]`).
`ImmutableSegment`	Struct	A read-only segment on disk. Fields: `segmentID uuid.UUID`, `minTime, maxTime int64`, `dataFile string` (compressed log records), `fieldIndexFile string`, `fullTextIndexFile string`.
`LogQuery`	Struct	Represents a user's search request. Fields: `QueryString string` (e.g., "timeout AND service:api"), `StartTime, EndTime int64`, `Filters map[string]string` (structured filters), `Limit int`, `Offset int`.

Architecture Decision Record: Segment-Based Indexing vs. Monolithic Database

Decision: Use a Segment-Based Architecture with Immutable On-Disk Segments

- **Context**: We need a log storage engine that supports high write throughput (ingestion) and low-latency, complex read queries (search). The data is immutable once written.
- **Options Considered**:
  1. **Traditional RDBMS (e.g., PostgreSQL with GIN index)**: Use a single, mutable database table with a full-text search index.
  2. **Monolithic Search Engine (single, mutable index)**: Maintain one global in-memory and on-disk index that is updated with every write.
  3. **Segment-Based Architecture (LSM-tree inspired)**: Use in-memory buffers (memtables) that are periodically flushed to immutable, sorted segment files on disk. Queries merge results from all segments.
- **Decision**: Option 3, segment-based architecture.
- **Rationale**:
  - **Write Performance**: Writes are extremely fast—they only append to a WAL and update an in-memory map. There is no overhead of updating complex on-disk B-tree structures for every insert (as in an RDBMS).
  - **Read Scalability**: Immutable segments are highly cache-friendly and can be searched in parallel. The system naturally supports partitioning data by time.
  - **Simplicity of Concurrency**: Immutable segments require no locks for reads. The only write lock needed is for the active memtable swap.
  - **Predictable Performance**: Background compaction controls write amplification and keeps query latency bounded by merging small segments into larger ones.
- **Consequences**:
  - **Eventual Consistency**: Data is only searchable after the memtable is flushed. This adds a small delay (seconds) between ingestion and searchability, which is acceptable for observability.
  - **Background Compaction Overhead**: Requires CPU and I/O for merging, but this can be throttled and scheduled during off-peak times.
  - **State Management**: The engine must maintain a manifest file listing all active segments and their time ranges.

Option	Pros	Cons	Chosen?
Traditional RDBMS	Strong consistency, ACID transactions, rich query language.	Write throughput often a bottleneck, full-text search performance can degrade at scale, operational complexity (tuning).	No
Monolithic Mutable Index	Simple conceptual model, low read latency for latest data.	Write performance degrades as index grows, expensive index updates block reads, risk of corruption.	No
Segment-Based Architecture	**Excellent write throughput, scalable reads via parallel segment search, simple concurrency model, natural support for time-based partitioning and retention.**	**Reads must merge multiple segments, eventual consistency, requires background compaction logic.**	**Yes**

Query Execution Algorithm

When the `LogStorageEngine` receives a `LogQuery`:

1. **Parse & Plan**: The `QueryString` is parsed into a list of required tokens (for full-text) and structured filters. The engine identifies which segments (both the active `InMemorySegment` and on-disk `ImmutableSegment`s) overlap with the query's time range.
2. **Fan-Out Search**: The engine concurrently searches each relevant segment: a. For the `InMemorySegment`, it checks the `fieldIndex` and `fullTextIndex` maps. b. For an `ImmutableSegment`, it loads the relevant index blocks from the `fieldIndexFile` and `fullTextIndexFile` into memory (cached) and performs the lookup.
3. **Merge Results**: Each segment returns a set of matching `LogRecord` IDs. These sets are intersected (for AND logic) or unioned (for OR logic) across segments.
4. **Fetch Data & Paginate**: For the final list of IDs, the engine retrieves the actual `LogRecord` data from the segment's data store (memory or disk file). It applies the `Limit` and `Offset` to paginate the results before returning.

Common Pitfalls

⚠️ Pitfall: Forgetting the WAL Before the Memtable

- **Description**: Writing a log record directly to the memtable and returning success to the client before the write is durable. If the process crashes, the data is lost forever, despite the client receiving an acknowledgment.
- **Why it's wrong**: It violates the durability guarantee of the ingestion pipeline, leading to silent data loss that is incredibly difficult to debug.
- **Fix**: **Always** follow the sequence: 1) Serialize the record, 2) Append it to the WAL and call `fsync` (or equivalent) to ensure it's on disk, 3) *Then* add it to the in-memory memtable, 4) Finally, acknowledge the write to the client.

⚠️ Pitfall: Not Bounding In-Memory Usage

- **Description**: Letting the `InMemorySegment` grow without limit, eventually consuming all RAM and causing the process to be killed by the OS (OOM).
- **Why it's wrong**: Causes catastrophic, ungraceful failure of the entire storage engine.

- **Fix**: Enforce a strict `maxSize` (in bytes or record count) on the memtable. When this limit is reached, immediately block new writes and synchronously flush the memtable to disk as a new immutable segment before accepting more data. This applies backpressure to the ingestion pipeline.

Metric Storage: Time-Series Database

The Metric Storage Engine is optimized for storing and retrieving numerical samples that are ordered by time. Its defining challenges are **high cardinality** (unique time series defined by `name` + `resource` + `attributes` combinations), efficient **compression**, and managing data **retention**.

Core Design: Series, Chunks, and Blocks

- **Series**: A single time series is uniquely identified by its metric `name`, `resource` (service, instance), and `attributes` map. The engine maintains a **series index** (e.g., in a `map[string]uint64` where the key is a hash of the identifier and the value is a numeric series ID).
- **Chunks**: Data for a series is stored in chunks, covering a fixed time window (e.g., 2 hours). A chunk contains compressed `(timestamp, value)` pairs. Using chunks allows us to efficiently query for a specific time range without decompressing an entire series' history.
- **Blocks**: Multiple chunks (potentially across many series) are grouped into a **block** for a larger time period (e.g., 1 day). A block is a self-contained directory on disk containing the chunk files for its time range and metadata (which series are present, time bounds). This is the unit of retention and deletion.

Compression Strategy (Gorilla)

For each chunk, we apply a compression algorithm like **Gorilla**, which is designed for time-series floats in memory but can be adapted for on-disk storage. The key ideas:

1. **Timestamp Compression**: Store the first timestamp in full. For subsequent timestamps, store the delta (difference) from the previous timestamp. Since metrics are often collected at regular intervals, these deltas are often constant and can be encoded in very few bits.
2. **Value Compression**: Use XOR compression. Store the first value as a full 64-bit float. For subsequent values, compute the XOR with the previous value. If the XOR is zero (value is identical), store a single '0' bit. Otherwise, store a '1' bit followed by the XOR value, often with leading and trailing zero bits stripped for further compression.

This achieves ~1.37 bytes per data point compared to 16 bytes (8 for timestamp, 8 for value) for raw storage—a >10x compression ratio.

Downsampling and Retention

To manage storage costs over long periods, we implement a **tiered storage** approach:

- **Raw Data**: Stored at original resolution (e.g., per-second) for a short period (e.g., 7 days).
- **Downsampled Data**: After a period (e.g., 24 hours), a background job aggregates raw data into lower-resolution chunks (e.g., 5-minute averages, maximums, minimums). This downsampled data is kept for much longer (e.g., 30 days or a year).
- **Retention Policy Engine**: A scheduler periodically scans block directories. Any block whose maximum timestamp is older than the configured retention period for its resolution is deleted from disk.

Data Structures

Name	Type	Description
`MetricStorageEngine`	Component	Manages series index, chunk writes, block creation, and queries.
`Series`	Struct	Metadata for a single time series. Fields: `id uint64`, `metricName string`, `resource ResourceAttributes`, `labels map[string]string` (attributes), `lastTimestamp int64`, `lastValue float64` (for Gorilla compression state).
`Chunk`	Struct	A compressed batch of points for one series. Fields: `seriesID uint64`, `minTime, maxTime int64`, `data []byte` (compressed Gorilla stream), `exemplars []Exemplar` (stored separately, not compressed).
`Block`	Struct	A directory containing chunks for a time range. Fields: `blockID uuid.UUID`, `minTime, maxTime int64`, `resolution Duration` (e.g., 1s, 5m), `chunkFiles []string`.
`MetricQuery`	Struct	A query for metric data. Fields: `metricName string`, `resourceFilter ResourceAttributes`, `labelMatchers map[string]string`, `startTime, endTime int64`, `aggregation string` (e.g., "avg", "sum", "rate"), `step int64` (query resolution step in seconds).

Architecture Decision Record: Rollup-on-Write vs. Rollup-on-Read

Decision: Perform Downsampling (Rollup) in a Scheduled Background Job (Rollup-on-Write)

- **Context**: We need to support queries over long time ranges (weeks, months) without scanning massive amounts of high-resolution data, which would be slow and I/O intensive.
- **Options Considered**:
  1. **Rollup-on-Read**: Store only raw data. At query time, if the user asks for a long range with a low step (e.g., 1 day of data with 5-minute resolution), the query engine dynamically aggregates the raw data in memory.
  2. **Rollup-on-Write (Background Job)**: Store raw data. A separate, scheduled process periodically reads raw data, computes downsampled aggregates (5-min avg, max, min), and writes them as new, separate time series with a lower resolution tag.
- **Decision**: Option 2, Rollup-on-Write via a background job.
- **Rationale**:
  - **Predictable Query Performance**: Queries over historical ranges are served from pre-computed, compact downsampled data, ensuring consistently fast response times regardless of the query range.
  - **Reduced Read I/O & CPU**: The expensive aggregation work is done once, offline, rather than repeatedly for every query. This saves substantial compute resources.
  - **Simpler Query Engine**: The query logic doesn't need complex dynamic aggregation pipelines; it just selects the appropriate resolution data source.
- **Consequences**:
  - **Increased Storage Write Amplification**: Data is written twice (once as raw, once as downsampled). This is a trade-off for read performance.
  - **Eventual Consistency for Historical Views**: There's a lag (e.g., 1 hour) before recent raw data appears in the downsampled view. This is acceptable for historical trend analysis.
  - **Job Management Overhead**: Requires a scheduler, monitoring, and failure handling for the background rollup job.

Option	Pros	Cons	Chosen?
Rollup-on-Read	No extra storage, always uses the freshest data for aggregation.	Query latency scales poorly with data volume, heavy repeated CPU load, complex query-time logic.	No
Rollup-on-Write (Background)	**Fast, consistent queries on historical data, aggregation done once, simpler query engine.**	**Extra storage for rolled-up data, data freshness lag for aggregates, needs background job.**	**Yes**

Exemplar Storage

As defined in the data model, a `MetricPoint` may contain an `Exemplar` linking to a trace. Storing this trace ID with every high-frequency data point would blow up cardinality. Therefore, we store exemplars separately from the compressed value stream. Each `Chunk` has a small, appended section or a sidecar file listing `Exemplar` records (trace_id, value, timestamp). This allows the correlation index (discussed later) to create a mapping from `trace_id` back to the metric chunk and offset where its exemplar is stored.

Trace Storage: Span Tree Reconstruction

Trace storage must handle two primary operations efficiently: 1) **ingesting individual spans** as they arrive (often out-of-order), and 2) **reconstructing the complete trace tree** on demand for visualization and root-cause analysis. A trace is a directed acyclic graph (tree) of spans linked by `parent_span_id`.

Design: Trace-Focused Indexing

We cannot afford to scan all spans to find those belonging to a given `trace_id`. Therefore, the primary index is on `trace_id`. Furthermore, to efficiently reconstruct the tree, we need to quickly find all spans for a trace *and* understand their parent-child relationships.

- **Trace Index**: A primary lookup table mapping `trace_id` -> `TraceMetadata`. The `TraceMetadata` contains summary information: `start_time` (min span start), `end_time` (max span end), `root_service_name`, `span_count`, and a list of `span_ids` belonging to the trace.
- **Span Storage**: Spans for a given trace are stored together, often in a columnar format optimized for the trace tree structure. A common approach is to store all spans for a trace in a single record (e.g., a Protobuf message or JSON array) keyed by `trace_id`. This makes retrieval of the entire trace a single read operation.
- **Service & Duration Indexes**: Secondary indexes are maintained to support queries like "find traces from service `api-gateway` that took longer than 2 seconds." These can be implemented as range-indexed maps (e.g., using an interval tree or a simple map from `service_name` to lists of `trace_id`s with their durations).

Span Tree Reconstruction Algorithm

When a query requests trace `T123`:

1. **Locate Trace**: Look up `T123` in the Trace Index to get its `TraceMetadata`.
2. **Fetch Span Data**: Using the information in the metadata (which might point to a specific file and offset), read the stored span data for trace `T123`. This is ideally one I/O operation.

3. **Build In-Memory Tree**: The fetched data is an array of `Span` objects. The engine iterates through them: a. Create a map `spanMap` keyed by `span_id` for quick lookup. b. Identify the root span(s) by finding spans where `parent_span_id` is empty (or "0"). There should typically be one. c. For each span, use its `parent_span_id` to find its parent in `spanMap` and attach it as a child. This builds the parent-child pointer structure.
4. **Return Tree**: The reconstructed root span, with its complete child hierarchy, is returned to the query engine.

Handling Out-of-Order & Delayed Spans

Spans can arrive minutes or even hours after a trace's "start" due to network delays or buffering in agents. The storage engine must handle this.

- **Trace Index as Mutable State**: The `TraceMetadata` is updated when a new, late-arriving span is ingested. `end_time` and `span_count` may need adjustment.
- **Append to Trace Record**: The late span is appended to the existing trace's span storage. If the trace data was stored in an immutable block, this may require a more complex "read-modify-write" operation or storing late spans in a separate "late arrivals" area that is merged at query time.

Data Structures

Name	Type	Description
`TraceStorageEngine`	Component	Manages trace and span storage, indexes, and queries.
`TraceMetadata`	Struct	Summary for a single trace. Fields: `traceID string`, `startTime, endTime int64`, `rootServiceName string`, `spanCount int32`, `spanIDs []string`, `storageLocation string` (e.g., file path + offset).
`SpanStorageBlock`	Struct	A block storing spans for many traces. Fields: `blockID uuid.UUID`, `minTime, maxTime int64`, `dataFile string` (e.g., columnar format like Parquet, or simple concatenated Protobuf).
`TraceQuery`	Struct	A query for traces. Fields: `serviceName string`, `operationName string`, `attributes map[string]string`, `minDuration, maxDuration Duration`, `startTime, endTime int64`, `limit int`.

Cross-Signal Correlation Index

This is the **master index** that binds the three specialized warehouses together. It's a secondary index that does not store telemetry data itself, but stores **relationships**—specifically, the mapping from a `trace_id` to the storage locations of all related logs and metric exemplars.

Why a Separate Index? We could, in theory, query each storage engine with a `trace_id` filter. The log engine would scan its field index for `trace_id=T123`, the metric engine would scan all exemplars, and the trace engine would look up its primary index. This is inefficient and slow for interactive debugging. The correlation index pre-computes these relationships at ingestion time, enabling near-instant lookups.

Index Structure The index is conceptually a key-value store where the key is the `trace_id` and the value is a `CorrelationRecord`.

- `CorrelationRecord` Fields:
  - `trace_id string`
  - `log_references []LogRef`
  - `exemplar_references []ExemplarRef`
- `LogRef` Fields (points to a log):
  - `log_id string` (unique ID of the log record)
  - `timestamp int64`
  - `storage_hint string` (e.g., "segment_789/offset_456") to help the log storage engine locate the record quickly.
- `ExemplarRef` Fields (points to a metric exemplar):
  - `metric_name string`
  - `series_id uint64`
  - `chunk_id string`
  - `exemplar_index int` (position within the chunk's exemplar list)
  - `timestamp int64`
  - `value float64`

Maintenance at Ingestion Time The index is updated by the Ingestion Pipeline's Normalizer component as it processes each item:

1. **For a `LogRecord`**: If `log.trace_id` is not empty, the normalizer generates a `LogRef` and appends it to the `CorrelationRecord` for that `trace_id`.
2. **For a `MetricPoint`**: If `metric.exemplar.trace_id` is not empty, the normalizer generates an `ExemplarRef` and appends it.
3. **For a `Span`**: The index is not updated directly. The existence of a span is already the primary record in the Trace Storage. However, when a trace's metadata is first created in the Trace Index, a corresponding empty `CorrelationRecord` might be initialized.

Query Pattern When a user asks, "Show me all logs and the representative trace for errors in service X in the last 5 minutes," the Query Engine:

1. Finds error logs for service X (via Log Storage).

2. For each log, extracts its `trace_id`.
3. Uses the **Cross-Signal Index** to instantly fetch the `CorrelationRecord` for that `trace_id`.
4. From the `CorrelationRecord`, it can:
    - Immediately retrieve *all other logs* for the same trace (using the `log_references`), not just the error ones.
    - Fetch the full trace tree from Trace Storage.
    - Find related metric exemplars that recorded samples during the trace's execution.

This transforms a sequential, multi-engine search into a fast, indexed lookup, making cross-signal correlation practical at human-interactive speeds.

Storage Considerations This index will be very write-heavy (updated with every log and metric that has a trace context) and read-moderately. It can be implemented using a key-value store like RocksDB or Badger, which offer high write throughput and efficient range queries (useful for cleaning up old records based on `timestamp`). The index must also support TTL (Time-To-Live) or be periodically pruned based on trace retention periods to avoid unbounded growth.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (for learning)	Advanced Option (for production)
Log Storage	Build your own segment-based engine with Go's `map` for in-memory indexes and `bolt` / `bbolt` for immutable segment storage. Use `bleve` for full-text indexing.	Use `ClickHouse` (columnar, excellent for logs) or `Elasticsearch` (mature search) as an embedded library or separate service.
Metric Storage	Implement Gorilla compression in pure Go. Use `bbolt` for chunk/block storage. Use a `sync.Map` for the series index.	Integrate `Prometheus TSDB` library (mature, production-grade) or use `VictoriaMetrics` as a storage backend.
Trace Storage	Store traces as JSON files per trace ID. Use `bbolt` for the Trace Index.	Use `Jaeger`'s storage plugins (Cassandra, Elasticsearch) or `Tempo` (object storage backend).
Cross-Signal Index	Use `bbolt` (key-value store) for the correlation map.	Use `BadgerDB` (higher performance, pure Go) or integrate with the same database used for logs/metrics if it supports secondary indexing.

B. Recommended File/Module Structure

```
project-root/
├── internal/
│   ├── storage/
│   │   ├── engine.go                  # Interface: StorageEngine
│   │   ├── log/
│   │   │   ├── engine.go              # LogStorageEngine implementation
│   │   │   ├── wal.go                 # WriteAheadLog
│   │   │   ├── segment.go             # InMemorySegment & ImmutableSegment
│   │   │   ├── index.go               # Field & Full-Text index logic
│   │   │   └── query.go               # LogQuery execution
│   │   ├── metric/
│   │   │   ├── engine.go              # MetricStorageEngine
│   │   │   ├── series.go              # Series index & management
│   │   │   ├── chunk.go               # Chunk with Gorilla compression
│   │   │   ├── block.go               # Block management
│   │   │   └── downsample.go          # Background rollup job
│   │   ├── trace/
│   │   │   ├── engine.go              # TraceStorageEngine
│   │   │   ├── trace_index.go         # TraceMetadata index
│   │   │   ├── span_store.go          # Span storage (per trace)
│   │   │   └── tree_builder.go        # Span tree reconstruction
│   │   └── correlation/
│   │       ├── index.go               # CrossSignalCorrelationIndex
│   │       ├── record.go              # CorrelationRecord, LogRef, ExemplarRef
│   │       └── kv_store.go            # Wrapper around bbolt/Badger
└── pkg/telemetry/                     # Shared data models (LogRecord, etc.)
```

C. Infrastructure Starter Code: Write-Ahead Log

This is a complete, usable WAL implementation. The learner can copy this and focus on the storage engine logic.

```go
// internal/storage/log/wal.go

package log


import (

    "encoding/binary"

    "fmt"

    "os"

    "path/filepath"

    "sync"

    "time"

)


// WriteAheadLog ensures durability of log records before they are indexed.

type WriteAheadLog struct {

    file     *os.File

    filePath string

    mu       sync.Mutex

    offset   int64 // current write offset

}


// OpenWAL opens or creates a WAL file at the given directory.

func OpenWAL(dataDir string) (*WriteAheadLog, error) {

    walPath := filepath.Join(dataDir, fmt.Sprintf("wal_%d.log", time.Now().UnixNano()))

    f, err := os.OpenFile(walPath, os.O_CREATE|os.O_RDWR|os.O_APPEND, 0644)

    if err != nil {

        return nil, fmt.Errorf("failed to open WAL file: %w", err)

    }

    stat, err := f.Stat()

    if err != nil {

        return nil, fmt.Errorf("failed to stat WAL file: %w", err)

    }

    return &WriteAheadLog{

        file:     f,

        filePath: walPath,

        offset:   stat.Size(),

    }, nil

}


// Append writes a serialized record to the WAL and fsyncs.

func (w *WriteAheadLog) Append(data []byte) (int64, error) {

    w.mu.Lock()

    defer w.mu.Unlock()
```

```go
    // Write length prefix (8 bytes)
    lenBuf := make([]byte, 8)
    binary.BigEndian.PutUint64(lenBuf, uint64(len(data)))
    if _, err := w.file.Write(lenBuf); err != nil {
        return 0, fmt.Errorf("failed to write length to WAL: %w", err)
    }

    // Write the actual record data
    if _, err := w.file.Write(data); err != nil {
        return 0, fmt.Errorf("failed to write record to WAL: %w", err)
    }

    // Ensure data is flushed to disk
    if err := w.file.Sync(); err != nil {
        return 0, fmt.Errorf("failed to sync WAL: %w", err)
    }

    startOffset := w.offset
    w.offset += int64(8 + len(data))
    return startOffset, nil
}

// ReadFrom reads records starting at the given offset.
func (w *WriteAheadLog) ReadFrom(startOffset int64) ([][]byte, error) {
    w.mu.Lock()
    defer w.mu.Unlock()

    if _, err := w.file.Seek(startOffset, io.SeekStart); err != nil {
        return nil, fmt.Errorf("failed to seek WAL: %w", err)
    }

    var records [][]byte
    reader := bufio.NewReader(w.file)
    for {
        lenBuf := make([]byte, 8)
        if _, err := io.ReadFull(reader, lenBuf); err == io.EOF {
            break
        } else if err != nil {
            return nil, fmt.Errorf("failed to read length from WAL: %w", err)
        }
        recordLen := binary.BigEndian.Uint64(lenBuf)

        recordData := make([]byte, recordLen)
        if _, err := io.ReadFull(reader, recordData); err != nil {
```

```
            return nil, fmt.Errorf("failed to read record data from WAL: %w", err)
        }
        records = append(records, recordData)
    }
    return records, nil
}

// Close closes the WAL file.
func (w *WriteAheadLog) Close() error {
    w.mu.Lock()
    defer w.mu.Unlock()
    return w.file.Close()
}
```

D. Core Logic Skeleton Code

Log Storage Engine - Core Write Path

```
// internal/storage/log/engine.go
func (e *LogStorageEngine) WriteBatch(records []*telemetry.LogRecord) error {
    // TODO 1: For each record in the batch, serialize it and append to the WAL.
    //         Use w.Append(serializedRecord) and collect the offsets.
    // TODO 2: After all records are successfully written to the WAL, add each
    //         record to the active InMemorySegment.
    //         - Generate a unique logID (e.g., ULID or UUID).
    //         - Add to logs map: e.activeSegment.logs[logID] = record.
    //         - Update fieldIndex: for each attr in record.resource + record.attributes,
    //           add logID to the list for that key-value pair.
    //         - Update fullTextIndex: tokenize record.body, for each token,
    //           add logID to the list for that token.
    // TODO 3: Check if the active segment has reached its maxSize (e.g., 10,000 records).
    //         If yes, call e.rotateActiveSegment().
    // TODO 4: Return nil on success. If any step fails, return an error (the WAL ensures
    //         no acknowledged data is lost).
}
```

Metric Storage Engine - Gorilla Compression

```go
// internal/storage/metric/chunk.go

func (c *Chunk) encodePoint(timestamp int64, value float64) ([]byte, error) {

    // TODO 1: If this is the first point in the chunk, store timestamp and value

    //         as full 64-bit integers (for timestamp) and float64 (for value).

    // TODO 2: For subsequent points, calculate delta of time: delta = timestamp - c.lastTimestamp.

    //         - If delta is 0, write a '0' bit.

    //         - Else, encode the delta as a variable-length integer (see Gorilla paper).

    // TODO 3: Calculate XOR of current value with c.lastValue.

    //         - If XOR is 0, write a '0' bit.

    //         - Else, write a '1' bit, then encode the XOR value:

    //           a) Count leading zeros and trailing zeros in the XOR.

    //           b) Write the number of leading zeros (5 bits).

    //           c) Write the length of the meaningful XOR bits (6 bits).

    //           d) Write the meaningful XOR bits (excluding leading/trailing zeros).

    // TODO 4: Update c.lastTimestamp and c.lastValue.

    // TODO 5: Append the encoded bits to the chunk's data byte slice.

    // Hint: Use a bit writer that can write individual bits to a byte buffer.

}
```

Trace Storage Engine - Tree Reconstruction

```go
// internal/storage/trace/tree_builder.go

func BuildTreeFromSpans(spans []*telemetry.Span) (*telemetry.Span, error) {

    // TODO 1: Create a map: spanMap := make(map[string]*telemetry.Span)

    // TODO 2: Iterate through spans, add each to spanMap keyed by span.span_id.

    // TODO 3: Identify the root span(s). Iterate through spans again, find those

    //         where span.parent_span_id is empty string or "0". There should be exactly one.

    //         If zero or multiple, handle as an error or pick one based on earliest start_time.

    // TODO 4: For each span (excluding the root), find its parent in spanMap using

    //         span.parent_span_id. If parent exists, attach the current span as a child

    //         of the parent (you may need to add a Children []*Span field to the Span struct).

    //         If parent not found, this span is an orphan; you can either attach it to the root

    //         as a child or store it separately.

    // TODO 5: Return the root span, which now has its full child hierarchy attached.

}
```

Cross-Signal Index - Update on Ingestion

```go
// internal/storage/correlation/index.go

func (idx *CrossSignalCorrelationIndex) IndexLogRecord(log *telemetry.LogRecord, storageHint string) error {

    // TODO 1: Check if the log has a trace context: if log.trace_id == "", return nil.

    // TODO 2: Generate a LogRef with:

    //         - log_id: a unique ID for the log (could be ULID based on timestamp).

    //         - timestamp: log.timestamp.

    //         - storage_hint: the provided hint (e.g., segment ID and offset).

    // TODO 3: Use the trace_id as the key. Fetch the existing CorrelationRecord from the KV store.

    //         If it doesn't exist, create a new one with empty slices.

    // TODO 4: Append the new LogRef to the record's log_references slice.

    // TODO 5: Write the updated CorrelationRecord back to the KV store under the trace_id key.

    // TODO 6: Consider batching these updates to reduce write pressure on the KV store.

}
```

E. Language-Specific Hints (Go)

- Use `sync.RWMutex` in the `InMemorySegment` to allow concurrent reads (from queries) and exclusive writes.
- For Gorilla compression, manage a `bitWriter` that buffers bits and flushes to bytes. The `encoding/binary` package can help with writing uint64.
- When storing `ImmutableSegment` to disk, consider using Go's `gob` or `json` encoding for the index maps, but for production, a more compact format like Protobuf is better.
- For the series index in metric storage, use a `sync.Map` if you have a high number of series that are mostly read-heavy after initial creation.
- Use `filepath.Join` for all filesystem paths to ensure cross-platform compatibility.
- When reading/writing large files, use buffered I/O (`bufio.Reader/Writer`) and consider memory-mapped files for immutable segment access.

F. Milestone Checkpoint After implementing the core of Milestone 3 (storage engines), you should be able to:

1. **Ingest and Retrieve Logs**: Run a test that writes 1000 `LogRecord` objects with varied `body` text and attributes, then execute a search query for a specific keyword. Verify the correct subset is returned in < 100ms.
   - Command: `go test ./internal/storage/log/... -v -run TestSearch`
   - Expected: Tests pass, and you can see the search results match the ingested data.
2. **Store and Query Metrics**: Write a series of `MetricPoint` values for a counter that increments every second for 5 minutes. Query for the series and verify the values are returned correctly, and that the on-disk size is significantly smaller than the raw data size (demonstrating compression).
   - Command: `go test ./internal/storage/metric/... -v -run TestQueryRange`
   - Expected: The queried values match the input, and a check of the data directory shows compressed chunk files.
3. **Reconstruct a Trace**: Create 5 `Span` objects that form a simple parent-child tree (A -> B, A -> C, B -> D, C -> E). Ingest them out of order. Query for the trace by `trace_id` and verify the returned root span has the correct, fully populated child hierarchy.
   - Command: `go test ./internal/storage/trace/... -v -run TestTreeBuilding`
   - Expected: The root span has 2 children (B, C), and child B has one child D, etc.
4. **Verify Correlation Index**: Ingest a log with a `trace_id` and a metric with an exemplar containing the same `trace_id`. Then, use a debug utility to look up that `trace_id` in the correlation index and verify both the `LogRef` and `ExemplarRef` are present.
   - Command: Write a small Go program that uses the `CrossSignalCorrelationIndex` to lookup a known trace ID and print the references.
   - Expected: The program prints the storage hints for both the log and the exemplar.

If logs don't appear in search, check that the WAL write succeeded before adding to the memtable. If metric queries are slow, verify chunk time ranges are being used to limit scans. If trace trees are incomplete, ensure the `parent_span_id` field is being correctly parsed and matched.

---

5. Component: Query Engine

> **Milestone(s):** Milestone 4 (Unified Query Interface)

The Query Engine is the **command center** of the observability platform, providing a unified interface to interrogate the specialized storage systems. While the ingestion pipeline handles the *write* path, and the storage engines organize data for efficient retrieval, the query engine orchestrates the complex *read* path,

transforming user questions into answers by intelligently navigating across logs, metrics, and traces. This component directly enables the core observability promise: moving from "something is wrong" to "this specific interaction between services X and Y at time Z is the root cause."

Mental Model: The Universal Library Catalog

Imagine a vast research library composed of three specialized wings:

1. **The Log Wing**: Contains billions of diary entries (logs) from every service, indexed by the words they contain and the metadata (who wrote it, when, and under what transaction).
2. **The Metrics Wing**: Houses precise numerical ledgers (metrics) recording system vitals over time, like request counts and latencies, organized by the specific service and operation they measure.
3. **The Trace Wing**: Archives detailed family trees (traces) of requests as they flow through the system, documenting parent-child relationships between spans.

A researcher (the user) arrives with a complex question: *"Show me all error diary entries from last Tuesday that are related to slow purchase transactions initiated from the mobile app."* Manually searching each wing would be impossible. Instead, they consult the **Universal Library Catalog** — our Query Engine.

The catalog is not a copy of all the data. It's a sophisticated index and a set of procedures. It knows that:

- To find "error diary entries," it must go to the Log Wing and use its full-text and field indexes.
- To find "slow purchase transactions," it must go to the Trace Wing and query for traces with a specific operation name and duration.
- The **connection** between these two searches is the `trace_id`. The catalog maintains a special cross-reference index (the Cross-Signal Correlation Index) that maps a `trace_id` to all related log entries and metric samples.

The librarian (the Query Execution Planner) formulates a plan: 1) First, find the relevant slow traces and collect their IDs. 2) Use those IDs to look up the associated log entries in the cross-reference index. 3) Fetch the full log records from the Log Wing. 4) Filter them to only show errors. This plan is executed efficiently, merging results from different storage systems into a single, coherent answer for the researcher.

This mental model clarifies the Query Engine's role: it's a **planner, coordinator, and unifier**, not a monolithic data store. It leverages the specialized query capabilities of each storage engine (full-text search, time-series aggregation, graph traversal) and weaves them together using correlation IDs.

Query Language and API Design

The query interface must balance expressiveness for power users with simplicity for common tasks. It exposes two primary access methods: a RESTful HTTP/JSON API for broad compatibility and a gRPC API for performance-critical programmatic use. Both APIs are built on a shared, structured query language.

Query Language Specification

The core query is a **filter expression** that can be applied across signals, with signal-specific extensions. The base structure is a JSON object (or Protocol Buffer message) specifying the signal type, time range, and a set of filters.

Base Query Structure:

Field	Type	Required	Description
`signal`	`string`	Yes	One of: `"logs"`, `"metrics"`, `"traces"`, or `"cross_signal"`. Determines which storage engines are targeted and the interpretation of other fields.
`startTime`	`int64`	Yes	Start of the query time range in Unix nanoseconds.
`endTime`	`int64`	Yes	End of the query time range in Unix nanoseconds.
`filters`	`Array<Filter>`	No	A list of filter predicates to apply. All filters are combined with a logical AND.
`limit`	`int32`	No	Maximum number of primary results to return. Defaults to 100.
`cursor`	`string`	No	Opaque pagination token returned by a previous query to fetch the next page of results.

Filter Specification: A filter is a predicate on a telemetry attribute. It uses a simple, typed expression language.

Field	Type	Required	Description
`field`	`string`	Yes	The attribute path to filter on. For logs/traces: e.g., `"resource.service_name"`, `"body"` (for logs), `"attributes.http.status_code"`. For metrics: `"name"`, `"resource.environment"`, `"attributes.operation"`.
`operator`	`string`	Yes	One of: `"equals"`, `"not_equals"`, `"contains"` (string substring), `"regex"` (string regex match), `"greater_than"`, `"less_than"`, `"greater_than_eq"`, `"less_than_eq"`, `"exists"`.
`value`	`any`	Yes	The value to compare against. Type must be compatible with the operator (string, number, boolean).

Signal-Specific Extensions:

- **Log Queries (`signal: "logs"`)**: Use a `LogQuery` structure. Adds a `queryString` field for free-text search (like `"ERROR timeout"`), which is translated into `contains` / `regex` filters against the `body` field. Results are ordered by `timestamp` descending.
- **Metric Queries (`signal: "metrics"`)**: Use a `MetricQuery` structure. Requires a `metricName` . Adds `aggregation` (e.g., `"rate"` , `"sum"` , `"avg"` , `"p99"`) and `step` (query resolution in seconds) fields for time-series queries. Returns a list of time-series data points.
- **Trace Queries (`signal: "traces"`)**: Use a `TraceQuery` structure. Adds fields for `serviceName` , `operationName` , `minDuration` , and `maxDuration` to find traces by performance characteristics. Returns complete trace trees.
- **Cross-Signal Queries (`signal: "cross_signal"`)**: Use a `CrossSignalQuery` structure. Contains a primary `rootQuery` (of any signal type) and a list of `correlateWith` instructions (e.g., `{"signal": "logs", "linkField": "trace_id"}`). This instructs the engine to first execute the root query, extract the correlation IDs (like `trace_id`), and then fetch the correlated data from other signals.

> **Design Insight:** The query language is deliberately *declarative*, not imperative. The user specifies *what* they want (all error logs for slow traces), not *how* to get it. This allows the Query Execution Planner to optimize the access pattern—for example, deciding whether to fetch traces first and then logs, or to use the cross-signal index to find log IDs directly.

API Endpoints

HTTP REST API:

Method	Endpoint	Request Body	Returns	Description
`POST`	`/api/v1/query`	Base query JSON	`QueryResponse`	Primary query endpoint for logs, metrics, or traces.
`POST`	`/api/v1/query/cross`	`CrossSignalQuery` JSON	`CrossSignalQueryResponse`	Endpoint for explicit cross-signal correlation queries.
`GET`	`/api/v1/query/{cursor}`	-	`QueryResponse`	Retrieve the next page of results using a cursor from a previous query.

gRPC API: The gRPC service mirrors the HTTP API with protocol buffer messages for better performance and bi-directional streaming potential (e.g., for tailing logs).

```protobuf
service QueryService {
  rpc Query(QueryRequest) returns (QueryResponse);
  rpc QueryCrossSignal(CrossSignalQueryRequest) returns (CrossSignalQueryResponse);
  rpc QueryStream(QueryRequest) returns (stream QueryChunk); // For streaming results
}
```

Response Format & Pagination: All queries return a paginated response. For single-signal queries, the `QueryResponse` contains a `results` array (of `LogRecord` , `MetricPoint` , or `Span` objects), a `nextCursor` if more data is available, and a `total` count (if efficiently computable). For cross-signal queries, `CrossSignalQueryResponse` contains a map keyed by signal type, each with its own paginated result set and cursor.

Pagination uses an **opaque cursor** strategy. The cursor is a serialized structure containing the original query, the offset into the internal result set, and a checksum. This allows the query engine to resume execution without recalculating intermediate results from scratch, which is crucial for expensive cross-signal joins.

> **ADR: Cursor-Based vs. Offset-Based Pagination**
>
> **Context:** Users need to navigate through large result sets that may be the product of complex joins across storage engines. Simple `limit` / `offset` pagination becomes inefficient and inconsistent at scale.
>
> **Options Considered:**
>
> 1. **Offset-Based (`limit` / `offset`):** Client specifies result index to start from. Simple to implement.
> 2. **Cursor-Based (Opaque Token):** Server returns a token representing a precise point in the result stream. Client passes it back for the next page.
> 3. **Keyset Pagination:** Paginate using a sorted, unique key (e.g., `timestamp, id`). Client passes the last seen key.
>
> **Decision:** Use **Cursor-Based (Opaque Token) pagination**.
>
> **Rationale:** Offset-based pagination has known deficiencies at scale: an `OFFSET 1000000` requires the database to scan and discard the first million rows, which becomes prohibitively slow. It's also prone to consistency issues if data is added/deleted between pages. Keyset pagination is efficient but requires exposing and sorting on a stable key, which is complex for merged results from multiple sources. An opaque cursor allows the query engine to store its complete execution state (e.g., the last processed trace ID from step 1, the current index into the correlated log list from step 2). This makes resuming the query cheap and keeps internal abstractions clean.
>
> **Consequences:** The client cannot jump to an arbitrary page number, only move forward. The cursor payload size must be managed (e.g., by compressing or storing state server-side with a short-lived ID). The server must handle expired or malformed cursors gracefully.

Query Execution Planner

The Query Execution Planner is the brain of the Query Engine. It transforms a declarative query specification into an efficient sequence of operations against the storage engines and the cross-signal index. Its job is to minimize latency, resource consumption, and complexity for the user.

Planner Architecture

The planner operates in four distinct phases: **Parse & Validate**, **Plan Generation**, **Execution**, and **Result Assembly & Pagination**.

Phase 1: Parse & Validate

1. The incoming query (JSON or gRPC) is deserialized into the internal `Query` struct hierarchy (`LogQuery` , `MetricQuery` , etc.).
2. Syntax and semantic validation occurs: checking time ranges are valid, filter field names are recognized, and signal-specific required fields (like `metricName`) are present.
3. For cross-signal queries, the planner validates that the correlation path is possible (e.g., you can link logs to traces via `trace_id` , but linking a metric name directly to a log body is not supported).

Phase 2: Plan Generation This is where the optimizer works. The planner analyzes the query and the available indexes to produce a `QueryExecutionPlan` . A plan is a tree of `PlanNode` objects, where leaf nodes represent data access (e.g., `ScanLogsByTimeRange`), and intermediate nodes represent operations (e.g., `FilterByAttributes` , `JoinByTraceID`).

> **ADR: Centralized vs. Federated Query Planning**
>
> **Context:** We need to execute queries that may span multiple independent storage engines (logs, metrics, traces). We must decide where the "join" logic lives.
>
> **Options Considered:**
>
> 1. **Centralized Planning (Pull Model):** The Query Engine is the sole planner. It requests raw data from each storage engine using simple filters, pulls it into its own memory, and performs joins, sorting, and pagination itself.
> 2. **Federated Planning (Push Model):** The Query Engine sends more complex sub-queries to each storage engine, asking each to pre-filter, pre-sort, or even pre-join data locally before returning a partial result. The Query Engine then performs a final merge.
> 3. **Hybrid Model:** Use a centralized plan for simple queries and cross-signal joins, but push down operations like filtering by resource attributes or time-range to the storage engines to reduce data transfer.
>
> **Decision:** Use a **Hybrid Model** with aggressive **filter and projection pushdown**.
>
> **Rationale:** A pure pull model would overwhelm the Query Engine with data transfer and processing for simple queries (e.g., "fetch all logs from service A"). A pure push model requires each storage engine to implement complex join logic, violating separation of concerns. The hybrid model is pragmatic: push filters (`resource.service_name = "A"`) and limits to the storage engines, so they return only relevant data. The Query Engine retains control over the complex cross-signal correlation logic, which is its core competency. This balances efficiency with architectural clarity.
>
> **Consequences:** Each storage engine's query interface must support a basic filter expression language. The Query Engine's planner must be capable of decomposing a query into push-downable fragments. This adds complexity to the planner but dramatically improves performance for common queries.

For a cross-signal query like *"Find logs containing 'NullPointerException' for traces with duration > 2s from service 'checkout'."*, a naive plan might be:

1. Fetch all traces from 'checkout' with duration > 2s (could be millions).
2. For each trace, fetch all its logs (massive fan-out).
3. Filter logs for 'NullPointerException'.

An optimized plan, leveraging the cross-signal index, would be:

1. **Pushdown:** Fetch trace IDs for 'checkout' service where duration > 2s. (`TraceStorageEngine` returns just IDs).
2. **Index Join:** Use the `CrossSignalCorrelationIndex` to find all `log_references` for those trace IDs.
3. **Batched Fetch:** Bulk-fetch the actual `LogRecord` objects for those references from `LogStorageEngine` .
4. **Filter:** In the Query Engine, filter the logs for 'NullPointerException' in the body.
5. **Paginate:** Apply the `limit` and produce a cursor.

The planner uses heuristics and statistics (like estimated cardinality from the cross-signal index) to choose the best plan. The `QueryExecutionPlan` data structure captures this:

Plan Node Types:

Node Type	Description	Children
`ScanLogs`	Scans log storage with push-down filters.	None
`ScanTraces`	Scans trace storage with push-down filters.	None
`ScanMetrics`	Scans metric storage for time-series.	None
`Filter`	Applies in-memory filters to a stream of records.	1 (the source node)
`CorrelationJoin`	Joins two streams using a correlation key (e.g., `trace_id`). Uses the cross-signal index for efficiency.	2 (left and right source nodes)
`MergeSort`	Merges multiple sorted streams (e.g., results from different storage engines) into a single sorted stream.	N
`Limit`	Applies pagination limit and generates a cursor.	1

Phase 3: Execution The plan is executed by a **directed acyclic graph (DAG) executor**. It starts from the leaf nodes (`Scan*`), which initiate requests to the storage engines. Data flows upwards asynchronously through channels or iterators. The `CorrelationJoin` node is the most complex: it typically uses a hash-join strategy, building a hash map from the smaller stream (e.g., trace IDs) and then probing it with the larger stream (e.g., log references from the index).

Phase 4: Result Assembly & Pagination As records flow to the root `Limit` node, they are serialized into the API response format. The `Limit` node stops the execution after `N` results but must store enough internal state (in the cursor) to resume. For example, if the plan was to first get 100 trace IDs, then fetch their logs, and we hit the limit after 80 log records, the cursor must remember which trace ID we were processing and our position within its associated log list.

Common Pitfalls

⚠️ Pitfall: The "N+1 Query" Problem in Correlation

- **Description:** Implementing a correlation join by first fetching N trace IDs, then issuing N+1 individual queries to the log storage to fetch logs for each trace. This creates overwhelming load and latency.
- **Why It's Wrong:** It scales linearly and catastrophically. A query for 1000 traces generates 1001 network calls, making the system unusable.
- **How to Fix:** Always use **batched requests**. The `CorrelationJoin` node should collect correlation keys (trace IDs) and perform a single, batched lookup against the cross-signal index (`Index.GetLogRefs(traceIDs []string)`). Then, use another batched fetch to retrieve the actual log records from storage.

⚠️ Pitfall: Ignoring Result Ordering Across Signals

- **Description:** Merging results from logs (ordered by timestamp) and traces (ordered by start time) into a single, chronologically coherent timeline for the user.
- **Why It's Wrong:** Presenting interleaved logs and span data in a random order destroys the user's ability to understand causality and sequence within a request.
- **How to Fix:** For cross-signal queries where a unified timeline is desired, the `MergeSort` node must be configured to use a global timestamp (e.g., log timestamp, span start time) as the sort key. This requires all storage engines to support returning data sorted by time, which is a fundamental design requirement for observability data.

⚠️ Pitfall: Memory Exhaustion from Large Intermediate Results

- **Description:** The planner fetches 10,000 trace IDs into memory, then builds a hash map for all their associated log references (which could be 100x larger), exhausting the Query Engine's memory.
- **Why It's Wrong:** A single query can take down the entire query service, creating a denial-of-service vulnerability.
- **How to Fix:** Implement **query quotas and limits**. This includes configurable limits on the number of intermediate correlation keys, total records fetched, and query execution time. The executor must be able to **stream** results where possible, processing and discarding records after they pass through a node, rather than materializing entire result sets.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Query Parsing	Hand-written parser for the JSON query structure.	Use a formal grammar with a parser generator (e.g., ANTLR) for a more expressive custom query language.
Plan Execution	Synchronous execution with Go routines and channels for each plan node.	Use a reactive streams library or a DAG execution framework for better backpressure and resource management.
Result Serialization	Standard `encoding/json` and `google.golang.org/protobuf`.	Implement protocol buffers with gRPC streaming for high-throughput result delivery.
Cursor Management	Serialize plan state to JSON and Base64 encode it.	Store minimal server-side state (e.g., in a Redis cache) and return a short-lived UUID as the cursor.

B. Recommended File/Module Structure

```
project-root/
├── internal/
│   ├── query/
│   │   ├── engine.go             # Main QueryEngine struct and public API (ExecuteQuery)
│   │   ├── parser.go             # Query parsing and validation
│   │   ├── planner.go            # Plan generation and optimization
│   │   ├── executor.go           # DAG execution of PlanNode(s)
│   │   ├── plan/
│   │   │   ├── node.go           # PlanNode interface and base struct
│   │   │   ├── nodes.go          # Concrete node types (ScanLogs, Filter, Join, etc.)
│   │   │   └── cursor.go         # Cursor serialization/deserialization logic
│   │   ├── api/
│   │   │   ├── http.go           # HTTP handler (POST /api/v1/query)
│   │   │   ├── grpc.go           # gRPC service implementation
│   │   │   └── models.go         # Request/Response structs for API layer
│   │   └── language/
│   │       ├── ast.go            # Abstract syntax tree for the query language
│   │       └── filter.go         # Filter expression evaluation
│   ├── storage/                  # Storage engines (from previous milestone)
│   └── correlation/              # Cross-signal index
└── cmd/
    └── query-server/
        └── main.go               # Service entry point
```

C. Infrastructure Starter Code `internal/query/api/models.go` *– Core request/response structures:*

```go
package query

import (
    "time"
    "github.com/your-org/observability/internal/telemetry"
)

// Filter represents a single predicate in a query.
type Filter struct {
    Field    string `json:"field"`
    Operator string `json:"operator"` // "equals", "contains", "greater_than", etc.
    Value    any    `json:"value"`
}

// QueryRequest is the generic root of all queries.
type QueryRequest struct {
    Signal     string    `json:"signal"`
    StartTime  int64     `json:"startTime"`
    EndTime    int64     `json:"endTime"`
    Filters    []Filter  `json:"filters,omitempty"`
    Limit      int32     `json:"limit,omitempty"`
    Cursor     string    `json:"cursor,omitempty"`
}

// QueryResponse is the paginated generic response.
type QueryResponse struct {
    Results    []any           `json:"results"` // []LogRecord, []Span, or []MetricPoint
    NextCursor string          `json:"nextCursor,omitempty"`
    Total      int64           `json:"total,omitempty"` // Estimated total matches
}

// CrossSignalQueryRequest specifies a primary query and correlations.
type CrossSignalQueryRequest struct {
    RootQuery      QueryRequest          `json:"rootQuery"`
    CorrelateWith  []CorrelationSpec     `json:"correlateWith"`
}

type CorrelationSpec struct {
    Signal    string `json:"signal"`   // e.g., "logs"
    LinkField string `json:"linkField"` // e.g., "trace_id"
}
```

D. Core Logic Skeleton Code `internal/query/engine.go` *– The main coordinator:*

```go
package query

import (
    "context"
    "fmt"
)

// QueryEngine orchestrates query parsing, planning, and execution.
type QueryEngine struct {
    logStorage     storage.LogStorageEngine
    metricStorage  storage.MetricStorageEngine
    traceStorage   storage.TraceStorageEngine
    correlationIdx correlation.CrossSignalCorrelationIndex
    // ... potential configuration and metrics
}

// ExecuteQuery is the primary public method.
func (e *QueryEngine) ExecuteQuery(ctx context.Context, req *QueryRequest) (*QueryResponse, error) {
    // TODO 1: Validate the request (time range, signal type, limit bounds).
    // TODO 2: If a cursor is provided, deserialize it to resume a previous execution plan.
    //         Otherwise, proceed to parse the fresh request.
    // TODO 3: Parse the request into an internal query AST (Abstract Syntax Tree).
    // TODO 4: Pass the AST to the planner to generate a QueryExecutionPlan.
    // TODO 5: Execute the plan using the executor, passing in the storage engine clients.
    // TODO 6: Collect results up to the limit, applying final sorting if needed.
    // TODO 7: If more results are available, serialize the plan's execution state into a new cursor.
    // TODO 8: Construct and return the QueryResponse with results, cursor, and total estimate.
    return nil, fmt.Errorf("not implemented")
}

// ExecuteCrossSignalQuery handles queries that correlate across signals.
func (e *QueryEngine) ExecuteCrossSignalQuery(ctx context.Context, req *CrossSignalQueryRequest) (*CrossSignalQueryResponse, error) {
    // TODO 1: Execute the rootQuery using ExecuteQuery, but intercept the plan before it runs.
    // TODO 2: Analyze the correlation specifications. For each `CorrelateWith` spec:
    // TODO 3: Modify the execution plan to add a CorrelationJoin node. This node will:
    //         a. Take the correlation keys (e.g., trace_id) from the root result stream.
    //         b. Use the correlationIdx to fetch references (LogRef/ExemplarRef) for those keys.
    //         c. Fetch the actual correlated records from the appropriate storage engine.
    // TODO 4: Execute the modified plan.
    // TODO 5: Assemble results into a map keyed by signal type, with separate pagination info for each.
    return nil, fmt.Errorf("not implemented")
}
```

`internal/query/planner.go` – *Generating an efficient plan:*

```go
func (p *Planner) BuildPlan(queryAST *language.AST) (*plan.PlanNode, error) {
    // TODO 1: Based on the signal type in the AST, create the appropriate leaf Scan node.
    //         e.g., for "logs", create a ScanLogsNode.
    // TODO 2: Analyze filters in the AST. Determine which can be pushed down to the storage engine.
    //         (e.g., `resource.service_name` can be pushed; a regex on `body` cannot).
    // TODO 3: Attach push-down filters to the Scan node.
    // TODO 4: Add in-memory Filter nodes for any remaining filters that couldn't be pushed down.
    // TODO 5: If the query involves sorting, add a MergeSort node.
    // TODO 6: Finally, add a Limit node at the root to handle pagination and cursor generation.
    // TODO 7: (Advanced) Apply cost-based optimizations: reorder joins, choose index scans.
    return nil, fmt.Errorf("not implemented")
}
```

E. Language-Specific Hints (Go)

- Use `context.Context` for cancellation and timeouts throughout the execution pipeline to prevent runaway queries.
- Lever Go's concurrency: Each `PlanNode` can be a goroutine reading from input channels and writing to an output channel. Use `select` with `ctx.Done()` to allow graceful cancellation.
- For the hash join in `CorrelationJoin`, use `sync.Map` or a regular map with a `sync.RWMutex` if the build side (the smaller set of keys) fits in memory.
- Use `encoding/json.RawMessage` to avoid fully unmarshaling storage engine responses until necessary.

F. Milestone Checkpoint After implementing the Query Engine, you should be able to:

1. **Start the service:** `go run cmd/query-server/main.go`
2. **Test a log query:** Use `curl` to send a POST request:

```bash
curl -X POST http://localhost:8080/api/v1/query \
  -H "Content-Type: application/json" \
  -d '{
    "signal": "logs",
    "startTime": 1672531200000000000,
    "endTime": 1672617599000000000,
    "filters": [{"field": "resource.service_name", "operator": "equals", "value": "checkout"}],
    "limit": 10
  }'
```

`**Expected Output:** A JSON response with an array of up to 10 `LogRecord` objects from the "checkout" service. Look for a `nextCur`

3. **Test a cross-signal query:** Perform a more complex query linking traces and logs. Verify that the results for logs and traces are returned in the same response and that log records correctly reference the `trace_id` of the found traces.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Query returns no results, but data exists.	Time range mismatch (client using milliseconds, server expecting nanoseconds).	Log the parsed `startTime` / `endTime` in the query engine and compare to timestamps in storage.	Ensure consistent time units (nanoseconds) across all APIs and document it clearly.
Cross-signal query times out.	The "N+1 query" problem or a Cartesian product in a join.	Add debug logs to count the number of calls to storage engines. Check the generated execution plan.	Implement batched fetching in the `CorrelationJoin` node. Add limits to the number of correlation keys processed.
High memory usage during query execution.	The plan is materializing entire result sets before applying limit/filter.	Use Go's `pprof` to see what is allocating memory. Check if `Scan` nodes are streaming or loading all data.	Ensure the executor uses streaming iterators. Implement spilling to disk for large intermediate hash tables in joins.
Pagination cursor is huge or invalid.	The cursor is serializing too much internal state.	Print/LOG the cursor string after Base64 decoding.	Simplify the cursor to store only essential resume points (e.g., last seen ID and offset). Consider server-side cursor storage.

---

6. Component: Alerting & Anomaly Detection Engine

> **Milestone(s):** Milestone 5 (Alerting & Anomaly Detection)

Mental Model: The Vigilant Watchtower

Imagine a medieval fortress with multiple watchtowers overlooking the kingdom. Each tower has dedicated lookouts scanning different parts of the horizon with specialized equipment: one has a telescope for distant movement, another listens for unusual sounds, and a third watches for smoke signals. These lookouts follow specific rules passed down from the castle commander: "If you see more than ten enemy banners at the northern border for five consecutive minutes, light the red warning beacon" or "If you hear distress horns from the western woods, send riders to investigate."

Our alerting engine operates on the same principles. The **watchtowers** are the rule evaluators constantly monitoring telemetry streams. The **lookouts** are individual alert rules with their specific conditions. The **horizon** they scan is the observability data flowing through our platform—metrics showing system health, logs recording events, and traces mapping request flows. When a lookout spots a concerning pattern (smoke on the horizon), they don't immediately panic the entire castle. First, they verify it persists ("Is that smoke still there after two minutes?"). If confirmed, they trigger the appropriate **signals** (lighting beacons, sending riders, ringing bells) which correspond to our notification channels (Slack, PagerDuty, email).

The critical insight is that **correlated alerts are more valuable than isolated ones**. A single lookout shouting "I see smoke!" is less useful than three lookouts coordinating: "Smoke from the north tower, increased traffic on the eastern road, AND missing supply carts from the warehouse." Our alerting engine must therefore evaluate conditions across all three telemetry signals, just as a wise commander would synthesize reports from all watchtowers before deciding on a response.

Rule Evaluation Engine

The Rule Evaluation Engine is the continuous monitoring subsystem that periodically checks defined alert conditions against current and historical telemetry data. Unlike traditional monitoring systems that only evaluate threshold rules against metrics, our engine supports **multi-signal correlation rules** that can combine conditions across logs, metrics, and traces.

Core Architecture

The engine operates on a **pull-based evaluation model**: it periodically executes queries against the Query Engine rather than processing streams directly. This simplifies the architecture significantly, as the Query Engine already provides the unified interface to all telemetry data. The alternative—building a separate streaming evaluation pipeline—would duplicate much of the query functionality and add considerable complexity.

Decision: Pull-Based Rule Evaluation via Query Engine

- **Context**: We need to evaluate alert conditions against logs, metrics, and traces. The Query Engine already provides a unified interface to query all three signal types with filtering, aggregation, and correlation capabilities.
- **Options Considered**:
  1. **Pull-based evaluation**: Periodically execute queries against the Query Engine
  2. **Push-based streaming**: Build a separate stream processing pipeline that consumes telemetry as it arrives
  3. **Hybrid approach**: Use streaming for simple metric thresholds and pull-based for complex correlations
- **Decision**: Use pull-based evaluation via the existing Query Engine for all alert rules
- **Rationale**:
  - **Simplicity**: Reuses existing query infrastructure rather than building a parallel processing system
  - **Consistency**: Ensures alert evaluations see the same data views as users querying manually
  - **Flexibility**: Supports arbitrarily complex queries including cross-signal correlations
  - **Implementation speed**: Faster to implement for a learning project
- **Consequences**:
  - **Latency**: Alerts may trigger with delay equal to evaluation interval plus query execution time
  - **Load**: Periodic query bursts may impact query performance for users
  - **Completeness**: May miss brief spikes between evaluation intervals (mitigated by appropriate interval selection)

Option	Pros	Cons	Chosen?
Pull-based via Query Engine	Simple, consistent with user queries, supports complex correlations	Evaluation latency, periodic load spikes, may miss brief events	**Yes**
Push-based streaming	Low latency, continuous evaluation, efficient for simple rules	Complex to implement, duplicates query logic, hard to support cross-signal	No
Hybrid approach	Best of both worlds for different rule types	Most complex, two systems to maintain, consistency challenges	No

Alert Rule Data Model

An alert rule encapsulates what condition to check, how long it must persist, what severity to assign, and how to notify when triggered. The `AlertRule` struct contains all necessary configuration:

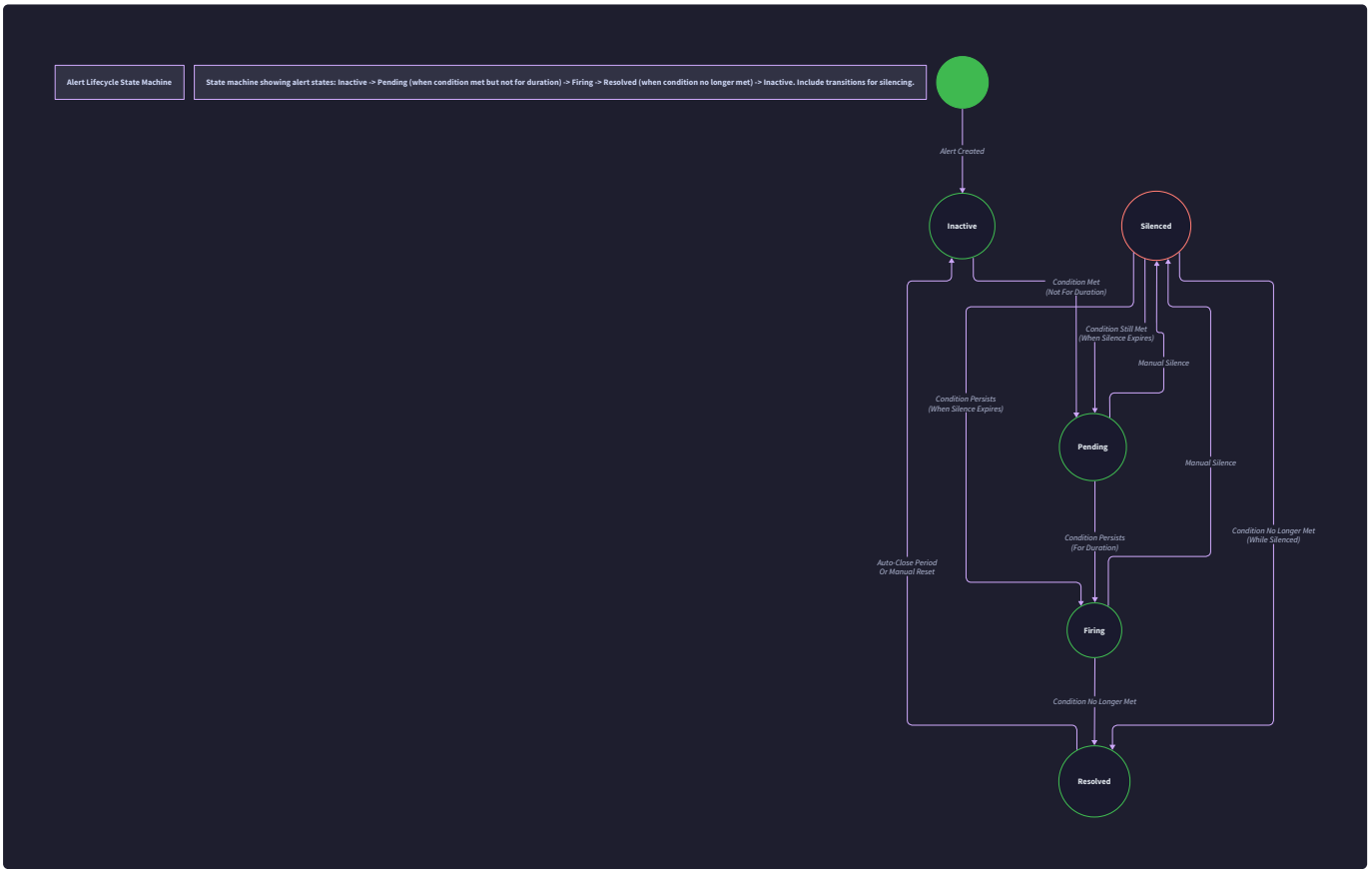
Field	Type	Description
`name`	`string`	Unique identifier for the rule (e.g., "high-error-rate")
`condition`	`string`	Query expression defining the alert condition (see Query Language section)
`for_duration`	`string`	Duration the condition must persist before alert fires (e.g., "5m", "1h")
`severity`	`string`	Severity level: "critical", "warning", "info"
`labels`	`map[string]string`	Key-value pairs attached to alerts for routing and grouping (e.g., `team: "backend"`, `service: "api"`)
`annotations`	`map[string]string`	Descriptive text for notifications (e.g., `summary: "High error rate detected"`, `description: "Error rate exceeds 5% for service {{.service_name}}"`)
`evaluation_interval`	`string`	How often to evaluate the rule (e.g., "30s", "1m")
`notification_channels`	`[]string`	List of channel names to notify (references NotificationChannel config)
`silenced`	`bool`	Whether the rule is currently silenced (manually disabled)
`created_at`	`int64`	Timestamp when rule was created
`updated_at`	`int64`	Timestamp when rule was last modified

The `condition` field uses a subset of the query language defined in the Query Engine section. Examples:

- Simple metric threshold: `metric:http_request_duration_seconds:p99 > 1.0`
- Log pattern alert: `log:message matches ".*OutOfMemoryError.*" AND severity >= "ERROR"`
- Cross-signal correlation: `(metric:error_rate > 5%) AND (trace:duration:p95 > 2s) AND (log:message matches ".*database timeout.*")`

Evaluation Process

The rule evaluation follows a deterministic state machine. Each rule maintains state for each distinct label combination (called an **alert instance**). For example, a rule with condition `metric:error_rate{service="*"} > 5%` would create separate instances for `service=api`, `service=payment`, etc., each with its own state.



The state machine transitions are:

Current State	Event	Next State	Actions Taken
`Inactive`	Condition evaluates to `true`	`Pending`	Start `for_duration` timer, record first true evaluation timestamp
`Pending`	Condition evaluates to `true` AND `for_duration` elapsed	`Firing`	Send initial notifications, record firing timestamp
`Pending`	Condition evaluates to `false`	`Inactive`	Reset timer, clear pending state
`Firing`	Condition evaluates to `true`	`Firing`	Continue firing (may send repeat notifications based on repeat interval)
`Firing`	Condition evaluates to `false`	`Resolved`	Send resolved notifications, start resolved retention period
`Resolved`	Resolved retention period elapsed	`Inactive`	Clean up alert instance state
Any state (except `Silenced`)	Silence rule via API	`Silenced`	Stop evaluations, cancel pending timers, mute notifications
`Silenced`	Unsilence rule via API	Previous state or `Inactive`	Resume evaluations from appropriate state

The evaluation algorithm for a single rule proceeds as follows:

1. **Parse Rule Configuration**: Load the `AlertRule` from storage and validate the `condition` query syntax.
2. **Determine Time Range**: Calculate the time window for evaluation based on current time and rule configuration. For most rules, this is "now minus evaluation_interval" to "now".
3. **Execute Condition Query**: Call `QueryEngine.ExecuteQuery()` with the parsed condition. The query must return a boolean result or a numeric/string value that can be compared against thresholds.
4. **Group Results by Labels**: The query may return multiple time series or log groups with different label combinations. Each unique label set becomes a potential alert instance.

5. **Update Instance States**: For each label combination:
    - Retrieve previous state from persistent store
    - Apply state machine transitions based on current evaluation result
    - Persist updated state
6. **Trigger Actions**: For instances transitioning to `Firing`, queue notifications. For instances transitioning to `Resolved`, queue resolution notifications.
7. **Record Metrics**: Emit metrics about evaluation duration, number of firing alerts, etc., for self-monitoring.

Rule Storage and Management

Alert rules need persistent storage for their configuration and runtime state. We use a simple file-based storage for the learning project:

Component	Purpose	Implementation
Rule Configuration Store	Stores `AlertRule` definitions	JSON file on disk, watched for changes
Alert Instance State Store	Stores current state per rule per label set	Embedded key-value store (e.g., BadgerDB) with TTL for resolved alerts
Silences Store	Stores active silences (time ranges when alerts are suppressed)	Separate JSON file or database table

The engine watches the rule configuration file for changes and dynamically reloads rules without restart. This enables quick iteration during development and operational updates in production.

Common Pitfalls

⚠️ Pitfall: Rule Evaluation Overload

- **Description**: Configuring too many rules with short evaluation intervals can overload the Query Engine and storage backends.
- **Why it's wrong**: Each rule evaluation executes one or more queries. With hundreds of rules evaluating every 30 seconds, the system may spend more resources evaluating alerts than serving user queries or ingesting data.
- **How to fix**: Implement query rate limiting per rule, increase default evaluation intervals, and aggregate similar rules. Add a `load_shedding` mode that skips non-critical rule evaluations under high load.

⚠️ Pitfall: Missing Alert Instance Garbage Collection

- **Description**: Not cleaning up `Resolved` alert instance state after retention period.
- **Why it's wrong**: Alert instances for transient issues (e.g., "high CPU for pod A") accumulate indefinitely after resolution. Pod A may no longer exist, but its alert state remains, wasting memory and disk.
- **How to fix**: Implement TTL (time-to-live) on resolved alert states. After an alert transitions to `Resolved`, schedule its state for deletion after a configurable period (e.g., 24 hours).

⚠️ Pitfall: Incorrect Time Window Selection

- **Description**: Using inappropriate time ranges for rule evaluation (e.g., evaluating "5-minute average" over only 1 minute of data).
- **Why it's wrong**: Queries may return incomplete or misleading results. A rule checking "error rate over 5 minutes" evaluated over 30 seconds might miss a spike that occurred 4 minutes ago.
- **How to fix**: Ensure evaluation time windows account for the intended aggregation period. For "5-minute average error rate," query at least 5 minutes of data, preferably slightly more to handle clock skew and data latency.

Anomaly Detection: Simple Statistical Baselines

While threshold-based alerting is essential, it requires operators to know what "normal" values are. Anomaly detection automatically learns baseline patterns and alerts when current behavior deviates significantly. For our learning platform, we implement **simple statistical baselines** using rolling windows of historical data.

Mental Model: The Seasoned Scout

Imagine a scout who has patrolled the same forest path for years. They know what's normal for each season: how many travelers pass in spring, what animals appear at dusk, which sounds indicate routine activity versus danger. They don't need specific rules like "alert if more than 50 travelers" because they know 20 is normal in winter but 100 is normal in harvest season. Their knowledge comes from observing patterns over time.

Our anomaly detection engine plays the role of this seasoned scout. It continuously observes metric streams, learns their normal ranges and patterns, and raises an alert when current behavior falls outside learned norms. Unlike the rule engine with explicit thresholds, it **infers** what's abnormal based on history.

Statistical Models

We implement two complementary statistical approaches suitable for different metric types:

Model	Best For	Algorithm	Parameters
Rolling Mean/StdDev	Metrics with relatively stable baselines and Gaussian distribution (e.g., request latency, success rate)	Calculate mean (μ) and standard deviation (σ) over rolling window. Alert when value > μ + kσ or < μ - kσ	Window size (e.g., 24h), sensitivity factor k (e.g., 3 for "3 sigma")
Rolling Percentiles	Metrics with non-Gaussian distributions or outliers (e.g., error counts, queue lengths)	Calculate p5, p50, p95 over rolling window. Alert when value > p95 + margin or < p5 - margin	Window size, percentile thresholds (e.g., p5/p95), margin multiplier
Simple Seasonality	Metrics with daily/weekly patterns (e.g., user traffic, business hours load)	Compare current value to same time previous day/week (e.g., "now vs 24h ago"). Alert when deviation exceeds threshold	Seasonality period (24h, 7d), deviation threshold

For the learning platform, we focus on the **Rolling Mean/StdDev** model as it provides a good balance of effectiveness and implementability. The algorithm for a single metric time series:

1. **Data Collection**: Store recent metric values in a circular buffer sized for the learning window (e.g., 24 hours of 1-minute samples = 1440 values).
2. **Baseline Calculation** (periodically, e.g., every 5 minutes):
   - Compute mean (μ) and standard deviation (σ) of values in the window
   - Store these baseline parameters with a timestamp
3. **Anomaly Detection** (on each new sample):
   - Retrieve the most recent baseline (μ, σ)
   - Calculate z-score: `z = (current_value - μ) / σ`
   - If `|z| > sensitivity_threshold` (e.g., 3.0), flag as anomaly
4. **Alert Triggering**:
   - If anomalies persist for configured duration, trigger alert
   - Include deviation magnitude in alert annotations

Integration with Alerting Engine

Anomaly detection integrates as a special rule type within the alerting engine. Instead of a static threshold condition, the rule specifies which metric to monitor and which statistical model to apply:

```yaml
# Example anomaly detection rule configuration
name: "unusual-request-latency"
type: "anomaly"
metric: "http_request_duration_seconds:p99"
model: "rolling_mean_stddev"
parameters:
  window: "24h"
  sensitivity: 3.0
  seasonality: "none"
for_duration: "5m"
severity: "warning"
```

The evaluation engine handles these specially:

- Regular rules: Execute query, compare result to threshold
- Anomaly rules: Retrieve metric values, compute baseline, compare current to baseline

Implementation Considerations

Storage Requirements: Anomaly detection needs to store historical metric values for baseline calculation. We can reuse the metric storage engine, but need efficient retrieval of rolling windows. Adding a dedicated cache for frequently evaluated anomaly rules improves performance.

Cold Start Problem: When a rule is first created or after a long silence, there's insufficient history to establish a baseline. We implement a **learning phase** where the rule collects data without alerting until the window is full.

Concept Drift: Systems evolve, and what was "normal" last month may not be normal today. The rolling window automatically adapts, but sudden permanent shifts (e.g., traffic doubling after feature launch) will cause false positives until the window rolls over. We can implement **change point detection** to accelerate adaptation

or allow manual baseline reset.

Common Pitfalls

⚠️ Pitfall: Sensitivity to Outliers in Baseline Calculation

- **Description**: A single extreme outlier in the training data skews mean and standard deviation, making the baseline unusable.
- **Why it's wrong**: If a metric spikes to 1000 (due to an incident) during the 24-hour window, the calculated mean might be 50 instead of 10, and standard deviation 200 instead of 5. Future values around 15 (actually high) won't trigger because they're within the inflated range.
- **How to fix**: Use robust statistics like median and median absolute deviation (MAD) instead of mean/stddev, or implement outlier filtering during baseline calculation.

⚠️ Pitfall: Ignoring Metric Cardinality

- **Description**: Creating anomaly rules for high-cardinality metrics without proper grouping.
- **Why it's wrong**: A rule like "detect anomaly in `request_duration{service="*", endpoint="*"}`" creates separate baselines for thousands of time series, consuming massive memory and CPU.
- **How to fix**: Apply anomaly detection selectively to aggregated metrics (e.g., per-service rather than per-endpoint) or implement dimensionality reduction techniques.

⚠️ Pitfall: Statistical Assumption Violations

- **Description**: Assuming metrics follow Gaussian distribution when they don't.
- **Why it's wrong**: Many operational metrics (error counts, queue lengths) follow Poisson or other distributions. Gaussian-based anomaly detection will have high false positive/negative rates.
- **How to fix**: Analyze metric distributions before choosing model, or use non-parametric methods (like percentiles) that make fewer assumptions.

Notification Routing and Deduplication

When an alert fires, it needs to reach the right people through the right channels with the right information. The notification system handles this routing while preventing **alert fatigue**—the phenomenon where teams become desensitized due to excessive, duplicate, or irrelevant notifications.

Mental Model: The Castle's Messaging System

In our medieval fortress analogy, different signals require different responses. A sighting of a lone rider might warrant a note to the gate captain. Smoke on the horizon might require ringing the alarm bell. An invading army demands lighting the beacon chain to neighboring castles. The messaging system must:

1. **Route to appropriate channels** (bell, beacon, rider)
2. **Format messages appropriately** (short codes for beacons, detailed reports for riders)
3. **Avoid duplicate messages** (don't send three riders with the same message)
4. **Escalate if unacknowledged** (if no response, send louder signal)

Our notification system provides the same capabilities for modern incident response.

Notification Channel Abstraction

We define a `NotificationChannel` interface that all notification backends implement:

Method	Parameters	Returns	Description
`Send()`	`alert *AlertInstance, config ChannelConfig`	`error`	Sends notification through this channel
`ValidateConfig()`	`config ChannelConfig`	`error`	Validates channel configuration
`Test()`	`config ChannelConfig`	`error`	Sends test message to verify channel works

Supported channel types for the learning platform:

Channel Type	Configuration Fields	Use Case
Webhook	`url`, `headers`, `timeout`	Generic integration (Slack, Teams, custom)
Email	`smtp_server`, `port`, `from`, `to`, `auth`	Critical alerts requiring paper trail
PagerDuty	`integration_key`, `severity_map`	On-call escalation
Slack	`webhook_url`, `channel`, `username`	Team collaboration & awareness

Each alert rule specifies which channels to use via `notification_channels` field containing channel names. Channels are configured separately in a configuration file:

```yaml
notification_channels:
  - name: "team-backend-slack"
    type: "slack"
    config:
      webhook_url: "${SLACK_WEBHOOK_URL}"
      channel: "#backend-alerts"
  - name: "oncall-pagerduty"
    type: "pagerduty"
    config:
      integration_key: "${PD_INTEGRATION_KEY}"
```

Alert Instance Data Structure

When an alert fires, we create an `AlertInstance` containing all context needed for notifications:

Field	Type	Description
`id`	`string`	Unique identifier for this firing instance
`rule_name`	`string`	Name of the rule that triggered
`labels`	`map[string]string`	Key-value pairs from rule evaluation
`annotations`	`map[string]string`	Descriptive text with templated values
`severity`	`string`	"critical", "warning", or "info"
`start_time`	`int64`	When alert entered `Firing` state
`end_time`	`int64`	When alert resolved (0 if still firing)
`current_value`	`string`	The value that triggered the alert (e.g., "error_rate=7.2%")
`generator_url`	`string`	URL to view the query that generated this alert
`fingerprint`	`string`	Hash of labels for deduplication

The `fingerprint` field is critical for deduplication. It's computed by hashing the sorted label key-value pairs (excluding the alert name). Two alert instances with identical fingerprints are considered the same logical alert, even if they fire at different times.

Deduplication Logic

Alert deduplication prevents multiple notifications for the same underlying issue. Consider a flapping service where error rate oscillates above/below threshold, causing the alert to repeatedly transition between `Pending` and `Firing`. Without deduplication, each transition would send a notification, causing alert fatigue.

Our deduplication system uses **alert grouping** and **notification batching**:

1. **Grouping**: Alerts with similar characteristics are grouped together. By default, alerts from the same rule are grouped, but we can configure grouping by labels (e.g., group all `service=api` alerts regardless of which specific endpoint triggered).
2. **Batching Period**: When the first alert in a group fires, we start a batching timer (e.g., 5 minutes). Additional alerts in the same group that fire within this period are batched together.
3. **Supression**: If the same alert (identical fingerprint) fires again while already in `Firing` state, we suppress the notification unless:
    - A significant time has passed (repeat interval, e.g., 4 hours for critical alerts)
    - The alert severity has changed
    - Important annotations have changed (e.g., error rate increased from 5% to 20%)

The deduplication state machine:

Scenario	Action
New alert with new fingerprint	Send immediate notification, record fingerprint as active
Same fingerprint within batching period	Add to batch, update internal state, don't send yet
Same fingerprint after batching period elapsed	Send batched notification summarizing all instances
Same fingerprint, already firing, repeat interval not elapsed	Suppress notification
Same fingerprint, already firing, repeat interval elapsed	Send repeat notification (e.g., "this alert is still firing")
Alert resolves (fingerprint transitions to `Resolved`)	Send resolution notification, remove from active set

Notification Template Engine

Alerts include templated annotations that get filled with actual values. The template engine uses Go's `text/template` with access to the `AlertInstance` fields:

```go
// Example annotation template

summary: "High error rate for {{.labels.service}}"

description: |

  Error rate is {{.current_value}} (threshold: 5%).

  Check dashboard: {{.generator_url}}

  Relevant logs: /logs?query=trace_id in (select trace_id from traces where service={{.labels.service}} and duration > 2s)
```

The engine also supports **cross-signal correlation hints** in notifications. When an alert fires, we can automatically include suggestions for related logs or traces, helping responders quickly investigate.

Escalation Policies

For critical alerts, we implement simple escalation policies. If an alert remains unacknowledged after a timeout, it escalates to additional channels or higher severity:

```yaml
escalation_policies:

  - name: "backend-critical"

    steps:

      - after: "5m"

        channels: ["team-backend-slack"]

      - after: "15m"

        channels: ["team-backend-slack", "oncall-pagerduty"]

      - after: "1h"

        channels: ["oncall-pagerduty", "manager-email"]
```

Escalation state is tracked per alert fingerprint. When an alert first fires, we record the start time. Periodic cleanup processes check for unacknowledged alerts that have exceeded their current step's timeout and escalate them.

Common Pitfalls

⚠️ Pitfall: Notification Storm During Outage

- **Description**: A widespread outage triggers hundreds of alerts simultaneously, overwhelming notification channels and responders.
- **Why it's wrong**: If 50 services depend on a failed database, you might get 50 "high error rate" alerts at once. Slack rate limits get hit, email queues back up, and the on-call engineer gets 50 pages in 60 seconds.
- **How to fix**: Implement alert grouping across services, throttle notifications per channel, and use **root cause analysis** to suppress dependent alerts (advanced feature).

⚠️ Pitfall: Missing Resolution Notifications

- **Description**: Sending "alert fired" notifications but not "alert resolved" notifications.
- **Why it's wrong**: Responders don't know when to stop investigating, leading to wasted effort and uncertainty about system state.
- **How to fix**: Always send resolution notifications through the same channels that received the initial alert. Include duration and final state in the resolution message.

⚠️ **Pitfall: Templating Injection Vulnerabilities**

- **Description**: Alert labels or annotations containing malicious template code get executed.
- **Why it's wrong**: If an attacker can inject `{{.Execute "rm -rf /"}}` into a metric label, it could execute arbitrary code during template rendering.
- **How to fix**: Sanitize label values, use restricted template functions, and avoid executing user-provided templates with full system access.

Implementation Guidance

Technology Recommendations Table

Component	Simple Option	Advanced Option
Rule Storage	JSON/YAML files on disk with file watching	SQLite/PostgreSQL with versioning
State Storage	In-memory map with periodic file snapshot	Embedded key-value store (BadgerDB) with TTL
Template Engine	Go's `text/template` package	Custom DSL with type checking
Notification Channels	Direct HTTP clients for webhook/email	Pluggable provider system with retry queues
Scheduler	Time.Ticker in goroutine	Distributed cron with leader election

Recommended File/Module Structure

```
project-root/
  cmd/
    alerting-engine/          # Separate binary for alerting (optional)
      main.go
  internal/
    alerting/
      engine/                 # Main alerting engine
        engine.go             # AlertingEngine type and main loop
        evaluator.go          # Rule evaluation logic
        state_manager.go      # Alert instance state management
      rules/
        rule.go               # AlertRule struct and methods
        parser.go             # Condition query parsing
        loader.go             # Rule configuration loading
      notifier/
        notifier.go           # Notification dispatcher
        channels/             # Channel implementations
          webhook.go
          slack.go
          email.go
          pagerduty.go
        deduplicator.go       # Deduplication logic
        templating.go         # Template rendering
      anomaly/
        detector.go           # Anomaly detection engine
        models/               # Statistical models
          rolling_mean.go
          percentiles.go
          seasonality.go
      storage/                # Alert state storage
        filestore.go          # File-based rule storage
        kvstore.go            # Key-value state storage
    query/                    # Query engine (from previous section)
      engine.go               # QueryEngine.ExecuteQuery()
```

Infrastructure Starter Code

Here's complete, working code for the rule configuration loader with file watching:

```go
// internal/alerting/storage/filestore.go

package storage


import (

        "encoding/json"

        "fmt"

        "io/ioutil"

        "os"

        "path/filepath"

        "sync"

        "time"


        "github.com/fsnotify/fsnotify"

        "go.uber.org/zap"

)


// RuleStore manages alert rule configuration stored in files

type RuleStore struct {

        mu          sync.RWMutex

        rules       map[string]*AlertRule

        rulesDir    string

        watcher     *fsnotify.Watcher

        logger      *zap.Logger

        updateChan  chan map[string]*AlertRule

        stopChan    chan struct{}

}


// NewRuleStore creates a new file-based rule store

func NewRuleStore(rulesDir string, logger *zap.Logger) (*RuleStore, error) {

        if err := os.MkdirAll(rulesDir, 0755); err != nil {

                return nil, fmt.Errorf("create rules directory: %w", err)

        }


        watcher, err := fsnotify.NewWatcher()

        if err != nil {

                return nil, fmt.Errorf("create file watcher: %w", err)

        }


        store := &RuleStore{

                rules:      make(map[string]*AlertRule),

                rulesDir:   rulesDir,

                watcher:    watcher,

                logger:     logger,
```

```
                updateChan: make(chan map[string]*AlertRule, 1),
                stopChan:   make(chan struct{}),
        }

        // Initial load
        if err := store.loadAllRules(); err != nil {
                return nil, fmt.Errorf("initial rule load: %w", err)
        }

        // Watch for changes
        if err := watcher.Add(rulesDir); err != nil {
                return nil, fmt.Errorf("watch directory: %w", err)
        }

        go store.watchFiles()
        return store, nil
}

// loadAllRules reads all .json and .yaml files in rulesDir
func (s *RuleStore) loadAllRules() error {
        s.mu.Lock()
        defer s.mu.Unlock()

        // Clear existing rules
        s.rules = make(map[string]*AlertRule)

        files, err := ioutil.ReadDir(s.rulesDir)
        if err != nil {
                return fmt.Errorf("read rules directory: %w", err)
        }

        for _, file := range files {
                if file.IsDir() {
                        continue
                }
                ext := filepath.Ext(file.Name())
                if ext != ".json" && ext != ".yaml" && ext != ".yml" {
                        continue
                }

                path := filepath.Join(s.rulesDir, file.Name())
                if err := s.loadRuleFile(path); err != nil {
                        s.logger.Error("Failed to load rule file",
                                zap.String("file", path),
```

```
                                zap.Error(err))
                }
        }

        s.logger.Info("Loaded rules", zap.Int("count", len(s.rules)))
        return nil
}

// loadRuleFile loads a single rule file
func (s *RuleStore) loadRuleFile(path string) error {
        data, err := ioutil.ReadFile(path)
        if err != nil {
                return fmt.Errorf("read file: %w", err)
        }

        var rules []AlertRule
        ext := filepath.Ext(path)

        switch ext {
        case ".json":
                if err := json.Unmarshal(data, &rules); err != nil {
                        return fmt.Errorf("parse JSON: %w", err)
                }
        case ".yaml", ".yml":
                // For simplicity, using JSON here - in practice use yaml.v3
                // yaml.Unmarshal would be similar
                if err := json.Unmarshal(data, &rules); err != nil {
                        return fmt.Errorf("parse YAML: %w", err)
                }
        default:
                return fmt.Errorf("unsupported file extension: %s", ext)
        }

        for _, rule := range rules {
                if rule.Name == "" {
                        return fmt.Errorf("rule missing name in file %s", path)
                }
                if err := rule.Validate(); err != nil {
                        return fmt.Errorf("invalid rule %s: %w", rule.Name, err)
                }
                s.rules[rule.Name] = &rule
        }
```

```
        return nil
}

// watchFiles monitors the rules directory for changes
func (s *RuleStore) watchFiles() {
        defer s.watcher.Close()

        for {
                select {
                case event, ok := <-s.watcher.Events:
                        if !ok {
                                return
                        }
                        if event.Op&fsnotify.Write == fsnotify.Write ||
                           event.Op&fsnotify.Create == fsnotify.Create {
                                s.logger.Debug("Rule file changed", zap.String("file", event.Name))
                                // Debounce rapid changes
                                time.Sleep(100 * time.Millisecond)
                                if err := s.loadAllRules(); err != nil {
                                        s.logger.Error("Failed to reload rules", zap.Error(err))
                                } else {
                                        // Notify engine of rule updates
                                        select {
                                        case s.updateChan <- s.GetAllRules():
                                                s.logger.Debug("Sent rule update")
                                        default:
                                                // Channel full, drop update (engine will reload on next eval)
                                        }
                                }
                        }
                case <-s.stopChan:
                        return
                }
        }
}

// GetAllRules returns a copy of all rules (thread-safe)
func (s *RuleStore) GetAllRules() map[string]*AlertRule {
        s.mu.RLock()
        defer s.mu.RUnlock()

        rulesCopy := make(map[string]*AlertRule)
```

```
        for k, v := range s.rules {
                // Deep copy
                ruleCopy := *v
                rulesCopy[k] = &ruleCopy
        }
        return rulesCopy
}

// GetRule returns a specific rule by name
func (s *RuleStore) GetRule(name string) (*AlertRule, bool) {
        s.mu.RLock()
        defer s.mu.RUnlock()

        rule, exists := s.rules[name]
        if !exists {
                return nil, false
        }
        // Return a copy
        ruleCopy := *rule
        return &ruleCopy, true
}

// UpdateChan returns a channel that receives updated rule sets
func (s *RuleStore) UpdateChan() <-chan map[string]*AlertRule {
        return s.updateChan
}

// Stop stops the file watcher
func (s *RuleStore) Stop() {
        close(s.stopChan)
}
```

Core Logic Skeleton Code

Here's the main alert evaluation skeleton with TODO comments mapping to the algorithm steps:

GO

```go
// internal/alerting/engine/evaluator.go

package engine


import (

	"context"

	"fmt"

	"time"


	"github.com/yourproject/internal/alerting/rules"

	"github.com/yourproject/internal/alerting/state"

	"github.com/yourproject/internal/query"

	"go.uber.org/zap"

)


// RuleEvaluator evaluates a single alert rule

type RuleEvaluator struct {

	rule        *rules.AlertRule

	stateStore  state.Store

	queryEngine query.Engine

	logger      *zap.Logger

	lastError   error

}


// Evaluate executes the rule condition and updates alert states

func (e *RuleEvaluator) Evaluate(ctx context.Context) error {

	// TODO 1: Parse the condition query from rule.Condition

	//         Convert to QueryRequest format expected by QueryEngine

	//         Hint: Use rules.ParseCondition(rule.Condition)


	// TODO 2: Determine time range for evaluation

	//         For most rules: start = now - evaluation_interval, end = now

	//         For anomaly rules: may need longer history for baseline calculation


	// TODO 3: Execute the query against QueryEngine

	//         Call queryEngine.ExecuteQuery(ctx, queryRequest)

	//         Handle errors (log and return, don't update states on query failure)


	// TODO 4: Process query results

	//         The query returns []any - need to convert based on signal type

	//         For metric queries: expect []MetricPoint

	//         For log queries: expect []LogRecord

	//         For trace queries: expect []Span


	// TODO 5: Group results by labels to create alert instances
```

```
        //         Each unique label combination becomes a potential alert instance
        //         Compute fingerprint = hash(sorted label key-value pairs)

        // TODO 6: For each alert instance (label combination):
        //         a. Retrieve previous state from stateStore
        //         b. Determine if condition is true/false for this instance
        //         c. Apply state machine transitions (Inactive->Pending->Firing->Resolved)
        //         d. Persist updated state to stateStore
        //         e. If state changed to Firing, trigger notifications
        //         f. If state changed to Resolved, trigger resolution notifications

        // TODO 7: Record evaluation metrics
        //         Count: total instances, firing instances, new firings, resolutions
        //         Duration: how long evaluation took
        //         Errors: any evaluation errors

        return nil
}

// evaluateCondition determines if the query result meets the alert condition
func (e *RuleEvaluator) evaluateCondition(result any, rule *rules.AlertRule) (bool, error) {
        // TODO 1: Check rule type (threshold vs anomaly)
        //         if rule.Type == "threshold" { ... } else if rule.Type == "anomaly" { ... }

        // TODO 2: For threshold rules:
        //         - Parse threshold from condition string (e.g., "> 5.0")
        //         - Compare result value to threshold
        //         - Return true if condition met

        // TODO 3: For anomaly rules:
        //         - Retrieve historical values for this metric/label combination
        //         - Calculate baseline (mean/stddev or percentiles)
        //         - Compute current deviation from baseline
        //         - Return true if deviation exceeds sensitivity threshold

        // TODO 4: Handle different result types
        //         MetricPoint: compare MetricPoint.Value to threshold
        //         LogRecord: check if log matches pattern/severity
        //         Span: check if duration exceeds threshold

        return false, nil
}

// updateAlertState implements the state machine transitions
func (e *RuleEvaluator) updateAlertState(
```

```
        prevState state.AlertState,
        conditionMet bool,
        duration time.Duration,
        now time.Time,
) (state.AlertState, []state.TransitionAction) {
        // TODO 1: Implement state transitions according to the state machine table
        //         Use a switch statement on prevState

        // TODO 2: Handle Inactive -> Pending transition
        //         When condition becomes true after being false
        //         Start for_duration timer (record firstTrueTime)

        // TODO 3: Handle Pending -> Firing transition
        //         When condition has been true for for_duration
        //         Return TransitionAction{Type: "fire", AlertInstance: ...}

        // TODO 4: Handle Firing -> Resolved transition
        //         When condition becomes false after being true
        //         Return TransitionAction{Type: "resolve", AlertInstance: ...}

        // TODO 5: Handle Resolved -> Inactive transition
        //         After resolved retention period elapses
        //         Clean up state

        // TODO 6: Handle all other transitions according to table

        return state.AlertState{}, nil
}
```

Language-Specific Hints

1. **Concurrency**: Use `sync.Map` for concurrent access to rule configurations, but regular `sync.RWMutex` for maps that are mostly read. Each rule evaluator should run in its own goroutine with proper cancellation via `context.Context`.
2. **Time Handling**: Use `time.Time` for timestamps internally, but convert to Unix nanoseconds (`int64`) for storage and APIs. Be careful with time zones—store everything in UTC.
3. **Hashing for Fingerprints**: Use `fnv64a` hash for alert fingerprints—it's fast and collisions are acceptable (unlike cryptographic hashes). Implement as:

```go
func fingerprint(labels map[string]string) string {
    h := fnv.New64a()
    keys := make([]string, 0, len(labels))
    for k := range labels { keys = append(keys, k) }
    sort.Strings(keys)
    for _, k := range keys {
        h.Write([]byte(k))
        h.Write([]byte(labels[k]))
    }
    return fmt.Sprintf("%x", h.Sum64())
}
```

4. **Template Rendering**: Use `text/template` with custom functions for formatting durations, percentages, etc. Escape all user-provided values to prevent injection:

```go
func renderTemplate(tmpl string, data any) (string, error) {
    t, err := template.New("alert").Funcs(template.FuncMap{
        "duration": func(ns int64) string { return time.Duration(ns).String() },
        "percent":  func(f float64) string { return fmt.Sprintf("%.2f%%", f*100) },
    }).Parse(tmpl)
    if err != nil { return "", err }

    var buf bytes.Buffer
    if err := t.Execute(&buf, data); err != nil {
        return "", err
    }
    return buf.String(), nil
}
```

Milestone Checkpoint

After implementing Milestone 5, you should be able to:

1. **Define and load alert rules**: Create a YAML file like `rules/high-error-rate.yaml`:

```yaml
- name: "high-error-rate"
  condition: "metric:http_requests_total{status=~\"5..\"} / metric:http_requests_total > 0.05"
  for_duration: "2m"
  severity: "critical"
  labels:
    team: "backend"
  annotations:
    summary: "High error rate for {{.labels.service}}"
  evaluation_interval: "30s"
  notification_channels: ["slack-backend"]
```

2. **Test rule evaluation**: Run the engine and verify it evaluates rules:

```bash
$ go run cmd/alerting-engine/main.go --rules-dir ./rules

INFO  Loading 3 alert rules from ./rules

INFO  Starting evaluation loop (30s interval)

INFO  Evaluating rule "high-error-rate"
```

3. **Trigger an alert**: Generate error traffic to exceed the 5% threshold for 2 minutes. Watch logs for:

```
INFO  Alert high-error-rate{service=api} transitioned to Pending
INFO  Alert high-error-rate{service=api} transitioned to Firing (error_rate=7.2%)
INFO  Sending notification via slack-backend
```

4. **Verify notifications**: Check your configured Slack channel or webhook receiver for the alert notification.

5. **Test deduplication**: Cause the condition to flap (true/false/true). Verify you don't get duplicate notifications within the repeat interval.

Debugging tips if alerts don't fire:

- Check that rule files are in the correct directory with proper permissions
- Verify the query condition syntax matches your metric names
- Ensure timestamps are aligned (evaluation uses UTC)
- Check state storage location—clearing it may help if states are stuck
- Enable debug logging to see evaluation details: `--log-level=debug`

Common Debugging Scenarios

Symptom	Likely Cause	How to Diagnose	Fix
No alerts firing	Rules not loading	Check engine logs for "Loading rules", verify file extensions (.yaml/.json)	Ensure rules directory exists and files are valid YAML/JSON
Alerts fire but no notifications	Channel misconfiguration	Check channel config, test with `Test()` method	Verify webhook URLs, API keys, network connectivity
Duplicate notifications	Missing fingerprint or deduplication	Check if identical alerts have different fingerprints	Ensure label sorting before hashing, check for label value variations
Alerts never resolve	Condition always true or state machine bug	Check evaluation logs, verify condition becomes false	Debug state transitions, check for infinite loops in evaluation
High memory usage	Too many alert instances accumulating	Check count of Resolved instances not cleaned up	Implement TTL cleanup for resolved alerts

> **Milestone(s):** Milestone 2 (Data Ingestion Pipeline), Milestone 4 (Unified Query Interface), Milestone 5 (Alerting & Anomaly Detection)

7. Interactions and Data Flow

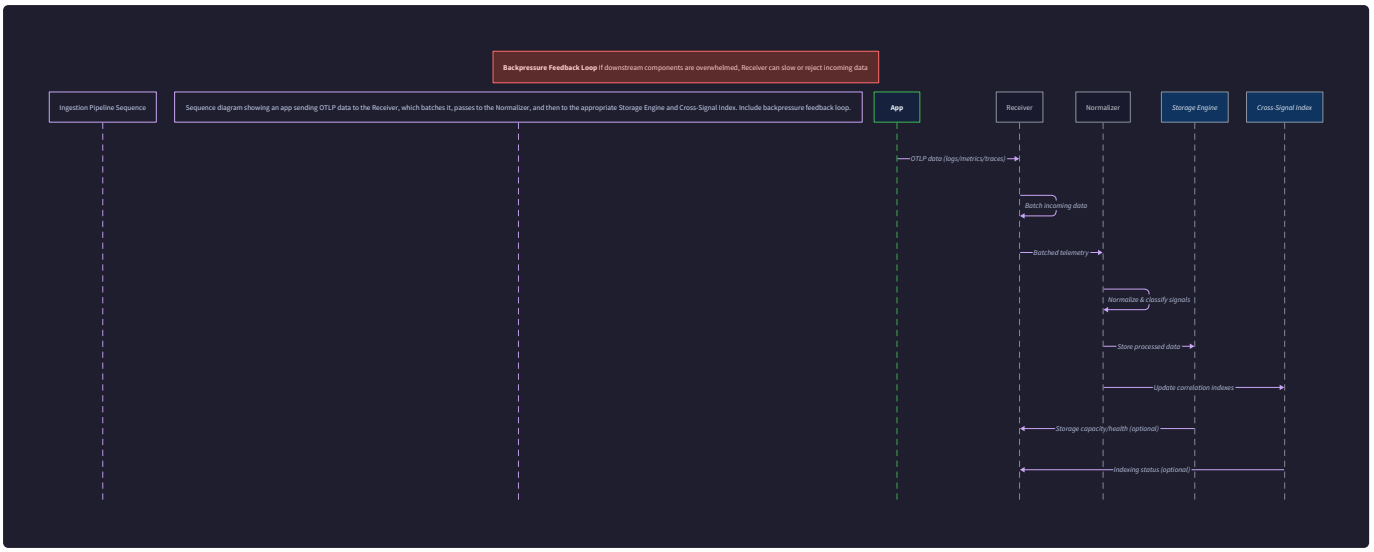
Understanding the static architecture is only half the battle. To truly grasp how the system operates, we must follow data and control signals as they move through the components in real-world scenarios. This section provides three concrete end-to-end journeys that trace the critical pathways through the observability platform. By walking through these flows step-by-step, we illuminate how the components interact, where data is transformed, and how the system responds to load and failures. These journeys correspond to the three primary user activities: sending telemetry, investigating an issue, and receiving proactive alerts.

Journey 1: Telemetry Ingestion Path

Mental Model: The Postal Sorting Facility Imagine a high-volume postal sorting facility. Mail trucks (applications) arrive at the loading dock, dumping sacks of mixed mail (telemetry batches). Workers quickly sort letters by type (logs, metrics, traces) and destination zip code (resource attributes). Each letter is then stamped with a standardized barcode (normalization), placed on the correct conveyor belt (pipeline channel), and routed to a specialized processing center for that mail type (storage engine). A master catalog (cross-signal index) is updated to record that a package (log) is associated with a particular registered mail tracking number (trace_id). The entire facility has traffic lights (backpressure) that turn red if any conveyor jams, telling trucks to wait before unloading more sacks.

This journey traces the path of a single span emitted by a microservice, along with its associated log line and a related metric, from the moment the application sends it until it's durably stored and indexed for future querying. This flow is the lifeline of the platform, and its efficiency and reliability are paramount.

Sequence of Steps:



1. **Application Emission:** A Go-based payment service processes an HTTP request. Using the OpenTelemetry SDK, it creates a `Span` with `trace_id="a1b2"`, `span_id="c3d4"`, and records the `start_time` and `end_time`. During the span's execution, the service logs a message (`LogRecord` with `body="Charging credit card"`) and increments a counter metric (`MetricPoint` for `payment_attempts_total`). The SDK batches this telemetry locally and, when ready, initiates an OTLP/gRPC export request to the platform's ingestion endpoint.
2. **OTLP/gRPC Reception:** The `IngestionPipeline` 's `otlpService` (listening on port `OTLP_GRPC_PORT` 4317) receives the `ExportTraceServiceRequest` , `ExportLogsServiceRequest` , and `ExportMetricsServiceRequest` . The gRPC server unmarshals the protocol buffer payloads. For each signal type (traces, logs, metrics), it calls the corresponding pipeline receive method (e.g., `Pipeline.ReceiveLog`). If the internal channel buffer (`INGEST_CHANNEL_BUFFER`) for that signal is full, the method immediately returns an `ErrPipelineFull` error, applying **backpressure** to the client SDK, which should implement retry-with-jitter logic.
3. **Pipeline Buffering & Batching:** Each received item (e.g., a `LogRecord`) is placed into a `batchBuffer` specific to its resource attributes and signal type. The `batchBuffer.add` method is called. The buffer holds items in memory. It checks two flush conditions: a size limit (`MAX_BATCH_SIZE`) and a time interval (`FLUSH_INTERVAL_MS`). When either condition is met, the `batchBuffer.take` method is invoked, which creates a `Batch` containing the items, their common `ResourceAttributes` , and the `signal_type` . This `Batch` is sent to the `batchChan` .
4. **Normalization:** A normalization stage goroutine reads `Batch` objects from `batchChan` . For each item in the batch, it calls the appropriate `Normalizer` method (e.g., `Normalizer.NormalizeSpan`). This stage performs critical data hygiene: it validates required fields (e.g., ensures `trace_id` is 32 hex characters), coerces timestamps to nanoseconds, enforces `OpenTelemetry semantic conventions` on attribute names (e.g., converts `http.status` to `http.status_code`), and drops records that are fundamentally invalid (logging an error). The normalized items are now in our platform's canonical internal format.
5. **Fanout to Storage & Indexing:** The normalized batch is fanned out to two destinations concurrently:
   - **Primary Storage Engine:** Based on `Batch.signal_type` , the batch is routed to the appropriate storage engine's write method. For a trace batch, `TraceStorageEngine.WriteBatch` is called. The storage engine begins its own internal persistence process (e.g., appending to a `WriteAheadLog` , updating in-memory structures).
   - **Cross-Signal Correlation Index:** For each item in the batch that contains correlation IDs, the `CrossSignalCorrelationIndex.IndexLogRecord` (or analogous method) is called. For our `LogRecord` with `trace_id="a1b2"` , the index creates or updates a `CorrelationRecord` for that `trace_id` , appending a new `LogRef` pointing to the log's storage location (e.g., segment ID and offset). For a `MetricPoint` with an `Exemplar` containing `trace_id="a1b2"` , it adds an `ExemplarRef` .
6. **Storage Acknowledgement:** Once the storage engine has durably persisted the data (e.g., after a successful `fsync` of the WAL), and the index update is complete, the ingestion path for that batch is considered successful. The gRPC handler can now send a successful `ExportResponse` back to the client application (though this may be sent earlier for lower latency, depending on durability configuration).

Data Transformation Summary Table:

Stage	Input Data Structure	Key Action	Output Data Structure
Application	OTel SDK Span, LogRecord, Metric	Instrumentation & local batching	OTLP Protocol Buffers
OTLP Receiver	`Export*Request` (protobuf)	Unmarshal, validate headers	Internal `Span` , `LogRecord` , `MetricPoint` structs
Batch Buffer	Individual telemetry items	Group by resource & signal, wait for size/time	`Batch` struct
Normalizer	`Batch` with raw items	Validate, standardize timestamps/attributes	`Batch` with canonical items
Storage Fanout	Canonical `Batch`	Persist to specialized store, update correlation map	Written to disk, index record updated

Common Pitfalls in the Ingestion Path:

⚠️ **Pitfall: Head-of-Line Blocking Across Signals**

- **Description:** Implementing a single, shared `batchChan` for all signal types (logs, metrics, traces). If trace processing becomes slow (e.g., due to a slow disk), logs and metrics are stuck behind it in the queue, causing unnecessary latency for those signals.
- **Why It's Wrong:** Violates the principle of separation of concerns and prevents independent scaling and performance tuning per signal type. It can cause alerts based on metrics to fire late because metric data is stuck behind a queue of large trace batches.
- **Fix:** Use separate channels and pipeline stages per signal type after the initial receiver, as shown in the `Pipeline` struct fields (`logChan` , `metricChan` , `traceChan`). This allows each signal to flow at its own pace.

⚠️ **Pitfall: Silent Data Loss on Indexing Failure**

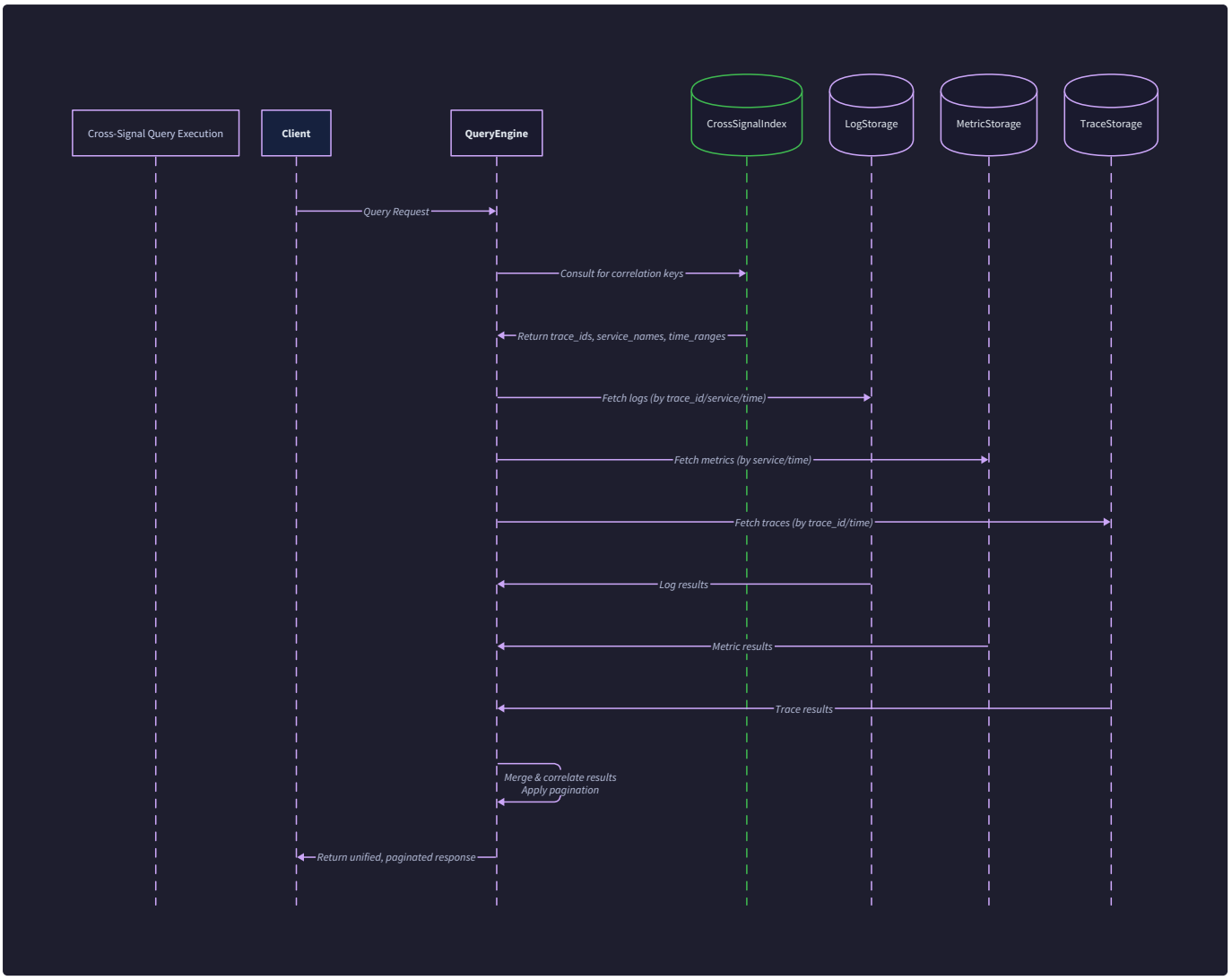
- **Description:** The primary storage engine write succeeds, but the update to the `CrossSignalCorrelationIndex` fails (e.g., due to a momentary memory issue). The pipeline reports success to the client, but the log can no longer be found by its `trace_id` in a cross-signal query.
- **Why It's Wrong:** Breaks a core platform promise: trace-log correlation. Debugging becomes impossible because the critical link is missing, and the user is unaware.
- **Fix:** Implement a two-phase commit *pattern* within the storage fanout stage. Write to both the primary store and the index within a single atomic operation if possible, or implement a compensating action: if the index update fails, the entire batch operation should be rolled back (or placed in a dead-letter queue for repair), and an error should be returned to the client.

Journey 2: Cross-Signal Query Execution

Mental Model: The Detective's Investigation Board A detective (the user) has a lead: "Service X is slow." They approach the platform's query interface. The detective doesn't know where the evidence is stored. The **Query Engine** acts as their assistant, who knows the archive layout. The assistant first checks the master case file index (`CrossSignalCorrelationIndex`) to find all "cases" (`trace_id` s) related to "Service X" and marked "slow". Then, the assistant sends parallel requests to three evidence rooms: the **Log Storage** (for witness statements logged during those slow traces), the **Trace Storage** (for the detailed timeline of events), and the **Metric Storage** (for system vitals like CPU during that period). The assistant gathers the returned evidence, correlates it by the common case number (`trace_id`), assembles a comprehensive report, and presents a paginated summary to the detective.

This journey executes a user query such as: *"Show me ERROR logs from the last 15 minutes for traces originating from service 'checkout' where the trace duration was greater than 2 seconds."* It demonstrates the power of the unified query interface and the importance of the cross-signal index.

Sequence of Steps:



1. **API Request:** A user submits a `CrossSignalQueryRequest` via the Query Engine's HTTP/gRPC API. The request specifies a root query (e.g., find slow traces) and a correlation specification (e.g., fetch logs linked to those traces). For our example, the `RootQuery` is a `TraceQuery` with `service_name="checkout"`, `minDuration=2s`, and a `startTime` / `endTime`. The `CorrelateWith` includes a `CorrelationSpec` with `Signal="logs"` and `LinkField="trace_id"`.
2. **Query Parsing & Planning:** The `QueryEngine.ExecuteCrossSignalQuery` method receives the request. It first parses the `RootQuery` and any correlation specs into an abstract syntax tree (AST). Then, the `Planner.BuildPlan` is invoked. The planner analyzes the AST and generates a tree of `PlanNode`s. For our query, the plan might be:
   - `ScanTracesNode`: Finds traces matching the service and duration filter.
   - `FilterNode`: Further filters traces by time range.
   - `CorrelationJoinNode`: Uses the `CrossSignalCorrelationIndex`. For each resulting `trace_id`, it looks up the associated `LogRef`s from the index.
   - `ScanLogsNode`: For each list of `LogRef`s, it fetches the actual `LogRecord` objects from the `LogStorageEngine`. It also applies an additional filter for `severity="ERROR"`.
   - `MergeSortNode`: Merges the results (traces with their associated logs) and sorts by trace start time.
   - `LimitNode`: Applies pagination based on the `Limit` and `Cursor` from the request.
3. **Distributed Execution:** The query engine begins executing the plan, starting at the leaf nodes. It leverages **filter pushdown** wherever possible. The `ScanTracesNode` translates its filters into a `TraceStorageEngine`-specific query and calls `TraceStorageEngine.Query`. The engine searches its `TraceMetadata` index and retrieves relevant `SpanStorageBlock`s. Concurrently, the `CorrelationJoinNode` queries the `CrossSignalCorrelationIndex` with the list of `trace_id`s obtained from the trace scan to get the `LogRef`s.
4. **Data Retrieval & Correlation:** The `ScanLogsNode` receives the list of `LogRef`s. It may group them by `storage_hint` (e.g., which log segment they reside in) to perform efficient batched reads from the `LogStorageEngine`. It fetches the log records and applies the severity filter in memory. At this stage, logs are "joined" to their parent traces by virtue of sharing the same `trace_id`.
5. **Result Assembly & Pagination:** The `MergeSortNode` receives streams of correlated trace-and-log data. It merges them into a single, time-ordered stream. The `LimitNode` takes the first N results (e.g., 50). It also calculates a `NextCursor`—an opaque token that encodes the position in the result stream (e.g., the

timestamp and ID of the last item returned)—so the client can request the next page.

6. **API Response:** The assembled results are packaged into a `QueryResponse` (or `CrossSignalQueryResponse`), containing the `Results` slice, the `NextCursor`, and a `Total` count if feasible. This is serialized to JSON/protobuf and returned to the user's UI, which can display a unified view: slow traces on a timeline, with expandable sections showing the ERROR logs that occurred within each trace.

Query Plan Optimization Table:

Optimization	Description	Benefit in Our Example
Filter Pushdown	Pushing `service_name="checkout"` and `minDuration=2s` down to the `TraceStorageEngine`.	Reduces the number of traces fetched from disk dramatically before any cross-signal join.
Index-Based Correlation	Using the `CrossSignalCorrelationIndex` to find `LogRef`s for a set of `trace_id`s.	Avoids a full scan of the log storage; directly locates only relevant logs.
Batch Log Fetch	Grouping multiple `LogRef`s targeting the same storage segment into a single read operation.	Reduces I/O operations and disk seeks, improving throughput.
Early Limit Application	Applying the `Limit` as early as possible in the plan (often at the root).	Reduces the amount of intermediate data processed and transferred between plan nodes.

Common Pitfalls in Query Execution:

⚠️ **Pitfall: Cartesian Product Explosion in Correlation**

- **Description:** Implementing a naive correlation join that, for each trace, scans *all* logs in the time range and compares `trace_id` fields. This creates a computational complexity of O(traces × logs), which becomes intractable at scale.
- **Why It's Wrong:** Makes cross-signal queries unusably slow. A query that should complete in milliseconds might take minutes or hours, rendering the platform ineffective for interactive debugging.
- **Fix: Always** use the `CrossSignalCorrelationIndex` for the initial join. The index provides an O(1) or O(log n) lookup from `trace_id` to the list of related log locations, eliminating the need for a full scan.

⚠️ **Pitfall: Ignoring Result Streaming for Large Datasets**

- **Description:** The query engine waits for *all* results from all storage engines to be collected in memory before starting to serialize the response to the client.
- **Why It's Wrong:** Can cause the query engine to run out of memory on large result sets (e.g., fetching all logs for a high-traffic service over an hour). It also increases query latency (time to first result).
- **Fix:** Implement streaming at the API layer (e.g., gRPC server-side streaming, HTTP chunked encoding) and within the query plan. The `MergeSortNode` should stream sorted results to the response writer as soon as they are available, and the `LimitNode` can stop the upstream nodes once enough results have been produced.

Journey 3: Alert Evaluation and Firing

Mental Model: The Automated Weather Station Consider a network of automated weather stations (alert rules) monitoring various conditions (metrics, log patterns). Each station has sensors (rule conditions) and a defined threshold (e.g., wind speed > 75 mph). The `RuleEvaluator` is the station's computer, which samples sensor data every `evaluation_interval`. When a threshold is breached, the station doesn't immediately sound the alarm; it waits for the condition to persist for a `for_duration` to avoid false alarms from gusts. Once confirmed, it activates its alert siren (changes state to `Firing`) and sends a formatted alert message via a designated communication channel (Slack, PagerDuty). A central monitor watches for duplicate alerts from nearby stations (deduplication) to prevent alarm fatigue.

This journey begins with a metric value crossing a threshold and ends with a human-readable notification appearing in an on-call engineer's Slack channel. It highlights the periodic pull-based evaluation model, the state machine that prevents flapping, and the routing and deduplication logic.

Sequence of Steps:

1. **Scheduled Evaluation Trigger:** A ticker fires every `evaluation_interval` (e.g., 15 seconds) within the `AlertingEngine`. The engine calls `AlertingEngine.EvaluateRules()`. This method fetches all active, non-silenced `AlertRule` objects from the `RuleStore.GetAllRules()`.
2. **Parallel Rule Evaluation:** For each rule, the engine (or a worker pool) spawns a `RuleEvaluator.Evaluate` goroutine. The evaluator parses the rule's `condition` string (e.g., `rate(api_errors_total{service="checkout"}[5m]) > 0.1`). It translates this into a `QueryRequest` for the `QueryEngine`—in this case, a `MetricQuery` for the `api_errors_total` metric with specific labels, aggregated with a `rate` over a 5-minute window.
3. **Query Execution & Condition Check:** The `RuleEvaluator` calls `QueryEngine.ExecuteQuery` with the constructed request. The query engine processes it as in Journey 2, but typically this is a simpler, single-signal query against the `MetricStorageEngine`. The query returns a numeric result (e.g., 0.15). The evaluator then calls `RuleEvaluator.evaluateCondition`, which compares the result (0.15) to the threshold (0.1). The condition is `true`.
4. **State Machine Transition:** The evaluator loads the previous `AlertState` for this specific rule and its label set (identified by a `fingerprint` hash). It calls `RuleEvaluator.updateAlertState` with the previous state, the condition result (`true`), the rule's `for_duration`, and the current time. The state machine logic determines the next state:

- If previous state was `Inactive` → transition to `Pending`. Start a timer for `for_duration`.
- If previous state was `Pending` and the `for_duration` has elapsed → transition to `Firing`. **This triggers alert notification**.
- If previous state was `Firing` and condition is still `true` → remain `Firing`. Respect the `repeat_interval` to decide if another notification should be sent.
- If condition is `false` → transition from `Firing` to `Resolved` (send a resolution notification), then to `Inactive`.

5. **Notification Generation & Deduplication:** Upon entering the `Firing` state, an `AlertInstance` object is created with all relevant labels, annotations, the current value, and a `fingerprint`. Before sending, the engine checks for active alerts with the same `fingerprint`. If one already exists and is within its `repeat_interval`, the new notification is **deduplicated** (suppressed). If not, it proceeds to routing.
6. **Channel Routing & Delivery:** The rule's `notification_channels` list (e.g., `["slack-payments-team", "pagerduty-primary"]`) is consulted. For each channel, the engine retrieves the corresponding `NotificationChannel` configuration (API keys, webhook URLs). It formats the `AlertInstance` into a channel-specific template (e.g., a Slack block with color, title, and links to the platform's UI) and calls `NotificationChannel.Send`. The channel implementation handles the external HTTP call, with retries for transient failures.
7. **Alert Lifecycle Management:** The `AlertInstance` is stored in an active alerts store. The UI can query this store to show currently firing alerts. When the condition later resolves, a final "resolved" notification is sent, and the instance is moved to a historical log.

Alert State Machine Table:

Current State	Event (Condition)	Next State	Actions
`Inactive`	`true`	`Pending`	Record transition time (start of `for_duration`).
`Pending`	`true` & `for_duration` elapsed	`Firing`	Create `AlertInstance`. Send notifications via all channels.
`Pending`	`false`	`Inactive`	Clear pending timer.
`Firing`	`true`	`Firing`	Update `AlertInstance` current value. Send repeat notification only if `repeat_interval` has passed.
`Firing`	`false`	`Resolved`	Send "resolved" notification. Set a short cleanup timer.
`Resolved`	(after cleanup delay)	`Inactive`	Remove `AlertInstance` from active store.
Any	Rule is `silenced`	Suppressed	No state change, no actions. Alerts are muted.

Common Pitfalls in Alerting:

⚠️ Pitfall: Synchronous, Sequential Rule Evaluation

- **Description:** The `AlertingEngine` iterates through rules in a single loop, evaluating each one synchronously and waiting for the query engine's response before moving to the next rule.
- **Why It's Wrong:** If one rule's query is slow (e.g., a complex multi-signal correlation), it delays the evaluation of all subsequent rules, potentially causing alerts to fire late or evaluation cycles to pile up.
- **Fix:** Evaluate rules **concurrently** using a worker pool with a bounded size. Each rule evaluation is a separate task. This isolates latency and ensures the evaluation loop completes in a predictable time.

⚠️ Pitfall: Storing State In-Memory Only

- **Description:** The alert state (which alerts are `Pending`/`Firing`) is stored only in the memory of the `AlertingEngine` process.
- **Why It's Wrong:** If the engine process crashes or restarts, it loses all memory of pending timers and firing alerts. Alerts in a `Pending` state may never fire (because the timer is lost), and firing alerts will stop sending repeat notifications or resolution messages.
- **Fix:** Persist the alert state machine to a durable store (e.g., a simple embedded database like SQLite, or the platform's own storage). On startup, the engine reloads the state and recovers timers. This ensures alert continuity across process restarts.

Implementation Guidance

A. Technology Recommendations Table:

Component	Simple Option	Advanced Option
Query API Transport	HTTP/JSON with `net/http` (simpler to debug with `curl`)	gRPC with streaming support (better performance for large result sets)
Query Plan Execution	Synchronous execution in a single goroutine (easier to reason about)	Parallel execution with a pipelined, reactive stream model (better throughput for complex plans)
Alert State Storage	In-memory map protected by `sync.RWMutex` (simple)	Embedded DB (Badger, SQLite) for crash recovery (production-ready)
Notification Channel	Direct HTTP calls with basic retry logic	Message queue (in-memory or Redis) for decoupling and guaranteed delivery

B. Recommended File/Module Structure: The interactions flow touches multiple components. The main orchestration logic for Journeys 1 and 2 resides in the service entry point and the query engine, while Journey 3 is in the alerting engine.

```
project-root/
  cmd/
    observability-server/
      main.go                    # Wires up all components, starts pipelines & schedulers
  internal/
    ingestion/
      pipeline.go                # Journey 1: Orchestrates stages (receive, batch, normalize, fanout)
    query/
      engine.go                  # Journey 2: Main query engine, calls planner and executor
      planner.go                 # Builds execution plan trees
      executor.go                # Walks the plan tree, calls storage engines
    alerting/
      engine.go                  # Journey 3: Main scheduler, manages rule evaluators
      state/                     # Alert state machine and persistent storage
        store.go
        machine.go
      notifier/
        router.go                # Handles channel routing and deduplication
        channels/                # Implementations for Slack, PagerDuty, etc.
          slack.go
          pagerduty.go
```

C. Infrastructure Starter Code (Query Result Stream Merger): A common need in Journey 2 is merging sorted streams from different storage engines. Here is a generic heap-based merger:

GO

```go
// internal/query/stream_merger.go

package query


import (

        "container/heap"

        "context"

)


// Stream represents a sorted stream of items with a common key (e.g., timestamp).

type Stream interface {

        Next(ctx context.Context) (interface{}, error) // Returns nil, io.EOF when done

        Key(interface{}) int64                         // Extract sort key (timestamp) from an item

}


// streamHeap implements heap.Interface for merging multiple streams.

type streamHeap struct {

        items []*streamItem

}


type streamItem struct {

        value interface{}

        key   int64

        index int

        next  func(ctx context.Context) (interface{}, error)

}


func (h streamHeap) Len() int            { return len(h.items) }

func (h streamHeap) Less(i, j int) bool  { return h.items[i].key < h.items[j].key }

func (h streamHeap) Swap(i, j int)       { h.items[i], h.items[j] = h.items[j], h.items[i]; h.items[i].index = i; h.items[j].index = j }


func (h *streamHeap) Push(x interface{}) {

        n := len(h.items)

        item := x.(*streamItem)

        item.index = n

        h.items = append(h.items, item)

}


func (h *streamHeap) Pop() interface{} {

        old := *h

        n := len(old.items)

        item := old.items[n-1]

        item.index = -1 // for safety

        h.items = old.items[0 : n-1]

        return item
```

```
}

// MergeStreams merges multiple sorted streams into a single sorted stream.
// It uses a min-heap to always pull the item with the smallest key.
func MergeStreams(ctx context.Context, streams []Stream) (chan interface{}, chan error) {
        out := make(chan interface{}, 10)
        errc := make(chan error, 1)

        go func() {
                defer close(out)
                defer close(errc)

                h := &streamHeap{}
                heap.Init(h)

                // Initial population of the heap with the first element from each stream.
                for i, stream := range streams {
                        val, err := stream.Next(ctx)
                        if err != nil {
                                // Stream might be empty; skip.
                                continue
                        }
                        heap.Push(h, &streamItem{
                                value: val,
                                key:   stream.Key(val),
                                index: i,
                                next:  stream.Next,
                        })
                }

                for h.Len() > 0 {
                        // Pop the smallest item.
                        smallest := heap.Pop(h).(*streamItem)
                        select {
                        case out <- smallest.value:
                        case <-ctx.Done():
                                errc <- ctx.Err()
                                return
                        }

                        // Get the next item from the same stream.
                        nextVal, err := smallest.next(ctx)
                        if err == nil {
```

```
                                // Re-insert with the new value.
                                heap.Push(h, &streamItem{
                                        value: nextVal,
                                        key:   streams[smallest.index].Key(nextVal),
                                        index: smallest.index,
                                        next:  smallest.next,
                                })
                        }
                        // If the stream is exhausted, do nothing (it's removed from the heap).
                }
        }()

        return out, errc
}
```

D. Core Logic Skeleton Code:

1. **Pipeline Fanout Stage (Journey 1, Step 5):**

```go
// internal/ingestion/pipeline.go

func (p *Pipeline) fanoutStage(ctx context.Context) {
    for {
        select {
        case batch := <-p.batchChan:
            // TODO 1: Determine signal type from batch.signalType
            // TODO 2: For each item in batch.items, call the appropriate Normalizer method
            // TODO 3: After normalization, start two goroutines for concurrent writing:
            //   - Goroutine A: Call the appropriate storage engine's WriteBatch method (e.g., LogStorageEngine.WriteBatch)
            //   - Goroutine B: For each normalized item, if it has correlation IDs (trace_id),
            //     call CrossSignalCorrelationIndex.IndexLogRecord (or IndexMetricExemplar)
            // TODO 4: Wait for both goroutines to complete (use sync.WaitGroup).
            // TODO 5: If either fails, log an error and consider placing the batch in a dead-letter queue for reprocessing.
        case <-ctx.Done():
            return
        }
    }
}
```

2. **Query Plan Execution (Journey 2, Step 2-4):**

```go
// internal/query/executor.go

func (e *Executor) executeNode(ctx context.Context, node PlanNode) (RowStream, error) {

    switch n := node.(type) {

    case *ScanTracesNode:

        // TODO 1: Construct a TraceQuery from the node's stored filters (serviceName, minDuration, etc.)

        // TODO 2: Call TraceStorageEngine.Query with the constructed query.

        // TODO 3: Return a stream that yields the resulting Span objects one by one.

    case *CorrelationJoinNode:

        // TODO 1: Execute the left child node (e.g., ScanTracesNode) to get a stream of traces.

        // TODO 2: For each trace, extract its trace_id.

        // TODO 3: Batch multiple trace_ids and query CrossSignalCorrelationIndex.BatchGet to get LogRefs.

        // TODO 4: Return a stream that yields pairs of (trace, []LogRef).

    case *ScanLogsNode:

        // TODO 1: Receive a stream of (trace, []LogRef) pairs from the parent node.

        // TODO 2: For each LogRef, fetch the full LogRecord from LogStorageEngine using the storage_hint.

        // TODO 3: Apply any additional filters from this node (e.g., severity == "ERROR") to the fetched logs.

        // TODO 4: Return a stream of enriched results (trace with its filtered logs).

    case *MergeSortNode:

        // TODO 1: Collect streams from all child nodes.

        // TODO 2: Use the MergeStreams helper to merge them into a single time-ordered stream.

        // TODO 3: Return the merged stream.

    case *LimitNode:

        // TODO 1: Execute the child node to get a stream.

        // TODO 2: Keep a counter. Yield items from the child stream until counter == n.Limit.

        // TODO 3: When limit is reached, stop requesting items from child (may require context cancellation).

        // TODO 4: Calculate the NextCursor based on the last yielded item.

    default:

        return nil, fmt.Errorf("unsupported plan node type: %T", node)

    }

}
```

3. **Alert State Transition (Journey 3, Step 4):**

```go
// internal/alerting/state/machine.go

func (m *StateMachine) updateAlertState(prevState AlertState, conditionMet bool, forDuration time.Duration, now time.Time) (AlertState, []TransitionAction) {

    var actions []TransitionAction

    var nextState AlertState


    // TODO 1: Implement the state transition logic as defined in the table above.

    // TODO 2: For each transition that triggers an action (e.g., Inactive->Pending, Pending->Firing, Firing->Resolved),

    // append the appropriate TransitionAction to the actions slice.

    // Example: TransitionAction{Type: SendNotification, Channel: "default"}

    // TODO 3: Handle the special case of 'silenced' rules: if silenced, nextState should be suppressed and no actions generated.

    // TODO 4: Return the calculated nextState and the slice of actions.


    return nextState, actions

}
```

E. Language-Specific Hints (Go):

- Use `context.Context` throughout the data flow to manage cancellation and timeouts, especially in query execution and notification sending.
- For the ingestion pipeline channels, use buffered channels (`INGEST_CHANNEL_BUFFER` , `BATCH_CHANNEL_BUFFER`) to absorb small bursts of traffic, but rely on the backpressure mechanism (`ErrPipelineFull`) for sustained overload.
- In the alerting engine, use `time.Ticker` for the evaluation loop, but ensure you stop it gracefully on shutdown using a `context.CancelFunc` .
- When implementing the query plan executor, consider using the `pipeline pattern` with goroutines and channels for each plan node to enable streaming and parallelism. The `MergeStreams` helper provided above is a key building block.

F. Milestone Checkpoint: *After implementing Milestone 2 (Ingestion) and 4 (Query), you should be able to run the following integrated test:*

1. Start the server: `go run cmd/observability-server/main.go` .
2. Use a script or `curl` to send a sample OTLP trace with an embedded log line to `http://localhost:4317/v1/traces` (HTTP/JSON) or via gRPC.
3. Wait a few seconds for the flush interval.
4. Query for that log by its `trace_id` : `curl 'http://localhost:8080/query/logs?trace_id=a1b2&limit=10'` .
5. **Expected:** The API should return a JSON response containing the log record you sent, with its body and associated resource attributes.
6. **Troubleshooting:** If no logs appear, check the ingestion pipeline logs for normalization errors. If the query returns empty but ingestion succeeded, verify that the `CrossSignalCorrelationIndex` is being updated correctly and that the log storage engine's query method is consulting it.

> **Milestone(s):** Milestone 2 (Data Ingestion Pipeline), Milestone 3 (Multi-Signal Storage), Milestone 4 (Unified Query Interface), Milestone 5 (Alerting & Anomaly Detection)

8. Error Handling and Edge Cases

In any distributed system handling real-time data, failures are not a matter of *if* but *when*. A robust observability platform must be **more reliable than the systems it observes**. This section defines the failure modes the platform will encounter, how to detect them, and the recovery mechanisms that ensure data integrity and availability even under adverse conditions. The design follows the principle of **graceful degradation**: when the system cannot operate perfectly, it should fail in predictable, safe ways that preserve core functionality and provide clear signals for operator intervention.

Mental Model: The Resilient Refinery

Imagine our telemetry refinery operates in a harsh environment—earthquakes, power surges, and supply chain disruptions are common. Our engineering team has designed multiple layers of protection:

1. **Shock Absorbers (Buffers & Backpressure):** Flexible pipelines and storage tanks that expand and contract to handle surges without rupturing.
2. **Pressure Relief Valves (Circuit Breakers & Dead Letter Queues):** Automatic valves that divert dangerous overpressure (malformed data, downstream failures) to safe containment areas, preventing catastrophic failure.
3. **Redundant Systems & Safety Inspections (Retry Queues & Health Checks):** Backup pumps and regular integrity checks that automatically engage when primary systems falter.

4. **Graceful Shutdown Procedures (Cleanup & Read-Only Fallbacks):** Well-rehearsed protocols to safely shut down parts of the refinery for maintenance while keeping critical monitoring functions online.

This mental model emphasizes that error handling is not an afterthought but a core architectural feature built into every component.

Failure Modes in Ingestion

The ingestion pipeline is the system's front line, exposed to the unpredictable volume and quality of client data. Its primary failure modes stem from **resource exhaustion**, **invalid inputs**, and **downstream unavailability**.

Failure Mode	Detection Strategy	Immediate System Response	Long-Term Impact if Unhandled
Backpressure Triggered (Upstream sends data faster than pipeline can process)	`IngestionPipeline.ReceiveOTLP` blocks on a full channel; `batchBuffer` reaches `MAX_BATCH_SIZE` too frequently.	Return `UNAVAILABLE` (gRPC code 14) or `429 Too Many Requests` (HTTP) to client. Log warning with rate metrics.	Client data loss (client may drop telemetry), increased client-side memory usage, frustrated users.
Malformed or Invalid Data (e.g., invalid `trace_id` length, non-numeric metric value, missing required `resource`).	Validation in `Normalizer.NormalizeLog/NormalizeSpan/NormalizeMetric` returns an error.	Record is rejected and placed into a **Dead Letter Queue (DLQ)**. Increment `ingestion_errors` counter with `cause="validation"`.	Loss of specific invalid records, DLQ growth consuming disk space, potential signal from a misconfigured service going unseen.
Transient Downstream Failure (e.g., `LogStorageEngine.WriteBatch` fails due to a full disk or network timeout).	Error returned from storage engine method call. The `Pipeline.fanoutStage` encounters the error.	Retry the batch operation with exponential backoff (up to a max limit). If retries exhausted, move batch to a **persistent retry queue**.	Increased ingestion latency, growth of retry queue, potential data delay or loss if queue is not durable.
Memory Exhaustion (e.g., gigantic batch or memory leak in normalization).	Go runtime panics on allocation; monitored via process memory metrics.	Process crashes. Orchestrator (e.g., Kubernetes) restarts it. Critical data in-flight may be lost.	Service interruption, possible data loss for buffered, unpersisted data.
Head-of-Line Blocking (One slow storage engine blocks processing for all signals).	Monitoring of per-signal channel depths (`logChan`, `metricChan`, `traceChan`) shows one consistently full while others are empty.	**Partial backpressure:** Only the affected signal's channel fills, eventually applying backpressure to its portion of the receiver.	Cross-contamination of latency; a trace storage slowdown could cause metric ingestion to stall.

> **Key Insight:** The ingestion pipeline must be a **good citizen** to both its upstream clients and downstream storage. It protects storage by applying backpressure, and it protects clients by providing clear, immediate feedback about acceptance or rejection of their data.

ADR: Handling Malformed Data with a Dead Letter Queue

> **Decision: Reject and Quarantine**
>
> - **Context:** Invalid data (e.g., malformed JSON, violated semantic conventions, extreme cardinality) can corrupt indexes, break queries, and exhaust resources. Simply dropping it destroys forensic evidence needed to debug the source of the problem.
> - **Options Considered:**
>   1. **Drop with Log:** Silently discard invalid records and log an error.
>   2. **Best-Effort Repair:** Attempt to fix common issues (e.g., truncate long fields) and ingest the repaired record.
>   3. **Reject and Quarantine:** Reject the request for the invalid record(s) and write the raw, unprocessed bytes to a separate, durable Dead Letter Queue (DLQ).
> - **Decision:** Option 3, Reject and Quarantine.
> - **Rationale:** Dropping data (Option 1) is operationally dangerous—we lose the signal that a service is misconfigured. Repair (Option 2) is brittle and can mask serious issues, leading to "silent data corruption." Quarantining preserves the raw evidence for offline analysis, allows for potential replay after fixes, and provides a clear, auditable path for data loss. It aligns with the observability platform's own need to be debuggable.
> - **Consequences:** Requires additional storage for the DLQ and operational processes to monitor and purge it. Adds complexity to the ingestion path (serialization to DLQ). The client still receives an error, encouraging them to fix their instrumentation.

Option	Pros	Cons	Chosen?
Drop with Log	Simple, no extra storage.	Data loss forever, no ability to recover or analyze corrupt data.	No
Best-Effort Repair	Maximizes data ingestion, user-friendly.	Unreliable, hides bugs, may produce misleading data.	No
Reject and Quarantine	Preserves forensic data, enables replay, clear failure signal.	Extra storage/processing, operational overhead.	**Yes**

Common Pitfalls in Ingestion Error Handling

⚠️ Pitfall: Ignoring Backpressure at the Client

- **Description:** The ingestion server correctly returns `UNAVAILABLE`, but the client (e.g., OpenTelemetry SDK) is not configured to retry or buffer, so data is dropped on the floor.
- **Why it's Wrong:** The system's reliability becomes dependent on the client's configuration, which is outside our control. We've moved the point of failure without solving it.
- **Fix:** Provide clear documentation and client configuration examples for recommended retry/backoff behavior. Consider a "best practices" SDK configuration that can be distributed to users of the platform.

⚠️ Pitfall: Retry Loops Without Backoff or Limits

- **Description:** On a transient storage failure, the pipeline immediately and endlessly retries the same batch in a tight loop.
- **Why it's Wrong:** This can turn a short-lived downstream problem (e.g., a 30-second storage blip) into a cascading failure, consuming all CPU/threads on the ingestion node and preventing it from recovering or handling new requests.
- **Fix:** Implement **exponential backoff with jitter** (e.g., wait 100ms, 200ms, 400ms...) and a **maximum retry count** (e.g., 5). After max retries, move the batch to a persistent queue for out-of-band recovery.

⚠️ Pitfall: Dead Letter Queue as a Black Hole

- **Description:** Data is written to the DLQ but never inspected, cleaned, or purged, leading to unbounded disk consumption.
- **Why it's Wrong:** Turns a safety mechanism into a reliability time bomb.
- **Fix:** Instrument the DLQ size as a critical metric. Build simple tooling to inspect and replay entries from the DLQ. Implement an automated age-based cleanup policy (e.g., delete records older than 7 days).

Failure Modes in Storage

Storage engines are the custodians of the platform's most valuable asset: historical telemetry data. Their failure modes are often **permanent** (data corruption, hardware failure) or **performance-related** (slow queries, full disks).

Failure Mode	Detection Strategy	Immediate System Response	Long-Term Impact if Unhandled
Disk Full	`os` package returns "no space left on device" on write; monitor filesystem usage via metrics.	`WriteAheadLog.Append` and storage engine writes will fail.	Ingestion halts completely (backpressure cascades to clients). New data is lost.
Data Corruption (Bit rot, faulty hardware, software bug).	Checksum verification fails on segment/block read; `encoding/binary.Read` returns unexpected errors.	The specific corrupted segment/block is marked `BAD` and skipped. Log a high-severity error with the segment ID.	Loss of the data in that segment. If it's an index, query results may be incomplete.
High Query Load (Many expensive concurrent queries).	Monitoring of query queue depth and latency (`query_duration_seconds`). Storage engine methods take longer than SLA to return.	Query engine starts rejecting new queries with `RESOURCE_EXHAUSTED` (or queues them with a timeout).	Poor user experience, timeouts, inability to debug issues during outages (when queries are most needed).
Memory Exhaustion in Indexes (e.g., extremely high-cardinality metric labels filling up the in-memory `Series` map).	Go runtime panics; monitor `process_resident_memory_bytes` and cardinality gauges.	Process crash and restart. May enter a crash loop if the data pattern persists.	Service unreliability, inability to ingest new data for the affected signal type.
Clock Skew / Bad Timestamps (Client sends data with timestamps far in the future or past).	`Normalizer` detects timestamps outside an acceptable window (e.g., not > system time - 1 day and < system time + 1 hour).	Record is rejected to DLQ (treated as malformed data).	If ingested, it corrupts time-based indexes and makes queries misleading.

ADR: Read-Only Fallback for Storage Failures

> **Decision: Fail Open for Reads, Fail Closed for Writes**
>
> - **Context:** A storage engine encounters a severe, unrecoverable error (e.g., disk corruption, failed recovery from WAL) that prevents it from accepting new writes. However, historical data may still be readable and valuable for debugging the ongoing incident.
> - **Options Considered:**
>   1. **Fail Closed (Crash):** The storage engine process panics, taking the entire node offline.
>   2. **Fail Closed (Block):** The engine enters a stuck state, refusing all operations (reads and writes) and causing upstream backpressure to stall.
>   3. **Fail Open for Reads:** The engine enters a **degraded, read-only mode**. It rejects all writes with an error but continues to serve queries against the last known good state.
> - **Decision:** Option 3, Fail Open for Reads.
> - **Rationale:** During a major incident, the ability to query historical logs, metrics, and traces is **more critical** than recording new data. Crashing (Option 1) destroys all utility. Blocking (Option 2) prevents both old and new data access. A read-only fallback provides maximum utility during a partial failure—operators can still investigate even as new data is temporarily lost. This aligns with the platform's primary goal: to enable debugging of system failures.
> - **Consequences:** Requires explicit state management in storage engines (`isReadOnly` flag). Writes (ingestion) will fail, causing data loss for the duration of the outage. Operators must monitor for "read-only mode" alerts and intervene to restore full functionality.

Option	Pros	Cons	Chosen?
Fail Closed (Crash)	Simple, clear failure state.	Total loss of service, no ability to debug.	No
Fail Closed (Block)	Prevents corruption of bad state.	Halts entire pipeline, loses both historical and new data access.	No
Fail Open for Reads	Preserves critical debugging capability.	Explicit design complexity, temporary write data loss.	**Yes**

Common Pitfalls in Storage Error Handling

⚠️ Pitfall: Not Fsyncing the Write-Ahead Log (WAL)

- **Description:** The `WriteAheadLog.Append` method writes to a file but does not call `file.Sync()` before returning success.
- **Why it's Wrong:** If the process crashes after the write returns but before the OS flushes to disk, the data is lost permanently, even though the ingestion pipeline thought it was durable. This violates the "durability" guarantee of the WAL.
- **Fix: Always `Sync` after writing a WAL entry.** This is a performance trade-off but non-negotiable for correctness. Use batch syncing (sync every N records) to mitigate performance impact if needed, acknowledging the small window of data loss it introduces.

⚠️ Pitfall: Letting Queries Run Amok

- **Description:** A user submits a query with no time range over all historical data, or a regex that triggers a full table scan. The query runs for minutes, consuming CPU, memory, and I/O, starving other queries and ingestion.
- **Why it's Wrong:** A single user can accidentally or maliciously degrade the service for everyone. This is a denial-of-service vector.
- **Fix:** Implement **query timeouts** (e.g., 30 seconds default) and **resource limits** (max rows scanned, memory per query). The `QueryEngine.ExecuteQuery` should enforce these, cancel the underlying storage operations, and return a clear error.

⚠️ **Pitfall: Ignoring Cardinality Explosion**

- **Description:** A misbehaving service emits a unique `user_id` or `request_id` as a metric label or log attribute for every event, creating millions of unique time series or index entries.
- **Why it's Wrong:** Can exhaust memory (for in-memory indexes) and disk, grind query performance to a halt, and increase costs exponentially.
- **Fix: Enforce cardinality limits at ingestion.** The `Normalizer` should have configurable limits per resource (e.g., max 500 unique label combinations per metric name). Excess cardinality should trigger rejection to DLQ and high-severity alerts. Educate users on appropriate vs. inappropriate use of high-cardinality dimensions.

Recovery and Mitigation Strategies

Recovery strategies are the **proactive and reactive plans** that return the system to a healthy state after a failure. They range from automatic retries to manual operator intervention.

Strategy	Mechanism	When Activated	Outcome
Circuit Breaker	Wraps calls to a downstream service (e.g., `LogStorageEngine`). After N consecutive failures, the circuit "opens" and fails fast for a cooldown period, then allows a test request ("half-open").	Transient or persistent downstream failures (e.g., storage engine overloaded, network partition).	Prevents cascading failures and resource exhaustion in the caller. Gives the downstream service time to recover.
Dead Letter Queue (DLQ)	A persistent, durable queue (e.g., disk-backed) where invalid or unprocessable records are written with metadata (error, timestamp, source).	`Normalizer` validation fails; a record cannot be indexed after max retries.	Preserves data for forensic analysis and potential replay. Isolates bad data from healthy processing pipeline.
Retry Queue with Exponential Backoff	A durable queue for batches that failed due to transient errors. A separate goroutine retries them with increasing delays.	Storage engine returns a retryable error (e.g., `io.ErrShortWrite`, `context.DeadlineExceeded`).	Maximizes data ingestion eventually without overwhelming the recovering downstream service.
Graceful Shutdown & Drain	On receiving SIGTERM, the `Pipeline.Stop()` method closes input channels, processes remaining in-flight batches, flushes buffers, and persists state before exiting.	Process is being terminated for deployment, scaling, or maintenance.	Minimizes data loss during restarts. Ensures WAL and in-memory buffers are not lost.
Read-Only Fallback Mode	Storage engine sets an internal flag after detecting an unrecoverable write error (e.g., disk full). All subsequent writes return an error; reads continue normally.	Permanent write impairment detected (disk full, corruption in write path).	Maintains availability of historical data for debugging during a partial outage.
Health Checks & Readiness Probes	HTTP endpoint (`/ready`) that checks critical dependencies: WAL writable, storage engines responsive, memory usage below threshold.	Orchestrator (Kubernetes) performs periodic checks. If fails, traffic is diverted from the instance.	Prevents routing traffic to a malfunctioning instance, enabling automatic failover in a cluster.
Data Repair & Compaction	Background processes that read corrupted segments, rebuild indexes from the WAL, or merge small files into larger, optimized segments.	Scheduled (nightly) or triggered by corruption detection.	Recovers storage space, improves query performance, and fixes minor data corruption.

Step-by-Step Recovery: Disk Full Scenario

This procedure outlines the automated and manual steps when the primary storage disk fills up.

1. **Detection:** The `LogStorageEngine.WriteBatch` receives an "no space left on device" error from the OS. The error is propagated to the `Pipeline.fanoutStage`.
2. **Immediate Automatic Response:** a. The pipeline's retry logic kicks in, but retries will fail immediately with the same error. b. After retries exhausted, the batch is moved to the persistent retry queue (which should be on a *different* filesystem). c. The affected storage engine (`LogStorageEngine`) transitions to **read-only fallback mode**. It logs a critical alert: `STORAGE_ENGINE_READ_ONLY` and increments a gauge `storage_read_only{engine="logs"}=1`. d. The ingestion pipeline continues to accept data for other signals (metrics, traces) if their storage is on separate disks.
3. **Alerting:** The `storage_read_only` gauge triggers a **Paging Alert** to the on-call engineer. The alert includes details: which engine, disk path, and free space percentage.
4. **Manual Operator Intervention:** a. Engineer receives alert and investigates. They may increase disk capacity, delete old data per retention policy, or clean up temporary files. b. Once space is freed, the operator sends a management command (e.g., HTTP POST to `/admin/logs/readonly?disable=true`) to take the storage engine out of read-only mode.

5. **Automatic Recovery:** a. The storage engine resets its `isReadOnly` flag and logs a recovery message. b. A separate "replay" process begins consuming batches from the persistent retry queue and writing them to storage. c. Normal ingestion resumes. The system catches up on queued data.

ADR: Use of Circuit Breakers for Downstream Calls

> **Decision: Implement Circuit Breakers for All Cross-Component Calls**
>
> - **Context:** The platform is a composition of microservices (ingestion → storage, query engine → storage). A slow or failing downstream component can cause threads/goroutines to pile up in the caller, leading to resource exhaustion and cascading failure.
> - **Options Considered:**
>   1. **Retries Only:** Rely solely on retry with backoff logic.
>   2. **Static Timeouts:** Use context timeouts on every call.
>   3. **Circuit Breaker Pattern:** Dynamically stop calling a downstream service after a threshold of failures, failing fast.
> - **Decision:** Option 3, Circuit Breaker Pattern, **in addition to** timeouts and retries (Options 2 & 1).
> - **Rationale:** Retries alone can worsen a downstream outage by hammering it with requests. Timeouts prevent a single call from hanging but don't protect against a high volume of calls that all timeout. A circuit breaker is a stateful pattern that **gives the downstream service room to breathe** by failing fast once a failure pattern is detected. This is critical for maintaining stability in the face of partial failures. It's a standard pattern for building resilient distributed systems.
> - **Consequences:** Adds complexity (state management, tuning of thresholds). Requires monitoring of circuit breaker state (open/closed/half-open) to understand system health. Must be combined with good fallback logic (e.g., for a open circuit to storage, the query engine might return a partial error or cached results if available).

Option	Pros	Cons	Chosen?
Retries Only	Simple, may eventually succeed.	Can exacerbate outages, waste resources.	No (used in combination)
Static Timeouts	Prevents indefinite hangs.	Does not reduce load under failure; many fast-failing calls can still overwhelm.	No (used in combination)
Circuit Breaker	Prevents cascading failures, gives downstream recovery time.	More complex, requires tuning.	**Yes**

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Circuit Breaker	Manual state management with counters and timers (`sync/atomic` , `time.Timer`).	Use the `github.com/sony/gobreaker` library, which provides a standardized, configurable implementation.
Dead Letter Queue	Append raw data + metadata to a dedicated file on local disk. Use JSON lines format for easy inspection.	Use a separate, robust queue system like Apache Kafka or Redis Streams for scalability and replay capabilities.
Retry Queue	In-memory priority queue (`container/heap`) with persistence to a file on shutdown.	Use a dedicated job queue database (e.g., SQLite with a jobs table) for durability and complex scheduling.
Health Checks	Simple HTTP handler checking a few conditions (`/ready` , `/health`).	Use structured health checks library like `github.com/hellofresh/health-go` with detailed component status.

B. Recommended File/Module Structure

Integrate error handling structures into the existing component directories.

```
project-root/
  internal/
    ingestion/
      pipeline.go           # Main Pipeline struct with circuit breaker fields
      dead_letter_queue.go  # DLQ implementation (file-based)
      retry_queue.go        # Retry queue logic
      normalizer.go         # Validation and rejection to DLQ
    storage/
      logs/
        engine.go           # LogStorageEngine with read-only mode flag
        wal.go              # WriteAheadLog with proper fsync
      metrics/
        engine.go           # MetricStorageEngine with cardinality limits
      traces/
        engine.go
      common.go             # Common storage errors and recovery utilities
    query/
      engine.go             # QueryEngine with timeouts and limits
    alerting/
      engine.go             # AlertingEngine with its own error handling
    health/
      health.go             # Centralized health check registration
```

C. Infrastructure Starter Code

File-Based Dead Letter Queue (DLQ): Provides a simple, durable store for bad records.

```go
// internal/ingestion/dead_letter_queue.go

package ingestion

import (
    "encoding/json"
    "fmt"
    "os"
    "path/filepath"
    "sync"
    "time"
)

// DLQRecord represents an item stored in the Dead Letter Queue.
type DLQRecord struct {
    ID          string      `json:"id"`           // Unique identifier (e.g., UUID)
    ReceivedAt  time.Time   `json:"received_at"`  // When the record arrived
    SignalType  string      `json:"signal_type"`  // "log", "metric", "trace"
    RawData     []byte      `json:"raw_data"`     // Original payload (JSON, protobuf, etc.)
    Error       string      `json:"error"`        // Why it was rejected
    Resource    string      `json:"resource,omitempty"` // Service name for triage
}

// FileDLQ implements a disk-backed Dead Letter Queue.
type FileDLQ struct {
    mu        sync.Mutex
    file      *os.File
    encoder   *json.Encoder
    directory string
    maxSize   int64 // Maximum file size before rotation
}

// NewFileDLQ creates or opens a DLQ in the given directory.
func NewFileDLQ(directory string) (*FileDLQ, error) {
    if err := os.MkdirAll(directory, 0755); err != nil {
        return nil, fmt.Errorf("creating DLQ directory: %w", err)
    }
    filePath := filepath.Join(directory, "dlq.json")
    file, err := os.OpenFile(filePath, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        return nil, fmt.Errorf("opening DLQ file: %w", err)
    }
    return &FileDLQ{
```

```
        file:      file,
        encoder:   json.NewEncoder(file),
        directory: directory,
        maxSize:   100 * 1024 * 1024, // 100MB default
    }, nil
}

// Write adds a record to the DLQ. It's thread-safe.
func (q *FileDLQ) Write(record DLQRecord) error {
    q.mu.Lock()
    defer q.mu.Unlock()

    // Simple rotation check
    if stat, err := q.file.Stat(); err == nil && stat.Size() > q.maxSize {
        q.file.Close()
        // Rotate file (e.g., rename with timestamp)
        newName := filepath.Join(q.directory, fmt.Sprintf("dlq-%d.json", time.Now().Unix()))
        os.Rename(filepath.Join(q.directory, "dlq.json"), newName)
        // Reopen fresh file
        file, err := os.OpenFile(filepath.Join(q.directory, "dlq.json"), os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
        if err != nil {
            return fmt.Errorf("rotating DLQ file: %w", err)
        }
        q.file = file
        q.encoder = json.NewEncoder(file)
    }

    record.ID = fmt.Sprintf("%d", time.Now().UnixNano())
    record.ReceivedAt = time.Now().UTC()
    return q.encoder.Encode(record)
}

// Close releases the DLQ file handle.
func (q *FileDLQ) Close() error {
    q.mu.Lock()
    defer q.mu.Unlock()
    return q.file.Close()
}
```

D. Core Logic Skeleton Code

Circuit Breaker in the Pipeline's fanoutStage:

```go
// internal/ingestion/pipeline.go (excerpt)

import (
    "github.com/sony/gobreaker"
)

type Pipeline struct {
    // ... existing fields ...
    logStorageBreaker   *gobreaker.CircuitBreaker
    metricStorageBreaker *gobreaker.CircuitBreaker
    traceStorageBreaker *gobreaker.CircuitBreaker
    dlq                 *FileDLQ
}

func NewPipeline(...) *Pipeline {
    // ...
    p.logStorageBreaker = gobreaker.NewCircuitBreaker(gobreaker.Settings{
        Name:        "log_storage",
        MaxRequests: 5,          // Half-open state: allow 5 test requests
        Interval:    60 * time.Second, // Time to reset failure count
        Timeout:     30 * time.Second, // How long circuit stays open
        ReadyToTrip: func(counts gobreaker.Counts) bool {
            // Trip after 10 consecutive failures
            return counts.ConsecutiveFailures > 10
        },
    })
    // ... create other breakers similarly
}

func (p *Pipeline) fanoutStage(ctx context.Context) {
    for {
        select {
        case batch := <-p.batchChan:
            var err error
            switch batch.signal_type {
            case "log":
                // Wrap storage call with circuit breaker
                _, err = p.logStorageBreaker.Execute(func() (interface{}, error) {
                    return nil, p.logStorage.WriteBatch(batch.items)
                })
            case "metric":
                _, err = p.metricStorageBreaker.Execute(func() (interface{}, error) {
```

```
                    return nil, p.metricStorage.WriteBatch(batch.items)
                })
            case "trace":
                _, err = p.traceStorageBreaker.Execute(func() (interface{}, error) {
                    return nil, p.traceStorage.WriteBatch(batch.items)
                })
            }
            if err != nil {
                // TODO 1: Check if error is retryable (e.g., network timeout, temporary overload).
                // TODO 2: If retryable, add batch to the persistent retry queue with a backoff timestamp.
                // TODO 3: If non-retryable (e.g., validation error from storage), write individual failed items to DLQ.
                // TODO 4: Log the error with appropriate severity and increment metrics.
            }
        case <-ctx.Done():
            return
        }
    }
}
```

Read-Only Mode in Storage Engine:

```go
// internal/storage/logs/engine.go (excerpt)

type LogStorageEngine struct {
    mu           sync.RWMutex
    isReadOnly   bool
    readOnlyReason string
    // ... other fields (segments, WAL, etc.)
}

func (e *LogStorageEngine) WriteBatch(records []*telemetry.LogRecord) error {
    e.mu.RLock()
    if e.isReadOnly {
        e.mu.RUnlock()
        return fmt.Errorf("storage engine is in read-only mode: %s", e.readOnlyReason)
    }
    e.mu.RUnlock()

    // TODO 1: Attempt to write to WAL. If fails with disk full error, call e.enterReadOnlyMode().
    // TODO 2: Process the batch (add to in-memory segment, update indexes).
    // TODO 3: If any step fails with a permanent write error, call e.enterReadOnlyMode().
    // TODO 4: Return appropriate error.
}

func (e *LogStorageEngine) enterReadOnlyMode(reason string) {
    e.mu.Lock()
    defer e.mu.Unlock()
    if !e.isReadOnly {
        e.isReadOnly = true
        e.readOnlyReason = reason
        // Log a CRITICAL alert and update metrics
        metrics.GaugeSet("storage_read_only", 1, map[string]string{"engine": "logs"})
    }
}

func (e *LogStorageEngine) ExitReadOnlyMode() error {
    e.mu.Lock()
    defer e.mu.Unlock()
    // TODO: Perform a health check (e.g., can write a test file to disk).
    // If healthy, set isReadOnly = false and update metrics.
}
```

E. Language-Specific Hints (Go)

- **For filesystem errors:** Use `os.IsNotExist(err)`, `os.IsPermission(err)`, and check for `syscall.ENOSPC` (disk full) via `errors.Is(err, syscall.ENOSPC)`.

- **For safe concurrent state changes:** Use `sync.RWMutex` for protecting flags like `isReadOnly`. Use `sync/atomic` for simple counters (e.g., failure counts for circuit breakers).
- **For context timeouts:** Always propagate `context.Context` to downstream calls and respect cancellation. Use `context.WithTimeout` to set deadlines in the query engine.
- **For exponential backoff:** Use `time.Sleep` with a calculated duration. The `math/rand` package can provide jitter. Consider `github.com/cenkalti/backoff/v4` for a robust implementation.

F. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Ingestion latency spikes periodically.	Storage engine is slow, causing circuit breaker to flip open/closed.	Check metrics for `circuit_breaker_state` (0=closed, 1=open). Look at storage engine resource usage (CPU, disk I/O).	Investigate storage performance (query load, compaction). Tune breaker thresholds or improve storage performance.
Dead Letter Queue file growing rapidly.	A specific service is sending malformed data consistently.	Use a script to sample the DLQ file (`tail -n 10 dlq.json`), look at the `Error` and `Resource` fields.	Identify the misconfigured service owner and share the error. Fix the instrumentation or normalization rules.
Queries timeout under load, but ingestion is fine.	Query engine lacks resource limits or is hitting a slow storage path.	Check query engine logs for slow query patterns. Monitor per-query resource usage (if instrumented).	Implement query timeouts and row limits in `QueryEngine.ExecuteQuery`. Add indexes for common query patterns.
Process crashes with OOM after high-cardinality metric ingestion.	Metric labels with unbounded values (like `request_id`) are being accepted.	Check logs before crash for cardinality warnings. Look at `metric_series_count` gauge.	Enforce cardinality limits in `Normalizer.NormalizeMetric` . Add a validation rule that rejects labels with values matching high-cardinality patterns.

9. Testing Strategy

> **Milestone(s):** Milestone 1 (Unified Data Model), Milestone 2 (Data Ingestion Pipeline), Milestone 3 (Multi-Signal Storage), Milestone 4 (Unified Query Interface), Milestone 5 (Alerting & Anomaly Detection)

Building a reliable observability platform requires a comprehensive testing strategy that validates each component in isolation and, critically, their interactions. Since the system ingests, stores, and correlates vast amounts of heterogeneous data, tests must ensure correctness, performance, and resilience at every layer. This section outlines a testing pyramid approach tailored to the distinct characteristics of each platform component and provides concrete implementation checkpoints for each milestone to verify progress.

The Testing Pyramid: Unit, Integration, and End-to-End

Think of the testing strategy as a **multi-layered defense system**, similar to a medieval castle. The outer walls (end-to-end tests) catch large-scale breaches in user workflows. The inner keep (integration tests) defends against failures in component interactions. The castle's armory (unit tests) ensures every individual weapon (function) is sharp and reliable. Each layer has a specific purpose and cost profile.

Unit Tests: Validating Core Logic and Data Structures

Unit tests target the smallest testable units—individual functions, methods, and data structures—in complete isolation. For this platform, unit tests are crucial for validating the data model's correctness, compression algorithms, indexing logic, and state machines.

Component	Primary Focus Areas	Test Isolation Strategy	Example Assertions
Data Model (`ResourceAttributes`, `LogRecord`, `Span`, `MetricPoint`)	Field validation, correlation ID handling, method logic (`IsValid`, `HasTraceContext`, `Duration`)	Instantiate structs directly with test data.	`ResourceAttributes` with empty `service_name` is invalid. A `LogRecord` with a `trace_id` returns `true` for `HasTraceContext`.
Ingestion Pipeline (`batchBuffer`, `Normalizer`)	Buffer flushing logic, batch size limits, normalization rules (timestamp conversion, severity mapping).	Mock channels and downstream dependencies.	`batchBuffer.add` triggers flush at `MAX_BATCH_SIZE`. `Normalizer.NormalizeLog` converts a numeric severity string to a `LogSeverity` enum.
Log Storage (`InMemorySegment`, `WriteAheadLog`)	Log appends, in-memory indexing, WAL append/read recovery.	Use temporary directories for file-based components.	`WriteAheadLog.Append` followed by `ReadFrom` returns identical data. `InMemorySegment` index lookup returns correct log IDs.
Metric Storage (`Chunk`, `Series`)	Gorilla compression encoding/decoding, exemplar attachment.	Create standalone `Chunk` instances with synthetic data points.	A sequence of timestamp-value pairs encoded then decoded yields the original values.
Trace Storage (`BuildTreeFromSpans`)	Span tree reconstruction from an unordered list.	Provide a list of `Span` structs with parent-child relationships.	The returned tree has the correct root span and hierarchical structure.
Query Engine (`Planner`, `PlanNode` types)	Query parsing, filter pushdown logic, plan node construction.	Mock the storage engine interfaces.	A query with a `service_name` filter generates a `ScanTracesNode` with that filter pushed down.
Alerting Engine (`RuleEvaluator`, state machine)	Condition evaluation, state transitions (`updateAlertState`), notification routing logic.	Mock the `QueryEngine` and `NotificationChannel`.	A rule with `for_duration` of "5m" transitions from `Inactive` to `Pending` when the condition is met, and to `Firing` after 5 minutes.

Common Pitfalls in Unit Testing:

- ⚠️ **Pitfall: Testing Implementation Details Instead of Behavior.** Writing tests that break when internal private methods change, even though the public API behavior remains correct.
  - **Why it's wrong:** Creates brittle tests that hinder refactoring. The tests should validate the contract, not the implementation path.
  - **How to avoid:** Test only through exported functions and verify observable state changes or return values. Use interfaces to define contracts.
- ⚠️ **Pitfall: Incomplete Validation of Edge Cases.** Only testing "happy path" scenarios for data structures (e.g., `Span.Duration()` with valid `start_time` and `end_time`).
  - **Why it's wrong:** The platform will ingest malformed data from the wild. Methods must handle zero, negative, or missing durations gracefully.
  - **How to avoid:** For every unit, create tests for boundary conditions: zero values, nil pointers, empty strings, maximum sizes, and invalid enumerations.

Integration Tests: Validating Component Interactions

Integration tests verify that multiple units work together correctly. They involve real instances of components, often with lightweight test doubles for external dependencies (e.g., using a temporary SQLite database instead of production PostgreSQL).

Integration Point	Components Involved	Test Scenario	Validation Focus
Pipeline to Storage	`Pipeline`, `Normalizer`, `LogStorageEngine`, `MetricStorageEngine`, `TraceStorageEngine`	Feed a batch of mixed telemetry through the pipeline.	Records are correctly routed, normalized, and persistently stored. The `CrossSignalCorrelationIndex` is updated.
Query Engine to Storage	`QueryEngine`, all three storage engines, `CrossSignalCorrelationIndex`	Execute a `QueryRequest` for logs within a time range.	The engine calls the correct storage engine's methods with the right filters and returns paginated results.
Cross-Signal Query	`QueryEngine`, `CorrelationIndex`, multiple storage engines.	Execute a `CrossSignalQueryRequest` to find logs for traces with high latency.	The engine uses the index to find trace IDs, fetches those traces, then fetches associated logs, merging results correctly.
Alert Rule Evaluation	`RuleEvaluator`, `QueryEngine`, `RuleStore`	Load a rule that queries metric averages, and trigger evaluation.	The evaluator fetches the rule, executes the query via the engine, and correctly updates the alert state based on the result.
Notification Routing	`AlertingEngine`, concrete `NotificationChannel` implementations (e.g., a test webhook).	Fire an alert and verify the notification is dispatched.	The alert is routed to the configured channel, and the channel's `Send` method is invoked with the correct `AlertInstance` data.

Design Decision: Using Docker for Integration Test Dependencies

> **Decision: Containerized Dependencies for Integration Tests**
>
> - **Context:** Several components (like the storage engines) may rely on external services for testing (e.g., a real full-text search index for logs). We need a reproducible, isolated environment for developers and CI.
> - **Options Considered:**
>   1. **Embedded In-Process Dependencies:** Use pure-Go libraries that emulate the storage functionality (e.g., an in-memory Bleve for search). This is fast but may not catch bugs specific to the real storage driver.
>   2. **Test-Containers with Docker:** Use Docker to spin up real backend services (like Elasticsearch, Prometheus, Jaeger) ephemerally for the duration of the test suite.
>   3. **Mock Everything:** Mock all external dependencies at the interface level. This is fast and hermetic but provides the weakest guarantee of real-world compatibility.
> - **Decision:** Use Docker-based test containers for integration tests that require external services, supplemented by extensive interface mocks for unit tests.
> - **Rationale:** This provides a strong balance. Developers and CI can run realistic integration tests that catch driver-level issues and configuration mismatches. The tests remain reproducible because the container versions are pinned. The added complexity is managed by Go's `testcontainers` library.
> - **Consequences:** Developers must have Docker installed to run the full test suite. Integration tests will be slower and are therefore tagged separately (`//go:build integration`). The CI pipeline must be configured to support Docker.

End-to-End (E2E) Tests: Validating User Journeys

E2E tests simulate real user scenarios from start to finish. They treat the entire platform as a black box, sending telemetry via the OTLP endpoint, waiting for processing, and then querying results. These tests are expensive but critical for validating the complete data flow and configuration.

User Journey	Test Steps	Success Criteria
Telemetry Ingestion & Correlation	1. Start the platform with all components. 2. Send a trace with an embedded log record and a metric with an exemplar via the OTLP receiver. 3. Wait for processing (use health/ready probes). 4. Query for the trace by its ID, then execute a cross-signal query for related logs and metrics.	The trace is stored and retrievable. The log record is found and linked to the trace via `trace_id`. The metric exemplar references the correct trace ID.
Alert Detection and Notification	1. Deploy the platform with a pre-loaded alert rule that triggers on a high error log rate. 2. Ingest a burst of error logs to exceed the threshold. 3. Wait for the alert evaluation interval. 4. Poll the alerting API for active alerts or capture notifications sent to a test webhook.	An alert instance is created in the `Firing` state. A notification is received at the test webhook with the correct alert metadata.
Query Performance SLO	1. Pre-populate storage with a known large dataset (e.g., 1M logs). 2. Execute a full-text search query with a filter for a specific service. 3. Measure the response time from the query API.	The query returns within the required latency (e.g., 500ms) and the results are correct.

Execution Strategy: E2E tests should be run in a dedicated CI stage, not on every developer commit. Use a dedicated test configuration with minimal resource allocations to keep runtimes manageable. Consider using **property-based testing** (via `github.com/leanovate/gopter` for Go) for the data model and serialization layers to automatically generate thousands of test cases exploring edge conditions in field values, Unicode in log bodies, and extreme timestamp ranges.

Milestone Implementation Checkpoints

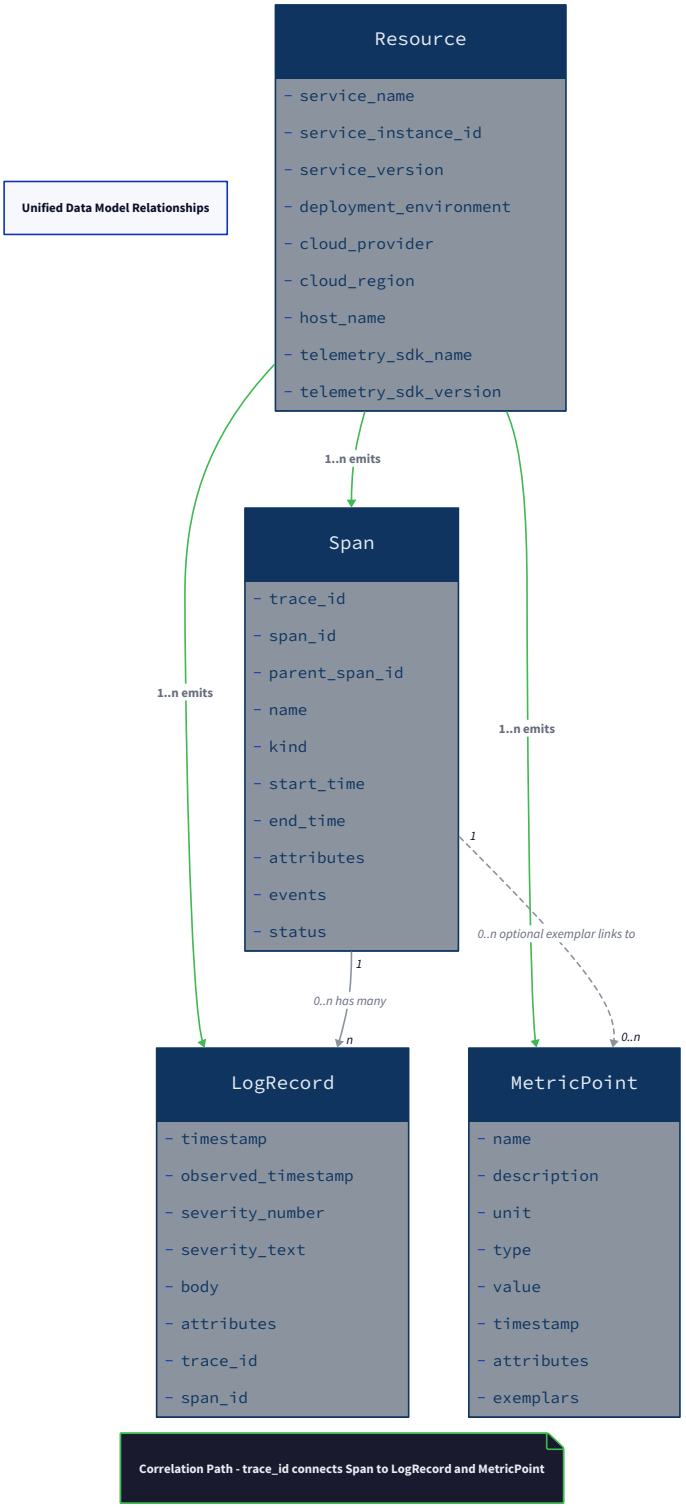
Each milestone delivers a testable slice of functionality. The checkpoints below describe the concrete tests to write and the expected outcomes to verify your implementation is on track. Use these as a guide to prioritize test development alongside feature implementation.

Milestone 1: Unified Data Model Checkpoint

Goal: Verify that the core data structures are correctly defined and that correlation mechanisms work.

Unified Data Model Relationships - Inspired by OpenTelemetry Semantic Conventions

Key Design Principle - Every piece of telemetry carries a universal identification card



Correlation Path - trace_id connects Span to LogRecord and MetricPoint

Test Area	What to Test	Expected Outcome
Resource Model Validation	Create a `ResourceAttributes` struct with missing `service_name`. Call `IsValid()`.	The method returns `false`. A valid struct returns `true`.
Trace-Log Correlation	Create a `LogRecord` with a valid 32-character `trace_id` and `span_id`. Call `HasTraceContext()`.	The method returns `true`. A log without these IDs returns `false`.
Exemplar Attachment	Create a `MetricPoint` with an `Exemplar` containing a `trace_id`. Call `HasExemplar()`.	The method returns `true`. The exemplar's `trace_id` matches the one set.
Span Relationships	Create three `Span` instances: a root (no `parent_span_id`), a child (points to root's `span_id`), and a grandchild. Pass them to `BuildTreeFromSpans()`.	The function returns a tree where the root span's children include the child span, and the child's children include the grandchild. `IsRoot()` returns `true` only for the root span.
Serialization Round-Trip	Serialize a `LogRecord` to JSON (or Protobuf), then deserialize it back into a new struct.	All fields are equal, including nested maps and the `ResourceAttributes`.

Checkpoint Verification: After implementing Milestone 1, run a dedicated test suite. You should be able to execute a Go test that demonstrates correlation:

```sh
go test ./internal/telemetry/... -v
```

The output should show passing tests for validation, correlation checks, and tree building. As a manual test, write a small Go program that creates a trace with two spans and a log attached to one span, then prints the trace tree structure. Verify the log is correctly associated.

Milestone 2: Data Ingestion Pipeline Checkpoint

Goal: Verify that the pipeline accepts OTLP data, applies backpressure, batches, normalizes, and passes data to downstream components.

Test Area	What to Test	Expected Outcome
OTLP/gRPC Server	Start the `otlpService` gRPC server on `OTLP_GRPC_PORT`. Send a valid OTLP trace export request using a gRPC client.	The server accepts the connection and returns a successful response. The traces appear in the pipeline's output channel (or a test double).
Batch Buffer Logic	Instantiate a `batchBuffer` with `maxSize=10`. Add 11 items using `add()`.	After the 10th item, `add()` returns `true` indicating a size-based flush. The 11th item starts a new batch.
Time-Based Flush	Configure `batchBuffer` with a short `flushInterval` (e.g., 10ms). Add one item and wait longer than the interval. Call `shouldFlushOnTime()`.	The method returns `true`. Calling `take()` returns the single item.
Backpressure	Set the `Pipeline`'s channel buffer to a small size (e.g., 1). Attempt to send more log records concurrently than the buffer can hold. Call `ReceiveLog()`.	Once the buffer is full, subsequent calls return `ErrPipelineFull` (or a similar error) without blocking.
Normalization	Send a `LogRecord` with a numeric severity "3" via the `Normalizer.NormalizeLog()`.	The normalized record's `severity` field is the string `"ERROR"` (or the `LogSeverity` enum value `SeverityError`).

Checkpoint Verification: After implementing Milestone 2, run integration tests that start the pipeline and send data. You can use a command like:

```sh
go test ./internal/pipeline/... -count=1
```

Additionally, run a **load test** using a simple Go program that spawns multiple goroutines sending telemetry to the gRPC endpoint. Monitor that the pipeline doesn't crash, memory usage remains stable, and data eventually flows through. Use the `pprof` endpoint to check for goroutine leaks.

Milestone 3: Multi-Signal Storage Checkpoint

Goal: Verify that each storage engine correctly persists and retrieves its respective data type, and that the cross-signal index enables correlation.

Test Area	What to Test	Expected Outcome
Log Storage Write/Read	Use `LogStorageEngine.WriteBatch()` to write 100 `LogRecord`s. Then, query for logs within the ingested time range using a `LogQuery`.	All 100 logs are returned. Full-text search for a unique word present in one log body returns only that log.
Log Indexing	Write logs with different `service_name` attribute values. Query with a filter `service_name=api`.	Only logs from the "api" service are returned.
Metric Compression	Write a continuous stream of metric points for a single series over 1,000 timestamps. Read back the series for the same time range.	The returned data points match the original values within a small tolerance (if using lossy compression). The on-disk size is significantly smaller than raw data.
Trace Tree Reconstruction	Write 50 spans belonging to 5 different traces (each with a tree structure). Query for a specific trace by its ID.	The returned trace contains all spans belonging to that trace, and the spans are organized in a correct tree hierarchy (accessible via a root span).
Cross-Signal Index	Write a trace, a log with that trace's ID, and a metric with an exemplar containing the same trace ID. Call `CrossSignalCorrelationIndex.Get()` for that trace ID.	The returned `CorrelationRecord` contains references to the log record and the metric exemplar with correct storage hints.

Checkpoint Verification: After implementing Milestone 3, run storage engine tests with real, persisted data in a temporary directory. Use a command that includes both unit and integration tests:

```sh
go test ./internal/storage/... -tags=integration
```

Verify that the tests clean up their temporary directories. As a manual integration check, start the platform, send correlated telemetry (trace+log+metric), then use a **debug endpoint** (to be implemented) on the `CrossSignalCorrelationIndex` to look up the trace ID and confirm it has references. Also, verify that a full-text search for a log returns in under 500ms for a dataset of 1M logs (you may need to generate synthetic load).

Milestone 4: Unified Query Interface Checkpoint

Goal: Verify that the query engine can parse queries, create efficient execution plans, execute them across storage engines, and return paginated results.

Test Area	What to Test	Expected Outcome
Query Parsing & Planning	Provide a query string: `logs{service_name="api" AND severity="ERROR"} limit 20`. Parse and build a plan.	
Filter Pushdown	Execute a query for traces with `duration > 2s`. Monitor the call to `TraceStorageEngine.Query`.	The engine is called with a `TraceQuery` where `minDuration` is set to 2 seconds, demonstrating the filter was pushed down and not applied post-scan.
Cross-Signal Query Execution	Execute a `CrossSignalQueryRequest`: find traces where `duration > 5s`, and fetch related logs.	The engine first queries the trace storage for slow traces, then uses the `CorrelationIndex` to find log IDs for those trace IDs, then fetches those logs. The response contains a merged result set with traces and their associated logs.
Cursor-Based Pagination	Execute a query for all logs with a limit of 10. The response includes a `NextCursor`. Use that cursor in a subsequent query.	The second query returns the next 10 logs, with no duplicates from the first set. After all logs are exhausted, the `NextCursor` is empty.
Query API Endpoints	Send an HTTP POST to `/api/v1/query` with a valid JSON `QueryRequest`.	The response is JSON with a `QueryResponse` structure, HTTP status 200.

Checkpoint Verification: After implementing Milestone 4, run the query engine tests and also start the platform's HTTP server to run manual API tests. Use `curl` or a script to send queries:

```sh
go test ./internal/query/... -v
# Then, in another terminal, start the server and test:
curl -X POST http://localhost:8080/api/v1/query -d '{"signal":"logs", "filters":[{"field":"service_name","op":"=","value":"api"}]}'
```

The response should be valid JSON with log results. Test pagination by sending a query with a limit of 1 and iterating through the cursor until all data is consumed. Verify that cross-signal queries return linked data.

Milestone 5: Alerting & Anomaly Detection Checkpoint

Goal: Verify that alert rules are loaded and evaluated, anomalies are detected based on statistical baselines, and notifications are routed correctly with deduplication.

Test Area	What to Test	Expected Outcome
Rule Loading	Place a YAML file defining an `AlertRule` in the rules directory watched by `RuleStore`. Call `RuleStore.GetAllRules()`.	The rule is loaded, parsed, and available via the API. Updates to the file trigger an update on the `UpdateChan()`.
Rule Evaluation	Create a rule with a condition `avg_over_time(metric_name[5m]) > 100`. Mock the `QueryEngine` to return a value of 150. Run `RuleEvaluator.Evaluate()`.	The evaluator calls the query engine with the correct query, determines the condition is met, and transitions the alert state from `Inactive` to `Pending` (or `Firing` if `for_duration` is zero).
State Machine Transitions	Simulate a rule with `for_duration=2m`. Feed condition-met signals for 1 minute, then a condition-not-met signal.	The alert state goes `Inactive` → `Pending` after the first evaluation, then back to `Inactive` after the second (because the condition didn't hold for the required duration).
Anomaly Detection Baseline	Feed a stream of normal metric values to the anomaly detector's learning phase, then feed a value 5 standard deviations away.	The detector marks the outlier value as an anomaly (e.g., returns a high z-score).
Notification Deduplication	Trigger the same alert (identical fingerprint) multiple times within a short interval.	Only one notification is sent per `repeat_interval`. Subsequent firings within that window do not trigger new notifications.
Channel Routing	Configure two notification channels: Slack and PagerDuty. Fire an alert with `notification_channels` listing both.	Both channel's `Send` methods are invoked with the alert data.

Checkpoint Verification: After implementing Milestone 5, run the alerting engine tests. Then, start the platform with a sample rule that triggers on a simple condition (e.g., log count > 0). Ingest a log that satisfies the condition and wait for the evaluation interval. Check the alerting API (`/api/v1/alerts`) for a firing alert. Verify that a test notification channel (like a mock webhook) receives the alert payload.

```sh
go test ./internal/alerting/... -v
```

For manual verification, look at the logs of the `AlertingEngine` to see the evaluation cycles and state transitions. Use the **silencing** feature to silence an alert and confirm no notifications are sent.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (for learning/first iteration)	Advanced Option (for production-readiness)
Unit Test Framework	Go's built-in `testing` package + `testify/assert` for assertions.	Same, plus `github.com/stretchr/testify/suite` for test suites.
Mock Generation	Hand-written mocks implementing interfaces.	`github.com/golang/mock` (mockgen) for automatic generation.
Property-Based Testing	Manual generation of edge cases.	`github.com/leanovate/gopter` for automatic property-based testing.
Integration Test Containers	Manually start Docker containers via scripts.	`github.com/testcontainers/testcontainers-go` for programmatic container lifecycle.
E2E Test Orchestration	Bash script to start components, send data via curl, assert outputs.	Go test with `os/exec` to manage processes, use `net/http` for API calls.
Code Coverage	`go test -cover`	`go test -coverprofile` + integrate with SonarQube/Codecov.
Performance Benchmarking	Go's built-in benchmarking (`go test -bench .`).	Custom load test scripts using `vegeta` for HTTP load testing.

B. Recommended File/Module Structure

Add a dedicated `test` directory for integration and E2E test utilities, keeping unit tests alongside the code they test (Go convention).

```
project-root/
├── cmd/
│   ├── server/                # Main server binary
│   └── cli/                   # CLI tools (e.g., for sending test data)
├── internal/
│   ├── telemetry/             # Data model (Milestone 1)
│   │   ├── model.go
│   │   ├── model_test.go      # Unit tests for data structures
│   │   └── correlation.go
│   ├── pipeline/              # Ingestion pipeline (Milestone 2)
│   │   ├── pipeline.go
│   │   ├── pipeline_test.go
│   │   ├── normalizer.go
│   │   └── batch_buffer.go
│   ├── storage/               # Storage engines (Milestone 3)
│   │   ├── log/
│   │   │   ├── engine.go
│   │   │   ├── engine_test.go
│   │   │   ├── wal.go
│   │   │   └── segment.go
│   │   ├── metric/
│   │   │   ├── engine.go
│   │   │   └── chunk.go
│   │   ├── trace/
│   │   │   └── engine.go
│   │   └── correlationindex/  # Cross-signal index
│   │       └── index.go
│   ├── query/                 # Query engine (Milestone 4)
│   │   ├── engine.go
│   │   ├── planner.go
│   │   ├── parser.go
│   │   └── exec/              # Execution nodes
│   │       ├── scan_node.go
│   │       └── join_node.go
│   └── alerting/              # Alerting engine (Milestone 5)
│       ├── engine.go
│       ├── rule_store.go
│       ├── evaluator.go
│       └── notifier/
│           ├── slack.go
│           └── webhook.go
├── test/                      # Integration and E2E tests
│   ├── integration/
│   │   ├── pipeline_integration_test.go
│   │   ├── storage_integration_test.go
│   │   └── testhelpers/       # Common setup utilities
│   │       └── containers.go  # Testcontainers helpers
│   └── e2e/
│       ├── telemetry_flow_test.go
│       └── alerting_flow_test.go
└── go.mod
```

C. Infrastructure Starter Code: Test Container Helper

To facilitate integration tests with Docker, here's a complete helper to manage a container running a backing service (e.g., a search engine for logs). This uses `testcontainers-go`.

```go
// test/integration/testhelpers/containers.go

package testhelpers

import (
    "context"
    "fmt"
    "time"

    "github.com/testcontainers/testcontainers-go"
    "github.com/testcontainers/testcontainers-go/wait"
)

// ElasticsearchContainer wraps a testcontainers instance of Elasticsearch.
// This is an example; adapt for your actual storage dependencies.
type ElasticsearchContainer struct {
    testcontainers.Container
    URI string
}

// StartElasticsearchContainer starts an Elasticsearch container with default settings.
func StartElasticsearchContainer(ctx context.Context) (*ElasticsearchContainer, error) {
    req := testcontainers.ContainerRequest{
        Image:        "docker.elastic.co/elasticsearch/elasticsearch:8.10.0",
        ExposedPorts: []string{"9200/tcp"},
        Env: map[string]string{
            "discovery.type": "single-node",
            "ES_JAVA_OPTS":   "-Xms512m -Xmx512m",
            "xpack.security.enabled": "false",
        },
        WaitingFor: wait.ForHTTP("/").
            WithPort("9200").
            WithStartupTimeout(120 * time.Second),
    }
    container, err := testcontainers.GenericContainer(ctx, testcontainers.GenericContainerRequest{
        ContainerRequest: req,
        Started:          true,
    })
    if err != nil {
        return nil, fmt.Errorf("failed to start elasticsearch container: %w", err)
    }

    ip, err := container.Host(ctx)
    if err != nil {
```

```
        return nil, fmt.Errorf("failed to get container host: %w", err)
    }
    mappedPort, err := container.MappedPort(ctx, "9200")
    if err != nil {
        return nil, fmt.Errorf("failed to get mapped port: %w", err)
    }

    uri := fmt.Sprintf("http://%s:%s", ip, mappedPort.Port())

    return &ElasticsearchContainer{Container: container, URI: uri}, nil
}
```

D. Core Logic Skeleton Code: Unit Test for `batchBuffer`

Here is a skeleton for a comprehensive unit test for the `batchBuffer`. Fill in the TODOs to implement the tests.

```go
// internal/pipeline/batch_buffer_test.go

package pipeline


import (

    "testing"

    "time"


    "github.com/stretchr/testify/assert"

    "github.com/yourproject/internal/telemetry"

)


func TestBatchBuffer_AddFlushOnSize(t *testing.T) {

    // TODO 1: Create a batchBuffer with maxSize = 5.

    // TODO 2: Create a mock resource attribute.

    // TODO 3: Call add() 5 times with dummy items.

    //       - After each of the first 4 calls, assert that add() returns false (no flush).

    //       - After the 5th call, assert add() returns true (flush triggered by size).

    // TODO 4: Call take() and verify it returns 5 items and the correct resource.

    // TODO 5: After take(), the buffer should be empty; verify by checking internal state or calling take again (should return empty slice).

}


func TestBatchBuffer_TimeBasedFlush(t *testing.T) {

    // TODO 1: Create a batchBuffer with flushInterval = 10 milliseconds.

    // TODO 2: Add one item.

    // TODO 3: Sleep for longer than flushInterval (e.g., 15ms).

    // TODO 4: Call shouldFlushOnTime() and assert it returns true.

    // TODO 5: Call take() and verify the item is returned.

}


func TestBatchBuffer_ConcurrentAccess(t *testing.T) {

    // TODO 1: Create a batchBuffer with a reasonably large maxSize.

    // TODO 2: Start multiple goroutines that concurrently call add() with different items.

    // TODO 3: Use a sync.WaitGroup to wait for all goroutines to finish.

    // TODO 4: Ensure no race conditions cause panics (run with `go test -race`).

    // TODO 5: Verify that the total number of items added matches the sum of items taken after flushing.

}
```

E. Language-Specific Hints

- **Use `t.Cleanup`:** In Go tests, use `t.Cleanup(func() { ... })` to register cleanup actions (like stopping containers, deleting temp files) that run after the test finishes, even if the test fails.
- **Parallel Tests:** Mark independent unit tests with `t.Parallel()` to speed up execution. Avoid this in tests that share global state or external resources.
- **Table-Driven Tests:** Use table-driven tests for testing multiple input/output combinations for a single function. This keeps tests concise and makes it easy to add new cases.

```go
func TestResourceAttributes_IsValid(t *testing.T) {
    cases := []struct {
        name     string
        input    telemetry.ResourceAttributes
        expected bool
    }{
        {"valid", telemetry.ResourceAttributes{ServiceName: "api", ServiceVersion: "1.0"}, true},
        {"missing service name", telemetry.ResourceAttributes{ServiceVersion: "1.0"}, false},
    }
    for _, tc := range cases {
        t.Run(tc.name, func(t *testing.T) {
            assert.Equal(t, tc.expected, tc.input.IsValid())
        })
    }
}
```

- **Golden Files:** For complex output (like serialized JSON or query plans), consider using golden files. Write the expected output to a file (`testdata/*.golden`) and compare during tests. This makes it easy to update expectations when the output changes intentionally.

F. Milestone Checkpoint

After completing each milestone, run the specific test suite and verify the acceptance criteria. For example, after **Milestone 3**, you should be able to:

1. Run the storage engine integration tests and see them pass:

```sh
cd test/integration && go test -tags=integration ./... -v 2>&1 | grep -E "PASS|FAIL"
```

Expect output showing `PASS` for all tests. 2. Manually test via a small script that:

- Starts the platform (`go run cmd/server/main.go`).
- Uses the CLI tool (`go run cmd/cli/main.go`) to send a trace, a log, and a metric.
- Queries the trace via the API and verifies the log and metric exemplar are linked.

If tests fail, check the logs of the storage engines for errors. Common issues include incorrect file permissions for temporary directories, missing environment variables for test containers, or misconfigured index mappings.

G. Debugging Tips for Tests

Symptom	Likely Cause	How to Diagnose	Fix
Unit test passes locally but fails in CI.	Environment differences (time zones, file system, missing binaries).	Check CI logs for error messages. Run the test with `-v` to see detailed output. Look for absolute paths or OS-specific code.	Use relative paths, temporary directories created with `t.TempDir()` , and avoid assumptions about external tools.
Integration test hangs indefinitely.	Container startup timeout or deadlock in the code under test.	Add debug logs in the test setup. Use `testcontainers` ' `Debug` mode to see container logs. Check for goroutine leaks with `pprof` .	Increase timeout for container readiness checks. Ensure proper cleanup of resources (close connections, stop goroutines).
Race condition detected by `-race` .	Concurrent access to a shared mutable data structure without synchronization.	The race detector output shows the offending lines. Look for maps or slices accessed from multiple goroutines.	Protect access with a `sync.Mutex` or `sync.RWMutex` . Use channels for communication instead of shared memory where possible.
Test passes but real usage fails.	Mock is not simulating real behavior accurately.	Review the mock implementation: does it return data in the same format as the real component? Check for subtle differences in error handling or data ordering.	Refine the mock to better match the contract. Consider using integration tests with the real component for critical paths.

> **Milestone(s):** Milestone 2 (Data Ingestion Pipeline), Milestone 3 (Multi-Signal Storage), Milestone 4 (Unified Query Interface), Milestone 5 (Alerting & Anomaly Detection)

10. Debugging Guide

Building a distributed observability platform involves complex interactions between ingestion pipelines, storage engines, and query systems. When things go wrong—data disappears, queries time out, or memory usage explodes—you need systematic approaches to diagnose and fix issues. This guide provides a mental model for debugging the platform, a comprehensive symptom-cause-fix catalog, and practical techniques for inspecting internal state.

Think of debugging this system as being an **aircraft mechanic troubleshooting a complex jet**. You have multiple subsystems (engines, hydraulics, avionics) that must work together. You can't just look at the whole plane and see the problem—you need diagnostic tools (flight data recorders, maintenance computers), systematic checklists for common failures, and the ability to isolate subsystems to pinpoint root causes. This section provides your maintenance manual and diagnostic toolkit.

Common Bugs: Symptom → Cause → Fix

The following table maps observable symptoms in the running platform to their likely root causes, investigation steps, and concrete fixes. Use this as your first reference when encountering problems during development or operation.

Symptom	Potential Causes	Investigation Steps	Fix
No logs appear in search results	1. **Logs not indexed**: The full-text or field index wasn't updated. 2. **Corrupted Write-Ahead Log (WAL)**: Log records were written to WAL but not processed to segments. 3. **Time range mismatch**: Query `StartTime` / `EndTime` doesn't overlap with ingested log timestamps. 4. **Filter too restrictive**: Query `Filters` exclude all records. 5. **Segment not searchable**: In-memory segment wasn't flushed, or immutable segment index is missing.	1. Check `LogStorageEngine` metrics for indexed vs. written counts. 2. Inspect WAL file size and check for errors in `WriteAheadLog.Append`. 3. Verify log `timestamp` normalization (should be Unix nanoseconds). 4. Temporarily remove filters in `LogQuery` to see if logs appear. 5. Check `InMemorySegment` size and `ImmutableSegment` directory for index files.	1. Ensure `LogStorageEngine.WriteBatch` index update after WAL write. 2. Implement WAL recovery on startup using `WriteAheadLog.ReadFrom`. 3. Normalize timestamps to UTC nanoseconds i `Normalizer.NormalizeLog`. 4. Validate filter syntax and ensure attribute nam stored `LogRecord.attributes`. 5. Verify background compaction flushes segme `maxSize` reached.
High memory usage in ingestion pipeline	1. **Backpressure not working**: Producers faster than consumers, causing unbounded channel growth. 2. **Batch buffer not flushing**: `batchBuffer` exceeds `maxSize` but isn't flushed due to timer bug. 3. **Memory leak in normalizer**: `Normalizer` retains references to processed items. 4. **Deadlock in pipeline stages**: One goroutine blocked, causing others to fill buffers.	1. Monitor channel lengths (`len(pipeline.logChan)`). 2. Check `batchBuffer.lastFlush` time and `batchBuffer.items` count. 3. Use Go's `pprof` heap profile to see which types accumulate. 4. Check for goroutine count growth with `pprof` goroutine profile.	1. Implement proper backpressure: return `ErrPipelineFull` from `Pipeline.Receiv` when channel buffer full. 2. Ensure `batchBuffer.shouldFlushOnTi` wall clock, not monotonic time. 3. Clear slices/nil references after processing in `Normalizer` methods. 4. Use `sync.Mutex` consistently and avoid cir channel dependencies.
Query times out or returns partial results	1. **Storage engine slow**: Full-text search scanning too many segments, metric query scanning uncompressed chunks. 2. **No query plan optimization**: `Planner.BuildPlan` creates inefficient plan (e.g., no filter pushdown). 3. **Pagination cursor broken**: `Cursor` in `QueryResponse` malformed, causing infinite loop. 4. **Cross-signal join too expensive**: `CorrelationJoinNode` materializes large intermediate results.	1. Check query latency metrics per storage engine. 2. Log the execution plan from `Planner.BuildPlan` and examine for full scans. 3. Test pagination with small `Limit` and verify `NextCursor` changes. 4. Profile query execution with `pprof` to see time spent in join logic.	1. Add query optimizations: time-range pruning i engines, metric chunk skipping via `minTime` / `maxTime`. 2. Implement **filter pushdown**: pass `Filters` `ScanLogsNode` etc. 3. Ensure cursor encodes enough state (last tim ID) to resume correctly. 4. Use streaming merge in `CorrelationJoin` `MergeStreams` instead of materializing.
Trace tree reconstruction fails or shows missing spans	1. **Parent-child references broken**: `Span.parent_span_id` points to non-existent span. 2. **Spans stored out of order**: Trace queries fetch spans from different storage blocks, missing some. 3. **TraceID mismatch**: Spans have same `trace_id` but different `resource` (different services). 4. **Span timestamp corruption**: `start_time` > `end_time` causing negative duration.	1. Validate `Span.parent_span_id` length equals `SpanIDLength` (16). 2. Check `TraceStorageEngine` query logic: does it fetch all blocks overlapping time range? 3. Verify `trace_id` normalization (32-character hex). 4. Add validation in `Normalizer.NormalizeSpan` for timestamps.	1. When building tree, handle orphan spans by n them root nodes with warning. 2. Ensure `TraceStorageEngine` queries use `TraceMetadata.spanIDs` or scans all releva `SpanStorageBlock`. 3. Normalize `trace_id` to lowercase hex in `Normalizer`. 4. Swap timestamps if `start_time` > `end_t` log correction.
Metrics show gaps or incorrect values	1. **Chunk compression corruption**: `Chunk.encodePoint` produces unreadable data for some timestamp patterns. 2. **Exemplar indexing failed**: `CrossSignalCorrelationIndex.IndexLogRecord` not called for metric exemplars. 3. **Downsampling artifacts**: Aggregated data loses precision, showing stair-step patterns. 4. **High cardinality explosion**: Too many unique `Series` due to unbounded attributes.	1. Write unit test for `Chunk.encodePoint`/decode roundtrip with random data. 2. Check if `MetricPoint.HasExemplar()` true but exemplar not indexed. 3. Compare raw vs. downsampled data for same time range. 4. Monitor `MetricStorageEngine` series count metric.	1. Fix Gorilla compression: ensure delta-of-delta calculation handles timestamp wraparound. 2. Call `CrossSignalCorrelationIndex.IndexLo` for each exemplar in metric batch. 3. Adjust downsampling resolution: use larger `CHUNK_DURATION` for older blocks. 4. Enforce attribute cardinality limits: reject metri >100 unique label combinations.
Alert rules fire incorrectly (false positives)	1. **Rule condition evaluation bug**: `RuleEvaluator.evaluateCondition` misinterprets query result. 2. **Missing `for_duration`** : Alert fires immediately without waiting for sustained condition. 3. **Query engine returns stale data**: Cached or delayed metric data triggers old threshold breach. 4. **Timestamp misalignment**: Rule evaluation uses incorrect time window (off by timezone).	1. Log the query result and condition evaluation details. 2. Check `AlertRule.for_duration` parsing and `RuleEvaluator.updateAlertState` logic. 3. Verify metric storage ingestion lag; check `MetricPoint.timestamp` vs. wall clock.	1. Implement thorough condition testing for all op (>, <, ==, etc.). 2. Implement alert state machine properly: transi `Inactive` → `Pending` → `Firing`. 3. Add data freshness check in query: reject met than threshold. 4. Normalize all timestamps to UTC in rule evalu context.

Symptom	Potential Causes	Investigation Steps	Fix
		4. Inspect rule evaluation timestamps: ensure using UTC nanoseconds.	
Alert notifications not sent	1. **Notification channel misconfigured**: `NotificationChannel.ValidateConfig` passes but `Send` fails. 2. **Alert deduplication too aggressive**: `fingerprint` collision suppresses legitimate alerts. 3. **Notification routing bug**: `AlertRule.notification_channels` empty or referencing missing channel. 4. **Silence rule active**: `AlertRule.silenced` flag set but UI doesn't show it.	1. Test channel with `NotificationChannel.Test` method. 2. Check `AlertInstance.fingerprint` calculation (hash of labels). 3. Verify rule YAML parsing includes `notification_channels` field. 4. Check `RuleStore.GetAllRules` for `silenced` status.	1. Add retry with exponential backoff in `NotificationChannel.Send`. 2. Use collision-resistant fingerprint algorithm (e. SHA256). 3. Validate channel references when rule loads; warning for missing channels. 4. Implement separate silence database; don't m object directly.
Cross-signal correlation queries return no linked data	1. **Correlation index empty**: `CrossSignalCorrelationIndex.IndexLogRecord` never called or fails silently. 2. **TraceID mismatch formats**: Log `trace_id` (string) vs. index key (normalized hex) mismatch. 3. **Index partitioning by time**: Query outside indexed time range returns no results. 4. **Storage hints outdated**: `CorrelationRecord.log_references` points to deleted log segments.	1. Check index size metrics; verify indexing called in `Pipeline.fanoutStage`. 2. Log both log `trace_id` and index lookup key; compare normalization. 3. Verify index query uses same time range as root query. 4. Check log retention policy deletes segments but not index entries.	1. Add index write confirmation logging; ensure r swallowing. 2. Normalize all `trace_id` values to lowercas `Normalizer`. 3. Implement time-range partitioning in index; qu relevant partitions only. 4. Add background job to clean orphaned index e (reference validation).
Ingestion pipeline stops processing data	1. **Context cancellation**: Pipeline's `context.CancelFunc` called, shutting down goroutines. 2. **Panic in a stage**: A goroutine panics, leaving channel senders blocked forever. 3. **Deadlock in batch buffer**: `batchBuffer.mu` held while trying to send on full channel. 4. **Storage engine write failure**: `LogStorageEngine.WriteBatch` returns error, not retried.	1. Check `pipeline.cancelFunc` call stack; look for graceful shutdown signals. 2. Look for panic logs; add `defer recover()` in each stage goroutine. 3. Use deadlock detection tool or visualize goroutine states with `pprof`. 4. Check `Pipeline.fanoutStage` error handling; look for `Dead Letter Queue` entries.	1. Implement proper shutdown signal handling (v goroutines). 2. Add panic recovery with error logging and rest stage goroutine. 3. Acquire mutexes in consistent order; use `sel` default for non-blocking sends. 4. Implement retry with backoff and `Dead Lett Queue` for persistent failures.
High disk I/O on metric storage	1. **Chunk fragmentation**: Too many small `Chunk` files due to frequent flushes. 2. **No compression**: Metric data stored uncompressed, bloating disk usage. 3. **Exemplar storage overhead**: `Exemplar` data stored inline with each point, not sampled. 4. **Compaction not running**: Old chunks not merged into `Block` files.	1. Inspect metric storage directory: count of `.chunk` files. 2. Check `Chunk.data` size vs. number of points stored. 3. Examine exemplar storage ratio (exemplars per 1000 points). 4. Check background compaction goroutine status and logs.	1. Increase `CHUNK_DURATION` to create larger (tradeoff: memory). 2. Verify Gorilla compression enabled in `Chunk.encodePoint`. 3. Sample exemplars: store only 1 exemplar per based on value deviation. 4. Implement compaction scheduler that merges older than threshold.

Debugging Techniques and Tools

Effective debugging requires both proactive instrumentation and reactive inspection tools. Below are systematic approaches to understanding platform behavior, from high-level monitoring to deep internal state examination.

Mental Model: The Diagnostic Control Room

Imagine you're in a **NASA mission control room** during a spacecraft launch. You have multiple consoles showing different subsystems: telemetry feeds (logs), trajectory plots (metrics), communication transcripts (traces), and alert status boards. When an anomaly occurs, you don't guess—you consult the relevant console, correlate across displays, and may even run diagnostic subroutines on specific components. Your debugging tools are these consoles and diagnostic routines.

1. Structured Logging with Correlation IDs

Add detailed structured logs to key components, ensuring each log includes correlation identifiers to trace operations across the system.

Logging Location	What to Log	Key Fields to Include
`IngestionPipeline.ReceiveOTLP`	Request receipt, backpressure status	`trace_id` (from request), `batch_size`, `signal_type`, `resource.service_name`
`batchBuffer.take`	Buffer flush events	`item_count`, `resource.instance_id`, `signal_type`, `flush_reason` (size/time)
`Normalizer.Normalize*`	Normalization errors, corrections	`original_value`, `normalized_value`, `error`
Storage engine write methods	Write latency, error conditions	`record_count`, `duration_ms`, `error`, `storage_hint` (segment/chunk ID)
`QueryEngine.ExecuteQuery`	Query planning and execution	`query_hash`, `plan_summary`, `storage_calls`, `duration_per_stage`
`RuleEvaluator.Evaluate`	Rule evaluation steps	`rule_name`, `query_result`, `condition_met`, `alert_state_before`, `alert_state_after`

> **Implementation Insight:** Use a **request-scoped logger** that automatically injects a correlation ID (like the trace ID from incoming telemetry or a generated UUID for internal operations). This allows you to filter logs for a specific request's journey through ingestion, storage, and query.

2. Runtime Profiling with pprof

Go's `pprof` tool provides visibility into CPU, memory, and goroutine usage. Enable it via an HTTP endpoint in your main server.

Key Profiles to Capture:

Profile Type	What It Reveals	How to Use
CPU Profile	Functions consuming most CPU time	Run under load, capture 30s profile. Look for hot paths in compression, indexing, or query execution.
Heap Profile	Memory allocation by function and object type	Capture during steady state and after load spikes. Identify memory leaks (types that grow unbounded).
Goroutine Profile	Stack traces of all goroutines	Capture when goroutine count seems high. Look for blocked goroutines (waiting on channels, mutexes).
Block Profile	Where goroutines block (mutex, channel)	Enable with `runtime.SetBlockProfileRate`. Find contention points in pipeline or storage engines.
Mutex Profile	Mutex contention hotspots	Identify locks causing serialization bottlenecks (e.g., `batchBuffer.mu` held too long).

Investigation Workflow:

1. Start the server with pprof HTTP endpoint (`import _ "net/http/pprof"`).
2. Reproduce the issue (e.g., run load test for high memory).
3. While issue persists, capture profile: `go tool pprof http://localhost:6060/debug/pprof/heap`.
4. Analyze top consumers: `top10`, visualize with `web`.
5. Compare profiles before/after fix to validate improvement.

3. Internal Metrics Exposition

Instrument each component to expose Prometheus-style metrics that can be scraped and visualized (even via simple dashboard).

Component	Critical Metrics to Expose
Ingestion Pipeline	`ingestion_requests_total`, `ingestion_errors_total`, `pipeline_channel_length`, `batch_flush_size`, `normalization_errors`
Storage Engines	`storage_writes_total`, `storage_query_duration_seconds`, `storage_items_stored`, `index_size_bytes`, `compaction_runs_total`
Query Engine	`query_requests_total`, `query_duration_seconds`, `query_stages_count`, `cross_signal_queries_total`
Alerting Engine	`alert_rules_total`, `alert_evaluations_total`, `alerts_firing`, `notification_sends_total`, `notification_errors_total`

> **Debugging Tip:** When debugging performance, graph **`pipeline_channel_length`** over time. A steadily increasing length indicates backpressure needed. **`storage_query_duration_seconds`** percentiles (p50, p95, p99) reveal slow storage engines.

4. Storage Engine Inspection Utilities

Build simple command-line tools or HTTP endpoints to inspect internal storage state without running full queries.

Storage Engine	Inspection Commands	What to Look For
Log Storage	List segments, view WAL tail, dump index terms	Segment count growing unbounded, WAL not truncating, index missing common terms.
Metric Storage	List series, view chunk statistics, check exemplar count	Series count exceeding cardinality limit, chunk size too small (many files), exemplar sampling rate.
Trace Storage	Get trace by ID, list trace metadata blocks	Orphan spans (no parent), trace tree depth abnormal, span count per trace distribution.
Cross-Signal Index	Lookup trace_id, count references	Missing trace IDs that should be present, reference count mismatch with actual data.

Example Inspection Routine for Missing Logs:

1. Use tool to list all log segments: verify new segment created after recent writes.
2. Check WAL: `tail -f wal.log` to see if writes appear.
3. Search index for a known term: verify term appears and points to segment containing log.
4. Query segment directly via debug API to confirm log physically present.

5. Trace-Inception: Using the Platform to Debug Itself

Inject **diagnostic traces** into the platform's own code (e.g., trace ingestion pipeline operations) and view them in your own trace UI. This meta-observability helps debug complex flows.

1. **Add OpenTelemetry instrumentation** to key methods in pipeline, storage, and query components.
2. **Ensure these internal traces are ingested** into your own trace storage (separate service name, e.g., `observability-platform`).
3. When debugging, **query for traces from the platform itself** to see timing, errors, and flow.

This creates a powerful feedback loop: your observability platform can observe its own behavior, revealing bottlenecks or errors in its processing stages.

6. Controlled Experimentation and Isolation

When facing intermittent issues, systematically isolate components to identify the faulty one.

Technique	How to Apply	Diagnostic Value
Load shedding	Temporarily stop ingesting one signal type (logs only).	If problem disappears, issue is in that signal's processing path.
Storage bypass	Write incoming data to simple file instead of storage engines.	If pipeline works, problem is in storage layer; if not, in pipeline.
Query simplification	Replace complex cross-signal query with simple single-signal queries.	If simple queries work, issue is in query planner or join logic.
Rule disablement	Silence all alert rules.	If high load continues, problem is not alert evaluation.

7. Common Debugging Workflow

When encountering any issue, follow this systematic checklist:

1. **Reproduce and isolate**: Can you reliably reproduce? Under what conditions (load, data shape)?
2. **Check logs**: Look for ERROR/WARN lines with correlation IDs around the time of issue.
3. **Examine metrics**: Are there spikes in error rates, latency, or resource usage?
4. **Profile if needed**: For performance issues, capture pprof profiles during reproduction.
5. **Inspect internal state**: Use debug endpoints to verify data actually reached storage.
6. **Simplify the scenario**: Reduce to minimal test case (single service, one log line).
7. **Add more instrumentation**: If root cause still elusive, add temporary debug logs at suspected boundaries.
8. **Hypothesize and test**: Form a hypothesis, implement fix, test if symptom resolves.

> **Key Insight:** The most powerful debugging tool is **the ability to ask the system questions about its own state**. Build this capability in from the start via metrics, logs, and inspection APIs. The time invested here pays exponential dividends when things go wrong.

Implementation Guidance

A. Technology Recommendations Table

Debugging Component	Simple Option	Advanced Option
Structured Logging	`log/slog` with JSON output	OpenTelemetry Logs with automatic correlation injection
Profiling	`net/http/pprof` HTTP endpoints	Continuous profiling with Parca or Pyroscope integration
Metrics Exposition	Custom HTTP `/metrics` endpoint with Prometheus format	OpenTelemetry Metrics SDK with automatic instrumentation
Inspection Tools	HTTP debug endpoints (`/debug/*`) with JSON output	gRPC reflection + CLI tool with rich formatting (like `grpcui`)
Tracing Internals	Manual trace span creation in key functions	OpenTelemetry auto-instrumentation for Go (low-code)

B. Recommended File/Module Structure

Add debug utilities in a separate module to avoid bloating production code:

```
project-root/
  cmd/
    debug-tools/           ← standalone CLI tools for inspection
      inspect-logs.go
      inspect-metrics.go
      inspect-traces.go
      inspect-index.go
  internal/
    debug/                 ← debug HTTP handlers and utilities
      pprof.go             ← pprof endpoint setup
      metrics.go           ← internal metrics registry
      inspection/          ← storage inspection APIs
        logs.go
        metrics.go
        traces.go
        index.go
    instrumentation/       ← observability for the platform itself
      tracing.go           ← OpenTelemetry setup for internal traces
      logging.go           ← structured logger setup with correlation
```

C. Infrastructure Starter Code (Complete Debug Endpoints)

File: `internal/debug/pprof.go`

```go
package debug

import (
    "net/http"
    _ "net/http/pprof" // side-effect: registers pprof handlers
    "strconv"
)

// StartPprofEndpoint starts an HTTP server for pprof profiling on the given port.
// This should be run in a separate goroutine in main().
func StartPprofEndpoint(port int) error {
    addr := "localhost:" + strconv.Itoa(port)
    return http.ListenAndServe(addr, nil)
}
```

File: `internal/debug/metrics.go`

```go
package debug

import (
    "net/http"
    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
    // Example metric for pipeline channel length
    PipelineChannelLength = prometheus.NewGaugeVec(
        prometheus.GaugeOpts{
            Name: "pipeline_channel_length",
            Help: "Current length of ingestion pipeline channels",
        },
        []string{"channel"},
    )
)

func init() {
    prometheus.MustRegister(PipelineChannelLength)
}

// MetricsHandler returns an HTTP handler for Prometheus metrics scraping.
func MetricsHandler() http.Handler {
    return promhttp.Handler()
}

// UpdateChannelLength updates the metric for a specific channel.
// Call this periodically in pipeline stages.
func UpdateChannelLength(channelName string, length int) {
    PipelineChannelLength.WithLabelValues(channelName).Set(float64(length))
}
```

D. Core Logic Skeleton Code (Debug Inspection Endpoints)

File: `internal/debug/inspection/logs.go`

```go
package inspection

import (
    "encoding/json"
    "net/http"
    "strconv"

    "yourproject/internal/storage/log"
)

type LogInspector struct {
    storage *log.LogStorageEngine
}

func NewLogInspector(storage *log.LogStorageEngine) *LogInspector {
    return &LogInspector{storage: storage}
}

// HandleSegments returns a list of all log segments (memory and immutable).
func (li *LogInspector) HandleSegments(w http.ResponseWriter, r *http.Request) {
    // TODO 1: Query storage engine for current in-memory segment info
    // TODO 2: Query storage engine for list of immutable segments on disk
    // TODO 3: Combine into response JSON with segment ID, time range, size
    // TODO 4: Write JSON response with appropriate headers
    // Hint: Add a method to LogStorageEngine to expose segment metadata
}

// HandleSearchIndex allows searching the inverted index directly for debugging.
func (li *LogInspector) HandleSearchIndex(w http.ResponseWriter, r *http.Request) {
    term := r.URL.Query().Get("term")
    if term == "" {
        http.Error(w, "term query parameter required", http.StatusBadRequest)
        return
    }

    // TODO 1: Access the full-text index from LogStorageEngine
    // TODO 2: Look up the term in the index
    // TODO 3: Return list of log IDs or segment references containing the term
    // TODO 4: Format as JSON array
}

// HandleWALStatus returns WAL statistics (file size, last offset, error count).
func (li *LogInspector) HandleWALStatus(w http.ResponseWriter, r *http.Request) {
```

```
    // TODO 1: Get WAL instance from LogStorageEngine
    // TODO 2: Get current file size and last write offset
    // TODO 3: Check for any corruption markers or errors
    // TODO 4: Return JSON with health status
}
```

File: `cmd/debug-tools/inspect-logs.go`

```go
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "time"

    "yourproject/internal/config"
)

func main() {
    // Simple CLI tool to query debug endpoints
    baseURL := "http://localhost:6066/debug" // debug port

    client := &http.Client{Timeout: 10 * time.Second}

    // Example: Fetch segment list
    resp, err := client.Get(baseURL + "/inspect/logs/segments")
    if err != nil {
        log.Fatal(err)
    }
    defer resp.Body.Close()

    var segments []interface{}
    if err := json.NewDecoder(resp.Body).Decode(&segments); err != nil {
        log.Fatal(err)
    }

    fmt.Printf("Found %d log segments:\n", len(segments))
    for _, seg := range segments {
        fmt.Printf("  %+v\n", seg)
    }
}
```

E. Language-Specific Hints

1. **Go pprof integration**: Import `_ "net/http/pprof"` and start an HTTP server. Use `go tool pprof -http=:8080 http://localhost:6060/debug/pprof/heap` to visualize heap in browser.
2. **Structured logging with slog**: Create a global logger with correlation ID support:

```go
import "log/slog"


type ctxKey string

const correlationIDKey ctxKey = "correlation_id"


func LoggerWithCorrelation(ctx context.Context) *slog.Logger {

    if corrID, ok := ctx.Value(correlationIDKey).(string); ok {

        return slog.Default().With("correlation_id", corrID)

    }

    return slog.Default()

}
```

3. **Metrics with Prometheus**: Use `prometheus.NewGaugeVec`, `NewCounterVec`, `NewHistogramVec`. Register with `prometheus.MustRegister`. Expose via `promhttp.Handler()`.
4. **Debug endpoints security**: Only enable debug endpoints on localhost or protected internal network. Consider adding simple authentication token for production debugging.

F. Debugging Tips for Common Scenarios

Symptom	How to Diagnose	Quick Fix
Goroutine leak	Run `go tool pprof http://localhost:6060/debug/pprof/goroutine?debug=2`. Look for goroutines stuck in channel operations.	Ensure channels are properly closed and not left waiting indefinitely.
Memory not freed	Compare heap profiles before and after GC (force GC with `debug.FreeOSMemory()`). Look for objects held by global caches.	Clear caches periodically or use weak references.
Query slow after many segments	Count segments in log storage; if >100, compaction may be lagging.	Increase compaction frequency or reduce segment size.
High CPU during ingestion	CPU profile will show hot function (often compression or serialization).	Consider batching compression operations or using faster algorithm.
Alert rules evaluating slowly	Log evaluation time per rule; identify rules with complex cross-signal queries.	Add query timeout per rule evaluation; optimize expensive queries.

G. Milestone Debugging Checkpoints

After implementing each milestone, verify these debugging capabilities work:

- **Milestone 2**: Start server with pprof, ingest sample data, verify metrics endpoint shows channel lengths changing.
- **Milestone 3**: Use debug tools to list log segments, metric series, and trace blocks after ingestion.
- **Milestone 4**: Execute a query and check debug logs for query plan; verify inspection endpoint shows index lookups.
- **Milestone 5**: Trigger an alert, check notification logs; use debug endpoint to view alert state transitions.

---

12. Glossary

> **Milestone(s):** All Milestones (reference)

This glossary defines key terms and concepts used throughout this design document. It serves as a quick reference for understanding the specialized vocabulary of observability and the specific architectural choices made in this platform.

Terms and Definitions

The following table lists terms alphabetically with their definitions and the primary section where the concept is introduced or explained in depth.

Term	Definition	Section Reference
Alert Deduplication	The process of identifying and suppressing multiple notifications for the same underlying issue, typically by computing a fingerprint from alert labels to group identical alert instances.	6. Component: Alerting & Anomaly Detection Engine
Alert Fatigue	The phenomenon where teams become desensitized or overwhelmed due to excessive, duplicate, or irrelevant alert notifications, reducing the effectiveness of the alerting system.	6. Component: Alerting & Anomaly Detection Engine
Alert Instance	A specific occurrence of an alert rule that has transitioned to the `Firing` state, with a unique set of label values identifying the particular condition (e.g., `{service="api", instance="host-1"}`).	6. Component: Alerting & Anomaly Detection Engine
Anomaly Detection	The process of identifying patterns in data that deviate significantly from an established baseline or expected behavior, often using statistical methods like z-scores or machine learning.	6. Component: Alerting & Anomaly Detection Engine
Backpressure	A feedback mechanism in a data pipeline where a downstream component that is overwhelmed signals upstream producers to slow down or pause data transmission, preventing system overload and data loss.	5. Component: Ingestion Pipeline
Batch	A collection of telemetry data items (logs, metrics, or traces) grouped together for efficient processing, transmission, or storage. Batches typically share common resource attributes.	5. Component: Ingestion Pipeline
Cardinality	In the context of metrics, the number of unique time series defined by the combination of a metric name and all its label (attribute) value pairs. High cardinality can strain storage and query performance.	4. Data Model
Circuit Breaker	A design pattern that detects failures and prevents cascading issues by failing fast when a downstream service or component is unhealthy, allowing it time to recover.	8. Error Handling and Edge Cases
Correlation ID	A unique identifier (specifically `trace_id`) that links related telemetry data across different signals, enabling the reconstruction of a request's journey through a distributed system.	4. Data Model
Cross-Signal Correlation Index	A secondary index that maps a `trace_id` to the storage locations (references) of related log records and metric exemplars, enabling efficient joins across different telemetry stores.	6. Component: Storage Engines
Cross-Signal Query	A query that joins or correlates data across different telemetry signals (logs, metrics, and traces) in a single request, such as "find all logs for traces with errors."	5. Component: Query Engine
Cursor-Based Pagination	A pagination method where a client uses an opaque token (the cursor) to resume a query from a specific point, rather than using numeric offsets. This is more stable for large, changing datasets.	5. Component: Query Engine
Dead Letter Queue (DLQ)	A persistent storage area for messages, records, or requests that cannot be processed normally due to errors (e.g., malformed data, transient downstream failure). Allows for inspection and manual reprocessing.	8. Error Handling and Edge Cases
Diagnostic Control Room	A mental model for debugging complex systems, imagining a control room with multiple consoles (logs, metrics, traces) that an engineer must correlate to diagnose issues.	10. Debugging Guide
Downsampling	The process of aggregating high-resolution time-series data into lower-resolution aggregates (e.g., converting 1-second samples to 1-minute averages) to reduce storage costs for long-term retention.	6. Component: Storage Engines
Exemplar	A sample measurement (e.g., a specific trace) linked to a metric data point. It provides a direct link from an aggregated metric (like a high latency histogram bucket) to a representative individual trace that caused that value.	4. Data Model
Exponential Backoff	A retry strategy where the wait time between retry attempts increases exponentially (e.g., 1s, 2s, 4s, 8s). Used to avoid overwhelming a recovering service.	8. Error Handling and Edge Cases
Filter Pushdown	A query optimization technique where filter predicates are evaluated as early as possible, ideally within the storage engine itself, to reduce the amount of data that must be transferred and processed.	5. Component: Query Engine
Fingerprint	In alerting, a hash (or other unique identifier) computed from the set of label key-value pairs of an alert instance, used for deduplication and grouping.	6. Component: Alerting & Anomaly Detection Engine
For Duration	In alerting, the length of time a rule condition must be continuously met before an alert transitions from `Pending` to `Firing`. Helps prevent flapping alerts from transient spikes.	6. Component: Alerting & Anomaly Detection Engine
Full-Text Search	A search technique that allows querying every word within a body of text (like a log message), typically powered by an inverted index mapping terms to the documents that contain them.	6. Component: Storage Engines
Golden File	A test artifact that contains the expected output for a given test input. The actual output is compared against the golden file to detect regressions. Useful for complex, deterministic outputs.	9. Testing Strategy
Gorilla Compression	A lossless compression algorithm for time-series data developed by Facebook. It uses delta-of-delta encoding for timestamps and XOR encoding for floating-point values, achieving high compression ratios.	6. Component: Storage Engines
Graceful Degradation	The property of a system to fail in predictable, safe ways that preserve core functionality. For example, a storage engine entering read-only mode when disk space is exhausted.	8. Error Handling and Edge Cases

Term	Definition	Section Reference
Inverted Index	A data structure that maps terms (like words from log messages) to the list of documents (log records) where they appear. It is the core of full-text search engines.	6. Component: Storage Engines
Log Severity	A classification of the importance or urgency of a log record. Common levels include `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL`.	4. Data Model
Meta-Observability	The observability of the observability platform itself—using its own capabilities (logs, metrics, traces) to monitor, debug, and understand its internal operations.	10. Debugging Guide
Notification Channel	A configured endpoint for sending alert notifications, such as a Slack webhook, PagerDuty integration, or email address.	6. Component: Alerting & Anomaly Detection Engine
OpenTelemetry Protocol (OTLP)	The vendor-agnostic protocol defined by the OpenTelemetry project for transmitting telemetry data (logs, metrics, traces). It is typically carried over gRPC or HTTP with protobuf encoding.	5. Component: Ingestion Pipeline
OpenTelemetry Semantic Conventions	Standardized attribute names and values for observability data (e.g., `http.method`, `db.system`). They ensure consistency and interoperability across different instrumentation sources.	4. Data Model
OTLP/gRPC	The combination of the OTLP protocol with gRPC as the transport layer. Provides high performance, binary encoding, and bidirectional streaming capabilities. The primary ingestion protocol for this platform.	5. Component: Ingestion Pipeline
Pipeline Pattern	A design pattern where data flows through a series of processing stages (like a factory assembly line), with each stage performing a specific transformation or action on the data.	5. Component: Ingestion Pipeline
Plan Node	A single operation (e.g., `ScanLogsNode`, `FilterNode`, `CorrelationJoinNode`) within a query execution plan tree. The tree defines the sequence and method of data retrieval and processing.	5. Component: Query Engine
Polyglot Persistence	An architectural pattern that uses multiple, specialized storage systems within a single application, each optimized for a specific data type and its access patterns (e.g., one store for logs, another for metrics).	6. Component: Storage Engines
Property-Based Testing	An automated testing methodology that verifies properties or invariants of code by generating many random inputs, rather than relying on a fixed set of example-based tests.	9. Testing Strategy
Pull-Based Evaluation	An alert evaluation approach where the alerting engine periodically queries the stored data (via the Query Engine) to check if rule conditions are met. Contrast with push-based evaluation from streaming data.	6. Component: Alerting & Anomaly Detection Engine
Query Engine	The component that provides a unified interface to query logs, metrics, and traces. It accepts a query request, constructs an execution plan, coordinates with storage engines, and returns paginated results.	5. Component: Query Engine
Query Execution Planner	The subcomponent of the Query Engine responsible for transforming an abstract query request into an efficient, concrete execution plan (a tree of `PlanNode`s).	5. Component: Query Engine
Read-Only Fallback	A degraded operating mode for a storage engine where it continues to serve read (query) requests but rejects all write (ingestion) operations. Used to prevent data corruption when underlying problems are detected.	8. Error Handling and Edge Cases
Repeat Interval	The minimum amount of time that must pass before a firing alert instance can send another notification to the same channels. Helps reduce noise during ongoing incidents.	6. Component: Alerting & Anomaly Detection Engine
Request-Scoped Logger	A logger instance that includes a correlation ID (typically from the context) in every log message it produces, making it easy to trace all logs related to a specific request.	10. Debugging Guide
Resource Attributes	A set of key-value pairs that describe the immutable source of telemetry data, such as the `service.name`, `service.version`, and `deployment.environment`. They are attached to all signals (logs, metrics, traces) from that source.	4. Data Model
Resource Model	The schema and semantics defining the identity of the source (e.g., a service instance) that produces telemetry. It is the foundation for grouping and filtering data.	4. Data Model
Rolling Mean/StdDev	A statistical model that calculates the mean and standard deviation over a sliding time window (e.g., the last 24 hours). Used as a baseline for simple anomaly detection.	6. Component: Alerting & Anomaly Detection Engine
Segment-Based Architecture	A storage design pattern that uses immutable, sorted files (called segments) as the primary unit of on-disk data. New data is written to an active in-memory segment and periodically flushed to a new immutable segment on disk.	6. Component: Storage Engines
Span	A named, timed operation representing a unit of work within a trace. It contains a `span_id`, `trace_id`, `parent_span_id`, start/end timestamps, and attributes.	4. Data Model
State Machine	A computational model with a finite number of states, where transitions between states are triggered by specific events. Used to model the lifecycle of alerts and other complex processes.	6. Component: Alerting & Anomaly Detection Engine

Term	Definition	Section Reference
Telemetry Refinery	The mental model for the overall platform architecture: raw telemetry is ingested (crude oil), processed and indexed (refined), and stored in specialized silos (storage tanks) for efficient querying (distribution).	3. High-Level Architecture
Testing Pyramid	A strategy for structuring automated tests, advocating for a large base of fast, isolated unit tests; a smaller middle layer of integration tests; and an even smaller top layer of slow, broad end-to-end tests.	9. Testing Strategy
Time-Series Database	A database system optimized for storing and querying data points indexed by time. Key features include efficient compression, downsampling, and aggregation over time windows.	6. Component: Storage Engines
Trace	A collection of spans that represent the entire path of a request as it travels through a distributed system. All spans in a trace share the same `trace_id`.	4. Data Model
Trace-Inception	The practice of using the observability platform to debug itself by instrumenting its internal components and generating traces for its own operations.	10. Debugging Guide
Trace-Log Correlation	The mechanism that links log records to traces by including the `trace_id` and `span_id` in log records, allowing logs to be found by querying for a specific trace.	4. Data Model
Trace Tree Reconstruction	The process of building a hierarchical tree of spans from individual span records by following `parent_span_id` pointers, enabling visualization and analysis of the complete request flow.	6. Component: Storage Engines
Unified Data Model	The core schema that defines common fields and relationships across logs, metrics, and traces, enabling correlation and unified querying. Centered around shared resource attributes and correlation IDs.	4. Data Model
Write-Ahead Log (WAL)	A durability mechanism where changes (intent to write) are recorded to a sequential log file and fsynced to disk *before* they are applied to the main data structures. This allows recovery after a crash.	6. Component: Storage Engines
Z-Score	A statistical measure expressing how many standard deviations a data point is from the mean of a distribution. Used in anomaly detection to flag values that fall outside an expected range (e.g.,	z