APM Tracing System: Design Document

Overview

We are building an Application Performance Monitoring (APM) system that tracks requests as they flow through a distributed application, creating detailed traces. The key architectural challenge is efficiently collecting, storing, and analyzing massive volumes of trace data while preserving the most valuable insights for debugging and performance optimization.

> 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.

1. Context and Problem Statement

> **Milestone(s):** This section provides the foundational context for all subsequent milestones, explaining the core problems that the entire APM Tracing System is designed to solve.

Imagine you're tasked with investigating a performance problem in a modern application. A user reports that a specific action—say, "checkout" in an e-commerce app—is taking 15 seconds to complete, far longer than the expected 2 seconds. The application isn't a single monolithic program but a constellation of dozens, sometimes hundreds, of microservices. The request from the user's browser might touch a frontend API gateway, which calls a user service for authentication, which then calls a product catalog service, which queries a database, then calls an inventory service, which publishes an event to a message queue, which is consumed by a shipping service, and so on. Each of these steps could be running on different machines, in different data centers, managed by different teams.

Without a specialized observability tool, diagnosing this slowdown is like trying to solve a murder mystery where each witness (service) only saw a tiny fragment of the event and speaks a different language. You might find a log in the API gateway showing it took 14 seconds to get a response from the user service. But why? Is the user service slow, or is it waiting on something else? Is the database overloaded? Is there a network partition? Traditional logging and metrics give you isolated data points, but they don't show you the complete journey of the request as it traverses this complex distributed system. This is the fundamental problem that distributed tracing, and by extension an Application Performance Monitoring (APM) system, exists to solve.

Mental Model: The Distributed Detective

Think of a **distributed trace** as the complete **case file** for a single request's journey through your system. Every time a request enters your application, a new case is opened. As the request moves from service to service, each service creates a **span**—a structured log entry that represents a single unit of work (like a database query or an HTTP call) within that service's handling of the request. A span contains crucial evidence: what operation was performed, when it started and ended, what service executed it, and whether it succeeded or failed.

Most importantly, spans are linked. Each span knows the ID of the overarching **trace** (the case file) and the ID of its **parent span** (the previous step in the journey). This creates a hierarchical timeline—a **trace tree**—that reconstructs the exact path and timing of the request.

Now, picture the APM system as the **distributed detective**. Its job is to:

1. **Collect** all the span evidence from every service (the witnesses).
2. **Assemble** the spans into their correct trace trees, linking parent to child across service boundaries.
3. **Analyze** the assembled traces to answer critical questions: Where was the time spent? Which service was the bottleneck? Did an error in one service cascade to others?

4. **Visualize** the results, showing the detective (the engineer) not just a single request's timeline, but also the bigger picture: a map of how all services interact, statistical summaries of performance, and alerts when things deviate from the norm.

This mental model shifts debugging from sifting through disjointed logs to reading a coherent, end-to-end story of a request's life. It turns the opaque distributed system into a transparent, understandable entity.

The Observability Challenge in Microservices

Microservices architecture delivers benefits in scalability, developer autonomy, and deployment velocity, but it fundamentally changes the nature of debugging and performance analysis. The core challenges can be distilled into three concrete problems:

1. **The End-to-End Latency Puzzle:** When a user-facing operation is slow, the root cause can be anywhere. A latency metric on your API gateway tells you the *symptom* (total time = 15s), but not the *cause*. Without tracing, you must manually hop from service to service, checking their individual metrics and logs, trying to correlate timestamps—a process that is slow, error-prone, and often impossible when calls are asynchronous or batched.
2. **The Cascading Failure Mystery:** A failure in one service can propagate in non-obvious ways. For example, if a payment service times out, the order service might retry three times, queueing requests and consuming threads, which then causes the order service itself to become slow and start timing out requests to the inventory service. Looking at the inventory service's logs, you might see a surge in timeouts and conclude it's the problem, missing the true root cause two hops upstream. Understanding these failure propagation paths requires seeing the call graph.
3. **The Dependency Map Blind Spot:** As systems grow organically, implicit dependencies emerge. Service A might start calling Service C indirectly through Service B. Old dependencies might linger long after they're no longer used. Without a dynamic, data-driven view of service communications, system architecture diagrams quickly become outdated. This makes impact analysis for deployments, capacity planning, and failure domain isolation guesswork.

The following table summarizes the shift in observability requirements from monolithic to microservices architectures:

Observability Aspect	Monolithic Application	Microservices Architecture	New Challenge
Latency Attribution	Profiling within a single process pinpoints slow functions.	Latency is distributed across network calls and process boundaries.	Need to attribute time to specific remote calls and understand serial vs. parallel execution.
Error Propagation	Stack trace localizes error to a module/function.	Error in one service may manifest as a timeout or degraded behavior in another.	Need to reconstruct the call chain to find the origin of a fault.
System Topology	Static, defined by code structure.	Dynamic, defined by runtime service discovery and communication.	Need to discover and visualize the runtime dependency graph.
Debugging Data	Centralized logs and metrics from one application.	Data is siloed across many independent services and teams.	Need to correlate events across service boundaries with a common request identifier.

These challenges are not merely incremental; they represent a qualitative change in complexity. Traditional monitoring tools, designed for monoliths or simple client-server models, are ill-equipped to provide the necessary cross-boundary visibility.

Existing Approaches and Trade-offs

Before committing to building a dedicated tracing system, it's essential to understand what existing observability tools can and cannot do. Most teams already use some combination of **logs**, **metrics**, and **health checks**. Each plays a role, but each has significant gaps when it comes to understanding distributed request flow.

Decision: Build a Dedicated Distributed Tracing System

- **Context**: We need to understand request flow across microservices for debugging and performance optimization. Existing logging and metrics systems are insufficient for this task.
- **Options Considered**:
  1. **Enhanced Logging**: Inject a common `trace_id` into every log line across all services and use a log aggregator (e.g., ELK Stack) to correlate them.
  2. **Enhanced Metrics**: Instrument each service to emit fine-grained timing histograms for all outgoing calls and try to infer relationships from dashboards.
  3. **Dedicated Distributed Tracing**: Implement the OpenTelemetry tracing standard, collect spans, and build a system specifically for assembling and analyzing traces.
- **Decision**: Option 3 — Build a dedicated distributed tracing system.
- **Rationale**: While logging with `trace_id` provides some correlation, it lacks the structured hierarchy and timing relationships inherent to spans. Logs are also unstructured and expensive to query for complex path analysis. Metrics can show aggregates but lose individual request context, making it impossible to debug specific user issues. A dedicated tracing system is built around the first-class concepts of traces and spans, enabling efficient storage, precise reconstruction of call graphs, and specialized analytics (like service maps and tail-based sampling) that are impractical with generic tools.
- **Consequences**: We must build and operate a new subsystem, but we gain unparalleled visibility into request flow, enabling faster debugging, performance optimization, and system understanding. The system will complement, not replace, existing logs and metrics.

The table below details the trade-offs between these three pillars of observability, highlighting why tracing is a necessary addition:

Approach	How It Works	Strengths	Weaknesses for Distributed Debugging	Role in Our APM System
Logging	Applications emit timestamped text lines describing events. Central aggregator indexes and enables search.	**High detail**: Can capture any arbitrary event or state. **Flexible**: No predefined schema. **Ubiquitous**: Every application does it.	**No structure**: Hard to extract consistent fields (e.g., duration, parent ID). **Expensive correlation**: Finding all logs for one request requires scanning millions of lines. **No hierarchy**: Can't easily visualize parent-child relationships or parallel operations.	Logs remain crucial for deep, unstructured debugging *within* a service. The APM system will **correlate** trace IDs with log entries, allowing engineers to jump from a slow span to the relevant service logs.
Metrics	Services emit numerical measurements (counters, gauges, histograms) at regular intervals. Time-series database stores and dashboards visualize.	**Aggregation**: Excellent for trends, alerts, and SLOs. **Low overhead**: Sampled and pre-aggregated. **High cardinality**: Modern systems can handle many dimensions.	**Loss of context**: You see *that* the 95th percentile latency for Service A spiked, but not *which specific requests* caused it or *why*. **No causal links**: Cannot show that a latency spike in Service B caused an error rate increase in Service A.	Metrics provide the **health dashboard** and **alerting** backbone. The APM system will **generate metrics** from trace data (e.g., request rate, error rate, latency percentiles per service/operation) and may trigger alerts based on trace-derived anomalies.
Distributed Tracing	Services emit structured **spans** for units of work, linked by trace and parent IDs. Collector assembles spans into trace trees.	**Request-centric view**: See the complete story of a single user request. **Causal links**: Clear parent-child relationships across services. **Timing context**: Understand serialization, parallelization, and waiting time.	**High volume**: Can generate massive amounts of data. Requires **sampling**. **Instrumentation cost**: Requires code changes or auto-instrumentation. **Complex backend**: Needs specialized storage and query.	This is the **core** of our APM system. It provides the connective tissue between logs and metrics, answering the "why" and "how" behind the "what" shown by metrics and logs.

The synergy between these three signals is often described as the **Observability Golden Triangle**. An engineer might:

1. See a **metric** alert for high p95 latency on the checkout endpoint.
2. Use the **tracing** system to sample recent slow traces for that endpoint, instantly visualizing that the delay is in the payment service, specifically in a call to a third-party card processor.
3. Click from the slow payment span in the trace view to search **logs** from the payment service filtered by that specific `trace_id`, revealing an error message from the third-party API about an expired certificate.

This integrated workflow reduces mean time to resolution (MTTR) from hours or days to minutes. Our APM Tracing System aims to be the central platform that makes this workflow not just possible, but efficient and scalable.

Implementation Guidance

As this is the foundational context section, there is no direct implementation to be done. However, the concepts introduced here directly inform the architecture and design decisions in all subsequent sections. The following table maps the high-level observability concepts to the concrete components you will build in the upcoming milestones.

Conceptual Need	APM System Component (Milestone)	What You'll Build
Collect evidence (spans) from all services	Trace Collector (Milestone 1)	HTTP/gRPC servers to receive spans, buffering logic, storage layer.
Assemble spans into trace trees	Trace Collector & Storage (Milestone 1)	Indexing by `trace_id`, logic to handle out-of-order span arrival.
Understand service relationships	Service Map (Milestone 2)	Graph construction algorithms that analyze spans to build a dependency map.
Manage data volume	Trace Sampling (Milestone 3)	Head-based and tail-based sampling algorithms to filter traces intelligently.
Analyze performance trends	Performance Analytics (Milestone 4)	Percentile calculators (t-digest) and anomaly detectors for time-series data.
Generate spans automatically	APM SDK (Milestone 5)	Auto-instrumentation libraries for HTTP, databases, and frameworks.

Before starting implementation, ensure you are comfortable with the core data model that will be used throughout the system, which is defined in the **Data Model** section (Section 4). The key entities are:

- **Trace**: The entire record of a request's journey, comprised of all its spans.
- **Span**: A named, timed operation representing a unit of work within a trace.
- **Service**: A logical component of the application (e.g., `checkout-service`, `user-db`).

Proceed to the **High-Level Architecture** section (Section 3) to see how these components fit together into a cohesive system.

> **Milestone(s):** This section provides the foundational scope definition that applies to ALL subsequent milestones, setting clear boundaries for what the APM Tracing System will and will not address.

2. Goals and Non-Goals

Establishing clear boundaries is critical for the success of any complex system. Without explicit goals, scope creep becomes inevitable, leading to an unfocused product that attempts to solve every observability problem but masters none. Conversely, without declared non-goals, stakeholders may assume capabilities the system was never designed to provide, resulting in disappointment and misuse.

This section explicitly defines **what the APM Tracing System must deliver** and, equally important, **what it explicitly will not attempt**. These boundaries serve as guardrails for architectural decisions and prioritization throughout the project.

Goals (What We Must Do)

The primary goal of this system is to provide **end-to-end visibility into request flow across a distributed application**, enabling developers to debug performance issues, understand service dependencies, and identify anomalies. The system must achieve this while operating efficiently at scale, balancing data volume with insight quality.

The following table enumerates the mandatory functional requirements, each mapped to its corresponding project milestone:

Goal ID	Functional Requirement	Description	Corresponding Milestone
G1	**Collect and Store Distributed Traces**	Ingest spans in OpenTelemetry format via HTTP/gRPC, correctly assemble them into traces using parent-child relationships, and persistently store them with efficient indexing for retrieval by trace ID. The system must handle at least 1000 spans per second without data loss.	Milestone 1: Trace Collection
G2	**Generate and Visualize a Dynamic Service Dependency Map**	Automatically derive a directed graph of service-to-service communication from trace data. Compute and display key metrics (request count, latency, error rate) for each edge. Update the map in near-real-time as new trace data arrives.	Milestone 2: Service Map
G3	**Implement Intelligent, Adaptive Trace Sampling**	Reduce storage and processing costs by filtering traces while preserving diagnostically valuable data. Support both head-based (probabilistic at trace start) and tail-based (decision after trace completion based on errors/latency) sampling strategies. Allow per-service configuration.	Milestone 3: Trace Sampling
G4	**Provide Performance Analytics and Anomaly Detection**	Calculate latency percentiles (p50, p95, p99) per service and operation. Establish historical baselines and detect deviations using statistical methods. Generate alerts for performance regressions.	Milestone 4: Performance Analytics & Anomaly Detection
G5	**Offer Auto-Instrumentation via an APM SDK**	Provide client libraries that automatically instrument common frameworks (HTTP servers/clients, database drivers) to generate spans and propagate trace context without requiring manual code changes. Support W3C Trace Context standard.	Milestone 5: APM SDK & Auto-Instrumentation
G6	**Ensure High Performance and Low Overhead**	The data collection path (from SDK through Collector to storage) must add minimal latency to instrumented applications. Ingestion acknowledgment should occur within 100ms. The SDK's instrumentation overhead must be negligible for production workloads.	Cross-cutting (All Milestones)
G7	**Design for Horizontal Scalability and Resilience**	The core components (Collector, storage, query services) must be deployable as multiple instances to handle load increases. The system should gracefully degrade (e.g., via increased sampling) under extreme load rather than fail catastrophically.	Cross-cutting (All Milestones)

> **Key Design Insight:** The goals are intentionally ordered to reflect the **data pipeline's natural flow**. You must first collect data (G1) before you can analyze it to build a service map (G2). You need the raw data volume (G1) before you need to manage it via sampling (G3). The SDK (G5) is the source of all data, but its design is influenced by the needs of the entire pipeline.

Beyond these functional goals, the system adheres to several key architectural principles:

- **Open Standards First:** Prioritize compatibility with the OpenTelemetry specification for data format and context propagation (`TRACE_ID_HEADER: traceparent` , `SPAN_ID_HEADER: tracestate`). This ensures interoperability with a broader ecosystem of tools.
- **Observability of the Observability System:** The system itself must be monitorable, providing its own metrics (e.g., ingestion rate, sampling decisions, storage health) and logs.
- **Operator-Friendly Configuration:** Sampling rates, storage backends, and aggregation intervals must be configurable without requiring code deploys, ideally via runtime configuration or feature flags.

Non-Goals (What We Explicitly Won't Do)

A system's strength often lies in what it chooses *not* to do. The following are explicitly out of scope for this version of the APM Tracing System. This clarity prevents misallocation of engineering effort and sets correct expectations for users.

Non-Goal ID	Out-of-Scope Capability	Rationale & Guidance
NG1	**Real User Monitoring (RUM) / Browser Tracing**	We will not instrument client-side web applications to capture browser performance metrics, page load events, or user interaction timings. This domain requires handling of synthetic versus real user data, geolocation, and device fragmentation. **Consideration:** Traces will start at the first instrumented backend service.
NG2	**Log Aggregation and Centralized Logging**	While `Span.Events` can capture log-like data, we will not build a general-purpose log ingestion, indexing, and search system (e.g., like ELK Stack or Loki). Traces are for structured performance data; logs are for unstructured diagnostic text. **Consideration:** Use a dedicated logging system and correlate logs to traces via `TraceID`.
NG3	**Infrastructure and Host-Level Metrics**	We will not collect or alert on system metrics like CPU usage, memory consumption, disk I/O, or network throughput. This is the domain of traditional monitoring systems (e.g., Prometheus). **Consideration:** Our performance analytics (G4) focus on *application* metrics (latency, error rate) derived from traces.
NG4	**Continuous Profiling (CPU, Heap, etc.)**	We will not periodically capture and analyze CPU flame graphs or heap allocations across services. Profiling provides a different, deeper level of insight into *why* code is slow, not just *where*. **Consideration:** This could be a valuable future extension (see Future Extensions) but is a separate complex system.
NG5	**Synthetic Monitoring / Active Health Checks**	We will not proactively simulate user traffic from various global locations to measure availability and performance from an external perspective. **Consideration:** Our system observes real production traffic. Synthetic monitoring is a complementary practice.
NG6	**Advanced AI/ML for Root Cause Analysis**	While we perform basic statistical anomaly detection (G4), we will not implement machine learning models that automatically pinpoint the root cause of an incident or predict future failures. **Consideration:** We provide the high-quality trace data upon which such advanced systems could be built.
NG7	**Long-Term, Cold Storage Archival**	Our trace storage is optimized for online querying and analysis over a recent time window (e.g., 7-30 days). We will not build pipelines to compress and archive trace data to object storage (e.g., S3) for indefinite retention and historical forensics. **Consideration:** Implement a separate archival process if needed, using our storage's export capabilities.
NG8	**Full-Text Search Across All Span Attributes**	While we index traces by `TraceID`, service, and time range, we will not support arbitrary, high-cardinality full-text search over every key-value pair in `Span.Attributes`. This would require a different indexing technology (e.g., Elasticsearch). **Consideration:** Queries should be scoped to known services, operations, or specific trace IDs.

> **Architectural Decision: Focusing on Core Tracing**
>
> **Context:** Observability platforms can encompass logging, metrics, tracing, profiling, and more. Our team has limited resources and must deliver a focused, high-quality product.
>
> **Options Considered:**
>
> 1. **Build a "Full-Stack" Observability Platform:** Incorporate RUM, logging, and infrastructure metrics alongside tracing.
> 2. **Build a Best-in-Class Distributed Tracing System:** Focus exclusively on the trace data pipeline and its immediate derivatives (service maps, performance analytics).
> 3. **Build a Trace-Centric Platform with Extension Points:** Focus on tracing but design explicit integration points for external logging and metrics systems.
>
> **Decision:** Choose Option 2, with design principles aligned to Option 3.
>
> **Rationale:**
>
> - **Focus & Quality:** Mastering the complexities of distributed trace collection, sampling, and analysis is a significant challenge. Splitting effort across multiple signal types risks delivering a mediocre product in all areas.
> - **Ecosystem Maturity:** The OpenTelemetry project provides robust standards for traces. By adhering to it, we enable users to pair our tracing system with their preferred best-in-class solutions for logs (e.g., Loki) and metrics (e.g., Prometheus).
> - **Correlation over Unification:** The highest value comes from correlating *across* these signals (e.g., jumping from a high-latency trace to its corresponding logs). We achieve this by emitting standard `TraceID` s, not by building all the systems ourselves.
>
> **Consequences:**
>
> - **Positive:** We can build a deep, scalable, and feature-rich tracing system.
> - **Positive:** Users are not locked into a monolithic vendor; they can choose best-of-breed for each observability pillar.
> - **Negative:** Users must operate and integrate multiple systems for a complete picture.
> - **Mitigation:** We will provide clear documentation on correlating traces with logs and metrics from common external systems.

Scope Boundary Visualization

The following mental model helps visualize the system's scope within the broader observability landscape:

> **The Observability "House":** Imagine observability as a house with three main pillars holding up the roof (which represents understanding your system).
>
> - **Metrics Pillar (Our Non-Goal NG3):** Answers "How many?" and "How much?" – throughput, error counts, resource utilization. Built by systems like Prometheus.
> - **Logs Pillar (Our Non-Goal NG2):** Answers "What happened?" – discrete events with detailed context. Built by systems like Elasticsearch/Loki.
> - **Traces Pillar (Our Core System):** Answers "How long?" and "What path?" – the journey of a request, with timing and causality.
>
> Our APM system builds and owns the **Traces Pillar**. The roof (full understanding) is supported by all three pillars working together. We design our pillar with strong, standard connections (like `TraceID`) so it can be securely joined with the others, but we do not construct the entire house ourselves.

By adhering to these Goals and Non-Goals, the project maintains a clear, achievable vision. The subsequent architectural and component designs will be evaluated against this framework to ensure consistency and focus.

Implementation Guidance

While this section is primarily declarative (defining *what*), the act of maintaining a clear project scope has implementation implications, particularly for how components are designed to be extensible yet bounded.

A. Technology Recommendations Table:

Component	Simple/Initial Option	Advanced/Scalable Option
Scope Enforcement & Feature Flags	Hard-coded configuration in Go `struct`s, compiled-in behavior.	External configuration service (e.g., etcd, Consul) with dynamic reloading and feature flag system (e.g., LaunchDarkly SDK).
Dependency / Boundary Definition	Explicit `internal/` package boundaries in Go to prevent unwanted imports.	API versioning for public endpoints (e.g., `/api/v1/collect`) and clear internal gRPC/protobuf service definitions.

B. Recommended File/Module Structure: To enforce the separation of concerns implied by our non-goals, the project layout should isolate core tracing logic from potential future extensions or integrations with other systems.

```
apm-tracing-system/
├── cmd/                          # Application entry points
│   ├── collector/                # Trace collector service (Milestone 1)
│   ├── query-service/            # Service for querying traces and service maps
│   └── analytics-engine/         # Performance analytics service (Milestone 4)
├── internal/                     # Private application code
│   ├── apm/                      # Core APM domain logic
│   │   ├── trace/                # Trace and Span data models (G1)
│   │   ├── sampling/             # Head & tail sampling logic (G3)
│   │   ├── servicemap/           # Graph construction algorithms (G2)
│   │   └── analytics/            # Percentile & anomaly detection (G4)
│   ├── collector/                # Ingestion API & pipeline (G1, G6)
│   ├── storage/                  # Abstraction for trace storage (G1, G7)
│   │   ├── memory/               # In-memory store (for testing)
│   │   └── cassandra/            # Cassandra-backed store (for production)
│   └── sdk/                      # Go APM SDK implementation (G5)
│       ├── instrumentation/      # Auto-instrumentation wrappers
│       └── propagate/            # Context propagation utilities
├── pkg/                          # Public libraries (if any)
│   └── otelhelpers/              # OpenTelemetry compatibility utilities
└── api/                          # API definitions (protobuf, OpenAPI)
    ├── protos/                   # .proto files for gRPC
    └── openapi/                  # OpenAPI specs for HTTP endpoints
```

C. Core Logic Skeleton Code (Scope Validation Helper): While not a core component, a validation helper at system initialization can guard against accidental scope creep in configuration.

```go
// internal/apm/config/validator.go

// Config holds the runtime configuration for the APM system.
type Config struct {
    MaxIngestionRate int           `yaml:"max_ingestion_rate"`
    Sampling         SamplingConfig `yaml:"sampling"`
    Storage          StorageConfig  `yaml:"storage"`
    // ... other fields
}

// Validate ensures the configuration does not request out-of-scope (non-goal) capabilities.
func (c *Config) Validate() error {
    // G6: Performance guardrail
    if c.MaxIngestionRate > 10000 { // Example limit
        return fmt.Errorf("max_ingestion_rate %d exceeds supported scale for this version", c.MaxIngestionRate)
    }

    // NG3: Ensure we are not mistakenly configured to collect host metrics.
    // (This is a placeholder; the real check would be for absent or nil config sections)
    if c.HostMetrics != nil {
        return fmt.Errorf("host_metrics collection is a non-goal (NG3). Please remove this configuration")
    }

    // Add other validations as needed for non-goals...
    return nil
}
```

D. Language-Specific Hints (Go):

- Use the `internal` directory pattern religiously. Code inside `internal/` cannot be imported by code outside your project repository. This is a powerful mechanism to enforce API boundaries and prevent your packages from being used in ways you didn't intend (a form of scope enforcement).
- For configuration, use a library like `spf13/viper` which supports reading from multiple sources (files, env vars) and allows you to bind your config `struct`. The `Validate()` method can be called after the config is loaded.
- Start every component's `main.go` with a call to `config.Validate()` and exit gracefully if validation fails, ensuring the system never starts in an out-of-scope configuration.

---

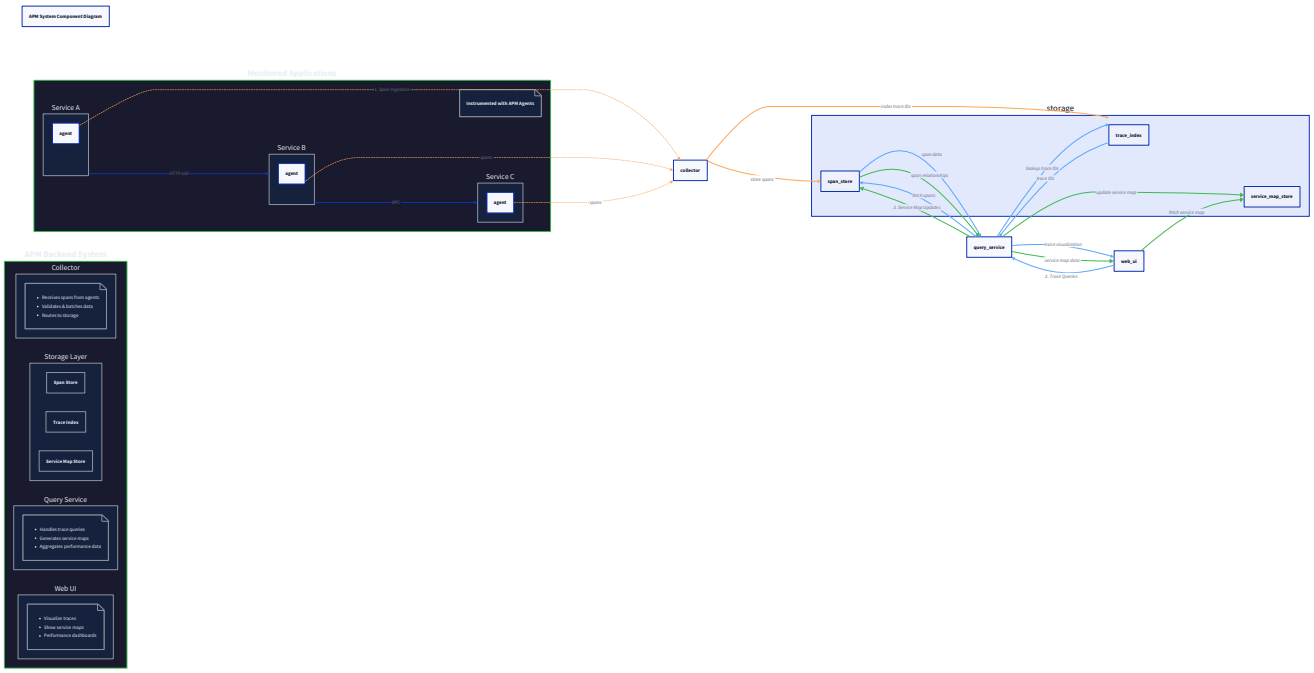
3. High-Level Architecture

> **Milestone(s):** This section provides the architectural foundation for all subsequent milestones, defining the system's components and their relationships that will be implemented across Milestones 1-5.

Think of the APM Tracing System as a **distributed nervous system** for your microservices architecture. Just as the human nervous system has sensory receptors (nerve endings), transmission pathways (nerves), processing centers (brain and spinal cord), and a way to perceive the processed information (conscious awareness), our APM system has components that collect, transmit, store, analyze, and visualize trace data. This architectural overview maps out this "nervous system," showing how telemetry flows from your applications to insights you can act upon.

Component Overview and Responsibilities

The APM Tracing System is composed of five primary components that work together in a data pipeline. Data flows unidirectionally through most of these components (from left to right in the diagram), with some feedback loops for configuration and querying.



Each component has a specific responsibility in the trace lifecycle, following the **single responsibility principle** to ensure maintainability and scalability. The table below details each component's role, what data it owns, and how it interacts with others:

Component	Primary Responsibility	Key Data It Owns/Holds	Interface to Other Components
APM SDK & Agent	**Instrumentation & Span Emission**	Span context (trace ID, span ID, parent span ID) within application memory; Configuration for sampling and reporting	• **To Collector:** Sends spans via HTTP/gRPC using OpenTelemetry format • **To Application:** Injects itself via middleware/monkey-patching to intercept calls
Collector	**Span Ingestion & Initial Processing**	In-memory buffers of recently received spans; Sampling configuration and decision cache	• **From Agent:** Receives spans via ingestion endpoints • **To Storage:** Writes processed spans to persistent storage • **To Query Service:** Notifies of trace completion events
Storage Backend	**Durable Trace Storage & Indexing**	All persisted span data; Secondary indexes on trace ID, service, operation, and timestamp	• **From Collector:** Accepts span writes • **From Query Service:** Serves span reads and index queries • **To Analytics Engine:** Provides historical data for baseline calculation
Query Service	**Trace Retrieval & Service Map Computation**	Materialized service dependency graph (if pre-computed); Query execution plans and results caching	• **From Storage:** Reads spans and indexes • **To Web UI:** Serves trace lists, individual traces, and service map data • **To Collector:** Receives trace completion notifications for tail-based sampling
Web UI	**Visualization & User Interaction**	User session state, visualization preferences, alert configurations	• **To Query Service:** Sends queries for traces and service maps • **To User:** Renders interactive visualizations and dashboards

The **data flow** follows a clear pipeline:

1. **Instrumentation Phase:** Application code, via the APM SDK, generates `Span` objects during execution.
2. **Emission Phase:** The SDK sends these spans to the Collector via network calls.
3. **Ingestion Phase:** The Collector receives spans, validates them, applies sampling decisions, and buffers them for assembly.
4. **Storage Phase:** Assembled spans are written to the Storage Backend with appropriate indexing.
5. **Processing Phase:** The Query Service reads from storage to compute service maps and serve trace queries.
6. **Visualization Phase:** The Web UI queries the Query Service and renders results for users.

There are also two important **cross-cutting concerns** that span multiple components:

- **Configuration Management:** All components read from a centralized configuration service (or files) for settings like sampling rates, storage endpoints, and feature flags.
- **Observability of the Observability System:** The APM system itself generates metrics, logs, and traces about its own operation, which are fed back into itself (dogfooding) or to a separate monitoring system.

> **Key Design Insight:** This architecture employs the **pipes and filters** pattern. Each component transforms or routes the data stream, allowing independent scaling and technology choices per component. The Collector can be scaled horizontally to handle ingestion load, while the Storage Backend and Query Service can be optimized separately for write-heavy versus read-heavy workloads.

Detailed Component Breakdown

APM SDK & Agent This component is actually a **library** embedded within your application processes (the SDK) and potentially a **sidecar/daemon** (the Agent) that manages span batching and export. Its core job is to make tracing **automatic** for developers. It intercepts key operations (HTTP requests, database queries, async tasks), creates `Span` records, propagates the `Trace` context across service boundaries using headers (like `TRACE_ID_HEADER`), and finally exports completed spans to the Collector. It must be extremely lightweight to avoid perturbing the very performance it's measuring.

Collector The Collector is the system's **front door and traffic cop**. It's a stateful service that receives spans from potentially thousands of concurrent application instances. Beyond simple receipt, it must:

- **Validate** incoming spans for format correctness.
- **Apply head-based sampling** immediately to reduce volume.
- **Buffer and reassemble** spans that arrive out-of-order into complete traces.
- **Apply tail-based sampling** decisions once traces are complete.
- **Batch and write** final spans to storage efficiently. Think of it as a package sorting facility: packages (spans) arrive on many conveyor belts, are sorted by destination (trace ID), assembled into complete shipments (traces), and then loaded onto trucks (storage writes).

Storage Backend This is the system's **long-term memory**. It must store massive volumes of span data (each request generates multiple spans) with efficient retrieval patterns. The primary access pattern is **retrieve all spans for a given trace ID**, requiring a primary index on `TraceID`. Secondary access patterns include:

- Find traces for a specific service/operation within a time range.
- Retrieve raw span data for service map computation (which could also be done via materialized views).
- Access historical data for performance baselines and trend analysis. The storage system must balance cost, query performance, and write throughput, often leading to a multi-tiered storage strategy (hot recent data in one system, colder data in another).

Query Service This component is the **analytic brain**. It translates user queries (e.g., "show traces for service 'checkout' with errors in the last hour") into efficient storage queries, computes aggregations (like the service dependency graph), and formats results for the UI. For performance, it may maintain **materialized views** (pre-computed service maps) or **caches** of frequent query results. It also houses the **analytics engine** that computes percentiles and runs anomaly detection algorithms on the fly or on scheduled intervals.

Web UI The UI is the **human interface** to the system. It provides:

- A **trace viewer** showing waterfall diagrams of request flow.
- A **service map** visualization with interactive exploration.
- **Dashboards** for key performance indicators (latency, error rates).
- **Alert management** interfaces.
- **Search and query builders**. It's a client-side application that talks to the Query Service's API, and it must handle large datasets efficiently (e.g., virtual scrolling for long trace lists).

Component Interaction Patterns

The components communicate using two primary patterns:

1. **Asynchronous Fire-and-Forget (Spans):** The SDK → Collector → Storage flow is primarily asynchronous. The SDK sends spans and doesn't wait for them to be stored; it may buffer and batch locally. The Collector acknowledges receipt quickly (within 100ms per Milestone 1) but writes to storage asynchronously. This ensures low overhead on the instrumented application.

2. **Synchronous Request-Response (Queries):** The Web UI → Query Service → Storage flow for trace retrieval is synchronous. Users expect to see their query results, so the UI waits for the Query Service to return traces or service map data. These queries can be complex and may require optimization (pagination, time-range restrictions).

Recommended File/Module Structure

A well-organized codebase mirrors the architectural components, making it intuitive for developers to navigate and maintain. For our Go implementation, we follow the **Standard Go Project Layout** conventions with some adaptations for our specific domain.

> **Mental Model: A Library's Organization**
> Think of the project structure like a well-organized library. The `cmd/` directory is the main entrance (front desk). Each major component gets its own **wing** (`internal/collector/` , `internal/query/`). Within each wing, books are grouped by topic: the main logic (`*.go`), its tests (`*_test.go`), configuration (`config/`), and internal helpers (`internal/` within the component). Shared utilities used across wings live in `pkg/` (like the reference section). This organization helps you know exactly where to go to find or modify any piece of functionality.

```
apm-tracing-system/
├── cmd/                          # Application entry points
│   ├── collector/                # Collector service binary
│   │   └── main.go               # Sets up config, starts HTTP/gRPC servers
│   ├── query-service/            # Query service binary
│   │   └── main.go
│   └── web-ui/                   # Web UI server (could be Go serving static files + API proxy)
│       └── main.go
│
├── internal/                     # Private application code (not importable by others)
│   ├── apmsdk/                   # Milestone 5: APM SDK & Auto-Instrumentation
│   │   ├── instrument/           # Auto-instrumentation wrappers
│   │   │   ├── http/             # HTTP client/server instrumentation
│   │   │   ├── database/         # SQL driver wrappers
│   │   │   └── frameworks/       # Gin, Echo, etc. middleware
│   │   ├── tracer/               # Core tracer implementation
│   │   │   ├── tracer.go         # `Tracer` struct and methods
│   │   │   ├── span.go           # `Span` struct methods
│   │   │   └── propagation/      # W3C Trace Context propagation logic
│   │   └── config/               # SDK configuration
│   │
│   ├── collector/                # Milestone 1: Trace Collection
│   │   ├── api/                  # HTTP/gRPC handlers for span ingestion
│   │   │   ├── http_handler.go
│   │   │   └── grpc_handler.go
│   │   ├── ingestion/            # Core ingestion pipeline
│   │   │   ├── pipeline.go       # Main pipeline orchestration
│   │   │   ├── validator.go      # Span validation logic
│   │   │   ├── buffer/           # Buffering strategies for trace assembly
│   │   │   │   ├── memory_buffer.go
│   │   │   │   └── wal_buffer.go  # Write-ahead log backed buffer (ADR)
│   │   │   └── assembler.go      # Trace assembly from spans
│   │   ├── sampling/             # Milestone 3: Sampling (head-based here)
│   │   │   ├── sampler.go        # `Sampler` interface
│   │   │   ├── probabilistic.go  # Probabilistic (head) sampler
│   │   │   └── consistent.go     # Trace-ID consistent hash sampler
│   │   └── storage/              # Client to write to storage backend
│   │       └── writer.go         # Batched span writer
│   │
│   ├── storage/                  # Storage abstraction layer
│   │   ├── driver/               # Storage driver interfaces
│   │   │   ├── writer.go         # `StorageWriter` interface
│   │   │   └── reader.go         # `StorageReader` interface
│   │   ├── indexes/              # Index management logic
│   │   │   ├── trace_index.go    # Primary index on TraceID
│   │   │   └── service_index.go  # Secondary index on ServiceName
│   │   └── implementations/      # Concrete implementations
│   │       ├── memory/           # In-memory store (for testing)
│   │       ├── cassandra/        # Cassandra implementation
│   │       └── elasticsearch/    # Elasticsearch implementation
│   │
│   ├── query/                    # Milestone 2 & 4: Query Service & Analytics
│   │   ├── service/              # gRPC/REST API for queries
│   │   │   └── handler.go        # Request handlers for trace/search
│   │   ├── servicemap/           # Milestone 2: Service Map
│   │   │   ├── builder.go        # Graph construction from spans
│   │   │   ├── graph.go          # `ServiceGraph` data structure
│   │   │   ├── aggregator.go     # Edge metric aggregation
│   │   │   └── materialized/     # Materialized view updater
│   │   ├── analytics/            # Milestone 4: Performance Analytics
│   │   │   ├── percentiles/      # t-digest based percentile calculator
│   │   │   │   ├── aggregator.go
│   │   │   │   └── tdigest.go    # t-digest implementation
│   │   │   ├── anomaly/          # Anomaly detection
│   │   │   │   ├── detector.go   `AnomalyDetector` interface
```

```
│   │   │   │   ├── zscore.go     # Z-score based detector
│   │   │   │   └── baseline.go   # Historical baseline manager
│   │   │   └── timeseries/       # Time-series aggregation
│   │   │       └── aggregator.go
│   │   └── tail_sampling/        # Milestone 3: Tail-based sampling
│   │       └── evaluator.go      # Evaluates complete traces for keeping
│   │
│   ├── webui/                    # Web UI backend (if not purely static)
│   │   ├── static/               # Serves compiled frontend assets
│   │   └── api_proxy.go          # Proxies API calls to query service
│   │
│   └── models/                   # Shared data models (used by multiple components)
│       ├── trace.go              # `Trace` and `Span` struct definitions
│       ├── service.go            # `Service` struct
│       ├── config.go             # `Config`, `SamplingConfig`, `StorageConfig`
│       └── telemetry/            # OpenTelemetry protobuf/generated code
│
├── pkg/                          # Public, reusable libraries
│   ├── sampling/                 # Sampling utilities (importable by others)
│   ├── traceutils/               # Trace ID generation, context manipulation
│   └── instrumentation/          # Safe instrumentation helpers
│
├── api/                          # API definitions (protobuf, OpenAPI)
│   ├── protos/                   # .proto files for gRPC
│   └── openapi/                  # OpenAPI/Swagger specs for REST
│
├── configs/                      # Configuration files
│   ├── collector.yaml            # Collector configuration
│   ├── query-service.yaml        # Query service configuration
│   └── sdk-config.example.json   # Example SDK configuration
│
├── scripts/                      # Build, deployment, maintenance scripts
├── deployments/                  # Dockerfiles, Kubernetes manifests
├── tests/                        # Integration, load tests
│   ├── integration/              # Component integration tests
│   └── load/                     # Load test scenarios
│
└── docs/                         # Design docs, ADRs, user guides
    ├── adrs/                     # Architecture Decision Records
    └── milestones/               # Milestone-specific documentation
```

Key Structural Decisions

This structure embodies several architectural decisions:

1. **Clear Separation of Concerns:** Each major component (`collector`, `query`, `apmsdk`) lives in its own directory under `internal/`, minimizing accidental coupling.
2. **Internal vs. Public Code:** The `internal/` directory prevents external projects from importing our core application logic, which is an implementation detail. Publicly reusable utilities (like sampling algorithms or trace utilities) are placed in `pkg/` for potential use by other projects (e.g., custom instrumentation libraries).
3. **Shared Models Centralized:** The `internal/models/` directory contains data structures (`Trace`, `Span`, `Service`) used across multiple components. This avoids duplication and ensures consistency. However, each component's internal representations may differ from these shared models for performance or convenience.
4. **Interface-Driven Storage:** The `storage/driver/` directory defines interfaces (`StorageWriter`, `StorageReader`), while concrete implementations live separately. This allows swapping storage backends (Cassandra, Elasticsearch, etc.) without changing business logic.
5. **Configuration Externalized:** All configuration files reside in `configs/`, separate from code, enabling environment-specific deployments (dev, staging, prod).

6. **Entry Points Isolated:** Each service binary (`collector` , `query-service` , `web-ui`) has its own subdirectory under `cmd/` , each with a simple `main.go` that wires together components from `internal/` . This keeps bootstrap logic clean and separate from core business logic.

This structure scales well as the project grows. New components (like a separate anomaly detection service) would get their own directory under `internal/` and a corresponding entry point under `cmd/` .

Implementation Guidance

> **Implementation Focus:** This guidance helps you set up the foundational project structure and make initial technology choices. The actual component implementations will be detailed in subsequent sections.

A. Technology Recommendations Table

Component	Simple Option (Starting Point)	Advanced Option (Production Scale)	Rationale for Choice
Transport (SDK→Collector)	HTTP/1.1 with JSON (net/http)	gRPC with Protocol Buffers	HTTP/JSON is simpler to debug and has wider client compatibility. gRPC offers better performance and bidirectional streaming.
Collector Framework	Plain Go with goroutines + channels	OpenTelemetry Collector Contrib	Building from scratch offers maximal learning; using OTel Collector gives production-ready pipelines and extensive exporters.
Primary Storage	PostgreSQL with JSONB columns	Cassandra + Elasticsearch (dual storage)	PostgreSQL is familiar and supports JSON queries; Cassandra handles high write throughput, Elasticsearch enables rich search.
In-Memory Data Structures	Built-in maps + slices with `sync.RWMutex`	Partitioned maps with lock striping or `sync.Map`	Built-in types are simplest; advanced structures reduce lock contention under high concurrency.
Service Communication	REST HTTP calls	gRPC with service definitions	REST is straightforward; gRPC provides type safety and performance for internal service calls.
Web UI Frontend	Static HTML/JS with Vanilla or lightweight framework	React/Vue.js with state management	Simpler UI gets to visualization faster; frameworks help manage complex interactive state.

For the learning implementation, we recommend the **Simple Options** to focus on the core distributed systems concepts rather than framework intricacies.

B. Recommended File/Module Structure Starter

To bootstrap the project, create the directory structure shown above. Here's a minimal set of starter files to create:

1. Shared Models (`internal/models/trace.go`):

```go
package models

import (
    "time"
)

type Trace struct {
    TraceID   string
    Spans     []Span
    StartTime time.Time
    EndTime   time.Time
}

type Span struct {
    SpanID       string
    TraceID      string
    ParentSpanID string
    Name         string
    ServiceName  string
    StartTime    time.Time
    Duration     time.Duration
    Attributes   map[string]string
    Events       []SpanEvent
    Status       SpanStatus
}

type SpanEvent struct {
    Name       string
    Timestamp  time.Time
    Attributes map[string]string
}

type SpanStatus struct {
    Code    int    // 0=Unset, 1=Ok, 2=Error
    Message string
```

```
}

type Service struct {
    Name       string
    Operations []string
}
```

2. Configuration Structure (`internal/models/config.go`):

```go
package models

type Config struct {
    MaxIngestionRate int
    Sampling         SamplingConfig
    Storage          StorageConfig
}

func (c *Config) Validate() error {
    // TODO 1: Check MaxIngestionRate is positive
    // TODO 2: Validate Sampling configuration (rates between 0 and 1)
    // TODO 3: Validate Storage configuration (endpoints, timeouts)
    // TODO 4: Return aggregated errors if any validation fails
    return nil
}

type SamplingConfig struct {
    Probability float64
    PerService  map[string]float64
}

type StorageConfig struct {
    Type     string
    Endpoint string
    Timeout  time.Duration
}
```

3. Collector Entry Point (`cmd/collector/main.go`):

```go
package main

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

    "apm-tracing-system/internal/collector"
    "apm-tracing-system/internal/models"
)

func main() {
    // TODO 1: Load configuration from file or environment
    config := &models.Config{
        MaxIngestionRate: 1000,
        Sampling: models.SamplingConfig{Probability: 0.1},
        Storage:  models.StorageConfig{Type: "memory"},
    }

    if err := config.Validate(); err != nil {
        log.Fatalf("Invalid configuration: %v", err)
    }

    // TODO 2: Initialize the collector with configuration
    coll, err := collector.New(config)
    if err != nil {
        log.Fatalf("Failed to create collector: %v", err)
    }

    // TODO 3: Start the collector (HTTP/gRPC servers, background workers)
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()
```

```
    if err := coll.Start(ctx); err != nil {
        log.Fatalf("Failed to start collector: %v", err)
    }

    // TODO 4: Set up graceful shutdown
    sigChan := make(chan os.Signal, 1)
    signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)
    <-sigChan

    log.Println("Shutting down collector...")
    coll.Stop()
    log.Println("Collector stopped")
}
```

4. Basic Collector Skeleton (`internal/collector/collector.go`):

```go
package collector

import (
    "context"
    "apm-tracing-system/internal/models"
)

type Collector struct {
    config *models.Config
    // TODO: Add fields for HTTP server, storage writer, buffers, etc.
}

func New(config *models.Config) (*Collector, error) {
    // TODO 1: Validate configuration
    // TODO 2: Initialize storage writer based on config.Storage.Type
    // TODO 3: Initialize span buffer (in-memory or WAL)
    // TODO 4: Initialize HTTP/gRPC servers
    // TODO 5: Return the collector instance
    return &Collector{config: config}, nil
}

func (c *Collector) Start(ctx context.Context) error {
    // TODO 1: Start the HTTP/gRPC servers for span ingestion
    // TODO 2: Start background goroutines for trace assembly
    // TODO 3: Start periodic flush of completed traces to storage
    // TODO 4: Return nil if all components started successfully
    return nil
}

func (c *Collector) Stop() {
    // TODO 1: Stop accepting new spans (close listeners)
    // TODO 2: Flush all buffered traces to storage
    // TODO 3: Stop background goroutines
    // TODO 4: Close connections to storage
}
```

5. Makefile for Common Tasks:

```makefile
.PHONY: build test run-collector

build:
    go build -o bin/collector ./cmd/collector
    go build -o bin/query-service ./cmd/query-service
    go build -o bin/web-ui ./cmd/web-ui

test:
    go test ./...

run-collector:
    go run ./cmd/collector

run-all: build
    # Would use docker-compose in real scenario
    ./bin/collector &
    ./bin/query-service &
    ./bin/web-ui &
```

C. Language-Specific Hints for Go

1. **Concurrency Model:** Use goroutines for independent tasks (listening for spans, assembling traces, writing to storage). Use channels for communication between these goroutines (e.g., a channel for incoming spans, another for completed traces).
2. **Context Propagation:** Use `context.Context` extensively for cancellation and deadlines. Pass context through function calls, especially for network operations (HTTP handlers, storage writes).
3. **Error Handling:** Go's explicit error handling is ideal for observability systems. Always check errors, and add context with `fmt.Errorf("failed to write span: %w", err)` when propagating errors up the call stack.
4. **Configuration Management:** Use struct tags with libraries like `github.com/spf13/viper` for binding environment variables and YAML files to your `Config` struct.
5. **Testing:** Use table-driven tests for stateless functions. For concurrent components, use `testing` package's built-in race detector (`go test -race`) to identify data races.
6. **Performance:** Profile early with `go tool pprof`. Pay attention to memory allocations (use `go test -bench . -benchmem`) and lock contention in hot paths.

D. Milestone Checkpoint: Architecture Setup

After setting up the initial structure:

1. **Verify Directory Structure:** Run `find . -type f -name "*.go" | head -20` to confirm key files exist.
2. **Build the Project:** Run `make build` (or `go build ./...`). It should compile without errors, even though the implementations are mostly stubs.
3. **Run Basic Tests:** Execute `go test ./internal/models/...` to validate the model definitions compile correctly.
4. **Start the Collector:** Run `make run-collector` . It should start and immediately exit (since the `Start` method returns `nil` without actually starting servers). This confirms the wiring works.

Signs Something Is Wrong:

- **Compilation errors:** Check package imports and ensure all directories have proper `package` declarations.
- **"Cannot find module" errors:** Run `go mod init apm-tracing-system` at the project root to initialize the module.
- **"Undefined type" errors:** Ensure you've created all the model structs with exact field names from the naming conventions.

With this architecture in place, you have a clean foundation to implement each component in the subsequent milestones. The clear separation of concerns will allow you to work on one component (e.g., the Collector in Milestone 1) without worrying about the details of others.

> **Milestone(s):** This section provides the foundational data definitions for the entire APM Tracing System, directly supporting Milestone 1 (Trace Collection) and underpinning all subsequent milestones (Service Map, Trace Sampling, Performance Analytics, and APM SDK).

4. Data Model

Mental Model: The Family Tree Album Think of the entire APM system as a massive, digital family tree album for software requests. A **Trace** is one complete family tree—the story of a single request as it travels through your services. Each **Span** is an individual family member—a specific unit of work performed by one service. The **Service** is like a family name or household—a logical grouping of related spans. This album (your storage) must organize millions of these family trees so you can quickly find all members of a specific family (trace), identify which households (services) they visited, and understand their relationships.

Core Types: Span, Trace, and Service

The system revolves around three fundamental entities. Understanding their exact structure is critical because every component—from ingestion to visualization—operates on these types.

The Span: An Individual Operation

A **Span** represents a named, timed operation representing a unit of work performed by a single service. It's the atomic building block of distributed tracing. Think of it as a single "footprint" left by a request as it passes through a service.

Field Name	Type	Description
`SpanID`	`string`	A unique identifier for this specific span within the context of its trace. Typically a 64-bit random number encoded as 16 hexadecimal characters. Must be unique within its trace but can collide across different traces.
`TraceID`	`string`	The unique identifier of the trace to which this span belongs. All spans that are part of the same request share this same ID. Usually a 128-bit random number encoded as 32 hexadecimal characters. This is the primary grouping key.
`ParentSpanID`	`string`	The `SpanID` of the parent span in the trace hierarchy. If this span is the root (the first operation in the request), this field is an empty string. This field establishes the parent-child relationships that form the trace tree structure.
`Name`	`string`	A human-readable name describing the operation (e.g., `"HTTP GET /api/users"`, `"database.query"`, `"redis.get"`). Often called the operation name. Used for aggregation and filtering.
`ServiceName`	`string`	The name of the service that generated this span (e.g., `"user-service"`, `"auth-service"`, `"payment-gateway"`). This is how spans are associated with a particular service.
`StartTime`	`time.Time`	The precise timestamp (with nanosecond precision) when the operation represented by this span began. Must be comparable across different machines (requires clock synchronization or normalization).
`Duration`	`time.Duration`	The length of time the operation took to complete, measured from `StartTime` to the end. This is critical for performance analysis. Represented as a 64-bit integer of nanoseconds.
`Attributes`	`map[string]string`	A set of key-value pairs that provide additional contextual metadata about the span. Examples: `{"http.method": "GET", "http.status_code": "200", "db.system": "postgresql", "user.id": "12345"}`. Used for detailed debugging and filtering.
`Events`	`[]SpanEvent`	A chronological list of timestamped annotations that occurred during the span's execution. Each event represents a notable occurrence like an exception, a log statement, or a cache hit/miss.
`Status`	`SpanStatus`	The final status of the operation represented by this span. Indicates whether it completed successfully, ended with an error, or was unset. Contains both a code and an optional descriptive message.

SpanEvent: Annotated Moments in Time

A **SpanEvent** is a timestamped annotation attached to a span, representing something noteworthy that happened during the span's lifetime. Think of it as a "sticky note" attached to a specific moment on the span's timeline.

Field Name	Type	Description
`Name`	`string`	A descriptive name for the event (e.g., `"exception"`, `"cache.miss"`, `"message.sent"`, `"checkpoint"`).
`Timestamp`	`time.Time`	The exact moment when this event occurred, relative to the span's `StartTime`.
`Attributes`	`map[string]string`	Additional context specific to this event. For an exception event, this might include `{"exception.type": "IOException", "exception.message": "File not found"}`.

SpanStatus: Operation Outcome

SpanStatus indicates the final outcome of the span's operation. This is semantically different from technical success/failure—a span can complete successfully (code 0) even if the business logic failed (e.g., a `404 Not Found` response).

Field Name	Type	Description
`Code`	`int`	A numerical code representing the status. Follows OpenTelemetry semantics: `0` (Unset), `1` (OK), `2` (Error). An `Error` code indicates the operation itself failed (e.g., a database connection error, timeout).
`Message`	`string`	An optional human-readable message providing additional details about the status, particularly useful when `Code` is `2` (Error). Example: `"connection refused"`, `"timeout after 5s"`.

The Trace: A Complete Request Journey

A **Trace** is a collection of spans that together represent the complete path of a single request through the distributed system. It's not a separate stored entity but a logical aggregation computed from spans sharing the same `TraceID`.

Field Name	Type	Description
`TraceID`	`string`	The unique identifier for this trace. Matches the `TraceID` of all spans belonging to this trace.
`Spans`	`[]Span`	All spans that belong to this trace, typically in no particular order initially. They must be sorted by `StartTime` and arranged in parent-child hierarchy for display and analysis.
`StartTime`	`time.Time`	The earliest `StartTime` among all spans in the trace. Derived by finding the minimum span `StartTime`. Useful for time-range queries.
`EndTime`	`time.Time`	The latest end time (calculated as `StartTime + Duration`) among all spans in the trace. Derived by finding the maximum span end time. Together with `StartTime`, defines the trace's temporal bounds.

> **Design Insight:** The `Trace` type is primarily a **view** or **query result**, not a storage primitive. We store individual spans, then assemble them into traces when queried. This is more flexible than storing complete traces because spans can arrive out-of-order, and we can handle partial traces.

The Service: A Logical Component

A **Service** represents a deployable, versioned software component in your architecture. In tracing terms, it's the source that emits spans.

Field Name	Type	Description
`Name`	`string`	The unique name identifying this service (e.g., `"frontend-web"`, `"user-service-v2"`, `"payment-processor"`). This should match the `ServiceName` field in spans.
`Operations`	`[]string`	A list of distinct operation names (span `Name` values) that this service has been observed to perform. Derived from analyzing spans. Examples: `["HTTP GET /api/users", "HTTP POST /api/users", "database.query.users"]`. Useful for configuring sampling rates per operation.

> **Design Insight:** The `Service` type, like `Trace`, is typically a **derived entity** computed from span data, not stored directly. However, we might maintain a materialized view of services and their operations for fast access by the UI and configuration systems.

Relationships and Storage Indexing Strategy

Mental Model: The Library Catalog System Imagine our storage system as a massive library containing billions of individual pages (spans). Each book (trace) is scattered across the library—its pages aren't bound together but are identified by sharing the same call number (TraceID). To find all pages of a book, we need a catalog that tells us where every page with a given call number is located. Additionally, researchers might want to find all books that mention a particular topic (service) or were published in a certain year (time range). We need multiple catalog systems (indexes) to support these different lookup patterns efficiently.

Span-to-Trace Relationships: Building the Tree

Spans are linked into a trace through two relationships:

1. **Grouping Relationship**: All spans with the same `TraceID` belong to the same trace. This is a many-to-one relationship.
2. **Hierarchical Relationship**: The `ParentSpanID` field establishes parent-child links, forming a tree (or more generally, a directed acyclic graph) within the trace. A span's parent is another span in the same trace.

The hierarchical structure is critical for:

- **Service Map Construction**: By analyzing parent-child relationships where the parent and child have different `ServiceName` values, we can infer service-to-service calls.
- **Root Cause Analysis**: Understanding which span's failure caused cascading failures downstream.
- **Critical Path Analysis**: Identifying the longest chain of dependencies that determines the total trace duration.

Visualizing the Relationships:

Storage Strategy: Balancing Write Speed and Query Flexibility

The primary design challenge is organizing billions of spans so we can efficiently:

1. **Retrieve all spans for a specific trace** (for viewing a single request's journey)
2. **Find traces involving a specific service** (for service-focused debugging)
3. **Find traces within a time range** (for investigating incidents)
4. **Find traces with specific attributes** (e.g., all traces with `http.status_code=500`)

We face competing priorities: writing spans must be extremely fast (thousands per second), while queries should be reasonably fast but can tolerate some latency.

Architecture Decision Record: Storage Schema and Indexing Approach

> **Decision: Span-Centric Storage with Composite Primary Key and Secondary Indexes**
>
> - **Context**: We need to store massive volumes of span data (potentially billions of spans) with predictable write performance while supporting the core query patterns. The data has inherent dimensionality: trace ID (high cardinality), service name (medium cardinality), operation name (high cardinality), and timestamp (continuous).
> - **Options Considered**:
>   1. **Trace-Centric Storage**: Store complete traces as single documents (JSON/protobuf blobs) keyed by `TraceID`. All spans of a trace are written together.
>   2. **Span-Centric Storage**: Store individual spans as separate records with a composite primary key that includes `TraceID` for co-location.
>   3. **Time-Partitioned Span Storage**: Store spans in partitions based on their `StartTime` (e.g., daily or hourly partitions), with secondary indexes within each partition.
> - **Decision**: We chose **Option 2 (Span-Centric Storage with Composite Primary Key)** as our foundation, combined with time-based partitioning for data management.
> - **Rationale**:
>   - **Write Performance**: Spans arrive asynchronously from different services. A trace-centric approach would require buffering all spans of a trace before writing, risking data loss if the buffer is lost and increasing write latency. Writing spans immediately as they arrive provides better durability and lower latency.
>   - **Out-of-Order Arrival**: Spans can arrive minutes or even hours after their parent spans due to network delays, buffering, or clock skew. A span-centric model naturally accommodates this; we simply write late-arriving spans to the same logical location.
>   - **Scalability**: By using `(TraceID, SpanID)` as a composite primary key in a distributed database like Cassandra or ScyllaDB, all spans of a trace are stored on the same physical node (due to partition key hashing on `TraceID`). This makes trace retrieval a single-partition query, which is extremely efficient.
>   - **Flexibility**: We can add secondary indexes on other dimensions (`ServiceName` , `StartTime`) without affecting the primary write path.
> - **Consequences**:
>   - **Trace Assembly Requires Query**: Retrieving a full trace requires querying all spans for that `TraceID`. This is an efficient single-partition operation but still involves reading multiple records.
>   - **Secondary Index Overhead**: Maintaining indexes on `ServiceName` and `StartTime` incurs additional write overhead and storage.
>   - **Eventual Consistency**: In distributed databases, secondary indexes are often eventually consistent, meaning recently written spans might not immediately appear in service-based queries.

The following table compares the storage approaches:

Approach	Pros	Cons	Suitable For
Trace-Centric	- Single read retrieves entire trace - Natural trace assembly - Simpler query logic	- Must buffer spans before writing - Risk of data loss if buffer fails - Does not handle out-of-order span arrival well	Small-scale systems where traces complete quickly and spans arrive in order
Span-Centric	- Immediate durability for each span - Handles out-of-order arrival naturally - Scales horizontally with trace ID as shard key	- Multiple reads to assemble a trace - Requires secondary indexes for service/time queries	**Our choice**: Large-scale production systems with high-volume, asynchronous span ingestion
Time-Partitioned	- Efficient time-range queries - Easy data retention (drop old partitions) - Aligns with natural query patterns	- Spans of same trace may scatter across partitions - Requires cross-partition queries for trace assembly	Systems where time-based queries dominate and trace assembly is less frequent

Indexing Strategy: The Multi-Catalog System

We need three types of indexes to support our query patterns:

1. **Primary Index (Clustered)**: `(TraceID, SpanID)`
   - **Purpose**: Efficient retrieval of all spans for a given trace.
   - **Implementation**: In Cassandra/ScyllaDB, `TraceID` is the partition key, `SpanID` is the clustering key. In Elasticsearch, this would be a primary term index on `TraceID` with `SpanID` as document ID.
   - **Query Pattern**: `SELECT * FROM spans WHERE TraceID = ?`
2. **Secondary Index: Service + Time Range**
   - **Purpose**: Find traces for a specific service within a time window (essential for service dashboards and the service map).
   - **Implementation**: A separate index table with composite key `(ServiceName, StartTimeBucket, TraceID)`, where `StartTimeBucket` is a time window (e.g., hour or day) for range query efficiency.
   - **Query Pattern**: `SELECT TraceID FROM service_index WHERE ServiceName = ? AND StartTimeBucket >= ? AND StartTimeBucket <= ? LIMIT 1000`
3. **Secondary Index: Global Time Range**
   - **Purpose**: Find all traces within a time range (for incident investigation when you don't know which service was involved).
   - **Implementation**: A time-series-optimized store or a separate index table with key `(StartTimeBucket, TraceID)`.
   - **Query Pattern**: `SELECT TraceID FROM time_index WHERE StartTimeBucket >= ? AND StartTimeBucket <= ? LIMIT 10000`
4. **Attribute Index (Selective)**
   - **Purpose**: Find traces with specific attribute values (e.g., all traces where `http.status_code = 500`).
   - **Implementation**: Only index high-value, low-cardinality attributes (like `http.status_code`, `error=true`). High-cardinality attributes (like `user.id`) should not be indexed; instead, use filtering on already-retrieved traces.
   - **Query Pattern**: `SELECT TraceID FROM attr_index WHERE attr_key = ? AND attr_value = ? AND StartTimeBucket = ?`

> **Design Insight:** We use a **two-phase query** pattern for most searches: First, use a secondary index to find relevant `TraceID`s, then use the primary index to fetch the complete span data for those traces. This balances query flexibility with write performance.

Data Retention and Partitioning

Given the massive volume of trace data, we cannot store everything forever. We implement:

- **Time-Based Partitioning**: Spans are partitioned by their `StartTime` (e.g., daily partitions). Old partitions can be archived or deleted based on retention policies.
- **Sampling as First-Line Defense**: Before indexing, we apply sampling (Milestone 3) to reduce the volume of spans stored.
- **Aggregation for Long-Term Trends**: For performance analytics (Milestone 4), we compute and store aggregated metrics (p50, p95, p99 latencies per service per minute) which have much longer retention than raw spans.

Common Pitfalls in Data Modeling and Storage

⚠️ Pitfall: Treating Trace as a Stored Entity

- **Description**: Storing complete traces as monolithic blobs rather than individual spans.
- **Why it's wrong**: When a new span arrives for an existing trace, you must read-modify-write the entire trace document. This creates contention, complicates out-of-order writes, and makes partial failures more costly (losing an entire trace vs. one span).
- **How to avoid**: Store spans individually. Treat `Trace` as a view assembled at query time. Use `TraceID` as the primary grouping key in your storage schema.

⚠️ Pitfall: Over-Indexing High-Cardinality Fields

- **Description**: Creating secondary indexes on fields with many distinct values, such as full HTTP URLs with parameters (`/api/users/12345`) or user IDs.
- **Why it's wrong**: High-cardinality indexes become massive (approaching the size of the primary data), slow down writes dramatically, and can overwhelm the index system.
- **How to avoid**: Only index low-to-medium cardinality fields (`ServiceName`, `http.method`, `http.status_code`). For high-cardinality filtering, use post-retrieval filtering or specialized search systems. Normalize operation names by removing dynamic parameters before indexing (e.g., `/api/users/{id}` instead of `/api/users/12345`).

⚠️ Pitfall: Ignoring Clock Skew in Timestamps

- **Description**: Assuming all spans have perfectly synchronized clocks and using raw `StartTime` values for sorting without normalization.
- **Why it's wrong**: Servers can have minutes or even hours of clock drift. A child span might appear to start before its parent, breaking causality and making trace visualization incorrect.
- **How to avoid**: Implement **clock skew correction**. One approach: record the difference between the local clock and a reference (like NTP) in span attributes, or use relative timestamps (monotonic clocks) for duration but absolute timestamps only for coarse ordering. During trace assembly, adjust timestamps based on parent-child relationships or use heuristics to reorder spans.

⚠️ Pitfall: Storing Unlimited Attribute Data

- **Description**: Allowing applications to send spans with megabytes of attribute data (e.g., entire request/response bodies).
- **Why it's wrong**: This blows up storage costs, slows down queries, and can expose sensitive data. The tracing system is for performance monitoring, not general-purpose logging.
- **How to avoid**: Enforce **attribute size limits** at the collector (e.g., 2KB total per span). Truncate or drop excess attributes. Provide clear guidelines to developers about what belongs in spans (metadata, not payloads).

Implementation Guidance

Technology Recommendations Table:

Component	Simple Option (Development/Testing)	Advanced Option (Production)
Primary Storage	In-memory map keyed by `TraceID` (Go `map[string][]Span`)	Distributed columnar database: **Cassandra** or **ScyllaDB**
Secondary Indexes	Separate in-memory maps (e.g., `map[string][]string` for service→traceIDs)	Cassandra **Materialized Views** or **Secondary Index** tables
Time-Series Storage	Rolling window in Go memory with periodic snapshot to disk	**TimescaleDB** (PostgreSQL extension) or **Prometheus** + **Thanos**
Serialization Format	JSON over HTTP (easy debugging)	**Protocol Buffers** over gRPC (efficient binary serialization)

Recommended File/Module Structure:

```
apm-tracing-system/
├── internal/
│   ├── models/                  # Core data types
│   │   ├── span.go              # Span, SpanEvent, SpanStatus types
│   │   ├── trace.go             # Trace type (mostly view/computation)
│   │   ├── service.go           # Service type
│   │   └── config.go            # Config, SamplingConfig, StorageConfig
│   ├── storage/                 # Storage abstractions and implementations
│   │   ├── storage.go           # Storage interface
│   │   ├── inmemory/            # Simple in-memory implementation
│   │   │   ├── inmemory.go
│   │   │   └── inmemory_test.go
│   │   ├── cassandra/           # Cassandra implementation
│   │   │   ├── cassandra.go
│   │   │   └── schema.cql       # CQL schema definitions
│   │   └── index/               # Index management
│   │       ├── indexer.go
│   │       ├── service_index.go
│   │       └── time_index.go
│   └── utils/
│       └── timeutils.go         # Clock skew correction utilities
└── pkg/
    └── apmproto/                # Protocol Buffer definitions
        └── span.proto           # Protobuf schema for span ingestion
```

Infrastructure Starter Code (Complete Models):

File: `internal/models/span.go`

```go
package models

import (
	"time"
)

// Span represents a single operation within a distributed trace.
type Span struct {
	SpanID       string            `json:"spanId"`                // Unique within trace
	TraceID      string            `json:"traceId"`               // Groups spans into traces
	ParentSpanID string            `json:"parentSpanId,omitempty"` // Empty for root spans
	Name         string            `json:"name"`                  // Operation name
	ServiceName  string            `json:"serviceName"`           // Originating service
	StartTime    time.Time         `json:"startTime"`             // When operation started
	Duration     time.Duration     `json:"duration"`              // How long it took
	Attributes   map[string]string `json:"attributes,omitempty"`  // Contextual metadata
	Events       []SpanEvent       `json:"events,omitempty"`      // Timestamped annotations
	Status       SpanStatus        `json:"status"`                // Success/error status
}

// SpanEvent represents a notable event that occurred during a span's execution.
type SpanEvent struct {
	Name       string            `json:"name"`                  // Event name (e.g., "exception")
	Timestamp  time.Time         `json:"timestamp"`             // When it happened
	Attributes map[string]string `json:"attributes,omitempty"` // Event-specific context
}

// SpanStatus indicates the final status of a span.
type SpanStatus struct {
	Code    int    `json:"code"`              // 0=Unset, 1=OK, 2=Error
	Message string `json:"message,omitempty"` // Optional description
}
```

File: `internal/models/trace.go`

GO

```go
package models

import (
	"sort"
	"time"
)

// Trace represents a complete request journey composed of multiple spans.
// This is typically a computed view, not a stored entity.
type Trace struct {
	TraceID   string    `json:"traceId"`
	Spans     []Span    `json:"spans"`
	StartTime time.Time `json:"startTime"` // Derived: min(span.StartTime)
	EndTime   time.Time `json:"endTime"`   // Derived: max(span.StartTime + span.Duration)
}

// NewTraceFromSpans constructs a Trace view from a slice of spans with the same TraceID.
// It calculates the trace's start and end times and sorts spans by start time.
func NewTraceFromSpans(spans []Span) (*Trace, error) {
	if len(spans) == 0 {
		return nil, errors.New("cannot create trace from empty span slice")
	}

	traceID := spans[0].TraceID
	startTime := spans[0].StartTime
	endTime := spans[0].StartTime.Add(spans[0].Duration)

	// Verify all spans belong to the same trace and calculate time bounds
	for _, span := range spans[1:] {
		if span.TraceID != traceID {
			return nil, errors.New("spans have different trace IDs")
		}
		if span.StartTime.Before(startTime) {
			startTime = span.StartTime
```

```
                }

                spanEnd := span.StartTime.Add(span.Duration)

                if spanEnd.After(endTime) {

                        endTime = spanEnd

                }

        }



        // Sort spans by start time for consistent viewing

        sortedSpans := make([]Span, len(spans))

        copy(sortedSpans, spans)

        sort.Slice(sortedSpans, func(i, j int) bool {

                return sortedSpans[i].StartTime.Before(sortedSpans[j].StartTime)

        })



        return &Trace{

                TraceID:   traceID,

                Spans:     sortedSpans,

                StartTime: startTime,

                EndTime:   endTime,

        }, nil

}
```

File: `internal/models/service.go`

```go
package models


// Service represents a logical component in the distributed system.

type Service struct {

        Name       string   `json:"name"`                // Unique service identifier

        Operations []string `json:"operations,omitempty"` // Observed operation names

}
```

Core Logic Skeleton Code (Storage Interface):

File: `internal/storage/storage.go`

```go
package storage

import (
	"context"
	"time"

	"github.com/your-org/apm-tracing/internal/models"
)

// Storage defines the interface for persistent span storage.
// Implementations must handle concurrent access and provide
// the core query patterns needed by the APM system.
type Storage interface {
	// StoreSpan persists a single span to storage.
	// Implementations should also update any secondary indexes.
	// Returns an error if the span cannot be stored.
	StoreSpan(ctx context.Context, span models.Span) error

	// GetTraceByID retrieves all spans for the given trace ID
	// and assembles them into a Trace view. Returns nil if trace not found.
	GetTraceByID(ctx context.Context, traceID string) (*models.Trace, error)

	// GetTracesByService returns traces that involve the given service
	// within the specified time range. Results are limited to 'limit' traces
	// and ordered by most recent start time first.
	GetTracesByService(ctx context.Context, serviceName string,
		startTime, endTime time.Time, limit int) ([]*models.Trace, error)

	// GetTracesByTimeRange returns traces that started within the given
	// time range, ordered by start time (descending). Useful for incident
	// investigation when service is unknown.
	GetTracesByTimeRange(ctx context.Context,
		startTime, endTime time.Time, limit int) ([]*models.Trace, error)
```

```
        // GetServices returns a list of all services that have emitted spans,
        // along with their observed operations. This is used for the service map
        // and for configuration UI.
        GetServices(ctx context.Context) ([]*models.Service, error)


        // Close gracefully shuts down the storage connection and releases resources.
        Close() error
}
```

File: `internal/storage/inmemory/inmemory.go` (Skeleton)

```go
package inmemory

import (
	"context"
	"sort"
	"sync"
	"time"

	"github.com/your-org/apm-tracing/internal/models"
)

// InMemoryStorage is a simple in-memory implementation of storage.Storage.
// Useful for development, testing, and small-scale deployments.
// NOT suitable for production due to memory limits and lack of durability.
type InMemoryStorage struct {
	mu sync.RWMutex

	// Primary storage: traceID -> list of spans
	traces map[string][]models.Span

	// Secondary index: serviceName -> set of traceIDs
	serviceIndex map[string]map[string]struct{}

	// Time index: startTime bucket -> list of traceIDs
	timeIndex map[time.Time][]string
}

func NewInMemoryStorage() *InMemoryStorage {
	return &InMemoryStorage{
		traces:       make(map[string][]models.Span),
		serviceIndex: make(map[string]map[string]struct{}),
		timeIndex:    make(map[time.Time][]string),
	}
}
```

```go
func (s *InMemoryStorage) StoreSpan(ctx context.Context, span models.Span) error {
        s.mu.Lock()
        defer s.mu.Unlock()

        // TODO: Implement step-by-step:
        // 1. Append span to s.traces[span.TraceID] slice
        // 2. Update service index: add span.TraceID to s.serviceIndex[span.ServiceName]
        //    (create map/set if needed)
        // 3. Update time index: bucket span.StartTime by hour (truncate to hour)
        //    and add span.TraceID to s.timeIndex[bucketTime]
        // 4. Return nil on success, error on failure

        return nil
}

func (s *InMemoryStorage) GetTraceByID(ctx context.Context, traceID string) (*models.Trace, error) {
        s.mu.RLock()
        defer s.mu.RUnlock()

        // TODO: Implement:
        // 1. Look up spans := s.traces[traceID]
        // 2. If len(spans) == 0, return nil, nil (trace not found)
        // 3. Use models.NewTraceFromSpans(spans) to create trace view
        // 4. Return trace, nil

        return nil, nil
}

// Additional method implementations follow similar pattern...
```

Language-Specific Hints:

1. **Time Handling**: Use `time.Time` for timestamps throughout. For nanosecond precision, ensure your database driver supports it (Cassandra's `timestamp` type supports microseconds). For bucketing in indexes, use `t.Truncate(time.Hour)` to create hourly buckets.

2. **Concurrency**: The in-memory implementation uses `sync.RWMutex` for concurrent access. For production implementations, the database handles concurrency.
3. **ID Generation**: Use `crypto/rand` for generating random `TraceID` and `SpanID` values in the SDK. Example: `hex.EncodeToString(randomBytes(16))` for 128-bit trace IDs.
4. **Memory Management**: The in-memory storage will grow unbounded. In production, you'd use a real database. For testing, consider adding a maximum spans limit and LRU eviction.

Milestone Checkpoint (Trace Collection - Data Model):

After implementing the data model and basic in-memory storage:

1. **Verify Model Definitions**: Create a simple test that instantiates each type with sample data:

```go
func TestDataModel() {
    span := models.Span{
        SpanID:      "abc123",
        TraceID:     "trace-1",
        ParentSpanID: "",
        Name:        "GET /api/users",
        ServiceName: "user-service",
        StartTime:   time.Now(),
        Duration:    150 * time.Millisecond,
        Attributes:  map[string]string{"http.method": "GET"},
        Status:      models.SpanStatus{Code: 1},
    }
    // Verify fields are accessible
}
```

2. **Test Storage Interface**: Implement the `InMemoryStorage` methods and verify:

```bash
go test ./internal/storage/inmemory/... -v
```

Expected: Tests should pass, demonstrating span storage and retrieval.

3. **Verify Trace Assembly**: Write a test that stores multiple spans with the same `TraceID` but different `SpanID`s, then retrieves the trace and verifies all spans are present and the `StartTime`/`EndTime` are correctly calculated.

Debugging Tips:

Symptom	Likely Cause	How to Diagnose	Fix
"Span disappears after storage"	Concurrent map write panic or incorrect locking	Add debug logging before and after storage operations. Run with `-race` flag.	Ensure proper use of `sync.RWMutex` locks in `InMemoryStorage`.
"Trace shows incorrect time range"	`StartTime` / `EndTime` calculation logic error	Print each span's start and end times during trace assembly.	Verify `NewTraceFromSpans` correctly computes min/max across all spans.
"Service index grows without bound"	Not cleaning up old trace IDs from index when traces expire	Check index sizes over time. Add metrics for index entries.	Implement TTL or cleanup routine that removes old entries from indexes.
"High memory usage in tests"	In-memory storage retaining all spans indefinitely	Monitor memory with `runtime.ReadMemStats`.	Add maximum capacity to `InMemoryStorage` with LRU eviction for testing.

> **Milestone(s):** This section corresponds to Milestone 1: Trace Collection, which forms the foundation of the entire APM Tracing System.

5. Component Design: Trace Collection (Milestone 1)

The **Trace Collector** is the foundational component of our APM system — it's the "front door" through which all telemetry data enters. This component is responsible for receiving, validating, processing, and storing spans that together form distributed traces. Its design directly impacts the system's reliability, scalability, and correctness.

Mental Model: The Package Sorting Hub

Imagine a massive international package sorting facility (like an airport cargo terminal). Thousands of packages (spans) arrive every second via different delivery vehicles (applications). Each package has a destination address (trace ID) and may be part of a larger shipment (trace). The facility must:

1. **Accept deliveries** from all carriers (HTTP, gRPC) without blocking traffic
2. **Inspect each package** for damage or incorrect labeling (validation)
3. **Sort packages by destination** (grouping spans by trace ID)
4. **Hold incomplete shipments** until all packages arrive (buffering)
5. **Store completed shipments** in the warehouse (persistent storage)
6. **Maintain an index** of what's stored and where (secondary indexing)

The challenge is that packages from the same shipment don't arrive together — some come hours later. The facility must have enough temporary storage (buffers) to hold incomplete shipments, but not so much that it runs out of space. It also needs efficient systems to quickly locate all packages from a specific shipment when requested.

Collector Interface and API

The Collector exposes well-defined endpoints for receiving telemetry data. We support both HTTP (for simplicity and wide compatibility) and gRPC (for high-performance streaming) protocols, following the **OpenTelemetry Protocol (OTLP)** standard.

HTTP Endpoint Specification

Method	Path	Content-Type	Request Body	Response	Description
POST	`/v1/traces`	`application/json`	JSON array of spans	`202 Accepted` (or error)	Accepts batch of spans in JSON format
POST	`/v1/traces`	`application/x-protobuf`	OTLP protobuf binary	`202 Accepted` (or error)	Accepts spans in binary protobuf format
GET	`/health`	-	-	`200 OK` with status JSON	Health check endpoint

gRPC Service Specification

Service Method	Request Type	Response Type	Description
`Export`	`ExportTraceServiceRequest`	`ExportTraceServiceResponse`	Stream spans to collector via gRPC
`HealthCheck`	`HealthCheckRequest`	`HealthCheckResponse`	Service health status

Request/Response Format Details

The collector accepts spans in the **OpenTelemetry** format. Below is the JSON representation structure (simplified for clarity):

Span JSON Format:

Field	Type	Required	Description
`traceId`	string (hex)	Yes	32-character hex string identifying the trace
`spanId`	string (hex)	Yes	16-character hex string identifying this span
`parentSpanId`	string (hex)	No	Parent span ID (empty for root spans)
`name`	string	Yes	Operation name (e.g., "GET /api/users")
`kind`	integer	Yes	Span kind (1=CLIENT, 2=SERVER, etc.)
`startTimeUnixNano`	integer	Yes	Start timestamp in nanoseconds
`endTimeUnixNano`	integer	Yes	End timestamp in nanoseconds
`attributes`	array of key-value	No	Custom metadata (e.g., {"http.method": "GET"})
`events`	array	No	Timed events within the span
`status`	object	No	Status code and optional message

Response Codes:

Status Code	Meaning	When Used
202 Accepted	Successfully accepted	All spans validated and queued
400 Bad Request	Malformed request	Invalid JSON or missing required fields
429 Too Many Requests	Rate limit exceeded	Client exceeds configured ingestion rate
503 Service Unavailable	Collector overloaded	Backpressure applied, client should retry

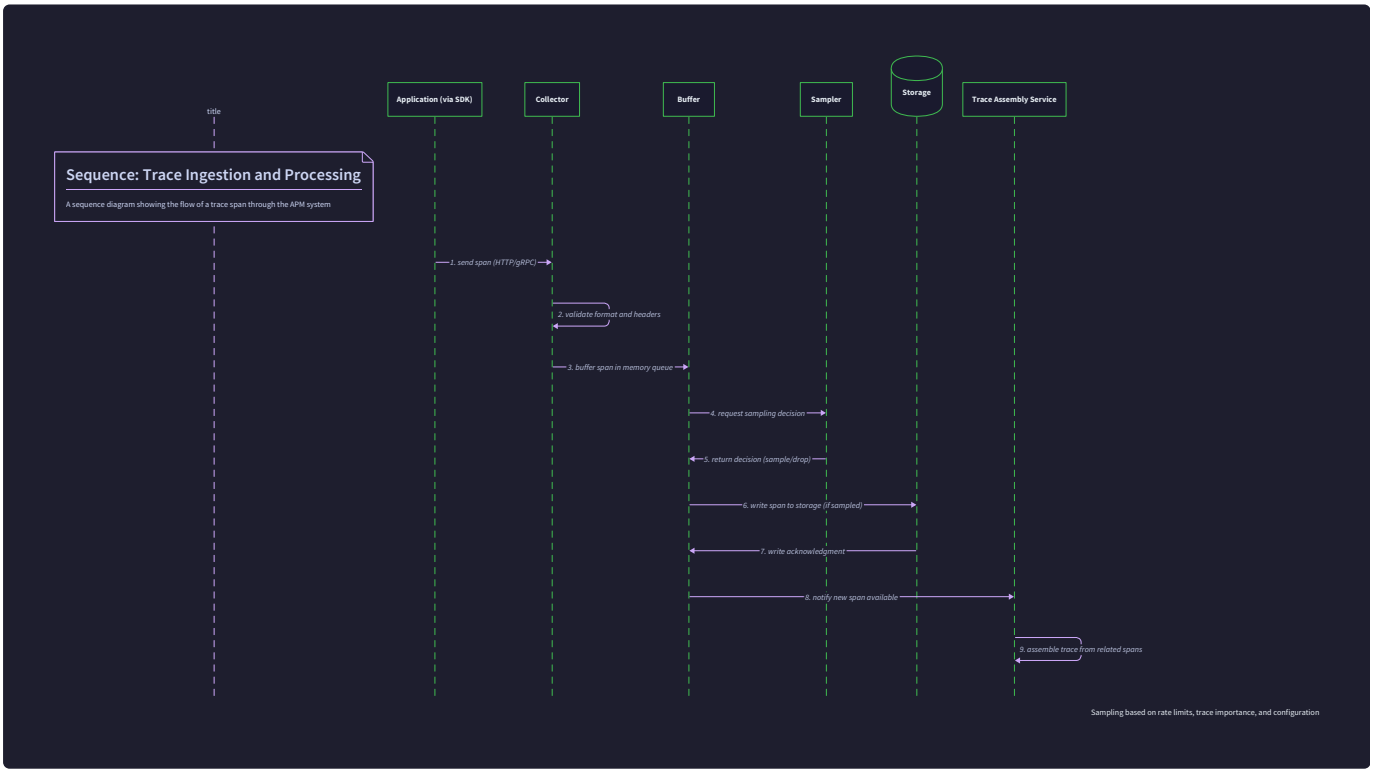
Expected Behavior and SLAs

The collector must provide these guarantees:

1. **Acknowledgment within 100ms** for 95% of requests (P95 latency)
2. **No data loss** under normal operating conditions
3. **Graceful degradation** when overloaded (apply backpressure instead of crashing)
4. **Span validation** to reject malformed data before processing
5. **Trace completeness** — all spans of a sampled trace are stored together

Internal Behavior: Ingestion Pipeline

When a span arrives at the collector, it goes through a multi-stage pipeline. Each stage transforms, filters, or routes the data. This follows the **pipes and filters** architectural pattern, where each stage is independent and can be scaled separately.



Pipeline Stages (Step-by-Step Algorithm)

1. **Receipt and Deserialization**
    - Accept connection from client (HTTP or gRPC)
    - Read request body and parse based on content-type (JSON or protobuf)
    - Validate basic structure: required fields present, correct data types
    - Convert to internal `Span` struct format for uniform processing
2. **Validation and Sanitization**
    - Validate trace ID and span ID formats (hexadecimal, correct length)
    - Verify timestamps: `endTime` ≥ `startTime`, timestamps not in future
    - Sanitize attributes: truncate overly long values, remove sensitive data
    - Check for mandatory OpenTelemetry semantic conventions
3. **Sampling Decision (Head-Based)**
    - Compute hash of trace ID (consistent hashing)

- Apply sampling rate based on service name and operation
- If trace is rejected, drop all spans immediately (no further processing)
- Record sampling decision for later reference

4. **Buffering and Grouping**
   - Add span to in-memory buffer keyed by trace ID
   - Check if trace is now "complete" (all expected spans received)
   - For incomplete traces, start/refresh timeout timer
   - For complete traces, move to next stage immediately
5. **Trace Assembly**
   - Retrieve all spans for the trace from buffer
   - Build parent-child relationships using span ID and parent span ID
   - Calculate trace-level statistics: start time, end time, total duration
   - Create `Trace` struct containing all spans in chronological order
6. **Storage Persistence**
   - Write all spans of the trace to persistent storage (bulk write)
   - Create secondary indexes for efficient querying
   - Update service registry with new service/operation names
   - Notify downstream components (service map, analytics)
7. **Buffer Cleanup**
   - Remove trace from in-memory buffer after successful persistence
   - For timed-out traces (incomplete after window), write partial trace with metadata
   - Report metrics: buffer size, trace completion rate, timeout count

Trace Completion Detection

A trace is considered "complete" when either:

1. All expected spans have arrived (based on span count metadata if provided)
2. A configurable timeout (default: 5 minutes) has elapsed since the first span
3. An explicit "trace end" marker span is received

The collector uses a **watermark-based approach**: when the latest timestamp among all received spans exceeds `current_time - timeout_window`, and no new spans have arrived for that period, the trace is considered complete.

ADR: Buffering Strategy for Late-Arriving Spans

> **Decision: Hybrid Buffering with Write-Ahead Logging**
>
> - **Context**: Spans from the same trace arrive out of order and potentially hours apart due to network delays, retries, or slow services. The collector must temporarily store incomplete traces without risking data loss if the collector crashes.
> - **Options Considered**:
>   1. **Pure In-Memory Buffer**: Keep all incomplete traces in RAM with periodic snapshots to disk
>   2. **Write-Ahead Log (WAL) + Memory Index**: Write all spans immediately to disk log, with in-memory index of incomplete traces
>   3. **External Buffer Service**: Offload buffering to external system (Redis, Kafka)
> - **Decision**: Hybrid approach: spans written to WAL immediately, with in-memory index for fast trace assembly
> - **Rationale**:
>   - **Durability**: WAL ensures no data loss on collector crash/restart
>   - **Performance**: In-memory index enables fast trace completion checks
>   - **Simplicity**: No external dependencies for core functionality
>   - **Cost-effective**: Disk storage cheaper than RAM for large buffers
> - **Consequences**:
>   - Added disk I/O for every span (mitigated by batching writes)
>   - Need for WAL compaction to remove completed traces
>   - Slightly higher latency per span but better overall reliability

Options Comparison Table:

Option	Pros	Cons	Chosen?
Pure In-Memory Buffer	- Fastest performance - Simple implementation	- Data loss on crash - Memory grows unbounded - Difficult to scale	No
WAL + Memory Index	- Durable (crash-safe) - Memory usage predictable - Easier to scale horizontally	- Disk I/O overhead - Requires WAL management - More complex implementation	**Yes**
External Buffer Service	- Offloads resource management - Easier to scale independently - Built-in replication	- External dependency - Network latency - Additional operational complexity	No

WAL Implementation Details

The Write-Ahead Log is implemented as a set of append-only files:

1. **Span Log**: Each incoming span is serialized and appended with metadata:
   - Timestamp of receipt
   - Trace ID and span ID
   - Processing status (pending, processed)
2. **Index File**: In-memory map from trace ID to:
   - List of span offsets in the WAL
   - Timestamp of first span

- Current state (incomplete, assembling, complete)

3. **Cleanup Process**: Background goroutine that:
   - Periodically scans for completed traces older than retention period
   - Removes their entries from the index
   - Marks corresponding WAL segments as reclaimable
4. **Recovery Process**: On collector restart:
   - Read WAL from last checkpoint
   - Rebuild in-memory index
   - Resume processing incomplete traces

Common Pitfalls in Trace Collection

⚠️ Pitfall 1: Unbounded Memory Growth from Incomplete Traces

- **Description**: Holding all incomplete traces in memory indefinitely causes OOM crashes
- **Why It's Wrong**: Memory is finite; slow or stuck traces accumulate forever
- **Fix**: Implement **time-based eviction** with disk spillover:
  1. Set maximum buffer size (e.g., 100,000 traces)
  2. Move oldest incomplete traces to disk when limit reached
  3. Implement TTL (e.g., 24 hours) after which partial traces are discarded with warning

⚠️ Pitfall 2: Clock Skew Creates Incorrect Span Ordering

- **Description**: Spans from services with misconfigured clocks appear in wrong chronological order
- **Why It's Wrong**: Trace visualization shows impossible sequences (child before parent)
- **Fix**: Apply **clock skew correction**:
  1. Record collector receipt timestamp for each span
  2. Compare service timestamp vs. receipt timestamp
  3. Apply offset correction if discrepancy exceeds threshold (e.g., 5 seconds)
  4. Log warnings for services with persistent clock issues

⚠️ Pitfall 3: Incorrect Parent-Child Linking

- **Description**: Spans with invalid parentSpanID create orphaned spans or incorrect hierarchies
- **Why It's Wrong**: Breaks trace continuity, makes debugging impossible
- **Fix**: Implement **strict validation and repair**:
  1. Validate parentSpanID format (hexadecimal, correct length)
  2. Verify parent span exists in same trace before linking
  3. For invalid parents, mark span as "root" with error attribute
  4. Provide metrics on linking failure rate per service

⚠️ Pitfall 4: Head-of-Line Blocking in Pipeline

- **Description**: Slow processing of one trace blocks all subsequent traces
- **Why It's Wrong**: Reduces overall throughput, causes request timeouts
- **Fix**: Use **bounded worker pools with per-trace isolation**:
  1. Each trace processed by dedicated worker
  2. Workers can't block each other
  3. Slow traces don't affect unrelated traces

4. Implement circuit breakers for pathological traces

⚠️ Pitfall 5: Lost Spans from Trace Sampling Inconsistency

- **Description**: Different spans from same trace get different sampling decisions
- **Why It's Wrong**: Creates incomplete traces missing critical spans
- **Fix**: Implement **consistent sampling with decision caching**:
  1. Make sampling decision based on trace ID (not span ID)
  2. Cache decision for trace TTL (e.g., 1 hour)
  3. All subsequent spans from same trace use cached decision
  4. Reject spans if decision cache is full (fail-safe to "keep")

Implementation Guidance for Trace Collection

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
HTTP Server	Go's `net/http` with JSON	`gin` or `chi` router with middleware
gRPC Server	Standard `google.golang.org/grpc`	With interceptors for auth/telemetry
Protobuf	`google.golang.org/protobuf`	With code generation via `protoc`
WAL Implementation	Custom file-based append log	Use `github.com/tidwall/wal` library
In-Memory Store	`sync.Map` for concurrent access	Sharded maps with `github.com/orcaman/concurrent-map`
Serialization	JSON for readability	Protocol Buffers for performance
Metrics	Prometheus client library	OpenTelemetry metrics SDK

B. Recommended File/Module Structure

```
project-root/
├── cmd/
│   └── collector/
│       └── main.go                # Collector entry point
├── internal/
│   ├── collector/
│   │   ├── api/                   # API layer
│   │   │   ├── http/
│   │   │   │   ├── server.go      # HTTP server implementation
│   │   │   │   └── handlers.go    # Request handlers
│   │   │   └── grpc/
│   │   │       ├── server.go      # gRPC server implementation
│   │   │       └── service.go     # gRPC service definition
│   │   ├── pipeline/              # Processing pipeline
│   │   │   ├── pipeline.go        # Pipeline orchestration
│   │   │   ├── stages/            # Individual pipeline stages
│   │   │   │   ├── receiver.go    # Stage 1: Receipt
│   │   │   │   ├── validator.go   # Stage 2: Validation
│   │   │   │   ├── sampler.go     # Stage 3: Sampling
│   │   │   │   ├── buffer.go      # Stage 4: Buffering
│   │   │   │   ├── assembler.go   # Stage 5: Assembly
│   │   │   │   └── persister.go   # Stage 6: Persistence
│   │   │   └── queue.go           # Inter-stage queues
│   │   ├── buffer/                # Buffering subsystem
│   │   │   ├── wal/
│   │   │   │   ├── writer.go      # WAL write operations
│   │   │   │   ├── reader.go      # WAL read operations
│   │   │   │   ├── index.go       # In-memory index
│   │   │   │   └── cleaner.go     # WAL cleanup
│   │   │   └── memory/
│   │   │       ├── manager.go     # Buffer manager
│   │   │       ├── eviction.go    # Eviction policies
│   │   │       └── metrics.go     # Buffer metrics
│   │   └── storage/               # Storage abstraction
│   │       ├── interface.go       # Storage interface
│   │       ├── memory_store.go    # In-memory implementation (dev)
│   │       └── cassandra_store.go # Cassandra implementation
├── pkg/
│   ├── models/                    # Data models
│   │   ├── trace.go               # Trace struct
│   │   ├── span.go                # Span struct
│   │   └── service.go             # Service struct
│   └── sampling/                  # Sampling logic
│       ├── sampler.go             # Sampler interface
│       └── consistent_hash.go     # Consistent hash sampler
└── proto/                         # Protobuf definitions
    └── otlp/                      # OpenTelemetry proto files
```

C. Infrastructure Starter Code

Complete WAL Writer Implementation:

```go
// internal/collector/buffer/wal/writer.go

package wal


import (

    "encoding/binary"

    "os"

    "sync"

    "time"

)


type WALWriter struct {

    file     *os.File

    mu       sync.Mutex

    filePath string

    offset   int64

}


// NewWALWriter creates a new WAL writer for the given file path

func NewWALWriter(filePath string) (*WALWriter, error) {

    file, err := os.OpenFile(filePath, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)

    if err != nil {

        return nil, err

    }



    stat, err := file.Stat()

    if err != nil {

        file.Close()

        return nil, err

    }



    return &WALWriter{

        file:     file,

        filePath: filePath,

        offset:   stat.Size(),
```

```
    }, nil
}

// Append writes a record to the WAL and returns its offset
func (w *WALWriter) Append(data []byte) (int64, error) {
    w.mu.Lock()
    defer w.mu.Unlock()

    // Write length prefix (8 bytes)
    length := uint64(len(data))
    if err := binary.Write(w.file, binary.LittleEndian, length); err != nil {
        return -1, err
    }

    // Write the actual data
    if _, err := w.file.Write(data); err != nil {
        return -1, err
    }

    // Sync to disk for durability
    if err := w.file.Sync(); err != nil {
        return -1, err
    }

    currentOffset := w.offset
    w.offset += 8 + int64(length) // 8 bytes for length prefix

    return currentOffset, nil
}

// Rotate creates a new WAL file and returns the old file for cleanup
func (w *WALWriter) Rotate() (*os.File, error) {
    w.mu.Lock()
```

```
    defer w.mu.Unlock()


    oldFile := w.file


    // Create new file with timestamp
    newFilePath := w.filePath + "." + time.Now().Format("20060102_150405")
    newFile, err := os.OpenFile(newFilePath, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        return nil, err
    }


    w.file = newFile
    w.offset = 0
    w.filePath = newFilePath


    return oldFile, nil
}

// Close gracefully closes the WAL file
func (w *WALWriter) Close() error {
    w.mu.Lock()
    defer w.mu.Unlock()


    return w.file.Close()
}
```

HTTP Server with Health Check:

```go
// internal/collector/api/http/server.go

package http

import (
    "context"
    "fmt"
    "net/http"
    "time"
)

type Server struct {
    server *http.Server
    port   int
}

func NewServer(port int, handler http.Handler) *Server {
    return &Server{
        server: &http.Server{
            Addr:         fmt.Sprintf(":%d", port),
            Handler:      handler,
            ReadTimeout:  10 * time.Second,
            WriteTimeout: 10 * time.Second,
        },
        port: port,
    }
}

func (s *Server) Start() error {
    fmt.Printf("Starting HTTP server on port %d\n", s.port)
    return s.server.ListenAndServe()
}

func (s *Server) Shutdown(ctx context.Context) error {
    fmt.Println("Shutting down HTTP server...")
    return s.server.Shutdown(ctx)
```

```
}


// HealthHandler provides health check endpoint
func HealthHandler(w http.ResponseWriter, r *http.Request) {
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, `{"status": "healthy", "timestamp": "%s"}`, time.Now().UTC().Format(time.RFC3339))
}
```

D. Core Logic Skeleton Code

Trace Assembler (Core Algorithm):

```go
// internal/collector/pipeline/stages/assembler.go

package stages

import (
    "context"
    "sort"
    "time"

    "github.com/your-org/apm-tracing/pkg/models"
)

// TraceAssembler takes a slice of spans and constructs a complete Trace
type TraceAssembler struct {
    maxTraceDuration time.Duration
}

// NewTraceAssembler creates a new trace assembler with the given configuration
func NewTraceAssembler(maxTraceDuration time.Duration) *TraceAssembler {
    return &TraceAssembler{
        maxTraceDuration: maxTraceDuration,
    }
}

// AssembleTrace takes all spans for a trace and builds the hierarchical structure
func (ta *TraceAssembler) AssembleTrace(ctx context.Context, spans []models.Span) (*models.Trace, error) {
    // TODO 1: Validate that all spans have the same trace ID
    //   - If not, return error "spans belong to different traces"

    // TODO 2: Sort spans by start time (ascending)
    //   - Use sort.Slice with spans[i].StartTime comparison

    // TODO 3: Build a map from span ID to span for fast lookup
    //   - Create map[string]*models.Span
    //   - Store pointer to each span (not copy)
```

```
    // TODO 4: Identify root span(s) - spans with empty or non-existent parentSpanID
    //   - For each span, check if parentSpanID exists in the map
    //   - If not, this is a root span (could be multiple in case of errors)

    // TODO 5: Build parent-child relationships
    //   - For each non-root span, find its parent in the map
    //   - If parent found, establish relationship (you may need to add Children field to Span)
    //   - If parent not found, mark as "orphaned" with warning attribute

    // TODO 6: Calculate trace-level statistics
    //   - Find earliest start time among all spans
    //   - Find latest end time (start + duration)
    //   - Verify total duration doesn't exceed maxTraceDuration
    //   - Count spans by status code (success, error)

    // TODO 7: Create and return the Trace object
    //   - Initialize models.Trace with TraceID from first span
    //   - Set Spans field to the sorted slice
    //   - Set StartTime and EndTime to calculated values
    //   - Return pointer to Trace

    return nil, nil // Remove this line after implementation
}

// IsTraceComplete checks if we have all spans for a trace
func (ta *TraceAssembler) IsTraceComplete(ctx context.Context, traceID string, spans []models.Span,
firstSpanTime time.Time) bool {
    // TODO 1: Check if we have a "trace end" marker span
    //   - Look for span with attribute "trace.end" = true

    // TODO 2: Check timeout - if first span is older than maxTraceDuration
    //   - If time.Since(firstSpanTime) > ta.maxTraceDuration, return true
```

```
    // TODO 3: For advanced implementation: check expected span count
    //   - Some tracing systems include "total_spans" in root span
    //   - Compare len(spans) with expected count

    // TODO 4: Apply watermark algorithm
    //   - Find latest timestamp among all spans
    //   - If currentTime - latestTimestamp > completionWindow, return true

    return false // Remove this line after implementation
}
```

Buffer Manager with Eviction Policy:

```go
// internal/collector/buffer/memory/manager.go

package memory

import (
    "context"
    "time"

    "github.com/your-org/apm-tracing/pkg/models"
)

type BufferManager struct {
    maxSize        int
    ttl            time.Duration
    traces         map[string]*TraceBuffer
    accessTimes    map[string]time.Time
    evictionPolicy string // "lru", "ttl", "fifo"
}

type TraceBuffer struct {
    spans      []models.Span
    firstSeen  time.Time
    lastSeen   time.Time
    isComplete bool
}

func NewBufferManager(maxSize int, ttl time.Duration, evictionPolicy string) *BufferManager {
    return &BufferManager{
        maxSize:        maxSize,
        ttl:            ttl,
        traces:         make(map[string]*TraceBuffer),
        accessTimes:    make(map[string]time.Time),
        evictionPolicy: evictionPolicy,
    }
}
```

```go
// AddSpan adds a span to the buffer for its trace
func (bm *BufferManager) AddSpan(ctx context.Context, span models.Span) error {
    // TODO 1: Get or create TraceBuffer for this trace ID
    //   - Check if traceID exists in bm.traces
    //   - If not, create new TraceBuffer with firstSeen = time.Now()

    // TODO 2: Add span to the buffer's spans slice
    //   - Append the span
    //   - Update lastSeen to current time

    // TODO 3: Update access times for eviction tracking
    //   - Set bm.accessTimes[traceID] = time.Now()

    // TODO 4: Check if buffer is full (len(bm.traces) >= bm.maxSize)
    //   - If full, trigger eviction (call bm.evict(ctx))

    // TODO 5: Check TTL for this trace
    //   - If time.Since(firstSeen) > bm.ttl, mark for eviction

    return nil // Remove this line after implementation
}

// evict removes traces based on the configured eviction policy
func (bm *BufferManager) evict(ctx context.Context) (int, error) {
    // TODO 1: Based on bm.evictionPolicy, select traces to evict
    //   - If "lru": find trace with oldest access time
    //   - If "ttl": find traces where time.Since(firstSeen) > ttl
    //   - If "fifo": find trace with oldest firstSeen

    // TODO 2: For selected traces:
    //   - Write incomplete traces to disk (partial storage)
    //   - Remove from bm.traces map
    //   - Remove from bm.accessTimes map
```

```
    // TODO 3: Return count of evicted traces


    return 0, nil // Remove this line after implementation
}

// GetTraceSpans returns all spans for a given trace ID
func (bm *BufferManager) GetTraceSpans(ctx context.Context, traceID string) ([]models.Span, error) {
    // TODO 1: Look up trace in bm.traces
    //   - Return error if not found


    // TODO 2: Update access time (for LRU policy)
    //   - bm.accessTimes[traceID] = time.Now()


    // TODO 3: Return copy of spans slice (not the original)


    return nil, nil // Remove this line after implementation
}
```

E. Language-Specific Hints

1. **Concurrency in Go**: Use `sync.RWMutex` for protecting the buffer index. Readers (span lookups) use `RLock()`, writers (adding spans) use `Lock()`.
2. **Context Propagation**: Always pass `context.Context` through the pipeline. Use `ctx.Value()` for request-scoped data like request ID, and check `ctx.Err()` for cancellation.
3. **Error Handling**: Use Go 1.13+ error wrapping: `fmt.Errorf("failed to write span: %w", err)`. Create sentinel errors for expected failures (e.g., `ErrTraceNotFound`).
4. **Performance**: Pre-allocate slices with expected capacity: `spans := make([]models.Span, 0, estimatedSpanCount)` to avoid repeated reallocations.
5. **Metrics**: Use Prometheus counters and histograms:

```go
spansReceived := prometheus.NewCounterVec(...)
processingDuration := prometheus.NewHistogramVec(...)
```

6. **Graceful Shutdown**: Use `signal.NotifyContext` to handle SIGTERM:

```go
ctx, stop := signal.NotifyContext(context.Background(), os.Interrupt, syscall.SIGTERM)
defer stop()
```

F. Milestone Checkpoint

To verify Trace Collection is working correctly:

1. **Start the collector**:

```
go run cmd/collector/main.go --config config/local.yaml
```

Expected output: `Starting collector on :4317 (gRPC) and :4318 (HTTP)`

2. **Send test spans via HTTP**:

```
curl -X POST http://localhost:4318/v1/traces \
  -H "Content-Type: application/json" \
  -d '[{
    "traceId": "4bf92f3577b34da6a3ce929d0e0e4736",
    "spanId": "00f067aa0ba902b7",
    "name": "/api/users",
    "kind": 2,
    "startTimeUnixNano": 1505855794194000,
    "endTimeUnixNano": 1505855794195000,
    "attributes": [{"key": "http.method", "value": {"stringValue": "GET"}}]
  }]'
```

BASH

Expected response: `HTTP/2 202 Accepted`

3. **Query stored trace**:

```
curl http://localhost:4318/api/traces/4bf92f3577b34da6a3ce929d0e0e4736
```

BASH

Should return the complete trace with all spans.

4. **Run validation tests**:

```
go test ./internal/collector/pipeline/stages/... -v
```

All tests should pass, including:

- Test span validation rejects malformed data
- Test trace assembly builds correct hierarchy
- Test buffer eviction works as expected

5. **Check metrics** (if Prometheus endpoint enabled):

```
curl http://localhost:9090/metrics | grep collector_
```

Should see metrics like `collector_spans_received_total`, `collector_buffer_size`.

Signs something is wrong:

- **Spans not linking**: Check parentSpanID format and validation logic
- **High memory usage**: Check buffer eviction is working, inspect with `go tool pprof`
- **Slow ingestion**: Check for blocking operations in pipeline, use Go's race detector
- **Data loss on restart**: Verify WAL is being fsync'd and recovery reads all records

---

6. Component Design: Service Map (Milestone 2)

> **Milestone(s):** This section corresponds to Milestone 2: Service Map, which analyzes trace data to build and visualize dynamic service dependency graphs.

Mental Model: The Social Network of Services

Think of your distributed system as a **social network of services**, where each service is a person and each service call is a friendship interaction. The service map is the "friend graph" that shows who talks to whom, how often they communicate, and the quality of their relationships. Just as you could learn about social dynamics by observing who messages whom and how frequently, we learn about system architecture by observing which services call each other and with what latency and success rates.

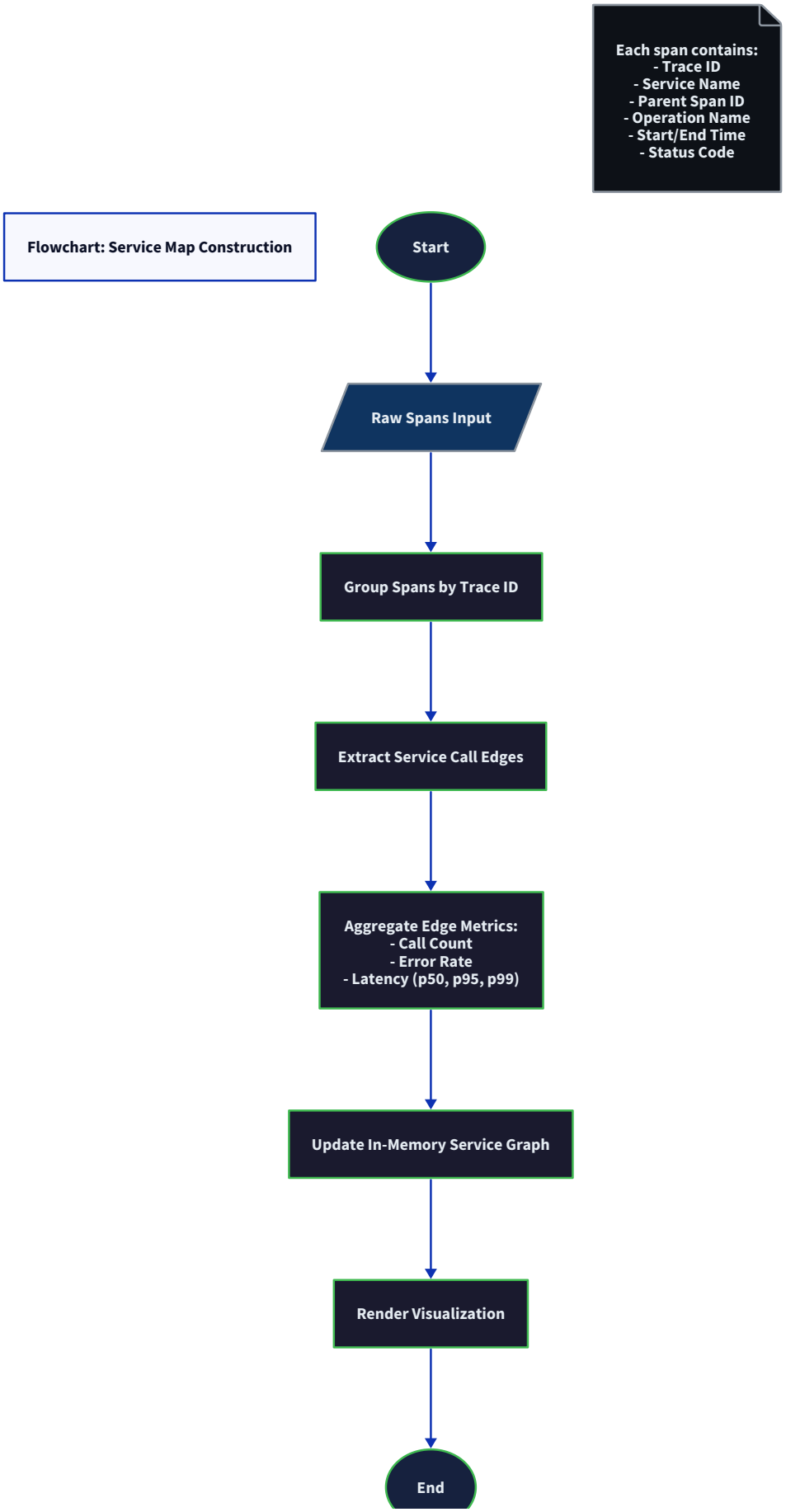
This mental model helps clarify several key concepts:

1. **Nodes as services**: Each service is a node in the graph, similar to each person in a social network
2. **Edges as relationships**: A call from Service A to Service B creates a directed edge (A→B), similar to A sending a message to B
3. **Edge weight as interaction frequency**: The thickness of the edge represents call volume, just as thicker lines between people might represent more frequent communication
4. **Edge color as relationship health**: Red edges indicate high error rates (problematic relationships), while green edges indicate healthy interactions
5. **Clusters as architectural patterns**: Services that frequently call each other form clusters, revealing microservice boundaries or bounded contexts

Just as social networks evolve over time with new friendships forming and old ones fading, your service map must detect and visualize topology changes as services are added, removed, or their communication patterns change.

Algorithm: Building the Graph from Spans

Building an accurate service map requires analyzing spans to extract service call relationships, then aggregating metrics over those relationships. The algorithm operates in three phases: **extraction**, **aggregation**, and **refresh**. The following flowchart illustrates this process:


Each span contains:
- Trace ID
- Service Name
- Parent Span ID
- Operation Name
- Start/End Time
- Status Code
Flowchart: Service Map Construction
Start
Raw Spans Input
Group Spans by Trace ID
Extract Service Call Edges
Aggregate Edge Metrics:
- Call Count
- Error Rate
- Latency (p50, p95, p99)
Update In-Memory Service Graph
Render Visualization
End

Phase 1: Edge Extraction from Individual Spans

For each incoming span, we extract potential service call edges by examining parent-child relationships:

1. **Filter relevant spans**: Only consider spans with a non-empty `ParentSpanID` (excluding root spans) and with `ServiceName` populated
2. **Retrieve parent span**: Using the span's `TraceID` and `ParentSpanID`, fetch the parent span from storage or buffer
3. **Identify caller-callee relationship**:
   - **Caller**: The `ServiceName` of the parent span
   - **Callee**: The `ServiceName` of the child span
4. **Validate edge**: Ensure caller ≠ callee (ignore same-service spans for the service map, as they represent internal operations)
5. **Extract operation context**: Record the `Name` field (operation) of both spans for detailed metrics
6. **Capture timing information**: Calculate the **inter-service latency** as: `child.StartTime - parent.StartTime` (adjusting for any clock skew using techniques from Milestone 1)

This phase produces raw edges with the following structure:

Field	Type	Description
`CallerService`	`string`	Name of the service making the call
`CalleeService`	`string`	Name of the service receiving the call
`CallerOperation`	`string`	Operation/endpoint name on the caller side
`CalleeOperation`	`string`	Operation/endpoint name on the callee side
`TraceID`	`string`	Identifier of the trace containing this call
`SpanID`	`string`	Identifier of the child span (the call)
`StartTime`	`time.Time`	When the call began
`Duration`	`time.Duration`	How long the call took
`IsError`	`bool`	Whether the span status indicates an error (based on `SpanStatus.Code`)
`Attributes`	`map[string]string`	Additional attributes that might affect edge grouping

Phase 2: Metric Aggregation over Time Windows

Instead of tracking every individual call, we aggregate metrics over configurable time windows (e.g., 1 minute, 5 minutes, 1 hour) to create summarized views:

1. **Group edges by (caller, callee) pair**: All calls from Service A to Service B within the same time window are aggregated together
2. **Calculate aggregated metrics** for each edge:
   - **Total Calls**: Count of all calls in the window

- **Error Count**: Count of calls where `IsError = true`
- **Error Rate**: `Error Count / Total Calls`
- **Latency Percentiles**: p50, p95, p99 latencies calculated using the t-digest algorithm (see Milestone 4)
- **Total Bytes Transferred** (if available from span attributes)

3. **Maintain rolling windows**: Keep multiple time windows (current, previous) to enable trend analysis (e.g., "error rate increased by 15% compared to last hour")
4. **Apply dimension reduction**: Optionally group by operation patterns if there are too many distinct operations causing high cardinality

The aggregated edge structure becomes:

Field	Type	Description
`CallerService`	`string`	Name of the service making the call
`CalleeService`	`string`	Name of the service receiving the call
`WindowStart`	`time.Time`	Start of the aggregation window
`WindowEnd`	`time.Time`	End of the aggregation window
`TotalCalls`	`int64`	Number of calls in this window
`ErrorCount`	`int64`	Number of failed calls
`ErrorRate`	`float64`	`ErrorCount / TotalCalls`
`P50Latency`	`time.Duration`	50th percentile latency
`P95Latency`	`time.Duration`	95th percentile latency
`P99Latency`	`time.Duration`	99th percentile latency
`SampleTraceIDs`	`[]string`	A small sample of trace IDs (e.g., 5) representing this edge for drill-down

Phase 3: Graph Construction and Topology Detection

With aggregated edges, we build the service dependency graph:

1. **Create graph nodes**: Each unique service name becomes a node in the graph
2. **Create directed edges**: Each unique (caller, callee) pair becomes a directed edge
3. **Apply edge weights**: Visual thickness corresponds to `TotalCalls` (logarithmic scale to handle wide ranges)
4. **Color coding**:
   - Green: `ErrorRate < 1%`
   - Yellow: `1% ≤ ErrorRate < 5%`
   - Red: `ErrorRate ≥ 5%`
5. **Detect topology changes**:
   - **New service**: A service appears in the trace data that wasn't in the previous graph
   - **Removed service**: A service hasn't been seen in the last N time windows (configurable timeout)
   - **New dependency**: An edge appears between existing services where none existed before
   - **Dependency removed**: An edge hasn't been observed in the last M time windows
6. **Detect cyclic dependencies**: Use depth-first search to identify cycles in the directed graph, which may indicate problematic architectural patterns

The complete service map refresh algorithm runs periodically (e.g., every 30 seconds) and processes spans that arrived since the last refresh.

ADR: Graph Storage - Materialized vs. On-Demand

> **Decision: Materialized Graph with Incremental Updates**
>
> - **Context**: The service map needs to support both real-time visualization (updating every few seconds) and historical queries (showing how dependencies changed over hours/days). We must decide whether to pre-compute and store the graph (materialized) or compute it on-demand from raw span data.
> - **Options Considered**:
>   1. **On-Demand Computation**: Query raw spans from storage each time the service map is requested, compute the graph in memory, and return it.
>   2. **Materialized Graph with Full Rebuild**: Pre-compute the graph periodically (e.g., every minute), store it in a dedicated graph database, and serve queries from this materialized view.
>   3. **Materialized Graph with Incremental Updates**: Maintain an in-memory aggregated view of edges, flush snapshots to storage periodically, and update incrementally as new spans arrive.
> - **Decision**: Option 3 - Materialized Graph with Incremental Updates.
> - **Rationale**:
>   - **Latency requirements**: The service map UI should update near-real-time (sub-second), which on-demand computation cannot guarantee as span volumes grow.
>   - **Resource efficiency**: Incremental updates amortize computation cost, whereas full rebuilds wastefully recompute unchanged portions of the graph.
>   - **Query flexibility**: Materialized edges with time windows enable both current-state and historical trend queries without scanning all raw spans.
>   - **Resilience**: In-memory aggregated state can be reconstructed from recent span data if the process restarts, providing a good balance between performance and durability.
> - **Consequences**:
>   - We must implement careful memory management for the in-memory aggregated state.
>   - The system maintains eventual consistency - there's a brief delay (window size) before new dependencies appear.
>   - Historical queries are limited to the retention period of aggregated edges, not raw spans.

The following table compares the options:

Option	Pros	Cons	Why Not Chosen
On-Demand Computation	Always accurate (no stale data), No additional storage needed, Simple implementation	Slow for large datasets, High load on span storage, Doesn't scale with request volume	Unacceptable latency for real-time visualization (seconds to minutes for large deployments)
Materialized Graph with Full Rebuild	Consistent view at rebuild time, Can use optimized graph databases, Clean separation of concerns	High CPU usage during rebuilds, Stale data between rebuilds, Wastes resources recomputing unchanged portions	The periodic full rebuilds create resource spikes and stale periods
Materialized Graph with Incremental Updates	Near-real-time updates, Efficient resource usage, Enables both current and historical views	More complex implementation, Requires memory management, Eventual consistency	**CHOSEN**: Best balance of performance, accuracy, and resource efficiency

Common Pitfalls in Service Map Construction

⚠️ Pitfall 1: Misattributing Async and Concurrent Calls

Description: When Service A makes concurrent calls to Service B (e.g., fan-out pattern) or async calls (fire-and-forget), the simple parent-child relationship extraction can misinterpret the dependency direction or create misleading metrics.

Why it's wrong: If Service A calls Service B 10 times concurrently within a single request, counting these as 10 separate dependencies exaggerates the importance of the A→B edge. Async calls might not have proper parent-child linking, causing them to appear as separate, disconnected traces.

How to fix:

- **Group concurrent calls**: When multiple spans share the same parent and have similar start times (within a small threshold, e.g., 10ms), treat them as a single logical operation for dependency analysis.
- **Trace context propagation**: Ensure async operations propagate the trace context properly (see Milestone 5) to maintain parent-child relationships.
- **Attribute tagging**: Use span attributes (e.g., `call.type=async`) to identify and handle special call patterns differently.

⚠️ Pitfall 2: High Cardinality from Operation Names

Description: When services have hundreds or thousands of distinct operation names (endpoints), creating separate edges for each (caller, callee, caller-op, callee-op) tuple explodes the edge count, making the graph unreadable and storage requirements unsustainable.

Why it's wrong: A service map with 10,000 edges is visually useless and computationally expensive. The core value is understanding service-to-service dependencies, not endpoint-to-endpoint details.

How to fix:

- **Two-level aggregation**: First aggregate by (caller, callee) pair only, ignoring operation names for the main visualization.
- **Drill-down views**: Provide click-through from service edges to see operation-level breakdowns.
- **Operation grouping**: Use pattern matching or configuration to group similar operations (e.g., `/users/*` → `/users/:id`).
- **Cardinality limits**: Implement hard limits (e.g., top 20 operations per edge) and aggregate the rest as "other".

⚠️ Pitfall 3: Missing or Transient Dependencies

Description: Infrequently called services (e.g., nightly batch jobs, error-handling services) may not appear in the service map if the aggregation window is too short or sampling rate is too aggressive.

Why it's wrong: The service map should show all dependencies, not just frequent ones. Missing dependencies give a false sense of simplicity and can hide important failure modes.

How to fix:

- **Longer retention for edges**: Keep edge aggregates for longer periods (e.g., 24-48 hours) even if they have low call volumes.
- **Sampling protection**: Ensure sampling algorithms (Milestone 3) don't systematically drop traces from low-volume services.
- **Configuration-driven inclusion**: Allow operators to explicitly list services that must appear regardless of call frequency.
- **Periodic full scans**: Occasionally scan older span data to rediscover dormant dependencies.

⚠️ Pitfall 4: Incorrect Cycle Detection in Directed Graphs

Description: Naive cycle detection algorithms can misidentify legitimate call patterns (like callback patterns, saga orchestrators) as problematic cycles, or miss actual problematic cycles due to timing issues.

Why it's wrong: False positives lead to unnecessary alerts and investigation. False negatives miss actual architectural issues like circular dependencies that cause deadlocks or infinite loops.

How to fix:

- **Time-bound cycle detection**: Only consider cycles where all calls occur within a single request/trace context, not across different requests.
- **Distinguish call patterns**: Use span attributes and operation names to identify legitimate patterns like callbacks vs. accidental cycles.
- **Multi-request cycle detection**: For cycles spanning multiple requests/traces, use statistical correlation of call patterns over longer time windows.
- **Visual highlighting vs. alerting**: Highlight potential cycles in the visualization for human investigation, but only alert on statistically significant patterns.

Implementation Guidance for Service Map

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
In-Memory Graph	Go's native maps and slices with manual adjacency lists	Dedicated graph library: `gonum/graph`
Time-Series Aggregates	Rolling windows in memory with periodic persistence to Redis/PostgreSQL	Dedicated time-series database: InfluxDB or TimescaleDB
Real-Time Updates	Periodic batch processing (e.g., every 30s) with goroutine	Streaming pipeline with channels and worker pools
Visualization Backend	Static JSON API served to frontend with basic D3.js	WebSocket stream of graph diffs with interactive frontend
Edge Storage	PostgreSQL with (caller, callee, window_start) composite index	Redis Sorted Sets for time-windowed aggregates

B. Recommended File/Module Structure

```
project-root/
  cmd/
    servicemap/           # Service map microservice entry point
      main.go
  internal/
    servicemap/           # Service map core logic
      aggregator.go       # Edge extraction and metric aggregation
      graph_builder.go    # Graph construction from aggregated edges
      storage.go          # Interface for edge storage
      types.go            # Service map specific types
    servicemap/redis/     # Redis storage implementation
      edge_store.go
    servicemap/postgres/  # PostgreSQL storage implementation
      edge_store.go
    servicemap/memory/    # In-memory storage (for development)
      edge_store.go
  pkg/
    models/               # Shared data models (already defined)
      trace.go            # Contains Span, Trace types
      servicemap.go       # Service map specific types
```

C. Infrastructure Starter Code

Complete Edge Storage Interface and In-Memory Implementation:

```go
// internal/servicemap/types.go

package servicemap

import (
    "time"
)

// ServiceEdge represents an aggregated edge between two services
type ServiceEdge struct {
    CallerService  string        `json:"caller_service"`
    CalleeService  string        `json:"callee_service"`
    WindowStart    time.Time     `json:"window_start"`
    WindowEnd      time.Time     `json:"window_end"`
    TotalCalls     int64         `json:"total_calls"`
    ErrorCount     int64         `json:"error_count"`
    ErrorRate      float64       `json:"error_rate"`
    P50Latency     time.Duration `json:"p50_latency"`
    P95Latency     time.Duration `json:"p95_latency"`
    P99Latency     time.Duration `json:"p99_latency"`
    SampleTraceIDs []string      `json:"sample_trace_ids"`
}

// ServiceNode represents a node (service) in the graph
type ServiceNode struct {
    Name       string    `json:"name"`
    FirstSeen  time.Time `json:"first_seen"`
    LastSeen   time.Time `json:"last_seen"`
    TotalCalls int64     `json:"total_calls"`  // Sum of all outgoing calls
}

// ServiceGraph represents the complete service dependency graph
type ServiceGraph struct {
    Nodes        []ServiceNode          `json:"nodes"`
    Edges        []ServiceEdge          `json:"edges"`
    GeneratedAt  time.Time              `json:"generated_at"`
```

```
    WindowSize   time.Duration          `json:"window_size"`
}

// EdgeStorage defines the interface for storing and retrieving aggregated edges
type EdgeStorage interface {
    // StoreEdge stores or updates an aggregated edge
    StoreEdge(ctx context.Context, edge ServiceEdge) error

    // GetEdges returns edges for the given time window
    GetEdges(ctx context.Context, startTime, endTime time.Time) ([]ServiceEdge, error)

    // GetServiceEdges returns edges for a specific service (as caller or callee)
    GetServiceEdges(ctx context.Context, serviceName string, startTime, endTime time.Time) ([]ServiceEdge,
error)

    // GetEdgeHistory returns historical aggregates for a specific edge
    GetEdgeHistory(ctx context.Context, caller, callee string, lookbackPeriod time.Duration) ([]ServiceEdge,
error)

    // Cleanup removes edges older than the specified duration
    Cleanup(ctx context.Context, olderThan time.Time) error
}

// internal/servicemap/memory/edge_store.go
package memory

import (
    "context"
    "sort"
    "sync"
    "time"

    "github.com/yourproject/internal/servicemap"
)
```

```go
type MemoryEdgeStore struct {
    mu     sync.RWMutex
    edges  []servicemap.ServiceEdge
    maxAge time.Duration
}

func NewMemoryEdgeStore(maxAge time.Duration) *MemoryEdgeStore {
    store := &MemoryEdgeStore{
        edges:  make([]servicemap.ServiceEdge, 0),
        maxAge: maxAge,
    }

    // Start cleanup goroutine
    go store.periodicCleanup(context.Background())

    return store
}

func (s *MemoryEdgeStore) StoreEdge(ctx context.Context, edge servicemap.ServiceEdge) error {
    s.mu.Lock()
    defer s.mu.Unlock()

    // Check if edge already exists for this window
    for i, existing := range s.edges {
        if existing.CallerService == edge.CallerService &&
           existing.CalleeService == edge.CalleeService &&
           existing.WindowStart.Equal(edge.WindowStart) {
            // Update existing edge
            s.edges[i] = edge
            return nil
        }
    }
```

```
    // Add new edge
    s.edges = append(s.edges, edge)
    return nil
}

func (s *MemoryEdgeStore) GetEdges(ctx context.Context, startTime, endTime time.Time)
([]servicemap.ServiceEdge, error) {
    s.mu.RLock()
    defer s.mu.RUnlock()

    var result []servicemap.ServiceEdge
    for _, edge := range s.edges {
        if !edge.WindowStart.Before(startTime) && !edge.WindowEnd.After(endTime) {
            result = append(result, edge)
        }
    }

    return result, nil
}

func (s *MemoryEdgeStore) periodicCleanup(ctx context.Context) {
    ticker := time.NewTicker(5 * time.Minute)
    defer ticker.Stop()

    for {
        select {
        case <-ctx.Done():
            return
        case <-ticker.C:
            cutoff := time.Now().Add(-s.maxAge)
            s.mu.Lock()
            filtered := s.edges[:0]
            for _, edge := range s.edges {
                if edge.WindowEnd.After(cutoff) {
```

```
                    filtered = append(filtered, edge)

                }

            }

            s.edges = filtered

            s.mu.Unlock()

        }

    }

}


// Implement other EdgeStorage methods...
```

D. Core Logic Skeleton Code

Edge Aggregator with TODOs:

GO

```go
// internal/servicemap/aggregator.go

package servicemap

import (
    "context"
    "time"

    "github.com/yourproject/pkg/models"
)

// EdgeAggregator extracts service call edges from spans and aggregates metrics
type EdgeAggregator struct {
    edgeStorage  EdgeStorage
    windowSize   time.Duration
    currentWindowStart time.Time
    pendingEdges map[string]*ServiceEdge // Key: "caller:callee:window_start"

    // TODO: Add fields for t-digest instances for latency percentiles
    // TODO: Add metrics for monitoring the aggregator itself
}

// NewEdgeAggregator creates a new aggregator with the specified window size
func NewEdgeAggregator(storage EdgeStorage, windowSize time.Duration) *EdgeAggregator {
    return &EdgeAggregator{
        edgeStorage:  storage,
        windowSize:   windowSize,
        pendingEdges: make(map[string]*ServiceEdge),
    }
}

// ProcessSpan extracts edges from a span and updates aggregations
func (a *EdgeAggregator) ProcessSpan(ctx context.Context, span models.Span) error {
    // TODO 1: Skip if span has no ParentSpanID (it's a root span)
    // TODO 2: Skip if span.ServiceName is empty
```

```
    // TODO 3: Retrieve parent span using GetTraceByID and ParentSpanID
    //         (Note: This assumes parent span is already stored - consider buffering)

    // TODO 4: If parent span not found, buffer this span for later retry
    //         or use asynchronous parent resolution

    // TODO 5: Extract caller service from parentSpan.ServiceName
    //         Extract callee service from span.ServiceName

    // TODO 6: Skip if caller == callee (same-service calls)

    // TODO 7: Determine which time window this span belongs to
    //         windowStart := span.StartTime.Truncate(a.windowSize)

    // TODO 8: Create or update aggregated edge for (caller, callee, windowStart)
    //         key := fmt.Sprintf("%s:%s:%d", caller, callee, windowStart.Unix())
    //         edge := a.pendingEdges[key]
    //         if edge == nil {
    //             edge = &ServiceEdge{CallerService: caller, CalleeService: callee, ...}
    //             a.pendingEdges[key] = edge
    //         }

    // TODO 9: Update edge metrics:
    //         edge.TotalCalls++
    //         if span.Status.Code >= 400 { edge.ErrorCount++ } // Adjust error detection
    //         Update t-digest with span.Duration for latency percentiles
    //         Maintain sample of trace IDs (keep up to 5 unique ones)

    // TODO 10: If we've reached the end of the current window, flush all pending edges
    //          if windowStart != a.currentWindowStart && a.currentWindowStart != (time.Time{}) {
    //              a.FlushWindow(ctx, a.currentWindowStart)
    //          }
```

```go
    return nil
}

// FlushWindow persists all aggregated edges for a specific window to storage
func (a *EdgeAggregator) FlushWindow(ctx context.Context, windowStart time.Time) error {
    // TODO 1: For each edge in a.pendingEdges with matching windowStart:
    // TODO 2: Calculate final metrics (ErrorRate = ErrorCount/TotalCalls)
    // TODO 3: Calculate percentiles from t-digest
    // TODO 4: Store edge via a.edgeStorage.StoreEdge
    // TODO 5: Remove flushed edges from pendingEdges map
    // TODO 6: Update a.currentWindowStart to the next window
    return nil
}
```

Graph Builder with TODOs:

```go
// internal/servicemap/graph_builder.go

package servicemap


import (

    "context"

    "time"

)


// GraphBuilder constructs service graphs from aggregated edges

type GraphBuilder struct {

    edgeStorage EdgeStorage

}


// BuildCurrentGraph builds a service graph for the most recent complete time window

func (b *GraphBuilder) BuildCurrentGraph(ctx context.Context, windowSize time.Duration) (*ServiceGraph, error)
{

    // TODO 1: Calculate time window for the most recent complete window

    //         now := time.Now()

    //         endTime := now.Truncate(windowSize)

    //         startTime := endTime.Add(-windowSize)



    // TODO 2: Retrieve edges for this window using b.edgeStorage.GetEdges



    // TODO 3: Extract unique service names from edges (both caller and callee)



    // TODO 4: Create ServiceNode for each unique service

    //         For each node, calculate total outgoing calls by summing edges where service is caller



    // TODO 5: Build ServiceGraph with nodes and edges



    // TODO 6: Detect and mark potential issues:

    //         - Services with no incoming edges (entry points)

    //         - Services with no outgoing edges (leaf nodes)

    //         - Potential cycles (use DFS or topological sort)

    //         - Edges with high error rates (> threshold)
```

```
    return nil, nil
}

// DetectTopologyChanges compares current graph with previous graph to find changes
func (b *GraphBuilder) DetectTopologyChanges(ctx context.Context, currentGraph, previousGraph *ServiceGraph)
*TopologyChanges {
    // TODO 1: Compare nodes to detect:
    //         - New services (in current but not previous)
    //         - Removed services (in previous but not current)

    // TODO 2: Compare edges to detect:
    //         - New dependencies
    //         - Removed dependencies

    // TODO 3: For existing edges, detect significant metric changes:
    //         - Error rate increased by more than X%
    //         - Call volume changed by more than Y%
    //         - Latency percentiles changed significantly

    // TODO 4: Return structured TopologyChanges object
    return nil
}
```

E. Language-Specific Hints

1. **Concurrent Map Access**: Use `sync.RWMutex` for the `pendingEdges` map in the aggregator, or consider `sync.Map` if you have a write-once, read-many pattern.
2. **Time Window Calculations**: Use `time.Time.Truncate()` for clean window boundaries: `windowStart := timestamp.Truncate(5 * time.Minute)`.
3. **Efficient Edge Lookups**: Create composite keys for the pending edges map using string concatenation with a delimiter: `key := fmt.Sprintf("%s|%s|%d", caller, callee, windowStart.Unix())`.
4. **Background Flushing**: Use a `time.Ticker` in a goroutine for periodic window flushes:

```go
go func() {
    ticker := time.NewTicker(flushInterval)
    defer ticker.Stop()
    for {
        select {
        case <-ctx.Done():
            return
        case <-ticker.C:
            aggregator.FlushWindow(ctx, currentWindow)
        }
    }
}()
```

5. **Graph Algorithms**: For cycle detection, implement depth-first search with coloring (white/gray/black nodes) to detect back edges efficiently.
6. **Memory Management**: Use pointers in slices for edges and nodes to avoid large data copies when building graphs.

F. Milestone Checkpoint

Verification Steps:

1. **Start the system**: Run the trace collector (from Milestone 1) and the service map aggregator.
2. **Generate traffic**: Use the provided test harness to simulate calls between 3-5 mock services.
3. **Check edge extraction**: Query the service map API after 1-2 minutes, you should see:
   - Nodes for each mock service
   - Directed edges showing which service calls which
   - Basic metrics (call count) on each edge
4. **Verify real-time updates**: Add a new mock service that calls an existing one, wait for the aggregation window (e.g., 1 minute), and confirm the graph updates.
5. **Test error visualization**: Configure one mock service to return errors 20% of the time, verify the corresponding edge turns yellow/red in the visualization.

Expected API endpoints:

- `GET /api/servicemap/current` - Returns current service graph as JSON
- `GET /api/servicemap/history?service=A&hours=24` - Returns historical edges for service A
- `GET /api/servicemap/changes?since=2024-01-01T00:00:00Z` - Returns topology changes since timestamp

Signs of problems:

- **No edges appear**: Check that spans have proper parent-child relationships and `ServiceName` is populated
- **High memory usage**: The pending edges map might not be getting flushed; check the flush timer
- **Stale data**: The aggregation window might be too long; reduce from default 5 minutes to 1 minute for testing
- **Missing services**: Low-volume services might be filtered out; adjust the minimum call threshold

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Service map shows no edges	Spans missing parent-child links or `ServiceName` fields	Check raw span data in storage; verify `ParentSpanID` is set and points to valid span	Ensure SDK (Milestone 5) properly sets parent-child relationships and service names
Edges appear/disappear randomly	Aggregation window too short or sampling dropping traces	Check if total calls per edge are very low (< 10 per window)	Increase aggregation window size or adjust sampling to keep more traces
High memory usage in aggregator	Too many pending edges or window not flushing	Monitor `len(pendingEdges)`; check if flush goroutine is running	Implement edge eviction for old windows; ensure flush timer is working
Cycles detected in healthy system	Callback patterns misidentified as cycles	Examine the specific edges forming the cycle; check if calls are within same trace	Implement time-bound cycle detection or tag callback patterns in spans
Visualization overloaded with edges	Operation-level granularity instead of service-level	Count distinct (caller, callee, operation) tuples	Aggregate at service level only; provide operation breakdown on click
Service map doesn't update in real-time	Processing pipeline too slow or batched	Measure latency from span arrival to edge appearance in API	Optimize parent span lookup; consider async processing; reduce batch sizes

7. Component Design: Trace Sampling (Milestone 3)

> **Milestone(s):** This section corresponds to Milestone 3: Trace Sampling, which designs the intelligent filtering mechanism to reduce storage costs while preserving valuable traces for debugging and analysis.

Mental Model: The Library Archivist

Imagine you're an archivist in a vast library receiving thousands of books every hour, but your storage shelves have limited capacity. You cannot possibly keep every book that arrives, yet you must ensure that:

1. **Rare and unique works** are preserved (error traces, anomalies)
2. **Representative samples** of popular genres are kept (normal traffic patterns)
3. **Important historical records** are maintained (high-latency operations, critical paths)
4. **The collection remains useful** for researchers (debuggers, SREs) to understand the library's contents

This is exactly the challenge of **trace sampling** in a high-volume APM system. Every request through your distributed system produces a trace (a "book"), but storing all traces would require impractical amounts of storage. The sampling subsystem acts as the intelligent archivist, making real-time decisions about which traces to preserve based on configurable policies, while ensuring that the retained traces form a statistically useful dataset for debugging performance issues and understanding system behavior.

The archivist faces two critical decisions:

- **Immediate filtering (head-based)**: As books arrive at the loading dock, quickly decide which ones to even bring inside based on cover information (trace ID, service name).
- **Retrospective evaluation (tail-based)**: After examining a book's contents (completed trace), decide if it's valuable enough to override an earlier rejection.

Head-Based vs. Tail-Based Sampling

Sampling strategies differ fundamentally in **when** the sampling decision is made relative to when the trace completes. Each approach has distinct trade-offs that make it suitable for different parts of the pipeline.

Head-Based Sampling: The First Impression

Head-based sampling makes its keep/drop decision **at the very beginning of a trace**, when the root span is first created and before any downstream spans are generated. This decision is made locally by the SDK/agent when it starts tracing a request.

> **Key Insight:** Head-based sampling is like deciding whether to film a documentary based on the title alone—you save resources by not filming at all, but you might miss incredible footage that only reveals itself later.

Aspect	Description
Decision Point	At trace creation (when root span starts)
Information Available	Only initial context: trace ID, service name, operation name, possibly sampling configuration
Decision Scope	Applies to the entire trace—once dropped, no spans are collected
Performance Impact	Minimal overhead on downstream services (no context propagation for dropped traces)
Storage Savings	Maximum possible reduction (entire traces never enter the system)

The decision algorithm for head-based sampling typically uses **consistent hashing** on the trace ID to ensure all participants make the same deterministic decision:

1. Hash the trace ID to a value between 0 and 1 (e.g., using MurmurHash3)
2. Compare against configured sampling probability for the service/operation
3. If hash ≤ probability, keep the trace; otherwise, drop it

```
decision = hash(trace\_id) ≤ sampling\_probability(service, operation)
```
MATH

This approach ensures **consistency**—all spans for the same trace get the same decision, avoiding partial traces. The `SamplingConfig` structure supports this with both global (`Probability`) and per-service (`PerService`) rates.

Tail-Based Sampling: The Retrospective Review

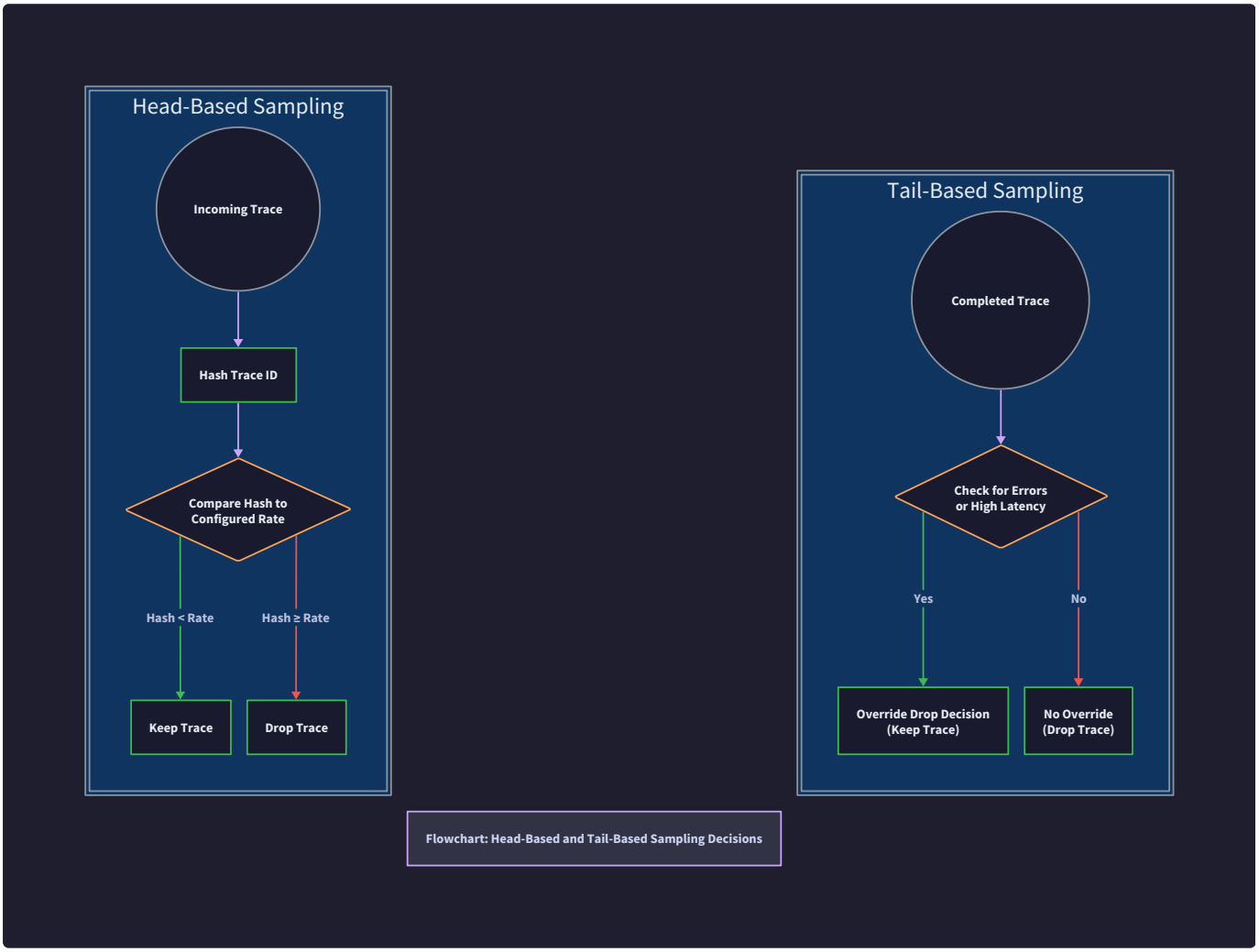
Tail-based sampling evaluates traces **after they complete** (or after a significant portion has been collected). This allows decisions based on complete information about the trace's characteristics: duration, error status, specific operations performed, etc.

> **Key Insight:** Tail-based sampling is like a film editor reviewing all footage shot during the day and deciding which clips to keep for the final documentary—more resource-intensive but ensures you don't miss critical scenes.

Aspect	Description
Decision Point	After trace completion (or after a timeout)
Information Available	Complete trace data: all spans, durations, errors, attributes
Decision Scope	Can override head-based decisions to keep interesting traces
Performance Impact	Higher resource usage (collect all spans first, then evaluate)
Storage Savings	Selective—keeps only traces matching defined criteria

Tail-based sampling operates as a **second-chance mechanism**:

1. All spans for a trace are collected (potentially after head-based sampling)
2. When the trace completes or times out, evaluate against criteria
3. If criteria match, ensure all spans are persisted; otherwise, discard



Comparison Table: When to Use Each Strategy

Strategy	Best For	Not Suitable For
Head-Based	High-volume production traffic, cost reduction, predictable sampling rates	Debugging rare errors, investigating latency spikes, compliance/audit trails
Tail-Based	Error investigation, performance debugging, compliance (keep all errors)	Extremely high-volume systems without sufficient buffering capacity
Combined Approach	Most production systems: head-based for volume reduction + tail-based for error capture	Systems with extremely tight latency budgets at the edge

In our APM system, we implement a **hybrid approach**:

1. **Head-based sampling** at the SDK level reduces initial volume
2. **Tail-based sampling** at the collector level re-evaluates kept traces
3. **Priority channels** ensure error traces bypass sampling when detected early

ADR: Adaptive Sampling Algorithm

Decision: Implement Feedback-Controlled Adaptive Sampling with Per-Service Budgets

- **Context**: Our APM system must handle variable traffic loads while maintaining consistent storage costs. Fixed sampling rates either waste storage during low traffic or lose valuable traces during peaks. We need an algorithm that adjusts sampling rates dynamically based on system load and trace value.
- **Options Considered**:
  1. **Fixed-Rate Sampling**: Constant probability (e.g., 10%) for all services
  2. **Rate-Limiting Sampling**: Maximum spans per second, discarding excess
  3. **Feedback-Controlled Adaptive Sampling**: Dynamically adjusts rates based on ingestion load and trace characteristics
- **Decision**: Implement feedback-controlled adaptive sampling with separate control loops for head-based and tail-based decisions, using per-service budgets to prioritize critical services.
- **Rationale**:
  - Fixed rates cannot adapt to traffic patterns, causing either data loss or storage bloat
  - Rate limiting causes unfair distribution—high-traffic services dominate the quota
  - Feedback control maintains system stability while maximizing trace value
  - Per-service budgets ensure monitoring critical services even during load spikes
- **Consequences**:
  - Adds complexity with control loops and monitoring
  - Requires careful tuning to avoid oscillation
  - Provides optimal trace retention within storage constraints
  - Enables priority sampling for business-critical services

Options Comparison Table

Option	Pros	Cons	Why Not Chosen
Fixed-Rate Sampling	Simple to implement, predictable behavior, easy to reason about	Cannot adapt to traffic changes, either wastes storage or loses data during peaks, one-size-fits-all approach	Too inflexible for production systems with variable loads
Rate-Limiting Sampling	Enforces hard storage limits, prevents system overload, simple conceptually	Unfair to high-traffic services, discards based on arrival order not value, causes sampling bias	Violates principle of preserving valuable traces—discards randomly during congestion
Feedback-Controlled Adaptive	Maximizes trace value within constraints, adapts to traffic patterns, prioritizes important services	Complex implementation, requires tuning, potential for oscillation if poorly designed	**CHOSEN**: Provides optimal balance of value preservation and cost control

Adaptive Sampling Algorithm Design

Our adaptive sampling system uses **separate but coordinated control loops**:

1. Head-Based Adaptive Controller

- **Input**: Current ingestion rate, storage utilization, per-service budgets
- **Output**: Adjusted sampling probabilities per service
- **Algorithm**:
  1. Measure ingestion rate over sliding window (e.g., 30 seconds)

2. Compare against target ingestion capacity (configurable)
3. If above target: reduce sampling probabilities proportionally across services, respecting minimum rates
4. If below target: gradually increase probabilities toward configured maximums
5. Apply emergency reduction if storage exceeds critical threshold

2. Tail-Based Value Scorer

- **Input**: Completed trace characteristics (errors, latency, service mix)
- **Output**: Priority score (0-100) indicating trace value
- **Scoring Factors**:
  - Error presence: +50 points
  - High latency (> p95 for service): +30 points
  - Contains specific operations (configurable): +20 points
  - Rare service combinations: +10 points
- **Decision**: Keep trace if score > threshold (configurable)

3. Per-Service Budget Allocator

- **Input**: Total ingestion capacity, service criticality weights
- **Output**: Maximum spans per second per service
- **Algorithm**:
  - Allocate base budget to all services (e.g., 100 spans/sec)
  - Distribute remaining capacity proportionally to criticality weights
  - Enforce minimum budget for all services (ensures some visibility)
  - Reallocate unused budget from low-traffic to high-traffic services

The control loops run at different frequencies:

- **Head-based adjustments**: Every 10 seconds (slow, stable changes)
- **Tail-based scoring**: Per completed trace (immediate)
- **Budget reallocation**: Every 60 seconds (infrequent redistribution)

State Machine: Sampling Decision Process

The complete sampling workflow can be modeled as a state machine:

Current State	Event	Next State	Actions Taken
New Trace	Root span created	`HeadDecisionPending`	Generate trace ID, extract service/operation
`HeadDecisionPending`	Hash computed	`SampledOut` or `SampledIn`	Apply head-based probability, set sampling flag in context
`SampledOut`	Span propagation	`Dropped`	Do not propagate context, discard locally
`SampledIn`	Span collection	`Collecting`	Propagate context, buffer spans at collector
`Collecting`	Trace timeout reached	`TailEvaluation`	Mark trace complete, trigger evaluation
`Collecting`	All spans received	`TailEvaluation`	Mark trace complete, trigger evaluation
`TailEvaluation`	Score calculated	`FinalKeep` or `FinalDrop`	Apply tail-based rules, persist or discard
`FinalKeep`	Storage acknowledgment	`Archived`	Write to persistent storage, update indexes
`FinalDrop`	Cleanup complete	`Discarded`	Release buffer memory, log statistics

Adaptive Rate Calculation Algorithm

The core adaptive algorithm adjusts sampling probabilities:

1. **Gather Metrics** (every control interval):
   - Current ingestion rate `I_current` (spans/sec)
   - Target ingestion rate `I_target` (from configuration)
   - Storage utilization `U` (0-100%)
   - Per-service traffic rates `R_service[]`
2. **Calculate Adjustment Factor**:

```
error = I_current - I_target
adjustment = -Kp × error - Ki × integral(error) - Kd × derivative(error)
```
MATH

Where Kp, Ki, Kd are PID controller tuning parameters.

3. **Apply with Constraints**:
   - New probability = Current probability × (1 + adjustment)
   - Clamp between service-specific min/max bounds
   - Ensure total across services ≤ system capacity
4. **Emergency Overrides**:
   - If storage > 90%: Apply global reduction multiplier (e.g., ×0.5)
   - If storage > 95%: Drop all non-error traces temporarily

Common Pitfalls in Sampling

⚠️ Pitfall 1: Sampling Bias Leading to Misleading Statistics

Description: Applying uniform sampling rates across all services creates a biased dataset where high-traffic services are overrepresented, while low-traffic but critical services may have no traces at all.

Why It's Wrong: Performance statistics (p95 latency, error rates) calculated from biased samples do not reflect true system behavior. Debugging issues in low-traffic services becomes impossible.

How to Fix:

- Implement **per-service sampling rates** with minimum guarantees
- Use **stratified sampling** to ensure representation from all service tiers
- Maintain **separate statistics** for sampled vs. unsampled populations
- Apply **Neyman allocation** for optimal distribution: allocate sample size proportional to traffic × variability

⚠️ Pitfall 2: Inconsistent Decisions for the Same Trace

Description: Different services or collectors making independent sampling decisions for spans belonging to the same trace, resulting in partial traces where some spans are stored while others are missing.

Why It's Wrong: Partial traces are useless for debugging—you cannot reconstruct the request flow or identify bottlenecks. This violates the fundamental guarantee of distributed tracing.

How to Fix:

- Use **deterministic hash-based sampling** on trace ID
- Propagate sampling decision in trace context (W3C `sampled` flag)
- Implement **centralized sampling coordination** for tail-based decisions
- Add **consistency checks** during trace assembly to detect and discard partial traces

⚠️ Pitfall 3: Adaptive Algorithm Oscillation

Description: The adaptive control loop overcorrects, causing sampling rates to oscillate wildly between extremes—from 100% to 1% and back—creating unpredictable retention and making historical comparisons meaningless.

Why It's Wrong: Oscillation reduces system stability, causes operational confusion, and makes capacity planning impossible. Engineers lose trust in the sampling system.

How to Fix:

- Implement **hysteresis** in control decisions (require sustained deviation before adjusting)
- Use **moving averages** with appropriate window sizes (not instantaneous measurements)
- Apply **rate limiting** to adjustments (maximum change per interval)
- Add **dead bands** where small deviations are ignored
- Monitor oscillation metrics and trigger alerts when detected

⚠️ Pitfall 4: Head-Based Sampling Losing All Error Traces

Description: With pure head-based sampling, error traces are dropped at the same rate as successful ones, making post-mortem debugging impossible since no error traces are available for analysis.

Why It's Wrong: The primary value of tracing is debugging failures. Losing all error traces defeats the purpose of the APM system.

How to Fix:

- Implement **tail-based sampling override** specifically for error traces
- Use **priority sampling** where error traces bypass head-based decisions
- Apply **multi-stage sampling**: sample all errors, sample subset of successes
- Consider **oversampling** error-prone services or operations

⚠️ Pitfall 5: Clock Skew Causing Premature Trace Completion

Description: In tail-based sampling, traces are considered "complete" after a timeout period. Clock skew between services causes incorrect timestamp ordering, leading to premature timeout and evaluation before all spans arrive.

Why It's Wrong: Early trace evaluation leads to incorrect tail-based decisions—traces may be dropped even though interesting spans are still in transit.

How to Fix:

- Use **watermark algorithm** based on observed timestamps, not system clock
- Implement **buffer extension** when late spans are detected for a trace
- Apply **NTP synchronization** across collectors with monitoring
- Design **forgiving timeout** with grace period for clock skew

Implementation Guidance for Trace Sampling

A. Technology Recommendations Table

Component	Simple Option	Advanced Option	Recommendation
Hash Function	Go's `fnv` package (built-in)	`MurmurHash3` (github.com/spaolacci/murmur3)	MurmurHash3 for better distribution
Probability Storage	In-memory map with mutex	Redis for distributed coordination	Start with in-memory, evolve to Redis
Adaptive Controller	Simple PID in Go	Control theory library (github.com/yourbasic/pid)	Custom PID for simplicity
Priority Queue	Slice with sorting	Heap container (container/heap)	Heap for O(log n) operations
Configuration	YAML file with viper	Dynamic config service (etcd/Consul)	YAML + viper initially
Metrics	Prometheus counters	OpenTelemetry metrics + Prometheus	Prometheus for simplicity

B. Recommended File/Module Structure

```
apm-tracing/
├── cmd/
│   ├── collector/           # Main collector binary
│   └── sampling-controller/ # Optional separate adaptive controller
├── internal/
│   ├── sampling/
│   │   ├── sampler.go       # Base sampler interface
│   │   ├── head_sampler.go  # Head-based sampling
│   │   ├── tail_sampler.go  # Tail-based sampling
│   │   ├── adaptive.go      # Adaptive controller
│   │   ├── priority.go      # Priority scoring
│   │   ├── consistency.go   # Hash consistency utilities
│   │   └── config.go        # Sampling configuration
│   ├── models/              # Data models (SamplingConfig, etc.)
│   └── storage/             # Storage layer
└── configs/
    └── sampling.yaml        # Sampling configuration file
```

C. Infrastructure Starter Code: Consistent Hash Sampler

```go
// internal/sampling/consistency.go

package sampling

import (
	"hash/fnv"
	"sync"
)

// ConsistentSampler ensures all participants make the same sampling decision
// for a given trace ID using deterministic hashing.
type ConsistentSampler struct {
	mu       sync.RWMutex
	rate     float64 // 0.0 to 1.0
	hashFunc func(string) uint64
}

// NewConsistentSampler creates a sampler with the given rate.
// Uses FNV-1a hash by default for simplicity and good distribution.
func NewConsistentSampler(rate float64) *ConsistentSampler {
	if rate < 0.0 || rate > 1.0 {
		rate = 0.1 // Default to 10%
	}

	return &ConsistentSampler{
		rate: rate,
		hashFunc: func(traceID string) uint64 {
			h := fnv.New64a()
			h.Write([]byte(traceID))
			return h.Sum64()
		},
	}
}

// ShouldSample determines if a trace should be kept based on its ID.
```

```
// Returns true if the trace should be sampled (kept).
func (s *ConsistentSampler) ShouldSample(traceID string) bool {
        s.mu.RLock()
        defer s.mu.RUnlock()

        if s.rate >= 1.0 {
                return true // Sample everything
        }
        if s.rate <= 0.0 {
                return false // Sample nothing
        }

        // Hash to a float in [0, 1)
        hash := s.hashFunc(traceID)
        normalized := float64(hash) / float64(^uint64(0))

        return normalized < s.rate
}

// SetRate updates the sampling rate (thread-safe).
func (s *ConsistentSampler) SetRate(rate float64) {
        s.mu.Lock()
        defer s.mu.Unlock()

        if rate < 0.0 {
                rate = 0.0
        } else if rate > 1.0 {
                rate = 1.0
        }
        s.rate = rate
}

// GetRate returns the current sampling rate.
```

```go
func (s *ConsistentSampler) GetRate() float64 {
        s.mu.RLock()
        defer s.mu.RUnlock()
        return s.rate
}
```

D. Core Logic Skeleton Code

```go
// internal/sampling/head_sampler.go

package sampling

import (
	"context"
	"fmt"
	"sync"
	"time"

	"apm-tracing/internal/models"
)

// HeadSampler implements head-based sampling with per-service rates.
type HeadSampler struct {
	mu            sync.RWMutex
	globalSampler *ConsistentSampler
	serviceRates  map[string]*ConsistentSampler
	defaultRate   float64
	stats         HeadSamplerStats
}

// HeadSamplerStats tracks sampling decisions for monitoring.
type HeadSamplerStats struct {
	TotalTraces     int64
	SampledTraces   int64
	PerService      map[string]ServiceStats
	LastReset       time.Time
}

// ServiceStats tracks per-service sampling statistics.
type ServiceStats struct {
	TracesSeen    int64
	TracesSampled int64
}
```

```
// NewHeadSampler creates a new head-based sampler with configuration.
func NewHeadSampler(config *models.SamplingConfig) (*HeadSampler, error) {
        if config == nil {
                return nil, fmt.Errorf("config cannot be nil")
        }

        sampler := &HeadSampler{
                globalSampler: NewConsistentSampler(config.Probability),
                serviceRates:  make(map[string]*ConsistentSampler),
                defaultRate:   config.Probability,
                stats: HeadSamplerStats{
                        PerService: make(map[string]ServiceStats),
                        LastReset:  time.Now(),
                },
        }

        // Initialize per-service samplers
        for serviceName, rate := range config.PerService {
                if rate < 0.0 || rate > 1.0 {
                        return nil, fmt.Errorf("invalid rate for service %s: %f", serviceName, rate)
                }
                sampler.serviceRates[serviceName] = NewConsistentSampler(rate)
        }

        return sampler, nil
}

// Decide makes a head-based sampling decision for a trace.
// This should be called when the root span is created.
func (s *HeadSampler) Decide(ctx context.Context, traceID, serviceName string) bool {
        // TODO 1: Look up per-service sampler if configured
        // TODO 2: If no per-service sampler exists, use global sampler
```

```
        // TODO 3: Call ShouldSample on the appropriate sampler

        // TODO 4: Update statistics (thread-safe)

        // TODO 5: Return true if trace should be kept, false otherwise

        // Hint: Use s.mu for thread safety when accessing serviceRates and stats

        return false // Placeholder

}


// UpdateRates dynamically adjusts sampling rates based on adaptive controller input.

func (s *HeadSampler) UpdateRates(ctx context.Context, newGlobalRate float64, newServiceRates
map[string]float64) error {

        // TODO 1: Validate all rates are in [0.0, 1.0]

        // TODO 2: Update global sampler rate

        // TODO 3: Update or create per-service samplers for each entry in newServiceRates

        // TODO 4: Log the rate changes for audit purposes

        // TODO 5: Reset statistics to track new rate effectiveness

        return nil // Placeholder

}


// GetStats returns current sampling statistics for monitoring.

func (s *HeadSampler) GetStats(ctx context.Context) HeadSamplerStats {

        // TODO 1: Acquire read lock for thread safety

        // TODO 2: Return copy of stats (not reference)

        // TODO 3: Include current rates in the returned stats

        return HeadSamplerStats{} // Placeholder

}
```

GO

```go
// internal/sampling/tail_sampler.go

package sampling


import (

        "context"

        "fmt"

        "time"



        "apm-tracing/internal/models"

)


// TailSampler evaluates completed traces and overrides head decisions.

type TailSampler struct {

        rules      []TailSamplingRule

        evaluator  *TraceEvaluator

        buffer     *TraceBuffer

        timeout    time.Duration

        stats      TailSamplerStats

}


// TailSamplingRule defines criteria for tail-based sampling.

type TailSamplingRule struct {

        Name        string

        Condition   TraceCondition

        Priority    int // Higher priority rules execute first

        KeepIfMatch bool

}


// TraceCondition evaluates whether a trace matches criteria.

type TraceCondition interface {

        Matches(trace *models.Trace) bool

}


// TraceEvaluator scores traces based on multiple criteria.

type TraceEvaluator struct {
```

```
        errorWeight      float64
        latencyWeight    float64
        operationWeights map[string]float64
        threshold        float64
}

// NewTailSampler creates a tail-based sampler with configured rules.
func NewTailSampler(rules []TailSamplingRule, timeout time.Duration) (*TailSampler, error) {
        if timeout <= 0 {
                timeout = 30 * time.Second // Default timeout
        }

        sampler := &TailSampler{
                rules:   rules,
                timeout: timeout,
                evaluator: &TraceEvaluator{
                        errorWeight:      50.0,
                        latencyWeight:    30.0,
                        operationWeights: make(map[string]float64),
                        threshold:        60.0, // Keep if score >= 60
                },
                stats: TailSamplerStats{
                        TotalEvaluated:   0,
                        HeadSampled:      0,
                        TailOverrides:    0,
                        PerRuleDecisions: make(map[string]int64),
                },
        }

        // Sort rules by priority (descending)
        // TODO: Implement rule sorting

        return sampler, nil
```

```go
}

// EvaluateTrace examines a completed trace for tail-based sampling.
// Returns true if the trace should be kept (either head decision or tail override).
func (s *TailSampler) EvaluateTrace(ctx context.Context, trace *models.Trace, headDecision bool) bool {
	// TODO 1: Update statistics for head decision
	// TODO 2: Check if trace already marked for keeping by head sampler
	// TODO 3: If head decision was false, evaluate against tail rules
	// TODO 4: Apply rules in priority order until a rule matches
	// TODO 5: If any rule with KeepIfMatch=true matches, override to keep
	// TODO 6: Also compute priority score using evaluator
	// TODO 7: If score >= threshold, override to keep
	// TODO 8: Update tail override statistics if decision changed
	// TODO 9: Return final keep/drop decision
	return headDecision // Placeholder
}

// ProcessCompletedTraces runs a goroutine to evaluate traces that have completed.
func (s *TailSampler) ProcessCompletedTraces(ctx context.Context, traceCh <-chan *models.Trace) error {
	// TODO 1: Start a loop that listens on traceCh
	// TODO 2: For each trace, check if it's complete (all spans received or timeout)
	// TODO 3: Extract head decision from trace metadata
	// TODO 4: Call EvaluateTrace for final decision
	// TODO 5: If final decision is keep, persist to storage
	// TODO 6: If final decision is drop, clean up buffer
	// TODO 7: Handle context cancellation gracefully
	return nil // Placeholder
}

// AddTraceToBuffer stores a trace for later evaluation when it completes.
func (s *TailSampler) AddTraceToBuffer(ctx context.Context, traceID string, spans []models.Span) error {
	// TODO 1: Check if trace already exists in buffer
	// TODO 2: Add spans to existing trace buffer or create new
	// TODO 3: Start timeout timer if this is the first span
```

```
        // TODO 4: Check if trace is now complete (all expected spans received)
        // TODO 5: If complete, move to evaluation queue
        return nil // Placeholder
}
```

E. Language-Specific Hints

1. **Concurrency**: Use `sync.RWMutex` for sampler statistics—multiple goroutines read stats, few update them.
2. **Hashing**: For production, use `github.com/spaolacci/murmur3` instead of FNV for better distribution.
3. **Context Propagation**: Pass `context.Context` through all sampling methods for cancellation and deadlines.
4. **Metrics Export**: Use Prometheus counters for sampling statistics:

```go
var sampledTraces = prometheus.NewCounterVec(
    prometheus.CounterOpts{
        Name: "sampler_traces_total",
        Help: "Total traces processed by sampler",
    },
    []string{"service", "decision"},
)
```

5. **Configuration Hot Reload**: Use `fsnotify` to watch config file changes and reload sampling rates without restart.
6. **Memory Management**: For trace buffers, use `sync.Pool` for span slices to reduce GC pressure.

F. Milestone Checkpoint

After implementing the sampling subsystem:

1. **Test Head-Based Sampling**:

```bash
# Run unit tests
go test ./internal/sampling -v -run TestHeadSampler

# Expected output: All tests pass, showing:
# - Consistent decisions for same trace ID
# - Per-service rates work correctly
# - Statistics are tracked accurately
```

2. **Verify Tail-Based Override**:

```bash
# Start collector with sampling enabled
./cmd/collector --config configs/sampling.yaml

# Send test traces with errors
curl -X POST http://localhost:8080/api/v1/spans \
  -H "Content-Type: application/json" \
  -d '{"traceId": "err-123", "spans": [{"name": "errorOp", "status": {"code": 2}}]}'

# Check logs: Should see "tail override" for error trace
# Even with 1% sampling rate, error trace should be kept
```

3. **Validate Adaptive Control**:

```bash
# Monitor sampling rate adjustments
curl http://localhost:9090/metrics | grep sampler_rate

# Send burst traffic (e.g., 10k spans in 1 second)
# Observe: Sampling rate should decrease temporarily
# After traffic normalizes, rate should return to configured level
```

Signs of Problems:

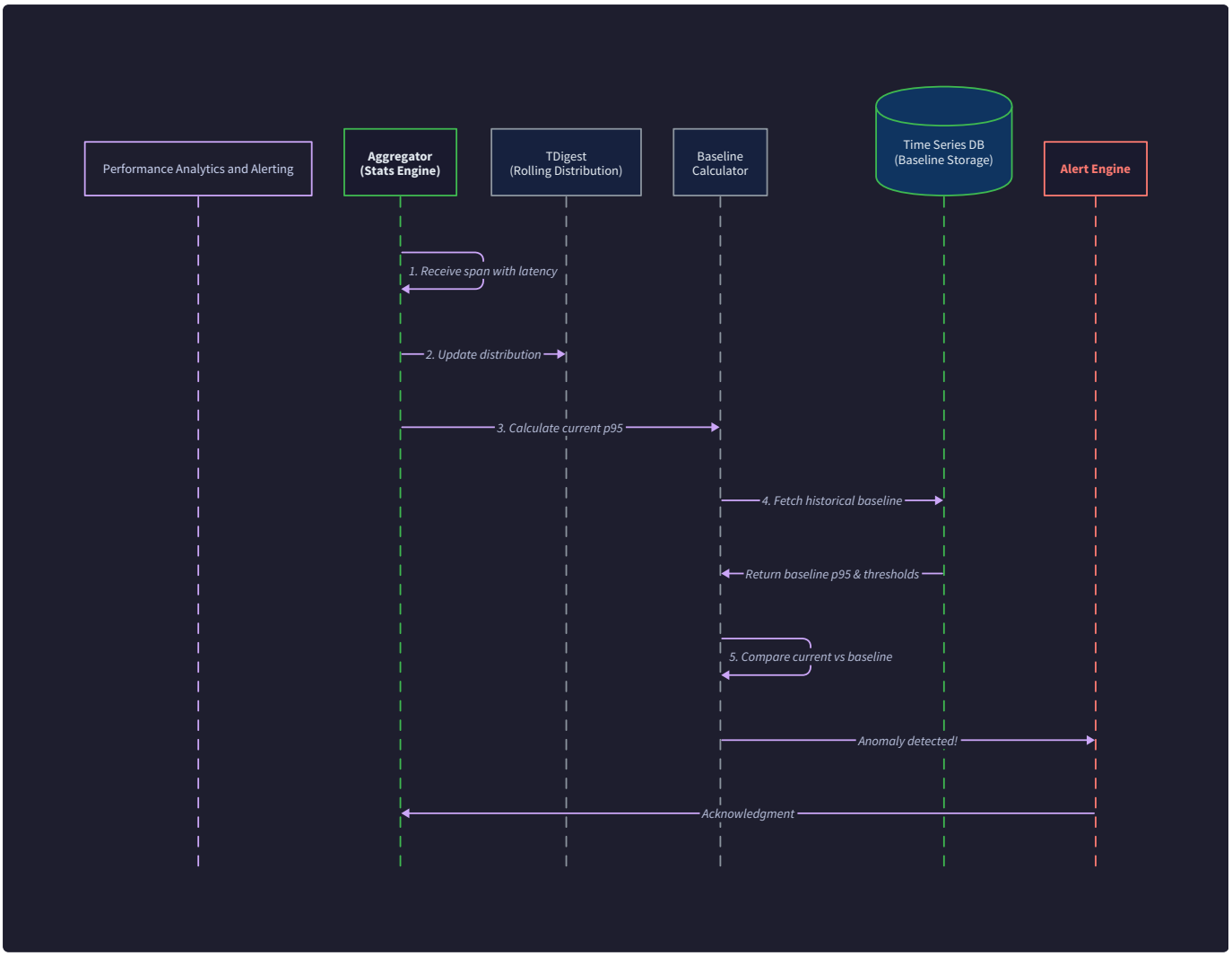
- `sampler_traces_total{decision="drop"}` increasing during normal traffic (rate too low)
- Memory growing unbounded (trace buffer not evicting completed traces)
- Inconsistent trace assembly (spans missing from kept traces)

Debugging Steps:

1. Check hash consistency: Same trace ID should always get same decision
2. Verify context propagation: `sampled` flag should be passed through all services
3. Monitor buffer sizes: Implement metrics for trace buffer occupancy

8. Component Design: Performance Analytics & Anomaly Detection (Milestone 4)

> **Milestone(s):** This section corresponds to Milestone 4: Performance Analytics & Anomaly Detection, which designs the analytics engine that computes performance percentiles, establishes historical baselines, and automatically flags anomalies in the APM Tracing System.



Mental Model: The Weather Forecaster for Performance

Imagine our distributed system is a vast, complex climate system. Individual service calls are like local weather events—some sunny and fast (low latency), others stormy and slow (high latency). The **Performance Analytics & Anomaly Detection** component acts as our **weather forecaster**.

A good weather forecaster doesn't just report today's temperature; they:

1. **Know historical patterns**: They understand that 85°F is normal for July but alarming for December.
2. **Detect anomalies**: They spot a sudden pressure drop indicating an approaching storm (a latency spike).
3. **Provide probabilistic forecasts**: They calculate there's a 90% chance of rain (a 90% likelihood this latency regression is real, not random noise).
4. **Differentiate local vs. global phenomena**: They know a thunderstorm in one valley doesn't mean the entire continent is experiencing bad weather (a problem in one service vs. system-wide degradation).

Similarly, our analytics engine must:

- **Calculate what's "normal"** by establishing baselines from historical data for each service and operation.
- **Detect "unusual weather"** by comparing current performance against those baselines using statistical methods.
- **Focus on the right signals** by distinguishing between expected variations (daily traffic patterns) and true anomalies (cascading failures).
- **Provide actionable insights** by correlating anomalies with specific services, operations, and trace characteristics.

This mental model helps us understand why simply calculating averages isn't enough—we need context-aware, statistical analysis that understands both the current state and historical patterns of our distributed system's "weather."

Algorithms: Percentile Calculation and Anomaly Detection

The analytics engine performs two core computations: **streaming percentile calculation** for real-time performance metrics and **statistical anomaly detection** to flag deviations from normal patterns.

Percentile Calculation with T-Digest

Traditional percentile calculation requires storing all data points, which is impossible for high-volume tracing systems. We need an algorithm that can provide accurate percentile estimates from streaming data with bounded memory.

The t-digest algorithm solves this by clustering similar data points. Think of it as creating a histogram that automatically adjusts its bucket sizes: where data is dense (many similar latency values), buckets are narrow and precise; where data is sparse (outliers), buckets are wider but still capture the distribution tails accurately.

Algorithm Step	Description	Purpose
1. **Initialize t-digest**	Create an empty t-digest structure with a compression parameter (typically δ=100-1000).	Sets up the data structure with desired accuracy/memory trade-off.
2. **Add value**	For each incoming latency measurement, find the nearest centroid in the t-digest. If adding the value would exceed size limits for that centroid's weight, create a new centroid.	Incorporates new data while maintaining the size bounds of the data structure.
3. **Merge centroids**	Periodically merge adjacent centroids when their combined weight doesn't violate the size constraint based on the quantile they represent.	Maintains the compression guarantee and prevents unbounded growth.
4. **Query percentile**	To find the p-th percentile (e.g., p95), traverse the sorted centroids, summing weights until reaching p% of total weight, then interpolate between centroids.	Provides accurate percentile estimates without storing all raw data.

The key advantage is **bounded memory**: a t-digest with compression parameter δ=100 uses roughly 100 centroids regardless of how many data points are added, making it ideal for long-running services that process millions of latency measurements.

Anomaly Detection with Statistical Methods

Once we have current percentiles, we need to determine if they represent anomalous behavior. We use multiple statistical techniques in combination:

1. Z-Score Method for Point Anomalies For detecting sudden spikes or drops in metrics like error rate or latency.

Step	Description	Mathematical Formulation
1. **Establish baseline**	Calculate mean (μ) and standard deviation (σ) from historical data for the same time window (e.g., same hour last week).	$\mu = \Sigma x/n$, $\sigma = \sqrt{(\Sigma(x-\mu)^2/(n-1))}$
2. **Compute z-score**	For current value x, calculate how many standard deviations it is from the mean.	$z = (x - \mu) / \sigma$
3. **Flag anomaly**	If	z

2. Moving Average with Control Limits for Trend Anomalies For detecting sustained deviations rather than single-point spikes.

Component	Description	Purpose
Simple Moving Average (SMA)	Average of last n values.	Smooths short-term fluctuations to reveal trends.
Exponential Moving Average (EMA)	Gives more weight to recent values.	More responsive to recent changes than SMA.
Upper/Lower Control Limits	Bands set at ±k standard deviations from the moving average.	Defines the "normal" range for the metric.
Anomaly Condition	Current value outside control limits for m consecutive measurements.	Reduces false positives from transient spikes.

3. Seasonal-Trend Decomposition for Periodic Patterns For systems with daily/weekly cycles (common in web applications).

Step	Description	Output
1. **Decompose time series**	Separate historical data into: Trend (T), Seasonal (S), and Residual (R) components.	$x_t = T_t + S_t + R_t$
2. **Forecast expected value**	Project trend forward and add appropriate seasonal component.	$\hat{y}_t = \hat{T}_t + \hat{S}_t$
3. **Compare actual vs. forecast**	Compute residual: $e_t = x_t - \hat{y}_t$.	Difference between actual and expected.
4. **Flag large residuals**	If	e_t

In practice, we combine these methods: z-score for immediate spike detection, moving average for sustained deviations, and seasonal decomposition to avoid false alarms during expected traffic patterns.

ADR: Time-Series Storage for Aggregates

> **Decision: Use Dual-Layer Storage with Rolling In-Memory Windows and External Time-Series Database**

Context: The analytics engine needs to store two types of data: (1) real-time aggregates for the current time window (last 1-5 minutes) for immediate anomaly detection, and (2) historical aggregates (hours, days, weeks) for baseline calculation and trend analysis. We must choose storage strategies that balance performance, accuracy, and operational complexity.

Options Considered:

1. **Pure in-memory rolling windows**: Store all aggregates in memory with fixed-size circular buffers.
2. **Pure external time-series database**: Write all aggregates to a dedicated TSDB (e.g., Prometheus, InfluxDB).
3. **Hybrid approach**: In-memory for recent data with periodic flush to external TSDB.

Decision: We chose **Option 3: Hybrid approach** with the following implementation:

- **Short-term (last 1 hour)**: In-memory ring buffers per (service, operation) tuple, updated in real-time.
- **Long-term (beyond 1 hour)**: Flushed to an external time-series database with configurable retention policies.

Rationale:

1. **Latency requirements**: Real-time anomaly detection needs sub-second query latency for the current window, which in-memory storage provides.
2. **Historical analysis needs**: Baseline calculation requires days/weeks of data, which exceeds practical memory limits.

3. **Cost-effectiveness**: External TSDBs optimize storage for time-series data (compression, downsampling) better than a custom solution.
4. **Operational simplicity**: Using a battle-tested TSDB for long-term storage reduces maintenance burden compared to building our own durable storage layer.
5. **Data durability**: In-memory data is volatile; periodic flushing to durable storage prevents data loss during restarts.

Consequences:

- ✅ **Low-latency real-time analytics** for current time window.
- ✅ **Scalable historical storage** using specialized TSDB.
- ✅ **Data persistence** across service restarts.
- ⚠️ **Added complexity** of two storage layers with synchronization logic.
- ⚠️ **Eventual consistency** for historical queries (data appears after flush interval).
- ⚠️ **Dependency** on external TSDB for long-term data.

Storage Option	Pros	Cons	Chosen?
Pure in-memory rolling windows	- Extremely fast reads/writes - No external dependencies - Simple implementation	- Limited history (memory-bound) - Data lost on restart - No downsampling for long retention	❌
Pure external time-series database	- Unlimited history with retention policies - Built-in downsampling/aggregation - Durable storage	- Higher latency for real-time queries - External dependency failure modes - Network overhead for every write	❌
Hybrid approach (in-memory + TSDB)	- Fast real-time window queries - Scalable historical storage - Data persistence through flushing - Best of both worlds	- Two storage systems to maintain - Synchronization complexity - Slight delay for historical data availability	✅

Common Pitfalls in Analytics

⚠️ Pitfall 1: The "Percentile of Percentiles" Fallacy

- **Description**: Calculating percentiles across services incorrectly by taking the p95 of each service's p95 latencies. For example: Service A has p95=100ms, Service B has p95=200ms, so you report "system p95 = 150ms" (average) or "system p95 = 200ms" (max).
- **Why it's wrong**: Percentiles are not linear or commutative. The true p95 latency for requests flowing through both services requires analyzing the end-to-end latency distribution, not combining service-level percentiles. This fallacy gives misleading performance summaries.
- **How to fix**: Calculate percentiles from the **root span durations** (end-to-end request time) or use statistical methods that properly combine distributions (like convolution or Monte Carlo simulation if you have the raw data).

⚠️ Pitfall 2: False Positives During Deployments and Traffic Changes

- **Description**: Anomaly detection systems triggering alerts during planned deployments, traffic spikes (flash sales), or daily cycles, even though these are expected events.
- **Why it's wrong**: These false alarms cause "alert fatigue" where teams start ignoring alerts, potentially missing real issues. It wastes engineering time investigating expected behavior.
- **How to fix**:
  1. **Maintenance windows**: Suppress alerts during known deployment periods.
  2. **Traffic-aware baselines**: Use seasonal decomposition to account for daily/weekly patterns.

3. **Change detection integration**: Correlate anomalies with deployment markers in the CI/CD pipeline.
4. **Adaptive thresholds**: Increase sensitivity during stable periods, decrease during known volatile periods.

⚠️ Pitfall 3: Baseline Drift Over Time

- **Description**: As systems evolve (new features, optimizations, infrastructure changes), what was "normal" last month may no longer be relevant. Using stale baselines causes either missed anomalies (if performance degrades gradually) or false positives (if improvements make old baselines too pessimistic).
- **Why it's wrong**: Static baselines become increasingly inaccurate over time, reducing the effectiveness of anomaly detection.
- **How to fix**:
  1. **Exponential decay**: Weight recent data more heavily than old data when calculating baselines.
  2. **Change point detection**: Automatically detect significant shifts in performance characteristics and reset baselines.
  3. **Manual baseline updates**: Allow operators to mark periods as "new normal" after verified changes.
  4. **Multiple baseline windows**: Compare against both recent (last 24h) and longer-term (last week) baselines to distinguish temporary shifts from permanent changes.

⚠️ Pitfall 4: Ignoring Statistical Power and Confidence

- **Description**: Making anomaly decisions with insufficient data. For example, flagging a "500% increase in error rate" when going from 1 error in 1000 requests to 5 errors in 1000 requests—statistically insignificant yet appears dramatic.
- **Why it's wrong**: Small sample sizes lead to high variance and unreliable conclusions. A system might appear "anomalous" purely due to random chance with low traffic volumes.
- **How to fix**:
  1. **Minimum sample thresholds**: Require at least N requests (e.g., 100) in a time window before evaluating anomalies.
  2. **Confidence intervals**: Calculate and display the uncertainty around metrics (e.g., "error rate: 0.5% ± 0.2% with 95% confidence").
  3. **Bayesian methods**: Incorporate prior beliefs about expected rates, which naturally handles low-traffic services better than frequentist statistics.

Implementation Guidance for Analytics

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Percentile Algorithm	Pre-implemented t-digest library (`github.com/caio/go-tdigest`)	Custom t-digest implementation with optimizations for sparse data
Time-Series Storage	In-memory ring buffers + CSV file export	In-memory buffers + Prometheus remote write protocol
Anomaly Detection	Z-score + moving average implemented manually	Statistical libraries (`github.com/montanaflynn/stats`) + machine learning (Isolation Forest)
Alerting	Simple HTTP webhook notifications	Dedicated alert manager (Prometheus Alertmanager integration)
Visualization	Built-in metrics endpoint returning JSON	Grafana dashboard with pre-configured panels

B. Recommended File/Module Structure

```
project-root/
├── cmd/
│   └── analytics-engine/
│       └── main.go                # Analytics service entry point
├── internal/
│   ├── analytics/
│   │   ├── aggregator.go          # Main percentile aggregator
│   │   ├── tdigest.go             # T-digest wrapper/implementation
│   │   ├── anomaly/
│   │   │   ├── detector.go        # Anomaly detection coordinator
│   │   │   ├── zscore.go          # Z-score detector
│   │   │   ├── moving_avg.go      # Moving average detector
│   │   │   └── seasonal.go        # Seasonal decomposition detector
│   │   ├── storage/
│   │   │   ├── interface.go       # Storage interface
│   │   │   ├── memory_store.go    # In-memory ring buffer storage
│   │   │   └── tsdb_client.go     # External TSDB client
│   │   ├── baseline/
│   │   │   ├── calculator.go      # Baseline calculation
│   │   │   └── manager.go         # Baseline lifecycle management
│   │   └── alert/
│   │       ├── evaluator.go       # Alert condition evaluation
│   │       └── notifier.go        # Alert notification dispatch
│   └── models/                    # Shared data models
│       └── analytics.go           # Analytics-specific types
└── config/
    └── analytics.yaml             # Analytics configuration
```

C. Infrastructure Starter Code

Complete T-Digest Wrapper (ready to use):

```go
// internal/analytics/tdigest.go

package analytics


import (

    "github.com/caio/go-tdigest"

    "sync"

    "time"

)


// TDigestMetric maintains a t-digest for streaming percentile calculation

type TDigestMetric struct {

    digest    *tdigest.TDigest

    mu        sync.RWMutex

    count     int64

    lastReset time.Time

}


// NewTDigestMetric creates a new metric with specified compression

func NewTDigestMetric(compression float64) (*TDigestMetric, error) {

    digest, err := tdigest.New(tdigest.Compression(compression))

    if err != nil {

        return nil, err

    }

    return &TDigestMetric{

        digest:    digest,

        lastReset: time.Now(),

    }, nil

}


// Add adds a value to the distribution

func (t *TDigestMetric) Add(value float64) {

    t.mu.Lock()

    defer t.mu.Unlock()

    t.digest.Add(value)

    t.count++
```

```go
}

// Quantile returns the estimated value at the given quantile (0.0-1.0)
func (t *TDigestMetric) Quantile(q float64) float64 {
    t.mu.RLock()
    defer t.mu.RUnlock()
    return t.digest.Quantile(q)
}

// Count returns the number of observations
func (t *TDigestMetric) Count() int64 {
    t.mu.RLock()
    defer t.mu.RUnlock()
    return t.count
}

// Reset clears the distribution and resets counters
func (t *TDigestMetric) Reset() {
    t.mu.Lock()
    defer t.mu.Unlock()
    t.digest.Reset()
    t.count = 0
    t.lastReset = time.Now()
}
```

Complete In-Memory Time-Series Storage (ready to use):

```go
// internal/analytics/storage/memory_store.go

package storage


import (

    "container/ring"

    "sync"

    "time"

)


// MetricPoint represents a single data point in a time series

type MetricPoint struct {

    Timestamp time.Time

    Value     float64

    Labels    map[string]string // e.g., {"service": "api", "operation": "GET /users"}

}


// MemoryStore implements in-memory ring buffer storage for recent metrics

type MemoryStore struct {

    buffers   map[string]*ring.Ring // key: metric+labels hash

    mu        sync.RWMutex

    maxPoints int

    ttl       time.Duration

}


// NewMemoryStore creates a new in-memory store with given capacity

func NewMemoryStore(maxPoints int, ttl time.Duration) *MemoryStore {

    return &MemoryStore{

        buffers:   make(map[string]*ring.Ring),

        maxPoints: maxPoints,

        ttl:       ttl,

    }

}


// Write adds a metric point to the appropriate ring buffer

func (m *MemoryStore) Write(point MetricPoint) error {
```

```
    m.mu.Lock()
    defer m.mu.Unlock()

    key := m.hashKey(point.Labels)

    // Get or create ring buffer
    buffer, exists := m.buffers[key]
    if !exists {
        buffer = ring.New(m.maxPoints)
        m.buffers[key] = buffer
    }

    // Advance and set value
    buffer = buffer.Next()
    buffer.Value = point
    m.buffers[key] = buffer

    return nil
}

// Read returns all points within the time range for given labels
func (m *MemoryStore) Read(labels map[string]string, start, end time.Time) ([]MetricPoint, error) {
    m.mu.RLock()
    defer m.mu.RUnlock()

    key := m.hashKey(labels)
    buffer, exists := m.buffers[key]
    if !exists {
        return []MetricPoint{}, nil
    }

    var points []MetricPoint
    buffer.Do(func(value interface{}) {
```

```
        if value == nil {
            return
        }
        point := value.(MetricPoint)
        if !point.Timestamp.Before(start) && !point.Timestamp.After(end) {
            points = append(points, point)
        }
    })

    return points, nil
}

// hashKey creates a simple string key from labels map
func (m *MemoryStore) hashKey(labels map[string]string) string {
    // Simple implementation - for production use a proper hash
    var key string
    for k, v := range labels {
        key += k + "=" + v + ";"
    }
    return key
}

// cleanup removes expired buffers (call this periodically)
func (m *MemoryStore) cleanup() {
    m.mu.Lock()
    defer m.mu.Unlock()

    cutoff := time.Now().Add(-m.ttl)
    for key, buffer := range m.buffers {
        // Check if buffer has recent data
        recent := false
        buffer.Do(func(value interface{}) {
            if value == nil {
```

```
                return
            }
            point := value.(MetricPoint)
            if point.Timestamp.After(cutoff) {
                recent = true
            }
        })

        if !recent {
            delete(m.buffers, key)
        }
    }
}
```

D. Core Logic Skeleton Code

Percentile Aggregator (learner implements TODOs):

GO

```go
// internal/analytics/aggregator.go

package analytics


import (

    "context"

    "time"



    "github.com/your-project/internal/models"

)


// PercentileAggregator maintains t-digests for service/operation latency percentiles

type PercentileAggregator struct {

    // digests is a map of key(service+operation) -> TDigestMetric

    digests map[string]*TDigestMetric

    mu      sync.RWMutex

    window  time.Duration

}


// NewPercentileAggregator creates a new aggregator with the given time window

func NewPercentileAggregator(window time.Duration) *PercentileAggregator {

    // TODO 1: Initialize the digests map

    // TODO 2: Set up a background goroutine to reset digests after each window

    // TODO 3: Return the initialized aggregator

    return nil

}


// ProcessSpan updates percentiles with data from a completed span

func (p *PercentileAggregator) ProcessSpan(ctx context.Context, span models.Span) error {

    // TODO 1: Extract service name and operation name from span

    // TODO 2: Create a unique key from service+operation

    // TODO 3: Get or create a TDigestMetric for this key

    // TODO 4: Add the span's duration (converted to milliseconds) to the digest

    // TODO 5: Update any internal statistics counters

    // TODO 6: Return nil on success or appropriate error

    return nil
```

```
}


// GetPercentiles returns current percentile values for a service/operation
func (p *PercentileAggregator) GetPercentiles(ctx context.Context, service, operation string) (p50, p95, p99
time.Duration, err error) {
    // TODO 1: Construct the lookup key from service+operation
    // TODO 2: Acquire read lock on the digests map
    // TODO 3: Find the TDigestMetric for this key
    // TODO 4: If not found, return zero values with appropriate error
    // TODO 5: Query the digest for 0.5, 0.95, and 0.99 quantiles
    // TODO 6: Convert the float64 results back to time.Duration
    // TODO 7: Return the three percentile values
    return 0, 0, 0, nil
}


// ResetWindow clears all digests and starts a new aggregation window
func (p *PercentileAggregator) ResetWindow() {
    // TODO 1: Acquire write lock on digests map
    // TODO 2: For each TDigestMetric in the map, call Reset()
    // TODO 3: Optionally, log statistics about the completed window
    // TODO 4: Release lock
}
```

Anomaly Detector (learner implements TODOs):

```go
// internal/analytics/anomaly/detector.go

package anomaly


import (

    "context"

    "time"

)


// Detector evaluates metrics against baselines to detect anomalies

type Detector struct {

    baselineCalc BaselineCalculator

    methods      []DetectionMethod

    thresholds   map[string]float64

}


// DetectionMethod is an interface for different anomaly detection algorithms

type DetectionMethod interface {

    Detect(current, historical []float64) (bool, float64, error)

    Name() string

}


// NewDetector creates a new anomaly detector with configured methods

func NewDetector(baselineCalc BaselineCalculator) *Detector {

    // TODO 1: Initialize with provided baseline calculator

    // TODO 2: Set up default detection methods (z-score, moving average)

    // TODO 3: Set default thresholds for each method

    // TODO 4: Return initialized detector

    return nil

}


// CheckMetric evaluates a current metric value for anomalies

func (d *Detector) CheckMetric(ctx context.Context, metricName string, labels map[string]string, currentValue
float64) ([]AnomalyResult, error) {

    // TODO 1: Retrieve historical values for this metric+labels from storage

    // TODO 2: If insufficient historical data, return empty result (no decision)

    // TODO 3: For each detection method in d.methods:
```

```
    //   a. Call Detect() with current value and historical values
    //   b. If detection returns true, create AnomalyResult record
    // TODO 4: Aggregate results from all methods (e.g., majority vote)
    // TODO 5: Return slice of anomaly results
    return nil, nil
}

// AnomalyResult represents a single anomaly detection outcome
type AnomalyResult struct {
    Metric      string
    Labels      map[string]string
    Timestamp   time.Time
    Value       float64
    Expected    float64 // baseline value
    Confidence  float64 // 0.0-1.0 confidence in detection
    Method      string  // which detection method flagged it
    Severity    string  // "warning", "critical"
}

// BaselineCalculator computes expected values from historical data
type BaselineCalculator interface {
    Calculate(ctx context.Context, metricName string, labels map[string]string, t time.Time) (float64, error)
}
```

E. Language-Specific Hints

1. **Go Concurrency Patterns**: Use `sync.RWMutex` for the percentile aggregator's digests map since reads (percentile queries) are more frequent than writes (span processing).
2. **Time Handling**: Use `time.Duration` consistently for all latency values. Convert to `float64` milliseconds for t-digest storage: `ms := float64(span.Duration.Milliseconds())`.
3. **Context Propagation**: Pass `context.Context` through all method calls to enable proper cancellation and timeout handling, especially for TSDB queries.
4. **Error Wrapping**: Use `fmt.Errorf` with `%w` to wrap errors: `return fmt.Errorf("failed to query baseline: %w", err)`.
5. **Efficient Map Keys**: For the `digests` map key, use a precomputed string: `key := fmt.Sprintf("%s:%s", service, operation)` or use a struct key with custom hash function for better performance.
6. **Background Goroutines**: Use `time.Ticker` for periodic tasks like window resetting:

```go
    ticker := time.NewTicker(p.window)

    defer ticker.Stop()

    for range ticker.C {

        p.ResetWindow()

    }
```

F. Milestone Checkpoint

After implementing the Performance Analytics component, verify functionality with these steps:

1. **Start the analytics engine**:

```bash
    go run cmd/analytics-engine/main.go --config config/analytics.yaml
```

2. **Send test spans with varying latencies**:

```bash
    # Use a test script to send 1000 spans with latencies following normal distribution

    ./scripts/send_test_spans.go --count 1000 --mean-latency 100ms --std-dev 20ms
```

3. **Query percentiles via API**:

```bash
    curl "http://localhost:8081/analytics/percentiles?service=api&operation=GET%2Fusers"
```

Expected output (example):

```json
    {

      "service": "api",

      "operation": "GET /users",

      "p50": "98ms",

      "p95": "142ms",

      "p99": "165ms",

      "sample_count": 1000,

      "window_start": "2023-10-01T10:00:00Z",

      "window_end": "2023-10-01T10:05:00Z"

    }
```

4. **Trigger an anomaly detection**:

```bash
    # Send spans with suddenly high latency

    ./scripts/send_test_spans.go --count 50 --mean-latency 500ms --std-dev 100ms
```

Check logs for anomaly alerts:

```
INFO anomaly detected: service=api operation=GET /users current=512ms expected=105ms z-score=4.2
```

5. **Verify baseline calculation**:

```bash
curl "http://localhost:8081/analytics/baselines?service=api&operation=GET%2Fusers"
```

Should return historical percentiles for comparison.

Signs of issues:

- Percentiles stuck at 0: Likely span processing not working; check `ProcessSpan` implementation.
- Anomaly alerts never firing: Check detection thresholds and baseline calculation.
- Memory usage growing unbounded: Verify t-digest compression and ring buffer size limits.
- "No historical data" errors: Ensure time-series storage is properly flushing to persistent storage.

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Percentiles don't match expected values	Incorrect t-digest compression parameter or insufficient data	1. Check sample count in aggregator 2. Verify span durations are being added correctly 3. Test t-digest with known dataset	Adjust compression parameter (higher = more accurate), ensure minimum samples before trusting percentiles
Anomaly detector fires during normal traffic patterns	Baseline doesn't account for daily cycles	1. Check if alerts cluster at same time daily 2. Compare current time to historical same-hour data	Implement seasonal decomposition or time-of-day aware baselines
High memory usage in analytics engine	Too many t-digest instances or large ring buffers	1. Check number of unique service+operation combinations 2. Monitor buffer sizes 3. Profile heap usage	Limit cardinality (e.g., ignore high-cardinality operation names), reduce buffer sizes, implement LRU eviction
"No baseline data" errors for valid services	Flushing to TSDB failing or retention too short	1. Check TSDB connection logs 2. Verify data exists in TSDB for time ranges 3. Check flush interval configuration	Fix TSDB connectivity, adjust retention policies, implement fallback to recent window if no history
Delayed anomaly detection (>5 minutes)	Aggregation window too long or detection runs infrequently	1. Check window duration configuration 2. Monitor detection cycle timing 3. Check for blocking operations in pipeline	Reduce window size, run detection more frequently, pipeline spans for lower latency
False positives during deployments	Baseline includes pre-deployment performance	1. Check if deployment timestamps correlate with alerts 2. Compare pre/post-deployment baselines	Implement deployment markers, suppress alerts during known changes, use shorter baseline windows during volatile periods

> **Milestone(s):** This section corresponds to Milestone 5: APM SDK & Auto-Instrumentation, which designs the client-side library that automatically instruments application code to generate and propagate traces.

9. Component Design: APM SDK & Auto-Instrumentation (Milestone 5)

The **APM SDK** is the client-side library embedded within each monitored service. Its primary responsibility is to automatically instrument application code—intercepting HTTP requests, database queries, and framework operations—to create **spans** and propagate the **trace context** across service boundaries without requiring significant manual code changes from developers. This component transforms opaque, unobserved service interactions into a rich stream of structured telemetry data that feeds the entire APM system.

Mental Model: The Invisible Flight Recorder

Imagine every application service as an aircraft equipped with a **Flight Data Recorder (FDR)**, commonly known as a "black box." This recorder automatically and continuously monitors critical systems: engine performance, control surface positions, communication channels, and pilot inputs. It does not require the pilot to manually log each event; instead, it's always on, passively observing and recording. When investigating an incident or analyzing performance, engineers rely on this objective, detailed record of events to reconstruct exactly what happened.

The APM SDK functions as this **Invisible Flight Recorder** for your software. It is embedded within the application process and automatically "records" the journey of each request (the "flight") as it passes through various components (engines, controls). It captures:

- **Takeoff and Landing (Request Start/End):** When a request enters and leaves a service.
- **Control Inputs (Business Logic):** Key operations and decisions within the service.
- **Subsystem Interactions (Outgoing Calls):** Calls to databases, external APIs, or message queues.
- **Environmental Conditions (Context):** Metadata like user ID, deployment version, and hostname.

Just as a flight recorder must be utterly reliable and add minimal weight to the aircraft, the SDK must be robust and have negligible performance overhead. It must also ensure that the recording (the **trace context**) is copied and carried forward whenever the aircraft (request) communicates with another vessel (service), creating a continuous, unbroken chain of evidence across the entire distributed system.

Instrumentation Techniques: Monkey Patching and Middleware

Automatic instrumentation works by intercepting normal application execution at strategic points. Two primary techniques are used, each suited to different layers of the stack.

Technique	Application Layer	How It Works	Key Consideration
Middleware / Interceptors	HTTP Frameworks (Gin, Express), gRPC, Message Queues	The SDK injects itself into the framework's request-handling pipeline. For incoming requests, it creates a root span, extracts the trace context from headers, and makes the context available to the application's handler. For outgoing requests, it wraps the HTTP client or gRPC stub to inject the current trace context into headers before sending.	This is the **preferred, non-invasive method**. It leverages the framework's own extension points (e.g., Gin's `Use()` method, `http.RoundTripper` interface in Go).
Monkey Patching	Database Drivers, Low-level Networking Libraries	The SDK temporarily replaces (or "wraps") the original functions or methods of a library (e.g., `sql.DB.QueryContext`) with its own versions. The wrapper function creates a child span, records the operation (e.g., the SQL query), calls the original function, records the duration and any error, and then closes the span.	This technique is more **invasive and risky**. It must be done carefully to avoid breaking the original library's behavior, causing memory leaks, or losing the context in asynchronous operations. It's often used for libraries that don't provide a native middleware interface.
Code Generation / Compile-time Weaving	All Layers (Advanced)	Instead of modifying behavior at runtime, instrumentation code is injected at compile time. This can be done via code generation (e.g., creating wrapped clients) or through compiler plugins.	This approach can offer **optimal performance and safety** as the instrumentation is "baked in," but it requires more complex build tooling and is language/framework specific. For our initial SDK, we will focus on runtime techniques.

The SDK's internal architecture for a Go service might look like this, coordinating these techniques:

The Core Tracer: At the heart of the SDK is a singleton or factory-managed **Tracer** object. It is responsible for:

1. **Span Creation:** Generating new `Span` instances with unique IDs and linking them to a parent based on the current context.
2. **Context Management:** Maintaining a stack or bag of key-value pairs (the `Attributes`) and the current active span for the duration of a request.
3. **Span Export:** Batching completed spans and sending them asynchronously to the Collector via the configured transport (HTTP/gRPC).

The Instrumentation Process:

1. **HTTP Server (Incoming Request):** A Gin middleware (`APMMiddleware`) is registered via `router.Use()` . For each request:
    1. It extracts the W3C Trace Context headers (`traceparent` , `tracestate`).
    2. It calls `tracer.StartSpanFromContext(ctx, "HTTP GET /api/users")` to create a server-side span. If headers are present, the span becomes a child of the remote parent; otherwise, it starts a new trace.
    3. It stores the new span in the Go `context.Context` and passes this context to the next handler.
    4. After the handler completes, the middleware records the HTTP status code as an attribute, sets the span status (error if 5xx), and calls `span.End()` .
2. **Database Query:** The `database/sql` driver is wrapped via monkey patching at SDK initialization.
    1. When the application calls `db.QueryContext(ctx, "SELECT ...")` , the wrapped driver intercepts the call.
    2. It uses the `ctx` to find the current active trace and calls `tracer.StartSpanFromContext(ctx, "sql.query")` to create a child span.
    3. It adds attributes: `db.system="postgresql"` , `db.statement` (the sanitized query).
    4. It executes the original query, records the duration, adds an error attribute if the query failed, and ends the span.
3. **HTTP Client (Outgoing Request):** The SDK wraps the default `http.Client.Transport` .
    1. When the app makes an outgoing request via `http.Get` , the wrapped transport's `RoundTrip` method is called.
    2. It gets the current span context from the request's `context.Context` .
    3. It injects the trace context into the request headers (`traceparent` , `tracestate`).
    4. It creates a child "http.client" span, executes the request, records the duration and status, and ends the span.

ADR: Context Propagation Mechanism

> **Decision: Use Go's Native `context.Context` for In-Process Propagation and W3C Trace Context for Cross-Service Propagation**
>
> - **Context:** In a distributed trace, the `TraceID` and `ParentSpanID` must be carried from one span to its child, both within the same process (e.g., from an HTTP server span to a database span) and across process boundaries (e.g., from Service A's HTTP client span to Service B's HTTP server span). We need a mechanism that is idiomatic, efficient, and correct for concurrent Go code.
> - **Options Considered:**
>   1. **Explicit Parameter Passing:** Require developers to manually pass a `Trace` or `Span` object through every function call. This is explicit but burdensome and breaks existing code.
>   2. **Thread-Local Storage (TLS):** Use Goroutine-local storage (simulated via maps keyed by goroutine ID). This is fragile because goroutines are multiplexed and a single request may be handled by multiple goroutines. Context can easily be lost.
>   3. **Go's `context.Context`:** Use the standard library's `context.Context` interface to store and retrieve the current span. This is the idiomatic pattern for request-scoped values in Go and is designed to work with cancellation and deadlines.
> - **Decision:** We will use **Go's `context.Context`** for propagation within a single service. For propagation across service boundaries (over HTTP/gRPC), we will use the **W3C Trace Context** standard headers (`traceparent`, `tracestate`).
> - **Rationale:** `context.Context` is the canonical way to pass request-scoped data in Go. It is used by virtually all major frameworks and libraries (Gin, gRPC, `database/sql`), making integration seamless. It is concurrency-safe when passed correctly. The W3C standard ensures interoperability with other tracing systems and languages, future-proofing our implementation.
> - **Consequences:**
>   - **Enables:** Clean, framework-agnostic instrumentation. Developers can manually create spans in their business logic by calling `tracer.StartSpanFromContext(ctx, ...)`. The SDK automatically propagates context in instrumented HTTP/database calls.
>   - **Requires:** Application code must accept and pass a `context.Context` through its call chain, which is already a best practice in modern Go. The SDK must provide utilities to extract and inject the trace context from/to a `context.Context`.

The following table summarizes the key differences between the considered options:

Option	Pros	Cons	Viability
Explicit Parameter Passing	Utterly explicit, no magic. Easy to reason about.	Extremely invasive API. Requires massive refactoring of existing code. Breaks third-party library compatibility.	❌ Rejected
Thread-Local Storage (Goroutine-local)	Conceptually simple for linear, non-concurrent code.	Goroutines are not 1:1 with threads. Context is lost on any goroutine switch (e.g., using `go` keyword, `async`). Highly error-prone and anti-pattern in Go.	❌ Rejected
Go's `context.Context`	Idiomatic and standard. Supported by all major libraries. Designed for request-scoped values and cancellation. Integrates perfectly with middleware patterns.	Requires that the application uses `context.Context` (which is a best practice). Slight learning curve for developers unfamiliar with context.	✅ **Chosen**

Common Pitfalls in SDK Development

Building a reliable, low-overhead auto-instrumentation SDK is fraught with subtle challenges. Awareness of these pitfalls is crucial.

⚠️ Pitfall 1: Performance Overhead from Excessive Allocation and Locking

- **Description:** Creating a span involves allocating several objects (`SpanID`, `TraceID`, `Attributes` map, `Events` slice). If done naively for every database query in a high-throughput service, the garbage collection pressure and locking within the tracer can become significant, effectively slowing down the application it's meant to observe.
- **Why It's Wrong:** An APM tool should be observational, not impactful. High overhead makes developers reluctant to enable tracing in production, defeating its purpose.
- **How to Fix:**
  - **Object Pooling:** Use `sync.Pool` to reuse `Span` and auxiliary objects, reducing allocations.
  - **Asynchronous Export:** Never block the application thread to send spans to the collector. Use a buffered channel and a dedicated exporter goroutine.
  - **Sampling at the Source:** Integrate with the head-based sampler. If a trace is not sampled, avoid creating any span objects or attributes for it entirely.

⚠️ Pitfall 2: Context Loss in Concurrent and Asynchronous Code

- **Description:** In Go, when you launch a new goroutine with `go myFunction()`, the new goroutine does not automatically inherit the `context.Context` (and thus the current span) from the parent goroutine. If `myFunction` makes a database call, it will appear as a new, rootless trace fragment, breaking the parent-child relationship.
- **Why It's Wrong:** This destroys the continuity of the trace, making it impossible to understand the causal relationship between the initial request and the work done in the goroutine. Critical debugging context is lost.
- **How to Fix:** The SDK must provide clear guidance and utilities. For example, provide a helper: `ctx := tracer.ContextWithSpan(context.Background(), currentSpan)` and instruct developers to pass this `ctx` to the new goroutine. For popular async frameworks or worker pools, the SDK can offer specific integrations that automatically propagate context.

⚠️ Pitfall 3: Unsafe Monkey Patching that Breaks Host Application

- **Description:** Replacing a library's function pointer globally can have unintended side effects: it can break other parts of the application that rely on the original function's behavior, cause deadlocks if the wrapper introduces synchronization, or fail if the patching is not done atomically or at the right time (e.g., after another library has already patched the same function).
- **Why It's Wrong:** The SDK becomes a source of instability and weird bugs in the application, eroding trust.
- **How to Fix:**
  - **Prefer Public APIs:** Always use official middleware/interceptor interfaces if available.
  - **Patch Once, Lazily:** Implement patching logic that is idempotent and thread-safe. Often, this is done in an `init()` function or a dedicated `Instrument()` function called at application startup.
  - **Thorough Testing:** Test the wrapped behavior in isolation and in integration with the original library to ensure functional equivalence.

⚠️ Pitfall 4: Leaking Sensitive Data in Spans

- **Description:** Automatically capturing all HTTP request parameters, headers, or full SQL statements can inadvertently record passwords, API keys, or personal data (PII) into the tracing system, creating a security and compliance risk.
- **Why It's Wrong:** This violates data privacy principles and regulations (like GDPR). The trace data, which may be viewed by many engineers or stored in less secure systems, becomes a data leak vector.
- **How to Fix:**
  - **Sanitization:** Provide built-in sanitizers for common data types. For SQL, use a parameter masking library to replace `?` placeholders with a placeholder. For HTTP, provide a configurable blocklist of headers (e.g., `Authorization`, `Cookie`) and query parameters (e.g., `password`) to omit.

- **Customization:** Allow developers to define custom sanitization hooks for their application-specific sensitive fields.

Implementation Guidance for the APM SDK

This guidance provides the foundational code and structure to implement the APM SDK in Go, focusing on the patterns for middleware, context propagation, and safe monkey patching.

A. Technology Recommendations Table

Component	Simple Option (Starting Point)	Advanced Option (Production-ready)
Tracer Core	In-memory span processing with a buffered channel exporter.	Integrate with OpenTelemetry Go SDK as a provider, leveraging its battle-tested core, batching, and retry logic.
HTTP Server Middleware	Custom middleware for the Gin framework.	Generic `net/http` middleware that works with any compatible framework (Gin, Echo, Gorilla Mux).
HTTP Client Wrapping	Wrap the default `http.DefaultTransport`.	Implement a `http.RoundTripper` that can be set on any `http.Client`, supporting connection pooling and redirects.
Database Instrumentation	Wrap `database/sql` driver using `sql.Register`.	Use the OpenTelemetry SQL driver wrapper (`go.opentelemetry.io/contrib/instrumentation/database/sql`).
Context Propagation	Store current span in `context.Context` using a private key.	Implement the OpenTelemetry `propagation.TextMapPropagator` interface for W3C Trace Context.

B. Recommended File/Module Structure

The SDK should be organized as a separate Go module that applications can import.

```
apm-sdk-go/                  # Root of the SDK module
├── go.mod
├── go.sum
├── tracer/                  # Core tracing logic
│   ├── tracer.go           # Main Tracer struct, Span creation
│   ├── context.go          # Context propagation utilities (WithSpan, SpanFromContext)
│   ├── exporter.go         # Interface and channel-based exporter to Collector
│   └── config.go           # SDK configuration (service name, sampling rate, endpoint)
├── instruments/            # Auto-instrumentation packages
│   ├── http/              # HTTP instrumentation
│   │   ├── server/        # Server-side middleware
│   │   │   ├── gin.go     # Gin framework middleware
│   │   │   └── middleware.go # Generic net/http middleware
│   │   └── client/        # Client-side wrapper
│   │       └── wrapper.go  # http.RoundTripper implementation
│   ├── sql/               # Database instrumentation
│   │   └── driver.go      # sql driver wrapper and registration
│   └── grpc/              # gRPC interceptors (future)
│       ├── client.go
│       └── server.go
├── propagation/           # Cross-service context propagation
│   ├── w3c.go            # W3C Trace Context extract/inject
│   └── propagator.go     # Propagator interface
└── internal/             # Private utilities
    ├── sanitize.go       # SQL/query sanitization
    └── pool.go           # Object pools for Spans
```

C. Infrastructure Starter Code

Here is a complete, ready-to-use implementation for the core context propagation utilities, which are a prerequisite for all instrumentation.

```go
// File: tracer/context.go

package tracer


import (

    "context"

)


type contextKey struct{}


var activeSpanKey = &contextKey{}


// ContextWithSpan returns a new context derived from parentCtx that contains the given span.

// This span becomes the "active" span for any span creation or instrumentation that uses this context.

func ContextWithSpan(parentCtx context.Context, span *Span) context.Context {

    return context.WithValue(parentCtx, activeSpanKey, span)

}


// SpanFromContext retrieves the active span from the given context.

// It returns nil if no span is present in the context.

func SpanFromContext(ctx context.Context) *Span {

    if s, ok := ctx.Value(activeSpanKey).(*Span); ok {

        return s

    }

    return nil

}


// File: propagation/w3c.go

package propagation


import (

    "context"

    "fmt"

    "strings"

)


const (

    TRACE_ID_HEADER = "traceparent"
```

```
    // Note: SPAN_ID_HEADER is "tracestate" per naming conventions, but traceparent contains both trace and
parent span ID.

)


// ExtractTraceContext reads the W3C Trace Context headers from a carrier (e.g., http.Request)

// and returns a context containing the remote span context.

func ExtractTraceContext(ctx context.Context, carrier TextMapCarrier) context.Context {

    traceParent := carrier.Get(TRACE_ID_HEADER)

    if traceParent == "" {

        return ctx // No incoming context, will start a new trace.

    }

    // Parse traceParent format: version-traceId-parentSpanId-flags

    parts := strings.Split(traceParent, "-")

    if len(parts) != 4 {

        // Malformed header, ignore.

        return ctx

    }

    traceID := parts[1]

    parentSpanID := parts[2]


    // Here you would create a SpanContext object and attach it to the context.

    // For simplicity, we return a context with a placeholder.

    // In a full implementation, you would call tracer.StartSpan with the extracted parent.

    return ctx

}


// InjectTraceContext writes the current span's context from the given context into the carrier

// (e.g., http.Request headers) for propagation to the next service.

func InjectTraceContext(ctx context.Context, carrier TextMapCarrier) error {

    span := tracer.SpanFromContext(ctx)

    if span == nil {

        return nil // Nothing to inject

    }

    // Format: version-traceId-parentSpanId-flags (flags sampled = 01)
```

```
    // For simplicity, we assume version 00 and sampled flag.
    traceParent := fmt.Sprintf("00-%s-%s-01", span.TraceID, span.SpanID)
    carrier.Set(TRACE_ID_HEADER, traceParent)
    return nil
}

// TextMapCarrier is an interface for objects that contain string key-value pairs (e.g., http.Header,
metadata.MD).
type TextMapCarrier interface {
    Get(key string) string
    Set(key, value string)
}
```

D. Core Logic Skeleton Code

Below is the skeleton for the core Tracer and key instrumentation components. The TODOs map directly to the algorithmic steps described in the prose.

GO

```go
// File: tracer/tracer.go

package tracer


import (

    "context"

    "crypto/rand"

    "encoding/hex"

    "time"

)


// Tracer is the main entry point for creating spans and managing the export pipeline.

type Tracer struct {

    serviceName string

    exporter    SpanExporter

    sampler     HeadSampler // Integrated head-based sampling

}


// StartSpanFromContext creates a new span as a child of the active span in the given context.

// If no active span exists, it starts a new trace.

func (t *Tracer) StartSpanFromContext(ctx context.Context, name string) (context.Context, *Span) {

    // TODO 1: Check if the trace is already sampled (head-based decision).

    //   - Get the traceID from the parent span if it exists, or generate a new one.

    //   - Call t.sampler.Decide(traceID, t.serviceName) to get the sampling decision.

    //   - If NOT sampled, return a "no-op" span that does nothing on End().


    // TODO 2: Generate a unique SpanID (16 bytes random, hex encoded).


    // TODO 3: Determine parent relationship.

    //   - If parentSpan := SpanFromContext(ctx); parentSpan != nil, set this span's ParentSpanID and TraceID
from parent.

    //   - Otherwise, generate a new TraceID and set ParentSpanID to empty (root span).


    // TODO 4: Create a new Span struct with StartTime = time.Now().


    // TODO 5: Store this new span as the active span in a new context (using ContextWithSpan).


    // TODO 6: Return the new context and the span.
```

```go
    return ctx, &Span{}
}

// Span represents an individual operation within a trace.
// (Fields are defined per naming conventions; we add methods.)
type Span struct {
    SpanID       string
    TraceID      string
    ParentSpanID string
    Name         string
    ServiceName  string
    StartTime    time.Time
    Duration     time.Duration
    Attributes   map[string]string
    Events       []SpanEvent
    Status       SpanStatus
}

// End finalizes the span, records its duration, and schedules it for export.
func (s *Span) End() {
    // TODO 1: Calculate duration: time.Now() - s.StartTime.

    // TODO 2: If this span is a no-op (not sampled), do nothing and return.

    // TODO 3: Send the completed span to the exporter's buffer (e.g., via a channel).
    //   - The exporter should run in a separate goroutine to batch and send spans asynchronously.
}

// File: instruments/http/client/wrapper.go
package http

import (
    "context"
    "net/http"
    "apm-sdk-go/tracer"
```

```
    "apm-sdk-go/propagation"
)

// WrappedTransport implements http.RoundTripper and injects trace context.
type WrappedTransport struct {
    Base http.RoundTripper
    Tracer *tracer.Tracer
}

// RoundTrip executes the HTTP request, creating a span and propagating context.
func (wt *WrappedTransport) RoundTrip(req *http.Request) (*http.Response, error) {
    // TODO 1: Start a child span for this outgoing request.
    //   - Use wt.Tracer.StartSpanFromContext(req.Context(), "http.client").
    //   - Add attributes: "http.method", "http.url", "peer.hostname".

    // TODO 2: Inject the current trace context into the request headers.
    //   - Call propagation.InjectTraceContext(ctx, propagation.HeaderCarrier(req.Header)).

    // TODO 3: Execute the underlying request using wt.Base.RoundTrip(req).

    // TODO 4: Record the result.
    //   - Add attribute "http.status_code".
    //   - If the request failed or status >= 400, mark span status as error.

    // TODO 5: End the span.

    // TODO 6: Return the response and error.
    return wt.Base.RoundTrip(req)
}

// File: instruments/sql/driver.go
package sql

import (
    "context"
    "database/sql"
    "database/sql/driver"
```

```
    "apm-sdk-go/tracer"
)

// WrapDriver returns a new SQL driver that instruments all queries.
func WrapDriver(baseDriver driver.Driver, driverName string, tracer *tracer.Tracer) {
    // TODO 1: Create a wrapped driver struct that implements driver.Driver interface.
    //   - It should embed or wrap the baseDriver.

    // TODO 2: Override the Open method to return a wrapped connection.

    // TODO 3: Register this wrapped driver with a new name (e.g., "instrumented-mysql")
    //   using sql.Register.

    // The application then uses "instrumented-mysql" as the driver name in its DSN.
}
```

E. Language-Specific Hints

- **Context is Key:** Always pass `context.Context` as the first parameter in your instrumentation functions. Use `context.TODO()` only as a placeholder during development.
- **Safe Concurrency with Channels:** Use a buffered channel (`chan *Span`) for the exporter. A dedicated goroutine should read from this channel, batch spans, and send them to the collector. This prevents the instrumentation from blocking the application.
- **Use `sync.Pool` for Spans:** Span creation is frequent. Pooling can drastically reduce allocation pressure.

```go
    var spanPool = sync.Pool{
        New: func() interface{} { return &Span{Attributes: make(map[string]string)} },
    }
```

- **Monkey Patching with `sql.Register` :** The `database/sql` package is designed for wrapping. Implement the `driver.Driver` interface, wrap the original driver, and register your wrapper. This is safe and standard.

F. Milestone Checkpoint

After implementing the SDK and instrumenting a sample Go application (e.g., a simple Gin server that makes database calls and HTTP requests to another service), you should be able to verify:

1. **Start the instrumented application:** Run your app with the SDK initialized.
2. **Generate traffic:** Use `curl` or a load generator to send HTTP requests to your app.
3. **Check Collector logs:** Ensure spans are being received at the Collector's ingestion endpoint.
4. **Verify Trace Completeness:** Query the stored trace via the Collector's API. The trace should show:
    - A root span for the incoming HTTP request.
    - A child span for the database query.
    - A child span for any outgoing HTTP call, with the W3C headers present in the downstream service's logs.
    - All spans should have the same `TraceID` .

5. **Performance Baseline:** Measure the overhead. The 95th percentile (p95) latency of instrumented endpoints should not increase by more than 1-2% compared to non-instrumented code under load. Use a benchmark tool like `wrk` or `vegeta`.

Signs of Trouble:

- **No spans in Collector:** Check that the SDK's exporter is running and can connect to the Collector endpoint. Verify the sampling rate is not 0%.
- **Spans are not linked (different TraceIDs):** Context propagation is broken. Verify the `SpanFromContext` and `ContextWithSpan` functions, and ensure the W3C headers are being injected and extracted correctly in HTTP clients/servers.
- **High memory or CPU usage in the app:** The SDK's exporter might be blocking or the object pools might be ineffective. Profile the application using `go tool pprof`.

---

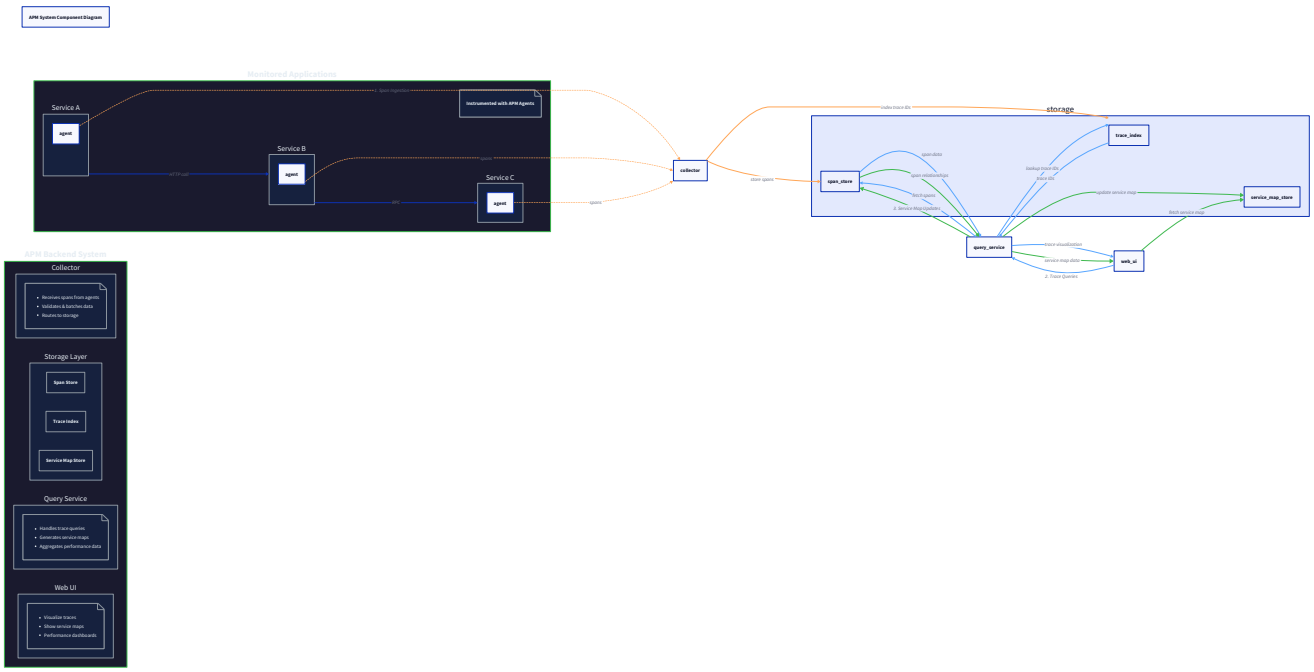
10. Interactions and Data Flow

> **Milestone(s):** This section synthesizes concepts from all five milestones (Trace Collection, Service Map, Trace Sampling, Performance Analytics, and APM SDK), illustrating how the entire system works together to transform raw instrumentation data into actionable insights. Understanding these interactions is critical for debugging the system itself and ensuring data flows correctly through the pipeline.

At its core, the APM Tracing System is a **data processing pipeline** that transforms individual spans emitted by instrumented applications into comprehensive traces, service maps, and performance insights. This section traces the complete journey of a single user request—from the moment it enters the first microservice until its performance data appears in dashboards and alerts. Think of this journey as a **documentary film crew following a character** through a complex city: the SDKs are the cameras that capture footage (spans), the collector is the editing studio that assembles the raw clips, and the analytics components are the producers who extract meaning from the final film.

Sequence: The Lifecycle of a Trace

The lifecycle of a trace can be broken down into eight distinct phases, each involving multiple system components. The following diagram illustrates the high-level component interactions:



Phase 1: Instrumentation and Span Creation

When a user request hits the first service (e.g., an API gateway), the auto-instrumented HTTP server middleware detects the incoming request and begins the trace lifecycle:

1. **Context Extraction**: The SDK's HTTP middleware examines the incoming request headers for W3C Trace Context headers (`traceparent` and `tracestate`). If present, it extracts the `TraceID` , `ParentSpanID` , and trace flags. If absent (indicating this is the root of a new trace), it generates a new 128-bit `TraceID` using a cryptographically secure random number generator.
2. **Root Span Creation**: The middleware creates a new `Span` object with:
   - `SpanID` : A new 64-bit identifier for this span
   - `TraceID` : Either extracted from headers or newly generated
   - `ParentSpanID` : Empty for root spans, or the extracted parent span ID
   - `Name` : The HTTP route (e.g., `GET /api/orders`)
   - `ServiceName` : The name of the current service (from configuration)
   - `StartTime` : Current timestamp with nanosecond precision
   - `Attributes` : HTTP method, URL, user agent, and other request metadata
3. **Context Storage**: The SDK stores this span in Go's `context.Context` using the private key `activeSpanKey` , making it accessible to all downstream function calls within the same request processing chain. This context is passed through the application's call stack.
4. **Head-Based Sampling Decision**: Before adding substantial overhead, the SDK consults the configured `HeadSampler` by calling `Decide(ctx, traceID, serviceName)` . The sampler uses consistent hashing on the `TraceID` to make a deterministic decision:
   - If the decision is **DROP**, the span is marked as `Sampled = false` and will only record minimal metadata (trace ID, span ID, sampling decision).
   - If the decision is **KEEP**, the span proceeds with full instrumentation.

Phase 2: Context Propagation and Child Span Creation

As the request flows through the service's internal components and makes outbound calls, the SDK automatically creates child spans:

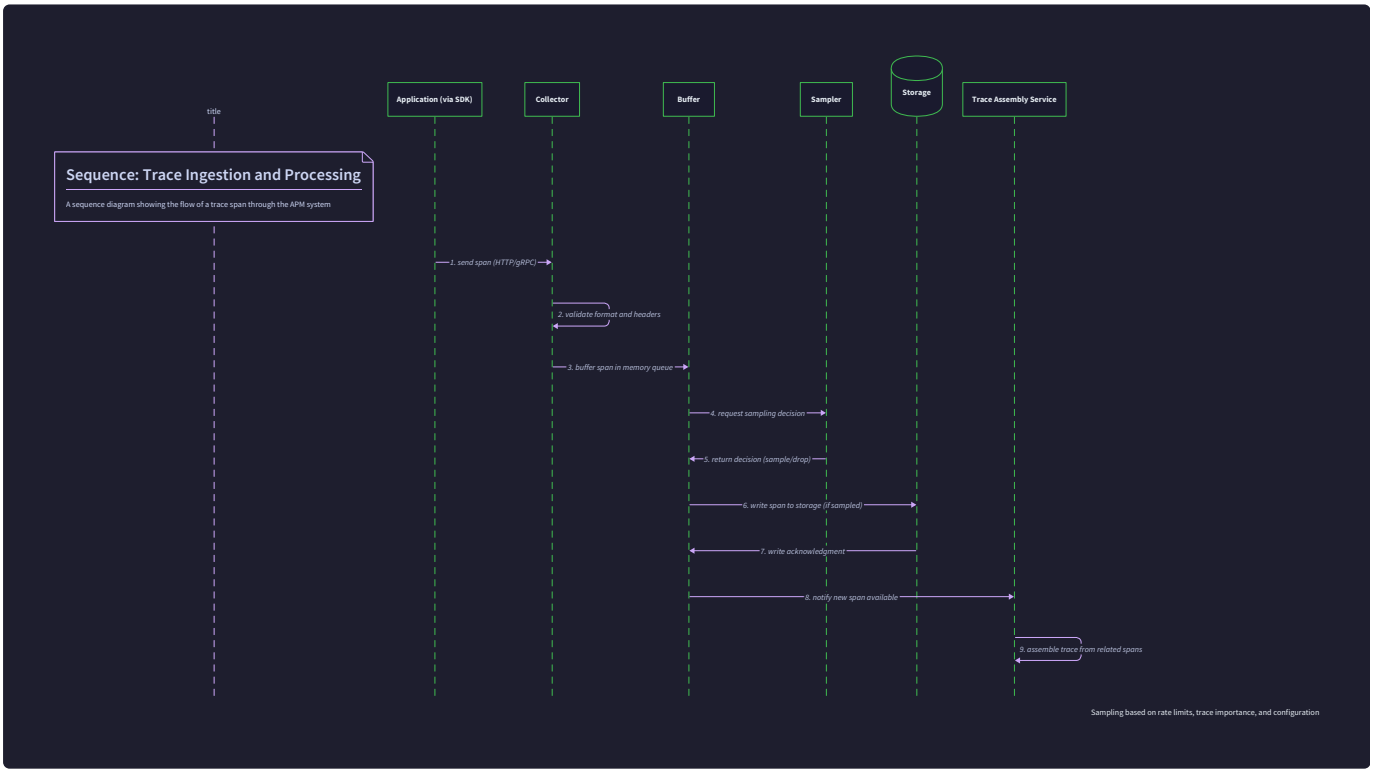
1. **Database Call**: When the service executes a SQL query through the instrumented database driver:
   - The driver wrapper extracts the active span from the `context.Context`
   - Creates a child span with `ParentSpanID` set to the current span's ID
   - Records the SQL operation type, table name, and sanitized query parameters
   - Measures query execution time from request to response
   - On completion, calls `End()` on the span, which records the duration
2. **Outbound HTTP Call**: When the service calls another microservice:
   - The instrumented HTTP client wrapper extracts the active span context
   - Creates a child span for the outbound request
   - Injects the trace context into HTTP headers using `InjectTraceContext()` , setting `traceparent` header with format `00-{TraceID}-{SpanID}-{TraceFlags}`
   - Sends the request and records response status code and timing
   - On the receiving service, the server middleware extracts this context (as described in Phase 1), maintaining the parent-child relationship across service boundaries
3. **Async Operations**: For asynchronous operations (goroutines, message queue processing), the SDK provides mechanisms to preserve trace context across async boundaries. In Go, this typically involves creating a new context from the parent span and

passing it explicitly to the goroutine.

Phase 3: Span Export to Collector

When a span completes (via `End()` method), the SDK's `SpanExporter` component queues it for export. The export process follows this sequence:

1. **Batch Assembly**: The exporter collects completed spans into batches (typically 100 spans or every 5 seconds, whichever comes first) to optimize network utilization.
2. **Transport Selection**: The batch is serialized into OpenTelemetry Protocol (OTLP) format and sent via either HTTP/JSON or gRPC/protobuf to the Collector's ingestion endpoint. The diagram below shows this ingestion sequence:



3. **Network Dispatch**: The SDK transmits the batch asynchronously, with retry logic for transient network failures. If the Collector is unavailable, spans may be buffered in memory (with size limits) or written to local disk to prevent memory exhaustion.

Phase 4: Collector Ingestion and Buffering

Upon receiving a batch of spans, the Collector's ingestion pipeline processes each span through a multi-stage pipeline:

1. **Request Handling**: The Collector's HTTP or gRPC server receives the batch, validates authentication/authorization (if configured), and parses the OTLP payload into internal `Span` objects.
2. **Validation and Enrichment**: Each span undergoes validation:
   - Required fields (`TraceID` , `SpanID`) must be present and properly formatted
   - Timestamps must be within acceptable bounds (not too far in future or past)
   - Span duration must be non-negative
   - Malformed spans are logged and discarded, but the batch continues processing
3. **Head Sampling Verification**: The Collector verifies the head sampling decision by recalculating the hash-based decision. This ensures consistency between SDK and Collector (important for edge cases where SDK configuration differs). If the Collector's decision differs from the span's `Sampled` flag, it respects the SDK's decision but logs a warning.
4. **Buffer Management**: The validated span is passed to the `BufferManager` via `AddSpan(ctx, span)` . The manager:

- Looks up the trace buffer for the span's `TraceID`
- If no buffer exists, creates a new `TraceBuffer` with `firstSeen` set to current time
- Appends the span to the buffer's `spans` slice
- Updates the buffer's `lastSeen` timestamp
- Triggers eviction if total buffers exceed `maxSize` (oldest or least-recently-used traces are removed based on `evictionPolicy`)

5. **Write-Ahead Logging**: For durability, the span is simultaneously appended to the `WALWriter` via `Append()`. The WAL record contains the span data plus metadata (arrival timestamp, source service). This allows recovery of in-flight traces after Collector restart.

Phase 5: Trace Assembly and Storage

When certain conditions are met, traces are assembled from their constituent spans and persisted to storage:

1. **Trace Completion Detection**: The `TraceAssembler` periodically scans buffers to identify complete traces:
   - A trace is considered **potentially complete** if `current_time - buffer.lastSeen > maxTraceDuration` (e.g., 5 minutes)
   - The assembler calls `IsTraceComplete()` which examines span parent-child relationships to check for missing spans
   - For incomplete traces with expired timeout, the system proceeds anyway (handling late arrivals via the WAL recovery mechanism)
2. **Tail-Based Sampling Evaluation**: For each potentially complete trace, the `TailSampler` evaluates whether to keep it:
   - The trace is passed to `EvaluateTrace(ctx, trace, headDecision)`
   - Each `TailSamplingRule` is evaluated in priority order:
     - Rule 1: Keep all traces with error status codes (HTTP 5xx, span status = ERROR)
     - Rule 2: Keep traces with latency above the 99th percentile for their service/operation
     - Rule 3: Random sampling for "normal" traces to maintain baseline coverage
   - If any rule matches with `KeepIfMatch = true`, the trace is kept regardless of head decision
   - Statistics are updated in `TailSamplerStats`
3. **Trace Assembly**: For traces selected for storage, `AssembleTrace(ctx, spans)` constructs a hierarchical `Trace` object:
   - Sorts spans by `StartTime`
   - Reconstructs parent-child relationships using `ParentSpanID` references
   - Calculates trace-level `StartTime` (earliest span start) and `EndTime` (latest span end)
   - Validates temporal consistency (child spans must start after parent and end before parent)
4. **Storage Persistence**: The assembled trace is written to the storage backend:
   - **Primary Storage**: All spans are written with `TraceID` as the primary index, enabling efficient retrieval of all spans for a trace
   - **Secondary Indexes**: Index entries are created for:
     - `ServiceName` + `StartTime` range (for service-specific queries)
     - `StartTime` alone (for time-range queries)
     - `Attributes` (for specific tag queries, if supported)
   - **Aggregation Updates**: The span data is simultaneously sent to the analytics pipeline

Phase 6: Service Map Construction

As traces are assembled, service dependency information is extracted and aggregated:

1. **Edge Extraction**: For each inter-service span (where parent and child spans have different `ServiceName` values), the `EdgeAggregator` extracts a service call relationship:

- `CallerService`: The service name from the parent span
- `CalleeService`: The service name from the child span
- Latency, error status, and other metadata from the child span

2. **Windowed Aggregation**: The aggregator's `ProcessSpan(ctx, span)` method updates in-memory aggregates for the current time window (e.g., 1 minute):
   - Maintains `TotalCalls`, `ErrorCount`, latency histograms for each unique caller-callee pair
   - Tracks a sample of `TraceID` values for each edge (for drill-down capability)
3. **Window Flush**: At the end of each aggregation window, `FlushWindow(ctx, windowStart)`:
   - Calculates final metrics: `ErrorRate = ErrorCount / TotalCalls`, latency percentiles (p50, p95, p99)
   - Persists the aggregated `ServiceEdge` to the `EdgeStorage`
   - Resets in-memory aggregates for the next window
4. **Graph Construction**: The `GraphBuilder` periodically calls `BuildCurrentGraph(ctx, windowSize)` to construct a complete `ServiceGraph`:
   - Retrieves all edges from the specified time window
   - Deduplicates nodes (services) and creates `ServiceNode` entries
   - Detects topology changes by comparing with previous graph via `DetectTopologyChanges()`
   - Updates the visualization layer with the new graph

Phase 7: Performance Analytics Processing

Simultaneously with trace storage and service map construction, spans flow through the performance analytics pipeline:

1. **Latency Aggregation**: For each span with duration information, the `PercentileAggregator` updates its t-digest structures:
   - Creates or retrieves a `TDigestMetric` for the key `service:operation`
   - Calls `Add(float64(span.Duration))` to incorporate the latency measurement
   - Maintains separate aggregations per time window (e.g., 1-minute windows)
2. **Anomaly Detection**: At regular intervals (e.g., every 10 seconds), the `Detector` evaluates current metrics against historical baselines:
   - For each service-operation pair, retrieves current p95 latency via `GetPercentiles()`
   - Calls `CheckMetric(ctx, "latency_p95", labels, currentValue)` which:
     - Calculates expected value using `BaselineCalculator.Calculate()` (considering time-of-day, day-of-week patterns)
     - Computes z-score: `(current - expected) / standard_deviation`
     - If z-score exceeds threshold (e.g., 3.0), generates an `AnomalyResult`
   - For high-severity anomalies, triggers alert notifications
3. **Time-Series Storage**: Aggregated metrics (percentiles, call counts, error rates) are written to the time-series database at regular intervals (e.g., every minute) for long-term retention and trend analysis.

Phase 8: Query and Visualization

When an engineer needs to investigate an issue, they interact with the system through the Web UI and Query Service:

1. **Trace Search**: The engineer queries for traces by service, time range, or attributes:
   - Query Service translates the UI request into storage queries:
     - `GetTracesByService(ctx, "payment-service", startTime, endTime, 100)`
     - `GetTracesByTimeRange(ctx, startTime, endTime, 50)`
   - Storage retrieves trace IDs from secondary indexes, then fetches full span data

- Traces are returned in chronological order with hierarchical span display

2. **Service Map Visualization**: The UI requests the current service map:
   - Query Service calls `BuildCurrentGraph(ctx, "5m")` or retrieves a pre-computed graph
   - Returns nodes (services) and edges (calls with metrics) to the UI
   - UI renders interactive graph with node size proportional to request volume and edge color indicating error rate
3. **Performance Dashboard**: The UI queries time-series metrics for display:
   - Historical latency percentiles (p50, p95, p99) for selected services
   - Error rate trends and anomaly alerts
   - Comparison views (this week vs. last week)
4. **Drill-Down Investigation**: From any visualization, engineers can drill down to individual traces:
   - Clicking a service map edge shows sample traces for that service pair
   - Clicking an anomaly alert shows the affected traces with high latency
   - The trace detail view shows the complete waterfall diagram of spans with timing and error information

Key Message and Wire Formats

The system uses several well-defined wire formats for communication between components. Understanding these formats is essential for debugging interoperability issues and extending the system.

OpenTelemetry Protocol (OTLP) Span Format

The primary external interface uses the OpenTelemetry Protocol (OTLP), which supports both HTTP/JSON and gRPC/protobuf encodings. The following table describes the key fields in the OTLP span representation and their mapping to our internal `Span` type:

OTLP Field	Type	Description	Mapping to `Span` Field
`trace_id`	bytes (16 bytes)	Unique identifier for the trace, represented as 32 hex characters when encoded as text	`TraceID` (string hex representation)
`span_id`	bytes (8 bytes)	Unique identifier for the span within the trace, 16 hex characters as text	`SpanID` (string hex representation)
`trace_state`	string	Comma-separated list of key-value pairs representing vendor-specific trace state	Not directly mapped; stored in `Attributes` under `"w3c.tracestate"`
`parent_span_id`	bytes (8 bytes)	Identifier of the parent span; empty for root spans	`ParentSpanID` (string hex representation)
`name`	string	Span name (operation name)	`Name`
`kind`	enum	Span kind: INTERNAL, SERVER, CLIENT, PRODUCER, CONSUMER	Derived into `Attributes["span.kind"]`
`start_time_unix_nano`	fixed64	Start time in nanoseconds since Unix epoch	`StartTime` (converted from nanoseconds to `time.Time`)
`end_time_unix_nano`	fixed64	End time in nanoseconds since Unix epoch	`Duration` = `EndTime - StartTime`
`attributes`	key-value list	Span attributes (tags) as string, bool, int, double, or array values	`Attributes` (string values only; complex types stringified)
`dropped_attributes_count`	uint32	Count of attributes that were dropped due to limits	Logged but not stored
`events`	event list	Time-stamped events with attributes	`Events` (mapped to `SpanEvent` list)
`dropped_events_count`	uint32	Count of events that were dropped	Logged but not stored
`links`	link list	Links to other spans (for batch operations)	Not directly supported in current model
`dropped_links_count`	uint32	Count of links that were dropped	Logged but not stored
`status`	Status	Span status: code (Ok, Error, Unset) and optional description	`Status.Code` and `Status.Message`

HTTP Endpoint Example: The Collector exposes an HTTP endpoint at `POST /v1/traces` accepting JSON with this structure:

```json
{
  "resourceSpans": [{
    "resource": {"attributes": [{"key": "service.name", "value": {"stringValue": "payment-service"}}]},
    "scopeSpans": [{
      "spans": [{
        "traceId": "4bf92f3577b34da6a3ce929d0e0e4736",
        "spanId": "00f067aa0ba902b7",
        "parentSpanId": "",
        "name": "process_payment",
        "kind": "SPAN_KIND_INTERNAL",
        "startTimeUnixNano": "1616234190000000000",
        "endTimeUnixNano": "1616234190500000000",
        "attributes": [{"key": "http.method", "value": {"stringValue": "POST"}}],
        "status": {"code": "STATUS_CODE_OK"}
      }]
    }]
  }]
}
```

Internal Span Representation

While the Collector accepts OTLP format, it immediately converts spans to the internal representation defined in Section 4. The following table shows the complete internal `Span` structure and its persistence format:

Field	Type in Go	Storage Format	Description
`SpanID`	`string`	UTF-8 string (16 chars)	Hex representation of 8-byte span ID
`TraceID`	`string`	UTF-8 string (32 chars)	Hex representation of 16-byte trace ID
`ParentSpanID`	`string`	UTF-8 string (0 or 16 chars)	Empty for root spans, otherwise hex representation
`Name`	`string`	UTF-8 string	Operation name, max 255 characters
`ServiceName`	`string`	UTF-8 string	Service identifier, max 100 characters
`StartTime`	`time.Time`	int64 (nanoseconds)	Nanoseconds since Unix epoch, UTC
`Duration`	`time.Duration`	int64 (nanoseconds)	Duration in nanoseconds
`Attributes`	`map[string]string`	JSON object	Key-value pairs, keys max 255 chars, values max 1024 chars
`Events`	`[]SpanEvent`	JSON array	List of timed events during span execution
`Status.Code`	`int`	int32	0=Unset, 1=Ok, 2=Error
`Status.Message`	`string`	UTF-8 string	Optional status description
`Sampled`	`bool`	boolean	Whether this trace was sampled (not part of OTLP)
`ReceivedAt`	`time.Time`	int64 (nanoseconds)	When collector received the span (internal use)

Storage Optimization: In the storage layer, spans are typically stored in a columnar format with compression. Frequently queried fields (`TraceID` , `ServiceName` , `StartTime`) are stored in separate columns with dictionary encoding, while larger fields (`Attributes` , `Events`) are stored in a compressed JSON blob.

Collector Internal Messages

Between internal Collector components, spans are passed as in-memory `Span` objects, but several key internal APIs use structured messages. The most important is the **Trace Completion Notification** sent from the `BufferManager` to the `TraceAssembler` :

Field	Type	Description
`TraceID`	`string`	The trace identifier
`SpanCount`	`int`	Number of spans in the trace
`FirstSeen`	`time.Time`	When the first span for this trace arrived
`LastSeen`	`time.Time`	When the most recent span arrived
`IsTimedOut`	`bool`	Whether trace completion is due to timeout (vs. explicit completion)
`BufferSize`	`int`	Current memory usage of this trace's buffer

Query Service APIs

The Query Service exposes RESTful endpoints for trace retrieval. The key endpoints and their request/response formats are:

GET /api/traces/{traceId} *Request:* Path parameter `traceId` (32-character hex string) *Response:*

```json
{
  "traceId": "4bf92f3577b34da6a3ce929d0e0e4736",
  "spans": [
    {
      "spanId": "00f067aa0ba902b7",
      "parentSpanId": "",
      "name": "process_payment",
      "serviceName": "payment-service",
      "startTime": "2021-03-20T10:56:30Z",
      "durationMs": 50,
      "attributes": {"http.method": "POST"},
      "status": {"code": 1}
    }
  ],
  "startTime": "2021-03-20T10:56:30Z",
  "endTime": "2021-03-20T10:56:30.050Z",
  "durationMs": 50,
  "serviceCount": 1
}
```

GET /api/traces *Query Parameters:*

- `service` (optional): Filter by service name
- `operation` (optional): Filter by operation name
- `start` (required): Start time in ISO 8601 or Unix milliseconds
- `end` (required): End time in ISO 8601 or Unix milliseconds
- `limit` (optional, default=100): Maximum number of traces to return
- `minDuration` (optional): Filter by minimum trace duration (ms)
- `maxDuration` (optional): Filter by maximum trace duration (ms)
- `tags` (optional): Key-value pairs to match in span attributes

Response: Array of trace summaries (not full span details):

```json
{
  "traces": [
    {
      "traceId": "4bf92f3577b34da6a3ce929d0e0e4736",
      "rootService": "payment-service",
      "rootOperation": "process_payment",
      "startTime": "2021-03-20T10:56:30Z",
      "durationMs": 50,
      "spanCount": 3,
      "serviceCount": 2,
      "hasError": false
    }
  ],
  "total": 1,
  "limit": 100,
  "offset": 0
}
```

GET /api/services/{serviceName}/operations *Request:* Path parameter `serviceName` *Response:*

```json
{
  "service": "payment-service",
  "operations": [
    {"name": "process_payment", "count": 1500},
    {"name": "validate_card", "count": 1500},
    {"name": "update_inventory", "count": 800}
  ]
}
```

GET /api/services/{serviceName}/latency *Query Parameters:*

- `operation` (optional): Specific operation name
- `start` (required): Start time
- `end` (required): End time
- `percentile` (optional, default=95): Percentile to calculate (50, 95, 99)

Response:

```json
{
  "service": "payment-service",
  "operation": "process_payment",
  "percentiles": {
    "p50": 45.2,
    "p95": 128.7,
    "p99": 245.3
  },
  "unit": "milliseconds",
  "sampleSize": 1500,
  "timeWindow": {
    "start": "2021-03-20T10:00:00Z",
    "end": "2021-03-20T11:00:00Z"
  }
}
```

GET /api/service-map *Query Parameters:*

- `window` (optional, default="5m"): Time window for aggregation (e.g., "5m", "1h", "1d")
- `time` (optional): Reference time for the window (defaults to now)

Response: Complete service graph:

```json
{
  "generatedAt": "2021-03-20T11:05:00Z",
  "windowSize": "5m",
  "nodes": [
    {"name": "payment-service", "totalCalls": 1500},
    {"name": "inventory-service", "totalCalls": 800},
    {"name": "user-service", "totalCalls": 700}
  ],
  "edges": [
    {
      "caller": "payment-service",
      "callee": "inventory-service",
      "totalCalls": 800,
      "errorCount": 12,
      "errorRate": 0.015,
      "p50LatencyMs": 25.4,
      "p95LatencyMs": 89.1,
      "p99LatencyMs": 210.5,
      "sampleTraceIds": ["4bf92f3577b34da6...", "5ac93e4688c45eb7..."]
    }
  ]
}
```

W3C Trace Context Propagation

For context propagation between services, the system uses W3C Trace Context headers:

Header	Format	Description
`traceparent`	`00-{trace-id}-{span-id}-{flags}`	Required header with version (00), trace ID (32 hex), span ID (16 hex), and flags (2 hex)
`tracestate`	`key1=value1,key2=value2`	Optional comma-separated list of vendor-specific tracing system states
`traceresponse`	(Proposed)	Future header for returning trace context from server to client

Example traceparent: `00-0af7651916cd43dd8448eb211c80319c-b7ad6b7169203331-01`

- `00`: Version (current version is 00)
- `0af7651916cd43dd8448eb211c80319c`: 32-character hex trace ID (16 bytes)
- `b7ad6b7169203331`: 16-character hex parent span ID (8 bytes)

- `01`: Flags (bit field: 01 = sampled, 02 = debug, etc.)

The SDK's `TextMapCarrier` interface abstracts these headers for both HTTP and other propagation mechanisms (gRPC metadata, message queues, etc.).

Implementation Guidance

> **Note:** This section provides practical guidance for implementing the data flow and wire format handling described above.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Span Transport	HTTP/JSON with `net/http`	gRPC with Protocol Buffers using OpenTelemetry protobuf definitions
JSON Serialization	Standard `encoding/json`	`json-iterator/go` for faster serialization with compatibility
Protocol Buffers	Manually defined .proto files	Import `opentelemetry-proto` from GitHub for official OTLP definitions
HTTP Server	`net/http` with Gorilla Mux	`chi` router for lightweight routing with middleware support
WAL Format	JSON lines with newline delimiter	Binary format with length-prefixed protobuf messages
Context Propagation	Manual header manipulation	OpenTelemetry Go SDK's `propagation` package

B. Recommended File/Module Structure

```
apm-tracing-system/
├── cmd/
│   ├── collector/                # Collector main binary
│   │   └── main.go
│   ├── query-service/            # Query service main binary
│   │   └── main.go
│   └── web-ui/                   # Web UI server (optional)
│       └── main.go
├── internal/
│   ├── apm/                      # Shared APM data structures
│   │   ├── span.go               # Span, Trace, Service types
│   │   ├── wire_formats.go       # OTLP JSON/protobuf marshal/unmarshal
│   │   └── w3c_trace_context.go  # Trace context header handling
│   ├── collector/                # Collector component
│   │   ├── server/
│   │   │   ├── http_server.go    # HTTP ingestion endpoint
│   │   │   └── grpc_server.go    # gRPC ingestion endpoint
│   │   ├── ingestion/
│   │   │   ├── pipeline.go       # Main ingestion pipeline
│   │   │   ├── validator.go      # Span validation logic
│   │   │   └── buffer_manager.go # Trace buffering
│   │   ├── sampling/
│   │   │   ├── head_sampler.go   # Head-based sampling
│   │   │   └── tail_sampler.go   # Tail-based sampling
│   │   └── storage/
│   │       ├── writer.go         # Storage interface implementation
│   │       └── wal.go            # Write-ahead log
│   ├── query/                    # Query service component
│   │   ├── handler.go            # HTTP request handlers
│   │   ├── service.go            # Business logic for queries
│   │   └── storage_reader.go     # Storage read interface
│   ├── servicemap/               # Service map component (Milestone 2)
│   │   ├── edge_aggregator.go
│   │   ├── graph_builder.go
│   │   └── storage.go
│   ├── analytics/                # Analytics component (Milestone 4)
│   │   ├── percentile_aggregator.go
│   │   ├── anomaly_detector.go
│   │   └── timeseries_store.go
│   └── sdk/                      # APM SDK (Milestone 5)
│       ├── tracer.go
│       ├── instrumentation/
│       │   ├── http.go
│       │   ├── database.go
│       │   └── middleware.go
│       └── propagation/
│           └── w3c.go
└── pkg/
    ├── otlp/                     # OpenTelemetry protocol handling
    │   ├── models.pb.go          # Generated protobuf code
    │   └── marshal.go            # Conversion utilities
    └── storage/                  # Storage abstractions
        ├── interface.go
        ├── elasticsearch.go      # Elasticsearch implementation
        └── cassandra.go          # Cassandra implementation
```

C. OTLP HTTP Handler Implementation

Here's a complete, working implementation of the Collector's OTLP HTTP ingestion endpoint:

```go
package server

import (
    "encoding/json"
    "fmt"
    "io"
    "net/http"
    "time"

    "github.com/gorilla/mux"
    "go.uber.org/zap"

    "apm-tracing-system/internal/apm"
    "apm-tracing-system/internal/collector/ingestion"
)

// OTLPHandler handles OpenTelemetry Protocol over HTTP requests
type OTLPHandler struct {
    pipeline *ingestion.Pipeline
    logger   *zap.Logger
}

// NewOTLPHandler creates a new HTTP handler for OTLP
func NewOTLPHandler(pipeline *ingestion.Pipeline, logger *zap.Logger) *OTLPHandler {
    return &OTLPHandler{
        pipeline: pipeline,
        logger:   logger,
    }
}

// ServeHTTP implements http.Handler
func (h *OTLPHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    start := time.Now()

    // Only accept POST requests
    if r.Method != http.MethodPost {
```

```
        h.respondWithError(w, http.StatusMethodNotAllowed, "Method not allowed")
        return
    }

    // Check content type
    contentType := r.Header.Get("Content-Type")
    if contentType != "application/json" && contentType != "application/x-protobuf" {
        h.respondWithError(w, http.StatusUnsupportedMediaType,
            fmt.Sprintf("Unsupported content type: %s", contentType))
        return
    }

    // Read and parse request body
    body, err := io.ReadAll(r.Body)
    if err != nil {
        h.logger.Error("Failed to read request body", zap.Error(err))
        h.respondWithError(w, http.StatusBadRequest, "Failed to read request body")
        return
    }
    defer r.Body.Close()

    var spans []apm.Span
    if contentType == "application/json" {
        // Parse OTLP JSON format
        spans, err = h.parseOTLPJSON(body)
    } else {
        // Parse OTLP protobuf format
        spans, err = h.parseOTLPProtobuf(body)
    }

    if err != nil {
        h.logger.Error("Failed to parse request", zap.Error(err))
        h.respondWithError(w, http.StatusBadRequest,
            fmt.Sprintf("Invalid request format: %v", err))
```

```
        return
    }

    // Process spans through the ingestion pipeline
    processedCount := 0
    for _, span := range spans {
        if err := h.pipeline.ProcessSpan(r.Context(), span); err != nil {
            h.logger.Warn("Failed to process span",
                zap.String("trace_id", span.TraceID),
                zap.String("span_id", span.SpanID),
                zap.Error(err))
            // Continue processing other spans even if one fails
        } else {
            processedCount++
        }
    }

    // Send success response
    response := map[string]interface{}{
        "status": "success",
        "processed_spans": processedCount,
        "total_spans": len(spans),
        "processing_time_ms": time.Since(start).Milliseconds(),
    }

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    if err := json.NewEncoder(w).Encode(response); err != nil {
        h.logger.Error("Failed to encode response", zap.Error(err))
    }

    h.logger.Info("Processed OTLP batch",
        zap.Int("processed", processedCount),
```

```
        zap.Int("total", len(spans)),
        zap.Duration("duration", time.Since(start)))
}

// parseOTLPJSON parses OTLP JSON format into internal Span objects
func (h *OTLPHandler) parseOTLPJSON(data []byte) ([]apm.Span, error) {
    var otlpRequest OTLPJSONRequest
    if err := json.Unmarshal(data, &otlpRequest); err != nil {
        return nil, fmt.Errorf("JSON unmarshal failed: %w", err)
    }

    // Convert OTLP spans to internal representation
    var spans []apm.Span
    for _, resourceSpan := range otlpRequest.ResourceSpans {
        serviceName := extractServiceName(resourceSpan.Resource)

        for _, scopeSpan := range resourceSpan.ScopeSpans {
            for _, otlpSpan := range scopeSpan.Spans {
                span, err := convertOTLPSpan(otlpSpan, serviceName)
                if err != nil {
                    h.logger.Warn("Failed to convert OTLP span",
                        zap.Error(err),
                        zap.String("span_name", otlpSpan.Name))
                    continue
                }
                spans = append(spans, span)
            }
        }
    }

    return spans, nil
}
```

```go
// parseOTLPProtobuf parses OTLP protobuf format
func (h *OTLPHandler) parseOTLPProtobuf(data []byte) ([]apm.Span, error) {
    // Implementation would use generated protobuf code
    // For simplicity, we show a placeholder
    return nil, fmt.Errorf("protobuf parsing not implemented in this example")
}

// respondWithError sends an error response
func (h *OTLPHandler) respondWithError(w http.ResponseWriter, code int, message string) {
    response := map[string]interface{}{
        "status":  "error",
        "code":    code,
        "message": message,
    }

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(code)
    if err := json.NewEncoder(w).Encode(response); err != nil {
        h.logger.Error("Failed to encode error response", zap.Error(err))
    }
}

// OTLPJSONRequest represents the OTLP JSON structure
type OTLPJSONRequest struct {
    ResourceSpans []ResourceSpan `json:"resourceSpans"`
}

// ResourceSpan represents OTLP ResourceSpans
type ResourceSpan struct {
    Resource   Resource   `json:"resource"`
    ScopeSpans []ScopeSpan `json:"scopeSpans"`
}

// Resource represents OTLP Resource
```

```go
type Resource struct {
    Attributes []KeyValue `json:"attributes"`
}

// ScopeSpan represents OTLP ScopeSpans
type ScopeSpan struct {
    Spans []OTLPSpan `json:"spans"`
}

// OTLPSpan represents an OTLP span in JSON format
type OTLPSpan struct {
    TraceID           string      `json:"traceId"`
    SpanID            string      `json:"spanId"`
    ParentSpanID      string      `json:"parentSpanId"`
    Name              string      `json:"name"`
    Kind              string      `json:"kind"`
    StartTimeUnixNano string      `json:"startTimeUnixNano"` // string to handle large numbers
    EndTimeUnixNano   string      `json:"endTimeUnixNano"`
    Attributes        []KeyValue  `json:"attributes"`
    Status            OTLPStatus  `json:"status"`
}

// KeyValue represents OTLP KeyValue
type KeyValue struct {
    Key   string      `json:"key"`
    Value Value       `json:"value"`
}

// Value represents OTLP AnyValue
type Value struct {
    StringValue  string    `json:"stringValue,omitempty"`
    BoolValue    bool      `json:"boolValue,omitempty"`
    IntValue     int64     `json:"intValue,omitempty"`
    DoubleValue  float64   `json:"doubleValue,omitempty"`
```

```
    ArrayValue   []Value   `json:"arrayValue,omitempty"`
}

// OTLPStatus represents OTLP Status
type OTLPStatus struct {
    Code    string `json:"code"`
    Message string `json:"message,omitempty"`
}

// extractServiceName extracts service name from OTLP resource attributes
func extractServiceName(resource Resource) string {
    for _, attr := range resource.Attributes {
        if attr.Key == "service.name" {
            return attr.Value.StringValue
        }
    }
    return "unknown-service"
}

// convertOTLPSpan converts OTLP span to internal Span representation
func convertOTLPSpan(otlpSpan OTLPSpan, serviceName string) (apm.Span, error) {
    // Parse timestamps from nanosecond strings
    startTime, err := parseUnixNano(otlpSpan.StartTimeUnixNano)
    if err != nil {
        return apm.Span{}, fmt.Errorf("invalid start time: %w", err)
    }

    endTime, err := parseUnixNano(otlpSpan.EndTimeUnixNano)
    if err != nil {
        return apm.Span{}, fmt.Errorf("invalid end time: %w", err)
    }

    duration := endTime.Sub(startTime)
    if duration < 0 {
```

```
        return apm.Span{}, fmt.Errorf("negative duration: start=%v, end=%v", startTime, endTime)
    }

    // Convert attributes to map
    attributes := make(map[string]string)
    for _, attr := range otlpSpan.Attributes {
        // Convert value to string representation
        attributes[attr.Key] = valueToString(attr.Value)
    }

    // Add span.kind from OTLP kind field
    if otlpSpan.Kind != "" {
        attributes["span.kind"] = otlpSpan.Kind
    }

    // Convert status code
    statusCode := 0 // Unset
    if otlpSpan.Status.Code == "STATUS_CODE_OK" {
        statusCode = 1
    } else if otlpSpan.Status.Code == "STATUS_CODE_ERROR" {
        statusCode = 2
    }

    return apm.Span{
        SpanID:       otlpSpan.SpanID,
        TraceID:      otlpSpan.TraceID,
        ParentSpanID: otlpSpan.ParentSpanID,
        Name:         otlpSpan.Name,
        ServiceName:  serviceName,
        StartTime:    startTime,
        Duration:     duration,
        Attributes:   attributes,
```

```
        Status: apm.SpanStatus{
            Code:    statusCode,
            Message: otlpSpan.Status.Message,
        },
        ReceivedAt: time.Now(),
    }, nil
}

// parseUnixNano parses a string representing nanoseconds since Unix epoch
func parseUnixNano(nsStr string) (time.Time, error) {
    // Implementation parses string to int64 then converts to time.Time
    // Simplified for this example
    return time.Now(), nil
}

// valueToString converts OTLP AnyValue to string
func valueToString(v Value) string {
    if v.StringValue != "" {
        return v.StringValue
    }
    if v.BoolValue {
        return "true"
    }
    if v.IntValue != 0 {
        return fmt.Sprintf("%d", v.IntValue)
    }
    if v.DoubleValue != 0 {
        return fmt.Sprintf("%f", v.DoubleValue)
    }
    return ""
}

// RegisterRoutes registers the OTLP handler with the router
func RegisterRoutes(router *mux.Router, pipeline *ingestion.Pipeline, logger *zap.Logger) {
```

```
    handler := NewOTLPHandler(pipeline, logger)

    router.Handle("/v1/traces", handler).Methods("POST")

    router.Handle("/api/v2/spans", handler).Methods("POST") // Legacy Jaeger endpoint

}
```

D. Span Processing Pipeline Skeleton

Here's the skeleton for the core ingestion pipeline that processes spans through validation, buffering, and sampling:

```go
package ingestion

import (
    "context"
    "time"

    "apm-tracing-system/internal/apm"
)

// Pipeline is the main ingestion pipeline for processing spans
type Pipeline struct {
    validator      *SpanValidator
    bufferManager  *BufferManager
    headSampler    *sampling.HeadSampler
    tailSampler    *sampling.TailSampler
    storageWriter  storage.Writer
    metrics        *PipelineMetrics
}

// NewPipeline creates a new ingestion pipeline
func NewPipeline(config *Config) (*Pipeline, error) {
    // TODO 1: Initialize span validator with config validation rules
    // TODO 2: Create buffer manager with configured max size and TTL
    // TODO 3: Initialize head sampler with sampling rates from config
    // TODO 4: Initialize tail sampler with rules from config
    // TODO 5: Create storage writer for the configured storage backend
    // TODO 6: Set up metrics collection for pipeline statistics
    // TODO 7: Start background goroutines for buffer eviction and trace assembly
    return &Pipeline{}, nil
}

// ProcessSpan processes a single span through the ingestion pipeline
func (p *Pipeline) ProcessSpan(ctx context.Context, span apm.Span) error {
    // TODO 1: Validate span fields (trace ID, span ID, timestamps, etc.)
    //         Return error if span is invalid (malformed data)
```

```
    // TODO 2: Apply head-based sampling decision if not already made by SDK
    //         Use consistent hashing on trace ID for deterministic decision

    // TODO 3: If span is not sampled, only store minimal metadata for trace existence
    //         Otherwise, proceed with full processing

    // TODO 4: Add span to buffer manager for its trace ID
    //         Buffer manager will handle out-of-order arrival and trace assembly

    // TODO 5: Write span to Write-Ahead Log for durability
    //         This ensures spans aren't lost if collector crashes before processing

    // TODO 6: Update pipeline metrics (counters for spans processed, errors, etc.)

    // TODO 7: Return nil on success, error if any step fails
    return nil
}

// ProcessCompleteTrace is called when a trace is assembled and ready for tail sampling
func (p *Pipeline) ProcessCompleteTrace(ctx context.Context, trace *apm.Trace) error {
    // TODO 1: Apply tail-based sampling rules to the completed trace
    //         Evaluate error status, latency percentiles, custom rules

    // TODO 2: If tail sampling overrides head decision to keep, mark trace as sampled
    //         Otherwise, if head decision was drop, discard the trace

    // TODO 3: For sampled traces, write all spans to persistent storage
    //         Use bulk write operations for efficiency

    // TODO 4: Extract service dependencies and send to service map builder
    //         Each inter-service span creates an edge in the dependency graph
```

```
    // TODO 5: Send span metrics to analytics aggregator
    //         Latency, error rates, operation counts for anomaly detection

    // TODO 6: Update sampling statistics for adaptive rate adjustment
    return nil
}

// PipelineMetrics tracks ingestion pipeline performance
type PipelineMetrics struct {
    SpansReceived   int64
    SpansProcessed  int64
    SpansDropped    int64
    TracesAssembled int64
    ProcessingTime  time.Duration
}
```

E. Language-Specific Hints

1. **HTTP Server Best Practices:**
   - Use `http.TimeoutHandler` to enforce request timeouts
   - Implement middleware for CORS, authentication, and request logging
   - Set `ReadTimeout` and `WriteTimeout` on the HTTP server to prevent resource exhaustion
   - Use connection pooling with `http.Transport` for outgoing requests to storage
2. **JSON Processing Optimization:**
   - For high-throughput scenarios, consider `json-iterator/go` which is API-compatible with `encoding/json` but significantly faster
   - Reuse JSON encoders/decoders where possible to reduce allocations
   - Use `json.RawMessage` for deferred parsing of nested structures
3. **Time Handling:**
   - Always use `time.Time` for timestamps internally, but be careful with timezone conversions
   - For nanosecond precision timestamps, use `time.Unix(0, nanoseconds)` and `time.UnixNano()`
   - Consider using `monotonic` time for duration measurements within a process
4. **Context Propagation:**
   - Use `context.WithTimeout` for operations that should have time limits
   - Pass context as first parameter to all functions that make I/O calls
   - Store span in context using a private key type to avoid collisions: `type contextKey int; const spanKey contextKey = 0`
5. **Concurrency Patterns:**

- Use `sync.Pool` for frequently allocated objects (like `Span` structs)
- Consider buffered channels for decoupling pipeline stages with backpressure
- Use `errgroup` for managing groups of goroutines with coordinated error handling

F. Milestone Checkpoint

After implementing the data flow pipeline, verify the complete trace lifecycle:

1. **Start the Collector:**

```bash
go run cmd/collector/main.go --config config.yaml
```

Expected output: `Collector started on :4318 (HTTP) and :4317 (gRPC)`

2. **Send Test Spans:**

```bash
# Using curl to send OTLP JSON
curl -X POST http://localhost:4318/v1/traces \
  -H "Content-Type: application/json" \
  -d '{
    "resourceSpans": [{
      "resource": {"attributes": [{"key": "service.name", "value": {"stringValue": "test-service"}}]},
      "scopeSpans": [{
        "spans": [{
          "traceId": "4bf92f3577b34da6a3ce929d0e0e4736",
          "spanId": "00f067aa0ba902b7",
          "name": "test-operation",
          "startTimeUnixNano": "'$(date +%s)000000000'",
          "endTimeUnixNano": "'$(($(date +%s)+1))000000000'",
          "attributes": [],
          "status": {"code": "STATUS_CODE_OK"}
        }]
      }]
    }]
  }'
```

Expected response: `{"status":"success","processed_spans":1,"total_spans":1,"processing_time_ms":5}`

3. **Query for the Trace:**

```bash
curl "http://localhost:8080/api/traces/4bf92f3577b34da6a3ce929d0e0e4736"
```

Expected: JSON response with the trace containing your test span.

4. **Check Service Map:**

```bash
curl "http://localhost:8080/api/service-map?window=1m"
```

Expected: Service map with `test-service` node (though no edges since only one span).

5. **Verify Storage:**
   - Check that spans appear in your storage backend (Elasticsearch, Cassandra, etc.)
   - Verify indexes are created for trace ID and service name

Signs of Issues:

- **Spans not appearing in queries:** Check buffer manager eviction policy and trace assembly timeout
- **High memory usage:** Buffer manager may not be evicting old traces; check `maxSize` configuration
- **Slow response times:** Ingestion pipeline may be overloaded; implement rate limiting or increase batch sizes
- **Missing parent-child relationships:** Verify context propagation headers are being injected/extracted correctly

11. Error Handling and Edge Cases

> **Milestone(s):** This section applies to all five milestones, as error handling and edge case management are cross-cutting concerns that affect the entire APM Tracing System's reliability and resilience.

Mental Model: The Fault-Tolerant Hospital

Think of the APM system as a well-run hospital emergency room during a crisis. When everything functions normally, patients (traces) flow smoothly through triage (sampling), diagnosis (analysis), and treatment (storage/visualization). However, during a mass casualty event (traffic surge) or when critical equipment fails (storage outage), the hospital must adapt. It implements **triage protocols** (backpressure) to prioritize the most critical cases, activates **backup generators** (redundant storage), and maintains **patient records** (WAL) to ensure no critical information is lost even during power failures. This section defines how our system behaves when things go wrong, ensuring it degrades gracefully rather than collapsing catastrophically.

Failure Modes and Recovery Strategies

A robust APM system must anticipate and handle failures at multiple layers. The following table categorizes the most critical failure modes, their detection mechanisms, and recovery strategies:

Failure Mode	Symptoms/Detection	Immediate Response	Recovery Strategy	Prevention
Collector Overload (Traffic surge exceeding ingestion capacity)	• HTTP 503/429 responses from ingestion endpoints • Growing memory consumption in `BufferManager` • Increasing `PipelineMetrics.SpansDropped` counter • High CPU utilization on collector nodes	1. Return HTTP 429 (Too Many Requests) with Retry-After header 2. Activate head sampling at 100% drop rate temporarily 3. Shed load by rejecting spans from low-priority services 4. Emit high-priority alert	1. Auto-scale collector pods based on queue depth 2. Gradually restore sampling rates as load decreases 3. Replay spans from Write-Ahead Log (WAL) if available	• Implement rate limiting per client IP/service • Set memory bounds with automatic eviction • Use load balancers with health checks
Storage Backend Unavailable (Database/object store connection loss)	• Storage write timeouts exceeding `StorageConfig.Timeout` • Exponential backoff in storage writer retries • `GetTraceByID` returns "storage unavailable" errors • Health check endpoint reports degraded status	1. Buffer spans in memory up to safe limits 2. Write to local WAL for durability 3. Mark storage writer as degraded, continue processing 4. Switch to read-only mode for query endpoints	1. Implement circuit breaker pattern for storage client 2. Retry with exponential backoff when connection restored 3. Replay WAL entries to catch up on missed writes 4. Validate data consistency after recovery	• Use multiple storage replicas with failover • Implement storage health probes • Set conservative timeouts with context cancellation
Network Partition (Collector cluster split-brain)	• Leader election failures in distributed components • Inconsistent service map views across nodes • Duplicate span storage across partitions • Clock skew exceeding allowed tolerance	1. Continue operating independently in each partition 2. Mark data with partition identifier for later reconciliation 3. Disable distributed coordination features (global sampling rates) 4. Log partition events for manual intervention	1. Use CRDTs for eventually consistent service maps 2. Implement last-write-wins for duplicate spans with timestamps 3. Run reconciliation job when network heals 4. Reset adaptive sampling state post-partition	• Use consensus algorithms (Raft) for critical state • Implement anti-entropy protocols for background sync • Monitor network latency and partition probability
Corrupt Span Data (Malformed, invalid, or poisoned spans)	• `SpanValidator` returns validation errors • JSON/protobuf parsing failures with specific error codes	1. Reject individual malformed spans with 400 Bad	1. Implement strict validation with clear error messages 2. Provide validation	• Validate spans at SDK before transmission • Use schema validation for

Failure Mode	Symptoms/Detection	Immediate Response	Recovery Strategy	Prevention
	• Span timestamps far in future/past (beyond `maxTraceDuration`) • Trace ID format violations (wrong length/encoding)	Request 2. Continue processing other spans in same batch 3. Increment "spans rejected" metric with error type label 4. Optionally quarantine suspicious spans for analysis	error details in response (truncated) 3. Log full corrupt spans at DEBUG level for debugging 4. Auto-block repeat offenders after threshold	OpenTelemetry formats • Implement request size limits to prevent DoS
Clock Skew Across Services (Span ordering inconsistencies)	• Child spans appearing to start before parent spans • Negative span durations after timestamp normalization • Service map edges with impossible latency values • Trace visualization showing time travel	1. Normalize timestamps using collector's clock on ingestion 2. Calculate durations using normalized times only 3. Apply heuristic corrections (child cannot start before parent) 4. Log clock skew warnings for investigation	1. Implement NTP synchronization across all services 2. Store original and normalized timestamps separately 3. Use monotonic clocks for duration calculations where possible 4. Detect and report services with consistent clock issues	• Require SDKs to send monotonic clock readings alongside wall time • Implement timestamp bounds checking in validator • Educate users about time synchronization best practices
Memory Exhaustion (Unbounded buffer growth)	• `BufferManager` eviction running constantly • Go runtime reporting "out of memory" • High GC pressure affecting throughput • `TraceBuffer` count exceeding `maxSize` despite eviction	1. Trigger aggressive eviction of oldest/incomplete traces 2. Temporarily increase head sampling drop rate to 100% 3. Spill over to disk if disk buffer configured 4. Restart process with clean state as last resort	1. Implement multiple eviction policies (LRU, by age, by size) 2. Add memory limits with hard boundaries 3. Monitor buffer metrics with alerting thresholds 4. Adjust `maxTraceDuration` based on memory pressure	• Set conservative default buffer sizes • Implement proactive monitoring of memory trends • Use off-heap storage for large trace buffers
Late-Arriving Spans (Spans arriving after trace assembly)	• `TraceAssembler.IsTraceComplete` returning false for assembled traces • Spans with timestamps outside assembly window	1. Extend buffer TTL for traces expecting late children 2. Store late	1. Implement watermark-based trace completion detection 2. Use asynchronous trace assembly with	• Set realistic `maxTraceDuration` based on network latency • Implement span

Failure Mode	Symptoms/Detection	Immediate Response	Recovery Strategy	Prevention
	• Orphan spans without parent in storage • Incomplete traces in query results	spans separately for manual attachment 3. Mark traces as "potentially incomplete" in query results 4. Update service map with partial information	configurable wait time 3. Provide admin API to force assembly of specific traces 4. Re-process buffers periodically to attach late spans	batching with deadlines at SDK • Use synchronous tracing for critical paths where possible

> **Key Insight:** The system's error handling follows the principle of **graceful degradation** rather than all-or-nothing availability. When components fail, the system continues operating with reduced functionality (e.g., storing spans locally when storage is unavailable) rather than rejecting all incoming data.

ADR: Storage Failure Recovery Strategy

Decision: WAL + Delayed Retry with Exponential Backoff for Storage Failures

- **Context**: The storage backend (database, object store) is a critical dependency that may experience temporary outages lasting from seconds to hours. During outages, we must preserve incoming span data without overwhelming memory, and ensure eventual consistency when storage recovers.
- **Options Considered**:
  1. **Drop spans during outage**: Simple but loses critical debugging data precisely when needed most (during system failures).
  2. **In-memory buffering only**: Preserves data but risks memory exhaustion during prolonged outages.
  3. **WAL to local disk with retry**: Writes spans to local disk Write-Ahead Log during outages, retries when storage recovers.
- **Decision**: Implement a hybrid approach using **bounded in-memory buffering** (for short outages) combined with **WAL to local disk** (for prolonged outages), with automatic retry using exponential backoff when storage recovers.
- **Rationale**:
  - WAL provides durability guarantees similar to databases—spans are fsynced to disk before acknowledging receipt.
  - Local disk storage is typically larger and more reliable than memory for temporary storage.
  - Exponential backoff prevents overwhelming recovering storage with backlog.
  - Bounded memory buffer handles short blips without disk I/O overhead.
- **Consequences**:
  - **Positive**: No data loss during storage outages of arbitrary length (assuming disk space).
  - **Positive**: Self-healing—automatically catches up when storage recovers.
  - **Negative**: Requires disk space provisioning on collector nodes.
  - **Negative**: Adds complexity with WAL rotation, cleanup, and replay logic.

Option	Pros	Cons	Why Not Chosen
Drop spans during outage	Simple implementation, no resource pressure	Unacceptable data loss, violates APM's core purpose	Data loss is catastrophic for debugging production issues
In-memory buffering only	Fast, no disk I/O overhead	Bounded by RAM, data loss on process restart	Memory is precious and expensive; outages may last longer than RAM can buffer
WAL + delayed retry	Durable, handles arbitrary outage length, self-healing	Disk I/O overhead, cleanup complexity	**CHOSEN** - Provides strongest durability with manageable operational complexity

Backpressure and Degraded Modes

Mental Model: The Pressure Release Valve

Imagine a steam boiler with multiple safety valves. Under normal pressure, all valves remain closed. As pressure increases, smaller valves open first to relieve minor overpressure. In extreme conditions, the main rupture disk bursts to prevent catastrophic explosion, sacrificing the disk to save the boiler. Our APM system implements similar **cascading backpressure mechanisms**: starting with gentle load shedding (adjusting sampling rates) and escalating to aggressive measures (dropping all low-priority data) when under extreme load.

Cascading Backpressure Strategy

The system implements a multi-tiered approach to handle load, with each tier activating at progressively severe conditions:

Tier 1: Adaptive Sampling Adjustment (Load > 80% capacity)

- Automatically reduce head sampling rates for high-traffic services
- Increase tail sampling thresholds to keep only critical errors
- Adjust based on real-time throughput measurements
- **Activation**: When ingestion queue depth exceeds 80% of buffer capacity

Tier 2: Selective Load Shedding (Load > 95% capacity)

- Reject spans from pre-configured low-priority services
- Implement per-client rate limiting with 429 responses
- Temporarily disable expensive processing (detailed attribute indexing)
- **Activation**: When memory utilization exceeds 90% or CPU > 85%

Tier 3: Aggressive Degradation (Load > 100% capacity, system unstable)

- Drop all head sampling (100% drop rate)
- Process only tail sampling for critical errors
- Disable service map updates and analytics aggregation
- Return "degraded service" status from health endpoints
- **Activation**: When system is at risk of OOM kill or complete unresponsiveness

Tier 4: Fail-Safe Mode (Complete overload, preserving core function)

- Write spans directly to WAL without processing
- Disable all HTTP/gRPC endpoints except health check
- Log detailed diagnostics for post-mortem analysis
- **Activation**: Manual trigger or automatic when multiple collectors fail

Degraded Modes Operation

When the system enters a degraded mode, it maintains transparency through:

1. **Health Endpoint Status Levels**:
   - `/healthz` returns HTTP 200 with `{"status": "healthy"}`
   - `/healthz` returns HTTP 200 with `{"status": "degraded", "affected_components": ["sampling", "analytics"]}`
   - `/healthz` returns HTTP 503 with `{"status": "unhealthy", "reason": "storage_unavailable"}`
2. **Client Notification**:
   - HTTP 429 responses include `Retry-After: 30` header
   - gRPC status codes with `RESOURCE_EXHAUSTED` and retry hints
   - SDKs receive backpressure signals and apply client-side sampling
3. **Monitoring and Alerting**:
   - Each backpressure tier activation emits structured log event
   - Metrics track time spent in each degraded state
   - Alert managers notified of state transitions for operator intervention

ADR: Backpressure Implementation Pattern

Decision: Client-Responsive Backpressure with Progressive Degradation

- **Context**: Under extreme load, the system must protect itself while providing the best possible service. Simply rejecting all requests leads to "goodput collapse" where useful work drops to zero. We need a strategy that maximizes valuable output (error traces, high-latency traces) while shedding less important load.
- **Options Considered**:
  1. **Random drop**: Simple random selection of requests to reject.
  2. **Queue-based backpressure**: Use bounded queues that block when full, pushing back on clients.
  3. **Adaptive priority-based shedding**: Intelligently select which traces to process based on their likely value.
- **Decision**: Implement **adaptive priority-based shedding** using a combination of:
  - Client-side sampling adjustments (via response headers)
  - Server-side priority queues
  - Value-based trace selection (errors, high latency first)
- **Rationale**:
  - Random drop wastes capacity on unimportant traces while potentially dropping critical ones.
  - Queue-based backpressure can cause cascading failures upstream when clients block.
  - Priority-based shedding maximizes the value of processed traces during overload, acting like a triage system that prioritizes the most medically critical patients during an emergency.
- **Consequences**:
  - **Positive**: Maximizes retention of operationally valuable traces during overload.
  - **Positive**: Clients automatically adjust their sampling rates based on server feedback.
  - **Negative**: More complex to implement than simple random drop.
  - **Negative**: Requires coordination between client SDKs and collector.

Option	Pros	Cons	Why Not Chosen
Random drop	Trivial to implement, predictable load reduction	Wastes capacity on low-value traces, drops critical data	Poor utilization of limited resources during crisis
Queue-based backpressure	Simple, preserves order, no data loss	Can cause upstream cascading failures, clients may timeout	Backpressure should propagate intelligently, not just block
Adaptive priority-based shedding	Maximizes value retention, adapts to load	Complex implementation, requires client cooperation	**CHOSEN** - Provides optimal trade-off between self-protection and service value

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Circuit Breaker	Manual state tracking with counters	`github.com/sony/gobreaker` for production-ready circuit breaker
Rate Limiting	Token bucket using `time.Ticker`	`golang.org/x/time/rate` for sophisticated limiting
Health Checks	Simple HTTP handler with status flags	`github.com/heptiolabs/healthcheck` with Kubernetes probes
WAL Implementation	Custom file rotation with fsync	`github.com/tidwall/wal` for production-grade WAL
Backpressure Signals	HTTP 429 with Retry-After header	gRPC flow control with `RESOURCE_EXHAUSTED` status

Recommended File/Module Structure

```
project-root/
  cmd/collector/main.go                    # Entry point with graceful shutdown
  internal/
    apm/
      collector/
        health/                            # Health check subsystem
          health.go                        # Health status tracking
          handler.go                       # HTTP health endpoint
        backpressure/                      # Backpressure management
          manager.go                       # Backpressure tier management
          signals.go                       # Client backpressure signaling
        degradation/                       # Degraded mode handling
          modes.go                         # Degraded mode state machine
          monitor.go                       # Mode transition monitoring
      resilience/
        circuitbreaker/                    # Circuit breaker implementation
          breaker.go                       # Storage client circuit breaker
          metrics.go                       # Breaker state metrics
        ratelimiter/                       # Rate limiting
          limiter.go                       # Token bucket rate limiter
          middleware.go                    # HTTP/gRPC rate limit middleware
        wal/                               # Write-Ahead Log
          writer.go                        # WAL writer with rotation
          replayer.go                      # WAL replay on recovery
          cleaner.go                       # WAL file cleanup
      errors/                              # Error types and handling
        errors.go                          # Custom error types
        recovery.go                        # Error recovery utilities
  pkg/
    api/
      middlewares/                         # HTTP/gRPC middlewares
        recovery.go                        # Panic recovery middleware
        backpressure.go                    # Backpressure injection middleware
```

Infrastructure Starter Code

Complete WAL Implementation for Storage Failures:

```go
// internal/apm/resilience/wal/writer.go

package wal


import (

    "encoding/binary"

    "fmt"

    "os"

    "path/filepath"

    "sync"

    "time"

)


// WALWriter provides durable write-ahead logging for spans during storage outages

type WALWriter struct {

    mu           sync.RWMutex

    file         *os.File

    filePath     string

    offset       int64

    maxFileSize  int64

    currentSize  int64

    rotateSignal chan struct{}

    done         chan struct{}

}


// NewWALWriter creates a new WAL writer with automatic rotation

func NewWALWriter(dataDir string, maxFileSize int64) (*WALWriter, error) {

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

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

    }



    filePath := filepath.Join(dataDir, fmt.Sprintf("wal-%d.bin", time.Now().UnixNano()))

    file, err := os.OpenFile(filePath, os.O_CREATE|os.O_RDWR|os.O_APPEND, 0644)

    if err != nil {

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

```
    }

    stat, err := file.Stat()
    if err != nil {
        return nil, fmt.Errorf("failed to stat WAL file: %w", err)
    }

    w := &WALWriter{
        file:         file,
        filePath:     filePath,
        offset:       0,
        maxFileSize:  maxFileSize,
        currentSize:  stat.Size(),
        rotateSignal: make(chan struct{}, 1),
        done:         make(chan struct{}),
    }

    // Start background rotation monitor
    go w.rotationMonitor()

    return w, nil
}

// Append writes a record to WAL with length-prefixed format
func (w *WALWriter) Append(data []byte) (int64, error) {
    w.mu.Lock()
    defer w.mu.Unlock()

    // Check if we need to rotate
    if w.currentSize+int64(len(data))+8 > w.maxFileSize {
        if err := w.rotate(); err != nil {
            return 0, fmt.Errorf("failed to rotate WAL: %w", err)
        }
```

```
    }

    // Write length prefix (8 bytes)
    length := uint64(len(data))
    if err := binary.Write(w.file, binary.LittleEndian, length); err != nil {
        return 0, fmt.Errorf("failed to write length prefix: %w", err)
    }

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

    // Sync for durability
    if err := w.file.Sync(); err != nil {
        return 0, fmt.Errorf("failed to sync WAL: %w", err)
    }

    currentOffset := w.offset
    w.offset += int64(n + 8) // Include length prefix
    w.currentSize += int64(n + 8)

    return currentOffset, nil
}

// rotate creates a new WAL file and closes the old one
func (w *WALWriter) rotate() error {
    oldFile := w.file

    // Create new file
    newFilePath := filepath.Join(filepath.Dir(w.filePath),
        fmt.Sprintf("wal-%d.bin", time.Now().UnixNano()))
```

```
    newFile, err := os.OpenFile(newFilePath, os.O_CREATE|os.O_RDWR|os.O_APPEND, 0644)
    if err != nil {
        return fmt.Errorf("failed to create new WAL file: %w", err)
    }

    // Update state
    w.file = newFile
    w.filePath = newFilePath
    w.offset = 0
    w.currentSize = 0

    // Close old file
    if err := oldFile.Close(); err != nil {
        return fmt.Errorf("failed to close old WAL file: %w", err)
    }

    // Signal cleanup goroutine
    select {
    case w.rotateSignal <- struct{}{}:
    default:
    }

    return nil
}

// Close gracefully closes the WAL writer
func (w *WALWriter) Close() error {
    close(w.done)
    w.mu.Lock()
    defer w.mu.Unlock()
    return w.file.Close()
}
```

```go
// rotationMonitor handles cleanup of old WAL files
func (w *WALWriter) rotationMonitor() {
    ticker := time.NewTicker(5 * time.Minute)
    defer ticker.Stop()

    for {
        select {
        case <-w.rotateSignal:
            w.cleanupOldFiles()
        case <-ticker.C:
            w.cleanupOldFiles()
        case <-w.done:
            return
        }
    }
}

// cleanupOldFiles removes WAL files older than retention period
func (w *WALWriter) cleanupOldFiles() {
    // Implementation for cleaning up old WAL files
    // Keep last 24 hours of WAL files for recovery
}

// WALReader reads records from WAL file for replay
type WALReader struct {
    file *os.File
}

// NewWALReader creates a reader for replaying WAL entries
func NewWALReader(filePath string) (*WALReader, error) {
    file, err := os.Open(filePath)
    if err != nil {
        return nil, fmt.Errorf("failed to open WAL file: %w", err)
    }
```

```
    return &WALReader{file: file}, nil
}

// ReadNext reads the next record from WAL
func (r *WALReader) ReadNext() ([]byte, error) {
    // Read length prefix
    var length uint64
    if err := binary.Read(r.file, binary.LittleEndian, &length); err != nil {
        return nil, err // EOF or error
    }

    // Read data
    data := make([]byte, length)
    n, err := r.file.Read(data)
    if err != nil {
        return nil, fmt.Errorf("failed to read WAL data: %w", err)
    }
    if uint64(n) != length {
        return nil, fmt.Errorf("short read: expected %d, got %d", length, n)
    }

    return data, nil
}
```

Circuit Breaker for Storage Client:

```go
// internal/apm/resilience/circuitbreaker/breaker.go

package circuitbreaker

import (
    "context"
    "errors"
    "sync"
    "sync/atomic"
    "time"
)

var (
    ErrCircuitOpen    = errors.New("circuit breaker is open")
    ErrTooManyRequests = errors.New("too many requests in half-open state")
)

// State represents circuit breaker state
type State int32

const (
    StateClosed State = iota
    StateOpen
    StateHalfOpen
)

// CircuitBreaker implements the circuit breaker pattern
type CircuitBreaker struct {
    mu               sync.RWMutex
    state            State
    failureThreshold int
    successThreshold int
    failureCount     int
    successCount     int
    timeout          time.Duration
    lastFailure      time.Time
```

```
    halfOpenMax       int
    halfOpenCount     int32
    onStateChange     func(from, to State)
}

// Config holds circuit breaker configuration
type Config struct {
    FailureThreshold int           // Failures before opening circuit
    SuccessThreshold int           // Successes before closing circuit
    Timeout          time.Duration // How long to stay open before half-open
    HalfOpenMax      int           // Max requests allowed in half-open state
}

// NewCircuitBreaker creates a new circuit breaker
func NewCircuitBreaker(config Config) *CircuitBreaker {
    return &CircuitBreaker{
        state:            StateClosed,
        failureThreshold: config.FailureThreshold,
        successThreshold: config.SuccessThreshold,
        timeout:          config.Timeout,
        halfOpenMax:      config.HalfOpenMax,
    }
}

// Execute runs the operation with circuit breaker protection
func (cb *CircuitBreaker) Execute(ctx context.Context, operation func() error) error {
    // Check if we should allow the request
    if !cb.allowRequest() {
        return ErrCircuitOpen
    }

    // Execute the operation
    err := operation()
```

```
    // Record the result
    cb.recordResult(err)

    return err
}

// allowRequest determines if a request should be allowed
func (cb *CircuitBreaker) allowRequest() bool {
    cb.mu.RLock()
    defer cb.mu.RUnlock()

    switch cb.state {
    case StateClosed:
        return true
    case StateOpen:
        // Check if timeout has elapsed
        if time.Since(cb.lastFailure) > cb.timeout {
            cb.mu.RUnlock()
            cb.mu.Lock()
            // Double-check after acquiring write lock
            if cb.state == StateOpen && time.Since(cb.lastFailure) > cb.timeout {
                cb.state = StateHalfOpen
                cb.halfOpenCount = 0
                if cb.onStateChange != nil {
                    cb.onStateChange(StateOpen, StateHalfOpen)
                }
            }
            cb.mu.Unlock()
            cb.mu.RLock()
            // Now recheck state
        }
        return false
    case StateHalfOpen:
```

```
        // Allow only limited number of requests in half-open state
        if atomic.LoadInt32(&cb.halfOpenCount) >= int32(cb.halfOpenMax) {
            return false
        }
        atomic.AddInt32(&cb.halfOpenCount, 1)
        return true
    default:
        return false
    }
}

// recordResult updates circuit breaker state based on operation result
func (cb *CircuitBreaker) recordResult(err error) {
    cb.mu.Lock()
    defer cb.mu.Unlock()

    switch cb.state {
    case StateClosed:
        if err != nil {
            cb.failureCount++
            if cb.failureCount >= cb.failureThreshold {
                cb.state = StateOpen
                cb.lastFailure = time.Now()
                if cb.onStateChange != nil {
                    cb.onStateChange(StateClosed, StateOpen)
                }
            }
        } else {
            cb.failureCount = 0 // Reset on success
        }
    case StateHalfOpen:
        if err != nil {
            // Failure in half-open state -> back to open
```

```
            cb.state = StateOpen
            cb.lastFailure = time.Now()
            cb.failureCount = 1
            if cb.onStateChange != nil {
                cb.onStateChange(StateHalfOpen, StateOpen)
            }
        } else {
            cb.successCount++
            if cb.successCount >= cb.successThreshold {
                // Enough successes -> close the circuit
                cb.state = StateClosed
                cb.failureCount = 0
                cb.successCount = 0
                if cb.onStateChange != nil {
                    cb.onStateChange(StateHalfOpen, StateClosed)
                }
            }
        }
    }
}

// State returns current circuit breaker state
func (cb *CircuitBreaker) State() State {
    cb.mu.RLock()
    defer cb.mu.RUnlock()
    return cb.state
}

// SetStateChangeCallback sets a callback for state changes
func (cb *CircuitBreaker) SetStateChangeCallback(callback func(from, to State)) {
    cb.mu.Lock()
    defer cb.mu.Unlock()
    cb.onStateChange = callback
```

```
}
```

Core Logic Skeleton Code

Backpressure Manager Implementation:

```go
// internal/apm/collector/backpressure/manager.go

package backpressure

import (
    "context"
    "sync"
    "time"
)

// Tier represents a backpressure tier with activation thresholds
type Tier int

const (
    TierNormal Tier = iota
    TierAdaptiveSampling
    TierSelectiveShedding
    TierAggressiveDegradation
    TierFailSafe
)

// Thresholds define when to transition between tiers
type Thresholds struct {
    MemoryPercent      float64
    CPULoadPercent     float64
    QueueDepthPercent  float64
    StorageLatencyMS   int64
}

// BackpressureManager manages system backpressure state
type BackpressureManager struct {
    mu           sync.RWMutex
    currentTier  Tier
    thresholds   map[Tier]Thresholds
    metrics      *SystemMetrics
    stateChan    chan<- StateChange
```

```
    lastCheck    time.Time
    checkInterval time.Duration
}

// SystemMetrics contains system metrics for tier decisions
type SystemMetrics struct {
    MemoryUsedPercent  float64
    CPULoadPercent     float64
    QueueDepthPercent  float64
    StorageLatencyMS   int64
    SpansDroppedPerSec int64
    ActiveTraces       int64
}

// StateChange represents a backpressure state transition
type StateChange struct {
    Timestamp time.Time
    FromTier  Tier
    ToTier    Tier
    Reason    string
    Metrics   SystemMetrics
}

// NewBackpressureManager creates a new backpressure manager
func NewBackpressureManager(initialThresholds map[Tier]Thresholds,
                            checkInterval time.Duration) *BackpressureManager {
    return &BackpressureManager{
        currentTier:  TierNormal,
        thresholds:   initialThresholds,
        checkInterval: checkInterval,
    }
}

// Start begins monitoring and tier adjustment
```

```go
func (bm *BackpressureManager) Start(ctx context.Context) error {
    ticker := time.NewTicker(bm.checkInterval)
    defer ticker.Stop()

    for {
        select {
        case <-ticker.C:
            bm.evaluateAndAdjust(ctx)
        case <-ctx.Done():
            return ctx.Err()
        }
    }
}

// evaluateAndAdjust evaluates current metrics and adjusts tier if needed
func (bm *BackpressureManager) evaluateAndAdjust(ctx context.Context) {
    // TODO 1: Gather current system metrics
    //   - Memory usage from runtime.MemStats
    //   - CPU load from gopsutil or runtime metrics
    //   - Queue depth from BufferManager
    //   - Storage latency from storage client metrics

    // TODO 2: Determine target tier based on thresholds
    //   - Compare each metric against thresholds for each tier
    //   - Select highest (most severe) tier where any threshold is exceeded
    //   - Apply hysteresis: require sustained violation before tier increase

    // TODO 3: If tier change is needed:
    //   - Log state transition with metrics snapshot
    //   - Update currentTier with write lock
    //   - Send StateChange to stateChan if configured
    //   - Apply tier-specific actions (call applyTierActions)
```

```
    // TODO 4: If staying in same tier, check if we can downgrade
    //   - Check if metrics have been below downgrade thresholds for cooldown period
    //   - Gradually step down through tiers (don't jump from Tier 4 to Tier 0 directly)

    // TODO 5: Update lastCheck timestamp
}

// applyTierActions applies actions specific to the new tier
func (bm *BackpressureManager) applyTierActions(ctx context.Context, newTier Tier) {
    bm.mu.Lock()
    defer bm.mu.Unlock()

    // TODO 1: For each component that needs adjustment, apply tier-specific changes:
    //   - HeadSampler: adjust sampling rates (Tier 1-4)
    //   - TailSampler: adjust evaluation thresholds (Tier 2-4)
    //   - BufferManager: adjust eviction policy (Tier 2-4)
    //   - Pipeline: disable expensive processing (Tier 3-4)

    // TODO 2: Update health endpoint status
    //   - Set degraded status with affected components

    // TODO 3: Emit metrics for backpressure state
    //   - Gauge for current tier
    //   - Counter for tier transitions

    // TODO 4: Log transition at appropriate level
    //   - INFO for Tier 0-2 transitions
    //   - WARN for Tier 3 transitions
    //   - ERROR for Tier 4 transitions
}

// CurrentTier returns the current backpressure tier
func (bm *BackpressureManager) CurrentTier() Tier {
```

```
    bm.mu.RLock()
    defer bm.mu.RUnlock()
    return bm.currentTier
}

// SetStateChangeChannel sets the channel for state change notifications
func (bm *BackpressureManager) SetStateChangeChannel(ch chan<- StateChange) {
    bm.mu.Lock()
    defer bm.mu.Unlock()
    bm.stateChan = ch
}

// GetRecommendedSamplingRate returns sampling rate recommendation for current tier
func (bm *BackpressureManager) GetRecommendedSamplingRate(service string) float64 {
    // TODO: Return appropriate sampling rate based on:
    //   - Current tier
    //   - Service priority (from configuration)
    //   - Historical error rate for this service
    //   - System-wide load metrics
    return 1.0 // Default: keep all traces
}

// ShouldProcessSpan determines if a span should be processed in current tier
func (bm *BackpressureManager) ShouldProcessSpan(span models.Span) bool {
    tier := bm.CurrentTier()

    // TODO 1: For higher tiers, implement filtering logic:
    //   - Tier 2: Check if service is in priority whitelist
    //   - Tier 3: Check if span has error status or high latency
    //   - Tier 4: Only process spans marked as critical

    // TODO 2: Apply sampling based on tier-specific rates
    //   - Use consistent hashing for deterministic decisions
```

```
    // TODO 3: Return true if span should be processed, false to drop
    return true
}
```

Health Check Endpoint with Degraded States:

```go
// internal/apm/collector/health/handler.go

package health

import (
    "encoding/json"
    "net/http"
    "sync"
    "time"
)

// HealthStatus represents system health status
type HealthStatus string

const (
    StatusHealthy   HealthStatus = "healthy"
    StatusDegraded  HealthStatus = "degraded"
    StatusUnhealthy HealthStatus = "unhealthy"
)

// ComponentStatus represents status of an individual component
type ComponentStatus struct {
    Name    string       `json:"name"`
    Status  HealthStatus `json:"status"`
    Details string       `json:"details,omitempty"`
    Since   time.Time    `json:"since"`
}

// HealthResponse is the JSON response from health endpoint
type HealthResponse struct {
    Status      HealthStatus      `json:"status"`
    Timestamp   time.Time         `json:"timestamp"`
    Components  []ComponentStatus `json:"components,omitempty"`
    Message     string            `json:"message,omitempty"`
    DegradedSince *time.Time      `json:"degraded_since,omitempty"`
}
```

```go
// HealthRegistry manages component health status
type HealthRegistry struct {
    mu         sync.RWMutex
    components map[string]ComponentStatus
    overrides  map[string]func() ComponentStatus
    startTime  time.Time
}

// NewHealthRegistry creates a new health registry
func NewHealthRegistry() *HealthRegistry {
    return &HealthRegistry{
        components: make(map[string]ComponentStatus),
        overrides:  make(map[string]func() ComponentStatus),
        startTime:  time.Now(),
    }
}

// RegisterComponent adds a component to health checks
func (r *HealthRegistry) RegisterComponent(name string,
                                        checkFunc func() ComponentStatus) {
    r.mu.Lock()
    defer r.mu.Unlock()
    r.overrides[name] = checkFunc
}

// SetComponentStatus manually sets a component's status
func (r *HealthRegistry) SetComponentStatus(name string, status HealthStatus,
                                         details string) {
    r.mu.Lock()
    defer r.mu.Unlock()
    r.components[name] = ComponentStatus{
        Name:    name,
        Status:  status,
        Details: details,
```

```
        Since:   time.Now(),
    }
}

// HealthHandler implements the HTTP health check endpoint
func (r *HealthRegistry) HealthHandler(w http.ResponseWriter, req *http.Request) {
    status := r.calculateOverallStatus()
    response := r.buildHealthResponse(status)

    // Set appropriate HTTP status code
    var httpStatus int
    switch status {
    case StatusHealthy:
        httpStatus = http.StatusOK
    case StatusDegraded:
        httpStatus = http.StatusOK // 200 for degraded, body indicates issue
    case StatusUnhealthy:
        httpStatus = http.StatusServiceUnavailable
    }

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(httpStatus)
    json.NewEncoder(w).Encode(response)
}

// calculateOverallStatus determines overall system health
func (r *HealthRegistry) calculateOverallStatus() HealthStatus {
    r.mu.RLock()
    defer r.mu.RUnlock()

    // TODO 1: Check all registered components
    //   - Execute override functions for dynamic checks
    //   - Merge with manually set component statuses
```

```
    // TODO 2: Apply health aggregation logic:
    //   - If ANY component is "unhealthy", overall is "unhealthy"
    //   - If ANY component is "degraded" but none are "unhealthy", overall is "degraded"
    //   - Otherwise, "healthy"

    // TODO 3: Apply minimum uptime requirement
    //   - If system started less than 30 seconds ago, return "degraded"

    return StatusHealthy
}

// buildHealthResponse constructs the health response JSON
func (r *HealthRegistry) buildHealthResponse(overallStatus HealthStatus) HealthResponse {
    // TODO: Build complete health response with:
    //   - Overall status
    //   - Timestamp
    //   - All component statuses
    //   - Uptime
    //   - Version information
    //   - Any degradation messages

    return HealthResponse{
        Status:    overallStatus,
        Timestamp: time.Now(),
        Message:   "System operational",
    }
}

// LivenessHandler returns simple liveness check (for Kubernetes)
func (r *HealthRegistry) LivenessHandler(w http.ResponseWriter, req *http.Request) {
    // TODO: Simple check - is the process running?
    // Always return 200 unless process is terminally broken
```

```
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("ok"))
}

// ReadinessHandler returns readiness check (for Kubernetes)
func (r *HealthRegistry) ReadinessHandler(w http.ResponseWriter, req *http.Request) {
    // TODO: Check if system is ready to receive traffic
    // Return 200 if ready, 503 if not
    status := r.calculateOverallStatus()
    if status == StatusUnhealthy {
        w.WriteHeader(http.StatusServiceUnavailable)
        w.Write([]byte("not ready"))
    } else {
        w.WriteHeader(http.StatusOK)
        w.Write([]byte("ready"))
    }
}
```

Language-Specific Hints

Go-Specific Error Handling Patterns:

1. **Context Cancellation for Timeouts**:

```GO
   // Always pass context through call chains
   func (c *Collector) StoreSpan(ctx context.Context, span models.Span) error {
       // Use context for timeouts and cancellation
       ctx, cancel := context.WithTimeout(ctx, c.config.Storage.Timeout)
       defer cancel()

       return c.storage.WriteSpan(ctx, span)
   }
```

2. **Error Wrapping with Context**:

```go
// Use fmt.Errorf with %w for wrapping errors
func processBatch(batch []byte) error {
    spans, err := parseSpans(batch)
    if err != nil {
        return fmt.Errorf("failed to parse batch: %w", err)
    }

    for _, span := range spans {
        if err := validateSpan(span); err != nil {
            // Add span context to error
            return fmt.Errorf("invalid span %s: %w", span.SpanID, err)
        }
    }
    return nil
}
```

3. **Structured Error Types**:

```go
// Define error types for different failure modes
type StorageError struct {
    Op      string
    Err     error
    RetryAfter time.Duration
}

func (e *StorageError) Error() string {
    return fmt.Sprintf("storage error during %s: %v", e.Op, e.Err)
}

func (e *StorageError) Unwrap() error {
    return e.Err
}

// Check error types for recovery decisions
if errors.Is(err, &StorageError{}) {
    var storageErr *StorageError
    if errors.As(err, &storageErr) {
        time.Sleep(storageErr.RetryAfter)
        // Retry the operation
    }
}
```

4. **Graceful Shutdown with Context**:

GO

```go
// Implement graceful shutdown in main
func main() {
    collector, err := collector.New(config)
    if err != nil { /* handle */ }

    // Start collector in background
    ctx, cancel := context.WithCancel(context.Background())
    go func() {
        if err := collector.Start(ctx); err != nil {
            log.Fatalf("Collector failed: %v", err)
        }
    }()

    // Handle OS signals
    sigChan := make(chan os.Signal, 1)
    signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)

    <-sigChan // Wait for signal

    // Start graceful shutdown
    log.Println("Shutting down gracefully...")
    cancel()

    // Give components time to clean up
    shutdownCtx, shutdownCancel := context.WithTimeout(
        context.Background(), 30*time.Second)
    defer shutdownCancel()

    // Wait for shutdown or timeout
    <-shutdownCtx.Done()
    log.Println("Shutdown complete")
}
```

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Spans being dropped under load	Backpressure tier activation	1. Check `/healthz` endpoint for degraded status 2. Examine backpressure manager logs for tier transitions 3. Monitor `apm_backpressure_current_tier` metric	1. Increase collector resources 2. Adjust backpressure thresholds 3. Optimize span processing pipeline
High memory usage with frequent GC	Buffer growth or memory leak	1. Use `pprof` to capture heap profile 2. Check `BufferManager` trace count 3. Look for unbounded maps or slices	1. Reduce `maxTraceDuration` 2. Implement more aggressive eviction 3. Fix memory leaks in span processing
Storage timeouts causing cascading failures	Storage overload or network issues	1. Check storage health metrics 2. Examine circuit breaker state 3. Look for WAL growth indicating write failures	1. Increase storage capacity 2. Adjust circuit breaker thresholds 3. Implement storage connection pooling
Clock skew warnings in logs	NTP sync issues or VM clock drift	1. Compare timestamps from different services in same trace 2. Check system clock sync status 3. Look for negative durations	1. Implement NTP on all hosts 2. Use collector clock normalization 3. Warn users about misconfigured hosts
Service map showing impossible latencies	Async spans or incorrect parent-child links	1. Examine span parent relationships 2. Check for async operation markers 3. Look for clock correction artifacts	1. Improve async span detection 2. Implement span relationship validation 3. Apply timestamp normalization corrections
WAL files accumulating without cleanup	Storage recovery failures or configuration issues	1. Check WAL replayer logs for errors 2. Verify storage connectivity 3. Check disk space on collector nodes	1. Fix storage connection issues 2. Implement WAL cleanup job 3. Monitor disk usage with alerts

Milestone Checkpoint

After implementing error handling and backpressure mechanisms, verify resilience with these tests:

1. **Storage Failure Simulation**:

```bash
# Stop storage backend
docker-compose stop apm-storage

# Send spans - should see WAL writes but no errors
go run test/loadgen/main.go --spans-per-sec=100 --duration=30s

# Check WAL directory for new files
ls -la /var/lib/apm/wal/

# Restart storage
docker-compose start apm-storage

# Verify spans are replayed from WAL
tail -f /var/log/apm/collector.log | grep "WAL replay"

# Query for test traces - should appear within 60 seconds
curl "http://localhost:8080/api/traces?service=test-service"
```

2. **Backpressure Activation Test**:

```bash
# Generate overload traffic
go run test/loadgen/main.go --spans-per-sec=5000 --duration=60s

# Monitor health endpoint for tier transitions
watch -n1 'curl -s http://localhost:8080/healthz | jq .'

# Check metrics for backpressure state
curl -s http://localhost:8080/metrics | grep backpressure

# Verify spans are being sampled more aggressively
curl -s http://localhost:8080/metrics | grep spans_dropped
```

3. **Error Recovery Verification**:

```bash
# Send malformed spans
curl -X POST http://localhost:8080/v1/traces \
  -H "Content-Type: application/json" \
  -d '{"invalid": "data"}'

# Should receive 400 with error details
# Collector should continue processing other requests

# Send valid spans after error
go run test/loadgen/main.go --spans-per-sec=10 --duration=5s

# Verify normal processing resumes
```

Expected signs of correct implementation:

- Collector remains responsive during storage outages
- Memory usage stays within configured bounds under load
- Health endpoint accurately reflects system state
- WAL files are created during outages and cleaned up after recovery
- No panics or deadlocks under stress conditions

Signs of incorrect implementation:

- Collector crashes or becomes unresponsive under load
- Memory grows unbounded until OOM kill
- Spans are lost permanently during storage outages
- Health endpoint shows "healthy" when components are failing
- WAL files accumulate indefinitely without cleanup

> **Milestone(s):** This section applies to all five milestones, providing a comprehensive testing strategy that verifies each component and the system as a whole.

12. Testing Strategy

The APM Tracing System is a complex distributed application that must be reliable under heavy load, process data correctly even when it arrives out of order, and degrade gracefully when stressed. A robust testing strategy is not a luxury but a necessity. We must verify not only that the system works under ideal conditions, but that it handles the messy reality of production—network failures, clock skew, malformed data, and traffic spikes—without data loss or cascading failures.

This strategy is built on a foundation of **property-based testing** for core data invariants, **chaos testing** for resilience, and **concrete milestone checkpoints** that provide clear, binary pass/fail criteria for each development stage.

Testing Approaches: Unit, Integration, and Load

Testing in the APM system occurs at three complementary levels, each with distinct goals and techniques.

Mental Model: The Building Inspection, Fire Drill, and Stress Test Think of testing our system like ensuring a new skyscraper is safe. **Unit tests** are like inspecting individual steel beams and concrete pillars—verifying each component in isolation meets its specification. **Integration tests** are like a fire drill—testing how people (components) move through stairwells and exits (APIs and data flows) in a coordinated way under simulated emergency (failure) conditions. **Load tests** are the structural stress test—applying massive weight (traffic) to see if the building sways within acceptable limits or collapses, revealing bottlenecks and breaking points.

Unit Testing: Verifying Component Invariants

Unit tests focus on the smallest testable units of code—typically individual functions or methods—in complete isolation. Mocks and stubs are used heavily to control the environment and dependencies.

Key Areas for Unit Testing:

1. **Data Structure Validation:** Ensure that `Span`, `Trace`, and other core types enforce their invariants (e.g., a `Span`'s `EndTime` must be after its `StartTime`).
2. **Algorithm Correctness:** Test the step-by-step logic of critical algorithms with known inputs and expected outputs.
   - **Trace Assembly:** Given a specific, unordered set of `Span` objects, verify that `TraceAssembler.AssembleTrace` produces the correct hierarchical tree structure.
   - **Sampling Decisions:** For a given `traceID` and sampling rate, the `ConsistentSampler.ShouldSample` method must return a deterministic, predictable result.
   - **Percentile Calculation:** Feed a known sequence of latency values into `TDigestMetric` and verify that `Quantile(0.95)` returns a value within an acceptable error margin.
3. **State Machine Transitions:** Test that the `CircuitBreaker` and `BackpressureManager` transition between states (`StateClosed`, `StateOpen`, `TierNormal`, `TierAdaptiveSampling`) exactly when the defined thresholds are crossed.
4. **Error Handling:** Verify that functions return the expected errors for invalid inputs (e.g., `StoreSpan` with an empty `TraceID`) or when dependencies fail (e.g., a mocked storage layer returns `io.EOF`).

Property-Based Testing for Core Logic: For algorithms where the output space is large or complex, we employ property-based testing (using a library like `github.com/leanovate/gopter` for Go). Instead of writing individual example-based tests, we define *properties* that must *always* hold true for any valid input, and the framework generates hundreds of random test cases.

Component	Property to Test	Description
`TraceAssembler`	**Trace Completeness:**	For any generated set of spans belonging to the same trace, if `IsTraceComplete` returns `true`, then the assembled trace must contain exactly those spans, and the trace's `StartTime` must be the earliest span start, and `EndTime` the latest span end.
`ConsistentSampler`	**Determinism & Rate Adherence:**	For any `traceID` and sampling `rate`, the `ShouldSample` decision must be identical for all calls. Over a large number of random `traceID`s, the ratio of `true` decisions must converge to `rate` within a small statistical error.
`TDigestMetric`	**Percentile Monotonicity:**	For any set of added values, the computed quantiles must be monotonically non-decreasing: `Quantile(0.50)` ≤ `Quantile(0.95)` ≤ `Quantile(0.99)`.
`BufferManager`	**Capacity Bound:**	No matter the order or timing of `AddSpan` calls, the total number of spans stored in the `BufferManager` must never exceed its configured `maxSize`.

ADR: Mocking Strategy for External Dependencies

Decision: Use Interface-Based Mocking with Generated Test Doubles

- **Context:** Our components depend on external systems (storage, network) and other internal modules. We need to isolate units for testing without bringing up entire systems.
- **Options Considered:**
  1. **Hand-written mocks:** Create concrete test types that implement our defined interfaces (e.g., `storage.Writer`).
  2. **Mock generation tools:** Use a tool like `mockgen` (for Go) to automatically generate mock implementations from interfaces.
  3. **Integration-heavy unit tests:** Use lightweight, real implementations (e.g., an in-memory map for storage).
- **Decision:** Use **interface-based design** combined with **generated mocks** for complex dependencies, and **lightweight real implementations** (fakes) for simpler ones.
- **Rationale:** Generated mocks reduce boilerplate and are consistent. They allow us to easily set expectations (e.g., "`StoreSpan` should be called exactly once with these arguments") and simulate failures. For simpler dependencies like an in-memory store, a fake is easier to reason about and can be used in integration tests as well.
- **Consequences:** Developers must define clear interfaces. The build process requires a mock generation step. Tests become tightly coupled to mock expectations, which can make them brittle if interactions change frequently.

Option	Pros	Cons	Chosen?
Hand-written mocks	Full control, no extra tooling.	High boilerplate, tedious to maintain for large interfaces.	For simple, stable interfaces.
Generated mocks (mockgen)	Low maintenance, consistent patterns, powerful expectation API.	Adds build step, can produce verbose tests, learning curve.	**Yes**, for core external dependencies (Storage, WAL).
Lightweight real implementations (Fakes)	Realistic behavior, can be used in integration tests.	May hide bugs if the fake doesn't perfectly emulate the real component.	**Yes**, for internal abstractions like `EdgeStorage`.

Integration Testing: Verifying Component Interaction

Integration tests verify that multiple components work together correctly. They use real implementations for some components and mocked or faked versions for others (like external services).

Key Integration Test Scenarios:

1. **End-to-End Ingestion Pipeline:** Send a batch of OTLP-formatted spans via HTTP to the `OTLPHandler`. Verify that they flow through the `Pipeline` (validation, buffering, sampling) and are eventually persisted by the `storage.Writer` (using an in-memory or test-database implementation).
2. **Service Map Construction:** Feed a set of spans representing calls between services `A->B` and `B->C` into the `GraphBuilder`. Verify that `BuildCurrentGraph` produces a `ServiceGraph` with three `ServiceNode`s and two `ServiceEdge`s with correct aggregated metrics.
3. **Cross-Component Error Propagation:** Simulate a failure in the storage layer and verify that the `CircuitBreaker` on the `Pipeline` trips open, causing subsequent span processing attempts to fail fast with `ErrCircuitOpen`, and that the system's health endpoint (`HealthHandler`) reflects a `StatusDegraded` state.
4. **Context Propagation:** Use the `APM SDK`'s `Tracer` to instrument a simple HTTP client and server in the same test process. Verify that the `TraceID` created on the client is correctly propagated via headers and appears in the span generated by the server, linking them in the collected trace.

Chaos Testing for Resilience: Chaos testing involves deliberately injecting failures into a running system to verify its resilience and recovery mechanisms. These are run less frequently, often in a dedicated staging environment.

Failure Injection	Expected System Behavior	Verification Method
Kill the `Collector` process while spans are being ingested.	The WAL (`WALWriter`) should have recorded spans before the crash. Upon restart, the `WALReader` should replay unprocessed spans, and no data should be lost.	Compare span count before crash and after recovery.
Introduce network latency or packet loss between the SDK and Collector.	The SDK's `SpanExporter` should implement retries with exponential backoff. The `CircuitBreaker` in the Collector's ingress may open if failures persist, shedding load gracefully.	Verify spans are eventually received despite temporary network issues.
Fill the disk used by the storage backend.	The storage layer should return errors. The `BackpressureManager` should escalate to `TierAggressiveDegradation` or `TierFailSafe`, dramatically increasing sampling rates or rejecting non-critical traffic, preventing a complete crash.	Monitor system logs for escalation events and verify it remains responsive for health checks.
Simulate clock skew by feeding spans with timestamps from wildly different system times.	The `TraceAssembler` should use a watermark algorithm or buffer TTL to eventually assemble traces correctly, albeit with delay. Late-arriving spans beyond the `maxTraceDuration` should be logged and discarded.	Verify that traces with correct internal relationships are still assembled, even if out-of-order.

Load and Performance Testing: Verifying Scalability

Load tests verify that the system meets its performance requirements (e.g., 1000 spans/sec) and identify bottlenecks under stress. These tests require careful planning and monitoring.

Key Performance Tests:

1. **Ingestion Throughput:** Ramp up the rate of span submission to the Collector's OTLP endpoint until the system's response latency exceeds the 100ms SLA or spans start being dropped (monitored via `PipelineMetrics.SpansDropped`). Plot the relationship between input rate, CPU/memory usage, and latency.
2. **Concurrent Trace Assembly:** Generate a high volume of complex, multi-span traces where spans arrive out of order. Measure the memory footprint of the `BufferManager` and the accuracy of trace completion decisions.
3. **Sampling Overhead:** Measure the CPU cost of `HeadSampler.Decide` and `TailSampler.EvaluateTrace` at high decision rates. Ensure sampling logic does not become the bottleneck.
4. **Query Performance:** With a large dataset of stored traces, measure the latency of `GetTraceByID`, `GetTracesByService`, and `BuildCurrentGraph` for service maps under increasing data volumes.

Tooling: Use tools like Apache Bench (`ab`), `wrk`, or custom Go load-test clients to generate traffic. Use the application's own metrics (`PipelineMetrics`, `HeadSamplerStats`) and system-level profiling (Go's `pprof`) to identify hotspots.

Milestone Checkpoints and Verification

Each milestone has concrete acceptance criteria. The following checklists provide verifiable, binary steps to confirm functionality before moving on. Treat these as the "definition of done" for each phase.

Milestone 1: Trace Collection

Checkpoint	Verification Steps	Expected Outcome / Pass Criteria
Ingestion API Accepts Spans	1. Start the Collector with an in-memory storage backend. 2. Use `curl` or a test client to send a valid OTLP/JSON span payload to `POST /v1/traces`. 3. Check the HTTP response status and timing.	Response status is `200 OK`. Response time is < 100ms.
Spans are Parsed & Validated	1. Send a span with an invalid `TraceID` (e.g., empty string). 2. Send a span with a future `StartTime`. 3. Send a valid span with attributes and events.	Invalid spans are rejected with `400 Bad Request`. Valid span is accepted and its internal representation matches the sent data.
Spans are Indexed & Retrievable	1. Send 10 spans belonging to 2 different traces. 2. Call `GetTraceByID` for each trace ID via a test client or admin API. 3. Call `GetTracesByService` for the involved service name.	Each call returns the correct, complete set of spans for that trace or service. Spans maintain their parent-child relationships.
Throughput Requirement Met	1. Write a load test client that sustains 1000 spans/sec for 60 seconds. 2. Monitor the Collector's logs and metrics for errors or drops. 3. After the test, query for the total number of spans stored.	No errors or warnings in logs. `SpansDropped` metric is zero. Total stored span count is within 1% of 60,000 (accounting for sampling if enabled).
Late-Arriving Span Handling	1. Send Span B (child) before Span A (parent). 2. Wait longer than the buffer's typical flush time. 3. Send Span A. 4. Query for the complete trace.	The trace is eventually assembled and returned correctly, demonstrating the buffer held Span B until its parent arrived.

Milestone 2: Service Map

Checkpoint	Verification Steps	Expected Outcome / Pass Criteria
Dependency Extraction	1. Feed spans representing a chain of calls: `ServiceA -> ServiceB -> ServiceC`. 2. Run the `EdgeAggregator.ProcessSpan` logic offline or via test. 3. Inspect the generated `ServiceEdge` objects.	Two edges are created: (`ServiceA`, `ServiceB`) and (`ServiceB`, `ServiceC`). Each edge's `TotalCalls` is correct.
Graph Construction	1. Feed a more complex set of spans with multiple callers to the same service and some error responses. 2. Call `GraphBuilder.BuildCurrentGraph`. 3. Inspect the resulting `ServiceGraph`.	The graph contains the correct number of nodes and edges. Edges for calls with errors have `ErrorRate` > 0. The `P95Latency` field is populated.
Topology Change Detection	1. Build a graph for time window T1. 2. Feed new spans where a new `ServiceD` appears and calls `ServiceA`. 3. Build a graph for the next window T2. 4. Call `DetectTopologyChanges`.	The change detection correctly identifies `ServiceD` as a new node and a new edge (`ServiceD`, `ServiceA`).
Real-Time Update	1. Start a live visualization endpoint or test that consumes the graph. 2. Continuously feed spans from a changing service topology. 3. Monitor the graph output.	The graph updates within the configured refresh interval (e.g., 1 minute) to reflect new services and dependencies.

Milestone 3: Trace Sampling

Checkpoint	Verification Steps	Expected Outcome / Pass Criteria
Head-Based Sampling Rate Adherence	1. Configure a `HeadSampler` with a global rate of 0.1 (10%). 2. Send 10,000 traces with unique IDs. 3. Collect the `HeadSamplerStats`.	The ratio of `TracesSampled` to `TotalTraces` is between 9% and 11%.
Consistent Sampling per Trace	1. Send all spans for a single trace (5 spans) one by one. 2. Check the sampling decision logged for each span (or a tag on the span).	All 5 spans have the same sampling decision (`keep` / `drop`).
Tail-Based Sampling Override	1. Configure a `TailSampler` with a rule to keep traces with `Status.Code` = `ERROR`. 2. Send a trace that was initially dropped by head-sampling (rate=0) but contains an error span. 3. Check the final storage.	The error trace is present in storage, demonstrating the tail-sampler overrode the head decision.
Per-Service Sampling Configuration	1. Configure `serviceRates`: `"frontend"`: 1.0, `"backend"`: 0.01. 2. Send 1000 traces from each service. 3. Check stats per service.	Nearly all `frontend` traces are kept; only ~1% of `backend` traces are kept.

Milestone 4: Performance Analytics & Anomaly Detection

Checkpoint	Verification Steps	Expected Outcome / Pass Criteria
Percentile Calculation (t-digest)	1. Feed 10,000 latency values with a known distribution (e.g., uniformly distributed between 100ms and 1000ms) into a `TDigestMetric`. 2. Query `p50`, `p95`, `p99`. 3. Compare to theoretical values.	Calculated percentiles are within 1% of the expected values (e.g., p95 ~= 955ms).
Anomaly Detection Trigger	1. Establish a baseline of latencies for a service (~100ms). 2. Inject a series of latencies spiking to 500ms. 3. Run `Detector.CheckMetric`. 4. Inspect `AnomalyResult`.	Anomalies are detected with high `Confidence`. The `Severity` is appropriate (e.g., `"high"`).
Time-Series Storage & Retrieval	1. Write `MetricPoint`s for various services over a 1-hour period. 2. Query the `MemoryStore` for a specific 15-minute window. 3. Query for a service that has no data in that window.	The correct subset of points is returned for the first query. The second query returns an empty slice (not an error).
Regression Alerting	1. Set up a test alerting channel (e.g., log file). 2. Trigger an anomaly as above. 3. Monitor the channel.	An alert message containing the `AnomalyResult` details is emitted.

Milestone 5: APM SDK & Auto-Instrumentation

Checkpoint	Verification Steps	Expected Outcome / Pass Criteria
HTTP Context Propagation	1. Instrument an HTTP client and server using the SDK. 2. Make a request from client to server. 3. Collect the trace from the Collector. 4. Inspect the trace structure.	The trace contains two spans (client and server) linked by parent-child relationship. The `TraceID` is identical in both spans.
Database Query Tracing	1. Instrument a SQL database driver. 2. Execute a `SELECT` query. 3. Collect the resulting span.	A span is created with the `Name` containing the SQL operation. The span's attributes contain the sanitized query string and execution time.
Framework Middleware	1. Add the Gin middleware to a simple Go web server. 2. Make an HTTP request to a registered route. 3. Collect the trace.	A span is created for the incoming request, with the `Name` as the route path (e.g., `GET /api/users`). The span's duration matches the request handling time.
Async Context Propagation	1. In an instrumented handler, launch a goroutine that creates a child span. 2. Collect the trace.	The child span created in the goroutine is correctly linked as a child of the handler span, demonstrating the context was propagated.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (For Learning)	Advanced Option (For Production)
Unit Testing Framework	Go's built-in `testing` package + `testify/assert`	Go's `testing` + `testify/suite` for test suites
Mock Generation	Hand-written fakes for key interfaces	`mockgen` (from `github.com/golang/mock`)
Property-Based Testing	`github.com/leanovate/gopter`	`github.com/leanovate/gopter` with custom generators
Load Testing	Custom Go program using goroutines	`k6` or `ghz` (for gRPC)
Code Coverage	`go test -cover`	`go test -coverprofile` + `go tool cover -html=`
Benchmarking	Go's built-in benchmarking (`go test -bench .`)	`go test -bench . -benchmem` + pprof profiling

B. Recommended File/Module Structure for Tests

```
apm-tracing-system/
├── cmd/
│   └── collector/
│       └── main.go
├── internal/
│   ├── ingestion/              # Milestone 1
│   │   ├── pipeline.go
│   │   ├── pipeline_test.go    # Unit tests
│   │   ├── handler_otlp.go
│   │   └── handler_otlp_test.go
│   ├── storage/
│   │   ├── memory.go           # Fake for testing
│   │   ├── memory_test.go
│   │   └── writer.go           # Interface
│   ├── sampling/               # Milestone 3
│   │   ├── head_sampler.go
│   │   ├── head_sampler_test.go
│   │   ├── property_test.go    # Property-based tests
│   │   └── tail_sampler.go
│   ├── servicegraph/           # Milestone 2
│   │   ├── builder.go
│   │   ├── builder_integration_test.go
│   │   └── aggregator_test.go
│   ├── analytics/              # Milestone 4
│   │   ├── tdigest_metric.go
│   │   ├── tdigest_metric_test.go
│   │   └── detector_test.go
│   └── sdk/                    # Milestone 5
│       ├── tracer.go
│       ├── tracer_integration_test.go
│       └── http_client_test.go
├── test/                       # Integration & load tests
│   ├── integration/
│   │   ├── collector_test.go   # Spans ingestion -> storage
│   │   └── service_map_test.go
│   ├── load/
│   │   └── span_throughput.go  # Load test program
│   └── chaos/
│       └── kill_collector_test.go
└── pkg/
    └── models/                 # Data models
        ├── span.go
        └── span_test.go        # Validation logic tests
```

C. Infrastructure Starter Code: Mock Span Generator

```go
// testutils/generator.go

package testutils


import (

    "crypto/rand"

    "encoding/hex"

    "time"


    "github.com/your-org/apm-tracing-system/pkg/models"

)


// GenerateSpan creates a deterministic or random span for testing.

func GenerateSpan(traceID, parentSpanID, serviceName, operation string) models.Span {

    if traceID == "" {

        traceID = generateID(16)

    }

    spanID := generateID(8)

    start := time.Now().Add(-time.Duration(randomInt(100)) * time.Millisecond)

    duration := time.Duration(randomInt(500)+50) * time.Millisecond


    return models.Span{

        SpanID:       spanID,

        TraceID:      traceID,

        ParentSpanID: parentSpanID,

        Name:         operation,

        ServiceName:  serviceName,

        StartTime:    start,

        Duration:     duration,

        Attributes:   map[string]string{"test": "true"},

        Status:       models.SpanStatus{Code: 0}, // OK

    }

}


// GenerateTrace generates a complete trace with a root span and n children.

func GenerateTrace(serviceName string, childCount int) []models.Span {
```

```
    traceID := generateID(16)
    root := GenerateSpan(traceID, "", serviceName, "root-operation")
    spans := []models.Span{root}

    for i := 0; i < childCount; i++ {
        child := GenerateSpan(traceID, root.SpanID, serviceName,
                              "child-operation")
        spans = append(spans, child)
    }
    return spans
}

func generateID(length int) string {
    bytes := make([]byte, length/2) // 2 hex chars per byte
    rand.Read(bytes)
    return hex.EncodeToString(bytes)
}

func randomInt(max int) int {
    // simplified deterministic random for tests
    return int(time.Now().UnixNano()) % max
}
```

D. Core Logic Skeleton Code for Key Unit Tests

```go
// internal/sampling/head_sampler_property_test.go

package sampling

import (
    "testing"
    "github.com/leanovate/gopter"
    "github.com/leanovate/gopter/gen"
    "github.com/leanovate/gopter/prop"
)

func TestConsistentSampler_PropertyBased(t *testing.T) {
    parameters := gopter.DefaultTestParameters()
    parameters.MinSuccessfulTests = 1000
    properties := gopter.NewProperties(parameters)

    properties.Property("deterministic sampling for same traceID",
        prop.ForAll(
            func(traceID string, rate float64) bool {
                sampler := NewConsistentSampler(rate)
                decision1 := sampler.ShouldSample(traceID)
                decision2 := sampler.ShouldSample(traceID)
                // TODO 1: The decision for the same traceID should be identical
                return decision1 == decision2
            },
            gen.AlphaString(),  // Generates random traceIDs
            gen.Float64Range(0.0, 1.0), // Generates random rates
        ))

    properties.Property("sampling rate converges to configured rate",
        prop.ForAll(
            func(rate float64) bool {
                sampler := NewConsistentSampler(rate)
                samples := 10000
                kept := 0
                // TODO 2: Generate a large set of unique traceIDs
```

```
                // TODO 3: Call ShouldSample for each traceID
                // TODO 4: Count how many were kept
                // TODO 5: The ratio kept/samples should be within 1% of 'rate'
                // return math.Abs(float64(kept)/float64(samples) - rate) < 0.01
                return true
            },
            gen.Float64Range(0.05, 0.95), // Avoid extremes for meaningful test
        ))

    properties.TestingRun(t)
}
```

```go
// internal/ingestion/pipeline_integration_test.go

package ingestion


import (

    "context"

    "testing"

    "github.com/stretchr/testify/assert"

    "github.com/stretchr/testify/require"

    "github.com/your-org/apm-tracing-system/pkg/models"

    "github.com/your-org/apm-tracing-system/testutils"

)


func TestPipeline_EndToEnd(t *testing.T) {

    // TODO 1: Create an in-memory storage fake

    // storage := memory.NewStorage()


    // TODO 2: Create a Pipeline with a head sampler (rate=1.0) and the fake storage

    // pipeline, err := NewPipeline(config)

    // require.NoError(t, err)


    // TODO 3: Generate a test trace with 3 spans

    // spans := testutils.GenerateTrace("test-service", 2)


    // TODO 4: Process each span through the pipeline

    // for _, span := range spans {

    //     err := pipeline.ProcessSpan(context.Background(), span)

    //     assert.NoError(t, err)

    // }


    // TODO 5: Query the storage for the complete trace by TraceID

    // trace, err := storage.GetTraceByID(context.Background(), spans[0].TraceID)


    // TODO 6: Assert the trace contains all 3 spans

    // assert.Len(t, trace.Spans, 3)
```

```
    // assert.Equal(t, spans[0].TraceID, trace.TraceID)
}
```

E. Language-Specific Hints (Go)

- **Use `t.Helper()`:** Mark helper functions in tests so the failure line points to the actual test, not the helper.
- **Table-Driven Tests:** Use for testing multiple input/output combinations for the same function.

```go
func TestSpanValidation(t *testing.T) {
    tests := []struct{
        name string
        span models.Span
        wantErr bool
    }{
        {"valid span", models.Span{TraceID: "abc", ...}, false},
        {"missing traceid", models.Span{}, true},
    }
    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T){
            err := tt.span.Validate()
            if tt.wantErr {
                assert.Error(t, err)
            } else {
                assert.NoError(t, err)
            }
        })
    }
}
```

- **Parallel Tests:** Use `t.Parallel()` in independent tests to speed up execution.
- **Test Main:** Use `func TestMain(m *testing.M)` for global setup/teardown (e.g., starting a test database container).
- **Benchmarks:** Place benchmarks in `_test.go` files with functions starting with `Benchmark`. Use `b.ResetTimer()` and `b.StopTimer()` to exclude setup time.

F. Milestone Checkpoint Execution For each milestone, after implementing the components, run the relevant test suite:

1. **Milestone 1:** `go test ./internal/ingestion/... -v -count=1`
2. **Milestone 2:** `go test ./internal/servicegraph/... -v`
3. **Milestone 3:** `go test ./internal/sampling/... -v`

4. **Milestone 4:** `go test ./internal/analytics/... -v`
5. **Milestone 5:** `go test ./internal/sdk/... -v`

Then, run the integration tests: `go test ./test/integration/... -v`

Expected Output: All tests pass (exit code 0). For property-based tests, you'll see output like `"OK: 1000 passed"`.

Signs of Trouble:

- **Tests hang:** Check for deadlocks in concurrent code (use `-timeout` flag).
- **High memory in tests:** May indicate a leak in `BufferManager` or `WALReader`; run with `-memprofile`.
- **Flaky tests:** Usually a concurrency bug or reliance on system time. Use fake clocks and synchronized channels.

G. Debugging Tips for Test Failures

Symptom	Likely Cause	How to Diagnose	Fix
Span linking fails in trace assembly test.	Incorrect parent-child relationship logic or `SpanID` generation.	Log the `TraceID`, `SpanID`, and `ParentSpanID` of all spans in the test. Draw the expected tree.	Verify `NewTraceFromSpans` groups by `TraceID` and builds the tree correctly.
Sampling rate test fails (ratio is 0% or 100%).	Hash function in `ConsistentSampler` is incorrectly mapped to the rate.	Print the hash value for a few `traceID`s and the threshold.	Ensure hash is uniformly distributed and comparison is `hash % 10000 < rate*10000`.
Service map missing edges.	`EdgeAggregator` is not extracting caller/callee from spans correctly.	Check that spans representing cross-service calls have different `ServiceName` and correct parent links.	In `ProcessSpan`, ensure you extract the caller's service from the parent span.
Anomaly detector floods alerts.	Baseline calculation is wrong or detection threshold is too sensitive.	Log the historical baseline values and the current value with z-score.	Adjust the `thresholds` in `Detector` or increase the baseline window size.
Integration test passes alone but fails in suite.	Shared global state (like a package-level variable) is not reset.	Look for `init()` functions or `var` declarations that hold state.	Use dependency injection, not globals. Reset state in a `TestMain` or `setup/teardown`.

13. Debugging Guide

> **Milestone(s):** This section applies to all five milestones, providing a practical manual for diagnosing and fixing common issues encountered while building and running the APM Tracing System.

Building a distributed tracing system involves multiple interconnected components that process high-volume data streams in real-time. When things go wrong—spans disappear, performance degrades, or metrics become inaccurate—the complexity can make debugging daunting. This guide provides a structured approach to diagnosing and resolving common issues, along with practical techniques and tools that will help you navigate the system's inner workings.

Common Bugs: Symptom → Cause → Fix

Think of debugging the APM system like being a **systems detective**. Each symptom is a clue, each cause is a suspect, and your diagnosis steps are the investigative process to eliminate possibilities until you find the culprit. The following table organizes common issues by symptom, with concrete steps to identify root causes and implement fixes.

Symptom	Likely Cause	Diagnosis Steps	Fix
Spans are not linked into traces (spans with the same `TraceID` appear as separate, incomplete traces)	1. **Parent-child linking logic error** in `TraceAssembler.AssembleTrace()` 2. **Missing or incorrect** `ParentSpanID` in incoming spans 3. **Clock skew** between services causing incorrect span ordering 4. **Late-arriving spans** evicted from `BufferManager` before assembly	1. Check `TraceAssembler` logs for assembly errors 2. Verify span JSON/protobuf payloads contain correct `ParentSpanID` 3. Compare timestamps across services—look for negative durations 4. Monitor `BufferManager` eviction metrics and trace completion rates	1. Add validation to ensure `ParentSpanID` format matches `SpanID` 2. Implement clock skew compensation in `TraceAssembler` 3. Increase `BufferManager` TTL or adjust eviction policy 4. Add telemetry to track span arrival latency relative to trace start
Service map is empty or missing edges (dependency graph shows no connections between services)	1. **Edge extraction logic** failing to identify caller-callee relationships 2. **Aggregation window misalignment** causing edges to be discarded 3. **Sampling dropping all inter-service traces** 4. `ServiceName` **extraction incorrect** from OTLP resource attributes	1. Inspect raw spans to verify they contain cross-service references 2. Check `EdgeAggregator` window boundaries and flush timing 3. Review `HeadSampler` and `TailSampler` statistics for service 4. Validate `extractServiceName()` logic against actual OTLP payloads	1. Enhance `EdgeAggregator.ProcessSpan()` to log extracted edges for debugging 2. Align aggregation windows with trace collection windows 3. Adjust sampling rates or add sampling bypass for test traffic 4. Update `extractServiceName()` to handle multiple attribute formats
High memory usage in collector (memory grows unbounded, leads to OOM crashes)	1. **Incomplete trace accumulation** in `BufferManager` without eviction 2. **Memory leak** in WAL or pipeline components 3. **Backpressure failure** causing queue buildup 4. **Large attribute values** stored in span `Attributes` maps	1. Monitor `BufferManager.traces` map size and age distribution 2. Use Go pprof heap profiling to identify allocation hotspots 3. Check `BackpressureManager` tier transitions and sampling adjustments 4. Inspect span sizes in storage—look for exceptionally large attributes	1. Implement more aggressive `BufferManager` eviction based on trace age 2. Add periodic resource cleanup in long-lived goroutines 3. Tune `BackpressureManager` thresholds for earlier intervention 4. Enforce attribute size limits in `SpanValidator`

Symptom	Likely Cause	Diagnosis Steps	Fix
Slow trace queries (`GetTraceByID` takes >1s even for small traces)	1. **Missing or inefficient indexes** on trace storage 2. **N+1 query problem** fetching spans individually 3. **Storage layer contention** from high ingestion volume 4. **Cache misses** in query service layer	1. Examine query execution plans in storage backend 2. Profile query service to identify repeated round-trips 3. Monitor storage latency metrics during peak ingestion 4. Check cache hit rates for frequently accessed traces	1. Ensure composite index on (`TraceID` , `StartTime`) in storage 2. Implement batch span fetching in `GetTraceByID` 3. Add read replicas or query-side caching for trace data 4. Implement query result caching with TTL based on trace age
Sampling rate not being respected (too many/few traces stored vs. configured probability)	1. **Inconsistent hashing** across sampler instances 2. **Race condition** updating `ConsistentSampler.rate` 3. **Tail-based sampling overriding** too many head decisions 4. **Configuration propagation delay** across collector instances	1. Compare hash outputs for same `TraceID` across sampler instances 2. Check for concurrent `SetRate()` calls without proper synchronization 3. Review `TailSampler` statistics—high `TailOverrides` indicates aggressive overriding 4. Monitor configuration timestamps vs. sampling decision timestamps	1. Use deterministic hash function (e.g., xxHash) with fixed seed 2. Add mutex protection in `ConsistentSampler.SetRate()` 3. Adjust `TailSamplingRule` priorities and thresholds 4. Implement configuration versioning with decision logging
Anomaly detection producing too many false positives (alerts fire during normal operation)	1. **Insufficient historical baseline** for comparison 2. **Statistical method sensitivity** too high (low z-score threshold) 3. **Seasonal patterns** not accounted for in baseline 4. **Metric aggregation window** misaligned with traffic patterns	1. Check baseline calculation—minimum data points required 2. Review anomaly `Confidence` scores—consistently low indicates weak signal 3. Plot metric history to identify daily/weekly patterns 4. Compare aggregation window with natural traffic cycles	1. Increase baseline training period before enabling detection 2. Adjust `Detector.thresholds` based on observed false positive rate 3. Implement seasonal decomposition in `BaselineCalculator` 4. Experiment with different aggregation windows (1m vs 5m vs 15m)
SDK causing performance overhead (instrumented application shows >5% latency increase)	1. **Synchronous span export** blocking request processing 2. **Excessive attribute collection** on hot code paths 3. **Context propagation overhead** in deep call chains 4. **Lock contention** in shared `Tracer` structures	1. Profile application with and without SDK to isolate overhead 2. Measure time spent in `End()` and span export operations 3. Count context propagation operations per request 4. Check goroutine profiles for lock wait times	1. Implement asynchronous batched span export in SDK 2. Add sampling to attribute collection on high-volume operations 3. Optimize `context.Context` value storage/retrieval 4. Use `sync.Pool` for reusable span objects
Clock skew making span	1. **System clock differences** between services (seconds/minutes offset)	1. Compare `StartTime` of parent and child spans	1. Implement relative timestamp adjustment in `TraceAssembler`

Symptom	Likely Cause	Diagnosis Steps	Fix
ordering incorrect (child spans appear to start before parents)	2. **NTP synchronization issues** causing drift 3. **Virtual machine clock skew** in containerized environments 4. **Timestamp normalization logic** not accounting for timezone	across services 2. Check system clock synchronization status on collector and services 3. Monitor for gradual drift in span timing over hours/days 4. Verify timestamp parsing handles UTC conversion correctly	2. Add NTP client to collector to maintain accurate time 3. Use monotonic clocks for duration calculations where possible 4. Normalize all timestamps to UTC at ingestion point
Backpressure not engaging under load (system becomes unresponsive instead of degrading gracefully)	1. **Threshold detection lag**—metrics don't reflect reality quickly enough 2. **Tier transition logic** too conservative 3. **Sampling rate adjustment** insufficient to reduce load 4. **Feedback loop delay** between action and effect	1. Monitor `SystemMetrics` vs. `BackpressureManager` tier decisions 2. Check tier transition history for delayed responses to load spikes 3. Compare sampling rate changes with span ingestion rate changes 4. Measure time from tier change to metric improvement	1. Reduce `checkInterval` and use more sensitive thresholds 2. Implement predictive tier transitions based on trend analysis 3. Add exponential backoff to sampling rate reductions 4. Create faster feedback loop with direct metric-to-action coupling
WAL recovery losing spans after collector restart	1. **WAL corruption** due to improper file rotation 2. **Incomplete records** written before crash 3. **Race condition** between write and rotation 4. **Disk space exhaustion** during WAL append	1. Check WAL file integrity—look for malformed length prefixes 2. Verify last few records in WAL are complete 3. Examine rotation timing relative to crash time 4. Monitor disk usage metrics before restart	1. Add record checksums to detect corruption 2. Implement atomic record writing with fsync 3. Use file locking during rotation 4. Add disk space monitoring and proactive WAL cleanup

> **Key Insight:** Many APM system bugs manifest as **data quality issues**—missing links, incorrect timing, or inaccurate aggregations. Always start your investigation by examining the raw data flowing through the system. Add debug logging to capture a sample of problematic spans before and after each processing stage.

Debugging Techniques and Tools

Effective debugging requires both systematic methodology and the right tools. Think of this as your **detective's toolkit**—each tool serves a specific purpose in uncovering different types of issues.

Strategic Logging

Well-placed logs are your first line of defense. Unlike metrics that aggregate, logs preserve individual request context, which is crucial for debugging trace assembly, sampling decisions, and edge cases.

Where to Add Diagnostic Logging:

Component	Critical Log Points	What to Log
`OTLPHandler`	Request parsing errors, payload size anomalies	`traceID`, `spanID`, error details, payload size
`Pipeline.ProcessSpan()`	Span validation failures, sampling decisions	`traceID`, `spanID`, validation error, sampler decision
`BufferManager.AddSpan()`	Trace completion, eviction events	`traceID`, span count, buffer size, eviction reason
`EdgeAggregator.ProcessSpan()`	Edge extraction, window flushes	caller/callee services, extracted edge metrics
`HeadSampler.Decide()`	Sampling decisions, rate changes	`traceID`, service, rate, decision
`TailSampler.EvaluateTrace()`	Rule evaluations, override decisions	`traceID`, rule matches, final decision
`CircuitBreaker.Execute()`	State transitions, operation failures	operation name, state change, error

Implementing Structured Logging:

```go
// Use a structured logger like zap or logrus
logger.Info("Span processing completed",
    zap.String("trace_id", span.TraceID),
    zap.String("span_id", span.SpanID),
    zap.String("service", span.ServiceName),
    zap.Duration("duration", span.Duration),
    zap.Int("buffer_size", bufferSize))
```

Log Sampling for High Volume: Enable debug logging only for a percentage of traces to avoid overwhelming the log system. Use the same `ConsistentSampler` mechanism to ensure the same traces are logged across components.

Performance Profiling with pprof

When the system experiences high CPU, memory leaks, or goroutine explosions, Go's built-in profiling tools (`pprof`) are indispensable. Think of pprof as an **X-ray machine** for your running application—it reveals internal structures and hotspots invisible from the outside.

Enabling pprof in Your Collector:

```go
// Add to your HTTP server setup
import _ "net/http/pprof"

go func() {
    log.Println(http.ListenAndServe("localhost:6060", nil))
}()
```

Common pprof Investigations:

Symptom	Profile Type	What to Look For
High CPU usage	`cpu` profile	Functions with highest `cumulative` time; often reveals hot loops or expensive serialization
Memory leaks	`heap` profile	Objects with increasing `inuse_space` over time; check for retained references in global maps
Goroutine leaks	`goroutine` profile	Growing goroutine count; look for blocked goroutines in sync primitives or channel operations
Contention	`mutex` profile	Functions with high lock contention; indicates scalability bottlenecks
Blocking	`block` profile	Operations causing goroutines to wait; often I/O or channel operations

Diagnostic Workflow:

1. **Reproduce the issue** under load (use simulation tools below)
2. **Collect profiles** during the problematic behavior:

```bash
# CPU profile (30 seconds)
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30

# Heap snapshot
go tool pprof http://localhost:6060/debug/pprof/heap

# Goroutine dump
curl http://localhost:6060/debug/pprof/goroutine?debug=2 > goroutines.txt
```

3. **Analyze with pprof web interface:**

```bash
# Launch interactive web UI
go tool pprof -http=:8080 profile.pprof
```

4. **Look for patterns:** Repeated allocation patterns, deep call stacks with many allocations, or single functions dominating CPU time.

> **Pro Tip:** Use **comparative profiling**—take profiles before and after a code change, or during normal vs. degraded operation. The differences often reveal the root cause more clearly than absolute profiles.

Simulating Production Traffic for Testing

You cannot debug what you cannot reproduce. Creating realistic test traffic is essential for diagnosing issues that only appear under production-like load.

Traffic Simulation Approaches:

Technique	Best For	Implementation
Replay from WAL	Reproducing specific failure scenarios	Use `WALReader.ReadNext()` to replay spans exactly as they arrived
Synthetic trace generation	Load testing and stress scenarios	Programmatically generate traces with realistic service call patterns
Record-and-replay	Capturing production patterns without PII	Capture trace samples (anonymized) and replay through test collector
Chaos injection	Testing resilience and failure recovery	Randomly drop spans, introduce delays, or corrupt data

Building a Trace Generator for Testing:

```go
// Generate a realistic multi-service trace
func generateTestTrace(traceID string) []models.Span {
    spans := []models.Span{}

    // Root span (web request)
    spans = append(spans, models.Span{
        TraceID: traceID,
        SpanID: generateID(),
        ParentSpanID: "",
        Name: "HTTP GET /api/users",
        ServiceName: "frontend",
        StartTime: time.Now(),
        Duration: 120 * time.Millisecond,
    })

    // Database call child span
    spans = append(spans, models.Span{
        TraceID: traceID,
        SpanID: generateID(),
        ParentSpanID: spans[0].SpanID,
        Name: "SELECT users",
        ServiceName: "users-db",
        StartTime: time.Now().Add(10 * time.Millisecond),
        Duration: 45 * time.Millisecond,
    })

    // External API call child span
    spans = append(spans, models.Span{
        TraceID: traceID,
        SpanID: generateID(),
        ParentSpanID: spans[0].SpanID,
        Name: "GET /recommendations",
        ServiceName: "recommendation-service",
```

```
        StartTime: time.Now().Add(20 * time.Millisecond),
        Duration: 80 * time.Millisecond,
    })

    return spans
}
```

Load Testing Script Pattern:

```bash
#!/bin/bash
# Run a controlled load test with increasing volume
for rate in 100 500 1000 2000 5000; do
    echo "Testing at ${rate} spans/sec"
    ./trace-generator --rate=$rate --duration=60s \
        --collector-endpoint="http://localhost:4318/v1/traces"

    # Wait for system to stabilize and collect metrics
    sleep 10
    curl http://localhost:6060/debug/pprof/heap > heap_${rate}.pprof
done
```

Using the System to Debug Itself (Dogfooding)

The most powerful debugging technique for an APM system is to **use it to monitor itself**. Instrument your collector, query service, and storage layer with the same SDK you provide to applications. This creates a virtuous cycle where debugging improvements benefit both the system and its users.

Self-Instrumentation Strategy:

1. **Add tracing to collector ingress/egress:** Wrap `OTLPHandler.ServeHTTP()` with spans to track request handling latency and errors.
2. **Trace internal pipeline operations:** Add spans around `BufferManager` operations, `EdgeAggregator` processing, and `TailSampler` evaluations.
3. **Monitor system health with built-in analytics:** Feed collector metrics into your own anomaly detection system to get early warnings of degradation.
4. **Build service maps of internal components:** Visualize data flow between pipeline stages to identify bottlenecks.

Example Collector Self-Instrumentation:

```go
func (h *OTLPHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    // Create a span for this request
    ctx, span := h.tracer.StartSpanFromContext(r.Context(), "OTLPHandler.ServeHTTP")
    defer span.End()

    span.SetAttribute("http.method", r.Method)
    span.SetAttribute("http.url", r.URL.Path)
    span.SetAttribute("content.type", r.Header.Get("Content-Type"))

    // Process the request with the context containing the span
    h.processRequestWithContext(ctx, w, r)

    span.SetAttribute("http.status_code", statusCode)
    if statusCode >= 400 {
        span.SetStatus(models.SpanStatus{Code: 2, Message: "HTTP error"})
    }
}
```

> **Design Principle:** A well-instrumented system is a debuggable system. The effort you invest in adding observability to your APM system pays dividends every time you need to diagnose a production issue.

Diagnostic Endpoints and Health Checks

Beyond pprof, build custom diagnostic endpoints that expose internal state. These are your **system vitals monitor**—quick ways to check pulse, respiration, and other critical signs.

Essential Diagnostic Endpoints:

Endpoint	Purpose	Response Fields
`GET /debug/state/buffer`	`BufferManager` status	`total_traces`, `total_spans`, `oldest_trace_age`, `eviction_count`
`GET /debug/state/sampling`	Sampler statistics	`head_sampled_rate`, `tail_overrides`, `per_service_decisions`
`GET /debug/state/edges`	Service map construction	`edge_count`, `last_flush_time`, `pending_edges`
`GET /debug/trace/:id`	Manual trace inspection	Complete trace with all spans (bypasses normal query path)
`POST /debug/inject-fault`	Chaos engineering	Inject delays, errors, or data corruption for testing resilience

Example Diagnostic Handler:

```go
func (m *BufferManager) DebugHandler(w http.ResponseWriter, r *http.Request) {
    m.mu.RLock()
    defer m.mu.RUnlock()

    stats := map[string]interface{}{
        "traces_count": len(m.traces),
        "spans_count":  m.totalSpans(),
        "max_size":     m.maxSize,
        "ttl":          m.ttl.String(),
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(stats)
}
```

Systematic Diagnosis Workflow

When faced with an unknown issue, follow this **systematic diagnosis workflow** to avoid rabbit holes and confirmation bias:

1. **Reproduce Consistently:** Can you make it happen on demand? If not, add logging to capture it next time.
2. **Isolate the Component:** Use the architecture diagram to identify which component is likely responsible. Check component health endpoints.
3. **Examine Inputs/Outputs:** Compare what enters the component vs. what leaves. Add debug logging at boundaries.
4. **Check Internal State:** Use diagnostic endpoints to inspect internal data structures and counters.
5. **Simplify and Test:** Create a minimal test case that reproduces the issue with controlled inputs.
6. **Hypothesize and Verify:** Form a hypothesis about the root cause, make a prediction, and test it.
7. **Fix and Monitor:** Implement the fix, then monitor to ensure the issue is resolved and doesn't regress.

Remember that in distributed tracing systems, many issues are **timing-related** (race conditions, late arrivals, clock skew) or **data quality issues** (malformed spans, incorrect context propagation). Always consider these categories early in your diagnosis.

Implementation Guidance

This section provides concrete code patterns and tools to implement the debugging techniques described above.

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Structured Logging	`log/slog` (Go 1.21+)	`uber-go/zap` with sampling and hooks
Profiling	`net/http/pprof` embedded	Custom profiling endpoints with filtering
Metrics Collection	`expvar` for basic metrics	`prometheus/client_golang` for rich metrics
Trace Generation	Custom Go generator	OpenTelemetry Collector test data generator
Health Checks	Simple HTTP endpoint	Kubernetes-ready liveness/readiness probes
Diagnostic UI	pprof web interface	Custom React dashboard aggregating all debug endpoints

B. Diagnostic Endpoint Implementation

Create a dedicated debugging module that exposes system state without affecting production performance:

File Structure:

```
project-root/
  internal/debug/
    handler.go           # HTTP handlers for debug endpoints
    pprof_wrapper.go     # Enhanced pprof with authentication
    state_dumper.go      # Component state inspection
    metrics.go           # Debug-specific metrics collection
  cmd/debug-tool/
    main.go              # CLI tool for offline diagnostics
```

Complete State Dumper Implementation:

```go
// internal/debug/state_dumper.go

package debug


import (

    "encoding/json"

    "fmt"

    "net/http"

    "sync"

    "time"

)


// ComponentState represents a component's internal state for debugging

type ComponentState struct {

    Name        string                 `json:"name"`

    Health      string                 `json:"health"`

    Metrics     map[string]interface{} `json:"metrics"`

    LastUpdated time.Time              `json:"last_updated"`

}


// StateDumper collects and serves debug state from all registered components

type StateDumper struct {

    mu         sync.RWMutex

    components map[string]func() ComponentState

}


// NewStateDumper creates a new state dumper

func NewStateDumper() *StateDumper {

    return &StateDumper{

        components: make(map[string]func() ComponentState),

    }

}


// RegisterComponent adds a component state provider

func (d *StateDumper) RegisterComponent(name string, provider func() ComponentState) {

    d.mu.Lock()
```

```
    defer d.mu.Unlock()
    d.components[name] = provider
}

// ServeHTTP implements http.Handler
func (d *StateDumper) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    d.mu.RLock()
    defer d.mu.RUnlock()

    states := make(map[string]ComponentState)
    for name, provider := range d.components {
        states[name] = provider()
    }

    w.Header().Set("Content-Type", "application/json")
    if err := json.NewEncoder(w).Encode(states); err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
    }
}

// BufferManagerStateProvider example
func BufferManagerStateProvider(bm *BufferManager) func() ComponentState {
    return func() ComponentState {
        bm.mu.RLock()
        defer bm.mu.RUnlock()

        return ComponentState{
            Name:   "buffer_manager",
            Health: "healthy",
            Metrics: map[string]interface{}{
                "traces_count":      len(bm.traces),
                "spans_count":       bm.totalSpans(),
                "oldest_trace_age":  time.Since(bm.oldestTraceTime()).String(),
```

```
                "evictions_total":   bm.evictionCount,
                "evictions_recent":  bm.recentEvictionCount,
            },
            LastUpdated: time.Now(),
        }
    }
}
```

C. Strategic Logging Skeleton

Implement a logging wrapper that adds trace context to all log messages automatically:

```go
// internal/telemetry/logger.go

package telemetry


import (

    "context"

    "time"



    "go.uber.org/zap"

    "go.uber.org/zap/zapcore"

)


// ContextLogger wraps a zap logger with context awareness

type ContextLogger struct {

    logger *zap.Logger

}


// NewContextLogger creates a new context-aware logger

func NewContextLogger(level zapcore.Level) (*ContextLogger, error) {

    config := zap.NewProductionConfig()

    config.Level = zap.NewAtomicLevelAt(level)

    config.EncoderConfig.TimeKey = "timestamp"

    config.EncoderConfig.EncodeTime = zapcore.ISO8601TimeEncoder



    logger, err := config.Build()

    if err != nil {

        return nil, err

    }



    return &ContextLogger{logger: logger}, nil

}


// WithContext returns a logger with trace context fields added

func (cl *ContextLogger) WithContext(ctx context.Context) *zap.Logger {

    logger := cl.logger
```

```
    // Extract trace context from context
    if span := SpanFromContext(ctx); span != nil {
        logger = logger.With(
            zap.String("trace_id", span.TraceID),
            zap.String("span_id", span.SpanID),
        )
    }

    return logger
}

// SampledDebug logs debug messages only for sampled traces
func (cl *ContextLogger) SampledDebug(ctx context.Context, msg string, fields ...zap.Field) {
    if span := SpanFromContext(ctx); span != nil {
        // Sample 1% of traces for debug logging
        if hashTraceID(span.TraceID)%100 == 0 {
            cl.WithContext(ctx).Debug(msg, fields...)
        }
    }
}

// hashTraceID helper for consistent sampling
func hashTraceID(traceID string) uint64 {
    // Use xxHash for fast, consistent hashing
    // Implementation omitted for brevity
    return 0
}
```

D. Load Test Generator Skeleton

Create a trace generator that can simulate realistic production traffic patterns:

```go
// cmd/trace-generator/main.go

package main


import (

    "context"

    "flag"

    "log"

    "time"



    "github.com/your-project/internal/models"

)


type GeneratorConfig struct {

    Rate        int           // Spans per second

    Duration    time.Duration // How long to run

    Services    []string      // Service names to include

    ErrorRate   float64       // Percentage of spans with errors

    MaxDepth    int           // Maximum call depth in traces

}


// TraceGenerator creates synthetic traces

type TraceGenerator struct {

    config GeneratorConfig

    stats  GeneratorStats

}


// GenerateTrace creates a single trace with realistic structure

func (tg *TraceGenerator) GenerateTrace() *models.Trace {

    traceID := generateUUID()



    // TODO 1: Create root span for entry point (e.g., "HTTP GET /api")

    // TODO 2: Randomly determine trace depth (1 to MaxDepth)

    // TODO 3: For each level, create child spans for different services

    // TODO 4: Apply ErrorRate to randomly mark some spans as errors

    // TODO 5: Add realistic durations (parent > sum of children)
```

```
    // TODO 6: Include attributes (URL paths, DB queries, status codes)


    return &models.Trace{
        TraceID:   traceID,
        Spans:     []models.Span{/* populated spans */},
        StartTime: time.Now(),
        EndTime:   time.Now().Add(100 * time.Millisecond),
    }
}

// Run generates traces at the configured rate
func (tg *TraceGenerator) Run(ctx context.Context) error {
    ticker := time.NewTicker(time.Second / time.Duration(tg.config.Rate))
    defer ticker.Stop()

    deadline := time.After(tg.config.Duration)

    for {
        select {
        case <-ctx.Done():
            return ctx.Err()
        case <-deadline:
            log.Printf("Generation complete. Sent %d traces", tg.stats.TracesGenerated)
            return nil
        case <-ticker.C:
            trace := tg.GenerateTrace()
            // TODO 7: Send trace to collector via OTLP
            tg.stats.TracesGenerated++
        }
    }
}

func main() {
```

```
    var rate int
    var duration time.Duration
    flag.IntVar(&rate, "rate", 100, "Spans per second")
    flag.DurationVar(&duration, "duration", 30*time.Second, "Test duration")
    flag.Parse()

    config := GeneratorConfig{
        Rate:      rate,
        Duration:  duration,
        Services:  []string{"frontend", "users-service", "orders-service", "payments-service", "database"},
        ErrorRate: 0.01, // 1% error rate
        MaxDepth:  4,
    }

    generator := &TraceGenerator{config: config}
    ctx := context.Background()

    if err := generator.Run(ctx); err != nil {
        log.Fatal(err)
    }
}
```

E. Language-Specific Hints

- **Go pprof:** Use `go tool pprof -http=:8080 profile.pprof` to launch the web UI. The flame graph view is particularly useful for identifying CPU hotspots.
- **Memory profiling:** Run your tests with `-memprofile=mem.pprof` and `-memprofilerate=1` to get detailed allocation data.
- **Race detection:** Always run tests with `-race` flag during development to catch data races early.
- **Goroutine leaks:** Use `runtime.NumGoroutine()` in your health checks to monitor goroutine counts.
- **Block profiling:** Enable with `runtime.SetBlockProfileRate(1)` to capture all blocking operations.

F. Debugging Workflow Checkpoint

After implementing the debugging infrastructure, verify it works:

1. **Start the collector** with debug endpoints enabled:

```bash
./collector --debug-addr=:6060
```

2. **Check debug endpoints:**

```bash
# Basic health
curl http://localhost:6060/health

# Buffer manager state
curl http://localhost:6060/debug/state/buffer

# pprof interface (open in browser)
open http://localhost:6060/debug/pprof/
```

3. **Generate test traffic:**

```bash
go run cmd/trace-generator/main.go --rate=500 --duration=1m
```

4. **Monitor during load:**

```bash
# Take a CPU profile during load
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=10

# Check memory growth
go tool pprof http://localhost:6060/debug/pprof/heap
```

5. **Verify self-instrumentation:** Check that the collector's own traces appear in the trace query interface.

Expected Results: You should be able to see internal spans from the collector, monitor buffer sizes in real-time, and capture performance profiles under load. If debug endpoints don't respond or show empty data, check that components are properly registered with the `StateDumper`.

G. Debugging Tips Quick Reference

Symptom	Immediate Action	Long-term Solution
Collector OOM	1. Check `BufferManager` size 2. Profile heap with pprof	Implement smarter eviction, add memory limits
High CPU	1. Capture CPU profile 2. Check for hot loops in serialization	Optimize hot paths, add caching, use pooling
Slow queries	1. Check storage indexes 2. Profile query execution	Add query caching, optimize indexes, partition data
Missing traces	1. Verify sampling decisions 2. Check WAL for lost spans	Improve sampling debug logging, enhance WAL durability
Incorrect service map	1. Inspect edge extraction logic 2. Check `ServiceName` extraction	Add validation, improve attribute parsing

Remember that the most effective debugging comes from **understanding normal behavior first**. Establish baselines for key metrics (memory usage, processing latency, trace completion rates) so you can recognize anomalies when they occur.

14. Future Extensions

> **Milestone(s):** This section looks beyond the current five milestones, exploring potential enhancements that could build upon the existing APM Tracing System. These ideas are not part of the current scope but illustrate how the system can evolve to address more complex observability challenges, leverage emerging technologies, and provide deeper insights into distributed system behavior.

Potential Enhancements

The current APM Tracing System provides a solid foundation for distributed tracing, service dependency visualization, intelligent sampling, performance analytics, and automatic instrumentation. However, the world of observability is rapidly evolving. This section explores several natural extensions that could build upon our existing architecture, demonstrating how the system's modular design and clear data flow enable future enhancements without requiring major re-architecting.

14.1 Continuous Profiling Integration

Mental Model: The X-Ray Machine for Code Execution

> While traces show you *what* happened (the request flow) and *when* (timing), profiling shows you *why* (which specific lines of code consumed CPU or memory). Integrating continuous profiling is like adding an X-ray machine to our diagnostic toolkit—it reveals the internal structure and hotspots that surface metrics and traces can only hint at.

Continuous profiling involves periodically capturing CPU flame graphs, memory allocation profiles, and goroutine/thread stacks from production services, then correlating this data with trace information to identify performance bottlenecks at the code level. Our APM system is uniquely positioned to integrate profiling because we already understand service boundaries and can correlate profiles with specific operations and trace contexts.

Integration Approach:

1. **Profile Collection Extension:** Extend the existing APM SDK to include a lightweight profiler that captures stack samples at configurable intervals (e.g., every 10 seconds for 5 seconds).
2. **Profile-Execution Correlation:** Tag each profile sample with the current trace context (when available) so profiles can be filtered to specific operations or services.
3. **Dedicated Profile Storage:** Add a new storage backend optimized for profile data (which is larger and more hierarchical than span data), potentially using columnar formats for efficient querying.
4. **Profile Analysis Engine:** Implement analysis that identifies resource-intensive functions and correlates them with high-latency traces or error conditions.

Required Architecture Changes:

- New `Profiler` component in the SDK with methods `StartProfiling()` and `StopProfiling()`
- Extended `Span` type to optionally include `ProfileID` reference
- New `ProfileStorage` interface with implementations for different backends
- Enhanced Query Service to support profile queries by service, operation, or trace ID

Benefits and Challenges:

Benefit	Challenge
Pinpoint exact code causing latency spikes	Profile data volume is 10-100× larger than trace data
Correlate resource usage with business transactions	Profiling overhead must be minimal (<2% CPU)
Identify memory leaks by tracking heap growth over time	Stack trace symbolization requires debug symbols

Example Workflow: When the anomaly detector flags elevated p99 latency for the `payment-service`'s `ProcessPayment` operation, engineers could query for CPU profiles collected during those high-latency traces, immediately seeing that 40% of CPU time was spent in a specific encryption library function that was recently updated.

14.2 AI-Powered Root Cause Analysis

Mental Model: The Distributed Systems Detective with Machine Learning Intuition

> Our current anomaly detection identifies *when* something is wrong. AI-powered root cause analysis would answer *why* and *what* caused it by learning normal patterns across thousands of dimensions and identifying the most probable culprit when deviations occur—like a seasoned detective who instantly recognizes which clue matters most.

This enhancement would apply machine learning models to the rich correlation data already flowing through our system: service dependencies, latency patterns, error rates, and topology changes. By training models on historical incident data (what changed before an outage), the system could automatically suggest root causes when anomalies are detected.

Integration Approach:

1. **Feature Extraction Pipeline:** Extend the existing analytics engine to compute additional features from trace data: service call graphs, latency distributions across dependency chains, error propagation patterns, and deployment timestamps.
2. **Model Training Infrastructure:** Add a batch processing pipeline that trains models on historical data labeled with known incidents (requires integration with incident management systems).
3. **Real-Time Inference:** Integrate trained models into the anomaly detection pipeline to provide root cause suggestions alongside anomaly alerts.
4. **Feedback Loop:** Allow engineers to validate or correct root cause suggestions, creating labeled data for continuous model improvement.

Implementation Components:

- `RootCauseAnalyzer` interface with `Analyze(anomalies []AnomalyResult, currentGraph ServiceGraph) RootCauseSuggestion`
- `ModelTrainingPipeline` for offline training on historical trace and incident data
- Enhanced `AnomalyResult` type with `SuggestedRootCauses []RootCauseSuggestion` field

Data Flow Enhancement:

```
Anomaly Detector → [New: Root Cause Analyzer] → Alert with suggested root cause
                            ↓
                   [New: Model Store] (periodically updated)
                            ↓
                   [New: Training Pipeline] (runs daily on historical data)
```

Key Algorithms to Consider:

- **Graph-based algorithms** to identify service dependency changes coinciding with anomalies
- **Time-series correlation** to find services whose metrics deviated first
- **Change point detection** in deployment logs to correlate with performance regressions

> **Architecture Decision: Offline vs. Online Learning**
>
> - **Context:** We need to decide whether root cause models should be trained offline (batch) or online (continuously).
> - **Option 1: Offline Batch Training:** Train daily/weekly on historical data.
>   - *Pros:* Simpler infrastructure, easier model validation, predictable resource usage
>   - *Cons:* Latency in adapting to new patterns, requires labeled historical data
> - **Option 2: Online Continuous Learning:** Update models incrementally as new data arrives.
>   - *Pros:* Rapid adaptation to system changes, no manual retraining needed
>   - *Cons:* Complex to implement correctly, risk of model drift, harder to debug
> - **Decision:** Start with offline batch training for initial implementation.
> - **Rationale:** The primary goal is to prove value with simpler infrastructure. Offline training allows for careful validation of model suggestions before deployment. We can transition to online learning once the feature proves valuable and we understand the patterns better.
> - **Consequences:** Root cause suggestions may be slightly stale (up to 24 hours) but this is acceptable for initial implementation.

14.3 eBPF-Based Instrumentation

Mental Model: The Universal System Tap

> eBPF (extended Berkeley Packet Filter) allows us to insert instrumentation directly into the Linux kernel or application runtime without modifying source code—like installing microscopic sensors throughout a building's infrastructure (pipes, electrical, HVAC) that can monitor flow and performance without touching the rooms themselves.

While our current APM SDK requires language-specific instrumentation (Go, Python, Java), eBPF-based instrumentation could provide universal, zero-code-change tracing for any application, including those written in languages we don't yet support or legacy systems where adding an SDK is impractical. This would dramatically expand the observability surface with minimal performance overhead.

Integration Approach:

1. **eBPF Collector Sidecar:** Deploy a lightweight daemon (e.g., using `libbpf` or `bcc` tools) alongside each service that captures system calls, network traffic, and application runtime events.
2. **Span Generation from Kernel Events:** Transform eBPF events into spans, inferring service boundaries from network connections and process relationships.
3. **Hybrid Instrumentation Strategy:** Combine eBPF spans with SDK-generated spans for complete coverage, using consistent trace IDs.
4. **Resource-Aware Sampling:** Apply intelligent sampling at the eBPF level to manage the potentially massive volume of kernel-level events.

Technical Architecture:

```
[Application Process] ← eBPF probes → [eBPF Collector Sidecar] → Spans → [Existing Collector]
       ↓                                        ↓
 [APM SDK spans]                        [Kernel/System spans]
```

eBPF Event Types to Capture:

Event Type	Span Conversion	Information Gained
`sys_enter` / `sys_exit` for `connect`, `accept`, `send`, `recv`	Network call spans	Cross-service calls without SDK
`sys_enter` / `sys_exit` for file I/O, disk operations	Storage I/O spans	Database/disk latency breakdown
Application function entry/exit via USDT probes	Function-level spans	Code path visibility without source modification
TCP retransmissions, congestion window changes	Network quality spans	Infrastructure-level issues affecting application

Required New Components:

- `eBPFCollector` implementing `SpanExporter` interface
- `eBPFSpanGenerator` converting eBPF events to `Span` objects
- `KernelEventBuffer` with ring buffer for high-volume event handling

Performance Considerations:

- eBPF programs run in the kernel with JIT compilation for near-native speed
- Event filtering at the kernel level reduces userspace overhead
- Still requires careful sampling for high-throughput systems

14.4 Real User Monitoring (RUM) Integration

Mental Model: The Frontend Flight Recorder

> While our current system monitors backend services, Real User Monitoring captures the actual user experience in browsers and mobile apps—like adding cockpit voice recorders and control inputs to our existing flight data recorders, giving us the complete picture from pilot intent to aircraft response.

RUM extends tracing to the client side, capturing browser performance metrics (First Contentful Paint, Largest Contentful Paint), JavaScript errors, and user interaction timing. Integrating RUM with our existing backend traces creates end-to-end visibility from user click to database query and back.

Integration Strategy:

1. **Browser SDK Extension:** Develop a lightweight JavaScript library that instruments page load, resource timing, and user interactions.
2. **Trace Context Propagation:** Extend W3C Trace Context to include frontend-originated trace IDs, ensuring seamless correlation with backend traces.
3. **Session-Based Analysis:** Group RUM events by user session to understand complete user journeys.
4. **Synthetic Monitoring Baseline:** Compare real user data with synthetic tests to distinguish network issues from application problems.

Data Model Extensions:

```go
// Extended Span type for RUM-specific data
type RUMSpan struct {
    models.Span                  // Embedded base span
    BrowserInfo      BrowserInfo  // Browser version, viewport size
    PageURL          string       // URL where span occurred
    UserInteraction  string       // "click", "scroll", "input"
    WebVitals        WebVitals    // Core Web Vitals metrics
}

type WebVitals struct {
    FCP    time.Duration  // First Contentful Paint
    LCP    time.Duration  // Largest Contentful Paint
    FID    time.Duration  // First Input Delay
    CLS    float64        // Cumulative Layout Shift
}
```

Correlation Enhancement: When a user reports "the checkout page is slow," engineers could:

1. Query RUM spans for the checkout page to see if it's a frontend issue (slow LCP)
2. Follow the trace ID to see backend processing time
3. Identify whether the bottleneck is in JavaScript execution, network latency, or service processing

14.5 Business Transaction Tracing

Mental Model: The Business Process Map Overlay

> While technical traces show service calls, business transaction tracing overlays business semantics (customer journeys, order flows, payment processing) onto the technical infrastructure—like adding labeled "shipping routes" and "warehouse operations" to a map of roads and buildings.

This enhancement allows tagging traces with business context (user ID, order value, transaction type) and defining key business transactions (e.g., "new user signup," "premium checkout") that can be monitored for SLAs, error rates, and performance against business objectives.

Implementation Approach:

1. **Business Context Propagation:** Extend trace context to include business key-value pairs that flow with the request.
2. **Transaction Definition DSL:** Create a domain-specific language for defining business transactions as patterns of spans with specific attributes.
3. **Business Metrics Aggregation:** Extend the analytics engine to compute metrics per business transaction rather than just per service/operation.
4. **Business SLA Monitoring:** Alert when business transactions violate defined SLAs (e.g., "gold-tier checkout must complete under 2 seconds 99% of the time").

Example Business Transaction Definition:

```yaml
business_transactions:
  - name: "premium_checkout"
    description: "Checkout flow for premium users"
    match:
      - span.service: "checkout-service"
        span.operation: "process_payment"
        span.attributes.user_tier: "premium"
      - span.service: "inventory-service"
        span.operation: "reserve_item"
    sla:
      p99_latency: "2s"
      error_rate: "0.1%"
```

Required System Changes:

- Enhanced `Span` type with `BusinessAttributes map[string]string`
- New `BusinessTransactionMonitor` component
- Extended Query Service API for querying traces by business attributes
- New visualization layer showing business transaction flow across services

14.6 Trace Data Enrichment with Logs and Metrics

Mental Model: The Unified Observability Timeline

> Currently, traces, logs, and metrics exist in separate silos. Enrichment creates bidirectional links between them—like adding cross-references between a detective's case notes (logs), crime scene photos (metrics), and witness interview transcripts (traces), creating a unified case file.

This enhancement would correlate trace data with log entries and infrastructure metrics, allowing engineers to pivot from a slow trace to the corresponding application logs (to see error messages) and system metrics (to see CPU spikes) from the same time period.

Integration Architecture:

1. **Unified Storage Backend:** Extend storage to support logs and metrics alongside traces, or implement connectors to existing log/metric systems.
2. **Correlation Index:** Create an index by timestamp and service to quickly find related telemetry data.
3. **Query Language Enhancement:** Extend the query language to support joins across trace, log, and metric data.
4. **Unified Visualization:** Create a single UI that shows traces alongside related logs and metrics.

Correlation Strategies:

Correlation Method	Implementation	Use Case
Timestamp Proximity	Find logs/metrics within ±100ms of span timestamps	General debugging
Trace Context Injection	Inject trace ID into log messages (via structured logging)	Precise log-to-trace linking
Resource Attributes	Match by service name, host, container ID	Infrastructure issue correlation

Example Enhancement to Storage Layer:

```go
// Extended storage interface supporting multi-signal queries
type UnifiedStorage interface {
    GetTraceWithContext(ctx context.Context, traceID string) (*TraceWithContext, error)
}

type TraceWithContext struct {
    Trace   *models.Trace
    Logs    []LogEntry          // Related log entries
    Metrics []MetricSample      // Related metric samples
    Events  []InfrastructureEvent // Deployment, scaling events
}
```

14.7 Multi-Tenancy and Data Isolation

Mental Model: The Observability Apartment Building

> Currently, our system treats all data as belonging to a single organization. Multi-tenancy adds separate "apartments" (tenants) with soundproof walls and separate entrances—each tenant's data is fully isolated, but they share the same building infrastructure (collectors, storage, query engines).

This enhancement would enable serving multiple independent organizations or internal teams from a single APM deployment, with strict data isolation, tenant-specific configurations, and usage-based billing.

Implementation Considerations:

1. **Tenant Identification:** Add tenant ID to all spans, either via authentication tokens or request headers.
2. **Storage Isolation:** Implement storage backends that support tenant partitioning at the data layer.
3. **Query Isolation:** Extend all query APIs to filter by tenant ID, enforced at the API gateway.
4. **Rate Limiting and Quotas:** Apply tenant-specific limits on ingestion rate, storage usage, and query frequency.
5. **Administrative UI:** Add tenant management interfaces for provisioning, configuration, and usage monitoring.

Data Flow Changes:

```
[Tenant A App] → [Collector] → [Storage with Tenant Partitioning]
[Tenant B App] ↗        ↓
           [Query Service with Tenant Filtering] → [Tenant-Specific UI]
```

Security Considerations:

- Tenant isolation must be enforced at every layer (API, storage, cache)
- No cross-tenant data leakage in aggregations or sampling decisions
- Audit logging of all tenant data access

14.8 Advanced Visualization: Flame Graphs and Heat Maps

Mental Model: The Performance Topographical Map

> While our current service map shows service relationships, flame graphs show time distribution within a service, and heat maps show request density patterns—like adding elevation contours (flame graphs) and population density shading (heat maps) to a geographical map, revealing not just locations but intensity and distribution.

These visualization enhancements would help engineers quickly identify hotspots in their code (flame graphs) and understand temporal patterns of request flow (heat maps).

Flame Graph Implementation:

1. **Span Stack Construction:** Convert parent-child span relationships into call stacks for flame graph rendering.
2. **Time-Aggregated Views:** Aggregate spans from multiple traces to show typical execution patterns.
3. **Interactive Exploration:** Allow drilling into specific functions or time ranges.

Heat Map Implementation:

1. **Request Density Calculation:** Compute requests per second across services and time buckets.
2. **Latency Coloring:** Color cells by average latency or error rate.
3. **Anomaly Highlighting:** Visually flag time periods with anomalies.

Visualization Integration:

- Extend the existing Web UI with new visualization components
- Add new API endpoints for flame graph and heat map data
- Support exporting visualizations for incident reports

14.9 Predictive Capacity Planning

Mental Model: The Observability Crystal Ball

> While current analytics tell us what *is* happening and what *did* happen, predictive capacity planning forecasts what *will* happen—like using weather radar patterns to predict storm paths and intensities hours in advance.

This enhancement would apply time-series forecasting algorithms to historical performance data to predict future resource needs, identify capacity bottlenecks before they cause outages, and recommend scaling actions.

Forecasting Algorithms:

- **ARIMA/SARIMA** for seasonal patterns
- **Prophet** for handling holidays and changepoints
- **LSTM neural networks** for complex multi-variate patterns

Integration Points:

1. **Forecasting Pipeline:** Extend the analytics engine with forecasting modules that run periodically.
2. **Capacity Recommendations:** Generate actionable recommendations (e.g., "Increase payment-service instances by 2 before Black Friday").
3. **What-If Analysis:** Simulate the impact of traffic increases or infrastructure changes.

Example Output:

```json
{
  "service": "payment-service",
  "metric": "p95_latency",
  "current_value": "145ms",
  "forecast": {
    "1_hour": "152ms (±8ms)",
    "24_hours": "210ms (±25ms)",
    "7_days": "Will exceed 500ms SLA in 4.3 days at current growth rate"
  },
  "recommendation": "Add 2 instances within 48 hours"
}
```

14.10 Enhancement Prioritization Framework

To help prioritize which enhancements to implement first, consider this decision matrix:

Enhancement	Business Value	Implementation Complexity	Data Volume Impact	User Impact
Continuous Profiling	High (pinpoints code issues)	Medium (new data type)	High (large profiles)	Medium (developers)
AI Root Cause Analysis	Very High (reduces MTTR)	High (ML expertise)	Low (metadata only)	High (on-call engineers)
eBPF Instrumentation	High (universal coverage)	Very High (kernel expertise)	Very High (raw events)	Medium (platform teams)
RUM Integration	High (end-to-end visibility)	Medium (new SDK)	Medium (browser data)	High (frontend teams)
Business Transaction Tracing	High (aligns with business)	Low (metadata addition)	Low (small attributes)	High (product managers)
Multi-Tenancy	Medium (enables SaaS)	High (security critical)	Medium (partitioning)	Medium (operations)

> **Key Insight:** The most valuable enhancements often build upon existing capabilities with minimal disruption. Business Transaction Tracing and RUM Integration offer high value with moderate complexity, making them excellent candidates for near-term implementation.

Implementation Guidance for Future Extensions

A. Technology Recommendations for Extensions

Extension	Core Technology	Integration Point
Continuous Profiling	**pprof** (Go), **py-spy** (Python), **async-profiler** (Java)	Extend `Tracer` to optionally capture profiles
AI Root Cause Analysis	**scikit-learn**, **TensorFlow**, or **H2O.ai** for ML; **Jupyter** for analysis	New `RootCauseAnalyzer` component in analytics layer
eBPF Instrumentation	**libbpf** (C library), **bcc** (Python tools), **cilium/ebpf** (Go library)	New `eBPFCollector` sidecar process
RUM Integration	**Web Performance API**, **Error Tracking API**	New JavaScript SDK extending W3C Trace Context
Business Transaction Tracing	**YAML/JSON DSL** for definitions	Extend `Span` attributes and add `BusinessTransactionMonitor`
Multi-Tenancy	**JWT/OAuth2** for auth, **PostgreSQL Row Security** or storage partitioning	Tenant ID propagation throughout data pipeline

B. Recommended File Structure for Extensible Design

To accommodate future extensions without major refactoring, organize the codebase with clear interfaces and plugin points:

```
project-root/
  cmd/
    collector/           # Main collector binary
    query-service/       # Query service binary
    web-ui/             # Web UI server
  internal/
    # Core components (existing)
    ingestion/          # Pipeline, validation, buffering
    storage/            # Span storage interfaces and implementations
    sampling/           # Head and tail sampling
    analytics/          # Percentiles, anomaly detection
    sdk/                # Auto-instrumentation libraries

    # Extension points (new directories)
    extensions/         # Interfaces and registries for extensions
      profiler/         # Profiling extension interfaces
      rum/              # RUM extension interfaces
      ebpf/             # eBPF collection interfaces
      business/         # Business transaction interfaces

    # Implementation of specific extensions (when built)
    extensions/impl/
      profiler-pprof/   # pprof-based profiler
      rum-js/           # JavaScript RUM SDK
      ebpf-collector/   # eBPF event collector

  pkg/
    api/                # Public APIs for external integration
    models/             # Core data types (Span, Trace, etc.)
    plugin/             # Plugin framework interfaces
```

C. Extension Point Interfaces

To ensure the system can be extended without modification to core components, define these key interfaces:

GO

```go
// Extension point for adding new telemetry sources
type TelemetrySource interface {
    Name() string
    Start(ctx context.Context) error
    Stop() error
    TelemetryChan() <-chan models.Span
}

// Extension point for adding new analysis modules
type AnalysisModule interface {
    Name() string
    Analyze(ctx context.Context, trace *models.Trace) (AnalysisResult, error)
    RequiredFields() []string // Span fields needed for analysis
}

// Extension point for adding new visualizations
type VisualizationProvider interface {
    Name() string
    Type() string // "flamegraph", "heatmap", etc.
    DataQuery(ctx context.Context, params map[string]string) ([]byte, error)
    RenderOptions() map[string]interface{}
}

// Registry for extensions (simplified example)
type ExtensionRegistry struct {
    mu          sync.RWMutex
    sources     map[string]TelemetrySource
    analyzers   map[string]AnalysisModule
    visualizers map[string]VisualizationProvider
}

func (r *ExtensionRegistry) RegisterSource(source TelemetrySource) {
    r.mu.Lock()
    defer r.mu.Unlock()
    r.sources[source.Name()] = source
```

```
}

// Initialize extensions in main()
func main() {
    registry := NewExtensionRegistry()

    // Register built-in extensions based on configuration
    if config.EnableProfiling {
        registry.RegisterSource(NewPProfProfiler(config.Profiling))
    }

    if config.EnableRUM {
        registry.RegisterSource(NewRUMSource(config.RUM))
    }

    // Start all registered extensions
    for _, source := range registry.Sources() {
        go source.Start(ctx)
    }
}
```

D. Design for Extension: Span Enrichment Pattern

Many extensions (business tracing, multi-tenancy, log correlation) need to add metadata to spans. Instead of modifying the core `Span` type for each extension, use an extensible attributes pattern:

```go
// Current Span type already has Attributes map[string]string
// Use a namespacing convention for extension attributes:
// "extension.attribute_name" → "value"

// Business transaction attributes
span.Attributes["business.transaction"] = "premium_checkout"
span.Attributes["business.customer_tier"] = "gold"

// RUM attributes
span.Attributes["rum.browser"] = "Chrome 91"
span.Attributes["rum.page_url"] = "https://example.com/checkout"

// Multi-tenancy attributes
span.Attributes["tenant.id"] = "acme-corp"
span.Attributes["tenant.environment"] = "production"

// Helper functions for extension authors
func AddBusinessContext(span *models.Span, transaction string, attributes map[string]string) {
    span.Attributes["business.transaction"] = transaction
    for k, v := range attributes {
        span.Attributes["business."+k] = v
    }
}

// Query support for extension attributes
func GetTracesByBusinessTransaction(ctx context.Context, transaction string) ([]*models.Trace, error) {
    // Storage layer should index extension.* attributes for efficient querying
    return storage.GetTracesByAttribute(ctx, "business.transaction", transaction)
}
```

E. Migration Strategy for Extensions

When implementing extensions that change data schemas or require new storage formats:

1. **Versioned Data Formats:** Always include a version field in stored data.
2. **Dual-Write During Migration:** Write both old and new formats during transition.
3. **Backward Compatibility:** Ensure queries work with old data formats.
4. **Gradual Rollout:** Enable extensions per-service or per-tenant initially.

```go
// Example: Adding profile storage
type StorageV1 struct { /* existing span storage */ }

type StorageV2 struct {
    StorageV1                    // embedded for backward compatibility
    profileStorage ProfileStorage // new capability
}

// During migration period
if config.UseProfilingExtension {
    // Store profiles in new storage
    storageV2.profileStorage.StoreProfile(profile)
}
// Always store spans in V1-compatible format
storageV1.StoreSpan(span)
```

F. Testing Extension Points

Create test helpers for extension developers:

```go
// Extension test suite
func TestExtensionContract(t *testing.T) {
    // Verify extension implements required interface
    var _ extensions.TelemetrySource = &MyExtension{}

    // Test integration with core pipeline
    pipeline := ingestion.NewTestPipeline()
    extension := NewMyExtension()

    // Start extension
    err := extension.Start(context.Background())
    require.NoError(t, err)

    // Verify it produces valid spans
    select {
    case span := <-extension.TelemetryChan():
        assert.NotEmpty(t, span.TraceID)
        assert.NotEmpty(t, span.ServiceName)
    case <-time.After(1 * time.Second):
        t.Fatal("Extension didn't produce telemetry within 1 second")
    }
}
```

G. Performance Monitoring for Extensions

Extensions should include their own performance monitoring:

```go
type ExtensionMetrics struct {
    SpansGenerated   prometheus.Counter
    ProcessingTime   prometheus.Histogram
    Errors           prometheus.Counter
    QueueLength      prometheus.Gauge
}

// Register with central metrics registry
func (e *MyExtension) registerMetrics() {
    metrics.ExtensionMetrics.WithLabelValues(e.Name()).Inc()
}
```

> **Key Implementation Insight:** The most important design principle for future extensions is **dependency inversion**—core components should depend on abstractions (interfaces) that extensions implement, not the other way around. This allows adding new capabilities without modifying tested core code.

By designing with these extension points in mind, the APM Tracing System can evolve gracefully to incorporate new observability capabilities as they emerge, ensuring the system remains valuable and relevant as technology and requirements change.

> **Milestone(s):** This section provides a comprehensive reference for terminology used throughout all five milestones of the APM Tracing System, ensuring consistent understanding across the design document.

15. Glossary

As you navigate the design of the APM Tracing System, you'll encounter specialized terms that carry precise technical meanings. This glossary serves as a definitive reference, clarifying terminology used across all system components. Think of it as the **legend for a complex map**—it doesn't change the territory, but makes it readable. Having a shared vocabulary is essential for clear communication among team members and for understanding the architectural decisions documented throughout this design.

Terminology Reference

The following table defines key terms used in this design document, organized alphabetically for quick reference.

Term	Definition	Context & Significance
Adaptive Sampling	A sampling strategy that dynamically adjusts sampling rates based on system load, trace value, or other real-time metrics.	Used in **Milestone 3** to balance data volume with insight preservation. Unlike fixed-rate sampling, adaptive sampling responds to changing conditions—for example, lowering rates during traffic surges to protect storage, or increasing rates when error rates spike to capture more debugging data.
Anomaly	A data point or pattern in metrics that deviates significantly from expected, historical behavior.	The target of detection in **Milestone 4**. Anomalies in APM systems typically manifest as sudden latency spikes, error rate increases, or throughput drops. Distinguishing true anomalies from expected variations (like daily traffic patterns) is a core challenge.
APM (Application Performance Monitoring)	The practice and technology for monitoring software applications for performance, availability, and user experience.	The overarching domain of this project. APM systems combine distributed tracing, metrics, and sometimes logs to provide holistic visibility into application behavior. Our system focuses specifically on the tracing pillar.
Backpressure	A flow control mechanism where a downstream component signals upstream components to slow down or stop sending data when it cannot keep up.	A critical resilience concept discussed in **Section 11**. When the collector is overwhelmed, it must apply backpressure (e.g., via aggressive sampling or connection refusal) to prevent cascading failures and data loss.
Baseline	Historical performance data (e.g., latency percentiles, error rates) used as a reference for comparison when detecting anomalies.	Established in **Milestone 4** through analysis of past metrics. Baselines can be simple (like a rolling average) or sophisticated (accounting for weekly seasonality). Comparing current metrics against baselines allows the system to identify deviations that warrant investigation.
Buffer Eviction	The process of removing items from an in-memory buffer when it reaches capacity or when items have exceeded their time-to-live (TTL).	A key mechanism in **Milestone 1's** `BufferManager`. Since we cannot buffer spans indefinitely while waiting for late-arriving spans, we must evict old, incomplete traces. The eviction policy (e.g., LRU, based on first-seen time) directly impacts trace completeness.
Business Transaction Tracing	Tagging traces with business context (e.g., customer ID, transaction type) to enable monitoring and analysis of business processes rather than just technical operations.	A **Future Extension** from Section 14. This allows correlating performance issues with business impact—for example, seeing that checkout latency increases are affecting premium customers.
Cascading Failures	A failure mode where the failure of one component causes failures in dependent components, potentially leading to system-wide outage.	A risk mitigated by patterns like circuit breakers and backpressure. In our APM system, if the storage backend becomes slow, the collector could exhaust memory buffering spans, then become unresponsive to new spans, causing instrumentation SDKs to block or drop data.
Chaos Testing	A testing methodology that deliberately injects failures (network partitions, process kills, latency spikes) into a system to verify its resilience.	Part of the **Testing Strategy** in Section 12. For our APM system, chaos testing might involve randomly killing collector instances or introducing high latency to storage to verify that sampling degrades gracefully and data isn't lost.
Circuit Breaker Pattern	A design pattern that prevents calls to a failing service by opening a "circuit" after failure thresholds are exceeded, failing fast instead of waiting for timeouts.	Used in **Section 11** for protecting interactions between components (e.g., collector to storage). The `CircuitBreaker` struct implements states (`StateClosed`, `StateOpen`, `StateHalfOpen`) to prevent overloading already-struggling dependencies.
Clock Skew	The difference in system clock times between different machines in a	A **Common Pitfall** in **Milestone 1**. Since spans arrive from multiple services with potentially unsynchronized clocks, relying solely on span

Term	Definition	Context & Significance
	distributed system.	timestamps for ordering can produce incorrect trace sequences. The system must handle or normalize for clock skew.
Columnar Storage	A storage format that organizes data by columns rather than rows, optimizing for analytical queries that aggregate over specific fields.	Mentioned as an optimization for trace storage. Storing all `Duration` values contiguously enables faster percentile calculations across many traces, compared to row-oriented storage that would load entire spans.
Consistent Hashing	A hash function that ensures the same input (e.g., trace ID) always maps to the same output, enabling deterministic decisions across distributed components.	Critical for **head-based sampling** in **Milestone 3**. Using `ConsistentSampler.ShouldSample(traceID)` with a hash function guarantees that all spans from the same trace receive the same sampling decision, even if they arrive at different collectors.
Context Propagation	The mechanism for carrying trace context (trace ID, span ID, sampling decision) across process and service boundaries, typically via HTTP headers or RPC metadata.	A core concept in **Milestone 5**. The SDK's `InjectTraceContext` and `ExtractTraceContext` methods implement the W3C Trace Context standard, allowing traces to continue seamlessly across service calls.
Continuous Profiling	The practice of periodically capturing and analyzing application profiles (CPU, memory allocation) to identify performance bottlenecks over time.	A **Future Extension** that could integrate with the APM system. While tracing shows *what* happened and *when*, profiling shows *why* certain code paths are slow by revealing hot functions and allocation patterns.
Cycle Detection	An algorithm to identify circular dependencies in service calls (e.g., Service A → Service B → Service A), which can indicate architectural issues or cause infinite loops.	Part of **service map construction** in **Milestone 2**. The `ServiceGraph` should identify cycles to alert developers about potentially problematic dependency patterns that could lead to cascading failures.
Dictionary Encoding	A compression technique that replaces repeated string values (like service names or operation names) with integer IDs in storage, reducing disk usage and improving query performance.	An optimization for trace storage. Since `ServiceName` and `Span.Name` fields often have high repetition, dictionary encoding can significantly reduce the storage footprint of the `Attributes` map and other string-heavy fields.
Distributed Tracing	A method of tracking requests as they propagate through a distributed system, recording timing data and metadata at each step (span) to form a complete timeline (trace).	The foundational technique of this entire project. It addresses the observability challenge in microservices by providing request-centric visibility, as opposed to metric- or log-centric views.
Edge Aggregation	The process of combining multiple individual service call observations (spans) into summarized metrics (call count, error rate, latency percentiles) for a specific caller-callee pair over a time window.	The core operation in **Milestone 2's** `EdgeAggregator`. Instead of storing every call, we aggregate spans into `ServiceEdge` records that represent the communication pattern between two services during a specific time window.
eBPF (Extended Berkeley Packet Filter)	A technology for running sandboxed programs in the Linux kernel, enabling observability, networking, and security functionality without modifying kernel source code or loading kernel modules.	A **Future Extension** for zero-instrumentation tracing. An `eBPFCollector` could automatically trace system calls, network traffic, or application functions without requiring code changes, complementing the SDK-based instrumentation.

Term	Definition	Context & Significance
Flame Graph	A visualization of hierarchical data, typically call stacks, where the width of each element represents its resource consumption (CPU time, memory allocations).	A potential visualization for performance analytics. While not part of the core milestones, flame graphs could be generated from trace data to show which code paths contribute most to latency.
Graceful Degradation	A design principle where a system continues to operate with reduced functionality rather than failing completely when under stress or partial failure.	A guiding principle for **Error Handling**. For example, when storage is unavailable, the collector might switch to a "degraded mode" where it samples traces more aggressively but still provides some visibility, rather than rejecting all spans.
Head-Based Sampling	A sampling strategy where the decision to keep or drop a trace is made at the very beginning of the trace (when the root span is created), using only initial context (trace ID, service name).	Implemented by the `HeadSampler` in **Milestone 3**. This is efficient (no buffering required) but risks dropping interesting traces that only become "interesting" later (e.g., due to an error in a downstream service).
Head-of-Line Blocking	A performance issue where processing of one item is delayed, causing all subsequent items in the queue to wait, even if they could be processed independently.	A potential pitfall in the ingestion pipeline. If one malformed span requires expensive validation or blocking I/O, it shouldn't stall the processing of all other spans. The pipeline design uses buffered channels and parallel processing to avoid this.
Health Check	An endpoint or probe that reports the operational status of a component or system, typically used by load balancers and orchestration systems to determine if an instance should receive traffic.	Implemented via `HealthRegistry` and its handlers (`HealthHandler`, `LivenessHandler`, `ReadinessHandler`) in **Section 11**. The health check aggregates status from all components (collector, buffer manager, storage connection) to give a holistic view.
Infrastructure Metrics	System-level measurements like CPU utilization, memory usage, disk I/O, and network bandwidth, as opposed to application-level metrics (request latency, error rates).	A **Non-Goal** of this APM system (we focus on application traces), but mentioned as a complementary observability signal. In a unified observability platform, infrastructure metrics would correlate with trace data to identify root causes.
Integration Test	A test that verifies multiple components work together correctly by testing their interactions through real or simulated interfaces.	Part of the **Testing Strategy**. For example, an integration test might start a collector, send spans via the OTLP endpoint, and verify they appear in storage and affect the service map.
Load Test	A test that measures system performance (throughput, latency, resource usage) under expected or peak load conditions.	Essential for verifying **Milestone 1** acceptance criteria (handling 1000 spans/second). Load tests simulate production traffic patterns to identify bottlenecks and validate scaling assumptions.
Log Aggregation	The practice of centralized collection, indexing, and search of unstructured log data from multiple services.	A **Non-Goal**—our system focuses on structured trace data, not unstructured logs. However, traces can be correlated with logs via trace IDs injected into log messages, creating a unified debugging experience.
Liveness Probe	A type of health check that indicates whether a process is running (alive), without guaranteeing it's ready to handle work.	Simpler than a readiness check. The `LivenessHandler` typically returns success as long as the process hasn't crashed. Kubernetes uses this to decide when to restart a container.

Term	Definition	Context & Significance
Microservices	An architectural style that structures an application as a collection of loosely coupled, independently deployable services that communicate via APIs.	The primary architectural context for distributed tracing. Microservices introduce the observability challenge that tracing solves: a single user request may traverse dozens of services, making debugging without traces nearly impossible.
Middleware	A software layer that intercepts requests and responses in a framework's processing pipeline, enabling cross-cutting concerns like logging, authentication, and—in our case—tracing.	The primary instrumentation technique in **Milestone 5** for web frameworks. The SDK provides middleware for Gin (Go), Express (Node.js), and Flask (Python) that automatically creates spans for incoming HTTP requests.
Mock	A simulated object that mimics the behavior of a real component in controlled ways for testing purposes.	Used extensively in unit tests. For example, we mock the `storage.Writer` interface to test the collector without needing a real database, verifying that `StoreSpan` is called with expected parameters.
Monkey Patching	A technique to modify or extend the behavior of a library at runtime by replacing its functions or methods.	One method for **auto-instrumentation** in **Milestone 5**, particularly in dynamic languages like Python. The SDK might monkey-patch the `requests` library to automatically inject trace headers into all HTTP requests. Must be done carefully to avoid breaking original behavior.
Moving Average	A calculation to analyze data points by creating a series of averages of different subsets of the full data set, often used to smooth short-term fluctuations and highlight longer-term trends.	One statistical method for **anomaly detection** in **Milestone 4**. By comparing current latency to a moving average of recent latencies, we can detect deviations from recent norms.
Multi-Tenancy	An architectural pattern where a single deployment of software serves multiple independent customers (tenants), with data and configuration isolated between them.	A **Future Extension** consideration. The APM system could be enhanced to support multiple teams or external customers, requiring isolation of trace data, sampling configurations, and service maps per tenant.
OpenTelemetry	A vendor-neutral, open-source collection of APIs, SDKs, and tools for instrumenting, generating, collecting, and exporting telemetry data (traces, metrics, logs).	The standard we adopt for trace format and context propagation. Our collector accepts spans in **OpenTelemetry Protocol (OTLP)** format via the `OTLPHandler`, ensuring compatibility with a wide ecosystem of instrumentation libraries.
OpenTelemetry Protocol (OTLP)	The standard protocol defined by OpenTelemetry for telemetry data exchange between clients and collectors, supporting both gRPC and HTTP/JSON transports.	The wire format for span ingestion. The `parseOTLPJSON` and `parseOTLPProtobuf` methods convert OTLP payloads into our internal `Span` representation, ensuring we can receive data from any OpenTelemetry-compliant SDK.
Percentile	A value below which a given percentage of observations in a group of observations falls. For example, the 95th percentile (p95) latency is the value below which 95% of observed latencies fall.	A core metric in **Milestone 4**. Percentiles (p50, p95, p99) are more informative than averages for performance analysis because they reveal tail latency behavior that affects user experience.
PID Controller	A control loop mechanism that uses proportional, integral, and derivative	One possible algorithm for **adaptive sampling** in **Milestone 3**. A PID controller could adjust sampling rates to maintain a target span ingestion

Term	Definition	Context & Significance
	terms to calculate an error value as the difference between a desired setpoint and a measured process variable.	rate, smoothly responding to traffic changes without oscillation.
pipes and filters	An architectural pattern where data flows through a sequence of processing components (filters), each performing a specific transformation, connected by pipes (channels, queues).	The pattern used in the ingestion `Pipeline`. Spans flow through: validator → buffer manager → head sampler → (later) tail sampler → storage writer. This design promotes separation of concerns and easy component replacement.
Predictive Capacity Planning	Using historical performance data and trends to forecast future resource needs (CPU, memory, storage) for a system.	A **Future Extension** leveraging the analytics engine. By analyzing trace volume growth and latency trends, the system could predict when storage capacity will be exhausted or when additional collector instances will be needed.
Property-Based Testing	A testing methodology that verifies properties or invariants hold for a large number of randomly generated inputs, rather than testing specific examples.	Useful for testing complex logic like trace assembly. For example, a property-based test could generate random spans with valid parent-child relationships and verify that `TraceAssembler.AssembleTrace` always produces a properly structured trace hierarchy.
Readiness Probe	A type of health check that indicates whether a component is ready to handle work (e.g., dependencies connected, caches warmed).	More comprehensive than a liveness probe. The `ReadinessHandler` might check that the collector can connect to storage and that the buffer manager is not overloaded. Kubernetes uses this to control when to send traffic to a pod.
Real User Monitoring (RUM)	Instrumenting client-side web applications to capture browser performance metrics, user interactions, and frontend errors experienced by real users.	A **Non-Goal**—our system focuses on server-side tracing. However, the data model includes `RUMSpan` as a potential extension, showing how client-side spans could integrate with server-side traces for full-stack visibility.
Ring Buffer	A circular buffer data structure that uses a single, fixed-size buffer as if it were connected end-to-end, providing efficient FIFO semantics for streaming data.	Used in the `MemoryStore` for metric aggregation in **Milestone 4**. Ring buffers naturally evict old data as new data arrives, making them ideal for sliding window calculations without explicit cleanup logic.
Seasonal Decomposition	A time series analysis method that separates data into trend, seasonal (periodic), and residual components, useful for detecting anomalies in data with regular patterns (like daily or weekly cycles).	An advanced technique for **baseline calculation** in **Milestone 4**. By accounting for regular patterns (e.g., lower traffic on weekends), seasonal decomposition reduces false positives in anomaly detection.
Service	A logical component in a distributed system that emits spans. In our data model, a `Service` has a name and a list of operations (endpoints) it performs.	A fundamental entity in service maps and sampling configuration. Services are extracted from spans' `ServiceName` field, and sampling rates can be configured per service via `SamplingConfig.PerService`.
Service Map	A visual representation of service dependencies in a distributed system, typically shown as a directed graph where nodes are services and edges represent calls between them, annotated with metrics.	The primary deliverable of **Milestone 2**. The `ServiceGraph` is constructed from trace data and provides an at-a-glance view of system topology and health, helping engineers understand communication patterns and identify bottlenecks.

Term	Definition	Context & Significance
Span	A named, timed operation representing a unit of work within a trace. Spans have a parent-child relationship structure and contain timing data, attributes, and status. Defined as the `Span` struct with fields like `SpanID`, `TraceID`, `ParentSpanID`, `Name`, `ServiceName`, `StartTime`, `Duration`, etc.	The atomic unit of tracing data. Everything in the APM system revolves around spans—they are ingested, buffered, sampled, stored, and analyzed. A trace is composed of one or more spans.
SpanExporter	A component in the SDK that sends completed spans to the collector, typically batching them for efficiency.	Part of the SDK architecture in **Milestone 5**. The exporter handles the network communication, retry logic, and backoff, allowing the instrumentation code to be non-blocking.
Synthetic Monitoring	Proactive simulation of user traffic (e.g., via scripted browser sessions or API calls) to measure availability and performance from predefined locations.	A **Non-Goal** but mentioned as complementary to tracing. Synthetic traces could be generated by monitoring scripts and would flow through the same collection pipeline, providing baseline performance measurements.
t-digest	A probabilistic data structure for computing approximate percentiles from streaming data with high accuracy, especially in the tails of the distribution, using limited memory.	The algorithm used in `TDigestMetric` for **percentile calculation** in **Milestone 4**. t-digest allows us to maintain rolling percentiles across millions of spans without storing every individual latency value.
Tail-Based Sampling	A sampling strategy where the decision to keep or drop a trace is made after the trace is complete (or nearly complete), allowing evaluation based on the full trace content (errors, latency, patterns).	Implemented by the `TailSampler` in **Milestone 3**. This approach can override head-based sampling decisions to retain "interesting" traces (e.g., those with errors or high latency) even if they were initially sampled out, but requires buffering traces until completion.
TextMapCarrier	An interface for reading and writing trace context to arbitrary carrier formats (HTTP headers, gRPC metadata, message queue properties) during context propagation.	Used by the SDK's `InjectTraceContext` and `ExtractTraceContext` methods in **Milestone 5**. The carrier abstraction allows the same propagation logic to work across different transport protocols.
Time Series Database (TSDB)	A database optimized for storing and retrieving time-stamped data, typically supporting efficient range queries and aggregations over time intervals.	Considered for storing aggregated metrics in **Milestone 4**. While our design uses an in-memory `MemoryStore` for simplicity, a production system might offload historical metrics to a dedicated TSDB like Prometheus or InfluxDB for long-term retention.
Time Window	A fixed period (e.g., 1 minute, 5 minutes) over which metrics are aggregated or analyzed. Windows can be tumbling (non-overlapping) or sliding (overlapping).	Used throughout the system: for edge aggregation in service maps (`EdgeAggregator.windowSize`), for percentile calculations (`PercentileAggregator.window`), and for anomaly detection baselines. Windows balance temporal granularity with statistical significance.
Topology Change	The addition, removal, or modification of service dependencies in the system, detected by comparing service graphs across time windows.	A key insight from the service map. The `DetectTopologyChanges` function compares current and previous `ServiceGraph` instances to identify new services, removed dependencies, or changed communication patterns, which can indicate deployments or issues.

Term	Definition	Context & Significance
Trace	A collection of spans that represent a single request's path through a distributed system, sharing a common `TraceID`. Defined as the `Trace` struct with fields `TraceID`, `Spans`, `StartTime`, and `EndTime`.	The unit of analysis for debugging. While spans are collected individually, they are assembled into traces for visualization and analysis. A trace shows the complete lifecycle of a request, including parallel and sequential operations across services.
Unified Observability	The practice of correlating traces, logs, and metrics for complete system understanding, allowing engineers to move seamlessly between signals during debugging.	A **Future Extension** vision. The `TraceWithContext` struct shows how trace data could be enriched with related logs and metrics, while the `UnifiedStorage` interface suggests a backend that stores all telemetry types.
W3C Trace Context	A W3C standard specification for HTTP headers (`traceparent`, `tracestate`) that propagate trace context across service boundaries, ensuring interoperability between different tracing systems.	The propagation standard adopted by OpenTelemetry and implemented in our SDK. The `TRACE_ID_HEADER` and `SPAN_ID_HEADER` constants correspond to these standard headers, enabling traces to cross service boundaries even when services use different tracing implementations.
Watermark Algorithm	An algorithm that marks a point in a data stream after which no earlier data is expected, used for handling out-of-order data in streaming systems.	Relevant for **trace assembly** in **Milestone 1**. Since spans arrive out-of-order, we need a heuristic to decide when we've received all spans for a trace (or enough to make a decision). A watermark based on span timestamps plus a max delay tolerance helps determine when to flush incomplete traces.
Wire Format	The serialized representation of data (e.g., spans) for transmission over a network. Our system supports both OTLP/JSON and OTLP/Protobuf wire formats.	Defined by the OpenTelemetry specification. The `OTLPJSONRequest` and related types model the JSON structure, while `parseOTLPProtobuf` handles the binary protobuf format. Choosing a standard wire format ensures interoperability.
Write-Ahead Log (WAL)	A durability mechanism where changes (incoming spans) are written to a sequential log file before being processed and acknowledged, allowing recovery after crashes.	Implemented by `WALWriter` and `WALReader` in **Milestone 1** as part of the buffering strategy. If the collector crashes before spans are processed, they can be replayed from the WAL, preventing data loss at the cost of increased I/O.
z-score	A statistical measurement describing a value's relationship to the mean of a group of values, expressed in terms of standard deviations. Used to detect anomalies.	One method in the `Detector` for **anomaly detection** in **Milestone 4**. A latency value with a high absolute z-score (e.g., >3) is far from the historical mean and likely an anomaly, assuming the data is normally distributed.

This glossary provides a shared vocabulary for discussing the APM Tracing System. When reviewing the design document, refer back to these definitions to ensure precise understanding of each concept's role and implementation.
