

Metrics & Alerting Dashboard: Design Document

Overview

This system collects, stores, and visualizes time-series metrics data while providing intelligent alerting capabilities. The key architectural challenge is building a scalable time-series database that can handle high-throughput metric ingestion, efficient storage with compaction, and real-time querying for both dashboards and alert evaluation.

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.

Context and Problem Statement

Milestone(s): All milestones (this foundational understanding applies throughout the project)

Building a metrics and alerting system is fundamentally about solving the **observability problem**: how do you understand what's happening inside complex software systems that you can't directly see or touch? Modern applications are like vast, interconnected organisms with thousands of moving parts, and without proper instrumentation, diagnosing problems becomes like trying to perform surgery in complete darkness.

This section establishes why metrics systems are essential, what makes them technically challenging to build correctly, and how existing solutions approach these challenges. Understanding these fundamentals will guide every architectural decision we make throughout this project.

The Hospital Monitoring Analogy

Think of a modern metrics and alerting system as the **comprehensive monitoring infrastructure** in a large hospital's intensive care unit. This analogy helps build intuition for the core concepts and challenges we'll encounter.

In an ICU, every patient has multiple sensors continuously measuring vital signs: heart rate, blood pressure, oxygen saturation, temperature, respiratory rate, and dozens of other physiological indicators. These sensors generate **time-series data** — measurements that only make sense when paired with the exact timestamp they were recorded. A heart rate of 85 BPM is meaningless without knowing when it was measured, just like a CPU usage reading of 73% is useless without its timestamp.

The hospital's monitoring system faces the same fundamental challenges our metrics system must solve:

High-Volume Data Ingestion: Across hundreds of patients, thousands of sensors generate measurements every few seconds. The monitoring system must ingest this constant stream of data without dropping measurements or falling behind. Similarly, our metrics system must handle thousands of application instances each reporting dozens of metrics every 10-30 seconds — potentially millions of data points per minute.

Efficient Storage with Retention Policies: The hospital can't store every heartbeat measurement forever — they need different retention strategies. Critical measurements from the last 24 hours are kept at full resolution. Data from the past week might be downsampled to one measurement per minute. Older data gets compressed into hourly or daily averages before eventually being archived or deleted. Our metrics system needs identical **compaction and retention** strategies to manage storage costs while preserving the right level of detail for different time ranges.

Real-Time Visualization: Nurses and doctors need live dashboards showing current patient status across multiple time scales — the last few minutes for immediate assessment, the last few hours to understand trends, and longer periods to track recovery

progress. These dashboards must update automatically as new measurements arrive. Our system's visualization layer serves the same role for application operators and developers.

Intelligent Alerting: The most critical function is automated alerting when vital signs indicate danger. But this is surprisingly complex — a temporary spike in heart rate during physical therapy is normal, but the same reading at 3 AM while sleeping could indicate cardiac distress. Alert rules must consider **context, duration, and patterns**, not just simple thresholds. They must also avoid "alert fatigue" — if every minor fluctuation triggers an alarm, staff will start ignoring them, potentially missing real emergencies.

Multi-Dimensional Context: Each measurement needs rich context to be meaningful. A blood pressure reading must be tagged with the patient ID, measurement location (left arm vs right arm), patient position (sitting vs lying down), and the specific device used. This **labeling and dimensionality** allows medical staff to filter and aggregate data meaningfully. Our metrics system uses the same concept — a CPU usage metric needs labels for the server hostname, application name, data center region, and deployment version.

Cardinality Management: The hospital must be careful about creating too many unique combinations of patient + measurement type + device + location. With 500 patients, 20 vital signs, 50 different monitoring devices, and 10 possible measurement locations, you could theoretically have 5 million unique time series. But most combinations never exist in practice, and creating storage allocation for all possibilities would be wasteful and slow. This is the **cardinality explosion** problem our metrics system must solve — preventing users from accidentally creating millions of sparsely-populated time series through poorly designed label schemes.

The hospital analogy illuminates why building an effective metrics system is challenging: it's not just about storing numbers, but about creating a reliable, efficient, and intelligent monitoring infrastructure that helps operators understand complex systems and respond to problems before they become catastrophic.

Technical Challenges

Building a production-ready metrics system involves solving several interconnected technical challenges that don't have obvious solutions. Each challenge influences the others, creating a web of architectural dependencies that must be carefully balanced.

High-Cardinality Data Management

Cardinality refers to the number of unique time series in your system, determined by every unique combination of metric name and label values. This becomes a scalability nightmare faster than most developers anticipate.

Consider a simple web application that tracks HTTP request duration with labels for `method`, `endpoint`, `status_code`, `datacenter`, and `version`. With just 5 HTTP methods, 20 endpoints, 10 possible status codes, 3 datacenters, and 5 active versions, you have $5 \times 20 \times 10 \times 3 \times 5 = 15,000$ unique time series for a single metric. Add user-specific labels like `user_id` or `session_id`, and you suddenly have millions of time series, most containing only a few data points.

High cardinality creates cascading problems:

- **Memory exhaustion:** Each time series requires index entries and metadata structures in memory. With millions of series, even lightweight metadata can consume gigabytes of RAM.
- **Query performance degradation:** Finding relevant time series requires scanning larger indexes. A query that should return results in milliseconds starts taking seconds.
- **Storage inefficiency:** Most high-cardinality series contain sparse data, leading to poor compression ratios and wasted disk space.
- **Compaction overhead:** Background processes that merge and downsample data must handle exponentially more files and indexes.

The challenge is designing validation and limiting mechanisms that prevent cardinality explosions without restricting legitimate use cases. Users need enough dimensional flexibility to slice and dice their data meaningfully, but the system must protect itself from abuse or misconfiguration.

Design Insight: The cardinality problem is fundamentally about the tension between expressiveness and scalability. Every additional label dimension multiplies potential cardinality, but removing dimensions reduces the system's analytical power.

Storage Efficiency and Compression

Time-series data has unique storage characteristics that make traditional database approaches inefficient. The data is **immutable** (historical measurements never change), **temporally ordered** (queries almost always involve time ranges), and **highly compressible** (consecutive values often have small deltas).

However, the data also arrives out of order — different application instances have clock skew, network delays vary, and some metrics are batch-uploaded from offline processing. The storage engine must efficiently handle both real-time ingestion and historical backfill while maintaining query performance.

Compression becomes critical at scale. Raw time-series data might consume terabytes, but with proper compression (delta encoding, variable-length integers, and block compression), storage requirements can shrink by 10-20x. However, compression complicates queries — you can't seek to arbitrary time points within compressed blocks without partial decompression.

The storage engine must also implement **compaction** — the process of merging small files into larger ones, downsampling high-resolution data to lower resolutions, and eventually deleting old data according to retention policies. Compaction runs continuously in the background but must not interfere with real-time ingestion or queries.

Query Performance at Scale

Users expect sub-second query responses even when requesting data across long time ranges or high-cardinality dimensions. A dashboard loading 20 charts simultaneously might generate 20 concurrent queries, each potentially scanning millions of data points across hundreds of time series.

Query performance depends on several factors:

Index Efficiency: Finding the right time series for a query requires efficient indexing by metric name and labels. Traditional B-tree indexes work poorly for high-cardinality label combinations. The system needs specialized indexing structures optimized for multi-dimensional filtering.

Data Layout: Physical storage layout dramatically affects query performance. Storing all metrics together requires extensive filtering. Storing each time series separately creates too many small files. The optimal layout depends on query patterns, which vary by application.

Aggregation Complexity: Most queries involve aggregation — summing values across multiple time series, computing percentiles, or calculating rates. These operations must be pushed down to the storage layer to avoid transferring massive amounts of raw data over the network.

Caching Strategy: Repeated queries for the same data should be cached, but time-series data is continuously growing. The caching layer must invalidate cached results appropriately while maximizing hit rates for relatively static historical data.

Alert Reliability and Accuracy

Alerting is perhaps the most critical component because alert failures can mean missing production outages. However, reliable alerting involves subtle challenges that are easy to get wrong.

Evaluation Timing: Alert rules must be evaluated at precise intervals, but evaluation itself takes time. If evaluating all alert rules takes 45 seconds, but the evaluation interval is 30 seconds, the system falls behind and might miss brief anomalies. The evaluation scheduler must handle timing constraints gracefully.

State Management: Alerts transition through states (inactive → pending → firing → resolved), and these transitions must be tracked reliably across system restarts. Alert state must be persisted durably, but state updates are frequent and must not become a performance bottleneck.

Flapping Prevention: Metrics that oscillate around alert thresholds can cause alerts to rapidly transition between firing and resolved states, generating notification spam. The system needs hysteresis mechanisms and minimum duration requirements to prevent flapping.

Notification Delivery: Once an alert fires, delivering notifications reliably is surprisingly complex. External services (email, Slack, PagerDuty) can be temporarily unavailable, rate-limited, or return ambiguous error responses. The system must implement retry logic, exponential backoff, and failure detection without losing notifications or creating duplicates.

Existing Solutions Comparison

Understanding how established metrics systems approach these challenges provides valuable context for our architectural decisions. Three representative solutions illustrate different trade-offs and design philosophies.

Prometheus: Pull-Based Open Source

Prometheus pioneered the modern metrics monitoring approach and remains the most influential open-source solution. Its design decisions reflect a philosophy prioritizing simplicity, reliability, and operational transparency.

Architecture Philosophy: Prometheus uses a **pull-based** model where the central server actively scrapes metrics from application endpoints. This inverts the traditional push model and provides several advantages: the monitoring system controls scraping frequency, can detect when applications become unreachable, and avoids overwhelming the central server with concurrent metric submissions.

Data Model: Prometheus uses a dimensional data model where each time series is identified by a metric name plus a set of key-value labels. This model is both powerful and dangerous — it enables flexible querying but makes cardinality explosions easy to create accidentally.

Storage Engine: The Prometheus storage engine (TSDB) uses a custom file format optimized for time-series data. Data is organized into time-based blocks, each containing compressed chunks of time series data plus indexes for efficient querying. The block-based approach enables efficient compaction and retention policy enforcement.

Query Language: PromQL (Prometheus Query Language) is a functional query language designed specifically for time-series data. It supports complex aggregations, mathematical operations, and time-based functions. However, PromQL has a steep learning curve and can produce queries that accidentally consume excessive resources.

Aspect	Advantages	Disadvantages
Pull Model	Service discovery integration, failure detection, backpressure control	Requires network connectivity to all targets, complex NAT/firewall traversal
Local Storage	Simple deployment, no external dependencies, predictable performance	Limited scalability, no built-in high availability, single point of failure
PromQL	Powerful aggregations, mathematical flexibility, time-series specific operations	Complex syntax, easy to write inefficient queries, limited statistical functions
Alertmanager	Sophisticated routing and grouping, silence management, template system	Complex configuration, separate component to manage, limited notification channels

Scalability Limitations: Prometheus is designed for single-node deployment with federation for multi-datacenter scenarios. A single Prometheus server can handle millions of time series, but scaling beyond that requires complex federation topologies or third-party solutions like Thanos or Cortex.

InfluxDB: Purpose-Built Time-Series Database

InfluxDB represents a different approach: building a complete database system optimized specifically for time-series workloads. This allows more sophisticated features but introduces additional complexity.

Architecture Philosophy: InfluxDB uses a **push-based** model where applications send metrics directly to the database. This approach scales more naturally for high-throughput scenarios but requires careful backpressure and rate limiting design.

Data Model: InfluxDB's data model includes measurements (metric names), tags (indexed labels), fields (actual values), and timestamps. The distinction between tags and fields is important — tags are indexed and support efficient filtering, while fields store the actual metric values and support mathematical operations.

Storage Engine: InfluxDB uses the TSM (Time-Structured Merge tree) storage engine, which combines concepts from LSM-trees with time-series-specific optimizations. Data is initially written to a WAL (Write-Ahead Log), then organized into TSM files with sophisticated compression and indexing.

Query Language: InfluxQL is SQL-like, making it more familiar to developers than PromQL. However, SQL wasn't designed for time-series operations, so some queries feel awkward or require complex syntax.

Aspect	Advantages	Disadvantages
Push Model	Better for high-throughput scenarios, works behind firewalls/NAT, simpler network topology	Requires backpressure handling, harder to detect application failures, potential overload scenarios
SQL-like Query Language	Familiar syntax for most developers, rich statistical functions, flexible data manipulation	Some time-series operations feel unnatural, complex joins for multi-metric queries
Clustered Architecture	Built-in horizontal scaling, high availability, automatic data distribution	Complex cluster management, eventual consistency trade-offs, higher operational overhead
Schema Flexibility	Dynamic schema creation, multiple field types per measurement, flexible tag structures	Easy to create inefficient schemas, tag/field distinction confusion, index bloat potential

Commercial Focus: InfluxData's business model centers on InfluxDB Enterprise and InfluxDB Cloud, with the open-source version having limitations in clustering and advanced features. This affects the development priority of community-requested features.

DataDog: Cloud-Native SaaS

DataDog represents the fully managed SaaS approach, where the metrics system complexity is hidden behind a service API. This illustrates how cloud providers solve scalability challenges that are difficult for individual organizations.

Architecture Philosophy: DataDog operates a **multi-tenant** metrics platform serving thousands of customers from shared infrastructure. This requires sophisticated resource isolation, security controls, and cost allocation mechanisms that single-tenant systems don't need.

Data Ingestion: DataDog supports multiple ingestion methods (StatsD, HTTP API, agent-based collection) and handles the complexity of routing, validation, and processing at massive scale. They can absorb traffic spikes and provide backpressure without customers needing to design these systems.

Storage and Retention: DataDog automatically manages retention policies, downsampling, and storage optimization. Customers specify retention requirements through pricing tiers rather than configuring technical parameters.

Alerting and Dashboards: The platform provides sophisticated alerting with machine learning-based anomaly detection, automatic correlation analysis, and integrated incident management workflows.

Aspect	Advantages	Disadvantages
Managed Infrastructure	No operational overhead, automatic scaling, built-in high availability	Vendor lock-in, limited customization, ongoing subscription costs
Advanced Analytics	Machine learning features, anomaly detection, correlation analysis	Less control over algorithms, potential false positives, black-box analysis
Integrated Ecosystem	Unified logging/metrics/tracing, pre-built integrations, collaborative features	Expensive for high-volume scenarios, feature coupling, data export limitations
Enterprise Features	SSO integration, audit trails, role-based access, compliance certifications	Overkill for simple use cases, complex pricing models, feature overwhelming

Cost Considerations: SaaS metrics platforms typically charge based on the number of custom metrics, data retention periods, and advanced feature usage. For high-cardinality applications, monthly costs can quickly reach thousands of dollars, making self-hosted solutions economically attractive.

Decision: Architecture Philosophy

- **Context:** We must choose between pull-based (Prometheus-style), push-based (InfluxDB-style), or hybrid approaches for metric collection
- **Options Considered:**
 1. Pure pull-based with service discovery
 2. Pure push-based with client libraries
 3. Hybrid supporting both models
- **Decision:** Hybrid approach supporting both push and pull models
- **Rationale:** Different applications have different constraints — containerized services work well with pull models, while batch jobs and edge devices need push capabilities. Supporting both provides maximum flexibility for adoption.
- **Consequences:** Increases implementation complexity but provides better real-world usability. Requires designing consistent APIs for both ingestion methods.

The comparison reveals that each approach optimizes for different priorities: operational simplicity (Prometheus), database sophistication (InfluxDB), or comprehensive managed services (DataDog). Our system will need to make similar trade-offs based on the specific requirements and constraints we're targeting.

Understanding these existing solutions helps us identify proven patterns worth adopting and pitfalls worth avoiding. However, building our own system allows us to make different trade-offs and potentially improve on areas where existing solutions have limitations.

Implementation Guidance

The context and challenges described above inform several critical technology choices for our implementation. Understanding the problem space deeply allows us to select appropriate tools and avoid common architectural mistakes.

Technology Stack Recommendations

Component	Simple Option	Advanced Option	Rationale
HTTP Server	<code>net/http</code> with <code>gorilla/mux</code>	<code>gin-gonic/gin</code> or <code>echo</code>	Standard library provides everything needed; frameworks add convenience but dependency overhead
Storage Format	JSON files with <code>encoding/json</code>	Protocol Buffers with <code>google.golang.org/protobuf</code>	JSON is human-readable for debugging; protobuf provides better performance and schema evolution
Time Series Storage	File-based with <code>os</code> package	Embedded <code>badger</code> or <code>bbolt</code> database	Files are simple and debuggable; embedded DBs provide better concurrency and consistency
Background Processing	<code>time.Ticker</code> with goroutines	<code>robfig/cron</code> for scheduling	Simple ticker adequate for basic needs; cron expressions provide more flexible scheduling
Configuration	<code>flag</code> package	<code>spf13/viper</code> with YAML/TOML	Command-line flags are simple; structured config files scale better for complex systems

Project Structure Foundation

Organize the codebase to support the four major milestones while maintaining clear separation of concerns:

```

metrics-system/
├── cmd/
│   ├── server/main.go           ← HTTP server entry point
│   └── query/main.go           ← Query CLI tool for testing
├── internal/
│   ├── ingestion/              ← Milestone 1: Metrics Collection
│   │   ├── handler.go          ← HTTP handlers for push API
│   │   ├── scraper.go          ← Pull-based metric scraping
│   │   ├── validator.go        ← Label validation and cardinality control
│   │   └── processor.go        ← Metric type processing pipeline
│   ├── storage/                ← Milestone 2: Storage & Querying
│   │   ├── engine.go           ← Time-series storage engine
│   │   ├── index.go            ← Label indexing and lookup
│   │   ├── compaction.go       ← Background compaction process
│   │   └── retention.go        ← Data retention policy enforcement
│   ├── query/                  ← Milestone 2: Query processing
│   │   ├── parser.go           ← Query language parsing
│   │   ├── executor.go         ← Query execution engine
│   │   └── aggregation.go      ← Aggregation function library
│   ├── dashboard/              ← Milestone 3: Visualization
│   │   ├── server.go           ← Web dashboard HTTP handlers
│   │   ├── websocket.go        ← Real-time data updates
│   │   └── renderer.go         ← Chart generation logic
│   └── alerting/               ← Milestone 4: Alerting System
│       ├── evaluator.go        ← Alert rule evaluation engine
│       ├── state.go             ← Alert state management
│       └── notifier.go          ← Notification delivery system
└── pkg/
    ├── types/                  ← Public APIs and shared types
    │   ├── metric.go            ← Core data structures
    │   ├── query.go             ← Metric, Sample, Label definitions
    │   └── alert.go              ← Query request/response types
    └── client/                 ← Alert rule and state types
        └── client.go            ← Client library for metric submission
└── web/
    ├── static/                 ← Client library for push API
    └── templates/              ← HTTP client for push API
    └── static/                 ← Dashboard static assets
        ├── CSS, JavaScript, images
        └── HTML templates
└── testdata/
    ├── metrics/                ← Sample metrics and test fixtures
    └── configs/                ← Example metric data files
                                ← Sample configuration files

```

Infrastructure Starter Code

HTTP Server Foundation (`internal/server/server.go`):

```
package server

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

    "github.com/gorilla/mux"
)

// Server wraps the HTTP server with graceful shutdown support

type Server struct {
    httpServer *http.Server
    router     *mux.Router
}

// NewServer creates a new HTTP server with default middleware

func NewServer(addr string) *Server {
    router := mux.NewRouter()

    // Add basic middleware
    router.Use(loggingMiddleware)
    router.Use(corsMiddleware)

    return &Server{
        httpServer: &http.Server{
            Addr:         addr,
            Handler:     router,
            ReadTimeout: 30 * time.Second,
            WriteTimeout: 30 * time.Second,
            IdleTimeout: 120 * time.Second,
        }
    }
}
```

```
        },

        router: router,
    }

}

// RegisterRoutes adds all API endpoints to the router

func (s *Server) RegisterRoutes(ingestionHandler, queryHandler, dashboardHandler http.Handler) {

    // Metrics ingestion endpoints

    s.router.PathPrefix("/api/v1/metrics").Handler(ingestionHandler)

    // Query API endpoints

    s.router.PathPrefix("/api/v1/query").Handler(queryHandler)

    // Dashboard and static assets

    s.router.PathPrefix("/").Handler(dashboardHandler)
}

// Start begins serving HTTP requests

func (s *Server) Start() error {

    log.Printf("Starting HTTP server on %s", s.httpServer.Addr)

    return s.httpServer.ListenAndServe()
}

// Shutdown gracefully stops the server

func (s *Server) Shutdown(ctx context.Context) error {

    log.Println("Shutting down HTTP server...")

    return s.httpServer.Shutdown(ctx)
}

func loggingMiddleware(next http.Handler) http.Handler {

    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {

        start := time.Now()

        next.ServeHTTP(w, r)

        log.Printf("%s %s %v", r.Method, r.URL.Path, time.Since(start))
    })
}
```

```
    })

}

func corsMiddleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Header().Set("Access-Control-Allow-Origin", "*")
        w.Header().Set("Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS")
        w.Header().Set("Access-Control-Allow-Headers", "Content-Type")

        if r.Method == "OPTIONS" {
            w.WriteHeader(http.StatusOK)
            return
        }

        next.ServeHTTP(w, r)
    })
}
```

Core Data Types (`pkg/types/metric.go`):

```
package types
```

GO

```
import (
    "fmt"
    "sort"
    "strings"
    "time"
)
```

```
// MetricType represents the type of metric being stored
```

```
type MetricType int
```

```
const (
```

```
    MetricTypeCounter MetricType = iota
    MetricTypeGauge
    MetricTypeHistogram
```

```
)
```

```
// String returns the string representation of the metric type
```

```
func (mt MetricType) String() string {
```

```
    switch mt {
        case MetricTypeCounter:
            return "counter"
```

```
        case MetricTypeGauge:
            return "gauge"
```

```
        case MetricTypeHistogram:
            return "histogram"
```

```
    default:
```

```
        return "unknown"
```

```
}
```

```
}
```

```
// Labels represents a set of key-value pairs attached to a metric
```

```
type Labels map[string]string
```

```

// String returns a canonical string representation of labels

func (l Labels) String() string {
    if len(l) == 0 {
        return "{}"
    }

    // Sort keys for consistent output
    keys := make([]string, 0, len(l))

    for k := range l {
        keys = append(keys, k)
    }

    sort.Strings(keys)

    parts := make([]string, 0, len(l))

    for _, k := range keys {
        parts = append(parts, fmt.Sprintf(`%s=%s`, k, l[k]))
    }

    return "{" + strings.Join(parts, ",") + "}"
}

// Copy creates a deep copy of the labels

func (l Labels) Copy() Labels {
    result := make(Labels, len(l))

    for k, v := range l {
        result[k] = v
    }

    return result
}

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

type Sample struct {
    Value     float64   `json:"value"`
}

```

```
    Timestamp time.Time `json:"timestamp"`

}

// Metric represents a complete metric with its metadata and samples

type Metric struct {

    Name      string      `json:"name"`

    Type     MetricType `json:"type"`

    Labels   Labels      `json:"labels"`

    Samples []Sample    `json:"samples"`

    Help     string      `json:"help,omitempty"`

}

// SeriesID generates a unique identifier for this metric's time series

func (m *Metric) SeriesID() string {

    return m.Name + m.Labels.String()

}

// Validate checks if the metric has valid values

func (m *Metric) Validate() error {

    if m.Name == "" {

        return fmt.Errorf("metric name cannot be empty")

    }

    if len(m.Samples) == 0 {

        return fmt.Errorf("metric must have at least one sample")

    }

    // Validate label keys and values

    for key, value := range m.Labels {

        if key == "" {

            return fmt.Errorf("label key cannot be empty")

        }

        if strings.HasPrefix(key, "__") {


```

```

        return fmt.Errorf("label key %s: keys starting with __ are reserved", key)

    }

    if len(value) > 1024 {

        return fmt.Errorf("label value for key %s exceeds maximum length", key)

    }

}

return nil
}

```

Milestone Checkpoints

After Milestone 1 (Metrics Collection): Run the following to verify ingestion is working:

```

# Start the server

go run cmd/server/main.go

# Send a test metric

curl -X POST http://localhost:8080/api/v1/metrics \
-H "Content-Type: application/json" \
-d '{
  "name": "http_requests_total",
  "type": "counter",
  "labels": {"method": "GET", "endpoint": "/api"},
  "samples": [{"value": 42, "timestamp": "2023-01-01T12:00:00Z"}]
}'

# Should receive 200 OK response

# Check server logs for ingestion confirmation

```

After Milestone 2 (Storage & Querying):

```

# Query the stored metric
curl "http://localhost:8080/api/v1/query?metric=http_requests_total&start=2023-01-01T11:00:00Z&end=2023-01-01T13:00:00Z"

# Should return JSON with the stored sample

# Verify data persists across server restarts

```

Common Debugging Issues

Symptom	Likely Cause	Diagnosis	Fix
HTTP 500 on metric ingestion	JSON parsing error or validation failure	Check request body format and server logs	Validate JSON structure and required fields
Metrics disappear after restart	In-memory storage without persistence	Check if data is being written to disk	Implement file-based storage in storage engine
Query returns empty results	Label matching issues or wrong time range	Verify metric name and labels match exactly	Add debug logging to query execution
High memory usage	Unbounded metric storage or label cardinality explosion	Monitor number of unique time series	Implement retention policies and label validation

Goals and Non-Goals

Milestone(s): All milestones (this section establishes the foundational scope and boundaries for the entire project)

The Goldilocks Principle for System Design

Building a metrics and alerting system is like designing a security system for a large office building. You need to balance comprehensive coverage with practical constraints. Install too few sensors and you miss critical events. Install too many and you're overwhelmed with false alarms, maintenance costs, and complexity. The key is finding the "just right" balance that meets your actual needs without overengineering.

This section establishes clear boundaries for what our system will and will not do. These boundaries are crucial because metrics systems can easily become sprawling, complex beasts that try to solve every observability problem under the sun. By explicitly defining our scope, we prevent feature creep and ensure we build a focused, reliable system that excels at its core mission.

The goals we set here directly drive every architectural decision throughout the project. When we face trade-offs between simplicity and features, performance and functionality, or immediate needs versus future flexibility, we'll return to these goals as our north star.

Functional Goals

These are the core capabilities our system must provide to be considered successful. Each goal maps directly to one or more project milestones and represents functionality that users will interact with directly.

Metric Collection and Storage

Our system must reliably collect, validate, and store three fundamental metric types that form the foundation of observability.

Counters track cumulative values that only increase over time, such as total HTTP requests or bytes processed. **Gauges** capture

point-in-time values that can fluctuate up or down, like current memory usage or active connections. **Histograms** bucket observations into configurable ranges to analyze distributions, enabling percentile calculations for latency or request size analysis.

The system must support both push-based and pull-based metric collection models. In the push model, applications actively send metrics to our collection endpoints, providing immediate data delivery and working well with dynamic environments like containers. In the pull model, our system scrapes metrics from application endpoints, offering better control over collection timing and reducing load on applications.

Metric Type	Purpose	Example Use Cases	Key Characteristics
Counter	Track cumulative events	HTTP requests, errors, bytes sent	Monotonically increasing, reset on restart
Gauge	Capture current state	Memory usage, queue depth, temperature	Can increase or decrease, point-in-time value
Histogram	Analyze distributions	Request latency, response size	Buckets observations, tracks count and sum

Decision: Support Multi-Dimensional Metrics

- **Context:** Modern applications need to slice metrics by various dimensions (service, region, user type)
- **Options Considered:** Flat metrics with no labels, limited predefined dimensions, flexible label system
- **Decision:** Implement flexible label system similar to Prometheus
- **Rationale:** Labels enable powerful querying and aggregation while maintaining storage efficiency through shared time series
- **Consequences:** Adds complexity to storage and indexing but provides essential functionality for real-world use

Time-Series Data Management

The system must efficiently store millions of time-series data points with high write throughput while maintaining fast query performance. Time-series data has unique characteristics: it's append-only, timestamp-ordered, and often exhibits predictable patterns that enable compression.

Data retention policies automatically manage the lifecycle of stored metrics. Recent data (last 24 hours) remains at full resolution for detailed analysis. Older data gets progressively downsampled to reduce storage costs while preserving long-term trends. Data older than the configured retention period gets automatically deleted.

Data Age	Resolution	Purpose	Storage Impact
0-24 hours	Original (15s intervals)	Real-time monitoring	High storage, fast queries
1-7 days	1 minute averages	Recent troubleshooting	Medium storage
1-4 weeks	5 minute averages	Trend analysis	Low storage
> 1 month	Deleted or archived	Compliance only	Minimal storage

Compaction processes run automatically in the background, merging small data files into larger, more efficient structures. This reduces the number of files that queries must read and improves compression ratios by processing larger data blocks together.

Query and Visualization Capabilities

The system must provide a query language that allows users to select metrics by name and labels, specify time ranges, and apply aggregation functions. Queries should feel natural to operations teams familiar with tools like Prometheus or SQL.

Basic aggregation functions include sum, average, maximum, minimum, and rate calculations over time windows. Rate functions are particularly important for converting counter metrics into meaningful rates (requests per second, error rate percentages).

Function	Purpose	Input	Output
sum()	Total across series	Multiple time series	Single aggregated series
avg()	Average across series	Multiple time series	Single averaged series
rate()	Change rate over time	Counter series	Rate per second
quantile()	Percentile calculation	Histogram series	Percentile value series

The web dashboard provides visual representations of metric data through line charts, bar charts, and heatmaps. Dashboard configurations persist as JSON documents, enabling sharing and version control. Real-time updates refresh charts automatically without requiring manual page reloads.

Alerting and Notification System

Alert rules define conditions that trigger notifications when metric values cross thresholds or exhibit anomalous behavior. Each rule specifies a metric query, comparison operator, threshold value, and evaluation interval.

Alert state management handles transitions between inactive, pending, firing, and resolved states. Alerts enter pending state when conditions are first met, then escalate to firing after persisting for the configured duration. This prevents noisy alerts from temporary spikes.

Alert State	Trigger Condition	Actions Taken	Next States
Inactive	Normal metric values	None	Pending
Pending	Threshold crossed	Start timer	Inactive, Firing
Firing	Duration exceeded	Send notifications	Inactive, Resolved
Resolved	Condition cleared	Send resolution notice	Inactive

Notification channels deliver alert messages through multiple integrations: email, Slack, PagerDuty, and generic webhooks.

Message templates allow customization of alert content with metric values, labels, and contextual information.

Non-Functional Goals

These requirements define how well the system performs its functional capabilities. They establish the operational characteristics needed for production deployment and long-term success.

Performance Requirements

The metrics ingestion system must handle sustained write loads of 10,000 samples per second on modest hardware (4 CPU cores, 8GB RAM). This throughput supports monitoring for hundreds of applications or services in a medium-sized organization.

Query response times should remain under 5 seconds for dashboard refreshes and under 30 seconds for complex analytical queries. These targets ensure dashboards feel responsive while accommodating the inherent costs of aggregating large time-series datasets.

Workload Type	Target Performance	Resource Constraints
Metric ingestion	10,000 samples/sec	4 CPU cores, 8GB RAM
Dashboard queries	< 5 seconds	Standard SSD storage
Analytical queries	< 30 seconds	Network bandwidth < 1Gbps
Alert evaluation	< 60 seconds	Configurable intervals

Storage efficiency targets aim for 80% compression ratios compared to raw metric data. Time-series data compresses well due to temporal locality and predictable patterns, making this target achievable with proper encoding techniques.

Reliability Requirements

The system must maintain 99.9% uptime, allowing for planned maintenance windows and occasional failures. This translates to less than 9 hours of downtime per year, appropriate for internal monitoring systems that aren't customer-facing.

Data durability guarantees ensure that metric data, once acknowledged as received, survives single-node failures. This requires synchronous persistence to disk before sending acknowledgments, with optional replication for higher availability environments.

Component	Availability Target	Recovery Time	Data Loss Tolerance
Ingestion API	99.9%	< 5 minutes	Zero (acknowledged data)
Query API	99.5%	< 10 minutes	Read-only degradation acceptable
Dashboard	99.5%	< 10 minutes	Cached data acceptable
Alerting	99.9%	< 2 minutes	Critical for incident response

Alert delivery reliability requires redundant notification channels and retry mechanisms. Critical alerts should reach on-call personnel even if primary notification channels fail.

Scalability Requirements

The system must scale vertically on single nodes before requiring distributed deployment. This simplifies operations and reduces complexity for organizations that don't need web-scale infrastructure.

Storage scalability targets support 1TB of metric data on standard hardware, sufficient for monitoring medium-sized infrastructures. Beyond this threshold, organizations typically invest in dedicated time-series databases or cloud services.

Resource	Scaling Target	Growth Strategy
CPU utilization	< 70% average	Vertical scaling first
Memory usage	< 80% available	Efficient data structures
Storage capacity	1TB total	Automatic compaction/retention
Network bandwidth	< 50% link capacity	Compression and batching

Horizontal scaling preparation means avoiding architectural choices that prevent future distribution. While the initial implementation runs on single nodes, the design should accommodate future sharding or federation.

Operational Requirements

The system must provide comprehensive observability into its own operation through self-monitoring metrics, structured logging, and health check endpoints. Operations teams need visibility into ingestion rates, storage usage, query performance, and alert

evaluation timing.

Configuration management uses file-based configuration with hot-reload capabilities for non-disruptive updates. This enables GitOps workflows where configuration changes flow through version control and automated deployment pipelines.

Operational Aspect	Requirement	Implementation
Self-monitoring	All components instrumented	Internal metrics exposure
Configuration	Hot-reload support	File watching mechanisms
Logging	Structured JSON logs	Configurable log levels
Health checks	HTTP endpoints	Dependency verification

Explicit Non-Goals

These are capabilities we explicitly choose not to build, either due to complexity constraints, timeline limitations, or because better solutions exist elsewhere. Defining non-goals is as important as defining goals because it prevents scope creep and helps resist the temptation to build unnecessary features.

Advanced Analytics and Machine Learning

Our system will not include anomaly detection algorithms, predictive analytics, or machine learning-based alerting. These capabilities require significant expertise in data science, large amounts of historical data for training, and ongoing model maintenance.

The complexity of implementing reliable machine learning features far exceeds the value for most monitoring use cases.
Traditional threshold-based alerting handles 90% of operational needs effectively.

Organizations needing advanced analytics should integrate dedicated ML platforms or cloud services that specialize in these capabilities. Our system can export data to these platforms through standard APIs.

Multi-Tenancy and Access Control

The system will not implement user authentication, role-based access control, or tenant isolation. These features add significant complexity to every component and require expertise in security architecture.

For production deployments requiring access control, the system should be deployed behind reverse proxies or API gateways that handle authentication and authorization. This separation of concerns keeps our system focused on metrics while leveraging specialized security tools.

Distributed Deployment and High Availability

The initial implementation will not support clustering, replication, or distributed deployment. Building reliable distributed systems requires expertise in consensus algorithms, failure detection, and data consistency that would dominate the project scope.

Organizations requiring high availability should deploy multiple independent instances or use cloud-managed time-series databases for critical production workloads. Our system focuses on being an excellent single-node solution.

Log Aggregation and Tracing Integration

While metrics are one pillar of observability, we will not build log aggregation or distributed tracing capabilities. These domains have different data models, storage requirements, and query patterns that warrant dedicated systems.

The system should provide integration points (webhook notifications, metric export APIs) that allow correlation with external logging and tracing systems, but will not attempt to replace specialized tools like ELK stack or Jaeger.

Advanced Visualization and Dashboarding

We will not build advanced chart types (3D visualizations, geographic maps, complex statistical plots) or dashboard features like annotation, collaboration, or advanced templating. These features require significant frontend development effort and compete with established visualization platforms.

The dashboard will focus on core time-series visualizations (line charts, bar charts, simple heatmaps) that handle 80% of monitoring use cases. Users needing advanced visualizations should export data to specialized tools like Grafana or custom applications.

Non-Goal Category	Rationale	Alternative Approach
Machine Learning	Complexity exceeds project scope	Integration with ML platforms
Multi-tenancy	Security complexity too high	Deploy behind auth proxy
Distributed Systems	Requires distributed systems expertise	Multiple independent instances
Log Aggregation	Different data model and requirements	Use dedicated logging platforms
Advanced Visualization	Frontend complexity	Export to Grafana or similar

Performance Beyond Single-Node Limits

We will not optimize for ingestion rates exceeding 100,000 samples per second or storage requirements beyond 10TB. These requirements indicate need for specialized time-series databases or distributed systems.

The design should avoid architectural choices that prevent future scaling, but will not implement complex optimization techniques like custom storage engines, advanced indexing structures, or distributed query processing.

Complex Query Languages and Analytics

The query language will not support joins between different metric series, complex mathematical functions, or SQL-like analytical queries. These features require query optimization engines and add significant implementation complexity.

Users needing complex analytics should export metric data to dedicated analytics platforms or data warehouses that provide full SQL support and analytical processing capabilities.

Key Insight: By explicitly limiting scope, we can build a robust, reliable system that excels at core metrics functionality rather than a complex system that attempts everything but does nothing well.

Success Criteria and Acceptance

Success will be measured by the system's ability to handle realistic monitoring workloads for development and staging environments. A successful implementation should enable a small operations team to monitor a few dozen applications and services effectively.

The system succeeds if it can replace simple monitoring solutions like basic Prometheus setups while providing better usability and integration capabilities. It should feel natural to operations engineers familiar with modern monitoring tools.

Success Metric	Target Value	Measurement Method
Metric ingestion rate	5,000+ samples/sec sustained	Load testing with realistic data
Query response time	95th percentile < 10 seconds	Dashboard usage simulation
Storage efficiency	> 70% compression ratio	Compare raw vs compressed sizes
Alert reliability	99%+ delivery success	Notification tracking logs

The ultimate test is whether the system provides value in real monitoring scenarios: detecting actual incidents, enabling root cause analysis through dashboard exploration, and reducing time to resolution for operational issues.

Implementation Guidance

Technology Foundation

The system builds on proven technologies that balance simplicity with capability. These choices prioritize reliability and maintainability over cutting-edge features.

Component	Simple Option	Advanced Option
HTTP Server	<code>net/http</code> with standard middleware	Fiber or Echo framework
Storage Backend	File-based with custom indexing	Embedded database (BadgerDB)
Data Serialization	JSON for config, binary for metrics	Protocol Buffers with compression
Frontend Framework	Vanilla JavaScript with Chart.js	React or Vue.js SPA

Project Structure

The codebase organization separates concerns clearly while maintaining simplicity for learning purposes:

```
metrics-dashboard/
├── cmd/
│   ├── server/main.go           ← HTTP server entry point
│   └── tools/                  ← Data import/export utilities
├── internal/
│   ├── metrics/                ← Core metric types and validation
│   │   ├── types.go             ← MetricType, Labels, Sample structs
│   │   └── validation.go       ← Validate() methods
│   ├── ingestion/              ← Metric collection engine
│   │   ├── handlers.go         ← HTTP endpoint handlers
│   │   ├── prometheus.go      ← Prometheus format support
│   │   └── validation.go       ← Label cardinality control
│   ├── storage/                ← Time-series storage engine
│   │   ├── engine.go           ← Main storage interface
│   │   ├── compaction.go      ← Background compaction
│   │   └── retention.go        ← Data lifecycle management
│   ├── query/                  ← Query processing engine
│   │   ├── parser.go           ← Query language parsing
│   │   ├── executor.go        ← Query execution
│   │   └── functions.go        ← Aggregation functions
│   ├── dashboard/              ← Web dashboard
│   │   ├── server.go           ← Static file serving
│   │   ├── api.go              ← Dashboard API endpoints
│   │   └── websocket.go        ← Real-time updates
│   └── alerting/               ← Alert evaluation and notification
│       ├── evaluator.go       ← Alert rule evaluation
│       ├── notifier.go        ← Notification delivery
│       └── state.go            ← Alert state management
└── web/
    ├── static/                ← Frontend assets
    └── templates/             ← CSS, JavaScript, images
config/
├── server.yaml               ← Configuration files
└── alerts.yaml               ← Main server configuration
                                ← Alert rule definitions
                                ← Documentation and examples
```

Core Type Definitions

These fundamental types establish the data model foundation that all components build upon:

```
// Package metrics defines core data structures for the metrics system.

package metrics

import (
    "fmt"
    "sort"
    "strings"
    "time"
)

// MetricType represents the type of metric being stored.

type MetricType int

const (
    MetricTypeCounter MetricType = iota
    MetricTypeGauge
    MetricTypeHistogram
)

// Labels represents key-value pairs attached to metrics for multi-dimensional querying.

type Labels map[string]string

// String returns the canonical string representation of labels for storage keys.

func (l Labels) String() string {
    // TODO 1: Extract keys into slice and sort alphabetically
    // TODO 2: Build string in format "key1=value1,key2=value2"
    // TODO 3: Handle special characters in keys/values
}

// Copy creates a deep copy of the labels map.

func (l Labels) Copy() Labels {
    // TODO 1: Create new map with same capacity
    // TODO 2: Copy each key-value pair
    // TODO 3: Return new map
}
```

```

// Sample represents a single metric measurement at a point in time.

type Sample struct {
    Value     float64   `json:"value"`
    Timestamp time.Time `json:"timestamp"`
}

// Metric represents a complete metric with metadata and sample data.

type Metric struct {
    Name      string    `json:"name"`
    Type      MetricType `json:"type"`
    Labels   Labels    `json:"labels"`
    Samples []Sample   `json:"samples"`
}

// SeriesID returns a unique identifier for this metric's time series.

func (m *Metric) SeriesID() string {
    // TODO 1: Combine metric name with sorted labels
    // TODO 2: Create deterministic string representation
    // TODO 3: Consider using hash for very long IDs
}

// Validate checks if the metric has valid fields and structure.

func (m *Metric) Validate() error {
    // TODO 1: Check metric name is non-empty and valid format
    // TODO 2: Verify metric type is valid enum value
    // TODO 3: Validate labels don't exceed cardinality limits
    // TODO 4: Check samples are sorted by timestamp
    // TODO 5: Validate sample values for metric type (counters non-negative, etc.)
}

```

HTTP Server Foundation

The server infrastructure provides the foundation for all API endpoints with proper middleware and graceful shutdown:

GO

```
// Package server provides HTTP server infrastructure for the metrics system.

package server

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

// Server wraps an HTTP server with middleware and graceful shutdown capabilities.

type Server struct {

    httpServer *http.Server

    mux        *http.ServeMux
}

// NewServer creates a new HTTP server listening on the specified address.

func NewServer(addr string) *Server {
    mux := http.NewServeMux()

    server := &http.Server{
        Addr:         addr,
        Handler:      mux,
        ReadTimeout:  15 * time.Second,
        WriteTimeout: 15 * time.Second,
        IdleTimeout:  60 * time.Second,
    }

    return &Server{
        httpServer: server,
        mux:        mux,
    }
}

// RegisterRoutes adds API endpoint handlers to the server.
```

```

func (s *Server) RegisterRoutes(handlers map[string]http.HandlerFunc) {

    // TODO 1: Iterate through handler map

    // TODO 2: Register each route with middleware

    // TODO 3: Add logging and metrics middleware

    // TODO 4: Add CORS headers for browser requests

}

// Start begins serving HTTP requests.

func (s *Server) Start() error {

    // TODO 1: Log server startup information

    // TODO 2: Start HTTP server

    // TODO 3: Handle startup errors

}

// Shutdown performs graceful shutdown with timeout.

func (s *Server) Shutdown(ctx context.Context) error {

    // TODO 1: Log shutdown initiation

    // TODO 2: Stop accepting new connections

    // TODO 3: Wait for existing requests to complete

    // TODO 4: Force close after timeout

}

```

Milestone Checkpoints

After implementing each milestone, verify the system works correctly with these concrete tests:

Milestone 1 - Metrics Collection:

- Start the server: `go run cmd/server/main.go`
- Send a counter metric: `curl -X POST localhost:8080/api/v1/metrics -d '{"name": "http_requests", "type": 0, "labels": {"service": "api"}, "samples": [{"value": 1, "timestamp": "2024-01-01T12:00:00Z"}]}'`
- Verify response: Should return 200 OK with no error message
- Check logs: Should show metric ingestion and validation messages

Milestone 2 - Storage & Querying:

- Query the stored metric: `curl "localhost:8080/api/v1/query?metric=http_requests&start=2024-01-01T11:00:00Z&end=2024-01-01T13:00:00Z"`
- Verify response: Should return JSON with the stored sample
- Check storage files: Should see new files created in data directory

- Test aggregation: `curl "localhost:8080/api/v1/query?metric=http_requests&start=2024-01-01T11:00:00Z&end=2024-01-01T13:00:00Z&func=sum"`

Milestone 3 - Visualization Dashboard:

- Open browser: Navigate to `http://localhost:8080/dashboard`
- Create chart: Should see interface for adding metric queries
- View data: Should see line chart with the stored metrics
- Test auto-refresh: Chart should update automatically every 30 seconds

Milestone 4 - Alerting System:

- Create alert rule: POST to `/api/v1/alerts` with threshold condition
- Trigger alert: Send metric values that exceed the threshold
- Verify notification: Check configured notification channel for alert message
- Test resolution: Send normal values and verify resolution notification

High-Level Architecture

Milestone(s): All milestones (this architectural foundation supports the entire metrics and alerting system)

The Orchestra Metaphor: Understanding System Architecture

Think of a metrics and alerting system like a symphony orchestra preparing for and performing a concert. The **Ingestion Engine** is like the stage crew collecting sheet music from various composers (applications) and organizing it by instrument and section. The **Storage Engine** acts as the music library, carefully cataloging and preserving every piece of music with efficient indexing so any composition can be quickly retrieved. The **Query Engine** functions as the conductor's assistant, who can instantly find specific musical passages, transpose them to different keys, or create medleys by combining multiple pieces. The **Dashboard** serves as the concert hall's display system, showing the audience (operators) what's currently being performed with beautiful visualizations of the musical data. Finally, the **Alerting System** operates like the concert hall's fire safety system, continuously monitoring for problems and immediately notifying the right people when something goes wrong.

Just as each orchestra section has distinct responsibilities but must coordinate seamlessly to create beautiful music, our metrics system components have clear ownership boundaries while collaborating through well-defined interfaces to provide comprehensive observability.

Component Responsibilities

Our metrics and alerting system divides functionality across five major components, each with distinct responsibilities and clear interfaces. This separation of concerns enables independent development, testing, and scaling of each component while maintaining clean integration points.

System Component Architecture



External Systems

metric streams

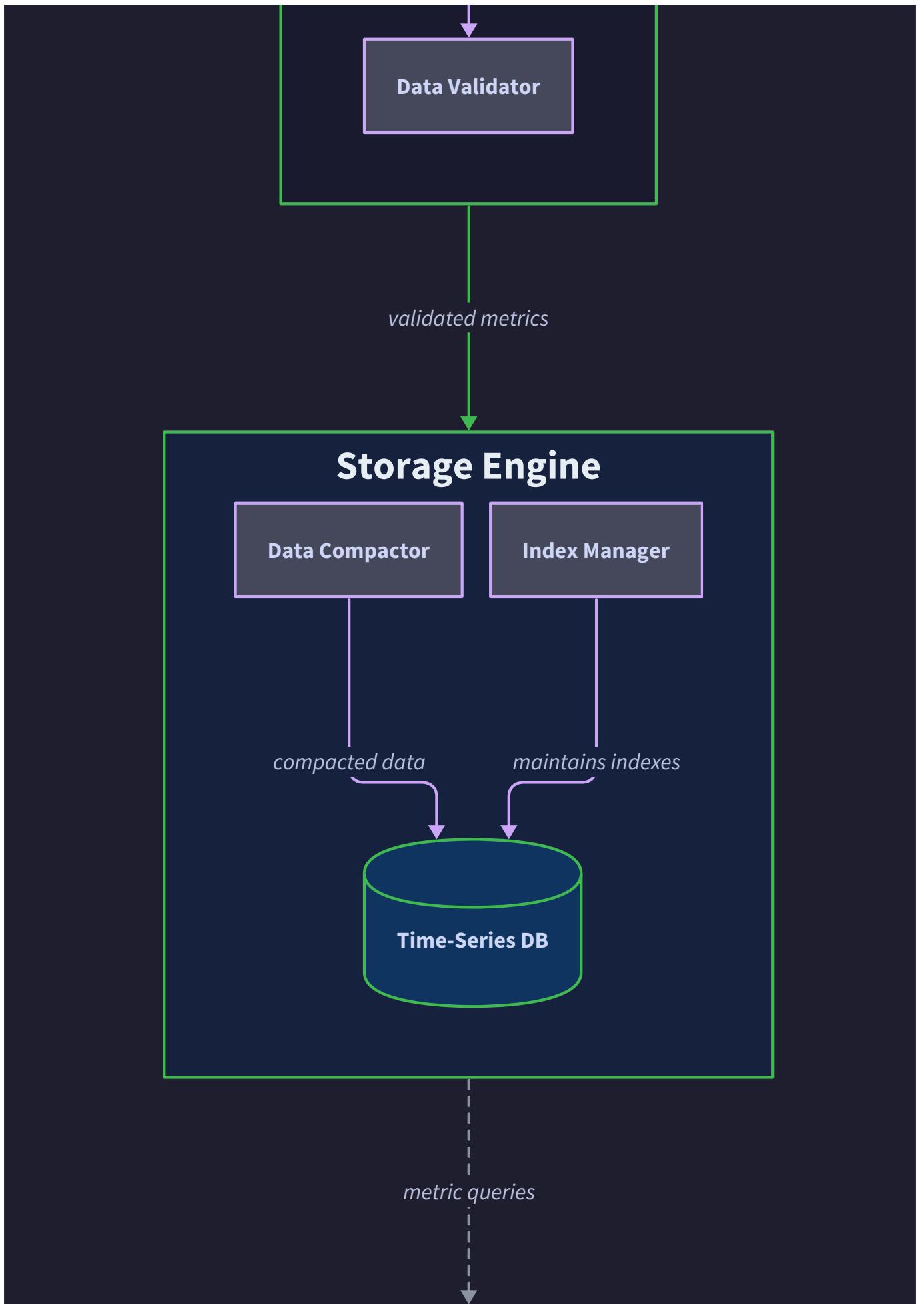
Ingestion Engine

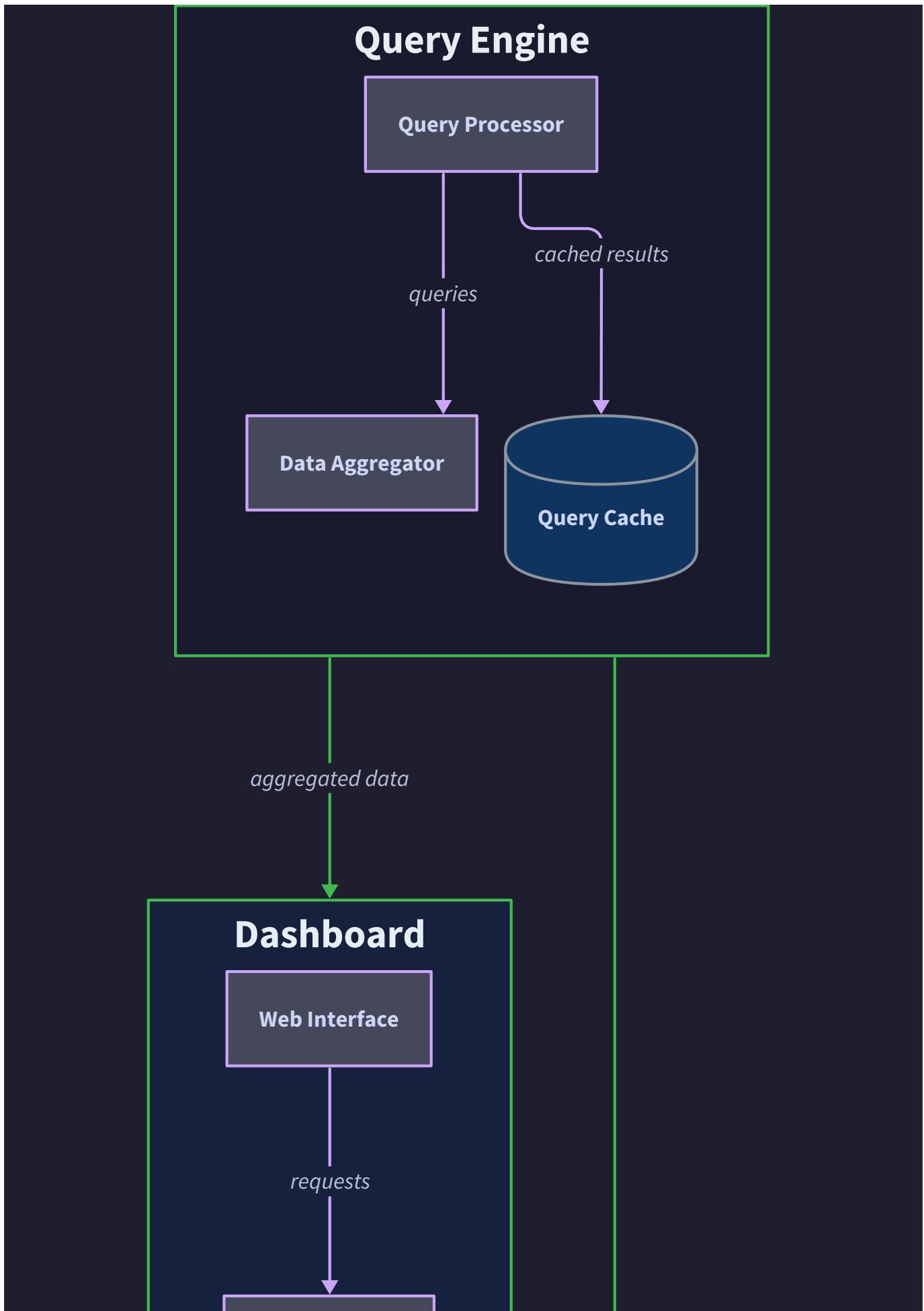
Metrics Collector

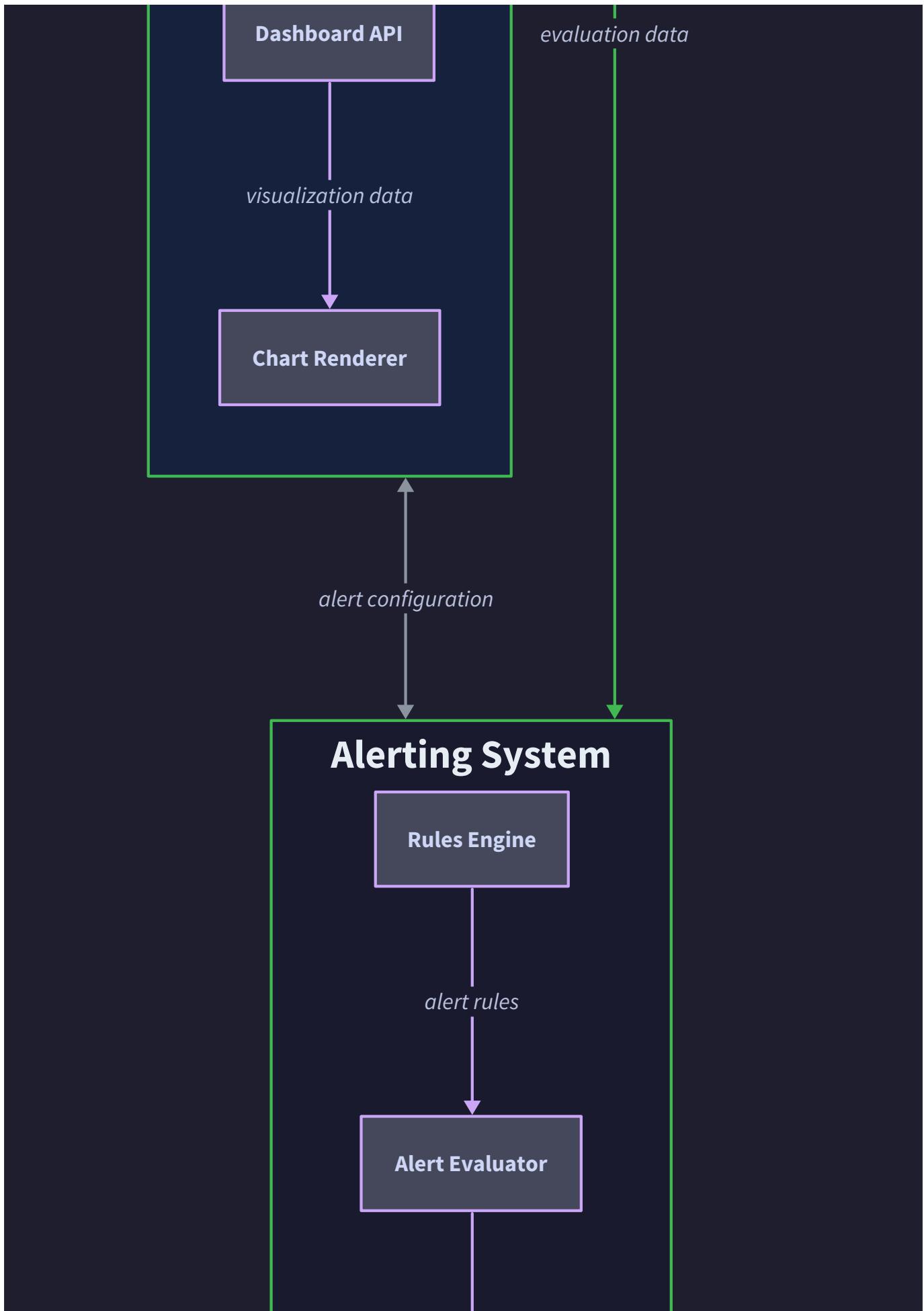
raw metrics

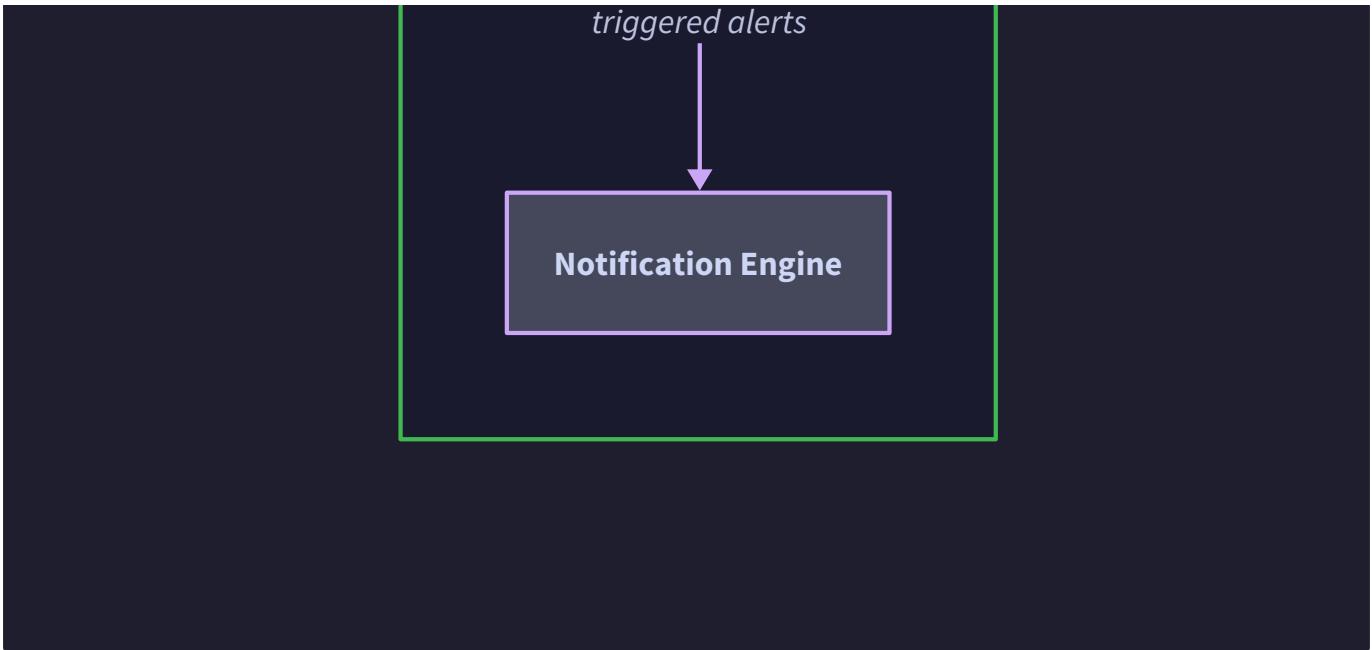
Data Parser

parsed data









The **Ingestion Engine** owns all aspects of receiving and preprocessing metrics data from external sources. It handles both push-based metrics (where applications actively send data to our system) and pull-based metrics (where our system scrapes metrics from application endpoints). The ingestion engine is responsible for validating incoming metric data, enforcing label cardinality limits to prevent explosion, normalizing metric names and label formats, and performing any necessary data transformations before storage. It also implements rate limiting and backpressure mechanisms to protect downstream components from being overwhelmed by metric data floods.

Component Responsibility	Description	Key Interfaces
Protocol Handling	Accept HTTP POST requests with metric data, scrape Prometheus endpoints	<code>IngestMetrics(metrics []Metric) error</code> , <code>ScrapeEndpoint(url string) error</code>
Data Validation	Verify metric types, validate label formats, enforce naming conventions	<code>ValidateMetric(metric Metric) error</code> , <code>CheckCardinality(labels Labels) error</code>
Preprocessing	Normalize metric names, sanitize label values, apply transformations	<code>NormalizeMetric(metric Metric) Metric</code> , <code>SanitizeLabels(labels Labels) Labels</code>
Backpressure Control	Implement rate limiting, queue management, graceful degradation	<code>CheckRateLimit(source string) bool</code> , <code>QueueMetric(metric Metric) error</code>

The **Storage Engine** takes complete ownership of persisting time-series data efficiently and providing fast access patterns for both recent and historical data. It implements a sophisticated storage format optimized for time-series workloads, with separate handling for metadata (metric names and labels) and sample data (timestamps and values). The storage engine automatically manages data lifecycle through compaction processes that merge small files into larger ones and downsample older data to reduce storage requirements. It also enforces retention policies by automatically deleting data that exceeds configured age limits.

Storage Responsibility	Description	Key Interfaces
Sample Storage	Persist timestamped metric values with efficient compression	<code>WriteSamples(samples []Sample) error</code> , <code>ReadSamples(query TimeRange) []Sample</code>
Metadata Management	Index metric names and labels for fast lookup	<code>IndexMetric(name string, labels Labels) SeriesID</code> , <code>LookupSeries(query LabelQuery) []SeriesID</code>
Data Lifecycle	Automatic compaction, downsampling, and retention enforcement	<code>CompactBlocks(blocks []BlockID) error</code> , <code>ApplyRetentionPolicy() error</code>
Query Optimization	Maintain indexes and statistics for efficient query execution	<code>GetSeriesStats(series SeriesID) SeriesStats</code> , <code>OptimizeQuery(query Query) QueryPlan</code>

The **Query Engine** serves as the intelligent data access layer, translating user queries into efficient storage operations and applying mathematical transformations to raw time-series data. It parses query expressions written in our custom query language, develops optimal execution plans that minimize storage I/O, and executes aggregation functions across multiple time series. The query engine also handles time alignment, ensuring that data points from different series are properly synchronized when performing mathematical operations.

Query Responsibility	Description	Key Interfaces
Query Parsing	Parse query strings into structured query objects	<code>ParseQuery(queryString string) (Query, error)</code>
Execution Planning	Optimize query execution order and storage access patterns	<code>PlanQuery(query Query) QueryPlan</code> , <code>EstimateCost(plan QueryPlan) int</code>
Data Aggregation	Apply mathematical functions across time series	<code>Aggregate(function AggFunc, series []TimeSeries) TimeSeries</code>
Result Formatting	Transform query results into client-requested formats	<code>FormatResults(results []TimeSeries, format OutputFormat) []byte</code>

The **Dashboard** component provides the web-based user interface for visualizing metrics data and managing system configuration. It maintains a persistent store of dashboard configurations, handles user authentication and authorization, and manages real-time data updates through WebSocket connections. The dashboard component also generates shareable URLs and embeddable widgets for distributing metrics visibility across different teams and systems.

Dashboard Responsibility	Description	Key Interfaces
Configuration Management	Save/load dashboard layouts and panel configurations	<code>SaveDashboard(config DashboardConfig) error</code> , <code>LoadDashboard(id string) DashboardConfig</code>
Real-Time Updates	Push live metric data to browser clients via WebSockets	<code>SubscribeToUpdates(client ClientID, queries []Query)</code> , <code>BroadcastUpdate(data MetricData)</code>
Visualization Rendering	Generate charts from time-series data	<code>RenderChart(data []TimeSeries, chartType ChartType)</code> <code>ChartData</code>
Access Control	Manage user permissions for viewing and editing dashboards	<code>CheckPermission(user User, dashboard DashboardID, action Action) bool</code>

The **Alerting System** continuously monitors metrics data against user-defined rules and manages the complete alert lifecycle from detection through resolution notification. It evaluates alert conditions at regular intervals, maintains alert state across restarts, and delivers notifications through multiple channels with proper rate limiting and escalation policies. The alerting system also provides alert silencing capabilities for maintenance windows and implements intelligent grouping to reduce notification noise.

Alerting Responsibility	Description	Key Interfaces
Rule Evaluation	Periodically check metric values against alert conditions	<code>EvaluateRule(rule AlertRule) AlertState</code> , <code>ScheduleEvaluation(rule AlertRule)</code>
State Management	Track alert states and manage transitions between states	<code>UpdateAlertState(alert AlertID, state AlertState)</code> , <code>GetAlertHistory(alert AlertID) []StateTransition</code>
Notification Delivery	Send alert messages through configured channels	<code>SendNotification(alert Alert, channel NotificationChannel) error</code>
Alert Grouping	Combine related alerts to reduce notification volume	<code>GroupAlerts(alerts []Alert) []AlertGroup</code> , <code>ShouldGroup(alert1, alert2 Alert) bool</code>

Design Insight: Clear Boundaries Enable Independent Evolution

By defining precise component responsibilities and stable interfaces, each component can evolve independently without breaking the entire system. The ingestion engine can add new protocol support, the storage engine can optimize its internal format, and the query engine can implement new aggregation functions, all without requiring changes to other components.

Data Flow Overview

Understanding how data flows through our metrics system is crucial for debugging issues, optimizing performance, and ensuring data consistency. The complete data journey follows several distinct paths depending on whether we're handling metric ingestion, dashboard queries, or alert evaluations.

Metric Ingestion Flow: The primary data flow begins when applications send metrics to our ingestion endpoints or when our system scrapes metrics from application endpoints. Raw metric data enters through HTTP requests containing JSON payloads or Prometheus exposition format text. The ingestion engine immediately performs validation, checking that metric names follow naming conventions, label values don't exceed cardinality limits, and timestamps fall within acceptable ranges. Valid metrics undergo preprocessing where label values are sanitized, metric names are normalized to lowercase, and any configured transformations are applied.

Processed metrics then flow to the storage engine through an internal queue that provides buffering and backpressure protection. The storage engine writes sample data to time-series blocks while simultaneously updating metadata indexes with new metric names and label combinations. Each metric sample generates a `SeriesID` that uniquely identifies the combination of metric name and labels, enabling efficient storage and retrieval of related data points.

Ingestion Stage	Input Format	Processing	Output Format
Protocol Reception	HTTP JSON/Prometheus text	Parse request body, extract metrics	<code>[]Metric</code> structs
Validation	<code>[]Metric</code>	Check names, labels, timestamps, cardinality	Valid <code>[]Metric</code>
Preprocessing	Valid <code>[]Metric</code>	Normalize names, sanitize labels	Processed <code>[]Metric</code>
Storage Writing	Processed <code>[]Metric</code>	Generate SeriesID, write samples and metadata	Persistent storage blocks

Dashboard Query Flow: When users view dashboards or create new visualizations, the dashboard component sends queries to the query engine requesting specific time-series data. These queries specify metric names, label filters, time ranges, and aggregation functions needed to generate chart data. The query engine parses each query string into a structured query object, then develops an execution plan that minimizes storage I/O by identifying which storage blocks contain relevant data.

Query execution retrieves raw samples from storage blocks, applies any necessary filtering based on label conditions, and performs mathematical aggregations like sum, average, or rate calculations across multiple time series. Results are formatted according to the client's requirements and sent back to the dashboard component, which renders them into interactive charts and updates browser clients through WebSocket connections.

Query Stage	Input	Processing	Output
Query Reception	Query string from dashboard	Parse into structured Query object	<code>Query</code> struct
Execution Planning	<code>Query</code> struct	Identify relevant storage blocks, optimize access order	<code>QueryPlan</code>
Data Retrieval	<code>QueryPlan</code>	Read samples from storage, apply label filters	Raw <code>[]Sample</code>
Aggregation	Raw <code>[]Sample</code>	Apply mathematical functions, align timestamps	<code>[]TimeSeries</code>
Result Formatting	<code>[]TimeSeries</code>	Convert to client format (JSON/CSV)	Formatted response

Alert Evaluation Flow: The alerting system runs on independent schedules, continuously evaluating alert rules against live metrics data. Each alert rule defines a query expression, comparison operator, threshold value, and evaluation frequency. The alert evaluator sends these queries to the query engine using the same interfaces as dashboard queries, but typically focuses on recent data within the last few minutes or hours.

Query results undergo threshold comparison to determine if alert conditions are met. The alerting system maintains state machines for each alert rule, tracking transitions between inactive, pending, firing, and resolved states. State changes trigger notification delivery through configured channels like email, Slack, or PagerDuty, with message content generated from customizable templates that include metric values and trend information.

Alert Stage	Input	Processing	Output
Rule Scheduling	Timer trigger	Load active alert rules from configuration	[]AlertRule
Query Generation	AlertRule	Convert rule conditions into query expressions	Query
Condition Evaluation	Query results	Compare metric values against thresholds	AlertState
State Management	Current and new AlertState	Determine state transitions, update persistence	State change events
Notification Delivery	State change events	Generate messages, send through channels	External notifications

Architecture Decision: Event-Driven Component Communication

- Context:** Components need to coordinate without tight coupling, enabling independent scaling and deployment
- Options Considered:**
 1. Direct method calls between components
 2. Event-driven communication with message queues
 3. Shared database with polling
- Decision:** Event-driven communication with internal message queues
- Rationale:** Provides loose coupling, natural backpressure, and enables async processing without external dependencies
- Consequences:** Adds complexity but enables independent component scaling and improves system resilience

Deployment Architecture

The deployment architecture determines how our components are packaged, configured, and operated in production environments. Our design supports both single-node deployments for development and testing, as well as distributed deployments for production scale and high availability.

Single-Node Deployment: For development, testing, and small production environments, all components run within a single process as separate goroutines communicating through Go channels. This deployment model minimizes operational complexity while providing the full functionality of our metrics system. The single binary includes HTTP servers for ingestion and dashboard endpoints, background goroutines for alert evaluation, and embedded storage using local filesystem.

Configuration is provided through a single YAML file that defines ingestion endpoints, storage paths, dashboard settings, and alert rules. The process listens on multiple ports: 8080 for metric ingestion, 3000 for dashboard web interface, and 9090 for health checks and administrative operations. Internal communication between components uses Go channels for metric data flow and shared memory for configuration and state information.

Deployment Component	Process	Ports	Storage
Ingestion Engine	Main process, goroutine pool	8080 (HTTP)	In-memory queues
Storage Engine	Main process, background workers	N/A (internal)	Local filesystem blocks
Query Engine	Main process, shared service	N/A (internal)	Shared memory caches
Dashboard	Main process, HTTP server	3000 (HTTP/WebSocket)	SQLite for configs
Alerting System	Main process, cron scheduler	N/A (internal)	Local state files

Multi-Node Deployment: Production environments requiring higher throughput and availability can deploy components as separate services communicating over HTTP APIs. The ingestion engine runs as a stateless service that can be horizontally scaled

behind a load balancer. Multiple ingestion instances write to a shared storage layer that coordinates writes through file locking and atomic operations.

The storage engine operates as a separate service with dedicated high-performance storage volumes and background processes for compaction and retention enforcement. Query engines run as stateless services that connect to storage over internal network interfaces, enabling horizontal scaling of query processing. Dashboard instances share configuration through an external database, while alerting systems use distributed coordination to prevent duplicate alert evaluations.

Service discovery enables components to find each other dynamically as instances are added or removed. Health checks monitor each service independently, and failures trigger automatic restarts or traffic redirection to healthy instances.

Service	Scaling Model	Dependencies	Communication
Ingestion Service	Horizontal (stateless)	Storage service	HTTP POST to storage
Storage Service	Vertical (shared state)	Persistent volumes	HTTP API for queries
Query Service	Horizontal (stateless)	Storage service	HTTP API calls
Dashboard Service	Horizontal (shared DB)	External database	HTTP + WebSocket
Alerting Service	Active/standby	Query service	HTTP queries, webhook notifications

Container Deployment: Both single-node and multi-node deployments support containerization using Docker images. The single-node deployment produces one image containing all components, while multi-node deployment creates separate images for each service. Container images include all necessary dependencies and use multi-stage builds to minimize image size.

Kubernetes deployment manifests define resource requirements, health checks, and scaling policies for each component. Persistent volumes ensure storage durability across container restarts, while ConfigMaps and Secrets manage configuration and sensitive credentials. Service meshes like Istio can provide additional features like traffic encryption, request tracing, and advanced load balancing.

Container Type	Image Size	Resource Requirements	Persistent Storage
Single-node	~50MB	512MB RAM, 1 CPU	10GB volume
Ingestion service	~20MB	256MB RAM, 0.5 CPU	None (stateless)
Storage service	~30MB	2GB RAM, 2 CPU	100GB+ volume
Query service	~25MB	1GB RAM, 1 CPU	None (stateless)
Dashboard service	~35MB	512MB RAM, 0.5 CPU	External database
Alerting service	~20MB	256MB RAM, 0.5 CPU	Small state volume

Design Principle: Deployment Flexibility Without Architecture Compromise

Our component architecture naturally supports multiple deployment models without requiring architectural changes. The same codebase can run as a single process for development or as distributed services for production, with deployment-specific configuration determining communication mechanisms.

Configuration Management: Deployment architecture includes comprehensive configuration management that adapts to different environments while maintaining consistency. Configuration is organized hierarchically with global defaults, environment-specific overrides, and runtime parameters. The single-node deployment uses local configuration files, while distributed deployments can integrate with configuration management systems like Consul or etcd.

Environment variables provide deployment-specific parameters like network addresses, credentials, and resource limits. Configuration validation occurs at startup to catch errors before services begin processing data. Hot configuration reloading enables operational changes without service restarts for non-critical settings like dashboard themes or alert message templates.

Configuration Layer	Source	Scope	Reload Support
Global Defaults	Compiled into binary	All deployments	Requires restart
Environment Config	YAML files or config service	Per environment	Hot reload
Runtime Parameters	Environment variables	Per instance	Hot reload
User Preferences	Database or local storage	Per user	Immediate

Common Pitfalls

⚠ Pitfall: Tight Component Coupling Through Direct Database Access Multiple components directly accessing the same database tables creates hidden dependencies and makes it difficult to evolve components independently. For example, if both the query engine and alerting system directly read from the same storage tables, changes to the storage schema require coordinating updates across multiple components. Instead, all data access should flow through well-defined service interfaces that can evolve their internal implementations without breaking clients.

⚠ Pitfall: Ignoring Backpressure in Data Flow Design Designing data flows without considering what happens when downstream components can't keep up leads to memory exhaustion and data loss. A common mistake is having the ingestion engine accept unlimited metrics without checking if the storage engine can handle the write load. This causes metric data to accumulate in memory until the process crashes. Always implement queue limits, circuit breakers, and graceful degradation when downstream services are overloaded.

⚠ Pitfall: Mixing Configuration and State in Component Design Storing configuration data (like dashboard definitions) in the same storage systems as operational data (like metric samples) creates operational complexity and scaling problems. Configuration data has different access patterns, consistency requirements, and backup needs than time-series data. Keep them separated with configuration using databases optimized for transactional consistency and operational data using storage optimized for time-series workloads.

⚠ Pitfall: Single Points of Failure in Distributed Architecture While distributed deployment provides scalability benefits, introducing single points of failure eliminates availability advantages. Common mistakes include having only one storage service instance, sharing state through a single external database, or requiring all components to register with a single service discovery instance. Design redundancy into every component that stores state, and ensure the system can operate with reduced functionality when individual components fail.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	Go <code>net/http</code> with <code>gorilla/mux</code> router	Go <code>gin-gonic/gin</code> or <code>echo</code> framework
Inter-Component Communication	Go channels for single-node, HTTP for multi-node	gRPC with Protocol Buffers
Configuration Management	YAML files with <code>gopkg.in/yaml.v3</code>	Consul or etcd integration
Service Discovery	Static configuration files	Kubernetes DNS or Consul
Health Checks	Simple HTTP endpoints returning status	Structured health checks with dependency status
Logging	Go <code>log/slog</code> with structured output	External logging systems like ELK or Prometheus logs

Recommended File Structure

```
metrics-system/
├── cmd/
│   ├── single-node/
│   │   └── main.go          ← Single-process deployment
│   └── distributed/
│       ├── ingestion/main.go    ← Ingestion service
│       ├── storage/main.go      ← Storage service
│       ├── query/main.go        ← Query service
│       ├── dashboard/main.go    ← Dashboard service
│       └── alerting/main.go     ← Alerting service
└── internal/
    ├── ingestion/
    │   ├── server.go           ← HTTP handlers and validation
    │   ├── processor.go         ← Metric preprocessing logic
    │   └── ingestion_test.go
    ├── storage/
    │   ├── engine.go            ← Storage interface implementation
    │   ├── blocks.go             ← Time-series block management
    │   └── storage_test.go
    ├── query/
    │   ├── engine.go            ← Query processing logic
    │   ├── parser.go             ← Query language parsing
    │   └── query_test.go
    ├── dashboard/
    │   ├── server.go            ← Web server and WebSocket handling
    │   ├── config.go             ← Dashboard configuration management
    │   └── dashboard_test.go
    ├── alerting/
    │   ├── evaluator.go          ← Alert rule evaluation
    │   ├── notifier.go            ← Notification delivery
    │   └── alerting_test.go
    └── shared/
        ├── types.go              ← Common data structures
        ├── config.go              ← Configuration loading
        └── health.go               ← Health check utilities
    ├── web/
    │   ├── static/                ← Dashboard HTML, CSS, JavaScript
    │   └── templates/              ← Go templates for dashboard rendering
    └── configs/
        ├── single-node.yaml        ← Single-node configuration
        └── distributed/              ← Per-service configurations
└── docker/
    ├── Dockerfile.single          ← Single-node container
    └── Dockerfile.service         ← Multi-stage service containers
```

Infrastructure Starter Code

The following code provides complete, working infrastructure that handles cross-cutting concerns so you can focus on the core metrics system logic:

Configuration Management (`internal/shared/config.go`):

```
package shared

import (
    "fmt"
    "os"
    "time"

    "gopkg.in/yaml.v3"
)

// Config represents the complete system configuration with all component settings.

// Load this once at startup and pass relevant sections to each component.

type Config struct {
    Server   ServerConfig   `yaml:"server"`
    Storage  StorageConfig  `yaml:"storage"`
    Alerting AlertingConfig `yaml:"alerting"`
}

type ServerConfig struct {

    IngestionPort int          `yaml:"ingestion_port"`
    DashboardPort int          `yaml:"dashboard_port"`
    ReadTimeout   time.Duration `yaml:"read_timeout"`
    WriteTimeout  time.Duration `yaml:"write_timeout"`
}

type StorageConfig struct {

    DataDir        string      `yaml:"data_dir"`
    RetentionPeriod time.Duration `yaml:"retention_period"`
    CompactionInterval time.Duration `yaml:"compaction_interval"`
}

type AlertingConfig struct {

    EvaluationInterval time.Duration `yaml:"evaluation_interval"`
    NotificationChannels []NotificationChannel `yaml:"notification_channels"`
}
```

GO

```

}

type NotificationChannel struct {

    Type string          `yaml:"type" // "email", "slack", "webhook"`

    Name string          `yaml:"name"`

    Config map[string]string `yaml:"config"`

}

// LoadConfig reads configuration from the specified file path with environment variable overrides.

// Returns a complete Config struct ready for use by all components.

func LoadConfig(configPath string) (*Config, error) {

    data, err := os.ReadFile(configPath)

    if err != nil {

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

    }

    var config Config

    if err := yaml.Unmarshal(data, &config); err != nil {

        return nil, fmt.Errorf("failed to parse config: %w", err)

    }

    // Apply environment variable overrides

    if port := os.Getenv("INGESTION_PORT"); port != "" {

        fmt.Sscanf(port, "%d", &config.Server.IngestionPort)

    }

    if dataDir := os.Getenv("STORAGE_DATA_DIR"); dataDir != "" {

        config.Storage.DataDir = dataDir

    }

    return &config, nil

}

// Validate checks that all required configuration values are present and valid.

```

```
// Call this after loading configuration to catch errors before starting services.

func (c *Config) Validate() error {

    if c.Server.IngestionPort <= 0 || c.Server.IngestionPort > 65535 {

        return fmt.Errorf("invalid ingestion port: %d", c.Server.IngestionPort)
    }

    if c.Storage.DataDir == "" {

        return fmt.Errorf("storage data directory must be specified")
    }

    if c.Storage.RetentionPeriod <= 0 {

        return fmt.Errorf("retention period must be positive")
    }

    return nil
}
```

Health Check System ([internal/shared/health.go](#)):

```
package shared GO

import (
    "context"
    "encoding/json"
    "net/http"
    "sync"
    "time"
)

// HealthStatus represents the health state of a component or dependency.

type HealthStatus int

const (
    HealthStatusHealthy HealthStatus = iota
    HealthStatusDegraded
    HealthStatusUnhealthy
)

func (h HealthStatus) String() string {
    switch h {
    case HealthStatusHealthy:
        return "healthy"
    case HealthStatusDegraded:
        return "degraded"
    case HealthStatusUnhealthy:
        return "unhealthy"
    default:
        return "unknown"
    }
}

// HealthCheck represents a single health check with its current status.

type HealthCheck struct {
```

```

Name      string          `json:"name"`

Status    HealthStatus    `json:"status"`

LastChecked time.Time     `json:"last_checked"`

Message   string          `json:"message,omitempty"`

CheckFunc func(ctx context.Context) (HealthStatus, string) `json:"-"`
}

// HealthManager coordinates health checks across all system components.

// Each component registers its health checks, and the manager periodically executes them.

type HealthManager struct {

    checks map[string]*HealthCheck

    mu     sync.RWMutex
}

// NewHealthManager creates a new health manager ready for check registration.

func NewHealthManager() *HealthManager {

    return &HealthManager{

        checks: make(map[string]*HealthCheck),
    }
}

// RegisterCheck adds a health check that will be executed periodically.

// The checkFunc should return the current health status and an optional message.

func (hm *HealthManager) RegisterCheck(name string, checkFunc func(ctx context.Context) (HealthStatus, string)) {

    hm.mu.Lock()

    defer hm.mu.Unlock()

    hm.checks[name] = &HealthCheck{

        Name:      name,
        Status:    HealthStatusUnhealthy, // Start unhealthy until first check
        CheckFunc: checkFunc,
    }
}

```

```
// RunChecks executes all registered health checks and updates their status.

// Call this periodically (e.g., every 30 seconds) to keep health status current.

func (hm *HealthManager) RunChecks(ctx context.Context) {

    hm.mu.Lock()

    defer hm.mu.Unlock()

    for _, check := range hm.checks {

        status, message := check.CheckFunc(ctx)

        check.Status = status

        check.Message = message

        check.LastChecked = time.Now()

    }

}

// GetOverallStatus returns the worst status among all checks.

// System is healthy only if all checks are healthy.

func (hm *HealthManager) GetOverallStatus() HealthStatus {

    hm.mu.RLock()

    defer hm.mu.RUnlock()

    worst := HealthStatusHealthy

    for _, check := range hm.checks {

        if check.Status > worst {

            worst = check.Status

        }

    }

    return worst

}

// ServeHTTP implements http.Handler to provide health check endpoint.

// Returns 200 for healthy, 503 for degraded/unhealthy with JSON details.

func (hm *HealthManager) ServeHTTP(w http.ResponseWriter, r *http.Request) {
```

```

hm.mu.RLock()

checks := make(map[string]*HealthCheck, len(hm.checks))

for name, check := range hm.checks {
    // Copy struct to avoid exposing internal pointers
    checkCopy := *check
    checkCopy.CheckFunc = nil // Don't serialize function
    checks[name] = &checkCopy
}

hm.mu.RUnlock()

overall := hm.GetOverallStatus()

response := struct {
    Status string `json:"status"`
    Checks map[string]*HealthCheck `json:"checks"`
}{

    Status: overall.String(),
    Checks: checks,
}

w.Header().Set("Content-Type", "application/json")

if overall != HealthStatusHealthy {
    w.WriteHeader(http.StatusServiceUnavailable)
}
}

json.NewEncoder(w).Encode(response)
}

```

Core Architecture Skeleton

The following code provides the main architectural structure with detailed TODOs mapping to the design concepts above:

Component Interface Definitions (`internal/shared/interfaces.go`):

```
package shared

import (
    "context"
    "time"
)

// MetricsIngestor defines the interface for receiving and processing metrics data.

// Implement this interface in your ingestion engine component.

type MetricsIngestor interface {

    // IngestMetrics processes a batch of metrics from push-based sources.

    // TODO 1: Validate each metric using Metric.Validate()

    // TODO 2: Check label cardinality to prevent explosion

    // TODO 3: Apply any configured preprocessing transformations

    // TODO 4: Send processed metrics to storage engine via queue

    // TODO 5: Return error if any metrics were rejected

    IngestMetrics(ctx context.Context, metrics []Metric) error

    // ScrapeEndpoint retrieves metrics from a pull-based Prometheus endpoint.

    // TODO 1: Make HTTP GET request to the provided URL

    // TODO 2: Parse Prometheus exposition format response

    // TODO 3: Convert parsed data to internal Metric structs

    // TODO 4: Call IngestMetrics() with converted metrics

    ScrapeEndpoint(ctx context.Context, url string) error
}

// TimeSeriesStorage defines the interface for persisting and retrieving time-series data.

// Implement this interface in your storage engine component.

type TimeSeriesStorage interface {

    // WriteSamples persists metric samples to storage blocks.

    // TODO 1: Generate SeriesID for each unique metric name + labels combination

    // TODO 2: Append samples to appropriate time-series blocks

    // TODO 3: Update metadata indexes with new series information
}
```

GO

```

// TODO 4: Trigger compaction if block size thresholds exceeded

// TODO 5: Return error if write operation fails

WriteSamples(ctx context.Context, samples []Sample) error


// QueryRange retrieves samples within the specified time range.

// TODO 1: Parse label selectors to identify matching series

// TODO 2: Find storage blocks that overlap with time range

// TODO 3: Read samples from blocks, applying label filters

// TODO 4: Sort samples by timestamp and return

QueryRange(ctx context.Context, query RangeQuery) ([]TimeSeries, error)


// GetSeriesMetadata returns metadata for series matching label selectors.

// TODO 1: Consult metadata indexes for matching series

// TODO 2: Return series names and label sets without sample data

GetSeriesMetadata(ctx context.Context, selectors []LabelSelector) ([]SeriesMetadata, error)

}

// QueryProcessor defines the interface for executing queries against time-series data.

// Implement this interface in your query engine component.

type QueryProcessor interface {

    // ExecuteQuery parses and executes a query string, returning formatted results.

    // TODO 1: Parse query string into Query struct

    // TODO 2: Validate query syntax and semantic correctness

    // TODO 3: Create execution plan optimized for storage access

    // TODO 4: Execute plan against storage engine

    // TODO 5: Apply aggregation functions and mathematical operations

    // TODO 6: Format results according to requested output format

    ExecuteQuery(ctx context.Context, queryString string) (QueryResult, error)

}

// AlertEvaluator defines the interface for processing alert rules and managing alert state.

// Implement this interface in your alerting system component.

type AlertEvaluator interface {

```

```
// EvaluateRules checks all active alert rules against current metric data.

// TODO 1: Load active alert rules from configuration

// TODO 2: Convert each rule condition to query expression

// TODO 3: Execute query using QueryProcessor interface

// TODO 4: Compare results against rule thresholds

// TODO 5: Update alert states based on comparison results

// TODO 6: Trigger notifications for state changes

EvaluateRules(ctx context.Context) error

// UpdateAlertState changes the state of an alert and triggers notifications.

// TODO 1: Load current alert state from persistence

// TODO 2: Validate state transition is allowed

// TODO 3: Update persistent state storage

// TODO 4: Generate notification message from template

// TODO 5: Send notification through configured channels

UpdateAlertState(ctx context.Context, alertID string, newState AlertState) error

}
```

Component Coordinator (`internal/coordinator/coordinator.go`):

```
package coordinator

import (
    "context"
    "fmt"
    "log/slog"
    "sync"
    "time"

    "metrics-system/internal/shared"
)

// ComponentCoordinator manages the lifecycle of all system components and coordinates
// their interactions through well-defined interfaces.

type ComponentCoordinator struct {
    config      *shared.Config
    logger      *slog.Logger
    health      *shared.HealthManager
    components map[string]Component
    wg          sync.WaitGroup
    ctx         context.Context
    cancel      context.CancelFunc
}

// Component represents a system component that can be started and stopped independently.

type Component interface {
    Start(ctx context.Context) error
    Stop(ctx context.Context) error
    Name() string
    HealthCheck(ctx context.Context) (shared.HealthStatus, string)
}

// NewCoordinator creates a new component coordinator with the provided configuration.
// Use this as the main orchestrator for both single-node and distributed deployments.
```

GO

```
func NewCoordinator(config *shared.Config, logger *slog.Logger) *ComponentCoordinator {
    ctx, cancel := context.WithCancel(context.Background())

    return &ComponentCoordinator{
        config:      config,
        logger:      logger,
        health:      shared.NewHealthManager(),
        components: make(map[string]Component),
        ctx:         ctx,
        cancel:      cancel,
    }
}

// RegisterComponent adds a component to the coordinator for lifecycle management.

// TODO 1: Validate component name is unique

// TODO 2: Store component in components map

// TODO 3: Register component's health check with health manager

// TODO 4: Log component registration for debugging

func (cc *ComponentCoordinator) RegisterComponent(component Component) error {
    // TODO: Implement component registration logic here

    // This should store the component and set up its health monitoring
    panic("implement me")
}

// Start begins all registered components in dependency order.

// TODO 1: Start components in order: Storage -> Query -> Ingestion -> Dashboard -> Alerting

// TODO 2: Wait for each component to report healthy before starting next

// TODO 3: Start health check monitoring goroutine

// TODO 4: Set up signal handlers for graceful shutdown

func (cc *ComponentCoordinator) Start() error {
    // TODO: Implement startup sequence here

    // Consider component dependencies and startup ordering
    panic("implement me")
}
```

```

}

// Stop gracefully shuts down all components in reverse dependency order.

// TODO 1: Cancel context to signal all components to stop

// TODO 2: Stop components in reverse order with timeout

// TODO 3: Wait for all component goroutines to exit

// TODO 4: Close any shared resources

func (cc *ComponentCoordinator) Stop() error {

    // TODO: Implement shutdown sequence here

    // Ensure graceful cleanup of all resources

    panic("implement me")

}

// HealthHandler returns an HTTP handler for system health checks.

func (cc *ComponentCoordinator) HealthHandler() http.Handler {

    return cc.health

}

```

Milestone Checkpoints

After implementing component interfaces:

- Run `go build ./cmd/single-node/` - should compile without errors
- All interface methods should be defined with proper signatures
- Component registration should work without panics

After implementing basic coordinator:

- Start the system with `./single-node -config configs/single-node.yaml`
- Health check endpoint at `http://localhost:9090/health` should return component status
- System should shut down gracefully with CTRL+C

After connecting components:

- Send a test metric: `curl -X POST localhost:8080/metrics -d '{"name":"test_metric","value":42}'`
- Query the metric: `curl "localhost:8080/query?query=test_metric"`
- Check dashboard at `http://localhost:3000` shows the metric

Data Model

Milestone(s): 1 (Metrics Collection), 2 (Storage & Querying), 3 (Visualization Dashboard), 4 (Alerting System) - The data model provides the foundational structures used across all system components

The Blueprint Metaphor: Understanding System Data Models

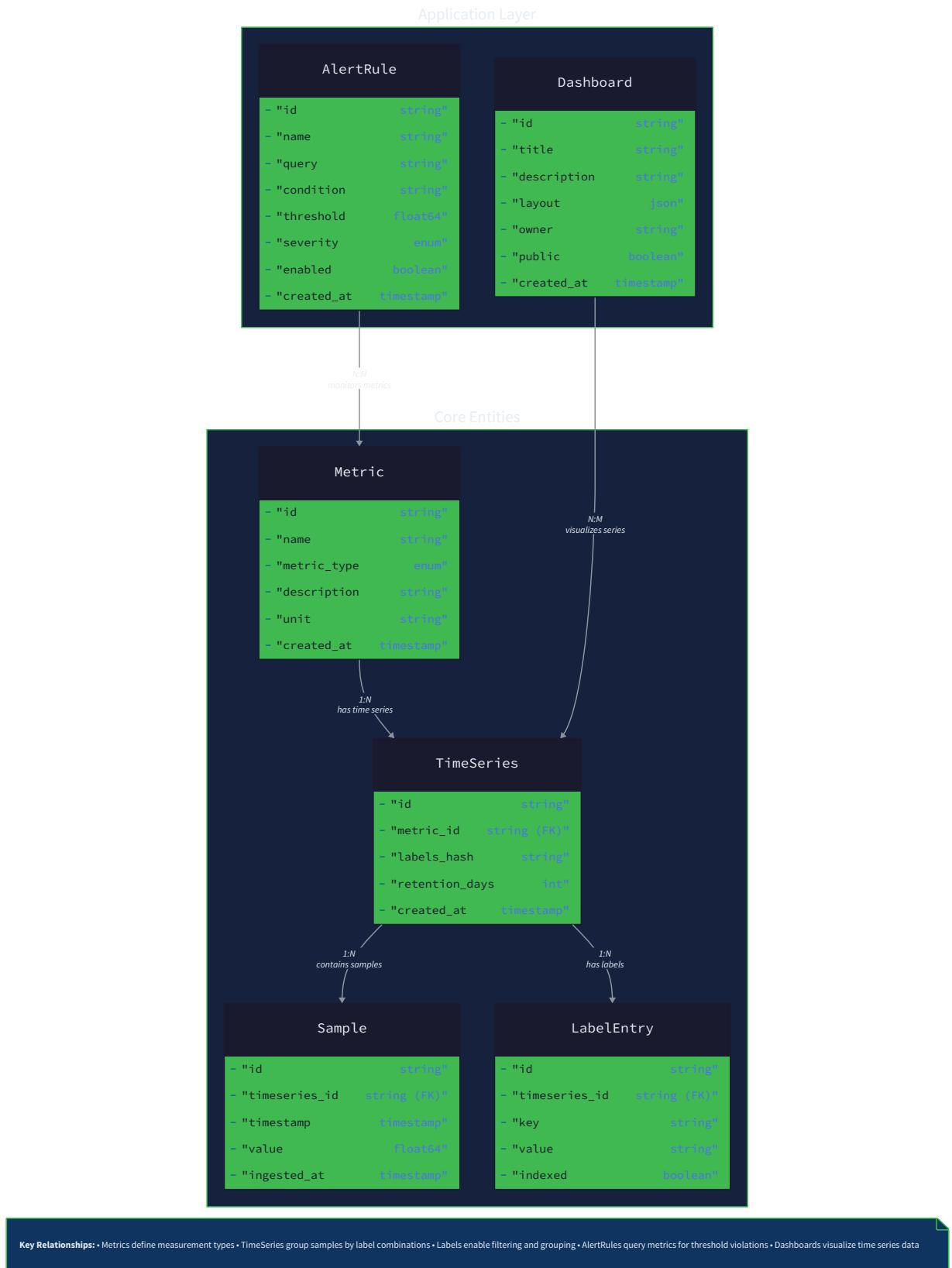
Think of a data model as the blueprint for a complex building project. Just as architectural blueprints define how rooms connect, what materials to use, and how utilities flow through the structure, our data model defines how metrics, time-series data, alerts, and dashboards are structured and relate to each other. A well-designed blueprint ensures that electricians, plumbers, and carpenters can work independently while their components integrate seamlessly. Similarly, our data model ensures that the ingestion engine, storage layer, query processor, and alerting system can operate independently while sharing a common understanding of data structures.

The data model serves as the contract between all system components. When the ingestion engine receives a metric, it must understand the difference between a counter that only increases and a gauge that can fluctuate. When the storage engine persists this data, it must organize time-series efficiently for later retrieval. When the alerting system evaluates rules, it must understand how to extract values from different metric types. This shared understanding prevents integration headaches and ensures data consistency across the entire system.

Metric Types and Structure

Understanding metric types requires thinking about what each type measures in the real world. A **counter** behaves like an odometer in a car - it only moves forward, tracking cumulative totals like total HTTP requests served or total bytes transmitted. A **gauge** behaves like a speedometer or thermometer - it shows the current value of something that can increase or decrease, like current CPU usage or active database connections. A **histogram** behaves like a survey response collector - it groups observations into ranges (buckets) to show distribution patterns, like response time distributions or request size distributions.

Each metric type has fundamentally different mathematical properties that affect how they are stored, queried, and aggregated. Counters require rate calculations to be meaningful (requests per second rather than total requests since boot). Gauges can be averaged, but their historical values may not be meaningful for trend analysis. Histograms require special aggregation logic to merge buckets across multiple instances or time windows.



The core metric structure must capture not just the value and timestamp, but also the rich metadata that makes metrics searchable and aggregatable. **Labels** provide this multi-dimensional metadata, allowing metrics to be sliced and diced across different attributes like service name, environment, region, or instance ID. However, labels create a fundamental tension: more labels

provide more flexibility but exponentially increase **cardinality** (the number of unique time-series combinations), which directly impacts storage requirements and query performance.

Decision: Metric Structure Design

- **Context:** Need to represent different types of metrics with their metadata while supporting efficient storage and querying
- **Options Considered:**
 1. Single unified metric structure with type discrimination
 2. Separate data structures for each metric type
 3. Inheritance-based hierarchy with base metric class
- **Decision:** Single unified structure with `MetricType` enum discrimination
- **Rationale:** Simplifies ingestion pipeline and storage layer by having consistent interfaces, while type-specific behavior is handled through switch statements rather than complex inheritance
- **Consequences:** Enables polymorphic handling in storage and query layers, but requires careful validation to ensure type-specific constraints are enforced

Field Name	Type	Description
Name	string	Metric identifier following hierarchical naming convention (e.g., "http_requests_total", "cpu_usage_percent")
Type	<code>MetricType</code>	Enum indicating Counter, Gauge, or Histogram behavior and aggregation rules
Labels	Labels	Key-value metadata map for multi-dimensional filtering and grouping
Samples	[]Sample	Ordered array of timestamped values representing the metric's evolution over time

The `MetricType` enumeration defines the fundamental behavior categories that determine how values should be interpreted and aggregated:

Metric Type	Numeric Value	Aggregation Behavior	Typical Use Cases
Counter	0	Rate calculations, cumulative sums	Request counts, error counts, bytes transmitted
Gauge	1	Point-in-time snapshots, averages	CPU usage, memory consumption, queue depth
Histogram	2	Bucket merging, percentile calculations	Response times, request sizes, batch processing durations

The `Labels` type provides powerful multi-dimensional indexing but requires careful cardinality management. Each unique combination of label values creates a distinct time-series, so labels with high cardinality values (like user IDs or request IDs) can create storage explosions. The system must provide both flexibility for legitimate use cases and protection against accidental cardinality bombs.

Label Constraint	Validation Rule	Purpose
Key Format	Must match <code>^[a-zA-Z_][a-zA-Z0-9_]*\$</code>	Ensures consistent naming and prevents injection attacks
Key Length	Maximum 128 characters	Prevents excessive memory usage in indexes
Value Length	Maximum 1024 characters	Balances expressiveness with storage efficiency
Total Labels	Maximum 32 per metric	Limits combinatorial explosion while supporting rich metadata
Reserved Keys	Cannot start with <code>__</code> (double underscore)	Reserves namespace for system-generated labels

The `Sample` structure captures the atomic unit of time-series data - a value paired with its observation timestamp. This pairing is fundamental to time-series analysis, enabling trend detection, rate calculations, and temporal aggregations.

Field Name	Type	Description
Value	float64	Numeric measurement value with IEEE 754 double precision for wide range support
Timestamp	time.Time	UTC timestamp with nanosecond precision for high-resolution time-series

⚠ Pitfall: Timestamp Precision Inconsistency Many beginners mix timestamp precisions (seconds vs milliseconds vs nanoseconds) across different components, leading to data alignment issues during queries. Always normalize to nanosecond precision at ingestion time and use UTC timezone to avoid daylight savings time complications. The storage engine can downsample precision during compaction, but ingestion should preserve maximum precision.

Time-Series Schema

Time-series data organization follows the principle that **storage locality drives query performance**. Think of time-series storage like organizing a vast library where books (data points) must be found quickly based on multiple criteria: author (metric name), topic (labels), and publication date (timestamp). The organization scheme determines whether finding all books by a specific author from a specific year requires scanning the entire library or jumping directly to the right shelf.

The fundamental insight is that time-series data exhibits strong **temporal locality** - queries typically request ranges of data points for the same metric and label combination. This access pattern drives the storage schema design toward grouping samples by time-series identity rather than by timestamp globally.

Decision: Time-Series Identification Strategy

- **Context:** Need to uniquely identify each time-series for efficient storage and retrieval while supporting label-based filtering
- **Options Considered:**
 1. Composite primary key of (metric_name, sorted_labels, timestamp)
 2. Hash-based series ID with separate label index
 3. Hierarchical key structure with metric name prefix
- **Decision:** Hash-based series ID with deterministic label sorting
- **Rationale:** Provides O(1) lookup performance while maintaining deterministic behavior across system restarts, and allows label indexing to be optimized separately from time-series storage
- **Consequences:** Enables efficient range scans within a time-series while supporting complex label queries through secondary indexes, but requires careful hash collision handling

Schema Component	Structure	Purpose
Series ID	SHA-256 hash of metric name + sorted labels	Provides unique, deterministic identifier for each time-series
Label Index	Inverted index mapping label keys/values to series IDs	Enables efficient label-based filtering and selection
Sample Blocks	Compressed chunks of samples grouped by series ID and time range	Optimizes storage efficiency and range query performance
Metadata Index	Series ID → metric metadata mapping	Provides series discovery and label enumeration

The **Series ID generation** algorithm ensures consistent identification across system components and restarts:

1. Concatenate metric name with newline separator
2. Sort label pairs by key name lexicographically
3. Append each sorted label as "key=value\n"
4. Compute SHA-256 hash of resulting string
5. Encode hash as hexadecimal string for human readability

This deterministic approach ensures that the same metric with identical labels always produces the same series ID, regardless of label order in the original submission or which system component processes it.

The **Label Index** provides efficient multi-dimensional filtering by maintaining inverted indexes for each label key-value pair. This allows queries like "find all HTTP request metrics for the production environment" to be resolved through index lookups rather than full table scans.

Index Type	Key Format	Value Format	Use Case
Label Key Index	<code>__key__: {label_key}</code>	Set of series IDs	Find all series with specific label keys
Label Value Index	<code>__kv__: {label_key}={label_value}</code>	Set of series IDs	Find all series with specific label key-value pairs
Metric Name Index	<code>__name__: {metric_name}</code>	Set of series IDs	Find all series for a specific metric
Series Metadata	<code>__meta__: {series_id}</code>	Metric name + labels JSON	Resolve series ID back to human-readable metadata

Sample Blocks organize the actual time-series data for optimal storage and retrieval. Each block contains samples for a single series within a specific time range, enabling efficient range queries and compression.

Block Component	Format	Purpose
Block Header	Series ID + time range + sample count	Enables range filtering without decompressing block data
Timestamp Array	Delta-compressed timestamps	Stores sample timestamps with compression for temporal locality
Value Array	Float64 values with optional compression	Stores metric values with optional bit-packing for integer-like values
Block Footer	Checksum + compression metadata	Ensures data integrity and provides decompression parameters

The storage layout prioritizes **write efficiency** for recent data while maintaining **read efficiency** for historical queries. Recent samples are accumulated in memory-based blocks that are periodically flushed to disk. Historical data is organized in immutable,

compressed blocks that can be efficiently scanned for range queries.

The key insight is that time-series workloads exhibit a "hot-warm-cold" access pattern. Recent data (last few hours) receives frequent writes and reads. Medium-aged data (days to weeks) receives occasional reads but no writes. Ancient data (months to years) receives rare reads but consumes the most storage space. The schema optimizes for each access pattern differently.

⚠️ Pitfall: Timestamp Alignment Assumptions Beginning developers often assume that samples from different metrics arrive with perfectly aligned timestamps, leading to complex join logic that fails in practice. Real-world metrics arrive with jittered timestamps due to network delays, processing variations, and clock skew between systems. Design aggregation and comparison logic to handle timestamp misalignment gracefully, typically by bucketing samples into time windows rather than expecting exact timestamp matches.

Alert and Dashboard Schema

The alerting and dashboard schemas must capture not just the static configuration but also the dynamic state transitions and user interactions that occur during system operation. Think of alert rules like smoke detectors - they must define what constitutes danger (threshold conditions), how to confirm it's not a false alarm (evaluation duration), and who to notify when action is required.

Dashboard configurations are like mission control displays - they must define what data to show, how to visualize it, and how to keep the information current.

Alert rule design balances expressiveness with reliability. Rules must be simple enough for operators to understand and debug, yet powerful enough to detect complex failure conditions. The schema captures both the declarative rule definition and the runtime state management required for proper alert lifecycle handling.

Decision: Alert Rule Expression Language

- **Context:** Need to define alert conditions that can reference metrics, apply mathematical operations, and specify thresholds while remaining debuggable
- **Options Considered:**
 1. Simple threshold-based rules with metric name + comparison operator
 2. SQL-like query language with aggregation functions
 3. PromQL-inspired expression language with time-series functions
- **Decision:** PromQL-inspired expressions with simplified function set
- **Rationale:** Provides sufficient expressiveness for complex conditions while maintaining compatibility with monitoring industry standards, and allows rule expressions to be tested independently of alert evaluation
- **Consequences:** Enables powerful alert conditions like rate calculations and multi-metric comparisons, but requires implementing a query parser and expression evaluator

Alert Rule Field	Type	Description
ID	string	Unique identifier for the alert rule, used for state tracking and notifications
Name	string	Human-readable alert name displayed in notifications and dashboard
Expression	string	Query expression that returns numeric values for threshold comparison
Threshold	float64	Numeric value that triggers alert when expression result crosses this boundary
Operator	string	Comparison operator: "gt" (greater than), "lt" (less than), "eq" (equal), "ne" (not equal)
Duration	time.Duration	How long condition must persist before transitioning from pending to firing state
EvaluationInterval	time.Duration	How frequently to evaluate the alert expression against current data
Labels	Labels	Static labels attached to all alert instances for routing and grouping
Annotations	map[string]string	Additional metadata like description, runbook URL, and dashboard links

Alert **state management** requires tracking the lifecycle of each alert instance as conditions change over time. The state machine ensures proper notification delivery and prevents alert flapping.

Alert State	Description	Entry Conditions	Exit Conditions
Inactive	Alert condition is not met	Expression result does not cross threshold	Expression result crosses threshold
Pending	Condition met but duration requirement not satisfied	Threshold crossed but duration timer not expired	Duration timer expires OR condition no longer met
Firing	Alert is actively triggering notifications	Duration requirement satisfied while condition remains true	Condition no longer met for resolution duration
Resolved	Alert has recovered from firing state	Firing alert condition returns to normal	Alert rule deleted OR condition becomes true again

Alert Instance Field	Type	Description
RuleID	string	References the parent alert rule configuration
InstanceLabels	Labels	Label values extracted from the metric data that triggered this instance
State	AlertState	Current state in the alert lifecycle (Inactive, Pending, Firing, Resolved)
StateChanged	time.Time	Timestamp when alert last transitioned between states
Value	float64	Most recent expression result value for debugging and display
FiredAt	*time.Time	Timestamp when alert first entered firing state (nil if never fired)
ResolvedAt	*time.Time	Timestamp when alert was resolved (nil if still active)

The **notification system** schema defines how alerts are delivered to external systems while handling delivery failures and rate limiting.

Notification Channel Field	Type	Description
Type	string	Channel type identifier: "email", "slack", "webhook", "pagerduty"
Name	string	Human-readable channel name for management and debugging
Config	map[string]string	Channel-specific configuration like URLs, tokens, and recipient lists
Enabled	bool	Whether this channel should receive notifications
RouteFilter	Labels	Label selector determining which alerts use this channel

Dashboard configuration captures the declarative specification of data visualizations while supporting real-time updates and user customization. The schema separates presentation logic from data queries to enable flexible visualization while maintaining performance.

Dashboard Field	Type	Description
ID	string	Unique dashboard identifier for URL routing and sharing
Title	string	Human-readable dashboard name displayed in browser title and navigation
Description	string	Optional dashboard description for documentation and search
Tags	[]string	Categorization tags for dashboard organization and discovery
Panels	[]Panel	Array of visualization panels arranged in the dashboard layout
TimeRange	TimeRange	Default time window for all panels unless overridden
RefreshInterval	time.Duration	How frequently panels should update their data automatically
Editable	bool	Whether dashboard configuration can be modified by viewers

Panel Field	Type	Description
ID	string	Unique panel identifier within the dashboard for layout and updates
Title	string	Panel title displayed above the visualization
Type	string	Visualization type: "line", "bar", "stat", "heatmap", "table"
GridPos	GridPosition	Panel position and size within the dashboard grid layout
Query	PanelQuery	Data source query configuration for this panel
Options	map[string]interface{}	Panel-specific configuration like axis labels, colors, and thresholds
AlertRule	*PanelAlertRule	Optional alert rule configuration tied to this panel's query

Panel Query Field	Type	Description
Expression	string	Time-series query expression to retrieve data for visualization
TimeRange	*TimeRange	Panel-specific time range override if different from dashboard default
RefreshInterval	*time.Duration	Panel-specific refresh rate override for high-frequency data
MaxDataPoints	int	Maximum number of data points to retrieve for performance and readability
MinInterval	time.Duration	Minimum step size for data aggregation to prevent over-resolution

⚠ Pitfall: Dashboard State Synchronization Developers often implement dashboard updates by polling for new data on a fixed interval, leading to unnecessary load and inconsistent user experience. Instead, implement event-driven updates where the query engine notifies dashboards when relevant metric data changes. This provides instant updates for rapidly changing metrics while avoiding unnecessary queries for stable metrics. Implement exponential backoff for panels that haven't changed recently to optimize resource usage.

⚠ Pitfall: Alert Flapping Alert rules that toggle rapidly between firing and resolved states create notification spam and erode operator confidence. This typically occurs when thresholds are set exactly at normal operating values or when evaluation intervals are too short relative to metric volatility. Implement hysteresis by requiring different thresholds for firing versus resolving (e.g., fire at 90% CPU but don't resolve until below 80%), and require minimum durations for both firing and resolution states.

Implementation Guidance

The data model implementation requires careful attention to serialization performance, validation consistency, and type safety across all system components. The following code structure provides a foundation for building robust data handling while maintaining flexibility for future enhancements.

Technology Recommendations:

Component	Simple Option	Advanced Option
Serialization	JSON with <code>encoding/json</code>	Protocol Buffers with <code>protobuf/proto</code>
Validation	Manual validation functions	Struct tags with <code>validator</code> library
Label Storage	Map with string keys/values	Interned strings with reference counting
Time Handling	<code>time.Time</code> with UTC normalization	Custom timestamp type with nanosecond precision
Hash Function	<code>crypto/sha256</code> for series IDs	<code>xxhash</code> for performance-critical paths

Recommended File Structure:

```
internal/model/
  types.go           ← Core data types (Metric, Sample, Labels)
  metric_types.go   ← MetricType enum and behavior
  series.go          ← Time-series identification and indexing
  alerts.go          ← Alert rule and state management types
  dashboard.go       ← Dashboard and panel configuration types
  validation.go      ← Input validation and constraint checking
  serialization.go   ← JSON/binary serialization helpers
  types_test.go       ← Comprehensive unit tests for all types
```

Core Types Infrastructure (Complete Implementation):

```
package model

import (
    "crypto/sha256"
    "fmt"
    "sort"
    "strings"
    "time"
)

// MetricType defines the fundamental behavior of a metric

type MetricType int

const (
    MetricTypeCounter    MetricType = 0
    MetricTypeGauge      MetricType = 1
    MetricTypeHistogram  MetricType = 2
)

func (mt MetricType) String() string {
    switch mt {
    case MetricTypeCounter:
        return "counter"
    case MetricTypeGauge:
        return "gauge"
    case MetricTypeHistogram:
        return "histogram"
    default:
        return "unknown"
    }
}

// Labels represents metric metadata with validation and indexing support

type Labels map[string]string
```

```

// Copy creates a deep copy of the labels map

func (l Labels) Copy() Labels {
    if l == nil {
        return nil
    }

    result := make(Labels, len(l))

    for k, v := range l {
        result[k] = v
    }

    return result
}

// String returns the canonical string representation for hashing and display

func (l Labels) String() string {
    if len(l) == 0 {
        return "{}"
    }

    pairs := make([]string, 0, len(l))

    for k, v := range l {
        pairs = append(pairs, fmt.Sprintf(`%s="%s"`, k, v))
    }

    sort.Strings(pairs)

    return "{" + strings.Join(pairs, ",") + "}"
}

// Sample represents a single timestamped measurement

type Sample struct {
    Value      float64     `json:"value"`
    Timestamp time.Time   `json:"timestamp"`
}

// Metric represents a named time-series with type and metadata

```

```

type Metric struct {

    Name      string      `json:"name"`

    Type     MetricType `json:"type"`

    Labels   Labels     `json:"labels"`

    Samples []Sample    `json:"samples"`

}

// SeriesID generates a unique, deterministic identifier for this time-series

func (m *Metric) SeriesID() string {

    // TODO 1: Create hash input string starting with metric name

    // TODO 2: Sort labels by key to ensure deterministic ordering

    // TODO 3: Append each label as "key=value\n" to hash input

    // TODO 4: Compute SHA-256 hash of the complete string

    // TODO 5: Return hexadecimal encoding of hash for readability

    // Hint: Use crypto/sha256 and fmt.Sprintf for hex encoding

}

// Validate checks metric data integrity and constraint compliance

func (m *Metric) Validate() error {

    // TODO 1: Validate metric name format (alphanumeric + underscore, no leading numbers)

    // TODO 2: Check metric type is one of the defined enum values

    // TODO 3: Validate each label key/value against length and format constraints

    // TODO 4: Ensure samples are sorted by timestamp (required for efficient storage)

    // TODO 5: Check for reasonable value ranges (no NaN/Inf for counters)

    // Hint: Use regular expressions for name/label validation

}

```

Alert and Dashboard Types (Skeleton Implementation):

```

// AlertRule defines conditions for triggering notifications

type AlertRule struct {

    ID          string      `json:"id"`

    Name        string      `json:"name"`

    Expression  string      `json:"expression"`

    Threshold   float64     `json:"threshold"`

    Operator    string      `json:"operator"`

    Duration    time.Duration `json:"duration"`

    EvaluationInterval time.Duration `json:"evaluation_interval"`

    Labels      Labels      `json:"labels"`

    Annotations map[string]string `json:"annotations"`

}

// Validate ensures alert rule configuration is valid and safe

func (ar *AlertRule) Validate() error {

    // TODO 1: Validate ID is non-empty and URL-safe

    // TODO 2: Check operator is one of: "gt", "lt", "eq", "ne"

    // TODO 3: Ensure duration and evaluation interval are positive

    // TODO 4: Validate expression syntax (implement basic PromQL parser)

    // TODO 5: Check threshold is finite (no NaN/Inf values)

}

// Dashboard represents a collection of metric visualization panels

type Dashboard struct {

    ID          string      `json:"id"`

    Title        string      `json:"title"`

    Description  string      `json:"description"`

    Tags         []string    `json:"tags"`

    Panels       []Panel     `json:"panels"`

    TimeRange    TimeRange   `json:"time_range"`

    RefreshInterval time.Duration `json:"refresh_interval"`

    Editable     bool        `json:"editable"`

}

```

```

// Panel represents a single visualization within a dashboard

type Panel struct {

    ID      string      `json:"id"`
    Title   string      `json:"title"`
    Type    string      `json:"type"`
    GridPos GridPosition `json:"grid_pos"`
    Query   PanelQuery  `json:"query"`
    Options  map[string]interface{} `json:"options"`
    AlertRule *PanelAlertRule `json:"alert_rule,omitempty"`
}

// TimeRange specifies absolute or relative time bounds for queries

type TimeRange struct {

    From time.Time `json:"from"`
    To   time.Time `json:"to"`
}

// GridPosition defines panel layout within dashboard grid

type GridPosition struct {

    X      int `json:"x"`      // Horizontal position (0-24)
    Y      int `json:"y"`      // Vertical position
    Width  int `json:"width"`  // Panel width (1-24)
    Height int `json:"height"` // Panel height in grid units
}

```

Validation and Serialization Helpers:

```
// ValidationError represents constraint violations during data validation

type ValidationError struct {
    Field  string `json:"field"`
    Value  string `json:"value"`
    Message string `json:"message"`
}

func (ve *ValidationError) Error() string {
    return fmt.Sprintf("validation failed for field '%s': %s", ve.Field, ve.Message)
}

// ValidateLabels checks label constraints for cardinality and format compliance

func ValidateLabels(labels Labels) error {
    // TODO 1: Check total label count against maximum limit (32)

    // TODO 2: Validate each key matches allowed pattern ^[a-zA-Z_][a-zA-Z0-9_]*$

    // TODO 3: Ensure keys don't start with __ (reserved for system labels)

    // TODO 4: Check key/value length limits (128/1024 characters)

    // TODO 5: Return ValidationError with specific constraint violations

    // Hint: Use regexp.MustCompile for pattern validation
}

// SerializeMetric converts metric to JSON with proper timestamp formatting

func SerializeMetric(m *Metric) ([]byte, error) {
    // TODO 1: Create a serialization wrapper with RFC3339 timestamp format

    // TODO 2: Convert time.Time values to standardized string representation

    // TODO 3: Use json.Marshal with proper error handling

    // TODO 4: Consider compression for large sample arrays

    // Hint: Create intermediate struct with string timestamps for JSON compatibility
}
```

Language-Specific Hints:

- Use `time.Time.UTC()` to normalize all timestamps and avoid timezone issues
- Implement `json.Marshaler` and `json.Unmarshaler` interfaces for custom timestamp formatting
- Use `sync.Pool` for label validation to avoid allocation overhead during high-throughput ingestion
- Consider using `unsafe.Pointer` for zero-copy string operations in label processing (advanced optimization)

- Use `sort.Slice` with stable sorting for deterministic label ordering in series ID generation

Milestone Checkpoint: After implementing the data model types, verify correctness with:

```
go test ./internal/model/...
```

BASH

Expected test coverage should validate:

- Series ID generation produces identical results for same metric+labels regardless of label order
- Label validation rejects malformed keys/values and enforces cardinality limits
- Metric validation catches type mismatches and constraint violations
- JSON serialization round-trips preserve all data with proper timestamp formatting
- Copy operations create true deep copies that don't share memory references

Signs of implementation issues:

- **Series ID inconsistency:** Same metric produces different IDs → check label sorting in `SeriesID()`
- **Validation bypass:** Invalid data passes validation → tighten constraint checking in `Validate()` methods
- **Memory leaks:** Label copying shares references → ensure `Copy()` creates independent map instances
- **Timestamp precision loss:** Nanosecond precision not preserved → use RFC3339Nano format in JSON serialization

Metrics Ingestion Engine

Milestone(s): 1 (Metrics Collection) - This section details the first major component that handles incoming metrics data with both push and pull collection models

Mental Model: The Data Processing Factory

Think of the metrics ingestion engine as a sophisticated **data processing factory** with multiple assembly lines working in parallel. Just like a modern manufacturing plant, our factory has several key characteristics:

Receiving Docks: The factory has two types of receiving areas. The **push dock** is like a loading bay where delivery trucks (applications) arrive at any time to drop off shipments of raw materials (metrics data). The **pull dock** operates more like a scheduled pickup service where factory workers (scrapers) drive out on regular routes to collect materials from various suppliers (application endpoints) according to a predetermined schedule.

Quality Control Stations: Before any raw materials enter the main production line, they pass through rigorous **quality control checkpoints**. Inspectors (validators) examine each shipment to ensure it meets specifications - checking that labels are properly formatted, metric types are valid, and that we're not receiving so many unique product variations that our storage systems would overflow. Materials that don't pass inspection are either rejected with detailed error reports or sent to a preprocessing station for standardization.

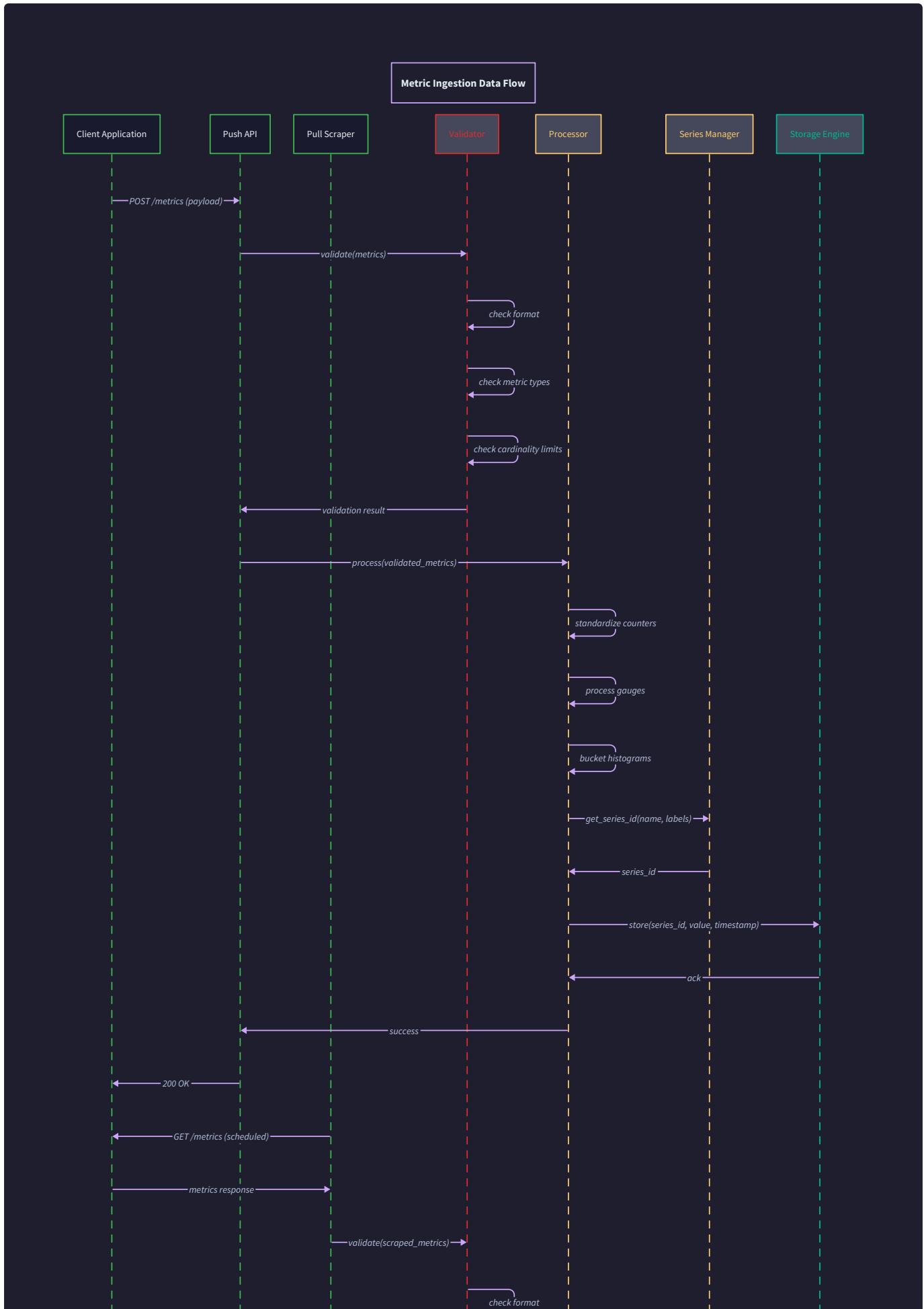
Assembly Line Processing: Once materials pass quality control, they enter the main **assembly line** where workers (processors) perform standardization operations. Counter materials get special handling to ensure they're always increasing, gauge materials are processed to capture point-in-time snapshots, and histogram materials are sorted into predefined buckets. Each processed item gets a unique identifier (series ID) based on its name and characteristics.

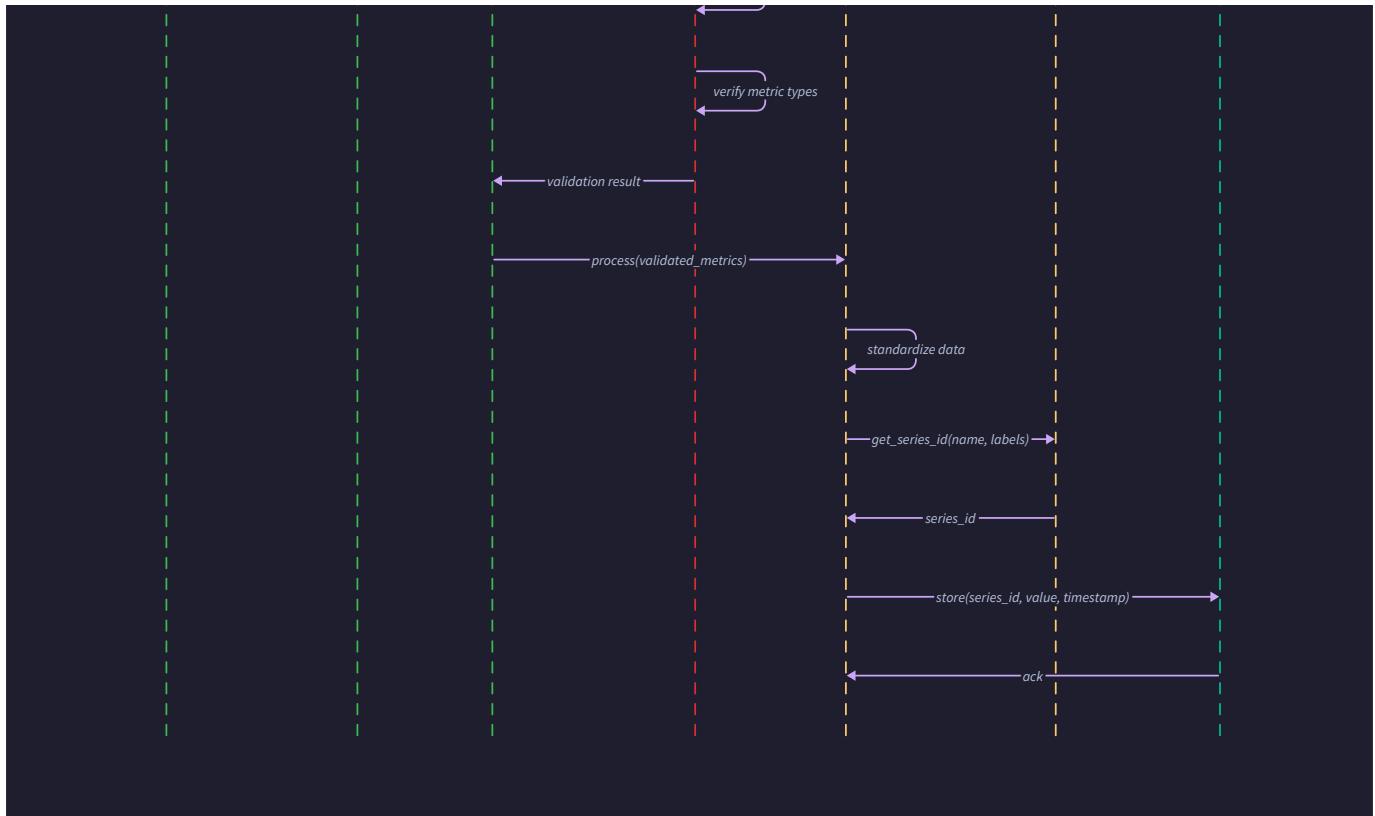
Dispatch Center: Finally, processed materials are packaged and sent to the **warehouse** (storage engine) with proper routing information. The dispatch center maintains strict ordering and batching to ensure the warehouse can efficiently store everything without becoming overwhelmed.

This factory operates 24/7 with **built-in resilience mechanisms**. If one assembly line gets backed up, work can be redistributed. If quality control detects a problematic supplier, it can temporarily reject their materials while alerting operations staff. The entire system is designed to handle sudden spikes in incoming materials while maintaining consistent quality and throughput.

Push vs Pull Ingestion Models

The choice between push and pull ingestion models represents one of the most fundamental architectural decisions in metrics collection systems. Each model has distinct advantages and addresses different operational patterns, making it essential to support both approaches in a comprehensive metrics platform.





Decision: Support Both Push and Pull Ingestion Models

- **Context:** Applications have different operational patterns - some generate metrics continuously and prefer to send data immediately, while others benefit from having a central system collect data on a schedule. Additionally, different deployment environments favor different approaches.
- **Options Considered:** Push-only (applications send data), Pull-only (server scrapes data), Hybrid approach (support both)
- **Decision:** Implement hybrid approach supporting both push and pull models
- **Rationale:** Push provides immediate data delivery and works well with ephemeral workloads, while pull provides centralized control and works better with service discovery. Supporting both maximizes compatibility and deployment flexibility.
- **Consequences:** Increased implementation complexity but significantly broader applicability across different environments and use cases.

Push-Based Ingestion Architecture

In the **push model**, applications take responsibility for actively sending their metrics data to the ingestion engine. This approach treats the metrics system as a service that applications can call whenever they have data to report. The ingestion engine operates as a passive receiver, exposing HTTP endpoints that accept metric payloads and respond with success or error indicators.

The push model excels in environments with **dynamic or ephemeral workloads**. Consider a serverless function that runs for only a few seconds - it can immediately push its execution metrics before terminating, ensuring no data is lost. Similarly, applications running in containers that scale up and down rapidly can send metrics as soon as they're generated, without requiring the metrics system to maintain awareness of their existence.

Push-based ingestion also provides **immediate data availability**. The moment an application generates a critical metric - such as an error rate spike or a performance degradation - it can send that data directly to the ingestion engine. This immediacy is crucial for time-sensitive alerting scenarios where even a few seconds of delay could impact response times.

However, the push model places **operational burden on applications**. Each application must implement metrics client libraries, handle network failures, implement retry logic, and manage authentication credentials for the metrics service. Applications must also be configured with the metrics service endpoint, creating operational dependencies that can complicate deployments.

Pull-Based Ingestion Architecture

The **pull model** inverts the responsibility relationship, making the ingestion engine an active collector that periodically scrapes metrics from application endpoints. Applications expose their current metrics via HTTP endpoints (typically `/metrics`) in a standardized format, and the ingestion engine maintains a schedule of which endpoints to scrape and how frequently.

This approach provides **centralized operational control**. The metrics system maintains the master list of what to monitor, how frequently to collect data, and how to authenticate with each endpoint. Operations teams can add new applications to monitoring, adjust collection frequencies, or temporarily disable problematic endpoints without requiring application changes or redeployments.

Pull-based collection also enables **sophisticated service discovery integration**. The ingestion engine can automatically discover new application instances through service registries, cloud provider APIs, or orchestration platforms like Kubernetes. As new instances start up, they automatically become part of the monitoring system without manual configuration.

The pull model provides **built-in health checking capabilities**. If an application becomes unresponsive, the scraping process will detect this condition through failed HTTP requests, enabling the metrics system to generate alerts about application availability.

This creates a natural integration between metrics collection and basic health monitoring.

However, pull-based collection has **timing limitations**. Metrics are only collected at predetermined intervals, meaning short-lived events or rapid changes might be missed between scrapes. Additionally, applications behind firewalls or NAT may not be reachable by external scrapers, limiting deployment options.

Implementation Strategy Comparison

Aspect	Push Model	Pull Model	Hybrid Approach
Data Freshness	Immediate (sub-second)	Delayed (scrape interval)	Best of both worlds
Operational Complexity	Distributed (each app)	Centralized (metrics system)	Higher initial complexity
Service Discovery	Manual configuration	Automatic discovery	Both manual and automatic
Network Requirements	Outbound from apps	Inbound to apps	Both directions
Ephemeral Workloads	Excellent support	Poor support	Excellent support
Debugging Visibility	Limited (app-side logs)	Excellent (central logs)	Comprehensive visibility
Authentication	Per-app credentials	Central credential management	Flexible auth strategies

Hybrid Implementation Design

Our metrics ingestion engine implements both models through a **unified processing pipeline** that normalizes data from both sources before validation and storage. This design ensures consistent behavior regardless of how metrics arrive in the system.

The **HTTP API layer** provides RESTful endpoints for push-based submissions while also hosting the scraping orchestrator that manages pull-based collection. Both pathways converge into the same validation and preprocessing pipeline, ensuring data consistency and uniform error handling.

Configuration management allows operators to define both push endpoints (for applications to submit data) and pull targets (for the system to scrape) in a single configuration file. This unified approach simplifies operational procedures and provides a single place to manage all metric collection policies.

The system maintains **separate health monitoring** for push and pull pathways. Push endpoints are monitored for request rates, error rates, and processing latency. Pull operations are monitored for scrape success rates, target reachability, and scrape duration. This comprehensive monitoring ensures operators can quickly identify issues with either collection method.

Validation and Preprocessing

The validation and preprocessing stage serves as the **quality gateway** between raw metric submissions and the storage engine. This component must handle the inherent messiness of real-world data while protecting the system from malicious or malformed inputs that could compromise performance or reliability.

Label Cardinality Control

Label cardinality explosion represents one of the most dangerous failure modes in metrics systems. Cardinality refers to the number of unique combinations of label key-value pairs across all metrics. When applications generate labels with unbounded or high-variability values - such as user IDs, request IDs, or timestamps - the number of unique time series grows exponentially.

Consider an application that creates a metric `http_requests_total` with labels for `method`, `status_code`, and `user_id`. If the application serves 1 million users, this single metric type generates 1 million unique time series (assuming one method and status code combination per user). With multiple methods and status codes, the cardinality explodes to millions or tens of millions of series.

Critical Insight: Each unique time series consumes memory for indexing and storage space for samples. Cardinality explosions can quickly exhaust system resources, causing the entire metrics platform to become unresponsive or crash.

The ingestion engine implements **multi-layered cardinality protection**:

Static Label Validation examines each metric submission to ensure labels conform to naming conventions and value constraints. Label names must follow DNS-like naming rules (alphanumeric characters and underscores only), and label values are checked against length limits and character restrictions.

Dynamic Cardinality Tracking maintains counters for unique time series being created in real-time. The system tracks cardinality per metric name, per application source, and globally. When any of these counters approaches configured thresholds, the system begins rejecting new series creation while continuing to accept samples for existing series.

High-Cardinality Detection uses statistical analysis to identify potentially problematic label patterns. If a metric name suddenly generates many new series in a short time period, or if specific label keys show high variability, the system flags these patterns for operator review and can automatically implement temporary restrictions.

Metric Type Validation

Each metric type has specific **semantic requirements** that must be enforced during ingestion to ensure data consistency and query correctness. The validation logic differs significantly across the three primary metric types.

Counter Validation ensures that counter metrics behave as monotonically increasing cumulative values. The ingestion engine tracks the last seen value for each counter series and rejects samples with values lower than the previous sample (which would indicate incorrect counter usage). When counter resets occur (such as application restarts), the system detects these by identifying large decreases in value and handles them appropriately by recording reset events.

Gauge Validation is more permissive since gauges represent point-in-time measurements that can increase or decrease freely. However, the system still validates that gauge values fall within reasonable numeric ranges and aren't special values like NaN or infinity unless explicitly configured to allow them.

Histogram Validation requires the most complex logic since histograms consist of multiple related series (buckets, count, and sum). The ingestion engine validates that bucket values are monotonically increasing across bucket boundaries, that the count

matches the sum of all bucket increments, and that the sum value is mathematically consistent with the observed count and bucket distributions.

Metric Type	Key Validations	Failure Actions
Counter	Monotonic increase, numeric range, reset detection	Reject decreasing values, log reset events
Gauge	Numeric range, finite values	Reject invalid numbers, allow all valid changes
Histogram	Bucket consistency, count/sum coherence, bucket ordering	Reject inconsistent histograms, validate all components

Data Normalization

Raw metric data arrives in various formats and with inconsistent naming patterns. The **data normalization process** standardizes this data into canonical formats that the storage engine can efficiently process.

Metric Name Normalization converts metric names to standard formats, typically lowercase with underscores separating words.

The system may also apply prefix rules (such as adding application or environment prefixes) and suffix standardization (ensuring counter metrics end with `_total` or similar conventions).

Label Standardization includes sorting label keys alphabetically, normalizing label values (such as converting HTTP status codes to strings or normalizing case), and applying label transformation rules. The system may also inject standard labels such as application name, environment, or collection timestamp.

Timestamp Processing handles the complexities of time-based data. Samples may arrive with timestamps in various formats (Unix seconds, milliseconds, nanoseconds, or ISO 8601 strings), in different time zones, or without timestamps at all. The normalization process converts all timestamps to a canonical format (typically Unix nanoseconds in UTC) and may apply clock skew detection to identify submissions with timestamps significantly different from the ingestion time.

Value Normalization ensures numeric consistency across different input formats. This includes handling scientific notation, different decimal precision levels, and unit conversions where configured (such as converting milliseconds to seconds for duration metrics).

Error Handling and Feedback

The validation and preprocessing pipeline must provide **detailed error feedback** while maintaining high throughput performance. This requires careful balance between comprehensive validation and processing speed.

Validation Error Classification categorizes errors into different severity levels:

- **Fatal errors** (malformed JSON, missing required fields) that cause entire batch rejection
- **Warning errors** (unusual cardinality patterns, timestamp skew) that allow processing but generate alerts
- **Individual metric errors** (single invalid metric in a batch) that reject specific metrics while processing others

Error Response Design provides structured feedback that applications can use to fix submission problems:

Error Type	HTTP Status	Response Format	Retry Recommendation
Malformed Request	400 Bad Request	JSON with field-specific errors	Fix format, retry immediately
Cardinality Limit	429 Too Many Requests	JSON with cardinality details	Reduce cardinality, retry with backoff
Authentication	401 Unauthorized	JSON with auth requirements	Fix credentials, retry immediately
Rate Limiting	429 Too Many Requests	JSON with retry-after header	Respect rate limits, retry after delay
Server Error	500 Internal Server Error	JSON with error tracking ID	Retry with exponential backoff

Batch Processing Logic handles mixed-success scenarios where some metrics in a batch are valid while others fail validation. The system processes all valid metrics and returns detailed error information for failed metrics, allowing applications to resubmit only the corrected data.

Common Pitfalls

⚠️ Pitfall: Ignoring Label Cardinality Limits Many developers create metrics with labels containing user IDs, request IDs, or other high-cardinality values without understanding the exponential impact on system resources. A single metric with a user ID label in a system with millions of users creates millions of unique time series. **Why it's wrong:** Each unique time series consumes memory for indexing and storage space for data points. High cardinality can quickly exhaust system memory and make queries extremely slow. **How to fix:** Implement strict cardinality limits per metric name (typically 1000-10000 series). Use high-cardinality values in logs, not metrics. Create aggregate metrics instead of per-entity metrics.

⚠️ Pitfall: Not Handling Clock Skew in Distributed Systems When multiple applications submit metrics with their local timestamps, clock differences between systems can cause samples to arrive with timestamps in the future or significantly in the past, breaking time-series ordering assumptions. **Why it's wrong:** Storage engines often assume samples arrive in roughly chronological order. Out-of-order samples can cause index corruption or query inconsistencies. **How to fix:** Detect timestamp skew during ingestion (samples more than 5 minutes in future or 1 hour in past). Either reject skewed samples or use server-side timestamps for all ingestion.

⚠️ Pitfall: Insufficient Validation of Counter Decreases Allowing counter metrics to decrease without proper handling breaks the fundamental counter semantic and makes rate calculations incorrect. **Why it's wrong:** Counters represent cumulative values that should only increase. Decreases typically indicate counter resets (like application restarts) that need special handling for accurate rate calculations. **How to fix:** Track previous counter values per series. Reject samples with decreased values unless they represent valid resets (detected by large decreases). Log reset events for debugging.

⚠️ Pitfall: Blocking Ingestion During Validation Performing expensive validation operations (like cardinality lookups or histogram consistency checks) in the main ingestion request path creates latency spikes and reduces throughput. **Why it's wrong:** Applications submitting metrics expect fast response times. Slow validation creates backpressure that can affect application performance. **How to fix:** Use asynchronous validation where possible. Perform expensive checks in background workers. Cache validation results to avoid repeated expensive operations.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	Go net/http with middleware	Fiber or Echo framework
Metric Parsing	encoding/json for JSON format	Prometheus text format parser
Cardinality Tracking	In-memory maps with sync.RWMutex	Redis or embedded key-value store
Validation Rules	Hard-coded validation functions	Rule engine with YAML configuration
Background Processing	Goroutines with channels	Worker pool with job queue
Health Checking	Simple HTTP endpoint	Comprehensive health check framework

Recommended File Structure

```
internal/ingestion/
  ingester.go          ← Main MetricsIngestor implementation
  ingester_test.go     ← Unit tests for ingestion logic
  validator.go          ← Validation and preprocessing logic
  validator_test.go    ← Validation tests
  scraper.go           ← Pull-based metric collection
  scraper_test.go      ← Scraper tests
  handlers/
    push_handler.go    ← HTTP handlers for push ingestion
    health_handler.go  ← Health check endpoints
    scrape_handler.go  ← Scrape target management
  models/
    metric.go          ← Core metric data structures
    validation.go       ← Validation error types
  config/
    ingestion_config.go ← Configuration structures
```

Infrastructure Starter Code

HTTP Server Infrastructure (complete implementation):

```
package ingestion

GO

import (
    "context"
    "encoding/json"
    "log/slog"
    "net/http"
    "time"
)

// Server wraps HTTP server with graceful shutdown and middleware

type Server struct {

    httpServer *http.Server
    logger     *slog.Logger
    ingester   MetricsIngestor
}

// NewServer creates HTTP server with configured routes and middleware

func NewServer(addr string, ingester MetricsIngestor, logger *slog.Logger) *Server {
    mux := http.NewServeMux()

    server := &Server{
        httpServer: &http.Server{
            Addr:         addr,
            Handler:     mux,
            ReadTimeout: 30 * time.Second,
            WriteTimeout: 30 * time.Second,
        },
        logger:   logger,
        ingester: ingester,
    }

    server.registerRoutes(mux)
}
```

```
    return server

}

func (s *Server) registerRoutes(mux *http.ServeMux) {
    // Push ingestion endpoint

    mux.HandleFunc("/api/v1/metrics", s.handlePushMetrics)

    // Health check

    mux.HandleFunc("/health", s.handleHealth)

    // Scrape target management

    mux.HandleFunc("/api/v1/scrape", s.handleScrapeConfig)
}

func (s *Server) Start() error {
    s.logger.Info("starting ingestion server", "addr", s.httpServer.Addr)
    return s.httpServer.ListenAndServe()
}

func (s *Server) Shutdown(ctx context.Context) error {
    s.logger.Info("shutting down ingestion server")
    return s.httpServer.Shutdown(ctx)
}
```

Validation Error Infrastructure (complete implementation):

```
package models

import (
    "encoding/json"
    "fmt"
    "strings"
)

// ValidationError represents field-specific validation failures

type ValidationError struct {
    Field  string `json:"field"`
    Value  string `json:"value"`
    Message string `json:"message"`
}

func (v ValidationError) Error() string {
    return fmt.Sprintf("validation failed for field '%s' with value '%s': %s",
        v.Field, v.Value, v.Message)
}

// ValidationErrors represents multiple validation failures

type ValidationErrors struct {
    Errors []ValidationError `json:"errors"`
}

func (v ValidationErrors) Error() string {
    var messages []string
    for _, err := range v.Errors {
        messages = append(messages, err.Error())
    }
    return strings.Join(messages, "; ")
}

func (v ValidationErrors) HasErrors() bool {
    return len(v.Errors) > 0
}
```

```
}

func (v *ValidationErrors) Add(field, value, message string) {
    v.Errors = append(v.Errors, ValidationError{
        Field:   field,
        Value:   value,
        Message: message,
    })
}

// ToJSON serializes validation errors for HTTP responses

func (v ValidationErrors) ToJSON() ([]byte, error) {
    return json.Marshal(v)
}
```

Core Logic Skeleton Code

Main Ingestion Interface (signatures + TODOs):

```
package ingestion

import (
    "context"
    "time"

    "internal/models"
)

// MetricsIngestor handles both push and pull metric collection

type MetricsIngestor interface {

    IngestMetrics(ctx context.Context, metrics []models.Metric) error

    ScrapeEndpoint(ctx context.Context, url string) error

    GetStats() IngestionStats
}

// IngestionStats provides operational metrics about the ingestion process

type IngestionStats struct {

    MetricsReceived      int64
    MetricsRejected     int64
    ValidationErrors    int64
    CardinalityViolations int64
    ScrapeTargets        int
    LastScrapeTime       time.Time
}

// Implementation skeleton

type ingestorImpl struct {

    validator      Validator
    storage        TimeSeriesStorage
    cardinalityMgr CardinalityManager
    logger         *slog.Logger
    stats          IngestionStats
}
```

```
// IngestMetrics processes a batch of metrics through validation and storage

func (i *ingesterImpl) IngestMetrics(ctx context.Context, metrics []models.Metric) error {

    // TODO 1: Initialize validation errors collector

    // TODO 2: For each metric in the batch:

    //     - Validate metric name and type using i.validator.ValidateMetric()

    //     - Check cardinality limits using i.cardinalityMgr.CheckCardinality()

    //     - Normalize labels and values using i.validator.NormalizeMetric()

    //     - Add to validated batch if all checks pass

    // TODO 3: If any metrics passed validation:
    //         - Convert to storage format using models.MetricToSamples()
    //         - Call i.storage.WriteSamples() with validated metrics
    //         - Update ingestion statistics

    // TODO 4: Return validation errors if any metrics failed

    // Hint: Use sync.Mutex to protect stats updates in concurrent scenarios

}
```

Validation Logic (signatures + TODOs):

```
package ingestion

import (
    "internal/models"
    "regexp"
    "strings"
)

// Validator handles metric validation and normalization

type Validator interface {

    ValidateMetric(metric *models.Metric) *models.ValidationErrors

    NormalizeMetric(metric *models.Metric) error
}

type validatorImpl struct {

    metricNameRegex *regexp.Regexp

    labelNameRegex *regexp.Regexp

    maxLabelLength int

    maxValueLength int
}

// ValidateMetric checks metric name, type, labels, and values

func (v *validatorImpl) ValidateMetric(metric *models.Metric) *models.ValidationErrors {

    errors := &models.ValidationErrors{}

    // TODO 1: Validate metric name:

    //     - Check if name matches regex pattern (alphanumeric + underscore)
    //     - Verify name length is within limits (1-200 characters)
    //     - Ensure name doesn't start with double underscore (reserved)

    // TODO 2: Validate metric type:

    //     - Check if Type is one of Counter, Gauge, Histogram
    //     - For Counter: verify all sample values are >= 0
    //     - For Histogram: validate bucket structure and consistency

    // TODO 3: Validate labels:
```

GO

```
//   - Check each label name matches regex pattern  
  
//   - Verify label names don't start with double underscore  
  
//   - Validate label values are within length limits  
  
//   - Check for reserved label names (__name__, job, instance)  
  
// TODO 4: Validate samples:  
  
//   - Ensure timestamps are within acceptable range (not too far in past/future)  
  
//   - Check that sample values are finite numbers  
  
//   - For counters, verify values don't decrease (unless reset detected)  
  
// Hint: Use errors.Add() to collect multiple validation failures  
  
  
return errors  
  
}
```

Cardinality Management (signatures + TODOs):

```
package ingestion

import (
    "fmt"
    "sync"

    "internal/models"
)

// CardinalityManager tracks and limits time-series cardinality

type CardinalityManager interface {

    CheckCardinality(metric *models.Metric) error

    GetCardinalityStats() CardinalityStats

    RegisterSeries(seriesID string) error
}

type CardinalityStats struct {

    TotalSeries      int

    SeriesPerMetric map[string]int

    MaxCardinality  int
}

type cardinalityManagerImpl struct {

    mu           sync.RWMutex

    seriesIndex   map[string]bool // seriesID -> exists

    metricCardinality map[string]int // metricName -> count

    maxSeriesTotal  int

    maxSeriesPerMetric int
}

// CheckCardinality verifies adding this metric won't exceed cardinality limits

func (c *cardinalityManagerImpl) CheckCardinality(metric *models.Metric) error {
    c.mu.RLock()

    defer c.mu.RUnlock()
```

```

// TODO 1: Generate series ID using metric.SeriesID() method

// TODO 2: Check if this series already exists in c.seriesIndex

//   - If exists, return nil (no new cardinality impact)

// TODO 3: Check global cardinality limit:

//   - If len(c.seriesIndex) >= c.maxSeriesTotal, return cardinality error

// TODO 4: Check per-metric cardinality limit:

//   - If c.metricCardinality[metric.Name] >= c.maxSeriesPerMetric, return error

// TODO 5: Temporarily register the series (actual registration happens after successful ingestion)

// Hint: Use metric name + sorted labels to create consistent series ID

return nil
}

```

Language-Specific Hints

Go-Specific Implementation Details:

- Use `sync.RWMutex` for cardinality tracking to allow concurrent read access while protecting writes
- Implement `http.HandlerFunc` for push endpoints with proper JSON parsing using `json.NewDecoder(r.Body)`
- Use `time.NewTicker` for scrape scheduling with proper cleanup in goroutines
- Handle context cancellation in long-running operations using `select` with `ctx.Done()`
- Use `slog.Logger` for structured logging with contextual fields
- Implement graceful shutdown by listening for OS signals with `signal.Notify`

Error Handling Patterns:

```

// Wrap validation errors with context

if err := validator.ValidateMetric(metric); err != nil {

    return fmt.Errorf("validation failed for metric %s: %w", metric.Name, err)
}

// Use typed errors for different failure modes

var ErrCardinalityExceeded = errors.New("cardinality limit exceeded")

```

Concurrency Patterns:

```
// Worker pool for batch processing

for i := 0; i < numWorkers; i++ {

    go func() {

        for batch := range workChan {

            if err := processBatch(batch); err != nil {

                errorChan <- err

            }
        }
    }()
}

}
```

Milestone Checkpoint

After implementing the ingestion engine, verify the following behaviors:

Push Ingestion Test:

```
# Start the server

go run cmd/server/main.go

# Submit test metrics via HTTP

curl -X POST http://localhost:8080/api/v1/metrics \
-H "Content-Type: application/json" \
-d '[{
    "name": "test_counter_total",
    "type": 0,
    "labels": {"job": "test", "instance": "localhost:8080"},
    "samples": [{"value": 42.0, "timestamp": "'$(date -u +%s)'"}]
}]'

# Expected: HTTP 200 response with empty JSON body {}

# Expected logs: "ingested 1 metrics successfully"
```

Validation Error Test:

```
# Submit invalid metric

curl -X POST http://localhost:8080/api/v1/metrics \
-H "Content-Type: application/json" \
-d '[{
  "name": "",
  "type": 999,
  "labels": {},
  "samples": []
}]'

# Expected: HTTP 400 response with validation errors JSON

# Expected response body contains field-specific error details
```

BASH

Cardinality Limit Test:

```
# Configure low cardinality limit in config

# Submit metrics with many unique label combinations

# Expected: HTTP 429 after hitting cardinality limit

# Expected logs: "cardinality limit exceeded for metric X"
```

BASH

Health Check Verification:

```
curl http://localhost:8080/health

# Expected: HTTP 200 with {"status": "healthy", "timestamp": "..."}
```

BASH

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Metrics not appearing in storage	Validation failures silently dropping data	Check ingestion logs for validation errors	Add comprehensive logging to validation pipeline
High memory usage	Cardinality explosion from unbounded labels	Monitor cardinality statistics endpoint	Implement stricter label validation and limits
Slow ingestion performance	Synchronous validation blocking request processing	Profile CPU usage during ingestion	Move expensive validation to background workers
Inconsistent timestamps	Clock skew between client and server	Compare ingestion timestamps with sample timestamps	Implement timestamp normalization and skew detection
Counter values decreasing	Application restarts not handled properly	Track counter resets in logs	Implement reset detection and handle counter resets gracefully

Time-Series Storage Engine

Milestone(s): 2 (Storage & Querying) - This section details the storage layer that persists metrics data and enables efficient querying for both dashboards and alerting systems

Mental Model: The Library Archive System

Think of our time-series storage engine as a vast, highly organized library archive system that specializes in chronological documents. Just as a library must efficiently store millions of books, organize them for quick retrieval, and manage space by moving older materials to different storage areas, our storage engine must handle millions of metric samples with similar organizational challenges.

In this library analogy, each **time-series** is like a specific periodical or journal - say "The Daily Weather Reports for New York City" or "Monthly Server CPU Usage for Database-01". Each **sample** within a time-series is like an individual issue of that periodical, with a specific date (timestamp) and content (the metric value). The **labels** on a metric are like the catalog classification system - they help us group related periodicals together and find exactly what we're looking for among millions of possibilities.

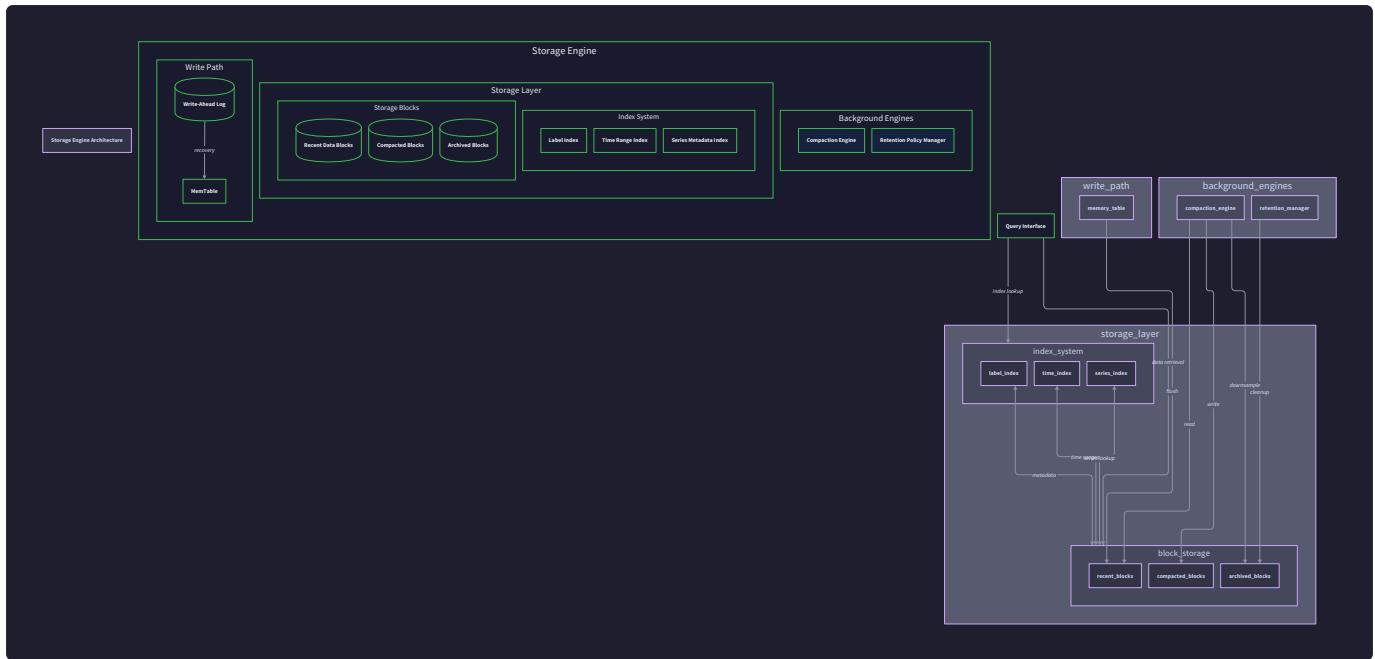
The library's **card catalog system** represents our indexing structure. Just as you can look up books by author, title, subject, or publication date, we need indexes that let us quickly find time-series by metric name, label combinations, and time ranges. Without proper indexing, finding specific data would require scanning through every stored sample - like searching for a specific magazine issue by checking every shelf in the building.

The library's **archival policies** mirror our compaction and retention strategies. Recent magazines are kept in the main reading room for immediate access (high-resolution recent data), older issues are moved to compact storage with lower accessibility (downsampled historical data), and very old materials are eventually discarded according to institutional policies (retention limits). This tiered storage approach balances accessibility, storage costs, and retrieval performance.

Just as librarians periodically reorganize sections, merge duplicate materials, and transfer items between storage areas, our storage engine continuously runs background processes to compact data files, merge small storage blocks into larger ones, and apply retention policies. This maintenance work happens transparently while the library remains open for readers - our storage engine must serve queries and accept new data even while reorganizing itself.

Storage Format and Indexing

The foundation of our time-series storage system rests on a **block-based architecture** where metric samples are organized into immutable storage blocks on disk. This design choice provides several critical advantages: it enables efficient compression within each block, simplifies concurrent access patterns, and makes background compaction operations straightforward to implement safely.



Storage Block Structure

Each storage block contains samples for multiple time-series within a specific time window, typically spanning 2-4 hours of data. Within a block, samples are stored in a columnar format that maximizes compression efficiency. The timestamp column uses delta encoding (storing differences between consecutive timestamps rather than absolute values), while the value column applies appropriate compression based on the data patterns detected during ingestion.

Block Component	Data Structure	Purpose
Block Header	BlockID, TimeRange, SeriesCount, CompressionType	Metadata for quick block filtering and loading
Series Index	SeriesID → ValueOffset mapping	Enables direct jump to specific time-series data within block
Timestamp Column	Delta-encoded timestamp array	Compressed temporal data shared across all series in block
Value Columns	Series-specific compressed value arrays	Actual metric values with optimal compression per series type
Block Footer	CRC checksum, Block size	Data integrity verification and corruption detection

The **series ID** serves as the primary key that uniquely identifies each time-series. It's computed as a hash of the metric name combined with the lexicographically sorted label pairs. For example, the series ID for `http_requests_total{method="GET", status="200"}` would be `hash("http_requests_total" + "method=GET, status=200")`. This deterministic approach ensures that samples belonging to the same logical time-series always map to the same series ID, regardless of the order in which labels appear in incoming metrics.

Decision: Block-Based vs. Row-Based Storage

- **Context:** Need to choose fundamental storage layout for time-series samples
- **Options Considered:**
 - Row-based: Each sample stored as complete record with all fields
 - Block-based: Samples grouped into columnar blocks by time window
 - Hybrid: Combination approach with recent data row-based, historical data block-based
- **Decision:** Block-based columnar storage with 2-4 hour time windows
- **Rationale:** Time-series data exhibits excellent compression characteristics when organized columnar, query patterns typically involve time ranges rather than individual samples, and block immutability simplifies concurrent access and compaction
- **Consequences:** Enables 5-10x compression ratios, optimizes for range queries, but requires more complex indexing for point lookups

Indexing Architecture

The storage engine maintains multiple index structures to support different query patterns efficiently. The **primary index** maps series IDs to the list of blocks containing data for that time-series. This index is kept entirely in memory for fast lookups and is persisted to disk for crash recovery. Each index entry contains the series metadata (metric name and labels) plus a sorted list of block references with their time ranges.

Index Type	Structure	Query Pattern Supported
Primary Index	SeriesID → BlockList mapping	Exact series lookup by metric name + labels
Label Index	LabelName → LabelValue → SeriesID list	Label-based filtering and series discovery
Time Index	TimeRange → BlockID list	Time-range queries and block pruning
Inverted Index	LabelValue → SeriesID list	Cross-metric label searches

The **label index** enables efficient filtering by label values without scanning all series. For each unique label name that appears in the dataset, we maintain a secondary index mapping each possible value to the list of series IDs that contain that label-value pair. This structure allows queries like "find all series with `status='500'`" to execute in logarithmic time rather than requiring a full series scan.

Compression Strategies

Different metric types exhibit distinct value patterns that benefit from specialized compression approaches. **Counter metrics** typically show monotonically increasing values, making them excellent candidates for delta-of-delta encoding where we store the difference between consecutive delta values. **Gauge metrics** often have values clustered around certain ranges, benefiting from dictionary compression that maps frequent values to short codes.

Metric Type	Compression Algorithm	Typical Compression Ratio	Rationale
Counter	Delta-of-delta encoding	8:1 to 12:1	Values increase monotonically with predictable rates
Gauge	Dictionary + RLE compression	4:1 to 8:1	Values cluster around common ranges with repetition
Histogram	Mixed: delta encoding for timestamps, dictionary for bucket counts	6:1 to 10:1	Bucket structure provides natural compression opportunities

File Organization

Storage blocks are organized into a hierarchical directory structure that facilitates efficient querying and maintenance operations. The root storage directory contains subdirectories organized by day, with each day containing hour-based subdirectories that hold the actual block files. This organization allows time-based query optimizations and simplifies retention policy implementation.

```
data/
└── 2024-01-15/
    ├── 00/ (midnight hour)
    │   ├── block_001.db
    │   ├── block_002.db
    │   └── index.db
    ├── 01/ (1 AM hour)
    │   ├── block_003.db
    │   └── index.db
    └── ...
└── 2024-01-16/
    └── ...
└── meta/
    ├── series_index.db
    └── label_index.db
```

This hierarchical approach provides several operational benefits. Time-range queries can immediately eliminate entire directories from consideration, reducing I/O overhead. Retention policies can delete entire day directories atomically. Background maintenance tasks can process data in convenient time-based chunks without interfering with current ingestion.

Compaction and Retention Policies

The storage engine's longevity and performance depend critically on automated data lifecycle management. Without compaction, the system would accumulate thousands of small storage blocks that degrade query performance and waste disk space. Without retention policies, historical data would consume unbounded storage and eventually exhaust available capacity.

Compaction Strategy

Our compaction system operates on multiple levels, similar to LSM-tree approaches but adapted for time-series workloads. **Level 0 compaction** runs continuously, merging small blocks created during active ingestion into larger, more efficiently compressed blocks. **Level 1 compaction** runs hourly, reorganizing data across longer time windows to optimize for common query patterns. **Level 2 compaction** runs daily, performing deep reorganization and applying downsampling policies to historical data.

Compaction Level	Trigger	Input Blocks	Output Blocks	Primary Goal
Level 0	Every 15 minutes	5-10 small ingestion blocks	1 medium block	Reduce file count, improve compression
Level 1	Hourly	All blocks in hour window	1-2 optimized blocks	Optimize query performance, remove duplicates
Level 2	Daily	Day's worth of historical blocks	Downsampled summary blocks	Apply retention policies, reduce storage footprint

The compaction process must handle several complex scenarios. **Overlapping time ranges** occur when blocks contain data from slightly different time windows - the compaction algorithm merges samples in timestamp order while detecting and removing duplicates. **Schema evolution** happens when new labels appear in the dataset - the compaction process updates index structures to accommodate new label combinations without breaking existing queries.

Downsampling Algorithms

As data ages, maintaining full-resolution samples becomes increasingly expensive while providing diminishing value for most use cases. Our downsampling system applies configurable policies that reduce data resolution while preserving the statistical properties most important for analysis and alerting.

Decision: Fixed vs. Adaptive Downsampling Windows

- **Context:** Need to balance storage efficiency with data fidelity for historical metrics
- **Options Considered:**
 - Fixed windows: Predetermined downsampling intervals (1h → 5m, 1d → 1h, 1w → 6h)
 - Adaptive windows: Dynamic intervals based on data volatility and query patterns
 - Query-driven: Downsample only when storage pressure or query performance degrades
- **Decision:** Fixed windows with configurable thresholds
- **Rationale:** Predictable storage growth, simplified implementation, deterministic behavior for alerting rules that depend on historical data
- **Consequences:** May oversample stable metrics or undersample volatile ones, but provides consistent query performance and storage characteristics

The downsampling process preserves different statistical properties depending on the metric type. For **counter metrics**, we maintain the rate-of-change information by storing periodic snapshots along with minimum and maximum observed rates within each downsampling window. For **gauge metrics**, we preserve the average, minimum, maximum, and last observed value within each window. For **histogram metrics**, we merge bucket counts and recalculate percentiles at the reduced resolution.

Downsampling Window	Original Resolution	Target Resolution	Statistical Preservation
1 hour - 24 hours	15 second intervals	1 minute intervals	Full statistical detail maintained
1 day - 7 days	1 minute intervals	5 minute intervals	Min/max/avg/last preserved per window
1 week - 30 days	5 minute intervals	30 minute intervals	Statistical summary with confidence intervals
30+ days	30 minute intervals	2 hour intervals	Trend data only, detailed statistics discarded

Retention Policy Implementation

Retention policies operate at multiple granularities to provide fine-grained control over data lifecycle management. **Global policies** apply default retention periods to all metrics, **metric-specific policies** override defaults for particular metric names, and **label-based policies** allow retention decisions based on label values such as environment or criticality levels.

The retention engine runs as a background process that evaluates policies every hour and marks eligible data for deletion. Rather than immediately removing data, the system uses a **two-phase deletion** approach: first marking blocks as expired, then physically removing them after a grace period. This approach prevents accidental data loss and allows recovery from misconfigured retention policies.

Policy Type	Evaluation Frequency	Scope	Example Use Case
Global	Every hour	All metrics	"Delete all data older than 90 days"
Metric-specific	Every hour	Named metrics	"Keep http_request logs for 180 days"
Label-based	Every hour	Metrics matching label patterns	"Keep production env data for 1 year, dev data for 30 days"
Storage-based	Every 30 minutes	Entire system	"When disk usage > 80%, delete oldest non-critical data"

The policy evaluation engine maintains a priority queue of deletion candidates, allowing the system to free storage space intelligently during capacity pressure situations. Critical metrics (those referenced by active alert rules or displayed on important dashboards) receive higher retention priority than metrics that haven't been queried recently.

Consistency and Durability Guarantees

The storage engine provides **eventual consistency** for compaction operations while maintaining **strong consistency** for retention policy application. During compaction, readers may temporarily observe both the original blocks and the newly compacted blocks, but query results remain correct because the block selection logic handles overlapping time ranges properly.

Write operations use a **write-ahead log (WAL)** to ensure durability. Incoming samples are first written to the WAL with an fsync operation, then asynchronously flushed to storage blocks. This approach provides immediate durability guarantees while allowing batched writes to optimize storage block creation. The WAL is automatically truncated as samples are successfully persisted to storage blocks.

⚠ Pitfall: Ignoring Compaction During Development Many developers focus on implementing the basic read/write functionality and defer compaction implementation until later. This approach leads to severe performance degradation as the system accumulates thousands of small storage blocks. Query performance degrades exponentially with block count, and storage space usage can be 5-10x higher than necessary. Implement basic compaction early, even if it's a simple periodic merger of small blocks.

⚠ Pitfall: Retention Policies That Don't Account for Alert Dependencies Implementing retention policies that delete historical data without considering active alert rules can break alerting functionality. Alert rules often need historical data for baseline calculations or trend analysis. The retention system must maintain metadata about which metrics are referenced by active alerts and apply appropriate retention extensions automatically.

Implementation Guidance

The time-series storage engine implementation requires careful attention to concurrent access patterns, efficient file I/O, and robust error handling. The following guidance provides the foundation for building a production-ready storage system.

Technology Recommendations

Component	Simple Option	Advanced Option
File Format	JSON lines with gzip compression	Custom binary format with advanced compression
Indexing	In-memory maps with periodic persistence	B+ tree or LSM-tree based indexes
Compression	Go's gzip/lz4 for general compression	Specialized time-series compression (Gorilla, etc.)
Concurrency	sync.RWMutex for index protection	Lock-free data structures with atomic operations
Background Tasks	time.Ticker with simple scheduling	Priority queue with adaptive scheduling

File Structure

```

internal/storage/
├── engine.go           ← Main storage engine implementation
├── block.go            ← Storage block read/write operations
├── index.go             ← In-memory and persistent indexing
├── compaction.go        ← Background compaction processes
├── retention.go         ← Data lifecycle management
├── wal.go               ← Write-ahead log for durability
├── compression.go       ← Compression algorithm implementations
├── query.go              ← Query execution against storage blocks
└── storage_test.go      ← Comprehensive integration tests

```

Core Data Structures

```
package storage
```

GO

```
import (
    "context"
    "sync"
    "time"
)

// StorageEngine manages time-series data with compaction and retention

type StorageEngine struct {

    config      *StorageConfig

    dataDir     string

    seriesIndex *SeriesIndex

    blockManager *BlockManager

    compactor    *Compactor

    retentionMgr *RetentionManager

    wal          *WriteAheadLog

    mu          sync.RWMutex

    closed      bool

}

// SeriesIndex maintains the mapping from series IDs to storage blocks

type SeriesIndex struct {

    series      map[string]*SeriesInfo

    labels      map[string]map[string][]string // label_name -> label_value -> series_ids

    mu         sync.RWMutex

    dirty      bool

}

// SeriesInfo contains metadata and block references for a time-series

type SeriesInfo struct {

    SeriesID    string

    MetricName  string

    Labels      Labels
}
```

```
    Blocks      [] *BlockReference

    LastSample  time.Time

}

// BlockReference points to a storage block containing series data

type BlockReference struct {

    BlockID      string

    FilePath     string

    TimeRange    TimeRange

    SeriesCount  int

    Compressed   bool

}

// StorageBlock represents an immutable collection of time-series samples

type StorageBlock struct {

    ID          string

    TimeRange   TimeRange

    Series      map[string][]Sample

    Metadata    map[string]interface{[]}

    compressed  bool

}
```

Core Storage Engine Interface

GO

```
// TimeSeriesStorage defines the primary storage interface

type TimeSeriesStorage interface {

    // WriteSamples persists metric samples to storage with durability guarantees

    WriteSamples(ctx context.Context, samples []Sample) error

    // QueryRange retrieves samples for specified series within time range

    QueryRange(ctx context.Context, seriesID string, start, end time.Time) ([]Sample, error)

    // QuerySeries finds all series matching label selectors

    QuerySeries(ctx context.Context, matchers []LabelMatcher) ([]SeriesInfo, error)

    // GetSeriesInfo retrieves metadata for a specific series

    GetSeriesInfo(ctx context.Context, seriesID string) (*SeriesInfo, error)

    // Close gracefully shuts down storage engine and background processes

    Close() error

}

// LabelMatcher defines criteria for series selection

type LabelMatcher struct {

    Name      string
    Value     string
    Operator  MatchOperator // Equal, NotEqual, RegexMatch, etc.

}
```

Write-Ahead Log Implementation

The WAL ensures durability by persisting samples before acknowledging writes. This complete implementation handles the critical path for data integrity:

```
package storage
```

GO

```
import (
```

```
    "bufio"
```

```
    "encoding/json"
```

```
    "fmt"
```

```
    "os"
```

```
    "path/filepath"
```

```
    "sync"
```

```
    "time"
```

```
)
```

```
// WriteAheadLog provides durability guarantees for metric samples
```

```
type WriteAheadLog struct {
```

```
    filePath    string
```

```
    file        *os.File
```

```
    writer      *bufio.Writer
```

```
    mu          sync.Mutex
```

```
    lastFlush   time.Time
```

```
    flushInterval time.Duration
```

```
}
```

```
// WALRecord represents a single entry in the write-ahead log
```

```
type WALRecord struct {
```

```
    Timestamp time.Time `json:"timestamp"`

    Type      string     `json:"type"`

    Data      []Sample   `json:"data"`

    Checksum uint32     `json:"checksum"`

}
```

```
// NewWriteAheadLog creates and initializes a new WAL instance
```

```
func NewWriteAheadLog(dataDir string, flushInterval time.Duration) (*WriteAheadLog, error) {
```

```
    walPath := filepath.Join(dataDir, "wal.log")
```

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

if err != nil {
    return nil, fmt.Errorf("failed to open WAL file: %w", err)
}

wal := &WriteAheadLog{
    filePath:      walPath,
    file:         file,
    writer:       bufio.NewWriter(file),
    flushInterval: flushInterval,
    lastFlush:     time.Now(),
}

// Start background flush routine
go wal.backgroundFlush()

return wal, nil
}

// AppendSamples writes samples to WAL with immediate durability
func (w *WriteAheadLog) AppendSamples(samples []Sample) error {
    w.mu.Lock()
    defer w.mu.Unlock()

    record := WALRecord{
        Timestamp: time.Now(),
        Type:      "samples",
        Data:      samples,
        Checksum: calculateChecksum(samples),
    }

    // Serialize record to JSON
}
```

```
data, err := json.Marshal(record)

if err != nil {
    return fmt.Errorf("failed to marshal WAL record: %w", err)
}

// Write to buffer

if _, err := w.writer.Write(data); err != nil {
    return fmt.Errorf("failed to write to WAL: %w", err)
}

if _, err := w.writer.WriteString("\n"); err != nil {
    return fmt.Errorf("failed to write newline: %w", err)
}

// Force immediate flush for durability

if err := w.writer.Flush(); err != nil {
    return fmt.Errorf("failed to flush WAL buffer: %w", err)
}

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

return nil
}

// backgroundFlush periodically ensures data is written to disk

func (w *WriteAheadLog) backgroundFlush() {
    ticker := time.NewTicker(w.flushInterval)
    defer ticker.Stop()

    for range ticker.C {

```

```
w.mu.Lock()  
  
w.writer.Flush()  
  
w.file.Sync()  
  
w.lastFlush = time.Now()  
  
w.mu.Unlock()  
  
}  
  
}
```

Storage Engine Core Implementation Template

The following template provides the structure for the main storage engine. Key methods are stubbed with detailed TODOs that correspond to the design algorithms described above:

```
// NewStorageEngine creates a new time-series storage engine
```

GO

```
func NewStorageEngine(config *StorageConfig) (*StorageEngine, error) {
```

```
    engine := &StorageEngine{
```

```
        config:      config,
```

```
        dataDir:     config.DataDir,
```

```
        seriesIndex: NewSeriesIndex(),
```

```
        blockManager: NewBlockManager(config.DataDir),
```

```
    }
```



```
    // TODO 1: Create data directory if it doesn't exist
```

```
    // TODO 2: Initialize write-ahead log with configured flush interval
```

```
    // TODO 3: Load existing series index from disk (if present)
```

```
    // TODO 4: Start background compaction goroutine
```

```
    // TODO 5: Start background retention policy goroutine
```

```
    // TODO 6: Register shutdown cleanup handlers
```



```
    return engine, nil
```

```
}
```



```
// WriteSamples persists metric samples with durability guarantees
```

```
func (e *StorageEngine) WriteSamples(ctx context.Context, samples []Sample) error {
```

```
    e.mu.RLock()
```

```
    defer e.mu.RUnlock()
```



```
    if e.closed {
```

```
        return fmt.Errorf("storage engine is closed")
```

```
    }
```



```
    // TODO 1: Write samples to WAL for immediate durability
```

```
    // TODO 2: Group samples by series ID for efficient batching
```

```
    // TODO 3: Update series index with new samples and time ranges
```

```
    // TODO 4: Add samples to current active storage block
```

```
// TODO 5: If block reaches size threshold, trigger compaction

// TODO 6: Update cardinality tracking for label explosion detection


return nil

}

// QueryRange retrieves samples for specified series within time range

func (e *StorageEngine) QueryRange(ctx context.Context, seriesID string, start, end time.Time) ([]Sample, error) {

    e.mu.RLock()

    defer e.mu.RUnlock()

    // TODO 1: Look up series info in index to get block references

    // TODO 2: Filter blocks by time range to minimize I/O

    // TODO 3: Load samples from each relevant block

    // TODO 4: Merge samples in chronological order

    // TODO 5: Apply any necessary decompression

    // TODO 6: Filter samples to exact time range requested

    return nil, nil
}
```

Background Compaction Process

```

// Compactor handles background compaction of storage blocks
GO

type Compactor struct {

    engine      *StorageEngine

    running     bool

    stopCh      chan struct{}`

    mu          sync.Mutex

}

// runCompaction executes the main compaction logic

func (c *Compactor) runCompaction(ctx context.Context) error {

    // TODO 1: Scan data directory for blocks eligible for compaction

    // TODO 2: Group blocks by time range and size for optimal merging

    // TODO 3: For each group, read all samples into memory

    // TODO 4: Sort samples by timestamp and remove duplicates

    // TODO 5: Apply compression and write to new compacted block

    // TODO 6: Update series index to reference new block

    // TODO 7: Mark old blocks for deletion after grace period

    // TODO 8: Update compaction statistics and metrics


    return nil
}

```

Language-Specific Optimization Hints

- **File I/O:** Use `os.File.ReadAt()` and `os.File.WriteAt()` for concurrent access to different file regions without seeking
- **Memory Management:** Pre-allocate slices with known capacity using `make([]Sample, 0, expectedCount)` to avoid repeated allocations
- **Concurrency:** Use `sync.RWMutex` for the series index since reads vastly outnumber writes in typical workloads
- **Error Handling:** Wrap file system errors with context using `fmt.Errorf("operation failed: %w", err)` for better debugging
- **JSON Performance:** Consider using `encoding/json.Encoder` with a pooled buffer for better performance than `json.Marshal`

Milestone Checkpoint

After implementing the storage engine, verify functionality with these concrete tests:

1. **Basic Write/Read Test:** Write 1000 samples across 10 different series, then query each series individually. Verify all samples are returned in correct chronological order.

2. **Compaction Test:** Write enough data to trigger compaction (based on your block size threshold), wait for compaction to complete, then verify queries still return correct results and storage directory contains fewer files.
3. **WAL Recovery Test:** Write samples, kill the process before graceful shutdown, restart and verify all samples are recovered from the WAL.
4. **Concurrent Access Test:** Run multiple goroutines writing different series while others query existing data. Verify no race conditions using `go test -race`.

Expected behavior: Write operations should complete in under 10ms for batches of 100 samples, queries should return results in under 50ms for 24-hour ranges, and compaction should reduce file count by 5-10x while maintaining query correctness.

Common Implementation Pitfalls

- ⚠ **Pitfall: Not Handling Partial Writes** File system operations can be interrupted, leaving storage blocks in an inconsistent state. Always write to temporary files first, then atomically rename them to the final location using `os.Rename()`. This ensures readers never see partially written data.
- ⚠ **Pitfall: Index Corruption During Crashes** The series index is critical for query performance but vulnerable to corruption if the process crashes during updates. Implement index versioning with atomic updates: write new index to `index.new`, `fsync`, then rename to `index.db`.
- ⚠ **Pitfall: Memory Leaks in Long-Running Compaction** Compaction processes can accumulate large amounts of data in memory when processing multiple blocks. Implement streaming compaction that processes blocks in chunks rather than loading entire datasets into memory simultaneously.

Query Engine

Milestone(s): 2 (Storage & Querying) - This section details the query processing system that executes metric queries for both dashboard visualization and alert evaluation

Mental Model: The Research Assistant

Think of the Query Engine as an experienced research assistant working in a vast scientific library. When you need specific information, you don't hand the assistant a raw storage location - instead, you make a request in natural language: "Find me all the temperature readings from Building A sensors between 2 PM and 4 PM yesterday, and calculate the average for each floor."

The research assistant then performs a sophisticated multi-step process. First, they parse your request to understand exactly what you're looking for - which metrics, what time period, which filtering criteria, and what calculations to perform. Next, they create a research plan - determining which sections of the library to visit, in what order, and how to efficiently gather the information. Then they execute this plan, navigating the library's indexing system to locate the relevant data quickly. Finally, they process and analyze the raw information, performing the requested calculations and presenting the results in the format you need.

This analogy captures the essence of query processing: translating high-level information requests into efficient data retrieval and computation operations. Just as a good research assistant knows shortcuts through the library and can optimize their research strategy based on the type of question, our Query Engine must understand how to leverage the time-series storage structure to answer metric queries efficiently.

The key insight is that query processing is not just about finding data - it's about finding data *efficiently* and performing meaningful computations on it. A research assistant who checks every single book in the library for your temperature readings would eventually find the answer, but would take far too long. Similarly, a query engine that scans every stored sample to answer a simple aggregation query would be functionally useless at scale.

Query Language Design

Our query language provides a simple but powerful syntax for expressing metric selection, time range filtering, and aggregation operations. The design balances expressiveness with simplicity, targeting the common use cases that appear in both dashboard visualizations and alert rule definitions.

Decision: Simplified Query Syntax Over Full Programming Language

- **Context:** We need a way for users to specify what metric data they want and how to process it, with options ranging from simple metric selection to complex mathematical expressions
- **Options Considered:**
 1. Simple metric selection with basic aggregation functions
 2. SQL-like query language with time-series extensions
 3. Full expression language with custom functions and variables
- **Decision:** Simple metric selection syntax with built-in aggregation functions and basic arithmetic
- **Rationale:** Dashboard users and alert definitions need to be readable by operations teams, not just developers. Complex query languages create a barrier to adoption and increase the chance of errors in critical alerting rules
- **Consequences:** Enables quick learning curve and reduces syntax errors, but limits advanced analytical capabilities and may require multiple queries for complex scenarios

Query Language Option	Pros	Cons	Chosen?
Simple Selection + Aggregation	Easy to learn, hard to misuse, fast parsing	Limited analytical power, may need multiple queries	✓ Yes
SQL-like Syntax	Familiar to many users, powerful joins and subqueries	Complex for time-series concepts, verbose syntax	No
Full Expression Language	Maximum flexibility, supports complex math	High learning curve, error-prone for alerts	No

The query language consists of four main components that work together to specify data selection and processing:

Metric Selection forms the foundation of every query, specifying which time-series to include in the result set. The basic syntax uses the metric name followed by optional label matchers enclosed in curly braces. Label matchers support exact matching, regular expressions, and negative matching to provide flexible filtering capabilities.

Selector Type	Syntax	Example	Description
Basic Metric	<code>metric_name</code>	<code>cpu_usage</code>	Selects all series for the named metric
Label Exact Match	<code>metric_name{label="value"}</code>	<code>cpu_usage{host="web01"}</code>	Matches series with exact label value
Label Regex Match	<code>metric_name{label=~"pattern"}</code>	<code>cpu_usage{host=~"web.*"}</code>	Matches series with label matching regex
Label Not Equal	<code>metric_name{label!="value"}</code>	<code>cpu_usage{host!="localhost"}</code>	Excludes series with specific label value
Label Not Regex	<code>metric_name{label!~"pattern"}</code>	<code>cpu_usage{host!~"test.*"}</code>	Excludes series matching regex pattern
Multiple Labels	<code>metric_name{label1="value1", label2="value2"}</code>	<code>cpu_usage{host="web01", cpu="0"}</code>	All label conditions must match

Time Range Specification determines the temporal scope of the query, supporting both absolute timestamps and relative time expressions. The time range can be specified as part of the query string or provided separately by the query execution context (such as dashboard time controls).

Time Range Type	Syntax	Example	Description
Relative Range	<code>[5m]</code>	<code>cpu_usage[5m]</code>	Last 5 minutes of data
Absolute Range	<code>[2024-01-15T10:00:00Z:2024-01-15T11:00:00Z]</code>	<code>cpu_usage[2024-01-15T10:00:00Z:2024-01-15T11:00:00Z]</code>	Specific time window
Duration Units	<code>s, m, h, d</code>	<code>[30s], [2h], [7d]</code>	Seconds, minutes, hours, days

Aggregation Functions process the selected time-series data to produce meaningful results for visualization and alerting. These functions operate either across time (reducing a time-series to a single value) or across series (combining multiple time-series into fewer series).

Function Category	Function Name	Syntax	Description	Use Case
Statistical	avg	avg(cpu_usage)	Average across all selected series	Overall system load
Statistical	sum	sum(http_requests_total)	Sum across all selected series	Total request rate
Statistical	min	min(disk_free_bytes)	Minimum value across series	Worst-case capacity
Statistical	max	max(response_time)	Maximum value across series	Worst-case latency
Statistical	count	count(up)	Number of series with data	Service availability
Temporal	rate	rate(http_requests_total[5m])	Per-second rate of increase	Request rate calculation
Temporal	increase	increase(counter_metric[1h])	Total increase over time range	Growth measurement
Percentile	quantile	quantile(0.95, response_time)	Percentile across series	SLA measurements

Arithmetic Operations allow combining metrics and constants using basic mathematical operators, enabling the calculation of derived metrics and ratios that provide business context to raw measurements.

Operator	Syntax	Example	Description
Addition	+	cpu_usage + memory_usage	Add metrics together
Subtraction	-	disk_total - disk_used	Calculate differences
Multiplication	*	cpu_usage * 100	Scale by constant
Division	/	http_errors / http_total	Calculate ratios
Parentheses	()	(cpu_usage + memory_usage) / 2	Control operation order

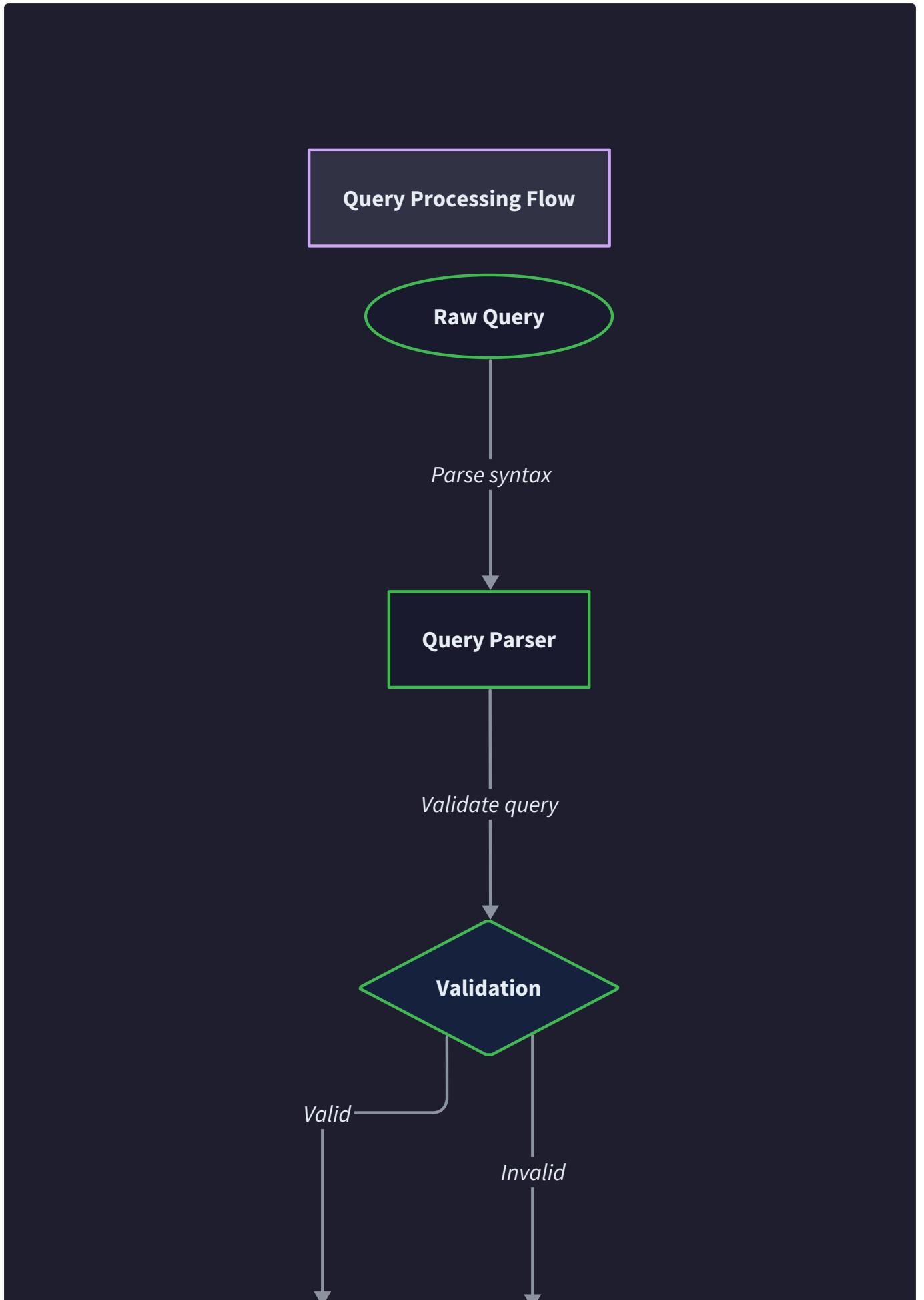
The query language supports function composition and chaining to build complex expressions from simple building blocks. For example, calculating an error rate percentage requires combining metric selection, rate calculation, and arithmetic:

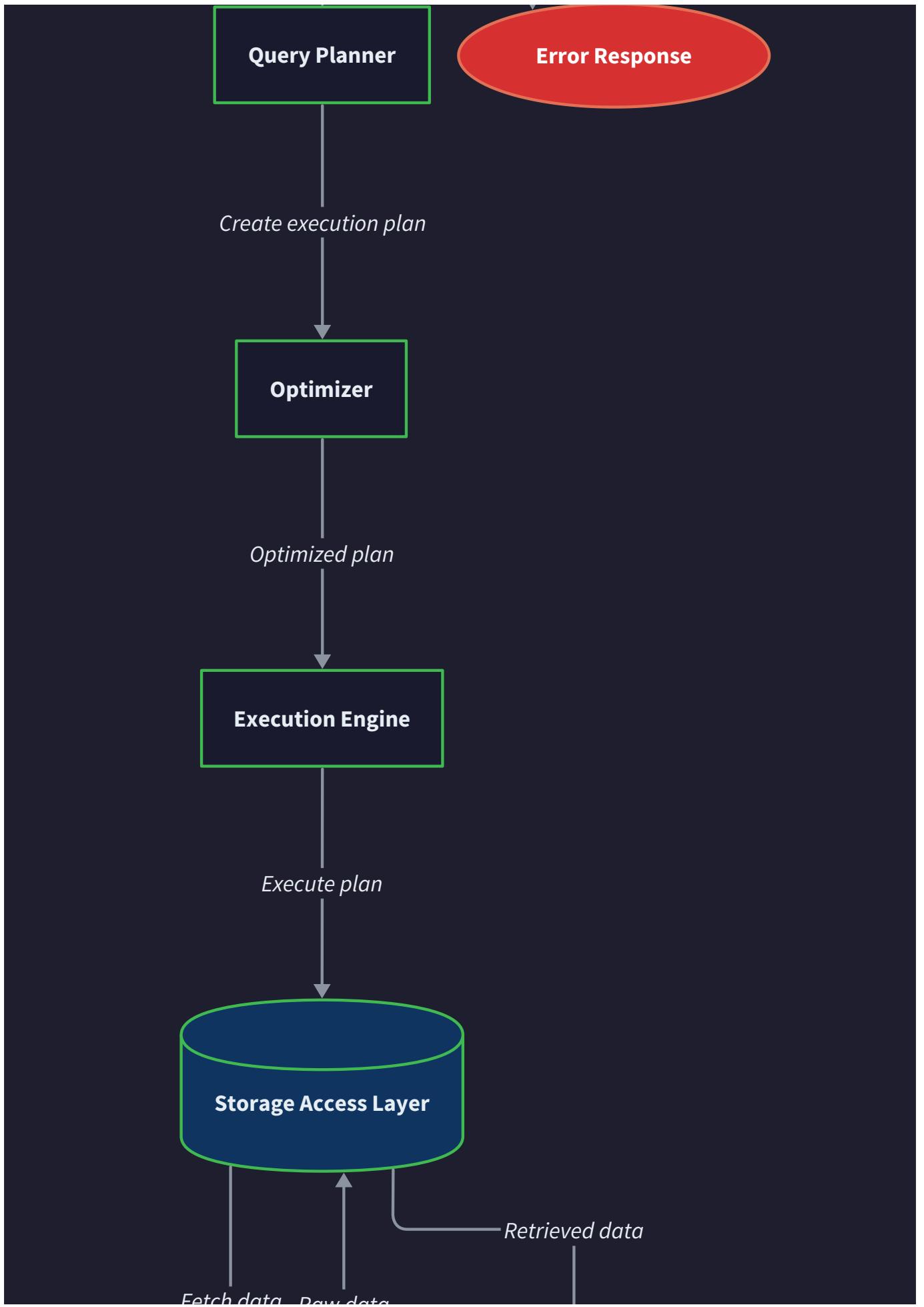
```
rate(http_errors_total[5m]) / rate(http_requests_total[5m]) * 100 .
```

The critical design insight is that query complexity should grow gradually - simple queries should have simple syntax, while complex analytics remain possible through composition rather than requiring entirely different syntax patterns.

Query Execution Pipeline

The query execution pipeline transforms user query strings into efficient operations against the time-series storage engine through a carefully orchestrated sequence of parsing, planning, and execution phases. This pipeline design separates concerns cleanly, enabling optimization opportunities while maintaining correctness guarantees.







Phase 1: Query Parsing and Validation converts the query string into a structured internal representation that can be processed programmatically. The parser employs a recursive descent approach that handles the nested nature of function calls and arithmetic expressions while providing clear error messages for syntax errors.

The parsing process follows these steps:

1. **Lexical Analysis:** The query string is tokenized into meaningful symbols - metric names, label selectors, function names, operators, and literals. This phase handles quoted strings, regular expressions, and numeric constants while detecting

malformed tokens.

2. **Syntax Analysis:** Tokens are organized into an Abstract Syntax Tree (AST) that represents the query structure. The parser validates that function calls have the correct number of arguments, operators are used in valid contexts, and parentheses are balanced.
3. **Semantic Validation:** The AST is analyzed for logical correctness - metric names are checked against known metrics, label names are validated, and function signatures are verified. This phase catches errors like attempting to calculate rates on gauge metrics or using undefined aggregation functions.
4. **Type Analysis:** Each node in the AST is annotated with its expected result type (scalar, vector, or matrix) to ensure operations are compatible. For example, arithmetic operations between vectors require matching label sets.

Parsing Component	Input	Output	Error Types Detected
Lexer	Raw query string	Token stream	Invalid characters, unterminated strings
Parser	Token stream	Abstract Syntax Tree	Syntax errors, malformed expressions
Semantic Analyzer	AST + Metric Metadata	Validated AST	Unknown metrics, invalid functions
Type Checker	Validated AST	Typed AST	Type mismatches, incompatible operations

Phase 2: Query Planning and Optimization analyzes the validated query to determine the most efficient execution strategy. The planner considers factors like storage layout, available indexes, and data locality to minimize the amount of data that must be read and processed.

The planning process generates an execution plan that specifies:

- Series Selection Strategy:** Determines which series indexes to consult and what filtering can be pushed down to the storage layer. For queries with highly selective label matchers, the planner can avoid reading irrelevant series entirely.
- Time Range Optimization:** Analyzes the query's time requirements to determine which storage blocks need to be accessed. Queries covering short recent time periods can skip older compacted blocks entirely.
- Aggregation Scheduling:** Decides whether aggregations can be performed incrementally as data is read (reducing memory usage) or need to buffer all data first (required for percentile calculations).
- Parallelization Opportunities:** Identifies independent operations that can be executed concurrently, such as processing different series or time ranges in parallel.

Planning Decision	Factors Considered	Optimization Applied
Index Usage	Label selectivity, cardinality	Choose most selective indexes first
Block Selection	Query time range, block boundaries	Skip blocks outside time range
Memory Management	Result set size, aggregation type	Stream processing vs. buffering
Parallelization	CPU cores, I/O patterns	Concurrent series processing

Phase 3: Storage Access and Data Retrieval executes the optimized query plan against the time-series storage engine, retrieving the minimum necessary data while maintaining consistency guarantees.

The execution engine coordinates several storage operations:

- Series Discovery:** Uses the series index to identify all time-series that match the query's label selectors. This phase leverages inverted indexes to quickly find candidate series without scanning all stored data.
- Block Access:** Reads the storage blocks that contain samples within the query's time range. The engine uses block metadata to skip blocks that don't contain relevant data and can read multiple blocks concurrently.
- Sample Filtering:** Applies additional filtering criteria that couldn't be pushed down to the storage layer, such as complex regular expression matches or range conditions.
- Data Streaming:** Organizes the retrieved samples into time-ordered streams that can be processed efficiently by the aggregation functions.

Phase 4: Aggregation and Result Computation applies the query's mathematical operations to the retrieved data, producing the final result set that will be returned to the caller.

Different aggregation types require different processing strategies:

- Series Aggregation:** Functions like `sum()` and `avg()` combine multiple time-series into fewer series. These operations must align timestamps across series and handle cases where series have different sampling intervals.
- Temporal Aggregation:** Functions like `rate()` and `increase()` analyze how individual time-series change over time. These operations must handle counter resets and missing data points appropriately.
- Statistical Functions:** Operations like `quantile()` require collecting all relevant data points before computation, using streaming algorithms when possible to manage memory usage.
- Arithmetic Operations:** Binary operations between metrics must align series by labels and handle cases where operand series don't have matching labels.

Aggregation Type	Processing Strategy	Memory Requirements	Special Considerations
Series Sum/Average	Streaming with timestamp alignment	$O(\text{time_points})$	Handle missing series data
Rate Calculation	Windowed streaming	$O(\text{window_size})$	Detect counter resets
Percentiles	Collect-then-compute	$O(\text{all_samples})$	May require approximate algorithms
Binary Arithmetic	Label-based matching	$O(\text{series_count})$	Handle label mismatches

Decision: Streaming vs. Buffering Execution

- Context:** Query results can range from a few data points to millions of samples, and available memory may be limited
- Options Considered:**
 - Always buffer all data in memory before processing
 - Always use streaming processing with fixed memory limits
 - Choose strategy based on query characteristics and system resources
- Decision:** Hybrid approach that uses streaming for compatible operations and buffering only when required
- Rationale:** Streaming enables processing datasets larger than available memory and provides consistent latency, while some operations (like percentiles) fundamentally require seeing all data
- Consequences:** Enables scaling to large datasets with predictable resource usage, but requires more complex execution engine implementation

Common Pitfalls

⚠ Pitfall: Timestamp Alignment Errors When aggregating multiple time-series, failing to properly align timestamps leads to incorrect results. This happens when developers assume all time-series have samples at identical timestamps, but in reality, metrics are collected at slightly different times. For example, averaging CPU usage across multiple hosts might miss samples or double-count values if timestamps aren't aligned to common intervals. The fix is to implement timestamp bucketing that groups nearby samples into common time slots before aggregation.

⚠ Pitfall: Counter Reset Handling Rate calculations become incorrect when counter metrics reset to zero (usually due to process restarts). A naive rate calculation sees a large negative change and produces meaningless results. The correct approach detects when the current value is less than the previous value and treats this as a reset, calculating the rate from the reset point rather than the previous sample.

⚠ Pitfall: Memory Explosion with High Cardinality Queries that select high-cardinality metrics without sufficient filtering can consume enormous amounts of memory. For example, querying all HTTP request metrics without filtering by endpoint or status code might return millions of time-series. The query engine must implement memory limits and provide clear error messages when queries would exceed available resources, guiding users to add more selective label filters.

⚠ Pitfall: Inefficient Range Query Patterns Repeatedly executing queries with small time ranges (like dashboard panels refreshing every 10 seconds) creates unnecessary load on the storage layer. Each query incurs parsing and planning overhead, and may read overlapping data. The solution is implementing query result caching and encouraging dashboard designs that use longer refresh intervals with appropriate time range selections.

⚠ Pitfall: Missing Data Point Handling Time-series data often has gaps due to collection failures or network issues. Aggregation functions that don't handle missing data appropriately can produce misleading results - for example, averaging over incomplete data sets or assuming zero values where no data exists. The query engine must distinguish between explicit zero values and missing data, providing options for interpolation, forwarding previous values, or excluding incomplete time periods from calculations.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Query Parser	Hand-written recursive descent parser	ANTLR or similar parser generator
Expression Evaluation	Direct AST traversal	Bytecode compilation with virtual machine
Result Serialization	JSON with standard library	MessagePack or Protocol Buffers
Query Caching	In-memory LRU cache	Redis with TTL-based invalidation
Concurrency Control	Worker pool with channels	Actor model with Akka-style messaging

File Structure

File	Description
internal/query/engine.go	← main QueryProcessor implementation
internal/query/parser.go	← query string parsing logic
internal/query/planner.go	← query optimization and planning
internal/query/executor.go	← execution engine coordination
internal/query/aggregation.go	← aggregation function implementations
internal/query/ast.go	← abstract syntax tree definitions
internal/query/cache.go	← query result caching
internal/query/engine_test.go	← comprehensive query engine tests
internal/query/testdata/	← test query files and expected results
internal/query/queries.json	← test cases with expected outputs

Query Parser Infrastructure

```
package query

import (
    "fmt"
    "regexp"
    "strconv"
    "strings"
    "time"
)

// TokenType represents different types of tokens in query strings

type TokenType int

const (
    TokenMetricName TokenType = iota
    TokenLabelName
    TokenString
    TokenRegex
    TokenNumber
    TokenFunction
    TokenOperator
    TokenLeftParen
    TokenRightParen
    TokenLeftBrace
    TokenRightBrace
    TokenLeftBracket
    TokenRightBracket
    TokenComma
    TokenEqual
    TokenNotEqual
    TokenRegexMatch
    TokenRegexNotMatch
)
```

GO

```
TokenEOF

TokenError

)

// Token represents a single token from query lexical analysis

type Token struct {

    Type      TokenType
    Value     string
    Position  int
}

// Lexer tokenizes query strings into structured tokens

type Lexer struct {

    input     string
    position int
    current   rune
}

// NewLexer creates a lexer for the given query string

func NewLexer(input string) *Lexer {
    l := &Lexer{
        input:     input,
        position: 0,
    }

    if len(input) > 0 {
        l.current = rune(input[0])
    }

    return l
}

// NextToken returns the next token from the input stream

func (l *Lexer) NextToken() Token {
    for l.current != 0 {
        if l.isWhitespace(l.current) {
```

```
    l.skipWhitespace()

    continue

}

switch l.current {

case '(':
    return l.singleCharToken(TokenLeftParen)

case ')':
    return l.singleCharToken(TokenRightParen)

case '{':
    return l.singleCharToken(TokenLeftBrace)

case '}':
    return l.singleCharToken(TokenRightBrace)

case '[':
    return l.singleCharToken(TokenLeftBracket)

case ']':
    return l.singleCharToken(TokenRightBracket)

case ',':
    return l.singleCharToken(TokenComma)

case '=':
    return l.handleEquals()

case '!':
    return l.handleNotEquals()

case '':
    return l.readString()

case '/':
    return l.readRegex()

default:
    if l.isLetter(l.current) {
        return l.readIdentifier()
    }

    if l.isDigit(l.current) {
```

```

        return l.readNumber()

    }

    return Token{Type: TokenError, Value: string(l.current), Position: l.position}

}

return Token{Type: TokenEOF, Position: l.position}

}

// QueryAST represents the parsed structure of a query

type QueryAST struct {

    Root ASTNode
}

// ASTNode represents a node in the query abstract syntax tree

type ASTNode interface {

    NodeType() string

    String() string
}

// MetricSelectorNode represents metric selection with label matchers

type MetricSelectorNode struct {

    MetricName    string

    LabelMatchers []LabelMatcher

    TimeRange     *TimeRangeNode
}

func (n *MetricSelectorNode) NodeType() string { return "MetricSelector" }

func (n *MetricSelectorNode) String() string {

    // Implementation for string representation

    return fmt.Sprintf("%s...", n.MetricName)
}

// FunctionCallNode represents aggregation function calls

type FunctionCallNode struct {

```

```
    FunctionName string

    Arguments      []ASTNode

    Parameters    map[string]interface{}

}

func (n *FunctionCallNode) NodeType() string { return "FunctionCall" }

func (n *FunctionCallNode) String() string {
    return fmt.Sprintf("%s(...)", n.FunctionName)
}

// BinaryOpNode represents arithmetic operations between expressions

type BinaryOpNode struct {

    Left      ASTNode

    Operator  string

    Right     ASTNode

}

func (n *BinaryOpNode) NodeType() string { return "BinaryOp" }

func (n *BinaryOpNode) String() string {
    return fmt.Sprintf("(%s %s %s)", n.Left, n.Operator, n.Right)
}

// TimeRangeNode represents time range specifications

type TimeRangeNode struct {

    Duration  time.Duration

    Start     *time.Time

    End       *time.Time

}

func (n *TimeRangeNode) NodeType() string { return "TimeRange" }

func (n *TimeRangeNode) String() string {
    if n.Duration > 0 {
        return fmt.Sprintf("[%s]", n.Duration)
    }
}
```

```
    return fmt.Sprintf("[%s:%s]", n.Start, n.End)  
}
```

Query Execution Engine Core

```
// QueryResult contains the result of query execution

type QueryResult struct {

    ResultType string           `json:"resultType" // "vector", "matrix", "scalar"`

    Data      interface{}       `json:"data"`

    Warnings  []string          `json:"warnings,omitempty"`

    Stats     *QueryExecutionStats `json:"stats,omitempty"`

}

// QueryExecutionStats provides metrics about query performance

type QueryExecutionStats struct {

    SeriesCount   int           `json:"seriesCount"`

    SamplesProcessed int         `json:"samplesProcessed"`

    ExecutionTime time.Duration `json:"executionTime"`

    BlocksRead    int           `json:"blocksRead"`

}

// ExecuteQuery processes a query string and returns results

// This is the main entry point that coordinates parsing, planning, and execution

func (qp *QueryProcessor) ExecuteQuery(ctx context.Context, queryString string) (QueryResult, error) {

    // TODO 1: Parse the query string into an AST

    // Hint: Use NewParser(queryString).Parse() and handle syntax errors


    // TODO 2: Validate the AST against available metrics and labels

    // Hint: Check metric names exist in storage, validate function signatures


    // TODO 3: Create an optimized execution plan

    // Hint: Analyze label selectivity, determine block access patterns


    // TODO 4: Execute the plan against storage engine

    // Hint: Retrieve samples, apply aggregations, handle errors


    // TODO 5: Format results according to query type
```

GO

```
// Hint: Convert internal format to JSON-serializable result

// TODO 6: Collect execution statistics for monitoring

// Hint: Track series count, samples processed, execution time

}

// PlanQuery analyzes a parsed query to create an optimized execution strategy

func (qp *QueryProcessor) PlanQuery(ctx context.Context, ast *QueryAST) (*ExecutionPlan, error) {

    // TODO 1: Analyze label matchers to estimate series selectivity

    // Hint: Query series index to count matching series before data retrieval

    // TODO 2: Determine optimal block access strategy

    // Hint: Check time range against block boundaries, skip unnecessary blocks

    // TODO 3: Plan aggregation execution order

    // Hint: Some aggregations can stream, others need all data collected first

    // TODO 4: Identify parallelization opportunities

    // Hint: Independent series can be processed concurrently

    // TODO 5: Set memory limits based on estimated result size

    // Hint: Calculate expected series count × time range to estimate memory usage

}

// ExecutionPlan contains optimized strategy for query execution

type ExecutionPlan struct {

    SeriesSelectors []SeriesSelector

    TimeRange       TimeRange

    AggregationPlan []AggregationStep

    MemoryLimit     int64

    Parallelism     int

}
```

```
// SeriesSelector defines how to efficiently find matching time-series

type SeriesSelector struct {

    MetricName      string

    LabelMatchers []LabelMatcher

    EstimatedSeries int

}

// AggregationStep defines one step in the aggregation pipeline

type AggregationStep struct {

    Function      string

    Parameters   map[string]interface{}

    InputType     string // "vector", "matrix", "scalar"

    OutputType    string

    CanStream    bool    // whether this step can process data incrementally

}
```

Aggregation Function Library

```
// AggregationFunction defines the interface for all aggregation operations          GO

type AggregationFunction interface {

    Name() string

    Aggregate(ctx context.Context, series []TimeSeries, params map[string]interface{}) ([]TimeSeries, error)

    CanStream() bool

    RequiredParameters() []string

}

// SumAggregation implements sum() function across multiple time-series

type SumAggregation struct{}


func (a *SumAggregation) Name() string { return "sum" }

func (a *SumAggregation) CanStream() bool { return true }

func (a *SumAggregation) RequiredParameters() []string { return []string{} }

func (a *SumAggregation) Aggregate(ctx context.Context, series []TimeSeries, params map[string]interface{}) ([]TimeSeries, error) {

    // TODO 1: Validate input series are compatible for summation

    // Hint: Check that series have overlapping time ranges


    // TODO 2: Create timestamp alignment buckets

    // Hint: Find common timestamp intervals across all input series


    // TODO 3: For each timestamp bucket, sum values from all series

    // Hint: Handle missing values appropriately (skip or interpolate)


    // TODO 4: Generate result time-series with combined labels

    // Hint: Remove labels that differ across input series


    // TODO 5: Handle streaming execution for large datasets

    // Hint: Process samples in time order without buffering entire series

}
```

```

// RateAggregation implements rate() function for counter metrics

type RateAggregation struct{}


func (a *RateAggregation) Name() string { return "rate" }

func (a *RateAggregation) CanStream() bool { return true }

func (a *RateAggregation) RequiredParameters() []string { return []string{"range"} }

func (a *RateAggregation) Aggregate(ctx context.Context, series []TimeSeries, params map[string]interface{}) ([]TimeSeries, error) {

    // TODO 1: Validate this is being applied to counter metrics only

    // Hint: Check metric type metadata or detect always-increasing values


    // TODO 2: For each series, calculate rate over the specified time range

    // Hint: rate = (current_value - previous_value) / time_difference


    // TODO 3: Handle counter resets (when current < previous)

    // Hint: When counter resets, use current_value / time_since_reset for rate


    // TODO 4: Apply rate calculation across the specified time window

    // Hint: Use the time range parameter to determine window size


    // TODO 5: Return new time-series with per-second rate values

    // Hint: Preserve original labels but update metric name to indicate rate

}

// QuantileAggregation implements quantile() function for percentile calculations

type QuantileAggregation struct{}


func (a *QuantileAggregation) Name() string { return "quantile" }

func (a *QuantileAggregation) CanStream() bool { return false } // needs all data for sorting

func (a *QuantileAggregation) RequiredParameters() []string { return []string{"q"} }

func (a *QuantileAggregation) Aggregate(ctx context.Context, series []TimeSeries, params map[string]interface{}) ([]TimeSeries, error) {

    // TODO 1: Extract quantile parameter (0.0 to 1.0)
}

```

```
// Hint: Validate q parameter is valid percentile value

// TODO 2: For each timestamp, collect all series values
// Hint: Align timestamps and gather values across all series

// TODO 3: Sort values at each timestamp
// Hint: Use sort.Float64s or implement streaming quantile algorithm for large datasets

// TODO 4: Calculate quantile value using interpolation
// Hint: For q=0.95, find value at position (count-1)*0.95 in sorted array

// TODO 5: Generate result series with quantile values over time
// Hint: Create single output series with quantile labels added

}
```

Query Result Caching System

```
// QueryCache provides caching for expensive query results          GO

type QueryCache struct {

    cache    map[string]*CacheEntry

    mu      sync.RWMutex

    maxSize  int64

    maxAge   time.Duration

    hitCount int64

    missCount int64

}

// CacheEntry represents a cached query result with metadata

type CacheEntry struct {

    Result    QueryResult

    QueryHash string

    CreatedAt time.Time

    AccessCount int64

    Size      int64

}

// NewQueryCache creates a query result cache with specified limits

func NewQueryCache(maxSize int64, maxAge time.Duration) *QueryCache {

    return &QueryCache{

        cache:    make(map[string]*CacheEntry),

        maxSize:  maxSize,

        maxAge:   maxAge,

    }

}

// Get retrieves cached query result if available and fresh

func (c *QueryCache) Get(ctx context.Context, queryString string, timeRange TimeRange) (QueryResult, bool) {

    // TODO 1: Generate cache key from query string and time range

    // Hint: Use hash of query + normalized time range for consistent keys
```

```

    // TODO 2: Check if cached entry exists and is still valid
    // Hint: Verify entry age is less than maxAge and data covers requested time range

    // TODO 3: Update access statistics for cache hit
    // Hint: Increment hit count and entry access count atomically

    // TODO 4: Return cached result if valid
    // Hint: Deep copy result to prevent caller modifications
}

// Put stores query result in cache with size limits

func (c *QueryCache) Put(ctx context.Context, queryString string, timeRange TimeRange, result QueryResult) {
    // TODO 1: Calculate result size for memory management
    // Hint: Estimate memory usage of result data structures

    // TODO 2: Check if adding this entry would exceed cache limits
    // Hint: Implement LRU eviction if cache is full

    // TODO 3: Store entry with metadata
    // Hint: Include creation time, access count, and size information

    // TODO 4: Update cache statistics
    // Hint: Track cache size, entry count, and miss count
}

```

Milestone Checkpoint

After implementing the Query Engine, verify the following behaviors:

Basic Query Execution: Start your metrics system and ingest sample data with multiple time-series. Execute a simple query like `cpu_usage{host="web01"}` and verify it returns the correct samples within the requested time range.

Aggregation Functions: Test aggregation functions by executing `sum(cpu_usage)` and `avg(cpu_usage)` queries. Verify that the results correctly combine multiple time-series and handle timestamp alignment appropriately.

Rate Calculations: For counter metrics, test `rate(http_requests_total[5m])` and verify it correctly calculates per-second rates while handling counter resets appropriately.

Error Handling: Submit invalid queries with syntax errors, unknown metrics, and incompatible operations. Verify that the query engine returns helpful error messages rather than crashing.

Performance Testing: Execute queries against datasets with varying cardinalities (100 series, 1000 series, 10000 series) and measure response times. Query execution should remain responsive even with high-cardinality metrics.

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Queries return empty results	Metric names or labels don't match stored data	Check series index for available metrics and labels	Verify query selectors match actual stored data
Aggregation results are incorrect	Timestamp alignment issues or missing data handling	Log timestamp distribution across input series	Implement proper timestamp bucketing and interpolation
Rate calculations show negative values	Counter resets not detected properly	Check for decreasing values in counter series	Add counter reset detection in rate function
Query execution is extremely slow	Reading too much data or inefficient aggregation	Profile query execution and check block access patterns	Add more selective label filters or optimize storage access
Memory usage grows without bounds	High cardinality queries without limits	Monitor memory allocation during query processing	Implement memory limits and streaming aggregation
Cached results are stale or incorrect	Cache invalidation logic has bugs	Check cache hit/miss ratios and entry timestamps	Fix cache key generation and expiration logic

Visualization Dashboard

Milestone(s): 3 (Visualization Dashboard) - This section details the web-based dashboard system that provides real-time metric visualization with configurable panels and sharing capabilities

Mental Model: The Mission Control Center

Think of a metrics dashboard as NASA's mission control center during a space launch. The mission control center has multiple large displays, each showing different aspects of the mission - rocket telemetry, weather conditions, trajectory calculations, and system health indicators. Each display is carefully positioned and sized based on its importance and the information it conveys. Mission controllers can quickly glance across all displays to understand the complete system state, and critical alerts are highlighted prominently to demand immediate attention.

Your metrics dashboard operates on the same principle. Each panel acts like one of those mission control displays, showing a specific aspect of your system's health through charts and graphs. Just as mission control displays update in real-time as new telemetry data arrives from the spacecraft, your dashboard panels refresh continuously as new metric samples flow into the storage engine. The dashboard's layout and panel configuration serve as the "mission plan" - defining what information to display, where to position it, and how frequently to update it.

The mission control analogy extends to user interaction patterns. Mission controllers don't constantly reconfigure their displays during critical operations - they rely on pre-configured views that show the right information at the right time. Similarly, effective dashboard design involves creating persistent configurations that teams can rely on during incident response. The ability to share dashboard views mirrors how mission control can broadcast their displays to other teams who need situational awareness.

This mental model helps us understand why dashboard performance and reliability are critical. Just as mission control cannot afford display failures during launch sequences, your metrics dashboard must maintain consistent performance even when visualizing thousands of time-series or handling rapid data updates. The real-time nature of the system means that stale or missing data can lead to incorrect operational decisions.

Dashboard Configuration System

The dashboard configuration system serves as the blueprint that transforms raw time-series data into meaningful visual representations. This system must balance flexibility with simplicity, allowing users to create sophisticated visualizations without requiring deep technical knowledge of the underlying query language or storage internals.

Dashboard Entity Structure

The core dashboard entity encapsulates all information needed to render and maintain a complete dashboard view. The relationship between dashboards and panels follows a hierarchical ownership model where each dashboard owns its panel configurations completely.

Field	Type	Description
ID	string	Unique identifier for the dashboard, used for persistence and sharing
Title	string	Human-readable dashboard name displayed in the interface
Description	string	Optional detailed description explaining the dashboard's purpose
Tags	[]string	Searchable keywords for dashboard categorization and discovery
Panels	[]Panel	Complete list of visualization panels with their configurations
TimeRange	TimeRange	Global time window applied to all panels unless overridden
RefreshInterval	time.Duration	Automatic update frequency for all panels in the dashboard
Editable	bool	Whether users can modify this dashboard configuration

Panel Configuration Architecture

Each panel represents an independent visualization component with its own query, display options, and positioning information. The panel system uses a plugin-like architecture where different panel types (line charts, bar charts, single-stat displays) share common configuration patterns while allowing type-specific customization.

Field	Type	Description
ID	string	Unique panel identifier within the dashboard scope
Title	string	Panel header text displayed above the visualization
Type	string	Panel renderer type (linechart, barchart, singlestat, table)
GridPos	GridPosition	Panel size and position within the dashboard grid layout
Query	PanelQuery	Metric query definition and time range specifications
Options	map[string]interface{}	Type-specific rendering options and styling preferences
AlertRule	*PanelAlertRule	Optional alert threshold configuration tied to this panel

The `GridPosition` structure implements a responsive grid system that ensures consistent layout across different screen sizes while allowing precise positioning control.

Field	Type	Description
X	int	Horizontal grid position (0-based, left to right)
Y	int	Vertical grid position (0-based, top to bottom)
Width	int	Panel width in grid units (typically 1-12 columns)
Height	int	Panel height in grid units (affects chart detail level)

Query Integration Architecture

The panel query system bridges the dashboard layer with the underlying query engine, translating user-friendly configuration into executable query plans. This abstraction allows dashboard users to work with intuitive concepts like "metric name" and "time range" without understanding the complexities of query optimization or storage access patterns.

Field	Type	Description
Expression	string	Query expression using the system's query language syntax
TimeRange	*TimeRange	Panel-specific time range override (inherits from dashboard if nil)
RefreshInterval	*time.Duration	Panel-specific refresh rate override
MaxDataPoints	int	Maximum number of data points to retrieve (controls resolution)
FormatAs	string	Result format preference (timeseries, table, scalar)

Decision: JSON-Based Configuration Storage

- **Context:** Dashboard configurations need persistent storage with human-readable format for version control and manual editing
- **Options Considered:** Binary protocol buffers, YAML files, JSON documents, SQL schema
- **Decision:** JSON documents stored as files with optional database backend
- **Rationale:** JSON provides excellent tooling support, version control compatibility, and direct JavaScript integration for the web interface
- **Consequences:** Enables easy dashboard sharing and backup, but requires careful schema versioning for backward compatibility

Configuration Validation and Normalization

The dashboard configuration system implements comprehensive validation to prevent runtime errors and ensure consistent behavior across different deployment environments. Validation occurs at multiple layers: structural validation during JSON parsing, semantic validation during dashboard loading, and runtime validation during query execution.

Dashboard validation encompasses several critical areas:

1. **Structural Validation:** Verifies that required fields are present, data types match expectations, and nested objects conform to their schemas
2. **Query Validation:** Ensures that panel queries use valid syntax and reference existing metrics
3. **Layout Validation:** Confirms that panel grid positions don't create overlapping layouts or exceed reasonable size constraints
4. **Reference Validation:** Checks that alert rule references point to valid notification channels and use appropriate threshold operators

The validation system produces detailed error reports that help users identify and fix configuration problems quickly. Rather than failing silently or with cryptic error messages, the system provides specific field-level feedback with suggested corrections.

Template and Variable System

Advanced dashboard configurations benefit from templating capabilities that allow dynamic metric selection and parameterized queries. This system enables creation of reusable dashboard templates that work across different environments or services without requiring manual reconfiguration.

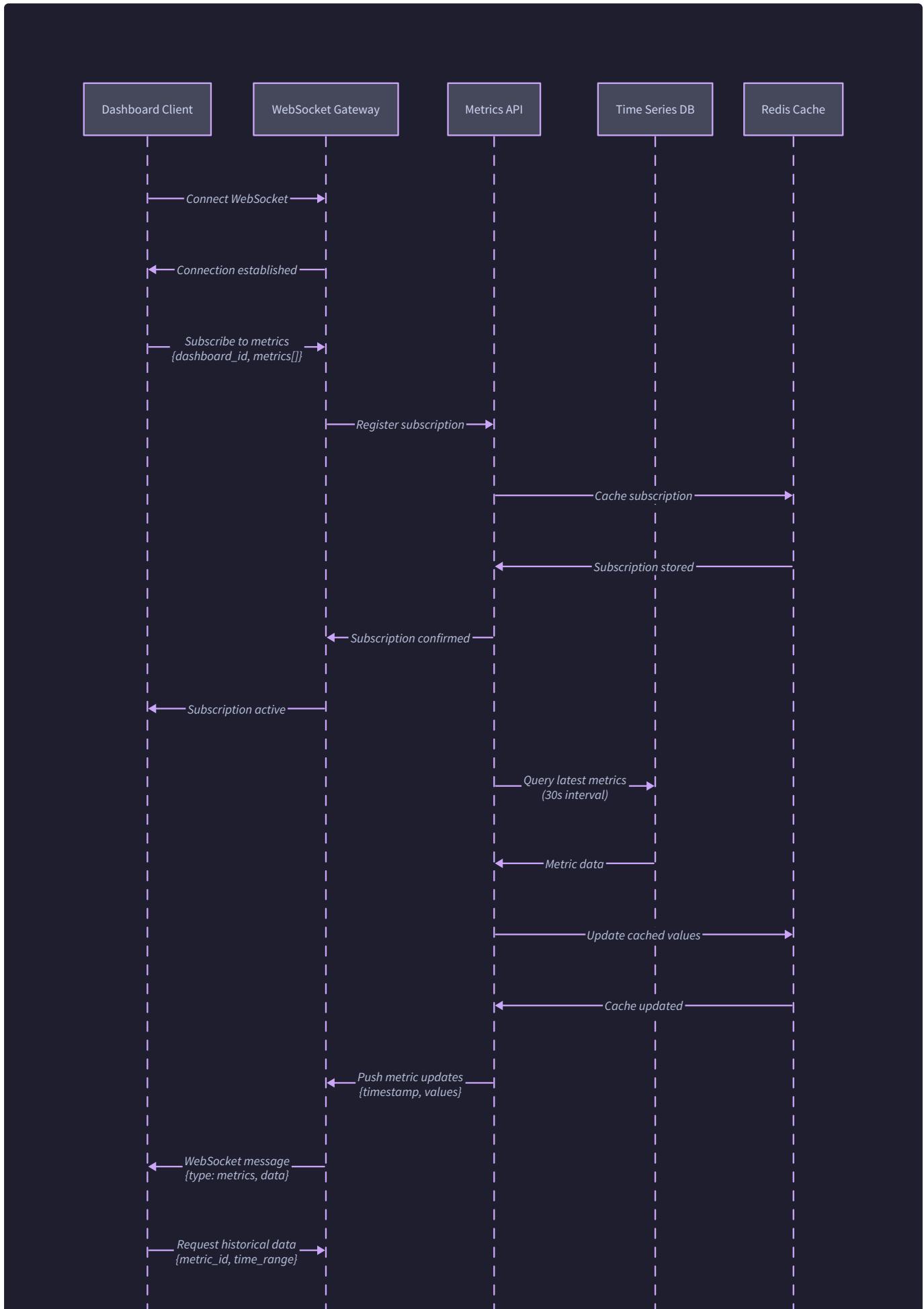
Dashboard variables support several interpolation patterns:

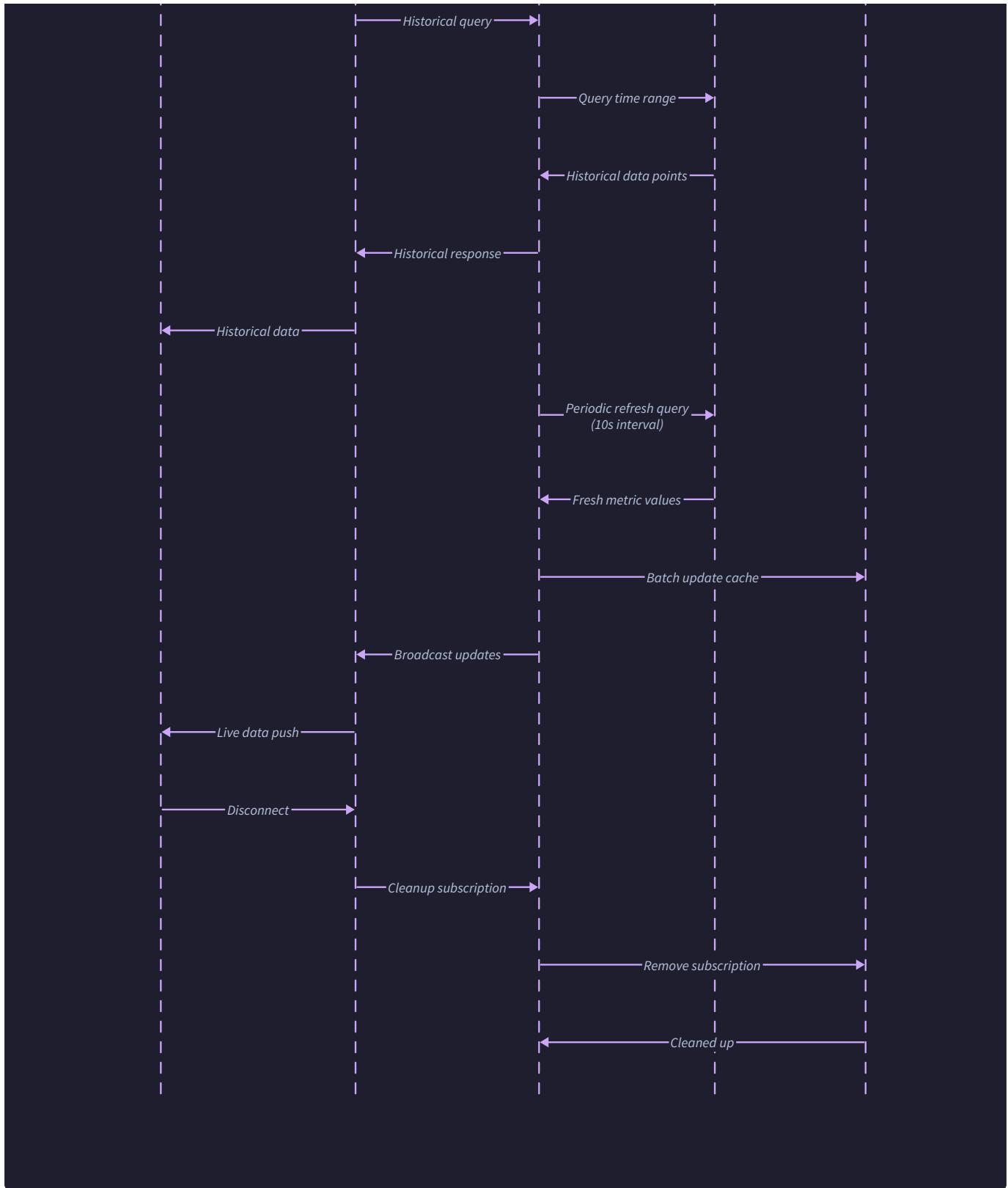
1. **Metric Name Variables**: Allow selection from available metrics matching a pattern
2. **Label Value Variables**: Provide dropdown selection of label values for filtering
3. **Time Range Variables**: Enable quick switching between common time windows
4. **Custom Variables**: Support arbitrary string substitution for advanced use cases

The template engine processes variable substitution during query generation, ensuring that the underlying query engine receives fully-resolved expressions while maintaining the user-friendly parameterized interface.

Real-Time Data Updates

Real-time dashboard updates represent one of the most technically challenging aspects of the visualization system. The architecture must efficiently push fresh data to multiple concurrent dashboard viewers while managing resource consumption and maintaining consistent user experience across varying network conditions.





WebSocket Communication Architecture

The real-time update system uses WebSocket connections to establish persistent, bidirectional communication channels between the dashboard server and client browsers. This approach provides significantly lower latency compared to HTTP polling while reducing server resource consumption through connection reuse.

The WebSocket protocol implementation handles several critical responsibilities:

- 1. Connection Lifecycle Management:** Establishes connections during dashboard load, maintains heartbeat monitoring, and gracefully handles disconnections with automatic reconnection logic

2. **Message Routing:** Distributes metric updates only to clients displaying relevant dashboards, avoiding unnecessary data transmission
3. **Update Scheduling:** Coordinates refresh cycles across panels to prevent overwhelming clients with excessive update frequencies
4. **Backpressure Handling:** Implements client-side buffering and server-side rate limiting when update rates exceed client processing capacity

Update Subscription Model

Each dashboard client establishes subscriptions for the specific metrics and time ranges displayed in its panels. The subscription system tracks active queries and their refresh requirements, enabling the server to batch related updates and minimize redundant query execution.

Subscription Component	Purpose	Key Attributes
Query Fingerprint	Identifies unique query/time-range combinations	Hash of expression, labels, time window
Refresh Schedule	Determines update timing for each subscription	Interval, next execution time, jitter offset
Client Set	Tracks which clients need specific updates	Connection IDs, panel mappings, active status
Result Cache	Stores recent query results for immediate delivery	Timestamped data, expiration time, memory limit

The subscription model optimizes resource usage by sharing query results across multiple clients viewing identical data. When multiple users access the same dashboard, the system executes each unique query only once per refresh cycle, then distributes results to all interested clients.

Incremental Update Strategy

Rather than retransmitting complete datasets on every refresh, the system implements incremental updates that send only new or changed data points. This approach dramatically reduces bandwidth consumption and improves client-side rendering performance, especially for dashboards displaying long time ranges with high-frequency updates.

The incremental update algorithm works through a multi-step process:

- Baseline Establishment:** New clients receive complete datasets for initial rendering, establishing a known synchronization point
- Delta Computation:** Subsequent updates calculate differences between current and previously transmitted data
- Efficient Encoding:** Delta messages use compact representation focusing on timestamp ranges and value changes
- Client-Side Merging:** Dashboard clients merge incremental updates with existing datasets, maintaining complete time-series views
- Periodic Resynchronization:** Full dataset refresh occurs periodically to prevent drift from accumulated deltas

Adaptive Update Frequency

The real-time update system implements adaptive refresh rates that balance data freshness with system performance. Rather than using fixed update intervals, the system monitors several factors to determine optimal refresh timing for each dashboard and panel combination.

Adaptive frequency calculation considers:

- Data Velocity:** Metrics with higher ingestion rates benefit from more frequent updates
- Panel Visibility:** Off-screen or inactive panels receive lower-priority update scheduling
- Client Performance:** Slow client connections or processing delays trigger automatic rate reduction
- System Load:** High query engine utilization results in reduced refresh frequencies across all clients
- User Preferences:** Explicit refresh rate settings override automatic calculations when specified

Decision: WebSocket-Based Real-Time Updates

- Context:** Dashboard users need immediate visibility into metric changes without manual refresh actions
- Options Considered:** HTTP polling, Server-Sent Events (SSE), WebSocket connections, UDP broadcast
- Decision:** WebSocket connections with intelligent subscription management
- Rationale:** WebSockets provide bidirectional communication needed for subscription management while maintaining low latency and efficient resource usage
- Consequences:** Enables true real-time experience but requires connection state management and fallback handling for network disruptions

Error Handling and Resilience

Real-time update systems must gracefully handle various failure modes without disrupting the overall dashboard experience. The architecture implements comprehensive error handling that maintains service availability even when individual components experience problems.

Connection-level error handling addresses network disruptions, client disconnections, and server restart scenarios. The client-side implementation includes exponential backoff reconnection logic that prevents overwhelming the server during widespread connectivity issues. Server-side connection tracking removes stale subscriptions and reclaims associated resources when clients disconnect unexpectedly.

Query-level error handling ensures that problems with individual panel queries don't affect other dashboard components. Failed queries trigger client-side error displays while continuing to update successful panels normally. The system maintains error state information to provide meaningful feedback about persistent query problems.

Implementation Guidance

This section provides concrete implementation details for building the dashboard visualization system, focusing on the web server architecture, real-time communication, and client-side rendering components.

Technology Recommendations

Component	Simple Option	Advanced Option
Web Server	net/http with gorilla/mux router	Gin or Echo framework with middleware
WebSocket	gorilla/websocket library	Custom WebSocket with compression
Frontend	Vanilla JavaScript + Chart.js	React/Vue with D3.js or Plotly
Static Assets	Embedded with go:embed	CDN with build pipeline
Configuration	JSON files with validation	YAML with schema validation

File Structure

```
project-root/
  cmd/
    dashboard-server/
      main.go           ← Dashboard server entry point
  internal/
    dashboard/
      server.go        ← HTTP server and WebSocket handling
      config.go         ← Dashboard configuration management
      subscription.go   ← Real-time update subscriptions
      panel.go          ← Panel rendering and query execution
      validation.go     ← Configuration validation logic
      templates.go      ← Variable substitution and templating
    dashboard/handlers/
      dashboard_handlers.go  ← REST API for dashboard CRUD operations
      websocket_handlers.go  ← WebSocket connection management
      static_handlers.go     ← Static asset serving
    dashboard/models/
      dashboard.go        ← Dashboard and Panel data structures
      subscription.go     ← Subscription and update models
  web/
    static/
      js/
        dashboard.js      ← Dashboard client-side logic
        chart-renderer.js  ← Chart rendering with Chart.js
        websocket-client.js  ← WebSocket communication handling
      css/
        dashboard.css      ← Dashboard styling
        index.html          ← Main dashboard interface
  configs/
    dashboard-server.yaml  ← Dashboard server configuration
```

Dashboard Server Infrastructure

```
package dashboard

import (
    "context"
    "encoding/json"
    "log/slog"
    "net/http"
    "sync"
    "time"

    "github.com/gorilla/mux"
    "github.com/gorilla/websocket"
)

// DashboardServer manages the web interface and real-time updates for metrics visualization

type DashboardServer struct {
    config          *ServerConfig
    logger         *slog.Logger
    queryProcessor QueryProcessor

    // WebSocket management
    upgrader      websocket.Upgrader
    clients       map[string]*WebSocketClient
    subscriptions *SubscriptionManager
    clientsMu     sync.RWMutex

    // Dashboard storage
    dashboards    map[string]*Dashboard
    dashboardsMu  sync.RWMutex

    server        *http.Server
    shutdown      chan struct{}}

}
```

GO

```

// WebSocketClient represents a connected dashboard client

type WebSocketClient struct {

    ID          string
    conn        *websocket.Conn
    send        chan []byte
    subscriptions map[string]*PanelSubscription
    mu          sync.RWMutex
}

// PanelSubscription tracks a client's interest in specific panel data

type PanelSubscription struct {

    PanelID      string
    Query        string
    RefreshInterval time.Duration
    LastUpdate   time.Time
    TimeRange    TimeRange
}

// NewDashboardServer creates a new dashboard server instance

func NewDashboardServer(config *ServerConfig, queryProcessor QueryProcessor, logger *slog.Logger) *DashboardServer {
    return &DashboardServer{
        config:      config,
        logger:      logger,
        queryProcessor: queryProcessor,
        upgrader:    websocket.Upgrader{
            CheckOrigin: func(r *http.Request) bool { return true }, // Configure appropriately for
            production
        },
        clients:     make(map[string]*WebSocketClient),
        subscriptions: NewSubscriptionManager(),
        dashboards:   make(map[string]*Dashboard),
        shutdown:    make(chan struct{}),
    }
}

```

```
}

// Start initializes the dashboard server and begins serving requests

func (ds *DashboardServer) Start() error {

    // TODO 1: Load existing dashboard configurations from storage

    // TODO 2: Set up HTTP routes for dashboard CRUD operations and static assets

    // TODO 3: Initialize WebSocket upgrade handler for real-time connections

    // TODO 4: Start subscription manager for coordinating panel updates

    // TODO 5: Begin periodic cleanup of stale connections and subscriptions

    // TODO 6: Start HTTP server on configured port with timeout settings

    // Hint: Use gorilla/mux for routing and implement graceful shutdown

}

// Stop gracefully shuts down the dashboard server

func (ds *DashboardServer) Stop(ctx context.Context) error {

    // TODO 1: Signal shutdown to all background goroutines

    // TODO 2: Close all active WebSocket connections with proper close frames

    // TODO 3: Save any modified dashboard configurations to persistent storage

    // TODO 4: Shutdown HTTP server with context timeout for graceful connection draining

    // TODO 5: Wait for all goroutines to complete or context timeout

}
```

WebSocket Connection Management

```
// HandleWebSocketUpgrade processes WebSocket connection requests from dashboard clients
```

GO

```
func (ds *DashboardServer) HandleWebSocketUpgrade(w http.ResponseWriter, r *http.Request) {
```

```
    // TODO 1: Upgrade HTTP connection to WebSocket using ds.upgrader
```

```
    // TODO 2: Generate unique client ID and create WebSocketClient instance
```

```
    // TODO 3: Register client in ds.clients map with proper locking
```

```
    // TODO 4: Start goroutines for reading client messages and writing updates
```

```
    // TODO 5: Handle client disconnection cleanup in defer statement
```

```
    // Hint: Use separate goroutines for reading and writing to avoid deadlocks
```

```
}
```

```
// HandleClientMessage processes incoming messages from WebSocket clients
```

```
func (ds *DashboardServer) HandleClientMessage(client *WebSocketClient, messageType int, data []byte) error {
```

```
    // TODO 1: Parse message JSON to determine message type (subscribe, unsubscribe, ping)
```

```
    // TODO 2: For subscribe messages, create PanelSubscription and register with SubscriptionManager
```

```
    // TODO 3: For unsubscribe messages, remove subscription and clean up resources
```

```
    // TODO 4: For ping messages, respond with pong to maintain connection health
```

```
    // TODO 5: Log any parsing errors and send error response to client
```

```
    // Hint: Define message types as constants and use type switches for handling
```

```
}
```

```
// BroadcastPanelUpdate sends updated data to all clients subscribed to a specific panel
```

```
func (ds *DashboardServer) BroadcastPanelUpdate(panelID string, data *PanelUpdateMessage) error {
```

```
    // TODO 1: Find all clients subscribed to this panel ID
```

```
    // TODO 2: Serialize update message to JSON format
```

```
    // TODO 3: Send message to each client's send channel (non-blocking)
```

```
    // TODO 4: Remove clients with full send channels (indicates slow/disconnected clients)
```

```
    // TODO 5: Log broadcast statistics for monitoring purposes
```

```
    // Hint: Use select with default case for non-blocking channel sends
```

```
}
```

Dashboard Configuration Management

GO

```
// LoadDashboard retrieves a dashboard configuration by ID

func (ds *DashboardServer) LoadDashboard(dashboardID string) (*Dashboard, error) {

    // TODO 1: Check in-memory cache first with read lock

    // TODO 2: If not cached, load from persistent storage (file or database)

    // TODO 3: Validate dashboard configuration using validation.go functions

    // TODO 4: Cache validated dashboard in memory with write lock

    // TODO 5: Return dashboard or appropriate error if not found/invalid

    // Hint: Use sync.RWMutex for efficient concurrent access to dashboard cache

}

// SaveDashboard persists a dashboard configuration

func (ds *DashboardServer) SaveDashboard(dashboard *Dashboard) error {

    // TODO 1: Validate complete dashboard configuration including all panels

    // TODO 2: Generate unique ID if this is a new dashboard

    // TODO 3: Write dashboard JSON to persistent storage with atomic operations

    // TODO 4: Update in-memory cache with write lock

    // TODO 5: Notify any clients viewing this dashboard about configuration changes

    // Hint: Use temporary files and atomic rename for safe persistence

}

// ValidateDashboard checks dashboard configuration for correctness

func (ds *DashboardServer) ValidateDashboard(dashboard *Dashboard) error {

    // TODO 1: Validate required fields (ID, Title) are non-empty

    // TODO 2: Check TimeRange values are reasonable (From before To, not too far in past/future)

    // TODO 3: Validate each panel configuration including GridPosition bounds

    // TODO 4: Parse and validate all panel queries using query engine

    // TODO 5: Check for panel ID uniqueness within the dashboard

    // Hint: Return ValidationErrors with specific field-level error messages

}
```

Client-Side Dashboard Implementation

```
// web/static/js/dashboard.js
```

JAVASCRIPT

```
class DashboardClient {

  constructor(dashboardId) {

    this.dashboardId = dashboardId;

    this.panels = new Map();

    this.websocket = null;

    this.reconnectAttempts = 0;

    this.maxReconnectAttempts = 5;

    this.reconnectDelay = 1000; // Start with 1 second delay

  }

  // Initialize dashboard and establish WebSocket connection

  async initialize() {

    // TODO 1: Load dashboard configuration from REST API

    // TODO 2: Create and render all panels based on configuration

    // TODO 3: Establish WebSocket connection for real-time updates

    // TODO 4: Subscribe to updates for all visible panels

    // TODO 5: Set up periodic connection health checks

    // Hint: Use fetch() for REST calls and native WebSocket API

  }

  // Handle incoming WebSocket messages with panel updates

  handleWebSocketMessage(event) {

    // TODO 1: Parse JSON message and determine message type

    // TODO 2: For panel updates, find corresponding panel and update data

    // TODO 3: For error messages, display user-friendly error in affected panel

    // TODO 4: For connection status changes, update UI connection indicator

    // TODO 5: Log any parsing errors to browser console for debugging

    // Hint: Use try-catch around JSON.parse() for robust error handling

  }

}
```

```
// Reconnect WebSocket with exponential backoff

reconnectWebSocket() {

    // TODO 1: Check if reconnection attempts exceed maximum limit

    // TODO 2: Calculate delay using exponential backoff with jitter

    // TODO 3: Schedule reconnection attempt using setTimeout

    // TODO 4: Reset reconnect counter on successful connection

    // TODO 5: Update UI to show connection status during reconnection

    // Hint: Use Math.random() for jitter to avoid thundering herd

}

}

class DashboardPanel {

    constructor(panelConfig, container) {

        this.config = panelConfig;

        this.container = container;

        this.chart = null;

        this.lastUpdate = null;

    }

    // Render panel with Chart.js or similar library

    render() {

        // TODO 1: Create canvas element for chart rendering

        // TODO 2: Initialize Chart.js with panel-specific configuration

        // TODO 3: Configure chart options based on panel type (line, bar, etc.)

        // TODO 4: Set up chart update methods for real-time data

        // TODO 5: Handle panel resize events for responsive layout

        // Hint: Use Chart.js responsive configuration for automatic resizing

    }

    // Update panel with new data from WebSocket

    updateData(newData) {
```

```

    // TODO 1: Validate incoming data format matches panel expectations

    // TODO 2: Merge new data with existing chart dataset

    // TODO 3: Update chart using Chart.js update() method

    // TODO 4: Update last update timestamp for debugging

    // TODO 5: Handle any rendering errors gracefully with fallback display

    // Hint: Use chart.update('none') for performance when data changes frequently

}

}

```

Milestone Checkpoint

After implementing the dashboard visualization system, verify the following behaviors:

- Dashboard Configuration:** Create a simple dashboard JSON file with 2-3 panels. Load it through the REST API and verify all panels render correctly with proper positioning.
- Real-Time Updates:** Start the dashboard server, open a dashboard in a browser, and inject new metrics through the ingestion API. Verify that charts update automatically without page refresh within the configured refresh interval.
- WebSocket Connectivity:** Use browser developer tools to monitor WebSocket traffic. Verify that subscription messages are sent when dashboards load and that panel updates arrive as JSON messages.
- Multiple Clients:** Open the same dashboard in multiple browser tabs or different browsers. Verify that all clients receive updates simultaneously and that server resource usage remains reasonable.
- Error Handling:** Stop the metrics ingestion engine while keeping the dashboard open. Verify that panels display appropriate error states and recover automatically when ingestion resumes.

Expected command output for successful implementation:

```
$ go run cmd/dashboard-server/main.go
2024/01/15 10:30:00 INFO Dashboard server starting port=3000
2024/01/15 10:30:00 INFO WebSocket handler registered endpoint=/ws
2024/01/15 10:30:00 INFO Static assets served path=/static/
2024/01/15 10:30:01 INFO Dashboard server ready for connections

# In another terminal, test dashboard loading

$ curl http://localhost:3000/api/dashboards/test-dashboard
{"id": "test-dashboard", "title": "System Overview", "panels": [...]}
```

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Charts not updating in real-time	WebSocket connection not established	Check browser console for WebSocket errors	Verify WebSocket endpoint URL and CORS configuration
High CPU usage during updates	Too frequent refresh intervals	Monitor WebSocket message frequency	Implement adaptive refresh rates based on client performance
Panels showing "No Data"	Query execution errors	Check server logs for query parsing failures	Validate panel queries against available metrics
Dashboard layout broken	Grid position conflicts	Check panel GridPosition values for overlaps	Implement layout validation during dashboard save
Memory leaks in browser	Chart.js instances not destroyed	Monitor browser memory usage over time	Call chart.destroy() when panels are removed
WebSocket connections dropping	Network issues or server overload	Check WebSocket close codes and server metrics	Implement exponential backoff reconnection logic

Alerting System

Milestone(s): 4 (Alerting System) - This section details the alerting system that monitors metric data in real-time, evaluates threshold conditions, manages alert states, and delivers notifications through multiple channels

Mental Model: The Security Guard System

Think of the alerting system as a sophisticated security monitoring service that guards a large corporate campus. Just like a security system, our alerting infrastructure operates on multiple layers of vigilance and response.

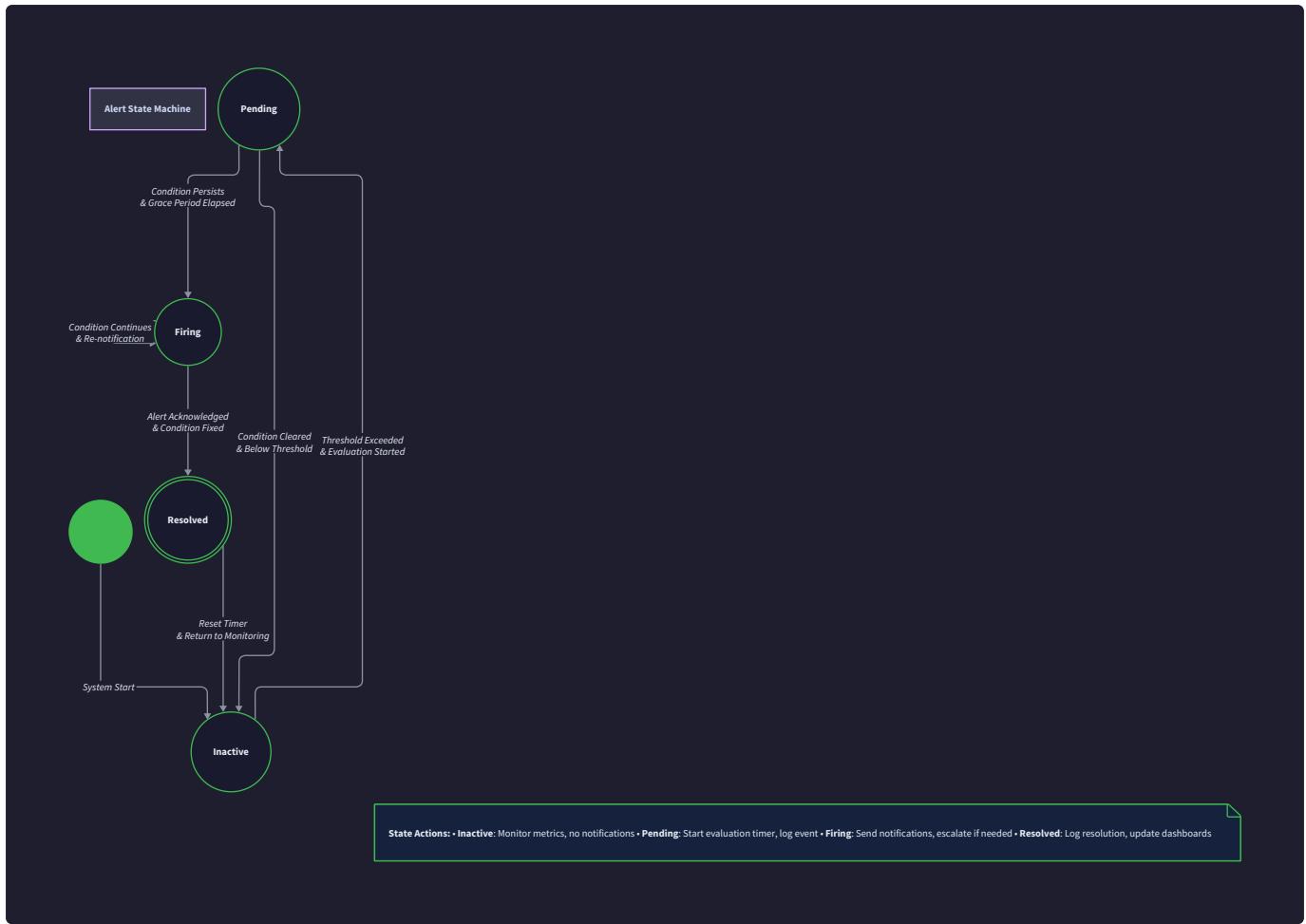
The **security guards** are our `AlertEvaluator` instances - they patrol their assigned zones (metric queries) on regular schedules, constantly checking for anomalies. Each guard has a **patrol route** (alert rule expression) that defines which areas to monitor and what constitutes suspicious activity (threshold conditions). When a guard notices something unusual - say, a door that should be locked is open (CPU usage above 90%) - they don't immediately sound the alarm. Instead, they watch carefully for a specified duration to ensure it's not a false alarm (evaluation period).

The **dispatch center** is our alert state management system. When guards report suspicious activity, the dispatch center tracks the evolving situation through distinct phases: **investigating** (pending state), **confirmed incident** (firing state), and **all clear** (resolved state). The dispatch center maintains detailed incident logs, ensuring that every status change is properly documented and that appropriate personnel are notified at each stage.

The **communication network** represents our notification channels - some incidents require immediate phone calls to on-duty personnel (PagerDuty for critical alerts), while others might warrant email reports to the facilities team (email for warnings), and some situations need real-time updates in the security chat room (Slack integration). Each communication channel has its own protocols, message formats, and delivery confirmation requirements.

Most importantly, the system includes **intelligent filtering** to prevent alert fatigue - just like how a security system shouldn't wake the entire building every time a cat triggers a motion sensor. This involves proper alert prioritization, silence periods during maintenance, and sophisticated grouping to avoid overwhelming responders with duplicate notifications.

This mental model helps us understand that alerting isn't just about detecting problems - it's about managing the entire lifecycle of incident awareness, from initial detection through resolution, while maintaining the trust and attention of the people who respond to these alerts.



Alert Rule Definition and Evaluation

Alert rules form the foundation of our monitoring system, defining the specific conditions that warrant human attention. Each rule represents a formal specification of what constitutes an abnormal system state and how the system should respond when that condition occurs.

Decision: Alert Rule Data Model

- Context:** We need a standardized way to define monitoring conditions that can be evaluated consistently across the system while providing sufficient flexibility for diverse monitoring scenarios.
- Options Considered:** Simple threshold-only rules, complex multi-condition rules with Boolean logic, time-series anomaly detection rules
- Decision:** Structured threshold-based rules with duration conditions and metadata annotations
- Rationale:** Threshold-based rules are intuitive for operators, predictable in behavior, and can handle 80% of common monitoring scenarios. Duration conditions prevent alert flapping from brief spikes.
- Consequences:** Simple to implement and understand, but may require multiple rules for complex scenarios. Future extensions can add advanced detection methods.

The `AlertRule` structure captures all essential information needed to evaluate and manage alerts:

Field	Type	Description
ID	string	Unique identifier for the alert rule, used for tracking and referencing
Name	string	Human-readable name displayed in notifications and dashboards
Expression	string	Metric query expression that returns the value to evaluate against threshold
Threshold	float64	Numeric value that triggers the alert when comparison condition is met
Operator	string	Comparison operator: "gt" (greater than), "lt" (less than), "eq" (equal), "ne" (not equal)
Duration	time.Duration	Minimum time the condition must persist before transitioning to firing state
EvaluationInterval	time.Duration	How frequently this rule should be evaluated against current metric data
Labels	Labels	Key-value pairs attached to alerts generated by this rule for routing and grouping
Annotations	map[string]string	Additional metadata for alert descriptions, runbook links, and context information

Alert rule expressions leverage our query engine to retrieve metric data for evaluation. The expression syntax follows the same patterns established in our query processor, enabling complex aggregations and transformations. For example, an expression might be `rate(http_requests_total[5m])` to monitor request rate over a 5-minute window, or `avg(cpu_usage) by (instance)` to track average CPU usage per server instance.

Decision: Duration-Based Alert Evaluation

- **Context:** Raw metric values can be noisy, leading to false alerts from brief spikes or temporary network issues.
- **Options Considered:** Immediate alerting on threshold breach, duration-based evaluation, statistical outlier detection
- **Decision:** Require conditions to persist for a specified duration before firing alerts
- **Rationale:** Duration thresholds dramatically reduce false positives while adding minimal delay for genuine issues. This matches operational intuition that brief spikes are usually not actionable.
- **Consequences:** Alerts may take longer to fire, but operator confidence in alert validity increases significantly.

The alert evaluation process follows a carefully orchestrated sequence:

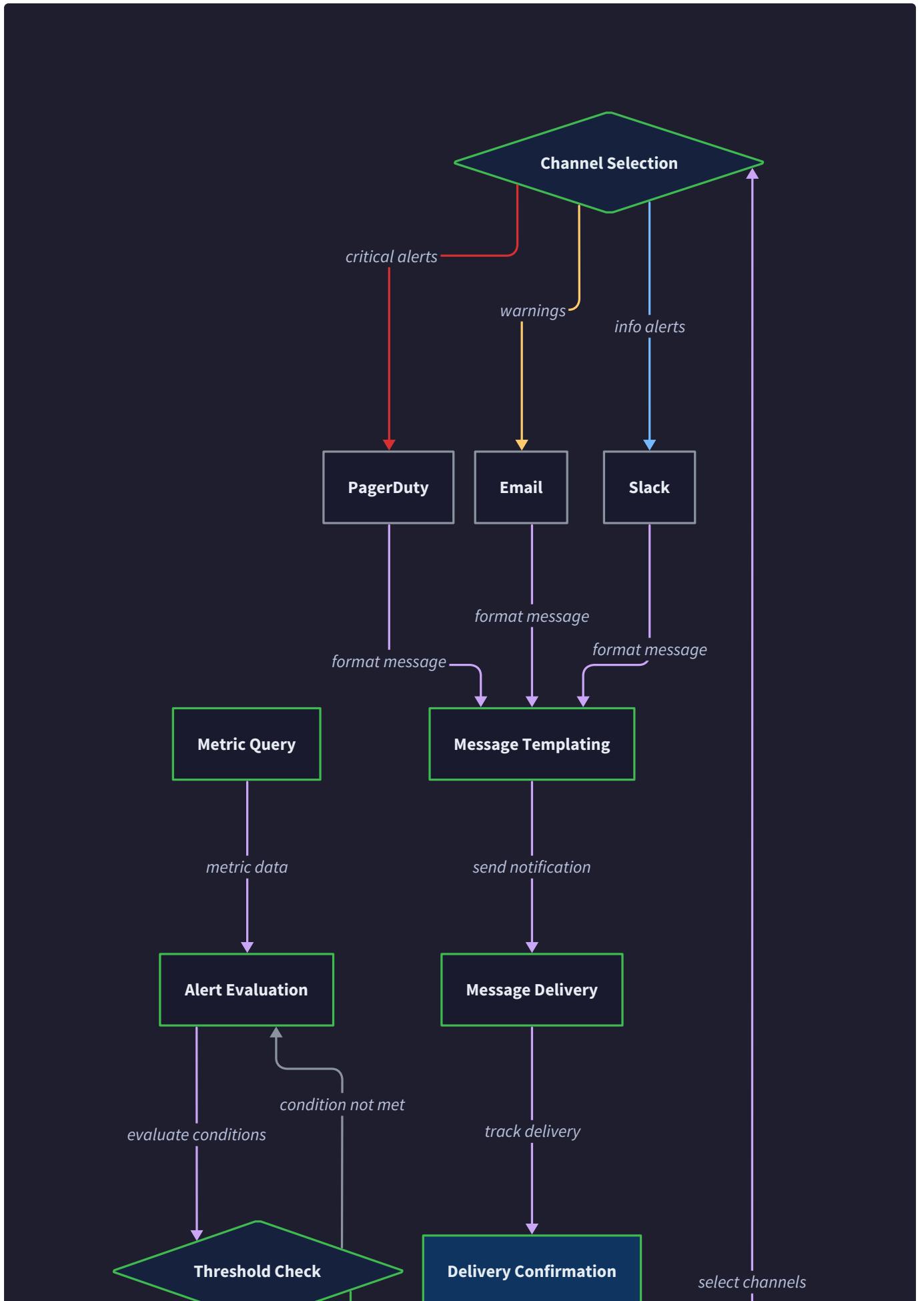
1. **Rule Scheduling:** The `AlertEvaluator` maintains an internal scheduler that tracks the next evaluation time for each active rule. Rules with shorter evaluation intervals receive priority scheduling to ensure timely detection of rapidly changing conditions.
2. **Query Execution:** When a rule's evaluation time arrives, the evaluator constructs a query request using the rule's expression and submits it to our `QueryProcessor`. The query includes the current timestamp and any necessary lookback window for rate calculations or aggregations.
3. **Threshold Comparison:** The evaluator extracts the numeric result from the query response and applies the specified comparison operator against the configured threshold. For queries returning multiple time series (e.g., per-instance metrics), each series is evaluated independently.
4. **Duration Tracking:** The system maintains state for each rule-series combination, recording when threshold conditions first became true. Only after the condition persists for the full duration period does the alert transition to firing state.
5. **State Transition:** Based on current conditions and historical state, the evaluator determines appropriate state transitions and triggers any necessary notifications or state persistence operations.

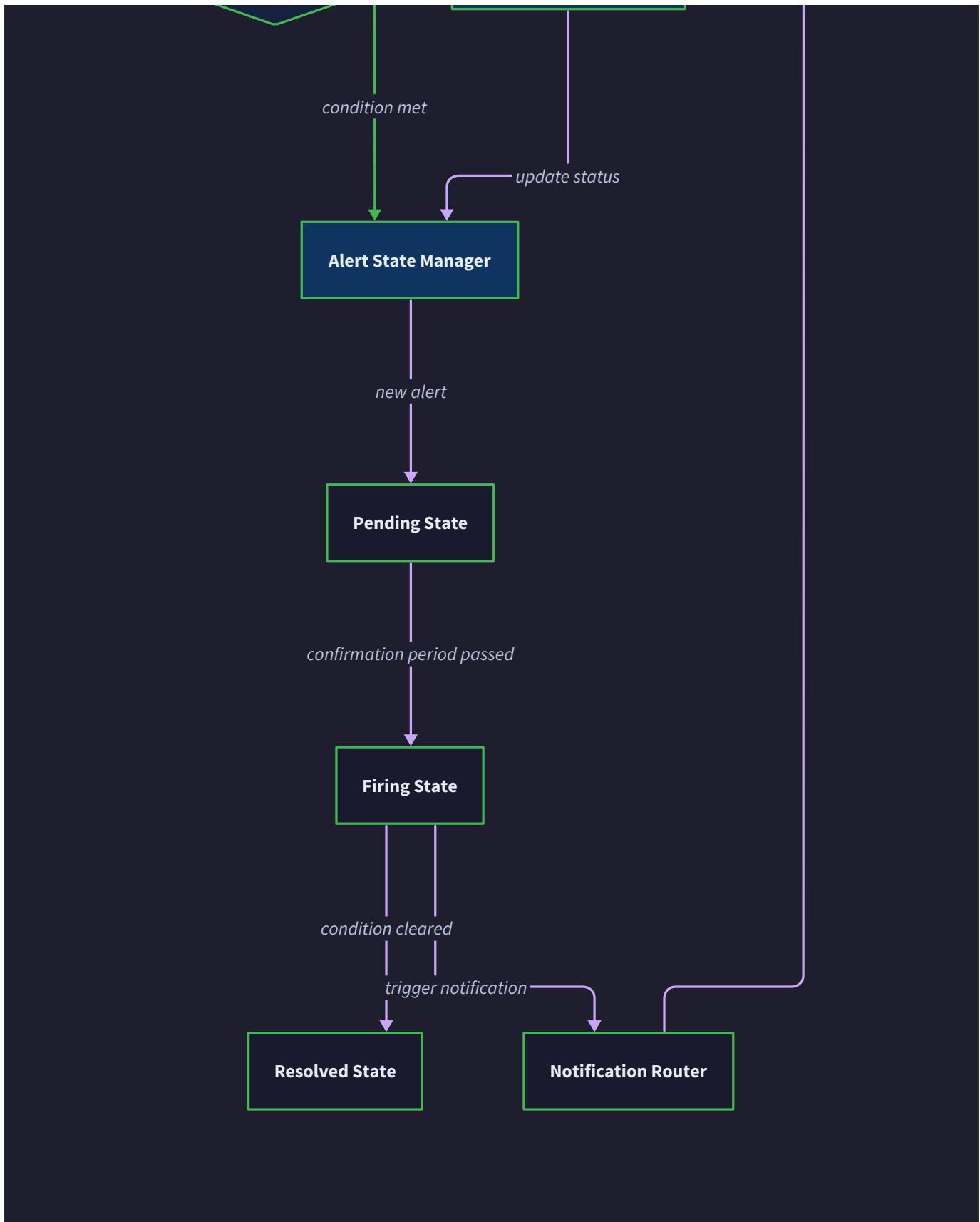
The `AlertEvaluator` interface defines the core evaluation contract:

Method	Parameters	Returns	Description
<code>EvaluateRules</code>	<code>ctx context.Context</code>	<code>error</code>	Processes all active alert rules, evaluating conditions and updating states
<code>UpdateAlertState</code>	<code>ruleID string, seriesLabels Labels, newState AlertState</code>	<code>error</code>	Persists alert state changes and triggers appropriate notifications
<code>GetActiveAlerts</code>	<code>ctx context.Context</code>	<code>[]ActiveAlert, error</code>	Retrieves currently firing or pending alerts for dashboard display
<code>SilenceAlert</code>	<code>ctx context.Context, ruleID string, duration time.Duration</code>	<code>error</code>	Temporarily suppresses notifications for specified alert during maintenance

⚠ Pitfall: Evaluation Interval vs Duration Confusion A common mistake is setting the evaluation interval longer than the alert duration. If a rule has a 30-second duration but evaluates every 60 seconds, the alert can never fire because the condition never persists long enough between evaluations. Always ensure $\text{evaluation interval} \leq \text{duration}/2$ for reliable detection.

⚠ Pitfall: Query Timeout Handling Alert evaluation queries must complete within the evaluation interval to maintain scheduling accuracy. Implement proper timeout handling and degraded operation modes when the query engine is overloaded. Consider using cached or approximate results rather than skipping evaluations entirely.





Alert State Management

Alert state management represents one of the most critical aspects of reliable alerting systems. Unlike simple binary on/off switches, production alerting requires sophisticated state tracking that accounts for the temporal nature of system problems and the human workflow of incident response.

Decision: Four-State Alert Model

- **Context:** Alerts need to represent the lifecycle of problems from initial detection through resolution, while avoiding false positives from brief anomalies.
- **Options Considered:** Binary firing/resolved model, three-state pending/firing/resolved model, four-state model with explicit inactive state
- **Decision:** Four distinct states: Inactive, Pending, Firing, and Resolved with explicit transition rules
- **Rationale:** Four states provide clear semantics for each phase of alert lifecycle while supporting duration-based evaluation and proper resolution tracking.
- **Consequences:** More complex state management logic, but significantly better operator experience and reduced alert fatigue.

The alert state machine governs all transitions between states with strict rules that ensure consistency and predictable behavior:

Current State	Condition	Next State	Actions Taken
Inactive	Threshold breached	Pending	Start duration timer, no notifications sent
Pending	Duration timer expires while condition persists	Firing	Send firing notification, update alert database
Pending	Condition returns to normal before duration expires	Inactive	Reset duration timer, no notifications sent
Firing	Condition returns to normal	Resolved	Send resolution notification, log resolution timestamp
Resolved	Condition breaches threshold again	Pending	Start new duration timer for potential re-firing
Firing	Manual silence activated	Silenced	Suppress notifications while maintaining state tracking
Silenced	Silence period expires	Firing or Resolved	Resume normal notification behavior based on current condition

This state model addresses several critical operational concerns. The **Pending** state prevents false alarms from brief spikes while still tracking that something unusual occurred. The **Resolved** state provides explicit confirmation that problems have cleared, which is essential for incident tracking and post-mortem analysis. The distinction between **Resolved** and **Inactive** ensures that resolution notifications are sent appropriately.

Each alert instance maintains detailed state information:

Field	Type	Description
AlertID	string	Unique identifier combining rule ID and series labels hash
RuleID	string	Reference to the alert rule that generated this alert instance
SeriesLabels	Labels	Specific label combination for this alert instance (e.g., instance=web-01)
State	AlertState	Current state in the alert lifecycle state machine
LastEvaluation	time.Time	Timestamp of the most recent rule evaluation for this alert
StateChanged	time.Time	When the alert last transitioned between states
FiredAt	*time.Time	When alert transitioned to firing state, nil if never fired
ResolvedAt	*time.Time	When alert transitioned to resolved state, nil if not resolved
Value	float64	Current metric value that was compared against threshold
ActiveSince	*time.Time	When condition first became true (pending state start)
Annotations	map[string]string	Resolved template variables and additional context

State transitions are not immediate but follow a careful evaluation process that considers timing constraints and current conditions. When evaluating a rule, the system must account for:

Timing Considerations: Each state transition has specific timing requirements. The pending-to-firing transition requires the condition to persist for the full duration period. Resolution detection should happen quickly to minimize notification delay, but must be confirmed across multiple evaluation cycles to avoid flapping.

Multi-Series Handling: Rules that return multiple time series (such as per-instance metrics) require independent state tracking for each series. An alert rule monitoring CPU usage across 10 servers creates 10 separate alert instances, each with its own state machine and notification lifecycle.

Persistence Requirements: Alert state must survive system restarts and component failures. The state management system implements write-ahead logging for all state transitions, ensuring that no alert state is lost even if the alerting system crashes during evaluation.

Decision: Alert State Persistence Strategy

- **Context:** Alert state must survive system restarts while maintaining high write throughput during normal operations.
- **Options Considered:** In-memory only with periodic snapshots, database with transactions, append-only log with periodic compaction
- **Decision:** Hybrid approach with in-memory state and write-ahead log for durability
- **Rationale:** In-memory operations provide fast evaluation performance, while WAL ensures durability without requiring complex database transactions.
- **Consequences:** Fast evaluation performance with strong durability guarantees, but requires careful recovery logic during startup.

The state management system implements several sophisticated features to handle edge cases:

Flapping Detection: When alerts rapidly transition between firing and resolved states, the system implements exponential backoff for notifications and marks the alert as flapping. This prevents notification storms while maintaining visibility into unstable conditions.

Staleness Handling: If metric data becomes unavailable (e.g., due to application crashes), the system implements configurable staleness policies. Alerts can be automatically resolved after a timeout period, or marked as "insufficient data" to distinguish from genuine resolution.

Group Notifications: Related alerts can be grouped together to reduce notification volume. For example, if 50 servers all lose network connectivity simultaneously, operators receive one grouped notification rather than 50 individual alerts.

⚠ Pitfall: State Transition Concurrency Alert state updates must be atomic to prevent race conditions when multiple evaluation goroutines update the same alert. Always use proper locking or atomic operations when modifying alert state, and ensure state transitions are logged before notifications are sent.

⚠ Pitfall: Resolution Detection Delay Setting evaluation intervals too long can delay resolution detection, leaving alerts in firing state long after problems clear. For critical alerts, consider using shorter evaluation intervals or implementing separate resolution checking with faster cycles.

Notification Channel Integration

The notification delivery system serves as the crucial bridge between alert detection and human response. A sophisticated notification system must handle multiple delivery channels, message formatting, delivery confirmation, and failure recovery while maintaining operator trust through reliable and well-formatted communications.

Decision: Multi-Channel Notification Architecture

- Context:** Different alerts require different notification urgency and routing, and organizations use diverse communication tools for incident response.
- Options Considered:** Single notification channel, plug-in architecture for channels, hardcoded integrations for common services
- Decision:** Unified notification interface with pluggable channel implementations and intelligent routing
- Rationale:** Pluggable architecture allows easy extension while unified interface ensures consistent behavior. Intelligent routing enables escalation policies and channel selection based on alert severity.
- Consequences:** More complex initial implementation, but highly flexible and maintainable for diverse organizational needs.

The `NotificationChannel` structure provides a standardized interface for all delivery methods:

Field	Type	Description
<code>Type</code>	<code>string</code>	Channel type identifier: "email", "slack", "pagerduty", "webhook"
<code>Name</code>	<code>string</code>	Human-readable channel name for configuration and debugging
<code>Config</code>	<code>map[string]string</code>	Channel-specific configuration parameters (API keys, URLs, etc.)
<code>Enabled</code>	<code>bool</code>	Whether this channel is currently active for notification delivery
<code>RateLimitBurst</code>	<code>int</code>	Maximum notifications that can be sent in rapid succession
<code>RateLimitPeriod</code>	<code>time.Duration</code>	Time window for rate limiting calculations
<code>RetryConfig</code>	<code>RetryConfig</code>	Backoff and retry policies for failed delivery attempts
<code>MessageTemplate</code>	<code>string</code>	Go template string for formatting notification messages
<code>Filters</code>	<code>[]NotificationFilter</code>	Rules determining which alerts should use this channel

Each notification channel type implements a common interface that abstracts delivery details:

Method	Parameters	Returns	Description
SendNotification	ctx context.Context, alert Alert, message NotificationMessage	error	Delivers notification through this channel with delivery confirmation
ValidateConfig	config map[string]string	error	Checks channel configuration for required parameters and connectivity
HealthCheck	ctx context.Context	error	Verifies channel availability and authentication status
FormatMessage	alert Alert, template string	NotificationMessage, error	Applies message templating to generate channel-specific content

Email Notification Implementation: Email channels support both plain text and HTML message formats with embedded metric charts and alert context. The implementation handles SMTP authentication, TLS encryption, and delivery status tracking. Email notifications include alert details, direct links to relevant dashboards, and suggested runbook procedures.

Slack Integration: Slack notifications leverage webhook integrations or bot API tokens to deliver rich formatted messages with interactive elements. Messages include color-coded severity indicators, metric snapshots, and action buttons for alert acknowledgment or silencing. The system supports both channel posting and direct message delivery based on alert routing rules.

PagerDuty Integration: For critical alerts requiring immediate response, PagerDuty integration creates incidents with proper escalation policies. The system maps alert severity levels to PagerDuty incident priorities and includes detailed context in incident descriptions. Auto-resolution is supported when alerts return to normal state.

Webhook Channels: Generic webhook support enables integration with custom tooling and services not directly supported. Webhook notifications include configurable payload formatting, authentication headers, and retry logic with exponential backoff.

Message templating provides flexible content customization while ensuring consistent formatting across channels:

```
Alert: {{ .Alert.RuleName }}  
Status: {{ .Alert.State }}  
Instance: {{ .Alert.Labels.instance }}  
Value: {{ .Alert.Value }} (threshold: {{ .Alert.Threshold }})  
Started: {{ .Alert.ActiveSince.Format "2006-01-02 15:04:05" }}  
Runbook: {{ .Alert.Annotations.runbook_url }}  
Dashboard: https://dashboard.example.com/d/{{ .Alert.Annotations.dashboard_id }}
```

The notification routing system determines which channels receive each alert based on configurable rules:

Filter Type	Parameters	Description
LabelFilter	label string, values []string	Route alerts with specific label values to designated channels
SeverityFilter	min_severity string, max_severity string	Channel receives alerts within specified severity range
TimeFilter	start_time string, end_time string, timezone string	Time-based routing for business hours vs. on-call coverage
TeamFilter	teams []string	Route alerts to channels based on team ownership labels
EscalationFilter	delay time.Duration	Secondary channels activated after specified delay if alert persists

Delivery Reliability: The notification system implements sophisticated retry logic to handle transient failures. Each channel maintains a delivery queue with exponential backoff for failed attempts. Critical notifications use multiple delivery attempts across different time intervals, and persistent failures trigger fallback notification channels.

Rate Limiting: To prevent notification storms during widespread outages, each channel implements configurable rate limiting. The system uses token bucket algorithms to allow burst notifications while preventing sustained high-volume delivery that could overwhelm recipients or trigger service quotas.

Delivery Confirmation: Where supported by the underlying service, the system tracks delivery confirmation and maintains metrics on notification success rates. Failed deliveries are logged for troubleshooting, and persistent delivery failures can trigger alerts about the alerting system itself.

⚠ Pitfall: Template Rendering Errors Notification templates can fail to render if alert data is missing expected fields or contains invalid characters. Always validate templates during configuration and implement fallback rendering with basic alert information when template processing fails.

⚠ Pitfall: Notification Loop Prevention Be careful not to create notification loops where alerts about the alerting system itself trigger recursive notifications. Implement special handling for infrastructure alerts and avoid routing alerting system failures through the same notification channels.

⚠ Pitfall: Credential Management Notification channels require API keys and credentials that must be securely stored and rotated. Never log sensitive configuration values, implement proper secret management, and provide clear error messages when authentication fails without exposing credentials.

Implementation Guidance

This section provides concrete implementation details for building the alerting system in Go, bridging the gap between the design concepts and working code.

Technology Recommendations

Component	Simple Option	Advanced Option
Alert Storage	In-memory with JSON file persistence	BadgerDB embedded database
Notification Queue	Go channels with worker pools	Redis with persistent queues
HTTP Client	net/http with custom retry logic	go-retryablehttp library
Message Templating	text/template standard library	Advanced templating with sprig functions
State Persistence	JSON files with atomic writes	Write-ahead log with periodic snapshots
Scheduling	time.Ticker with goroutines	Cron-like scheduler with distributed coordination

File Structure

```
project-root/
  cmd/server/main.go
  internal/alerting/
    evaluator.go
    evaluator_test.go
    state_manager.go
    notification_manager.go
    channels/
      email.go
      slack.go
      webhook.go
      pagerduty.go
    rules/
      parser.go
      validator.go
    templates/
      default tmpl
      slack tmpl
  internal/storage/
    interfaces.go
  pkg/config/
    alerting.go
```

```
          ← server entry point
          ← alerting system implementation
          ← core alert evaluation logic
          ← evaluation engine tests
          ← alert state tracking
          ← notification delivery coordination
          ← notification channel implementations
          ← email notification channel
          ← Slack integration
          ← generic webhook channel
          ← PagerDuty integration
          ← alert rule management
          ← rule configuration parsing
          ← rule validation logic
          ← notification templates
          ← default message template
          ← Slack-specific template
          ← shared storage interfaces
          ← storage contracts
          ← configuration management
          ← alerting configuration structures
```

Alert State Management Infrastructure

```
package alerting

import (
    "context"
    "encoding/json"
    "fmt"
    "sync"
    "time"
)

// AlertState represents the current state of an alert in the lifecycle

type AlertState int

const (
    AlertStateInactive AlertState = iota
    AlertStatePending
    AlertStateFiring
    AlertStateResolved
    AlertStateSilenced
)

// AlertInstance tracks the state and history of a specific alert

type AlertInstance struct {

    AlertID      string      `json:"alert_id"`
    RuleID       string      `json:"rule_id"`
    SeriesLabels Labels      `json:"series_labels"`
    State        AlertState   `json:"state"`
    LastEvaluation time.Time `json:"last_evaluation"`
    StateChanged  time.Time   `json:"state_changed"`
    FiredAt      *time.Time  `json:"fired_at,omitempty"`
    ResolvedAt   *time.Time  `json:"resolved_at,omitempty"`
    Value         float64     `json:"value"`
    ActiveSince   *time.Time  `json:"active_since,omitempty"`
}
```

GO

```

Annotations      map[string]string     `json:"annotations"`

NotificationLog []NotificationRecord  `json:"notification_log"`

}

// NotificationRecord tracks when and how notifications were delivered

type NotificationRecord struct {

    Timestamp   time.Time `json:"timestamp"`

    Channel     string    `json:"channel"`

    State       AlertState `json:"state"`

    Success     bool      `json:"success"`

    ErrorMsg    string    `json:"error_msg,omitempty"`

    RetryCount  int       `json:"retry_count"`

}

// StateManager handles alert state transitions and persistence

type StateManager struct {

    alerts      map[string]*AlertInstance

    mu         sync.RWMutex

    persistence StatePersistence

    logger     *slog.Logger

}

// StatePersistence interface for alert state durability

type StatePersistence interface {

    SaveState(alertID string, state *AlertInstance) error

    LoadState(alertID string) (*AlertInstance, error)

    LoadAllStates() (map[string]*AlertInstance, error)

    DeleteState(alertID string) error

}

// NewStateManager creates a new alert state manager with persistence

func NewStateManager(persistence StatePersistence, logger *slog.Logger) *StateManager {

    return &StateManager{

        alerts:      make(map[string]*AlertInstance),

```

```

    persistence: persistence,
    logger:      logger,
}

}

// UpdateAlertState processes alert state transitions with proper validation

func (sm *StateManager) UpdateAlertState(ctx context.Context, ruleID string, seriesLabels Labels,
currentValue float64, threshold float64, conditionMet bool, rule *AlertRule) error {

    // TODO 1: Generate unique alert ID from rule ID and series labels

    // TODO 2: Acquire write lock and get current alert state

    // TODO 3: Determine new state based on current state and condition

    // TODO 4: Validate state transition is legal according to state machine

    // TODO 5: Update alert instance with new state and metadata

    // TODO 6: Persist state change to storage

    // TODO 7: Release lock and trigger notifications if state changed

    // Hint: Use SeriesID() method to generate consistent alert identifiers

    return nil
}

// GetActiveAlerts returns all alerts in firing or pending states

func (sm *StateManager) GetActiveAlerts(ctx context.Context) ([]*AlertInstance, error) {

    // TODO 1: Acquire read lock for safe concurrent access

    // TODO 2: Iterate through all alerts and filter by active states

    // TODO 3: Create slice of active alerts for return

    // TODO 4: Release lock and return filtered results

    return nil, nil
}

// CleanupResolvedAlerts removes old resolved alerts from memory and storage

func (sm *StateManager) CleanupResolvedAlerts(ctx context.Context, maxAge time.Duration) error {

    // TODO 1: Calculate cutoff time based on maxAge parameter

    // TODO 2: Acquire write lock for alert map modifications

    // TODO 3: Identify resolved alerts older than cutoff time

    // TODO 4: Remove alerts from memory and delete from persistence
}

```

```
// TODO 5: Log cleanup statistics and release lock  
  
return nil  
  
}
```

Alert Evaluation Engine Core

```
package alerting

import (
    "context"
    "fmt"
    "time"
)

// AlertEvaluator handles the core alert evaluation logic and scheduling

type AlertEvaluator struct {

    queryProcessor    QueryProcessor
    stateManager     *StateManager
    ruleManager      *RuleManager
    notificationMgr *NotificationManager
    logger           *slog.Logger
    stopCh          chan struct{}
    evaluationStats *EvaluationStats
}

// EvaluationStats tracks alert evaluation performance metrics

type EvaluationStats struct {

    TotalEvaluations    int64
    SuccessfulEvaluations int64
    FailedEvaluations   int64
    AverageEvaluationTime time.Duration
    LastEvaluation       time.Time
}

// NewAlertEvaluator creates a new alert evaluation engine

func NewAlertEvaluator(queryProcessor QueryProcessor, stateManager *StateManager, ruleManager *RuleManager,
notificationMgr *NotificationManager, logger *slog.Logger) *AlertEvaluator {

    return &AlertEvaluator{
        queryProcessor: queryProcessor,
```

```
stateManager: stateManager,
ruleManager: ruleManager,
notificationMgr: notificationMgr,
logger: logger,
stopCh: make(chan struct{}),
evaluationStats: &EvaluationStats{},

}

}

// Start begins the alert evaluation loop with proper scheduling

func (ae *AlertEvaluator) Start(ctx context.Context) error {
    // TODO 1: Initialize evaluation scheduler with configurable intervals
    // TODO 2: Start background goroutine for continuous evaluation loop
    // TODO 3: Implement graceful shutdown handling with context cancellation
    // TODO 4: Set up periodic statistics reporting and health checks
    // TODO 5: Return any initialization errors
    // Hint: Use time.NewTicker for regular evaluation scheduling
    return nil
}

// EvaluateRules processes all active alert rules against current metric data

func (ae *AlertEvaluator) EvaluateRules(ctx context.Context) error {
    // TODO 1: Get list of active alert rules from rule manager
    // TODO 2: Iterate through each rule and check if evaluation is due
    // TODO 3: Execute rule query expression against query processor
    // TODO 4: Extract numeric results and compare against thresholds
    // TODO 5: Update alert state through state manager
    // TODO 6: Trigger notifications for state changes
    // TODO 7: Update evaluation statistics and handle any errors
    // Hint: Handle multi-series query results by evaluating each series independently
    return nil
}
```

```
// evaluateRule processes a single alert rule against current metric data

func (ae *AlertEvaluator) evaluateRule(ctx context.Context, rule *AlertRule) error {

    // TODO 1: Build query request with rule expression and current time

    // TODO 2: Execute query and handle timeout/error conditions

    // TODO 3: Parse query results into individual time series

    // TODO 4: For each series, extract current value and labels

    // TODO 5: Apply threshold comparison using rule operator

    // TODO 6: Update alert state with new condition and value

    // TODO 7: Log evaluation results and any errors encountered

    return nil
}

// Stop gracefully shuts down the alert evaluator

func (ae *AlertEvaluator) Stop(ctx context.Context) error {

    // TODO 1: Signal stop to evaluation loop via channel

    // TODO 2: Wait for current evaluations to complete

    // TODO 3: Clean up any pending notifications

    // TODO 4: Log final evaluation statistics

    close(ae.stopCh)

    return nil
}
```

Notification Delivery System

```
package alerting

import (
    "bytes"
    "context"
    "text/template"
    "time"
)

// NotificationManager coordinates notification delivery across multiple channels

type NotificationManager struct {

    channels     map[string]NotificationChannel
    templates    map[string]*template.Template
    deliveryQueue chan *NotificationRequest
    workers      int
    logger       *slog.Logger
}

// NotificationRequest represents a pending notification delivery

type NotificationRequest struct {

    Alert        *AlertInstance
    Channels    []string
    State        AlertState
    Timestamp   time.Time
    RetryCount  int
    MaxRetries  int
}

// NotificationChannel interface for all notification delivery methods

type NotificationChannel interface {

    SendNotification(ctx context.Context, alert *AlertInstance, message string) error
    ValidateConfig(config map[string]string) error
    HealthCheck(ctx context.Context) error
}
```

GO

```

FormatMessage(alert *AlertInstance, template string) (string, error)

Name() string

Type() string

}

// NewNotificationManager creates a notification manager with configured channels

func NewNotificationManager(channels []NotificationChannel, workers int, logger *slog.Logger)
*NotificationManager {

mgr := &NotificationManager{

    channels:     make(map[string]NotificationChannel),
    templates:    make(map[string]*template.Template),
    deliveryQueue: make(chan *NotificationRequest, 1000),
    workers:      workers,
    logger:       logger,
}

// Register notification channels

for _, channel := range channels {

    mgr.channels[channel.Name()] = channel
}

return mgr
}

// SendNotification queues alert notification for delivery through specified channels

func (nm *NotificationManager) SendNotification(ctx context.Context, alert *AlertInstance, channels []string)
error {

// TODO 1: Validate that requested channels are configured and healthy

// TODO 2: Create notification request with alert details and channel list

// TODO 3: Apply any routing rules or filters for channel selection

// TODO 4: Queue notification request for background delivery

// TODO 5: Return immediate acknowledgment or queueing errors

// Hint: Non-blocking queue insertion to avoid deadlocks during high load

return nil
}

```

```
}

// Start begins notification delivery workers

func (nm *NotificationManager) Start(ctx context.Context) error {

    // TODO 1: Start configured number of worker goroutines

    // TODO 2: Each worker processes delivery queue continuously

    // TODO 3: Implement graceful shutdown with context cancellation

    // TODO 4: Set up delivery retry logic with exponential backoff

    // TODO 5: Track delivery statistics and health metrics

    return nil
}

// deliverNotification handles actual delivery to a specific channel

func (nm *NotificationManager) deliverNotification(ctx context.Context, req *NotificationRequest, channelName string) error {

    // TODO 1: Get notification channel by name from registered channels

    // TODO 2: Select appropriate message template based on channel type

    // TODO 3: Render notification message using template and alert data

    // TODO 4: Attempt delivery through channel with timeout handling

    // TODO 5: Record delivery attempt result and implement retry logic

    // TODO 6: Update notification log in alert instance

    return nil
}

// LoadTemplates reads notification templates from configuration

func (nm *NotificationManager) LoadTemplates(templateDir string) error {

    // TODO 1: Scan template directory for .tmpl files

    // TODO 2: Parse each template file using text/template

    // TODO 3: Validate template syntax and required variables

    // TODO 4: Store parsed templates in manager template map

    // TODO 5: Set up template reloading on configuration changes

    return nil
}
```

```
// GetDeliveryStats returns notification delivery statistics

func (nm *NotificationManager) GetDeliveryStats() *DeliveryStats {
    // TODO 1: Aggregate delivery statistics from all channels
    // TODO 2: Calculate success rates and average delivery times
    // TODO 3: Return structured statistics for monitoring
    return nil
}
```

Email Notification Channel Implementation

```
package channels

import (
    "context"
    "crypto/tls"
    "fmt"
    "net/smtp"
    "strings"
    "time"
)

// EmailChannel implements notification delivery via SMTP email

type EmailChannel struct {

    name      string
    config    EmailConfig
    auth      smtp.Auth
    client   *smtp.Client
    logger   *slog.Logger
}

// EmailConfig holds SMTP configuration parameters

type EmailConfig struct {

    SMTPHost      string `yaml:"smtp_host"`
    SMTPPort      int    `yaml:"smtp_port"`
    Username      string `yaml:"username"`
    Password      string `yaml:"password"`
    FromAddress   string `yaml:"from_address"`
    ToAddresses  []string `yaml:"to_addresses"`
    UseTLS       bool   `yaml:"use_tls"`
    Subject       string `yaml:"subject"`
}

// NewEmailChannel creates a new email notification channel
```

GO

```

func NewEmailChannel(name string, config EmailConfig, logger *slog.Logger) (*EmailChannel, error) {

    // TODO 1: Validate SMTP configuration parameters

    // TODO 2: Set up SMTP authentication with provided credentials

    // TODO 3: Test SMTP connection to validate configuration

    // TODO 4: Initialize email channel with working SMTP client

    // TODO 5: Return configured channel or connection errors

    return nil, nil
}

// SendNotification delivers alert notification via email

func (ec *EmailChannel) SendNotification(ctx context.Context, alert *AlertInstance, message string) error {

    // TODO 1: Format email subject using alert data and configuration

    // TODO 2: Build complete email message with headers and body

    // TODO 3: Connect to SMTP server with TLS if configured

    // TODO 4: Authenticate using stored credentials

    // TODO 5: Send email to all configured recipients

    // TODO 6: Handle SMTP errors and implement retry logic

    // Hint: Use net/smtp package with proper TLS configuration

    return nil
}

// ValidateConfig checks email channel configuration for required fields

func (ec *EmailChannel) ValidateConfig(config map[string]string) error {

    // TODO 1: Check that SMTP host and port are provided

    // TODO 2: Validate authentication credentials

    // TODO 3: Verify from address and recipient list

    // TODO 4: Test SMTP connectivity if possible

    // TODO 5: Return detailed validation errors

    return nil
}

// FormatMessage renders alert data into email-formatted message

func (ec *EmailChannel) FormatMessage(alert *AlertInstance, tmplStr string) (string, error) {

```

```
// TODO 1: Parse message template string

// TODO 2: Create template data context with alert information

// TODO 3: Execute template rendering with alert data

// TODO 4: Handle template errors gracefully

// TODO 5: Return formatted message or rendering errors

return "", nil

}

// HealthCheck verifies SMTP connectivity and authentication

func (ec *EmailChannel) HealthCheck(ctx context.Context) error {

// TODO 1: Establish SMTP connection with timeout

// TODO 2: Verify authentication credentials

// TODO 3: Test basic SMTP commands without sending email

// TODO 4: Return health status or connectivity errors

return nil

}

func (ec *EmailChannel) Name() string { return ec.name }

func (ec *EmailChannel) Type() string { return "email" }
```

Slack Notification Channel Implementation

```
package channels

import (
    "bytes"
    "context"
    "encoding/json"
    "fmt"
    "net/http"
    "time"
)

// SlackChannel implements notification delivery via Slack webhooks

type SlackChannel struct {

    name      string
    webhookURL string
    channel   string
    username  string
    iconEmoji string
    httpClient *http.Client
    logger    *slog.Logger
}

// SlackMessage represents a Slack webhook payload

type SlackMessage struct {

    Channel  string          `json:"channel,omitempty"`
    Username string          `json:"username,omitempty"`
    IconEmoji string          `json:"icon_emoji,omitempty"`
    Text     string          `json:"text"`
    Attachments []SlackAttachment `json:"attachments,omitempty"`
}

// SlackAttachment provides rich formatting for Slack messages

type SlackAttachment struct {
```

GO

```

Color      string          `json:"color"`

Title     string          `json:"title"`

Text      string          `json:"text"`

Fields    []SlackField   `json:"fields"`

Timestamp int64           `json:"ts"`

Footer    string          `json:"footer"`

}

// SlackField represents a field in a Slack attachment

type SlackField struct {

    Title string `json:"title"`

    Value string `json:"value"`

    Short bool   `json:"short"`

}

// NewSlackChannel creates a new Slack notification channel

func NewSlackChannel(name, webhookURL, channel, username, iconEmoji string, logger *slog.Logger) *SlackChannel {

    return &SlackChannel{

        name:      name,

        webhookURL: webhookURL,

        channel:    channel,

        username:   username,

        iconEmoji:  iconEmoji,

        httpClient: &http.Client{Timeout: 10 * time.Second},

        logger:     logger,

    }

}

// SendNotification delivers alert notification to Slack channel

func (sc *SlackChannel) SendNotification(ctx context.Context, alert *AlertInstance, message string) error {

    // TODO 1: Build Slack message with proper formatting and attachments

    // TODO 2: Set color coding based on alert state (red=firing, green=resolved)

    // TODO 3: Include alert metadata as attachment fields
}

```

```
// TODO 4: Serialize message to JSON payload

// TODO 5: Send HTTP POST request to Slack webhook URL

// TODO 6: Handle HTTP errors and implement retry logic

// Hint: Use different colors for different alert states to provide visual cues

return nil

}

// FormatMessage creates Slack-formatted message with rich attachments

func (sc *SlackChannel) FormatMessage(alert *AlertInstance, tmplStr string) (string, error) {

    // TODO 1: Create Slack attachment with alert details

    // TODO 2: Add fields for alert name, state, value, and duration

    // TODO 3: Include links to dashboards and runbooks if available

    // TODO 4: Format timestamp and alert metadata

    // TODO 5: Return formatted Slack message structure

    return "", nil
}

// ValidateConfig verifies Slack channel configuration

func (sc *SlackChannel) ValidateConfig(config map[string]string) error {

    // TODO 1: Check that webhook URL is provided and valid

    // TODO 2: Validate channel name format if specified

    // TODO 3: Test webhook connectivity with a simple message

    // TODO 4: Return validation errors with helpful messages

    return nil
}

// HealthCheck tests Slack webhook connectivity

func (sc *SlackChannel) HealthCheck(ctx context.Context) error {

    // TODO 1: Send test message to Slack webhook

    // TODO 2: Verify HTTP response indicates successful delivery

    // TODO 3: Handle rate limiting and authentication errors

    // TODO 4: Return health status

    return nil
}
```

```
}
```



```
func (sc *SlackChannel) Name() string { return sc.name }

func (sc *SlackChannel) Type() string { return "slack" }
```

Milestone Checkpoints

Checkpoint 1: Alert Rule Evaluation After implementing the basic alert evaluation logic:

```
go test ./internal/alerting/...
```

BASH

Expected behavior:

- Rules evaluate against mock metric data
- State transitions work correctly (inactive → pending → firing → resolved)
- Duration-based firing prevents false alarms
- Multiple series from single rule create separate alert instances

Checkpoint 2: Notification Delivery After implementing notification channels:

```
go run cmd/server/main.go --config=test-config.yaml
```

BASH

Test manual notification delivery:

- Send test alert via API: `curl -X POST localhost:8080/api/v1/alerts/test`
- Verify email delivery to configured SMTP server
- Check Slack webhook receives properly formatted messages
- Confirm notification logs are recorded in alert instances

Checkpoint 3: End-to-End Alerting After complete implementation:

- Configure alert rule monitoring CPU usage > 80%
- Generate load to trigger threshold breach
- Verify alert transitions: inactive → pending → firing
- Confirm notifications sent during state transitions
- Test alert resolution when condition clears
- Validate silence functionality during maintenance windows

Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
Alerts never fire	Duration longer than evaluation interval	Check rule timing configuration	Set evaluation interval \leq duration/2
Alert flapping	Threshold too close to normal values	Analyze metric variance over time	Adjust threshold or use averaging
Missing notifications	Notification channel configuration error	Check channel health endpoints	Validate SMTP/webhook credentials
Duplicate notifications	State transition logic error	Review alert state logs	Fix state persistence or locking
Slow evaluation	Expensive query expressions	Profile query execution times	Optimize queries or reduce evaluation frequency
Memory growth	Alert instances not cleaned up	Monitor resolved alert counts	Implement periodic cleanup of old alerts
Notification delays	Delivery queue backlog	Check notification worker metrics	Increase worker pool size or optimize delivery

Component Interactions and Data Flow

Milestone(s): All milestones (component interactions span the entire system architecture from metrics ingestion through storage, querying, visualization, and alerting)

The City Infrastructure Metaphor: Understanding System Communication

Think of our metrics system as a modern city's infrastructure. The **metrics ingestion engine** acts like the city's logistics network, receiving shipments from various sources and routing them to appropriate destinations. The **storage engine** functions as the city's warehouses and archives, organizing and preserving goods for future retrieval. The **query engine** operates like the city's information services, helping residents find and access what they need. The **dashboard** serves as the city's command center, providing real-time visibility into city operations. Finally, the **alerting system** acts as the emergency response network, monitoring conditions and coordinating responses when problems arise.

Just as a city's infrastructure components must communicate seamlessly through well-defined protocols and channels, our metrics system components interact through carefully designed APIs, message formats, and data flows. Each component has specific responsibilities but must coordinate with others to achieve the overall system goals. Understanding these interactions is crucial because the system's reliability and performance depend not just on individual component quality, but on how effectively they work together.

The key insight is that **data flows through the system in predictable patterns**, and each component transformation adds value while maintaining data integrity. When a metric enters the system, it triggers a cascade of interactions: validation, storage, indexing, and potentially alerting. When a user queries data, it initiates a different interaction pattern: query parsing, execution planning, storage retrieval, and result formatting. Understanding these flows helps developers debug issues, optimize performance, and extend functionality.

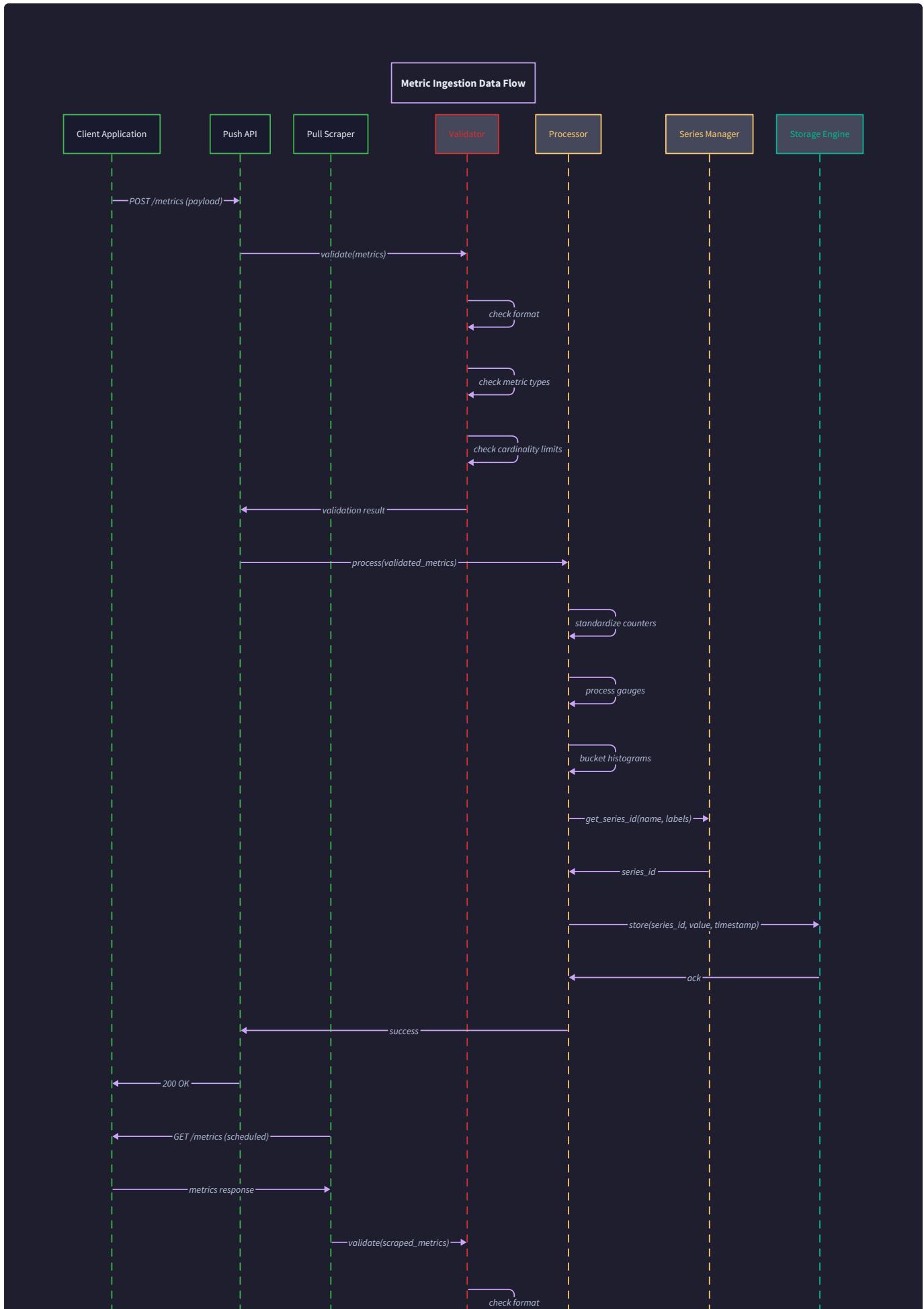
Decision: Event-Driven vs Request-Response Communication

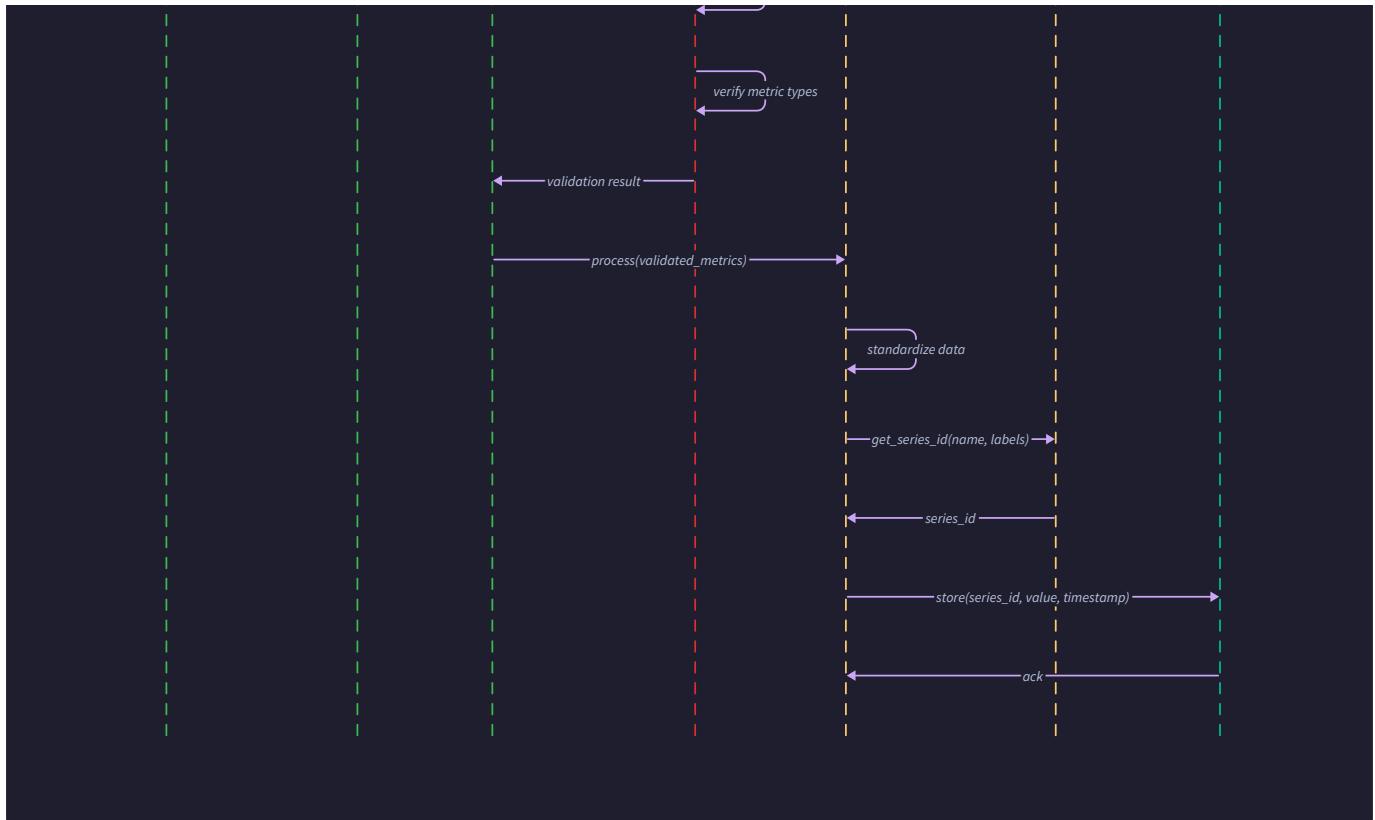
- **Context:** Components need to communicate both for real-time data flow and on-demand operations
- **Options Considered:** Pure event-driven messaging, pure request-response APIs, hybrid approach
- **Decision:** Hybrid approach with request-response for queries and event-driven for data flow
- **Rationale:** Request-response provides immediate feedback for user interactions and error handling, while event-driven enables efficient real-time data processing and decoupling
- **Consequences:** Enables both interactive user experience and high-throughput data processing, but requires managing two communication patterns

Communication Pattern	Use Cases	Implementation	Benefits	Trade-offs
Request-Response	Dashboard queries, API calls, health checks	HTTP REST APIs with JSON	Immediate feedback, error handling, debugging	Higher latency, tighter coupling
Event-Driven	Metric ingestion, alert notifications, real-time updates	WebSocket connections, internal channels	Low latency, loose coupling, scalability	Complex error handling, eventual consistency
Hybrid	Complete system operations	Both patterns appropriately applied	Best of both worlds	Increased complexity, multiple failure modes

Metric Ingestion Flow

The metric ingestion flow represents the system's primary data entry pipeline, transforming raw metric data from external sources into properly validated, normalized, and stored time-series data. This flow must handle high throughput, validate data integrity, manage cardinality constraints, and ensure durability guarantees.





Push-Based Ingestion Process

The push-based ingestion process begins when external applications send metric data to the system's ingestion endpoints. This follows a multi-stage pipeline designed to validate, process, and store metrics while maintaining high throughput and reliability.

Stage 1: HTTP Request Reception and Initial Processing

The ingestion process starts when the `MetricsIngestor` receives an HTTP POST request containing metric data. The HTTP server performs initial request validation, checking content-type headers, request size limits, and authentication tokens if configured. The request body, typically containing a batch of metrics in JSON format, gets parsed into the internal `Metric` data structures.

During this stage, the system performs basic request hygiene checks: ensuring the payload size doesn't exceed configured limits (typically 1MB to prevent memory exhaustion), validating the content-type is application/json, and verifying any required authentication headers. If these basic checks fail, the system immediately returns an HTTP 400 or 401 response without further processing.

Stage 2: Metric Validation and Normalization

Each metric in the batch undergoes comprehensive validation through the `Validator` component. This validation ensures data quality and prevents common issues that could corrupt the storage engine or cause query failures.

The validation process examines multiple aspects of each metric:

- Metric Name Validation:** Confirms the metric name follows the required naming conventions (alphanumeric characters, underscores, and dots only), doesn't exceed maximum length limits (typically 256 characters), and doesn't use reserved prefixes that could conflict with system metrics.
- Metric Type Consistency:** Verifies that the `MetricType` field contains a valid enum value (Counter, Gauge, or Histogram) and that the associated sample data is appropriate for the metric type. For example, counter values must be non-negative and monotonically increasing.
- Label Validation:** Ensures all label keys and values in the `Labels` map follow naming conventions, don't exceed length limits, and don't contain prohibited characters that could interfere with query parsing. The system also validates that reserved label

names (like `_name_` and `_timestamp_`) are not used.

4. **Sample Data Validation:** Checks that timestamp values are reasonable (not too far in the past or future), numeric values are valid (not NaN or infinite), and the sample count is within acceptable limits.

5. **Cardinality Validation:** The `CardinalityManager` evaluates whether accepting this metric would create excessive cardinality by generating too many unique time-series combinations. This prevents "cardinality explosions" that could overwhelm the storage system.

After validation, the `NormalizeMetric` function standardizes the metric format: converting timestamps to UTC, sorting labels alphabetically for consistent series ID generation, and applying any configured label transformations or defaults.

Stage 3: Series ID Generation and Indexing

For each validated metric, the system generates a unique series ID by computing a hash of the metric name and sorted labels. This series ID serves as the primary key for time-series data storage and enables efficient retrieval during queries.

The series ID generation process combines the metric name and all label key-value pairs into a canonical string representation, then computes a cryptographic hash (typically SHA-256) to create a unique identifier. This approach ensures that metrics with identical names and labels always generate the same series ID, enabling proper time-series continuity.

The `SeriesIndex` maintains mapping between series IDs and their metadata, including the metric name, full label set, and references to storage blocks containing the series data. When a new series ID is encountered, the index creates a new `SeriesInfo` entry; for existing series, it updates the last sample timestamp to track data freshness.

Stage 4: Write-Ahead Log Persistence

Before acknowledging the ingestion request, the system writes all validated samples to the `WriteAheadLog` for durability. This ensures that even if the process crashes before samples reach permanent storage, the data can be recovered and reprocessed during system restart.

The WAL write process creates a `WALRecord` containing the timestamp, metric samples, and a checksum for integrity verification. The record gets appended to the current WAL file, and the system calls `fsync()` to ensure the data reaches persistent storage before proceeding. This durability guarantee is essential for preventing data loss in production environments.

Stage 5: Storage Engine Integration

After successful WAL persistence, the samples get forwarded to the `StorageEngine` through the `WriteSamples` method. The storage engine handles the complex process of organizing samples into time-based blocks, managing in-memory buffers, and scheduling background compaction operations.

The storage engine may buffer samples in memory before writing them to disk blocks, depending on the configured flush intervals and memory pressure. However, since samples are already persisted in the WAL, this buffering doesn't compromise durability - it only affects query latency for the most recent data points.

Stage 6: Response Generation and Statistics Updates

Once all samples in the batch have been successfully processed and stored, the ingestion endpoint returns an HTTP 200 response to the client. The response includes statistics about the processing operation: number of metrics accepted, number rejected due to validation failures, and any warnings about potential issues.

The system updates internal metrics about ingestion performance, including throughput rates, error counts, and processing latencies. These operational metrics help monitor system health and identify performance bottlenecks or data quality issues.

Pull-Based Scraping Process

The pull-based scraping process involves the metrics system actively retrieving data from external endpoints that expose metrics in standard formats like Prometheus exposition format. This process runs on configurable intervals and handles endpoint discovery, metric parsing, and error recovery.

Endpoint Discovery and Scheduling

The `ScrapeEndpoint` method manages a registry of target endpoints that should be scraped for metrics. Each target includes the endpoint URL, scraping interval, timeout settings, and any required authentication credentials. The system maintains a scheduler that triggers scrape operations at the appropriate intervals for each target.

The scraping scheduler uses a priority queue to manage scrape timing, ensuring that endpoints are scraped at their configured intervals while distributing the load evenly over time. If a scrape operation takes longer than expected, the scheduler can detect this and adjust future scheduling to prevent overlapping scrapes that could overwhelm either the metrics system or the target endpoint.

HTTP Client Operations and Format Parsing

When scraping an endpoint, the system makes an HTTP GET request with appropriate timeout settings and authentication headers. The response body typically contains metrics in Prometheus exposition format, which uses a text-based representation with metric names, labels, values, and timestamps.

The parsing process converts the text format into internal `Metric` data structures, handling the various metric types and label formats supported by the exposition format. This includes parsing counter metrics (with `_total` suffixes), gauge metrics, histogram metrics (with `_bucket`, `_count`, and `_sum` suffixes), and extracting label values from the metric names and label sets.

Error Handling and Retry Logic

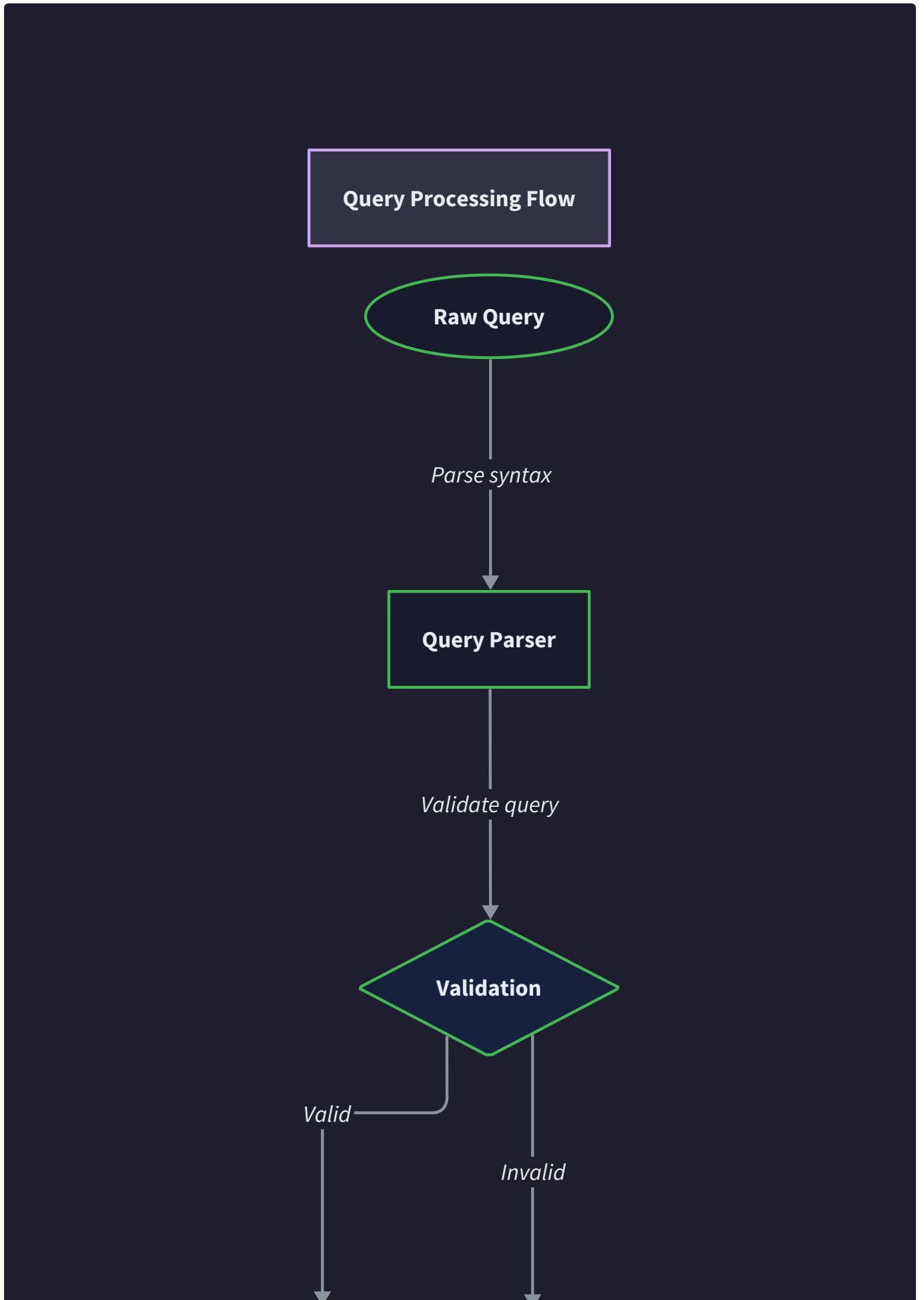
Scraping operations can fail due to network issues, endpoint unavailability, or invalid data formats. The system implements exponential backoff retry logic to handle transient failures without overwhelming failing endpoints. For persistent failures, the system logs errors and continues attempting scrapes at reduced frequencies while alerting operators to the issue.

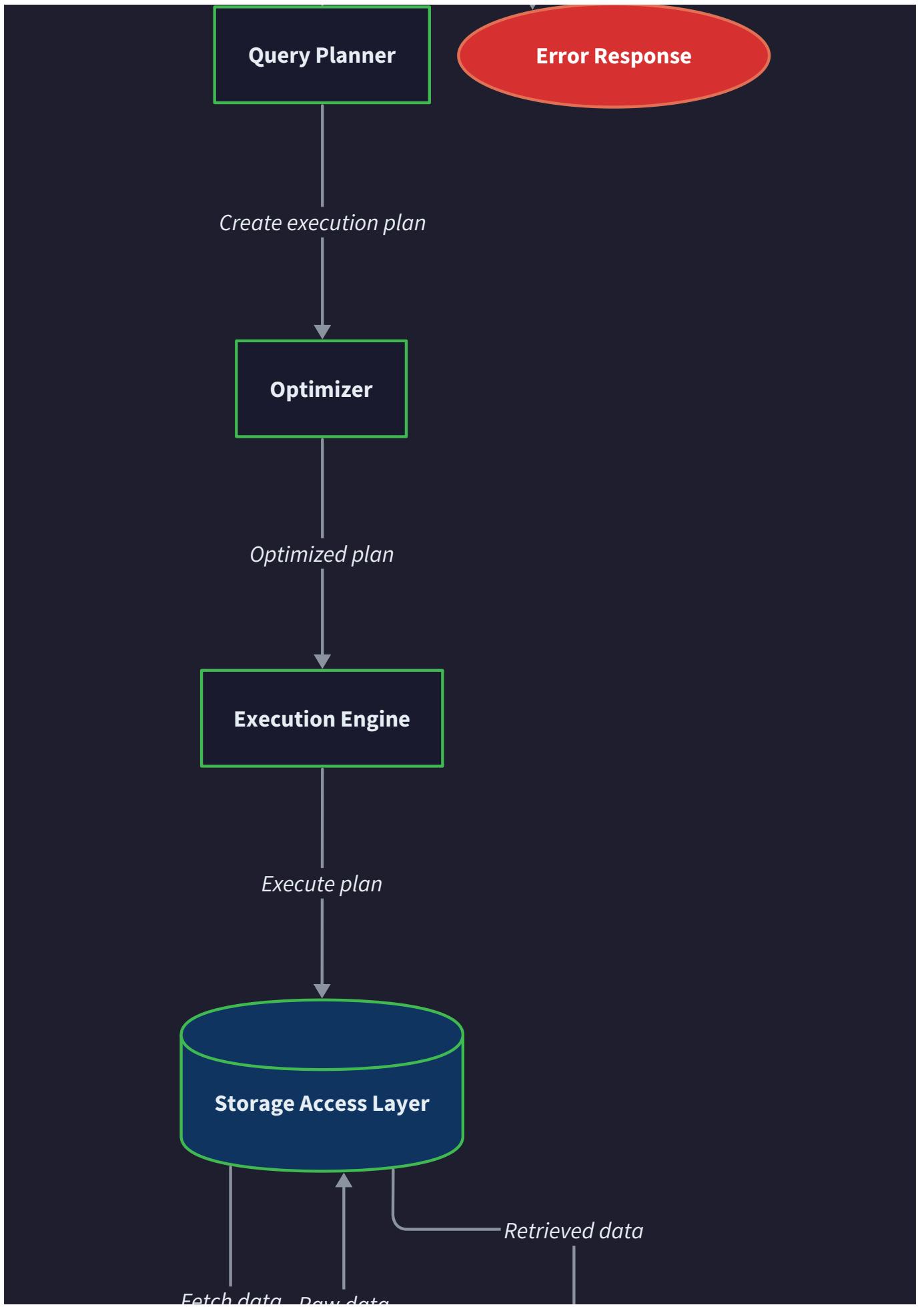
Common Ingestion Flow Patterns

Flow Type	Trigger	Processing Steps	Success Outcome	Failure Handling
Batch Push	HTTP POST with metric array	Validate → Normalize → Store → Respond	200 OK with statistics	400/500 with error details
Single Push	HTTP POST with single metric	Same as batch but optimized	200 OK with confirmation	Detailed validation errors
Prometheus Scrape	Timer-triggered pull	Fetch → Parse → Validate → Store	Updated series data	Retry with backoff
Health Check	Monitor request	Quick validation only	System status	Degraded/unhealthy status

Query Processing Flow

The query processing flow transforms user requests for metric data into efficient storage operations and properly formatted results. This flow must handle query parsing, execution planning, storage retrieval, data aggregation, and result formatting while maintaining good performance and accurate results.







Query Request Initiation

Query processing begins when a client submits a query request through either the dashboard API or alerting system. The request contains a query expression string, time range specification, and optional parameters like maximum data points or output format preferences.

The `QueryProcessor` receives these requests through its `ExecuteQuery` method, which serves as the main entry point for all query operations. The processor immediately assigns a unique query ID for tracking and logging purposes, starts timing

measurement for performance monitoring, and validates basic request parameters like time range sanity (end time after start time, reasonable date ranges).

Query Request Validation

Before parsing the query string, the system performs preliminary validation to catch obvious errors early and prevent resource waste on malformed requests. This validation checks that the time range is reasonable (not requesting data from the year 2050 or from before the system was deployed), the query string is not empty, and any optional parameters are within acceptable ranges.

For queries originating from the dashboard, additional validation ensures that the requesting user has appropriate permissions to access the requested metrics, particularly if the system implements any access control policies based on metric labels or namespaces.

Query Parsing and AST Generation

The query parser transforms the query string into a structured `QueryAST` that represents the query's logical structure. This parsing process uses a lexical analyzer (`Lexer`) to break the query string into tokens, followed by a recursive descent parser that builds

the abstract syntax tree.

Lexical Analysis Phase

The lexer scans the query string character by character, identifying tokens like metric names, label selectors, function calls, operators, and literal values. Each token gets classified with a `TokenType` (such as `TokenMetricName`, `TokenLabelName`, `TokenFunction`) and stored with its string value and position in the original query.

The lexical analysis handles various query syntax elements: quoted strings for label values containing special characters, numeric literals for threshold values and time durations, and reserved keywords for functions and operators. Error handling during lexing provides detailed feedback about syntax issues, including the character position where parsing failed.

Syntax Tree Construction

The parser consumes tokens from the lexer and constructs a hierarchical `QueryAST` representing the query's logical structure.

The tree contains different node types: `MetricSelectorNode` for basic metric selection with label filters, `FunctionCallNode` for aggregation and transformation operations, and `BinaryOpNode` for arithmetic operations between query results.

Each AST node implements the `ASTNode` interface with methods for type identification and string representation. This design enables the system to traverse the tree, validate query semantics, and generate execution plans while maintaining clean separation between parsing and execution concerns.

Semantic Validation

After syntax parsing, the system performs semantic validation to ensure the query makes logical sense. This validation checks that referenced metric names exist in the system, label names follow proper conventions, function calls use correct parameter types and counts, and time range specifications are consistent with the query structure.

Semantic validation also detects potential performance issues like queries that would need to scan excessive amounts of data or generate extremely high-cardinality result sets. These checks help prevent queries that could overwhelm the system or consume excessive memory during execution.

Query Execution Planning

The query planner analyzes the parsed AST and generates an optimized `ExecutionPlan` that specifies how the query should be executed against the storage engine. This planning phase is crucial for query performance because it determines the order of operations, identifies optimization opportunities, and estimates resource requirements.

Series Selection Optimization

The planner examines `MetricSelectorNode` elements in the AST to identify which time series need to be retrieved from storage. It analyzes label matchers to estimate the number of matching series and determines the most efficient retrieval strategy based on available indexes and label cardinality.

For queries with multiple label filters, the planner determines the optimal order for applying filters: starting with the most selective filters to minimize the amount of data that needs to be processed by subsequent filters. This optimization can dramatically improve query performance when dealing with high-cardinality metrics.

Aggregation Strategy Selection

When the query includes aggregation functions (like sum, average, or percentile calculations), the planner determines whether aggregations can be performed incrementally as data is retrieved (streaming aggregation) or require buffering entire result sets in memory. This decision depends on the specific function requirements and available system memory.

The planner also identifies opportunities for query result caching, particularly for expensive aggregations over static time ranges that are unlikely to change. Cached results can be reused for identical queries or combined with incremental calculations for queries that extend previous time ranges.

Parallel Execution Planning

For queries that need to process large amounts of data, the planner can divide the work into parallel execution phases. This might involve processing different time ranges concurrently, distributing aggregation calculations across multiple worker threads, or parallelizing series retrieval operations.

The parallelization strategy considers system resources, storage architecture, and query characteristics to balance performance improvements against coordination overhead. Simple queries may execute faster with single-threaded processing, while complex aggregations over large datasets benefit significantly from parallel execution.

Storage Interaction and Data Retrieval

The execution engine interacts with the `StorageEngine` through well-defined interfaces to retrieve the time-series data needed for query execution. This interaction must efficiently handle different query patterns while minimizing storage system load and memory usage.

Series Discovery and Selection

Using the label matchers from the execution plan, the query engine calls `QuerySeries` to identify all time series that match the query criteria. This operation returns `SeriesInfo` objects containing series metadata but not the actual sample data, allowing the system to understand the scope of data retrieval before loading samples into memory.

The series selection process leverages the `SeriesIndex` to quickly identify matching series without scanning raw storage blocks. For queries with multiple label filters, the selection process can use index intersection operations to efficiently find series that match all criteria simultaneously.

Sample Data Retrieval

For each selected series, the query engine calls `QueryRange` to retrieve sample data within the specified time range. The storage engine returns arrays of `Sample` objects containing timestamps and values, which the query engine then processes according to the execution plan.

To manage memory usage and processing efficiency, the query engine may retrieve sample data in batches rather than loading entire time ranges into memory simultaneously. This streaming approach enables processing of very large result sets without exhausting system memory.

Data Alignment and Interpolation

When processing multiple time series, the query engine often needs to align samples from different series to common timestamps for aggregation or arithmetic operations. This alignment process may involve interpolating missing values or selecting the most recent value before each timestamp.

The alignment strategy depends on the metric types and query requirements: counter metrics typically use rate calculations that account for counter resets, gauge metrics may use linear interpolation for missing values, and histogram metrics require special handling for bucket boundaries and statistical calculations.

Aggregation and Result Processing

After retrieving raw sample data from storage, the query engine applies aggregation functions and transformations specified in the query to compute final results. This processing phase must handle various aggregation types while maintaining numerical accuracy and handling edge cases like missing data.

Statistical Aggregation Functions

The system implements various `AggregationFunction` types for different statistical operations: sum and average for combining multiple series, min and max for finding extreme values, and percentile calculations for understanding data distributions. Each function handles streaming aggregation where possible to minimize memory usage.

For histogram metrics, the system provides specialized aggregation functions that properly combine histogram buckets from multiple series, calculate quantile estimates across merged histograms, and maintain statistical accuracy even when dealing with different bucket boundaries across series.

Time-Based Processing

Many queries require processing data over time windows, such as calculating rates of change for counter metrics or computing moving averages for gauge metrics. The query engine implements these time-based operations by maintaining sliding windows of sample data and updating calculations as new samples are processed.

Rate calculations for counter metrics require special handling for counter resets, where the counter value decreases (typically due to process restarts). The system detects these resets and adjusts rate calculations to maintain accuracy across reset boundaries.

Result Formatting and Output

The final query processing step formats the computed results according to the client's requirements. Dashboard queries typically receive results as time-series arrays with timestamp-value pairs, while alerting queries may only need scalar values or boolean results indicating threshold violations.

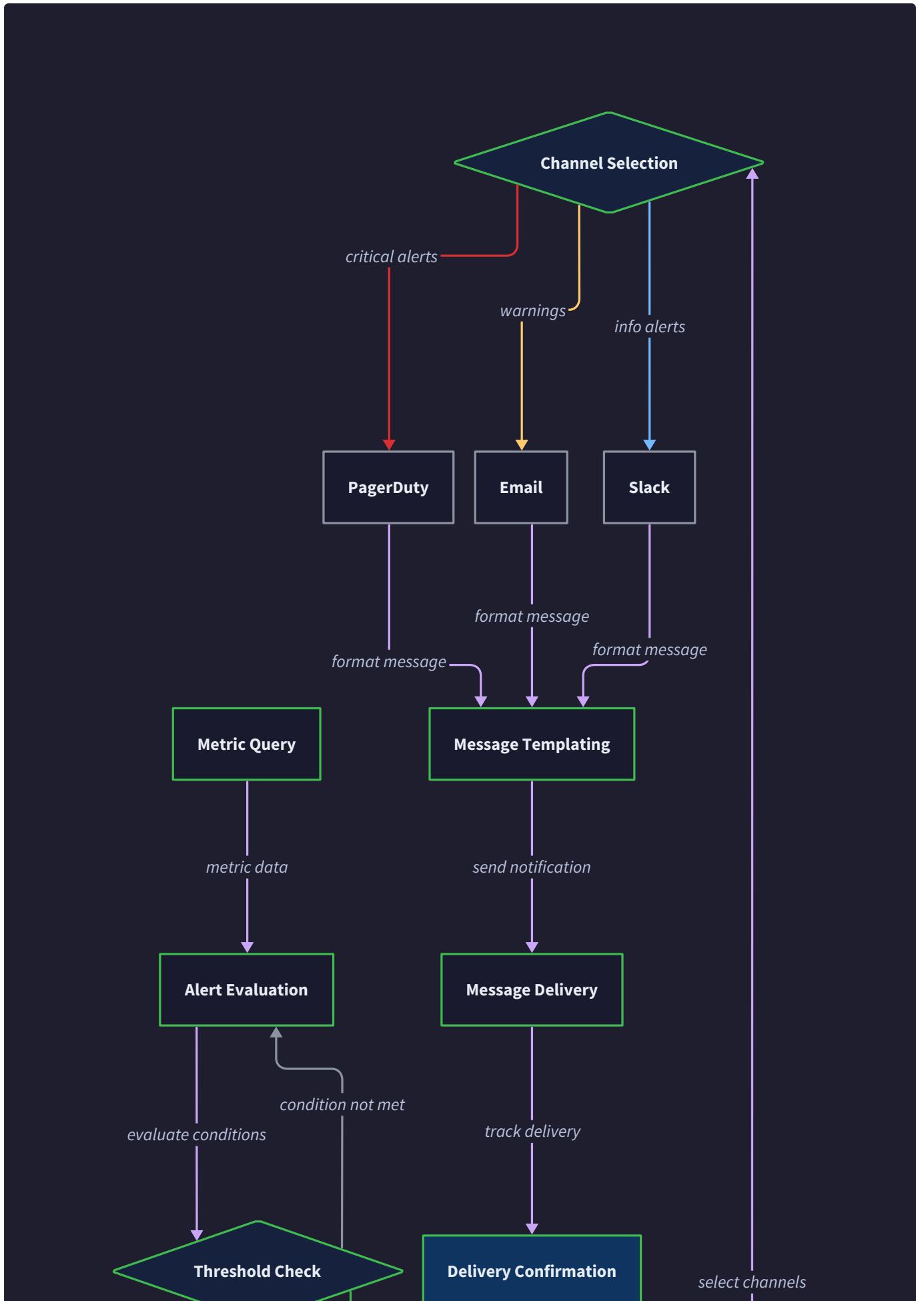
The result formatting process includes unit conversion (if requested), timestamp formatting for the client's time zone preferences, and data point sampling or aggregation if the client requested a maximum number of data points. This final processing ensures that results are immediately usable by the requesting component without additional transformation.

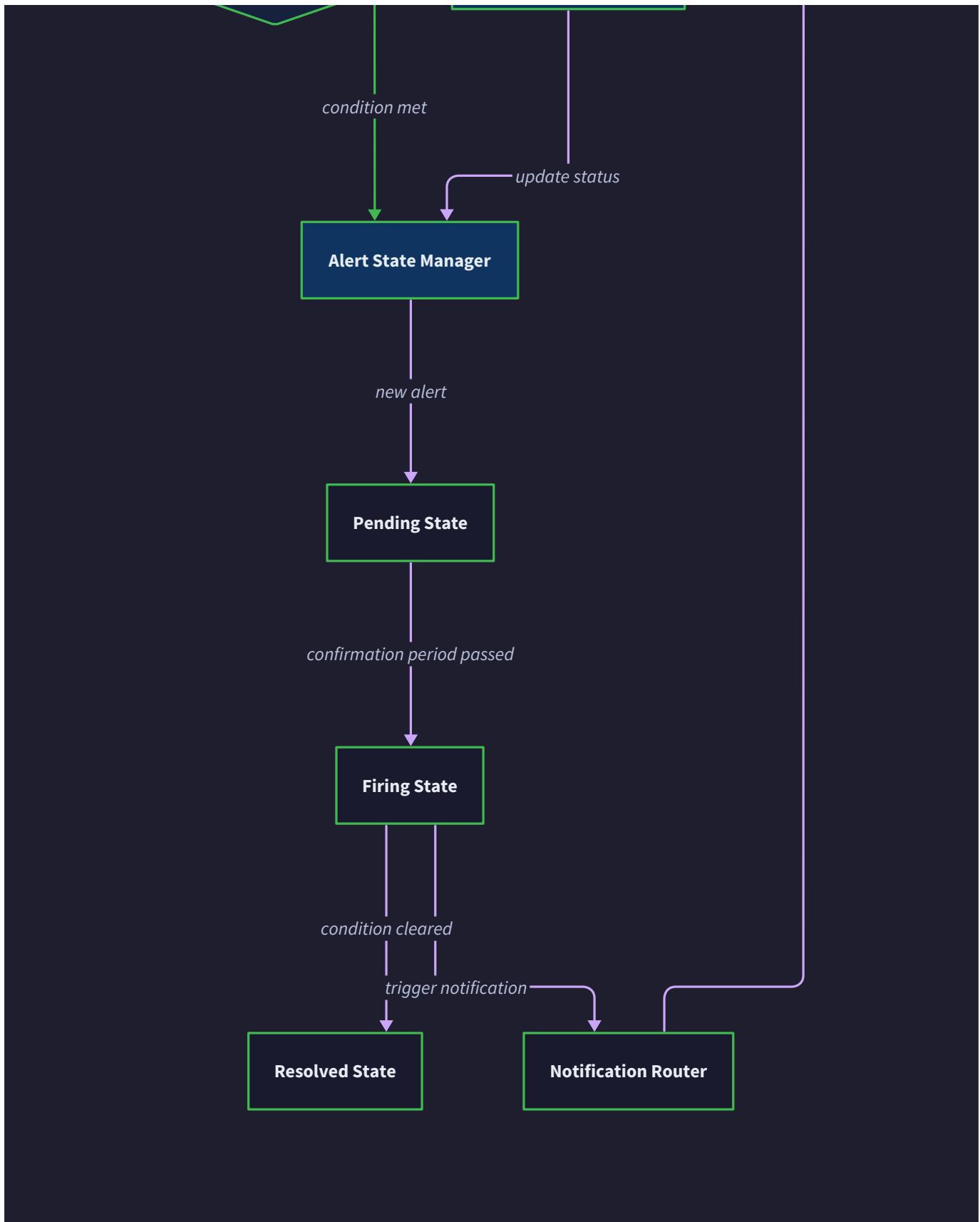
Query Processing Flow Patterns

Query Type	Parsing Complexity	Storage Operations	Aggregation Needs	Result Format
Simple Series	Low (single selector)	Single series lookup	None	Raw time-series
Multi-Series Sum	Medium (aggregation function)	Multiple series retrieval	Sum across series	Aggregated time-series
Rate Calculation	Medium (rate function)	Series with counter handling	Rate computation	Derived time-series
Complex Dashboard	High (multiple subqueries)	Optimized batch retrieval	Mixed aggregations	Multiple result sets
Alert Evaluation	Low (threshold comparison)	Recent data only	Threshold comparison	Boolean or scalar

Alert Evaluation Flow

The alert evaluation flow continuously monitors metric data against defined alert rules, manages alert state transitions, and triggers appropriate notifications when conditions change. This flow must provide reliable monitoring, minimize false positives, and ensure timely notification delivery.





Alert Rule Evaluation Cycle

The alert evaluation process runs on a regular schedule, typically every 15-60 seconds, to check all active alert rules against current metric data. The `AlertEvaluator` coordinates this process, managing the evaluation schedule, tracking alert states, and triggering notifications when state changes occur.

Evaluation Scheduling and Coordination

The `AlertEvaluator` maintains a scheduler that triggers evaluation cycles at the configured `EvaluationInterval`. Each evaluation cycle processes all active `AlertRule` definitions, executing their query expressions against the current metric data and comparing results to defined thresholds.

The scheduling system uses a ticker mechanism to ensure consistent evaluation intervals, but includes logic to handle cases where evaluations take longer than the scheduled interval. If an evaluation cycle is still running when the next cycle should start, the system can either wait for completion or run evaluations in parallel, depending on system resources and configuration.

Rule Query Execution

For each `AlertRule`, the evaluator executes the rule's query expression using the same `QueryProcessor` used by the dashboard system. This ensures consistency between what users see in dashboards and what the alerting system monitors. The query typically retrieves recent data (last few minutes) to determine current metric values.

Alert rule queries often include aggregation functions to reduce multiple time series to scalar values that can be compared against thresholds. For example, a rule monitoring average CPU usage might aggregate samples from multiple servers, while a rule monitoring error rates might calculate the ratio of error count to total request count.

Threshold Comparison and Condition Evaluation

After executing the rule query, the evaluator compares the resulting values against the rule's threshold using the specified comparison operator (greater than, less than, equals, etc.). This comparison determines whether the alert condition is currently met for each time series that matches the rule's selector.

For rules that monitor multiple time series (like per-server metrics), the evaluator creates separate `AlertInstance` objects for each series that violates the threshold. Each instance tracks its own state and history independently, allowing fine-grained alerting on individual series while maintaining overall rule coherence.

Duration-Based Alert Logic

Many alert rules include a duration requirement: the condition must persist for a specified time before the alert fires. This prevents spurious alerts from brief metric spikes or temporary network issues. The evaluator tracks how long each condition has been true and only transitions alerts to firing state after the duration threshold is met.

The duration tracking uses the alert instance's state history to determine when the condition first became true. If the condition stops being met before the duration expires, the instance returns to inactive state and the duration timer resets. This ensures that only sustained threshold violations trigger alerts.

Alert State Management

The `StateManager` component maintains the lifecycle of all alert instances, tracking state transitions, managing persistence, and ensuring proper notification behavior. Alert states follow a well-defined state machine that prevents inconsistent behavior and ensures reliable alerting.

Alert State Machine Implementation

Each `AlertInstance` progresses through defined states based on evaluation results and time constraints:

Current State	Condition Met	Duration Exceeded	Next State	Actions Taken
<code>AlertStateInactive</code>	No	N/A	<code>AlertStateInactive</code>	No action
<code>AlertStateInactive</code>	Yes	No	<code>AlertStatePending</code>	Start duration timer
<code>AlertStatePending</code>	Yes	Yes	<code>AlertStateFiring</code>	Send firing notification
<code>AlertStatePending</code>	No	N/A	<code>AlertStateInactive</code>	Cancel duration timer
<code>AlertStateFiring</code>	Yes	N/A	<code>AlertStateFiring</code>	No notification (already firing)
<code>AlertStateFiring</code>	No	N/A	<code>AlertStateResolved</code>	Send resolved notification
<code>AlertStateResolved</code>	No	N/A	<code>AlertStateInactive</code>	Cleanup after grace period
Any	Silenced	N/A	<code>AlertStateSilenced</code>	Suppress notifications

State Persistence and Recovery

The `StateManager` persists alert state information to survive system restarts and ensure continuity of alert monitoring. This persistence includes current state, state change timestamps, condition values, and notification history for each alert instance.

During system startup, the state manager loads persisted alert states and reconciles them with current rule definitions. If rules have been modified or deleted, the system handles orphaned alert instances appropriately, either updating them to match new rule

definitions or marking them for cleanup.

State Change Notification Triggering

When an alert instance transitions to a new state that requires notification (`AlertStateFiring` or `AlertStateResolved`), the state manager creates `NotificationRequest` objects and queues them for delivery through the `NotificationManager`.

These requests include all necessary information for generating and sending appropriate notifications.

The state manager also maintains statistics about alert behavior, tracking metrics like time spent in each state, frequency of state changes, and notification success rates. This information helps operators understand alert behavior patterns and identify rules that may need tuning.

Notification Processing and Delivery

The notification system handles the complex process of generating appropriate messages for different channels, managing delivery queues, handling failures, and ensuring reliable notification delivery even under adverse conditions.

Notification Channel Selection and Routing

When an alert state change triggers a notification, the system determines which notification channels should receive the alert based on rule configuration, channel filters, and current system state. Different alert rules may route to different channels based on severity, affected services, or time of day.

The routing logic evaluates each configured `NotificationChannel` against the alert instance to determine if the channel should receive the notification. This evaluation considers channel filters (which may match alert labels), channel availability status, and any active silencing rules that might suppress notifications for specific channels.

Message Templating and Formatting

For each selected notification channel, the system generates appropriate message content using configured message templates.

The templating system has access to all alert instance data, including current values, threshold settings, alert history, and custom annotations from the alert rule definition.

Different channels require different message formats: email notifications might include detailed formatting with tables and charts, Slack notifications use rich message formatting with action buttons, while SMS notifications need concise text that fits within message length limits. The templating system handles these variations while maintaining consistent information content.

Delivery Queue Management and Retry Logic

Notification delivery uses a queue-based system that can handle high notification volumes and temporary delivery failures without blocking alert evaluation or losing notifications. The `NotificationManager` maintains separate delivery queues for different channel types, allowing independent processing and retry logic.

When notification delivery fails (due to network issues, service outages, or configuration problems), the system implements exponential backoff retry logic with maximum retry limits. Failed notifications are logged with detailed error information to help operators diagnose and resolve delivery issues.

Rate Limiting and Notification Throttling

To prevent notification storms that could overwhelm receiving systems or alert recipients, the notification system implements rate limiting based on channel configuration and alert frequency. This includes burst limiting (maximum notifications per minute) and sustained rate limiting (maximum notifications per hour).

The rate limiting system can also implement intelligent throttling that reduces notification frequency for frequently-flapping alerts while maintaining normal delivery for stable alerts. This helps reduce alert fatigue while ensuring that genuine issues receive appropriate attention.

Alert Evaluation Flow Patterns

Evaluation Type	Trigger	Query Complexity	State Changes	Notification Volume
Simple Threshold	Scheduled timer	Single metric comparison	Infrequent (stable conditions)	Low (binary firing/resolved)
Multi-Series Rule	Scheduled timer	Aggregated query	Variable (per-series instances)	Medium (multiple instances)
Flapping Alert	Scheduled timer	Same query repeatedly	Frequent (unstable conditions)	High (requires throttling)
Composite Condition	Scheduled timer	Multiple subqueries	Complex (dependent conditions)	Variable (conditional logic)

Integration with Dashboard and Query Systems

The alert evaluation system leverages the same query processing infrastructure used by the dashboard system, ensuring consistency between what users observe in visualizations and what triggers alerts. This shared infrastructure reduces system complexity and maintains behavioral consistency.

Shared Query Processing Pipeline

Alert rule evaluation uses the same `QueryProcessor`, query parsing logic, and storage interfaces as dashboard queries. This means that alert expressions can use the same syntax and functions available in dashboard queries, and operators can test alert conditions by running equivalent queries in the dashboard interface.

The shared pipeline also ensures that optimizations and bug fixes in the query processing system automatically benefit both dashboard and alerting functionality. This reduces maintenance burden and improves overall system reliability.

Real-Time Data Consistency

Since both dashboard and alerting systems query the same underlying storage, users can observe the same metric values that drive alert decisions. This consistency is crucial for debugging alert behavior and understanding why alerts fire or resolve at specific times.

The system maintains this consistency even during high-load periods by using the same data retrieval and aggregation logic for both use cases. However, alert queries typically focus on recent data and use simpler aggregations to minimize evaluation latency and resource usage.

Implementation Guidance

This section provides practical implementation advice for building the component interaction and data flow systems described above. The code examples and patterns shown here will help translate the design concepts into working software.

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	Go net/http with gorilla/mux	Go fiber or gin frameworks
Message Serialization	JSON with encoding/json	Protocol Buffers with protobuf
Concurrency Control	Go channels and sync package	WorkerPool patterns with semaphores
Logging	Go slog package	Structured logging with zerolog
Metrics Instrumentation	Prometheus client library	Custom metrics with OpenTelemetry
Configuration	YAML files with gopkg.in/yaml	Consul/etcd with dynamic updates

Project Structure

```
project-root/
├── cmd/
│   ├── metrics-server/
│   │   └── main.go           ← Main server entry point
│   └── metrics-cli/
│       └── main.go          ← CLI tools for testing
├── internal/
│   ├── coordinator/
│   │   ├── coordinator.go    ← ComponentCoordinator implementation
│   │   ├── component.go      ← Component interface and base types
│   │   └── coordinator_test.go
│   ├── ingestion/
│   │   ├── ingestor.go       ← MetricsIngestor implementation
│   │   ├── validator.go      ← Validation and normalization
│   │   ├── cardinality.go    ← CardinalityManager implementation
│   │   └── handlers.go       ← HTTP handlers for ingestion
│   ├── storage/
│   │   ├── engine.go         ← StorageEngine implementation
│   │   ├── series.go         ← SeriesIndex and SeriesInfo
│   │   ├── wal.go            ← WriteAheadLog implementation
│   │   └── blocks.go         ← Block storage management
│   ├── query/
│   │   ├── processor.go      ← QueryProcessor implementation
│   │   ├── parser.go          ← Query parsing and AST
│   │   ├── executor.go        ← Query execution engine
│   │   └── cache.go           ← QueryCache implementation
│   ├── dashboard/
│   │   ├── server.go          ← DashboardServer implementation
│   │   ├── websocket.go       ← WebSocket client management
│   │   └── handlers.go        ← HTTP handlers for dashboard API
│   ├── alerting/
│   │   ├── evaluator.go       ← AlertEvaluator implementation
│   │   ├── state.go            ← StateManager implementation
│   │   ├── notifications.go    ← NotificationManager implementation
│   │   └── channels/           ← Channel-specific implementations
│   └── common/
│       ├── types.go           ← Shared type definitions
│       ├── config.go          ← Configuration structures
│       └── health.go          ← HealthManager implementation
└── pkg/
    └── client/                ← Client library for external use
└── scripts/
    ├── docker-compose.yml     ← Development environment
    └── test-data/              ← Sample metrics for testing
```

Core Component Coordination Infrastructure

The following code provides the foundation for component lifecycle management and coordination:

GO

```
// internal/coordinator/component.go

package coordinator

import (
    "context"
    "fmt"
    "log/slog"
    "sync"
    "time"
)

// Component represents a system component with lifecycle management

type Component interface {

    // Start initializes and starts the component

    Start(ctx context.Context) error

    // Stop gracefully shuts down the component

    Stop(ctx context.Context) error

    // Name returns the component identifier

    Name() string

    // HealthCheck returns the current component health status

    HealthCheck(ctx context.Context) error
}

// ComponentCoordinator manages the lifecycle of all system components

type ComponentCoordinator struct {

    config      *Config
    logger      *slog.Logger
    health      *HealthManager
    components map[string]Component
    mu          sync.RWMutex
}
```

```
started    bool

stopCh     chan struct{}

wg         sync.WaitGroup

}

// NewCoordinator creates a new component coordinator

func NewCoordinator(config *Config, logger *slog.Logger) *ComponentCoordinator {
    return &ComponentCoordinator{
        config:    config,
        logger:   logger,
        health:   NewHealthManager(),
        components: make(map[string]Component),
        stopCh:    make(chan struct{}),
    }
}

// RegisterComponent adds a component to the coordinator

func (c *ComponentCoordinator) RegisterComponent(component Component) error {
    c.mu.Lock()
    defer c.mu.Unlock()

    if c.started {
        return fmt.Errorf("cannot register component %s: coordinator already started", component.Name())
    }

    if _, exists := c.components[component.Name()]; exists {
        return fmt.Errorf("component %s already registered", component.Name())
    }

    c.components[component.Name()] = component
    c.logger.Info("registered component", "name", component.Name())

    return nil
}
```

```
// Start initializes and starts all registered components

func (c *ComponentCoordinator) Start(ctx context.Context) error {
    c.mu.Lock()

    defer c.mu.Unlock()

    if c.started {
        return fmt.Errorf("coordinator already started")
    }

    // TODO 1: Start each component in dependency order
    // TODO 2: Register health checks for each component
    // TODO 3: Start background health monitoring
    // TODO 4: Set up graceful shutdown signal handling
    // TODO 5: Mark coordinator as started

    return nil
}

// Stop gracefully shuts down all components

func (c *ComponentCoordinator) Stop(ctx context.Context) error {
    c.mu.Lock()

    defer c.mu.Unlock()

    if !c.started {
        return nil
    }

    // TODO 1: Signal all components to stop
    // TODO 2: Wait for components to finish with timeout
    // TODO 3: Force stop any components that don't respond
    // TODO 4: Clean up resources and close channels
}
```

```
    return nil  
}  
}
```

Ingestion Flow Infrastructure

```
// internal/ingestion/ingester.go                                     GO

package ingestion

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

// MetricsIngester handles incoming metric data with validation and storage

type MetricsIngester struct {

    storage      TimeSeriesStorage
    validator    Validator
    cardinality  CardinalityManager
    logger       *slog.Logger
    stats        *IngestionStats

    // Configuration
    maxBatchSize int
    timeout      time.Duration
}

// NewMetricsIngester creates a new metrics ingester

func NewMetricsIngester(storage TimeSeriesStorage, validator Validator, cardinality CardinalityManager,
    logger *slog.Logger) *MetricsIngester {
    return &MetricsIngester{
        storage:      storage,
        validator:    validator,
        cardinality: cardinality,
        logger:       logger,
        stats:        &IngestionStats{},
    }
}
```

```
    maxBatchSize: 1000,  
  
    timeout:      30 * time.Second,  
}  
  
}  
  
  
// IngestMetrics processes a batch of metrics through the complete ingestion pipeline  
  
func (m *MetricsIngestor) IngestMetrics(ctx context.Context, metrics []Metric) error {  
  
    startTime := time.Now()  
  
    defer func() {  
  
        m.logger.Debug("ingestion completed",  
            "duration", time.Since(startTime),  
            "metric_count", len(metrics))  
  
    }()  
  
  
    // TODO 1: Validate batch size and basic request parameters  
  
    // TODO 2: For each metric in the batch:  
  
    //   a. Validate metric structure and values using m.validator.ValidateMetric()  
  
    //   b. Check cardinality limits using m.cardinality.CheckCardinality()  
  
    //   c. Normalize metric format using m.validator.NormalizeMetric()  
  
    //   d. Generate series ID and update series index  
  
    // TODO 3: Write all validated samples to storage using m.storage.WriteSamples()  
  
    // TODO 4: Update ingestion statistics (success/failure counts)  
  
    // TODO 5: Return any validation errors or storage failures  
  
  
    return nil  
}  
  
  
// ScrapeEndpoint retrieves metrics from a Prometheus-compatible endpoint  
  
func (m *MetricsIngestor) ScrapeEndpoint(ctx context.Context, url string) error {  
  
    // TODO 1: Make HTTP GET request to the endpoint with timeout  
  
    // TODO 2: Parse Prometheus exposition format from response body  
  
    // TODO 3: Convert parsed metrics to internal Metric structures  
  
    // TODO 4: Call IngestMetrics() with the converted metrics
```

```
// TODO 5: Handle scraping errors and update endpoint statistics

    return nil
}

// HTTP handler for push-based ingestion

func (m *MetricsIngestor) HandlePushMetrics(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Validate HTTP method and content-type

    // TODO 2: Read and parse JSON request body into Metric slice

    // TODO 3: Call IngestMetrics() with parsed metrics

    // TODO 4: Return appropriate HTTP status and response body

    // TODO 5: Log request details and performance metrics

}
```

Query Processing Infrastructure

```
// internal/query/processor.go                                     GO

package query

import (
    "context"
    "fmt"
    "time"
)

// QueryProcessor executes metric queries and returns formatted results

type QueryProcessor struct {

    storage     TimeSeriesStorage
    parser      *QueryParser
    cache       QueryCache
    logger      *slog.Logger

    // Execution limits

    maxSeries    int
    maxSamples   int64
    queryTimeout time.Duration
}

// NewQueryProcessor creates a new query processor

func NewQueryProcessor(storage TimeSeriesStorage, logger *slog.Logger) *QueryProcessor {
    return &QueryProcessor{
        storage:     storage,
        parser:      NewQueryParser(),
        cache:       NewQueryCache(1000, 5*time.Minute),
        logger:      logger,
        maxSeries:   10000,
        maxSamples:  1000000,
        queryTimeout: 30 * time.Second,
    }
}
```

```
}

}

// ExecuteQuery processes a query string and returns formatted results

func (q *QueryProcessor) ExecuteQuery(ctx context.Context, queryString string, timeRange TimeRange)
(*QueryResult, error) {

    startTime := time.Now()

    defer func() {

        q.logger.Debug("query executed",
            "duration", time.Since(startTime),
            "query", queryString)

    }()

    // TODO 1: Check query cache for existing results

    // TODO 2: Parse query string into AST using q.parser.Parse()

    // TODO 3: Validate query semantics and estimate resource requirements

    // TODO 4: Generate optimized execution plan using PlanQuery()

    // TODO 5: Execute plan against storage:
    //
    //   a. Retrieve matching series using storage.QuerySeries()
    //
    //   b. Fetch sample data using storage.QueryRange()
    //
    //   c. Apply aggregation functions and transformations

    // TODO 6: Format results according to client requirements

    // TODO 7: Update query cache with results

    // TODO 8: Return QueryResult with data and execution statistics

    return nil, nil
}
```

Alert Evaluation Infrastructure

```
// internal/alerting/evaluator.go                                     GO

package alerting

import (
    "context"
    "sync"
    "time"
)

// AlertEvaluator continuously evaluates alert rules and manages alert states

type AlertEvaluator struct {

    queryProcessor     QueryProcessor
    stateManager      *StateManager
    ruleManager       *RuleManager
    notificationMgr  *NotificationManager
    logger            *slog.Logger

    // Evaluation control

    evaluationInterval time.Duration
    stopCh             chan struct{}
    wg                sync.WaitGroup
    running           bool
    mu               sync.RWMutex
}

// NewAlertEvaluator creates a new alert evaluator

func NewAlertEvaluator(queryProcessor QueryProcessor, stateManager *StateManager, notificationMgr *NotificationManager, logger *slog.Logger) *AlertEvaluator {
    return &AlertEvaluator{
        queryProcessor:     queryProcessor,
        stateManager:      stateManager,
        notificationMgr:  notificationMgr,
        logger:            logger,
    }
}
```

```
    evaluationInterval: 15 * time.Second,
    stopCh:           make(chan struct{}),
}

}

// Start begins the alert evaluation loop

func (a *AlertEvaluator) Start(ctx context.Context) error {
    a.mu.Lock()
    defer a.mu.Unlock()

    if a.running {
        return fmt.Errorf("alert evaluator already running")
    }

    // TODO 1: Start the evaluation ticker
    // TODO 2: Launch background goroutine for evaluation loop
    // TODO 3: Set up context cancellation handling
    // TODO 4: Mark evaluator as running

    a.running = true
    return nil
}

// EvaluateRules processes all active alert rules against current metric data

func (a *AlertEvaluator) EvaluateRules(ctx context.Context) error {
    // TODO 1: Get all active alert rules from rule manager
    // TODO 2: For each rule:
    //     a. Execute rule query using query processor
    //     b. Compare results against rule threshold
    //     c. Update alert instance state using state manager
    //     d. Trigger notifications if state changed
    // TODO 3: Clean up resolved alerts older than retention period
    // TODO 4: Update evaluation statistics and health metrics
}
```

```

    return nil
}

// UpdateAlertState manages alert state transitions and notifications

func (a *AlertEvaluator) UpdateAlertState(ctx context.Context, ruleID string, seriesLabels Labels,
currentValue float64, threshold float64, conditionMet bool, rule *AlertRule) error {

    // TODO 1: Get current alert instance from state manager

    // TODO 2: Determine next state based on current state and condition

    // TODO 3: Check duration requirements for state transitions

    // TODO 4: Update alert instance with new state and value

    // TODO 5: If state changed to firing or resolved, queue notifications

    return nil
}

```

Milestone Checkpoints

After Implementing Component Coordination (Foundation)

- Run: `go test ./internal/coordinator/...`
- Expected: All tests pass, components start/stop cleanly
- Manual test: Start server, check health endpoint returns 200 OK
- Debug: Check logs for component registration and startup sequence

After Implementing Ingestion Flow (Milestone 1)

- Run: `curl -X POST http://localhost:8080/api/v1/metrics -d '[{"name":"test_counter","type":0,"labels":{"job":"test"},"samples":[{"value":42,"timestamp":"2024-01-01T12:00:00Z"}]]'`
- Expected: 200 OK response with ingestion statistics
- Manual test: Send invalid metrics, verify proper error responses
- Debug: Check ingestion logs and storage files for metric data

After Implementing Query Processing (Milestone 2)

- Run: `curl "http://localhost:8080/api/v1/query?query=test_counter&start=2024-01-01T11:00:00Z&end=2024-01-01T13:00:00Z"`
- Expected: JSON response with time-series data
- Manual test: Query non-existent metrics, verify empty results
- Debug: Check query logs and execution time statistics

After Implementing Alert Evaluation (Milestone 4)

- Configure alert rule with low threshold, send metrics that exceed it
- Expected: Alert transitions to firing state, notification sent

- Manual test: Verify alert resolves when condition clears
- Debug: Check alert evaluation logs and state persistence

Common Debugging Patterns

Symptom	Likely Cause	Diagnostic Steps	Resolution
Metrics not appearing in queries	Ingestion validation failure	Check ingestion endpoint logs, verify metric format	Fix metric names/labels, check cardinality limits
Slow query performance	Inefficient series selection	Enable query execution logging, check series count	Add label indexes, optimize query selectors
Alerts not firing	Query execution errors	Check alert evaluation logs, test queries manually	Fix alert rule expressions, verify metric availability
WebSocket disconnections	Client timeout or server overload	Monitor connection counts, check message queue sizes	Tune timeout settings, implement backpressure
Memory usage growth	Query result caching or buffer leaks	Profile memory usage, check cache sizes	Implement cache eviction, fix buffer cleanup

⚠ Pitfall: Blocking Operations in Component Startup Components that perform blocking I/O during startup (like connecting to external services) can prevent the entire system from starting. Always use timeouts and context cancellation for startup operations. If a component fails to start, the coordinator should continue with other components rather than failing completely.

⚠ Pitfall: Missing Error Context in Data Flows When errors occur in the middle of data processing pipelines, insufficient context makes debugging nearly impossible. Always include relevant identifiers (series ID, query ID, alert rule ID) in error messages and logs. Use structured logging to capture processing context.

⚠ Pitfall: Inconsistent Time Handling Different components using different time zones or timestamp formats can cause data correlation issues. Always use UTC internally and only convert to local time zones for user display. Include timezone information in all external APIs.

Error Handling and Edge Cases

Milestone(s): All milestones (comprehensive error handling is critical throughout the entire metrics and alerting system, from ingestion through visualization to notification delivery)

The Defense-in-Depth Metaphor: Building Resilient Systems

Think of error handling in a metrics system like a medieval castle's defense system. A well-designed castle doesn't rely on a single wall to protect against attack—it employs multiple layers of defense working together. The outer moat catches obvious threats, the walls deflect serious attacks, the inner courtyard provides fallback positions, and the keep serves as the final refuge. Each layer has different responsibilities and failure modes, but together they create a robust defense against various types of attacks.

Similarly, our metrics and alerting system employs defense-in-depth error handling. Input validation acts as the outer moat, catching malformed data before it enters the system. Rate limiting and backpressure serve as the castle walls, protecting against overwhelming traffic. Graceful degradation provides fallback positions when components fail, and persistent state recovery acts as the keep—ensuring the system can rebuild itself even after catastrophic failure.

The key insight is that errors are not exceptional events to be avoided, but predictable phenomena to be managed systematically. Just as castle architects studied siege warfare to design appropriate defenses, we must study the failure modes of distributed

systems to build appropriate error handling mechanisms.

Decision: Comprehensive Error Taxonomy

- **Context:** Metrics systems face numerous failure modes across ingestion, storage, querying, and alerting
- **Options Considered:**
 1. Handle errors locally in each component
 2. Central error handling service
 3. Layered error handling with component-specific recovery
- **Decision:** Layered error handling with component-specific recovery strategies
- **Rationale:** Each component has unique failure modes and recovery requirements that demand specialized handling
- **Consequences:** More complex implementation but significantly better system resilience and observability

Error Classification Framework

Our error handling strategy categorizes failures into distinct classes, each requiring different detection, reporting, and recovery mechanisms. This classification helps us build appropriate responses rather than treating all errors the same way.

Error Class	Characteristics	Detection Method	Recovery Strategy	User Impact
Transient Network	Temporary connectivity issues, timeouts	Request timeout, connection refused	Retry with exponential backoff	Brief delays in data availability
Resource Exhaustion	Memory, disk, file handles depleted	System monitoring, allocation failures	Backpressure, graceful degradation	Reduced throughput, some data loss
Data Corruption	Invalid data format, checksum mismatches	Validation failures, parsing errors	Skip corrupt data, request resend	Missing data points in time series
Configuration Errors	Invalid settings, missing required fields	Startup validation, runtime checks	Fail-fast with clear error messages	Service unavailable until fixed
Component Failures	Process crashes, hardware failures	Health checks, heartbeat monitoring	Restart, failover to backup	Service interruption, potential data loss
Cascading Failures	One failure triggers others	Dependency monitoring, circuit breakers	Isolate failures, prevent propagation	System-wide degradation

The critical principle here is that different error classes require fundamentally different handling strategies. Retrying a configuration error will never succeed, while giving up immediately on a transient network error wastes an opportunity for automatic recovery.

Ingestion Error Handling

The metrics ingestion engine faces several distinct failure modes that require specialized handling approaches. The ingestion pipeline must balance data durability with system availability, making intelligent decisions about when to accept, reject, or defer incoming metrics.

Mental Model: The Quality Control Checkpoint

Think of metrics ingestion error handling like a quality control checkpoint at a manufacturing plant. Raw materials (metrics) arrive continuously from suppliers (client applications), and the checkpoint must quickly assess each batch for quality and safety. Defective materials are rejected with clear feedback to the supplier. When the production line becomes overwhelmed, the

checkpoint implements controlled throttling rather than shutting down completely. Critical defects that could damage downstream equipment trigger immediate safety protocols.

This quality control metaphor captures the key insight that ingestion error handling is about maintaining system integrity while providing useful feedback to metric producers.

Invalid Metrics Processing

The ingestion engine encounters various forms of invalid metric data that must be handled without disrupting overall system operation. Each validation failure type requires specific handling to provide meaningful feedback while protecting the system from malformed data.

Validation Failure	Detection Method	Error Response	Recovery Action	Prevention Strategy
Invalid Metric Name	Regex pattern matching	HTTP 400 with field details	Reject metric, continue processing batch	Client-side validation libraries
Label Cardinality Explosion	Label combination counting	HTTP 429 with cardinality info	Reject high-cardinality metrics	Cardinality monitoring and alerts
Timestamp Out of Range	Time bounds checking	HTTP 400 with acceptable range	Reject samples, accept others	Clock synchronization monitoring
Invalid Sample Values	Numeric validation	HTTP 400 with value constraints	Skip invalid samples	Input sanitization at source
Missing Required Fields	Schema validation	HTTP 422 with missing fields	Reject entire metric	Schema documentation and examples
Malformed JSON/Protocol	Parsing failure	HTTP 400 with parse error	Reject request, maintain connection	Protocol version negotiation

The `MetricsIngestor` component implements a multi-stage validation pipeline that processes metrics through increasingly sophisticated validation checks. Early-stage validation catches obvious format errors quickly, while later stages perform more expensive operations like cardinality analysis and duplicate detection.

Decision: Batch-Level vs Individual Metric Error Handling

- **Context:** When processing batches of metrics, individual metrics may fail validation while others succeed
- **Options Considered:**
 1. Fail entire batch if any metric is invalid
 2. Process all valid metrics, report errors for invalid ones
 3. Fail fast on first error
- **Decision:** Process valid metrics, report detailed errors for invalid ones
- **Rationale:** Maximizes data availability while providing actionable error feedback
- **Consequences:** More complex error tracking but better system resilience

The validation pipeline maintains detailed error statistics that help identify systematic problems in metric production. These statistics are exposed through the health monitoring system and can trigger alerts when error rates exceed acceptable thresholds.

Network Failures and Connectivity Issues

Network-related failures in metrics ingestion require careful handling to balance data durability with system responsiveness. The ingestion engine must distinguish between temporary connectivity issues that warrant retry attempts and persistent failures that

indicate configuration or infrastructure problems.

The ingestion system implements a sophisticated backpressure mechanism that dynamically adjusts acceptance rates based on downstream capacity. When the storage engine becomes overwhelmed, the ingestion engine gradually increases response latencies and begins rejecting new requests with HTTP 503 status codes that include retry-after headers.

Failure Type	Symptom	Detection	Response	Recovery
Client Timeout	Connection drops during request	Request context cancellation	Return partial success status	Client implements retry logic
DNS Resolution	Cannot resolve ingestion endpoint	DNS lookup failure	Return service unavailable	Verify DNS configuration
TLS Handshake	Certificate validation fails	TLS negotiation error	Return TLS error with details	Check certificate validity
TCP Connection	Network unreachable	Connection refused/timeout	Return connection error	Verify network connectivity
HTTP Parsing	Malformed HTTP request	Parse error during request processing	Return HTTP 400	Validate client HTTP implementation
Request Size	Payload exceeds limits	Content-length or memory checks	Return HTTP 413	Implement client-side batching

The ingestion engine maintains circuit breakers for downstream dependencies like the storage engine and validation services. When failure rates exceed configurable thresholds, the circuit breaker opens and begins failing requests immediately rather than waiting for timeouts. This prevents cascade failures and provides faster feedback to clients.

Backpressure and Flow Control

When the metrics ingestion rate exceeds the system's processing capacity, the ingestion engine must implement intelligent backpressure mechanisms that maintain system stability while maximizing throughput. The backpressure system operates on multiple levels, from individual request handling to system-wide flow control.

The ingestion engine monitors several key metrics to detect capacity constraints:

- Storage engine write latency and error rates
- Memory usage in validation and processing pipelines
- CPU utilization in metric parsing and validation
- Network buffer utilization for incoming connections
- Queue depths in asynchronous processing stages

When backpressure conditions are detected, the system implements graduated responses:

1. **Initial Response:** Increase response latencies slightly to encourage client-side rate limiting
2. **Moderate Pressure:** Begin rejecting lowest-priority metrics while accepting high-priority ones
3. **High Pressure:** Return HTTP 503 responses with retry-after headers to implement explicit backoff
4. **Critical Pressure:** Temporarily refuse new connections while processing existing request backlog
5. **Emergency Mode:** Activate emergency shedding of non-critical metrics to prevent system collapse

The backpressure system maintains fairness by implementing per-client rate limiting that prevents any single metric producer from overwhelming the system. Clients that consistently produce high-quality, well-formatted metrics receive higher priority during backpressure conditions.

Pitfall: Ignoring Cardinality Explosion During High Load Many implementations focus on request rate limiting but ignore cardinality validation during high load conditions. This can lead to memory exhaustion as the system accepts high-cardinality metrics that consume excessive indexing resources. Always validate cardinality even when implementing backpressure—reject high-cardinality metrics first during resource constraints.

Storage Error Handling

The time-series storage engine faces unique challenges related to data persistence, consistency, and recovery from various failure modes. Storage errors can have long-lasting impact on system reliability and data availability, requiring sophisticated detection and recovery mechanisms.

Mental Model: The Bank Vault System

Think of storage error handling like a bank's vault security system. The vault has multiple layers of protection: tamper-evident seals detect unauthorized access, environmental sensors monitor temperature and humidity, backup power systems ensure continuous operation, and detailed audit logs track every access. When problems are detected, the system has specific protocols for different scenarios—some trigger immediate lockdown, others activate backup systems, and some simply log the event for later investigation.

Similarly, our storage engine employs multiple protective mechanisms. Write-ahead logging serves as the tamper-evident seal, ensuring we can detect and recover from interrupted operations. Health monitoring acts as environmental sensors, detecting degraded performance before complete failure. Backup and replication provide continuity during hardware failures. And comprehensive error logging enables forensic analysis of storage problems.

The key insight is that storage systems must be paranoid about data integrity while remaining operational during various failure conditions. Like a bank vault, the storage engine must balance security (data integrity) with accessibility (query performance).

Disk Failures and Hardware Issues

Hardware-related storage failures require immediate detection and graceful degradation to prevent data loss while maintaining system availability. The storage engine implements comprehensive monitoring of disk health, filesystem status, and I/O performance to detect impending failures before they cause data corruption.

Hardware Failure	Early Warning Signs	Detection Method	Immediate Response	Recovery Procedure
Disk Bad Sectors	Increasing read/write latency	SMART monitoring, I/O error tracking	Mark affected blocks read-only	Copy data to healthy storage, mark disk for replacement
Filesystem Corruption	Checksum validation failures	File integrity verification	Switch to read-only mode	Run filesystem repair, restore from backup if needed
Out of Disk Space	Free space below threshold	Disk usage monitoring	Activate emergency compaction	Delete oldest data per retention policy
I/O Subsystem Failure	All disk operations failing	System call error detection	Activate read-only mode	Restart storage engine with backup configuration
Memory Corruption	Unexpected data patterns	Data validation checks	Isolate corrupt memory regions	Restart process, reload data from persistent storage
Network Partition	Cannot reach replica nodes	Network connectivity monitoring	Continue serving local data	Implement split-brain prevention protocols

The storage engine maintains multiple redundancy mechanisms to handle hardware failures gracefully. The write-ahead log provides immediate durability for recent writes, while periodic snapshots enable recovery of the complete dataset. Block-based

storage with checksums detects corruption early, and automatic compaction can often recover from minor filesystem issues.

When hardware failures are detected, the storage engine transitions through a series of degraded operational modes:

1. **Normal Operation:** All read and write operations functioning normally
2. **Degraded Writes:** Read operations continue, writes queued to WAL only
3. **Read-Only Mode:** Serve existing data, reject all write operations
4. **Emergency Mode:** Serve cached data only, prepare for shutdown
5. **Offline Mode:** Reject all operations, preserve data integrity for recovery

Data Corruption Detection and Recovery

Data corruption in time-series storage can occur at multiple levels, from individual sample values to entire storage blocks. The storage engine implements comprehensive corruption detection and recovery mechanisms that operate continuously during normal operations.

The storage engine uses a multi-layered approach to corruption detection:

Block-Level Integrity: Each storage block maintains cryptographic checksums that are validated on every read operation. When checksum mismatches are detected, the system immediately marks the block as corrupt and attempts to recover data from replicas or backup copies.

Series-Level Validation: The storage engine periodically validates the consistency of time-series data, checking for impossible timestamp sequences, duplicate samples, and value ranges that violate metric type constraints. These validation passes run during low-traffic periods to minimize performance impact.

Cross-Reference Validation: The storage engine maintains multiple indexes for the same data and periodically cross-validates these indexes to detect inconsistencies that might indicate memory corruption or software bugs.

Corruption Type	Detection Method	Recovery Strategy	Prevention Measure
Block Checksum Mismatch	Read-time validation	Restore from replica or backup	Regular checksum verification
Invalid Timestamps	Series consistency checks	Remove invalid samples	Timestamp validation on write
Duplicate Samples	Index consistency validation	Deduplicate during compaction	Write-time duplicate detection
Missing Index Entries	Cross-reference validation	Rebuild index from raw data	Atomic index updates
Truncated Files	File size validation	Restore from backup or WAL	Atomic file operations
Invalid Sample Values	Range validation checks	Mark samples as invalid	Input validation pipeline

The recovery process for detected corruption follows a systematic approach:

1. **Isolation:** Immediately mark corrupted data as unavailable for queries
2. **Assessment:** Determine the extent of corruption and available recovery options
3. **Recovery:** Restore data from the most recent clean backup or replica
4. **Validation:** Verify the recovered data passes all integrity checks
5. **Integration:** Gradually reintroduce recovered data into normal operations
6. **Analysis:** Investigate root cause to prevent similar corruption

⚠ Pitfall: Assuming Filesystem Reliability Many storage implementations assume the underlying filesystem provides perfect data integrity. However, filesystems can experience silent corruption, especially under high write loads or during power failures. Always implement application-level checksums and never rely solely on filesystem guarantees for data integrity.

Storage Resource Management

The storage engine must carefully manage various system resources to prevent resource exhaustion while maintaining optimal performance. Resource management errors can cascade through the system, causing failures in other components that depend on storage availability.

Memory management in the storage engine involves multiple pools with different lifecycle characteristics. The ingestion buffer pool handles short-lived allocations for incoming metric samples. The query result pool manages medium-lived allocations for query processing. The index cache pool maintains long-lived allocations for frequently accessed index data.

File descriptor management becomes critical in systems handling thousands of time series, as each active series may require multiple file handles for data blocks, indexes, and WAL segments. The storage engine implements sophisticated file handle pooling and lazy loading to stay within system limits while maintaining performance.

Resource Type	Monitoring Method	Warning Threshold	Critical Response	Prevention Strategy
Memory Usage	Process memory monitoring	80% of available RAM	Activate memory pressure relief	Implement LRU caching with configurable limits
File Descriptors	Open file handle counting	90% of ulimit	Close least-recently-used files	File handle pooling and sharing
Disk I/O Bandwidth	I/O utilization tracking	90% of measured capacity	Throttle non-critical operations	Separate high/low priority queues
Network Connections	Active connection monitoring	Connection pool 90% full	Reject new connections	Connection pooling and reuse
CPU Usage	Process CPU monitoring	95% sustained usage	Reduce background tasks	Asynchronous processing pipelines
Temporary Disk Space	Temp directory monitoring	95% of available space	Purge temporary files	Aggressive temporary file cleanup

The storage engine implements resource quotas that prevent any single operation from consuming excessive resources. Query operations have memory limits that prevent runaway aggregations from exhausting system memory. Compaction operations have I/O bandwidth limits that prevent them from interfering with real-time ingestion and queries.

Alerting Error Handling

The alerting system must maintain high reliability while handling various failure modes that could prevent critical notifications from reaching their intended recipients. Alerting failures can have severe operational consequences, making robust error handling essential for production deployments.

Mental Model: The Emergency Communication Network

Think of alerting error handling like an emergency communication network during a natural disaster. The network has multiple communication channels (radio, satellite, cellular), backup power systems, redundant message routing, and detailed logging of all communication attempts. When the primary communication channel fails, the system automatically switches to backup channels. When messages can't be delivered immediately, they're queued for retry with escalating urgency. Critical messages are sent through multiple channels simultaneously to ensure delivery.

This emergency network metaphor captures the essential characteristics of reliable alerting systems: redundancy, persistence, escalation, and comprehensive audit trails. Just as emergency responders need absolute confidence that their communications will reach recipients, operations teams need confidence that critical alerts will be delivered even during system failures.

Notification Delivery Failures

Notification delivery failures represent some of the most critical errors in the metrics system, as they can prevent teams from responding to production incidents. The alerting system must implement sophisticated retry logic, escalation policies, and fallback mechanisms to ensure critical notifications reach their intended recipients.

The notification delivery system categorizes failures into distinct classes that require different handling approaches:

Failure Class	Examples	Retry Strategy	Escalation Policy	Fallback Action
Transient Network	DNS timeouts, connection refused	Exponential backoff, max 5 retries	Try alternate endpoints after 3 failures	Switch to backup notification channel
Authentication	Invalid API keys, expired tokens	No retry, immediate escalation	Alert configuration team	Use emergency webhook endpoint
Rate Limiting	API quota exceeded, throttling	Respect rate limits, queue messages	Spread notifications across time	Batch multiple alerts into single message
Recipient Invalid	Email bounces, invalid Slack channel	No retry, log error	Remove invalid recipient from future notifications	Send to backup recipient list
Message Format	Invalid JSON, template errors	Fix format if possible, otherwise skip	Alert template maintainers	Use plain text fallback message
Service Unavailable	Slack/PagerDuty outage	Extended retry with circuit breaker	Try all configured channels	Store notifications for later delivery

The notification delivery engine maintains detailed logs of every delivery attempt, including timestamps, recipient information, delivery status, error details, and retry attempts. These logs are essential for troubleshooting notification issues and proving compliance with SLA requirements.

Decision: Notification Persistence Strategy

- **Context:** Notifications may fail delivery due to various transient issues, but critical alerts must eventually reach recipients
- **Options Considered:**
 1. Best-effort delivery with no persistence
 2. In-memory queues with limited retry
 3. Persistent notification queue with guaranteed delivery
- **Decision:** Persistent notification queue with configurable retry policies
- **Rationale:** Critical alerts justify the complexity of persistent delivery guarantees
- **Consequences:** Increased system complexity but much higher notification reliability

The notification persistence system stores failed notifications in a durable queue that survives system restarts and network outages. Each notification includes metadata about retry attempts, delivery windows, and escalation policies. The system implements intelligent retry scheduling that backs off exponentially while respecting recipient preferences for notification frequency.

Alert Flapping Prevention

Alert flapping—rapid transitions between firing and resolved states—can overwhelm notification channels and desensitize operations teams to legitimate alerts. The alerting system implements sophisticated flapping detection and suppression mechanisms that maintain alert responsiveness while preventing notification storms.

The flapping detection algorithm analyzes alert state transition patterns over configurable time windows. It considers factors like transition frequency, state duration, and the underlying metric volatility to distinguish between legitimate alerts and flapping conditions.

Flapping Pattern	Detection Criteria	Suppression Action	Resolution Strategy
Rapid Fire/Resolve	>5 transitions in 10 minutes	Extend evaluation duration	Increase alert threshold hysteresis
Threshold Bouncing	Value oscillates around threshold	Apply smoothing to evaluation	Use moving averages instead of instant values
Intermittent Failures	Alternating success/failure states	Require sustained failure condition	Implement "consecutive failures" logic
Metric Spikes	Brief value spikes trigger alerts	Ignore single-sample anomalies	Require multiple samples above threshold
Clock Skew Issues	Timestamp inconsistencies cause false alerts	Detect and correct timestamp drift	Implement timestamp validation and adjustment
Configuration Errors	Overly sensitive thresholds	Temporarily disable problematic rules	Alert rule configuration validation

The alerting system implements configurable hysteresis for threshold-based alerts, requiring different thresholds for firing and resolving conditions. For example, a CPU usage alert might fire when usage exceeds 90% but only resolve when usage drops below 85%. This prevents flapping when metric values oscillate around the threshold.

The flapping prevention system maintains state for each alert rule, tracking recent transition history and applying appropriate suppression policies. When flapping is detected, the system can take several actions:

1. **Temporary Suppression:** Silence notifications while allowing the alert to continue evaluation
2. **Extended Evaluation:** Increase the duration requirement before firing alerts
3. **Threshold Adjustment:** Temporarily modify thresholds to reduce sensitivity
4. **Aggregation Mode:** Switch to evaluating aggregated metrics instead of instant values
5. **Maintenance Mode:** Mark the alert rule for manual review and adjustment

⚠ Pitfall: Over-Aggressive Flapping Suppression Implementing flapping suppression that's too aggressive can prevent legitimate alerts from firing during rapidly changing conditions. Always maintain escape hatches that allow critical alerts to bypass flapping suppression, and regularly review suppressed alerts to ensure they represent genuine flapping rather than system issues.

Silence Management and Maintenance Windows

Operations teams need the ability to temporarily suppress notifications during planned maintenance windows or when investigating known issues. The alerting system implements sophisticated silence management that prevents unwanted notifications while maintaining audit trails and preventing accidentally permanent suppressions.

The silence management system supports multiple types of suppressions:

Time-Based Silences: Suppress alerts for specific time periods, commonly used for maintenance windows. These silences have definite start and end times and automatically expire to prevent forgotten suppressions.

Label-Based Silences: Suppress alerts matching specific label combinations, useful for suppressing alerts from problematic hosts or services during investigation.

Alert-Specific Silences: Suppress specific alert rules entirely, typically used for rules that are being debugged or reconfigured.

Escalation Silences: Allow alerts to fire but suppress specific notification channels, useful for reducing noise during incident response.

Silence Type	Configuration	Audit Requirements	Expiration Policy
Maintenance Window	Start/end time, affected labels	Who created, reason, affected systems	Automatic expiration at end time
Investigation Silence	Label matchers, duration	Incident ticket reference	Manual renewal required after 24 hours
Rule Debugging	Specific rule ID, duration	Configuration change justification	Maximum 1 week duration
Channel Suppression	Channel and alert combinations	Escalation acknowledgment	Automatic expiration after incident resolution
Emergency Silence	Broad label matchers, short duration	Incident commander authorization	Maximum 4 hours, requires manual renewal

The silence management system maintains comprehensive audit logs that track silence creation, modification, and expiration. These logs include the identity of the user creating the silence, the justification for the suppression, and the expected impact on alerting coverage.

All silences require explicit expiration times to prevent accidentally permanent suppressions. The system sends notifications to silence creators before silences expire, allowing them to extend or modify the suppression if needed. Critical silences that broadly suppress alerts require additional approval workflows and shorter maximum durations.

The alerting system provides visibility into active silences through the dashboard interface, showing which alerts are currently suppressed and why. This visibility helps prevent situations where important alerts are inadvertently suppressed during critical incidents.

Common Pitfalls in Error Handling

Understanding common error handling mistakes helps avoid subtle bugs that can compromise system reliability. These pitfalls represent patterns frequently encountered when building production metrics systems.

⚠ Pitfall: Cascading Failure Amplification When one component fails, poorly designed error handling can amplify the failure across the entire system. For example, if the storage engine becomes slow, aggressive retry logic in the ingestion engine can make the problem worse by increasing load. Always implement circuit breakers and exponential backoff to prevent failure amplification.

⚠ Pitfall: Silent Data Loss During Errors Systems sometimes silently drop data during error conditions without proper logging or alerting. This is particularly dangerous in metrics systems where missing data points can hide real problems. Always log data loss events and implement monitoring for data ingestion rates.

⚠ Pitfall: Inconsistent Error Response Formats Different components returning errors in different formats makes client error handling difficult and unreliable. Standardize error response formats across all API endpoints and include sufficient detail for programmatic handling.

⚠ Pitfall: Inadequate Error Context Error messages that lack context make troubleshooting difficult and time-consuming. Always include relevant identifiers (metric names, timestamps, client IDs) in error messages to enable efficient debugging.

⚠ Pitfall: Blocking Operations During Error Handling Error handling code that performs blocking operations can cause the entire system to become unresponsive during failure conditions. Use timeouts, asynchronous processing, and non-blocking I/O in all error handling paths.

Implementation Guidance

This implementation guidance provides practical code structure and patterns for implementing comprehensive error handling throughout the metrics and alerting system.

Technology Recommendations

Component	Simple Option	Advanced Option
Error Logging	Standard library logging with structured fields	Structured logging with correlation IDs (slog)
Health Monitoring	HTTP endpoint with JSON status	Prometheus metrics with detailed health checks
Circuit Breakers	Simple failure counting with timeouts	Hystrix-style circuit breakers with metrics
Retry Logic	Exponential backoff with jitter	Sophisticated retry policies with circuit breakers
Error Persistence	In-memory error queues	Persistent error storage with WAL
Monitoring Integration	Basic metrics collection	Full observability with tracing and metrics

File Structure for Error Handling

```
internal/
  └── errors/
    ├── types.go          ← error type definitions and interfaces
    ├── validation.go     ← input validation errors and handling
    ├── storage.go        ← storage-specific error types
    ├── network.go        ← network and transport errors
    └── recovery.go       ← error recovery and retry logic
  └── health/
    ├── manager.go        ← health check coordination
    ├── checks.go         ← individual health check implementations
    └── status.go         ← health status aggregation
  └── circuit/
    ├── breaker.go        ← circuit breaker implementation
    └── metrics.go         ← circuit breaker metrics and monitoring
  └── retry/
    ├── policy.go          ← retry policy definitions
    ├── backoff.go         ← backoff algorithms
    └── queue.go           ← persistent retry queue
```

Core Error Type Definitions

```
package errors

import (
    "context"
    "time"
    "fmt"
)

// ValidationErrors represents a collection of validation failures

type ValidationErrors struct {
    Errors []ValidationErrors `json:"errors"`
}

// ValidationError represents a single field validation failure

type ValidationError struct {
    Field   string `json:"field"`
    Value   string `json:"value"`
    Message string `json:"message"`
}

// Add appends a new validation error to the collection

func (ve *ValidationErrors) Add(field, value, message string) {
    // TODO 1: Create new ValidationError with provided parameters
    // TODO 2: Append to the Errors slice
    // TODO 3: Ensure thread safety if called concurrently
}

// ToJSON converts validation errors to JSON for API responses

func (ve *ValidationErrors) ToJSON() ([]byte, error) {
    // TODO 1: Use encoding/json to marshal the struct
    // TODO 2: Handle marshaling errors appropriately
    // TODO 3: Return formatted JSON bytes
}
```

GO

```
// StorageError represents errors from the storage engine

type StorageError struct {

    Operation string      `json:"operation"`

    Cause     string      `json:"cause"`

    Timestamp time.Time   `json:"timestamp"`

    Retryable bool        `json:"retryable"`

}

// Error implements the error interface

func (se *StorageError) Error() string {

    return fmt.Sprintf("storage %s failed: %s", se.Operation, se.Cause)

}

// NetworkError represents network-related failures

type NetworkError struct {

    Endpoint  string      `json:"endpoint"`

    Operation string      `json:"operation"`

    Timeout   time.Duration `json:"timeout"`

    Attempt   int          `json:"attempt"`

}

// Error implements the error interface

func (ne *NetworkError) Error() string {

    return fmt.Sprintf("network %s to %s failed on attempt %d", ne.Operation, ne.Endpoint, ne.Attempt)

}
```

Health Management Implementation

```
package health

import (
    "context"
    "sync"
    "time"
    "log/slog"
)

// HealthManager coordinates health checks across system components

type HealthManager struct {

    checks    map[string]*HealthCheck
    mu        sync.RWMutex
    logger    *slog.Logger
    interval  time.Duration
    stopCh    chan struct{}
}

// HealthCheck represents a single component health check

type HealthCheck struct {

    Name      string      `json:"name"`
    Status    HealthStatus `json:"status"`
    LastChecked time.Time  `json:"last_checked"`
    Message   string      `json:"message"`
    CheckFunc func(context.Context) error
}

// NewHealthManager creates a new health check coordinator

func NewHealthManager(logger *slog.Logger, interval time.Duration) *HealthManager {
    // TODO 1: Initialize HealthManager struct with provided parameters
    // TODO 2: Create empty checks map
    // TODO 3: Create stop channel for graceful shutdown
    // TODO 4: Return configured manager instance
}
```

GO

```
}

// RegisterCheck adds a health check function for a component

func (hm *HealthManager) RegisterCheck(name string, checkFunc func(context.Context) error) {

    // TODO 1: Acquire write lock for thread safety

    // TODO 2: Create HealthCheck struct with name and function

    // TODO 3: Initialize status as unknown and set initial timestamp

    // TODO 4: Add to checks map

    // TODO 5: Log registration event

}

// RunChecks executes all registered health checks

func (hm *HealthManager) RunChecks(ctx context.Context) {

    // TODO 1: Acquire read lock to get copy of checks

    // TODO 2: Iterate through all registered checks

    // TODO 3: Execute each check function with timeout

    // TODO 4: Update check status based on result

    // TODO 5: Log any status changes

    // TODO 6: Handle context cancellation gracefully

}

// GetOverallStatus returns the worst status among all checks

func (hm *HealthManager) GetOverallStatus() HealthStatus {

    // TODO 1: Acquire read lock

    // TODO 2: Initialize status as Healthy

    // TODO 3: Iterate through all checks

    // TODO 4: Find the worst status (Unhealthy > Degraded > Healthy)

    // TODO 5: Return overall system status

}
```

Circuit Breaker Implementation

```
package circuit

import (
    "context"
    "sync"
    "time"
    "errors"
)

// Breaker implements circuit breaker pattern for fault tolerance

type Breaker struct {

    name          string
    maxFailures   int
    resetTimeout  time.Duration
    state         BreakerState
    failures      int
    lastFailure   time.Time
    mu            sync.RWMutex
    onStateChange func(string, BreakerState, BreakerState)
}

// BreakerState represents circuit breaker states

type BreakerState int

const (
    StateClosed BreakerState = iota
    StateOpen
    StateHalfOpen
)

// Execute runs the provided function through the circuit breaker

func (cb *Breaker) Execute(ctx context.Context, operation func(context.Context) error) error {
    // TODO 1: Check current circuit breaker state
}
```

GO

```
// TODO 2: If open, check if reset timeout has elapsed

// TODO 3: If closed or half-open, attempt to execute operation

// TODO 4: Handle operation success (reset failure count)

// TODO 5: Handle operation failure (increment count, possibly trip breaker)

// TODO 6: Update state and notify state change listeners

// TODO 7: Return appropriate error based on state and operation result

}

// tripBreaker changes state to open and records the failure time

func (cb *Breaker) tripBreaker() {

    // TODO 1: Acquire write lock

    // TODO 2: Change state to StateOpen

    // TODO 3: Record current time as lastFailure

    // TODO 4: Call state change callback if registered

}

// resetBreaker changes state to closed and clears failure count

func (cb *Breaker) resetBreaker() {

    // TODO 1: Acquire write lock

    // TODO 2: Change state to StateClosed

    // TODO 3: Reset failures count to zero

    // TODO 4: Call state change callback if registered

}
```

Retry Policy Implementation

```
package retry

import (
    "context"
    "time"
    "math"
    "math/rand"
)

// Policy defines retry behavior for failed operations

type Policy struct {
    MaxAttempts      int
    InitialDelay    time.Duration
    MaxDelay        time.Duration
    Multiplier      float64
    Jitter          bool
    RetryableErrors []error
}

// Execute attempts an operation with retry logic

func (p *Policy) Execute(ctx context.Context, operation func(context.Context) error) error {
    // TODO 1: Initialize attempt counter and delay

    // TODO 2: Loop until max attempts reached or context cancelled

    // TODO 3: Execute operation and check result

    // TODO 4: If successful, return nil

    // TODO 5: If error not retryable, return immediately

    // TODO 6: Calculate next delay with exponential backoff

    // TODO 7: Apply jitter if configured

    // TODO 8: Wait for delay duration or context cancellation

    // TODO 9: Continue to next attempt
}

// calculateDelay computes the next retry delay with exponential backoff
```

GO

```

func (p *Policy) calculateDelay(attempt int) time.Duration {
    // TODO 1: Calculate exponential delay: initialDelay * (multiplier ^ attempt)
    // TODO 2: Apply maximum delay limit
    // TODO 3: Add jitter if enabled (±25% random variation)
    // TODO 4: Return final delay duration
}

// isRetryable checks if an error should trigger a retry attempt

func (p *Policy) isRetryable(err error) bool {
    // TODO 1: Check if error is nil (should not retry success)
    // TODO 2: If no retryable errors configured, retry all errors
    // TODO 3: Iterate through configured retryable error types
    // TODO 4: Use errors.Is() to match error types
    // TODO 5: Return true if error matches any retryable type
}

```

Milestone Checkpoints

After Implementing Error Handling Infrastructure:

1. Run `go test ./internal/errors/... -v` to verify error type handling
2. Test health checks with `curl http://localhost:8080/health` - should return JSON with component statuses
3. Verify circuit breaker functionality by simulating downstream failures
4. Confirm retry policies work by testing with flaky network connections
5. Check error logging appears in structured format with appropriate detail levels

Signs of Proper Error Handling:

- Error responses include actionable information for clients
- System remains responsive during various failure conditions
- Health checks accurately reflect component status
- Circuit breakers prevent cascade failures during outages
- Retry logic recovers from transient failures automatically

Common Issues to Debug:

- Errors returning without sufficient context for troubleshooting
- Circuit breakers tripping too frequently due to overly sensitive thresholds
- Retry logic causing thundering herd problems during recovery
- Health checks reporting false positives or negatives
- Error handling blocking normal operations during failure conditions

Testing Strategy

Milestone(s): All milestones (comprehensive testing is essential throughout the entire metrics and alerting system, from initial metric ingestion through storage, querying, visualization, and alerting capabilities)

The Quality Assurance Laboratory: Understanding Testing Approach

Think of our testing strategy as operating a quality assurance laboratory for a pharmaceutical company. Just as a pharmaceutical lab has multiple testing phases—from testing individual chemical compounds in isolation, to testing drug interactions, to full clinical trials with real patients—our metrics system requires layered testing that validates everything from individual component behavior to complete end-to-end workflows. Each testing layer serves a specific purpose and catches different categories of problems, much like how pharmaceutical testing catches different types of safety and efficacy issues at each stage.

The fundamental challenge in testing a metrics and alerting system lies in the inherently time-dependent and stateful nature of the system. Unlike testing a simple web API where requests are largely independent, our system maintains complex state across time-series data, alert conditions, and dashboard subscriptions. We must validate not just that individual operations work correctly, but that the system maintains consistency and correctness over time, handles various failure scenarios gracefully, and performs adequately under realistic load conditions.

Our testing strategy employs three complementary layers that work together to provide comprehensive validation coverage. Unit testing validates individual component behavior in isolation, integration testing verifies component interactions and data flows, and milestone validation checkpoints ensure that each major system capability works correctly in realistic scenarios. This layered approach allows us to catch bugs early in development while also validating that the complete system meets its functional and non-functional requirements.

Unit Testing Strategy

Unit testing forms the foundation of our testing pyramid, focusing on validating individual components in complete isolation from their dependencies. The primary goal is to verify that each component correctly implements its specified behavior under all possible input conditions, including edge cases and error scenarios that might be difficult to reproduce in integration testing.

Component Isolation and Mocking Strategy

Our unit testing approach treats each component as a black box with clearly defined inputs and outputs. We use dependency injection and interface-based design to isolate components from their dependencies, allowing us to substitute mock implementations that provide predictable behavior during testing.

Decision: Interface-Based Mocking

- **Context:** Need to test components in isolation without external dependencies like storage, network, or file system
- **Options Considered:** Direct mocking libraries, interface-based mocks, test doubles with recording capabilities
- **Decision:** Interface-based mocks with behavior verification
- **Rationale:** Provides type safety, clear contracts, and ability to verify interaction patterns
- **Consequences:** Requires interfaces for all major dependencies but provides robust isolation and clear test intent

The following table outlines our mocking strategy for major component dependencies:

Component Under Test	Mock Dependencies	Mock Behavior	Verification Points
MetricsIngestor	TimeSeriesStorage, Validator, CardinalityManager	Return predictable responses for storage operations	Verify correct validation sequence, storage calls, error propagation
QueryProcessor	TimeSeriesStorage, QueryCache	Simulate various data availability scenarios	Verify query optimization, cache utilization, result formatting
AlertEvaluator	QueryProcessor, StateManager, NotificationManager	Control metric values and state transitions	Verify rule evaluation logic, state changes, notification triggers
StorageEngine	File system, network resources	Simulate disk operations and failures	Verify data persistence, compaction logic, error recovery
DashboardServer	QueryProcessor, WebSocket connections	Control query results and client interactions	Verify subscription management, real-time updates, error handling

Core Component Testing Coverage

Each component requires comprehensive testing that covers normal operation, boundary conditions, error scenarios, and concurrent access patterns. The testing approach varies based on the component's responsibility and complexity.

Metrics Ingestion Testing

The `MetricsIngestor` component requires extensive testing of its validation and processing pipeline. We test each metric type individually to ensure proper handling of counters, gauges, and histograms. Critical test scenarios include:

- Metric Type Validation:** Verify that each metric type is processed according to its semantics—counters must be monotonically increasing, gauges can have arbitrary values, and histograms must include proper bucket definitions.
- Label Validation and Cardinality Control:** Test label parsing, normalization, and cardinality explosion prevention. This includes testing the rejection of metrics that would exceed cardinality limits and proper handling of malformed label specifications.
- Batch Processing:** Validate that metric batches are processed atomically and that partial failures in a batch don't corrupt the storage state.
- Backpressure Handling:** Test behavior when the storage layer cannot keep up with ingestion rates, ensuring proper flow control and graceful degradation.

Time-Series Storage Testing

The `StorageEngine` component requires testing of its complex persistence and compaction logic. Key testing areas include:

- Write Path Validation:** Test sample appending through the write-ahead log, ensuring proper durability guarantees and crash recovery capabilities.
- Compaction Logic:** Verify that compaction correctly merges blocks, maintains temporal ordering, and applies retention policies without data loss.
- Query Path Optimization:** Test series lookup, time range queries, and label filtering to ensure correct results and acceptable performance characteristics.
- Concurrent Access:** Validate thread safety during simultaneous read and write operations, ensuring no data corruption or race conditions.

Query Engine Testing

The `QueryProcessor` component requires testing of its parsing, planning, and execution pipeline:

1. **Query Parsing:** Test the lexical analyzer and parser with various query syntaxes, including edge cases like empty queries, malformed expressions, and complex nested functions.
2. **Query Planning:** Verify that the execution planner generates optimal plans for different query patterns and correctly estimates resource requirements.
3. **Aggregation Functions:** Test each aggregation function individually with various input patterns, including sparse data, missing values, and extreme values.
4. **Cache Integration:** Validate cache hit/miss behavior, cache invalidation, and memory management under different query patterns.

Alert Evaluation Testing

The `AlertEvaluator` component requires testing of its state machine and notification logic:

1. **State Transitions:** Test all possible alert state transitions, ensuring proper timing constraints and notification triggers.
2. **Rule Evaluation:** Verify that alert rules are evaluated correctly with various metric patterns, including missing data and counter resets.
3. **Notification Dispatch:** Test the notification queue and channel integration, including retry logic and failure handling.
4. **Timing Accuracy:** Validate evaluation intervals and duration-based conditions are handled with appropriate precision.

Error Condition Testing

Each component must handle various error conditions gracefully, maintaining system stability and providing useful diagnostic information. Our error testing strategy covers the following categories:

Error Category	Test Scenarios	Expected Behavior	Validation Method
Input Validation	Malformed metrics, invalid labels, type mismatches	Reject invalid input with descriptive errors	Assert specific error messages and codes
Resource Exhaustion	Memory limits, disk space, connection limits	Graceful degradation with backpressure	Monitor resource usage and response times
External Dependencies	Storage failures, network timeouts, cache misses	Proper error propagation and retry logic	Inject failures and verify recovery
Concurrent Access	Race conditions, deadlocks, inconsistent reads	Thread-safe operation without data corruption	Concurrent test execution with verification
Configuration Errors	Invalid settings, missing parameters, type errors	Startup validation with clear error messages	Configuration validation testing

Performance and Resource Usage Testing

Unit tests must also validate performance characteristics and resource usage to prevent regressions:

1. **Memory Usage:** Test that components maintain bounded memory usage even with large inputs or extended operation periods.
2. **Processing Latency:** Verify that individual operations complete within acceptable time bounds under various load conditions.
3. **Resource Cleanup:** Ensure that components properly release resources (file handles, connections, goroutines) when stopped or during error conditions.
4. **Scalability Characteristics:** Test behavior with varying data sizes to identify algorithmic complexity issues early.

Common Unit Testing Pitfalls

⚠ Pitfall: Testing Implementation Details Instead of Behavior Many developers write tests that verify internal implementation details rather than observable behavior. For example, testing that a specific internal method was called with particular parameters rather than testing that the component produces the correct output for given inputs. This makes tests brittle and couples them unnecessarily to implementation details.

Why it's wrong: Tests that depend on implementation details break when the implementation changes, even if the behavior remains correct. This reduces confidence in refactoring and makes maintenance more difficult.

How to fix: Focus tests on the public interface and observable behavior. Test inputs and outputs, not internal method calls. Use behavior verification sparingly and only when the interaction pattern is part of the contract.

⚠ Pitfall: Insufficient Error Scenario Coverage Unit tests often focus heavily on happy-path scenarios while neglecting error conditions. This leaves error handling code untested and can lead to poor error messages or improper resource cleanup during failures.

Why it's wrong: Error conditions are often the most critical code paths for system reliability. Untested error handling can lead to data corruption, resource leaks, or cascading failures.

How to fix: Systematically test every error condition that can be triggered. Use dependency injection to simulate failures in external dependencies. Test resource cleanup during both normal and error conditions.

⚠ Pitfall: Non-Deterministic Test Behavior Tests that depend on timing, random values, or external state can produce inconsistent results, making them unreliable for continuous integration and debugging.

Why it's wrong: Flaky tests reduce confidence in the test suite and make it difficult to identify real regressions. Teams often ignore failing tests that are known to be unreliable.

How to fix: Use deterministic inputs and mock time-dependent operations. Control randomness with fixed seeds. Isolate tests from external state and shared resources.

Integration Testing

Integration testing validates the interactions between components and verifies that data flows correctly through the complete system pipeline. Unlike unit tests that use mocks to isolate components, integration tests use real implementations of dependencies to validate that components work correctly together.

End-to-End Workflow Validation

Our integration testing strategy focuses on validating complete workflows that span multiple components. Each workflow represents a critical system capability that must function correctly for the system to provide value to users.

Metric Ingestion to Storage Workflow

This integration test validates the complete path from metric submission through validation, processing, and persistence. The test creates a realistic mix of metric types and verifies that data is correctly stored and retrievable:

1. **Setup Phase:** Initialize a complete storage engine with real file system persistence, configure the ingestion pipeline with actual validators and cardinality managers.
2. **Data Submission:** Submit a variety of metrics including counters with increasing values, gauges with fluctuating values, and histograms with different bucket distributions.
3. **Processing Verification:** Verify that metrics are processed according to their type semantics and that validation rules are properly applied.
4. **Storage Validation:** Query the storage engine directly to verify that samples are persisted with correct timestamps, values, and label associations.

- Compaction Testing:** Allow time for compaction to occur and verify that data remains accessible and correct after compaction operations.

Query Processing to Dashboard Workflow

This integration test validates the complete query execution pipeline from parsing through result delivery to dashboard clients:

- Data Preparation:** Populate the storage engine with time-series data covering multiple metrics with various label combinations and time ranges.
- Query Execution:** Execute queries of varying complexity, including simple metric selection, label filtering, time range specification, and aggregation functions.
- Result Validation:** Verify that query results contain the expected data points with correct values and timestamps.
- Dashboard Integration:** Test the dashboard server's ability to execute queries and deliver results to WebSocket clients with proper formatting.
- Real-Time Updates:** Verify that new data points trigger appropriate updates to subscribed dashboard panels.

Alert Evaluation to Notification Workflow

This integration test validates the complete alerting pipeline from rule evaluation through notification delivery:

- Alert Rule Configuration:** Configure alert rules with various threshold conditions and notification channels.
- Metric Data Injection:** Submit metric data that should trigger alert conditions, including scenarios for alert firing and resolution.
- Evaluation Process:** Verify that the alert evaluator correctly processes rules according to their evaluation intervals.
- State Management:** Validate that alert states transition correctly through the state machine with proper timing constraints.
- Notification Delivery:** Verify that notifications are generated and delivered through configured channels with correct message content.

Cross-Component Data Flow Testing

Integration tests must validate that data maintains consistency and correctness as it flows between components. This includes testing serialization/deserialization, data transformations, and error propagation.

Data Flow Path	Test Scenarios	Validation Points	Expected Behavior
Ingestion → Storage	Metric batches, label normalization, type validation	Data persistence, retrieval accuracy, error handling	Metrics stored exactly as specified with proper indexing
Storage → Query	Time range queries, label filtering, aggregation	Result correctness, performance, partial results	Accurate query results within performance bounds
Query → Dashboard	Real-time updates, subscription management, formatting	Data freshness, update frequency, client synchronization	Timely delivery of formatted data to subscribed clients
Query → Alerting	Rule evaluation, state transitions, threshold checking	Evaluation accuracy, timing precision, state consistency	Correct alert firing and resolution based on metric values
Alerting → Notification	Message formatting, channel routing, delivery confirmation	Template processing, channel selection, retry logic	Reliable notification delivery with proper formatting

Database and File System Integration

The storage engine requires extensive integration testing with the actual file system to validate persistence, durability, and recovery capabilities:

- Crash Recovery Testing:** Simulate system crashes at various points during write operations to verify that the write-ahead log correctly recovers uncommitted data.
- Compaction Integration:** Test compaction operations with real file I/O to verify that data is correctly merged and old files are properly cleaned up.
- Concurrent Access:** Run multiple ingestion and query processes simultaneously to validate file locking and concurrent access patterns.
- Disk Space Management:** Test behavior when disk space is exhausted, including proper error handling and graceful degradation.

Network and Protocol Integration

The system's network interfaces require testing with real HTTP clients and WebSocket connections:

- HTTP API Testing:** Test the metric ingestion API with various HTTP clients, including proper handling of request timeouts, connection limits, and error responses.
- WebSocket Integration:** Test dashboard WebSocket connections with real clients, including connection lifecycle management, message delivery, and error handling.
- Prometheus Integration:** Test compatibility with Prometheus exposition format and scraping protocols.
- Load Balancer Integration:** Test behavior when deployed behind load balancers, including proper session affinity for WebSocket connections.

Configuration and Environment Integration

Integration tests must validate that the system works correctly across different configuration scenarios and deployment environments:

Configuration Area	Test Scenarios	Validation Approach
Storage Configuration	Different retention policies, compaction intervals, data directories	Verify policy enforcement and resource usage
Network Configuration	Various port assignments, timeout settings, connection limits	Test connectivity and performance characteristics
Alert Configuration	Different evaluation intervals, notification channels, message templates	Validate alert behavior and message formatting
Security Configuration	Authentication settings, access controls, encryption options	Test security enforcement and proper access control

Performance Integration Testing

Integration tests must validate system performance under realistic conditions:

- Throughput Testing:** Measure ingestion rates with realistic metric loads and verify that storage can keep up with sustained ingestion.
- Query Performance:** Test query response times with realistic data volumes and concurrent query loads.
- Memory Usage:** Monitor memory consumption during extended operation to identify memory leaks or excessive usage patterns.
- Resource Scaling:** Test system behavior as resource usage approaches configured limits.

The key insight for integration testing is that it validates the system's behavior in realistic operational conditions, catching issues that unit tests cannot detect because they occur only when components interact with real dependencies.

Common Integration Testing Pitfalls

⚠ Pitfall: Insufficient Test Data Diversity Integration tests often use simple, uniform test data that doesn't reflect the complexity and diversity of real-world metric data. This can miss edge cases that only occur with specific data patterns or label combinations.

Why it's wrong: Real-world metrics data has complex patterns including sparse data, high cardinality labels, irregular timestamps, and extreme values. Simple test data doesn't exercise the system under realistic conditions.

How to fix: Use representative test data that includes various metric types, cardinality patterns, and temporal characteristics. Generate test data that reflects actual usage patterns and includes edge cases like missing data points and counter resets.

⚠ Pitfall: Ignoring Resource Cleanup Between Tests Integration tests that don't properly clean up resources between test runs can have interdependencies that make test results non-deterministic or cause resource exhaustion.

Why it's wrong: Resource leaks or shared state between tests makes it difficult to isolate test failures and can cause tests to pass or fail based on execution order rather than actual correctness.

How to fix: Implement thorough cleanup procedures that reset all persistent state, close network connections, and release file handles. Use isolated test environments and verify resource cleanup as part of the test framework.

⚠ Pitfall: Inadequate Error Injection Integration tests often focus on happy-path scenarios and don't adequately test error conditions that can occur when components interact with real dependencies.

Why it's wrong: Many errors only manifest when components interact with real dependencies under stress or failure conditions. These scenarios are critical for system reliability but are difficult to test without proper error injection.

How to fix: Use chaos engineering techniques to inject failures into dependencies. Test network partitions, disk failures, memory pressure, and other realistic failure scenarios. Verify that the system maintains consistency and provides meaningful error messages during failures.

Milestone Validation Checkpoints

Milestone validation checkpoints provide structured verification that each major system capability works correctly before moving to the next development phase. These checkpoints bridge the gap between technical testing and user-visible functionality, ensuring that the system meets its acceptance criteria at each milestone.

Milestone 1: Metrics Collection Validation

The first milestone validation focuses on verifying that the system can correctly ingest and store different types of metrics with proper validation and error handling.

Functional Verification Checkpoints

Validation Area	Test Procedure	Expected Behavior	Success Criteria
Counter Ingestion	Submit counter metrics with increasing values	Values stored with proper monotonic validation	Counter resets detected and handled correctly
Gauge Processing	Submit gauge metrics with fluctuating values	All values stored without validation constraints	Gauge updates reflected immediately in storage
Histogram Handling	Submit histogram metrics with bucket distributions	Bucket counts and sums calculated correctly	Percentile calculations work with stored data
Label Validation	Submit metrics with various label combinations	Labels parsed, normalized, and indexed correctly	Label cardinality limits enforced properly
Prometheus Compatibility	Scrape metrics from Prometheus exposition format	Prometheus metrics ingested correctly	Standard Prometheus metrics work without modification

Performance Validation Checkpoints

The system must demonstrate acceptable performance characteristics under realistic load conditions:

- Ingestion Throughput:** Submit 10,000 metrics per second for 60 seconds and verify that all metrics are processed and stored within acceptable latency bounds (95th percentile < 100ms).
- Memory Usage:** Monitor memory consumption during ingestion and verify that it remains bounded even with sustained high ingestion rates.
- Storage Efficiency:** Verify that stored data uses reasonable disk space and that compression ratios are within expected ranges.
- Cardinality Performance:** Test ingestion performance with varying cardinality levels and verify that high cardinality doesn't cause unacceptable performance degradation.

Error Handling Validation

The system must handle various error conditions gracefully:

- Invalid Metric Rejection:** Submit malformed metrics and verify that they are rejected with appropriate error messages without affecting valid metrics.
- Cardinality Explosion Prevention:** Submit metrics that would exceed cardinality limits and verify that they are rejected with clear error messages.
- Network Failure Recovery:** Simulate network failures during metric ingestion and verify that the system recovers properly when connectivity is restored.
- Disk Space Exhaustion:** Fill the disk space and verify that the system provides clear error messages and doesn't corrupt existing data.

Manual Verification Procedures

- Start the metrics ingestion service with debug logging enabled
- Submit a variety of metrics using curl commands targeting the ingestion API
- Verify that metrics appear in the storage engine using direct storage queries
- Check logs for any error messages or warnings
- Monitor system resource usage during sustained ingestion
- Verify that Prometheus metrics exposition format works by configuring a Prometheus server to scrape the metrics endpoint

Milestone 2: Storage & Querying Validation

The second milestone validation focuses on verifying that the storage engine correctly persists data and that the query engine can retrieve and aggregate data accurately.

Storage Validation Checkpoints

Validation Area	Test Procedure	Expected Behavior	Success Criteria
Time-Series Persistence	Store millions of samples across multiple series	Data persisted with compression and indexing	Query performance remains acceptable with large datasets
Retention Policy Enforcement	Configure retention policies and wait for enforcement	Old data deleted according to policy	Disk space reclaimed and queries continue working
Compaction Operation	Allow compaction to run and verify data integrity	Data merged correctly with improved performance	No data loss and improved query response times
Range Query Accuracy	Execute queries with various time ranges	Correct data returned within specified ranges	Results match expected values with proper timestamps
Aggregation Correctness	Execute sum, average, and percentile queries	Aggregation results mathematically correct	Aggregated values match manual calculations

Query Engine Validation Checkpoints

The query engine must demonstrate accurate query processing across various query patterns:

- Simple Metric Selection:** Execute queries that select individual metrics and verify that results contain the expected data points.
- Label Filtering:** Execute queries with label filters and verify that only matching series are returned.
- Time Range Specification:** Execute queries with various time ranges and verify that results are properly bounded.
- Aggregation Functions:** Test each aggregation function with known data sets and verify that results match expected calculations.
- Complex Query Processing:** Execute queries that combine multiple operations and verify that results are correct.

Performance Validation Checkpoints

- Query Response Times:** Execute queries against large datasets and verify that response times remain within acceptable bounds (95th percentile < 500ms).
- Concurrent Query Handling:** Execute multiple concurrent queries and verify that the system maintains good performance without resource contention.
- Memory Management:** Monitor memory usage during query execution and verify that it remains bounded even with complex queries.
- Cache Effectiveness:** Verify that query caching improves performance for repeated queries without affecting result accuracy.

Manual Verification Procedures

- Populate the storage engine with a realistic dataset spanning several days
- Execute queries using the query API and verify results manually
- Monitor query execution times and resource usage
- Verify that compaction operations complete successfully
- Test retention policy enforcement by configuring short retention periods

- Verify that aggregation functions produce mathematically correct results

Milestone 3: Visualization Dashboard Validation

The third milestone validation focuses on verifying that the dashboard system correctly displays metric data with real-time updates and proper user interaction.

Dashboard Functionality Validation

Validation Area	Test Procedure	Expected Behavior	Success Criteria
Chart Rendering	Configure panels with various chart types	Charts display data accurately with proper formatting	Visual representations match underlying data
Configuration Persistence	Save and load dashboard configurations	Configurations persist correctly across sessions	Dashboard state maintained after restart
Auto-Refresh Functionality	Enable auto-refresh with various intervals	Charts update automatically without manual intervention	Data freshness maintained according to refresh intervals
Time Range Selection	Use time range controls to change displayed periods	Chart data updates to reflect selected time ranges	Historical and real-time data displayed correctly

Real-Time Update Validation

The dashboard must demonstrate proper real-time behavior:

- WebSocket Connection Management:** Verify that WebSocket connections are established correctly and remain stable during extended sessions.
- Live Data Updates:** Submit new metrics and verify that dashboard panels update automatically without requiring page refresh.
- Subscription Management:** Verify that panels only receive updates for metrics they are configured to display.
- Update Frequency Control:** Test different auto-refresh intervals and verify that updates occur at the specified frequency.
- Error Handling:** Test dashboard behavior when the backend service is unavailable and verify proper error messages and recovery.

User Interface Validation

- Panel Configuration:** Test the ability to create, modify, and delete dashboard panels with various configurations.
- Layout Management:** Verify that panels can be arranged in different grid layouts and that configurations persist correctly.
- Chart Interaction:** Test chart zoom, pan, and tooltip functionality to ensure proper user experience.
- Responsive Design:** Test dashboard functionality across different screen sizes and browser configurations.

Manual Verification Procedures

- Access the dashboard web interface and create a new dashboard
- Configure panels with different chart types and metric queries
- Verify that charts display data correctly and update in real-time
- Test time range controls and auto-refresh functionality
- Save and reload dashboard configurations
- Test WebSocket connectivity and error handling by restarting backend services

Milestone 4: Alerting System Validation

The fourth milestone validation focuses on verifying that the alerting system correctly evaluates alert rules, manages alert states, and delivers notifications reliably.

Alert Rule Validation Checkpoints

Validation Area	Test Procedure	Expected Behavior	Success Criteria
Threshold Evaluation	Configure alerts with various threshold conditions	Alerts fire when thresholds are exceeded for specified duration	Alert timing accuracy within evaluation interval tolerance
State Transition Management	Submit metric data that triggers state changes	Alert states transition correctly through the state machine	Proper notifications sent for each state transition
Duration-Based Conditions	Configure alerts with different duration requirements	Alerts only fire after conditions persist for specified time	Flapping alerts properly suppressed by duration requirements
Multi-Channel Notification	Configure alerts with multiple notification channels	Notifications delivered to all configured channels	Channel-specific message formatting applied correctly

Notification System Validation

The notification system must demonstrate reliable delivery across different channels:

1. **Email Integration:** Configure email notifications and verify that alert messages are delivered with proper formatting and timing.
2. **Slack Integration:** Configure Slack notifications and verify that messages appear in designated channels with appropriate urgency indicators.
3. **Webhook Delivery:** Configure webhook notifications and verify that HTTP requests are sent with correct payloads and retry logic.
4. **Message Templating:** Test custom message templates and verify that alert data is properly substituted into message content.
5. **Rate Limiting:** Test notification rate limiting to verify that notification storms are properly controlled.

Alert Lifecycle Validation

1. **Alert Firing:** Submit metric data that exceeds thresholds and verify that alerts transition to firing state within the expected timeframe.
2. **Alert Resolution:** Submit metric data that resolves conditions and verify that alerts transition to resolved state with appropriate notifications.
3. **Alert Persistence:** Restart the alerting system and verify that alert states are properly restored from persistence storage.
4. **Silence Management:** Test alert silencing functionality and verify that notifications are suppressed during maintenance windows.

Manual Verification Procedures

1. Configure alert rules targeting test metrics with low threshold values
2. Submit metric data that triggers alert conditions
3. Verify that alerts fire and notifications are delivered to configured channels
4. Submit metric data that resolves alert conditions
5. Verify that resolution notifications are sent
6. Test alert silencing and verify that notifications are suppressed
7. Restart the alerting system and verify that alert states persist correctly

The milestone validation checkpoints serve as quality gates that prevent progression to the next development phase until the current capabilities are fully functional and meet their acceptance criteria. This approach ensures that the system maintains quality throughout the development process rather than deferring quality concerns to the end.

Comprehensive System Validation

After completing all individual milestones, a comprehensive system validation verifies that all components work together correctly in realistic scenarios:

- End-to-End Workflow:** Execute a complete workflow from metric ingestion through dashboard visualization and alert notification.
- Load Testing:** Subject the system to realistic load conditions and verify that all components maintain acceptable performance.
- Failure Recovery:** Simulate various failure scenarios and verify that the system recovers properly with minimal data loss.
- Operational Readiness:** Verify that the system provides adequate monitoring, logging, and diagnostic capabilities for operational deployment.

System Integration Scenarios

Scenario	Test Procedure	Expected Outcome	Validation Method
High-Volume Ingestion	Submit sustained high metric volume	System maintains performance without data loss	Monitor throughput and verify data completeness
Complex Dashboard Usage	Multiple users with different dashboards	All dashboards update correctly without interference	Verify independent panel updates and resource usage
Alert Storm Handling	Trigger multiple simultaneous alerts	Notifications delivered reliably without system overload	Verify notification delivery and system stability
Extended Operation	Run system continuously for extended period	No memory leaks or performance degradation	Monitor resource usage trends over time

Implementation Guidance

This section provides practical guidance for implementing the comprehensive testing strategy outlined above, with specific focus on Go-based testing tools and patterns that support both unit and integration testing requirements.

Technology Recommendations

Testing Layer	Simple Option	Advanced Option	Recommended Choice
Unit Testing Framework	<code>testing</code> package with table-driven tests	<code>testify</code> suite with assertions and mocks	<code>testing</code> package (built-in simplicity)
Mocking Strategy	Manual interface mocks	<code>testify/mock</code> with code generation	Manual mocks (explicit dependencies)
Integration Testing	Docker Compose for dependencies	Kubernetes test environments	Docker Compose (simpler setup)
Load Testing	Simple goroutine-based generators	<code>k6</code> or <code>vegeta</code> for sophisticated scenarios	<code>k6</code> (better reporting and scripting)
Test Data Management	Hardcoded test fixtures	<code>gofakeit</code> for generated test data	Combination of both approaches
Assertion Library	Standard <code>if</code> statements with <code>t.Error</code>	<code>testify/assert</code> with rich assertions	<code>testify/assert</code> (better error messages)

Recommended Project Structure

The testing infrastructure should be organized to support both unit and integration testing while maintaining clear separation of concerns:

```
project-root/
  cmd/
    server/main.go
    server/main_test.go      ← integration tests for server startup
  internal/
    ingestion/
      ingestor.go
      ingestor_test.go      ← unit tests for ingestion logic
      ingestor_integration_test.go ← integration tests with real storage
    storage/
      engine.go
      engine_test.go        ← unit tests for storage operations
      engine_integration_test.go ← integration tests with file system
    query/
      processor.go
      processor_test.go     ← unit tests for query processing
      processor_integration_test.go ← integration tests with storage
    alerting/
      evaluator.go
      evaluator_test.go     ← unit tests for alert logic
      evaluator_integration_test.go ← integration tests with notifications
  test/
    fixtures/              ← shared test data and utilities
    metrics.go             ← test metric generators
    storage.go             ← test storage helpers
    time.go                ← time manipulation utilities
    integration/           ← system-level integration tests
      end_to_end_test.go
      performance_test.go   ← complete workflow tests
      ← load and performance tests
    mocks/                 ← interface implementations for testing
      storage_mock.go       ← storage interface mock
      notifier_mock.go      ← notification interface mock
  docker/
    test-compose.yml       ← Docker composition for integration testing
  Makefile               ← test execution automation
```

Testing Infrastructure Components

Mock Storage Engine Implementation

```
// File: test/mocks/storage_mock.go                                     GO

package mocks

import (
    "context"
    "sync"
    "time"

    "github.com/metrics-system/internal/storage"
)

// MockTimeSeriesStorage provides a controllable storage implementation for testing

type MockTimeSeriesStorage struct {

    samples      map[string][]storage.Sample // seriesID -> samples
    series       map[string]*storage.SeriesInfo
    writeErr     error                      // error to return on writes
    queryErr     error                      // error to return on queries
    writeDelay   time.Duration            // simulated write latency
    queryDelay   time.Duration            // simulated query latency
    mu          sync.RWMutex

    // Operation tracking for verification
    WriteCalls []WriteCall
    QueryCalls []QueryCall
}

type WriteCall struct {
    Timestamp time.Time
    Samples   []storage.Sample
    Context   context.Context
}

type QueryCall struct {
    Timestamp time.Time
```

```

    SeriesID    string

    StartTime   time.Time

    EndTime     time.Time

    Context     context.Context

}

// NewMockTimeSeriesStorage creates a new mock storage instance

func NewMockTimeSeriesStorage() *MockTimeSeriesStorage {
    return &MockTimeSeriesStorage{
        samples: make(map[string][]storage.Sample),
        series:  make(map[string]*storage.SeriesInfo),
    }
}

// WriteSamples implements the TimeSeriesStorage interface

func (m *MockTimeSeriesStorage) WriteSamples(ctx context.Context, samples []storage.Sample) error {
    // TODO 1: Record the write call for verification

    // TODO 2: Check if write error should be returned

    // TODO 3: Simulate write delay if configured

    // TODO 4: Store samples in the mock storage

    // TODO 5: Update series information

    // Hint: Use time.Sleep for delay simulation

    // Hint: Group samples by series ID for storage
}

// QueryRange implements the TimeSeriesStorage interface

func (m *MockTimeSeriesStorage) QueryRange(ctx context.Context, seriesID string, start, end time.Time) ([]storage.Sample, error) {
    // TODO 1: Record the query call for verification

    // TODO 2: Check if query error should be returned

    // TODO 3: Simulate query delay if configured

    // TODO 4: Filter samples by time range

    // TODO 5: Return matching samples in chronological order

    // Hint: Use sort.Slice for chronological ordering
}

```

```
}

// SetWriteError configures the mock to return errors on write operations

func (m *MockTimeSeriesStorage) SetWriteError(err error) {

    m.mu.Lock()

    defer m.mu.Unlock()

    m.writeErr = err

}

// SetWriteDelay configures simulated write latency

func (m *MockTimeSeriesStorage) SetWriteDelay(delay time.Duration) {

    m.mu.Lock()

    defer m.mu.Unlock()

    m.writeDelay = delay

}

// GetWriteCalls returns all recorded write operations for verification

func (m *MockTimeSeriesStorage) GetWriteCalls() []WriteCall {

    m.mu.RLock()

    defer m.mu.RUnlock()

    calls := make([]WriteCall, len(m.WriteCalls))

    copy(calls, m.WriteCalls)

    return calls

}

// Reset clears all stored data and operation history

func (m *MockTimeSeriesStorage) Reset() {

    m.mu.Lock()

    defer m.mu.Unlock()

    m.samples = make(map[string][]storage.Sample)

    m.series = make(map[string]*storage.SeriesInfo)

    m.WriteCalls = nil

    m.QueryCalls = nil

    m.writeErr = nil
```

```
m.queryErr = nil  
m.writeDelay = 0  
m.queryDelay = 0  
}
```

Test Fixture Generation

```
// File: test/fixtures/metrics.go                                         GO

package fixtures

import (
    "fmt"
    "math/rand"
    "time"

    "github.com/metrics-system/internal/storage"
)

// MetricGenerator provides utilities for creating test metrics

type MetricGenerator struct {
    rand    *rand.Rand
    labels []storage.Labels
}

// NewMetricGenerator creates a generator with deterministic randomness

func NewMetricGenerator(seed int64) *MetricGenerator {
    return &MetricGenerator{
        rand: rand.New(rand.NewSource(seed)),
    }
}

// GenerateCounterSamples creates realistic counter metric samples

func (g *MetricGenerator) GenerateCounterSamples(metricName string, labels storage.Labels, duration time.Duration, interval time.Duration) []storage.Sample {
    // TODO 1: Calculate number of samples based on duration and interval
    // TODO 2: Generate monotonically increasing counter values
    // TODO 3: Add realistic noise and occasional resets
    // TODO 4: Create samples with proper timestamps
    // TODO 5: Return samples in chronological order
    // Hint: Counter values should generally increase over time
    // Hint: Occasional resets simulate process restarts
}
```

```

}

// GenerateGaugeSamples creates realistic gauge metric samples

func (g *MetricGenerator) GenerateGaugeSamples(metricName string, labels storage.Labels, duration time.Duration, interval time.Duration) []storage.Sample {
    // TODO 1: Calculate number of samples based on duration and interval
    // TODO 2: Generate gauge values with realistic fluctuation patterns
    // TODO 3: Add seasonal patterns and random walk behavior
    // TODO 4: Create samples with proper timestamps
    // TODO 5: Return samples in chronological order
    // Hint: Gauge values can increase or decrease freely
    // Hint: Use sine waves and random walk for realistic patterns
}

// GenerateHistogramSamples creates realistic histogram metric samples

func (g *MetricGenerator) GenerateHistogramSamples(metricName string, labels storage.Labels, buckets []float64, duration time.Duration, interval time.Duration) []storage.Sample {
    // TODO 1: Calculate number of samples based on duration and interval
    // TODO 2: Generate observation counts for each bucket
    // TODO 3: Ensure bucket counts are monotonically increasing across buckets
    // TODO 4: Generate count and sum values consistent with buckets
    // TODO 5: Create samples for all histogram series (buckets, count, sum)
    // Hint: Bucket counts must be cumulative (each bucket includes previous buckets)
    // Hint: Generate separate samples for each bucket, plus _count and _sum series
}

// GenerateHighCardinalityLabels creates label combinations for cardinality testing

func (g *MetricGenerator) GenerateHighCardinalityLabels(baseLabels storage.Labels, cardinalityLevel int) []storage.Labels {
    // TODO 1: Start with base labels as foundation
    // TODO 2: Generate additional label dimensions based on cardinality level
    // TODO 3: Use realistic label names and values (service, instance, etc.)
    // TODO 4: Ensure label combinations are unique
    // TODO 5: Return label sets sorted for deterministic testing
    // Hint: Common high-cardinality labels include instance_id, user_id, request_id
}

```

```
// Hint: Use consistent naming patterns for realistic scenarios
```

```
}
```

Integration Test Harness

GO

```
// File: test/integration/harness.go

package integration

import (
    "context"
    "fmt"
    "path/filepath"
    "testing"
    "time"

    "github.com/metrics-system/internal/config"
    "github.com/metrics-system/internal/coordinator"
)

// TestHarness provides a complete system instance for integration testing

type TestHarness struct {

    Config      *config.Config

    Coordinator *coordinator.ComponentCoordinator

    TempDir     string

    BasePort    int

    // Component references for direct testing

    Storage   storage.TimeSeriesStorage

    Query     query.QueryProcessor

    Dashboard dashboard.DashboardServer

    Alerting   alerting.AlertEvaluator

}

// NewTestHarness creates a complete system instance for testing

func NewTestHarness(t *testing.T) *TestHarness {
    // TODO 1: Create temporary directory for test data
    // TODO 2: Generate unique port numbers to avoid conflicts
    // TODO 3: Create test configuration with appropriate settings
}
```

```

// TODO 4: Initialize component coordinator with test config

// TODO 5: Start all system components

// TODO 6: Register cleanup function with testing.T

// Hint: Use t.TempDir() for automatic cleanup

// Hint: Use port 0 for automatic port assignment or find free ports

// Hint: Use t.Cleanup() to ensure proper shutdown

}

// IngestTestMetrics submits metrics through the ingestion API

func (h *TestHarness) IngestTestMetrics(ctx context.Context, metrics []storage.Metric) error {

    // TODO 1: Format metrics according to ingestion API requirements

    // TODO 2: Submit metrics via HTTP POST to ingestion endpoint

    // TODO 3: Verify successful response codes

    // TODO 4: Wait for metrics to be processed and stored

    // TODO 5: Return any errors encountered during ingestion

    // Hint: Use http.Client for API calls

    // Hint: Add retry logic for transient failures

}

// QueryMetrics executes queries against the system

func (h *TestHarness) QueryMetrics(ctx context.Context, queryString string, timeRange query.TimeRange)
(*query.QueryResult, error) {

    // TODO 1: Format query according to query API requirements

    // TODO 2: Submit query via HTTP GET to query endpoint

    // TODO 3: Parse response into QueryResult structure

    // TODO 4: Validate response format and content

    // TODO 5: Return parsed results or error

    // Hint: Use URL encoding for query parameters

    // Hint: Handle different result types (time series, scalar, etc.)

}

// WaitForAlert waits for an alert to reach the specified state

func (h *TestHarness) WaitForAlert(ctx context.Context, alertID string, expectedState alerting.AlertState,
timeout time.Duration) error {

```

```

// TODO 1: Set up polling loop with context timeout

// TODO 2: Query alert status via alerting API

// TODO 3: Check if alert has reached expected state

// TODO 4: Return success when state matches or timeout when exceeded

// TODO 5: Provide informative error messages for debugging

// Hint: Use time.Ticker for regular polling

// Hint: Include current state in timeout error messages

}

// GetSystemMetrics retrieves internal system metrics for verification

func (h *TestHarness) GetSystemMetrics(ctx context.Context) (map[string]float64, error) {

    // TODO 1: Query system metrics endpoint

    // TODO 2: Parse metrics in Prometheus exposition format

    // TODO 3: Extract key metrics for verification (ingestion rate, storage usage, etc.)

    // TODO 4: Return metrics as key-value map

    // TODO 5: Handle parsing errors gracefully

    // Hint: Look for metrics like metrics_ingested_total, storage_bytes_used

    // Hint: Use Prometheus client library for parsing if available

}

// Shutdown gracefully stops all system components

func (h *TestHarness) Shutdown(ctx context.Context) error {

    // TODO 1: Stop component coordinator gracefully

    // TODO 2: Wait for all components to shutdown completely

    // TODO 3: Clean up temporary files and directories

    // TODO 4: Close any open network connections

    // TODO 5: Return any errors encountered during shutdown

    // Hint: Use context timeout to prevent hanging shutdown

    // Hint: Log shutdown progress for debugging

}

```

Milestone Checkpoint Implementation

Milestone 1 Validation Test

```
// File: test/integration/milestone1_test.go
```

GO

```
package integration
```

```
import (
```

```
    "context"
```

```
    "testing"
```

```
    "time"
```

```
    "github.com/stretchr/testify/assert"
```

```
    "github.com/stretchr/testify/require"
```

```
    "github.com/metrics-system/test/fixtures"
```

```
)
```

```
func TestMilestone1_MetricsCollection(t *testing.T) {
```

```
    harness := NewTestHarness(t)
```

```
    ctx := context.Background()
```

```
    t.Run("Counter Metrics Ingestion", func(t *testing.T) {
```

```
        // TODO 1: Generate counter metrics with increasing values
```

```
        // TODO 2: Submit metrics through ingestion API
```

```
        // TODO 3: Query storage directly to verify persistence
```

```
        // TODO 4: Validate that counter semantics are enforced
```

```
        // TODO 5: Check that monotonic increases are preserved
```

```
        generator := fixtures.NewMetricGenerator(12345)
```

```
        // Implementation continues with detailed test steps...
```

```
    })
```

```
    t.Run("Gauge Metrics Processing", func(t *testing.T) {
```

```
        // TODO 1: Generate gauge metrics with fluctuating values
```

```
        // TODO 2: Submit metrics and verify immediate storage
```

```
        // TODO 3: Validate that gauge values can increase and decrease
```

```
// TODO 4: Check that latest values are queryable

// TODO 5: Verify proper label handling and indexing

// Implementation continues with detailed test steps...

})

t.Run("Histogram Metrics Handling", func(t *testing.T) {

    // TODO 1: Generate histogram metrics with proper bucket distributions

    // TODO 2: Submit histograms and verify bucket processing

    // TODO 3: Validate bucket count calculations

    // TODO 4: Check that sum and count values are correct

    // TODO 5: Verify percentile calculation capability

    // Implementation continues with detailed test steps...

})

t.Run("Label Validation and Cardinality", func(t *testing.T) {

    // TODO 1: Test label parsing and normalization

    // TODO 2: Submit metrics with high cardinality labels

    // TODO 3: Verify cardinality limits are enforced

    // TODO 4: Check that valid labels are processed correctly

    // TODO 5: Validate error messages for invalid labels

    // Implementation continues with detailed test steps...

})

t.Run("Performance Validation", func(t *testing.T) {

    // TODO 1: Generate high-volume metric load

    // TODO 2: Monitor ingestion rates and latency

    // TODO 3: Verify memory usage remains bounded

    // TODO 4: Check that system maintains responsiveness

    // TODO 5: Validate that no metrics are lost during load
```

```
startTime := time.Now()

// Generate 10,000 metrics over 60 seconds

metricsCount := 10000

duration := 60 * time.Second

// Implementation continues with performance testing...

})

}

// validateCounterSemantics verifies counter-specific behavior

func validateCounterSemantics(t *testing.T, harness *TestHarness, samples []storage.Sample) {

    // TODO 1: Verify that counter values are monotonically increasing

    // TODO 2: Check for proper counter reset detection

    // TODO 3: Validate that rate calculations work correctly

    // TODO 4: Ensure proper handling of counter overflow

    // TODO 5: Verify storage optimization for counter data

    // Implementation continues with validation logic...

}

// validateIngestionPerformance checks performance characteristics

func validateIngestionPerformance(t *testing.T, harness *TestHarness, metricsSubmitted int, duration time.Duration) {

    // TODO 1: Calculate actual ingestion rate

    // TODO 2: Check that 95th percentile latency is acceptable

    // TODO 3: Verify memory usage didn't spike excessively

    // TODO 4: Ensure no metrics were dropped or lost

    // TODO 5: Validate system remained responsive during load

    systemMetrics, err := harness.GetSystemMetrics(context.Background())

    require.NoError(t, err)
```

```

actualRate := systemMetrics["metrics_ingested_total"] / duration.Seconds()

expectedMinRate := float64(metricsSubmitted) / duration.Seconds() * 0.95 // Allow 5% tolerance

assert.GreaterOrEqual(t, actualRate, expectedMinRate,
    "Ingestion rate %f/sec below expected minimum %f/sec", actualRate, expectedMinRate)

// Continue with additional performance validations...

}

```

This comprehensive testing strategy provides multiple layers of validation that ensure system correctness, performance, and reliability throughout development. The combination of unit tests, integration tests, and milestone checkpoints creates a robust quality assurance framework that catches issues early and validates that the system meets its requirements under realistic conditions.

Debugging Guide

Milestone(s): All milestones (debugging techniques are essential throughout the entire metrics and alerting system lifecycle, from initial development through production operations)

The Detective Work Metaphor: Understanding System Debugging

Think of debugging a metrics system like being a detective investigating a complex case. You have multiple witnesses (log files), physical evidence (metric data), and crime scene locations (system components). The key is knowing which evidence to collect, how to interpret the clues, and following a systematic investigation process. Just as a detective starts with the most obvious suspects before diving into complex conspiracy theories, we debug metrics systems by checking the most common failure modes first—network connectivity, configuration errors, and resource constraints—before investigating more subtle issues like timing races or cardinality explosions.

Unlike debugging a simple web application where problems are often isolated to a single request-response cycle, metrics systems involve continuous data flows with multiple asynchronous components. A single problem can manifest symptoms in multiple locations: metrics might disappear at ingestion, queries might return partial results, dashboards might show stale data, or alerts might fire unexpectedly. The challenge is connecting these distributed symptoms back to their root cause while the system continues operating.

Common Problem Patterns in Metrics Systems

Before diving into specific debugging techniques, it's important to understand the common failure patterns that occur in metrics and alerting systems. These patterns help guide your investigation and prioritize which components to examine first.

Failure Pattern	Typical Symptoms	Investigation Priority	Common Root Causes
Data Pipeline Failures	Missing metrics, partial ingestion, query timeouts	High	Network issues, storage full, validation errors
Resource Exhaustion	Slow queries, high memory usage, ingestion backlog	High	Cardinality explosion, insufficient capacity, memory leaks
Configuration Errors	Alerts not firing, incorrect dashboards, connection failures	Medium	Typos, wrong endpoints, invalid credentials
Timing and Synchronization	Intermittent failures, race conditions, inconsistent behavior	Medium	Clock skew, concurrent access, async processing
State Corruption	Inconsistent data, crashed components, recovery failures	Low	Hardware failures, bugs in storage layer, power loss

Key Insight: Most production issues in metrics systems stem from resource exhaustion or configuration problems rather than complex algorithmic bugs. Start your investigation with resource monitoring and configuration validation before diving into code-level debugging.

The debugging approach for metrics systems follows a structured methodology:

1. **Establish the Failure Scope:** Determine whether the problem affects all metrics, specific metric types, certain time ranges, or particular dashboard panels
2. **Trace the Data Path:** Follow metric data from ingestion through storage, querying, and visualization to identify where the pipeline breaks
3. **Correlate Symptoms Across Components:** Check if problems in one component are causing cascading failures in dependent systems
4. **Validate System Resources:** Ensure adequate CPU, memory, disk space, and network capacity for current load
5. **Reproduce with Controlled Input:** Use synthetic metrics and queries to isolate variables and confirm fixes

Ingestion Issues

The ingestion engine is the entry point for all metric data, making it a common source of problems when metrics fail to appear in dashboards or alerts don't fire as expected. Ingestion issues typically manifest as missing metrics, partial data ingestion, or performance degradation during high-volume periods.

Metrics Not Appearing

The most common ingestion problem is metrics disappearing somewhere between client submission and storage persistence. This creates a frustrating debugging experience because the problem could exist at multiple layers of the ingestion pipeline.

Systematic Investigation Approach:

Investigation Step	Commands/Actions	What to Look For	Next Step If Found
Verify Network Connectivity	<code>curl -X POST /api/metrics -d @test-metric.json</code>	HTTP response code, connection timeouts	Check firewall rules, DNS resolution
Check Ingestion Logs	<code>grep "metric ingestion" /var/log/metrics.log</code>	Validation errors, parsing failures	Review metric format against schema
Validate Metric Format	Compare against <code>Metric</code> struct requirements	Missing fields, invalid types, malformed JSON	Fix client-side metric serialization
Examine Storage Writes	Monitor <code>WriteSamples</code> method calls and errors	Write failures, disk space errors	Check storage configuration and capacity
Verify Label Cardinality	Check cardinality manager logs and metrics	High-cardinality rejections, quota exceeded	Reduce label diversity or increase limits

Decision: Ingestion Pipeline Observability Strategy

- **Context:** When metrics don't appear, determining where they're lost requires visibility into each pipeline stage
- **Options Considered:**
 1. Per-component error logging only
 2. End-to-end tracing with correlation IDs
 3. Pipeline metrics with stage-specific counters
- **Decision:** Hybrid approach combining pipeline metrics with correlation IDs for complex failures
- **Rationale:** Pipeline metrics provide immediate visibility into healthy vs. failing stages, while correlation IDs enable detailed investigation of specific metric flows when needed
- **Consequences:** Enables rapid problem isolation but requires additional instrumentation overhead

⚠ Pitfall: Assuming Network Problems Are External

Many developers assume that HTTP 200 responses mean successful metric ingestion, but the ingestion pipeline continues after the HTTP response. The `MetricsIngestor` might accept the request but later reject metrics during validation or storage. Always check the complete pipeline, not just the HTTP layer.

Common Validation Failures:

Validation Error	Symptom	Debug Command	Fix
Invalid Metric Name	<code>ValidationErrors</code> in logs with "invalid metric name"	Check metric name against regex pattern	Use alphanumeric names with underscores
Label Cardinality Exceeded	<code>CardinalityManager</code> rejection logs	Review unique label combinations	Reduce label diversity or increase limits
Timestamp Out of Range	"timestamp too old/new" errors	Compare timestamps to retention policy	Adjust timestamp or retention configuration
Malformed Sample Values	"invalid float64" parsing errors	Validate sample values are numeric	Fix client serialization of infinity/NaN
Missing Required Labels	"required label missing" validation errors	Check against label requirements config	Add missing labels to metric submissions

Cardinality Explosions

Cardinality explosions occur when metrics contain too many unique label combinations, overwhelming storage and query performance. This is one of the most dangerous ingestion problems because it can quickly consume all available system resources.

Detection Strategies:

The `CardinalityManager` component tracks unique label combinations and provides early warning when cardinality approaches dangerous levels. Monitor these metrics continuously:

Cardinality Metric	Healthy Range	Warning Threshold	Critical Threshold	Response Action
Unique Series Count	< 100K per metric	> 500K per metric	> 1M per metric	Enable label filtering
New Series Rate	< 1K/minute	> 10K/minute	> 50K/minute	Investigate metric source
Label Combination Diversity	< 10 per label	> 100 per label	> 1000 per label	Review label design
Memory Usage per Series	< 1KB average	> 5KB average	> 10KB average	Optimize storage format

Key Insight: Cardinality problems are exponential—each new label dimension multiplies the potential series count. A metric with 10 possible values for 3 labels creates 1,000 unique series (10^3), but adding one more label with 10 values creates 10,000 series (10^4).

Investigation Process:

- Identify High-Cardinality Metrics:** Query the series index to find metrics with excessive unique combinations
- Analyze Label Distribution:** Examine which labels contribute most to cardinality growth
- Review Metric Design:** Determine if high-cardinality labels are necessary for observability goals
- Implement Cardinality Controls:** Configure limits, sampling, or label filtering to manage growth

Common High-Cardinality Label Patterns:

Problematic Pattern	Example	Why It's Dangerous	Better Alternative
Request IDs in Labels	<code>request_id="uuid-123"</code>	Every request creates new series	Use request ID in annotations or separate tracing
Timestamp Labels	<code>hour="2024-01-15-14"</code>	Creates series per time bucket	Use native timestamp field instead
User IDs in Labels	<code>user_id="user123"</code>	Series grows with user base	Aggregate by user groups or roles
Full URLs as Labels	<code>url="/api/users/123/profile"</code>	Creates series per URL variant	Extract URL patterns or endpoints only
Random Values	<code>session_id="random123"</code>	Unbounded series growth	Remove or use in annotations

Performance Problems During High Volume

Ingestion performance problems typically manifest as increased latency, memory growth, or eventually request timeouts when the system cannot keep up with incoming metric volume.

Performance Monitoring Approach:

The `MetricsIngestor` component should expose these performance metrics for continuous monitoring:

Performance Metric	Description	Healthy Baseline	Investigation Trigger
Ingestion Throughput	Metrics processed per second	> 10K metrics/sec	< 1K metrics/sec sustained
Ingestion Latency P99	Time from request to storage write	< 100ms	> 500ms
Validation Queue Depth	Pending metrics awaiting validation	< 1000	> 10000
Storage Write Latency	Time for <code>WriteSamples</code> to complete	< 50ms	> 200ms
Memory Usage per Batch	Memory consumed processing metric batches	< 10MB per batch	> 100MB per batch

Scalability Bottleneck Analysis:

Bottleneck Location	Symptoms	Root Cause Analysis	Scaling Strategy
HTTP Request Processing	High request latency, connection timeouts	Single-threaded request handlers	Increase <code>GOMAXPROCS</code> , use connection pooling
Metric Validation	CPU spikes during validation, processing delays	Complex validation rules, inefficient regex	Optimize validation algorithms, cache compiled patterns
Label Processing	Memory growth with label-heavy metrics	Inefficient label storage, no deduplication	Implement label interning, optimize <code>Labels</code> type
Storage Writes	Write latency increases, disk I/O saturation	Synchronous writes, no write batching	Implement write batching, async persistence
Memory Management	GC pauses, out-of-memory errors	Large object allocations, no object pooling	Use sync.Pool for metric objects, optimize GC tuning

⚠ Pitfall: Optimizing the Wrong Component

Performance problems often appear in one component but originate in another. For example, slow query performance might be caused by inefficient ingestion creating poorly structured storage blocks, not the query engine itself. Always profile the entire pipeline before optimizing individual components.

Storage and Query Issues

Storage and query problems are particularly challenging to debug because they often involve subtle interactions between the storage engine's complex components: the write-ahead log, block manager, compaction engine, and query processor.

Slow Query Performance

Query performance problems typically stem from inefficient storage layouts, missing indexes, or queries that scan excessive amounts of data. The challenge is determining whether the problem is in the query itself, the underlying storage structure, or resource constraints.

Query Performance Investigation Framework:

Investigation Phase	Diagnostic Actions	Key Metrics to Examine	Resolution Path
Query Analysis	Parse query complexity, time range scope	Series count, sample count, aggregation types	Optimize query structure or add filters
Index Utilization	Check series selection efficiency	Index hit ratio, series scan count	Add indexes or improve label filtering
Storage Layout	Examine block structure and compaction	Block count, block time ranges, overlap ratio	Trigger manual compaction or adjust policies
Resource Constraints	Monitor CPU, memory, I/O during queries	CPU utilization, memory allocation, disk reads	Scale resources or implement query limits

Common Query Performance Anti-Patterns:

Anti-Pattern	Example Query Symptom	Storage Impact	Fix Strategy
Unbounded Time Range	Query spans months of data	Scans thousands of storage blocks	Add explicit time range limits
High-Cardinality Grouping	Group by high-diversity labels	Processes millions of series	Use sampling or pre-aggregated metrics
Inefficient Label Matching	Regular expressions on label values	Full index scan required	Use exact matches or optimize regex
Cross-Series Math	Complex arithmetic across many series	Memory explosion during aggregation	Break into smaller queries or use streaming
Missing Filters	No label selectors in metric queries	Processes all series for metric	Add specific label matchers

Decision: Query Optimization Strategy

- **Context:** Queries can become extremely expensive as data volume grows, potentially impacting system stability
- **Options Considered:**
 1. Query timeout limits only
 2. Resource-based query rejection (memory/CPU limits)
 3. Query complexity analysis with progressive limits
- **Decision:** Hybrid approach with complexity analysis, resource limits, and progressive timeouts
- **Rationale:** Prevents runaway queries while allowing legitimate complex queries during low-load periods
- **Consequences:** Requires query complexity estimation but provides better user experience than hard timeouts

Query Execution Pipeline Debugging:

The `QueryProcessor` breaks query execution into distinct phases, each of which can be monitored and optimized independently:

1. **Query Parsing Phase:** Convert query string to abstract syntax tree
 - Monitor: Parse time, syntax errors, unsupported functions
 - Debug: Enable query parser logging, validate AST structure
2. **Query Planning Phase:** Generate execution plan with series selection strategy

- Monitor: Planning time, estimated series count, memory requirements
- Debug: Log execution plans, compare estimated vs. actual resource usage

3. Series Selection Phase: Identify time series matching query selectors

- Monitor: Index lookup time, series count, label matching efficiency
- Debug: Log index utilization, profile label matching algorithms

4. Data Retrieval Phase: Load sample data from storage blocks

- Monitor: Block read count, decompression time, I/O wait time
- Debug: Track block access patterns, identify hot/cold data distribution

5. Aggregation Phase: Process samples according to query functions

- Monitor: Aggregation time, memory usage, output series count
- Debug: Profile aggregation functions, monitor streaming vs. batch processing

Missing or Inconsistent Data

Data consistency problems in time-series systems often result from race conditions between ingestion, compaction, and querying processes. These problems are particularly insidious because they may only affect specific time ranges or metric combinations.

Data Consistency Investigation Process:

Investigation Step	Diagnostic Approach	Tools and Commands	Expected Findings
Verify Data Ingestion	Check if samples reached storage	Review ingestion logs and <code>WriteSamples</code> calls	All submitted samples should appear in WAL
Examine WAL State	Inspect write-ahead log for sample presence	WAL reader tools, log file analysis	Samples should exist with correct timestamps
Check Block Structure	Verify samples exist in storage blocks	Block inspection tools, storage debugging	Samples should be in appropriate time-bucketed blocks
Analyze Compaction Impact	Determine if compaction corrupted data	Compaction logs, before/after block comparison	Sample count and values should be preserved
Query Path Validation	Trace query execution through storage layers	Query debugging logs, execution plan analysis	Queries should access all relevant blocks

Common Data Consistency Issues:

Consistency Problem	Typical Manifestation	Root Cause Analysis	Resolution Strategy
Samples Disappear After Compaction	Queries return fewer points than expected	Compaction algorithm bug or configuration error	Validate compaction logic, check downsampling rules
Duplicate Samples for Same Timestamp	Aggregations produce incorrect results	Multiple ingestion paths or replay issues	Implement timestamp deduplication, fix replay logic
Time Range Gaps	Missing data for specific time periods	WAL corruption, failed writes, or clock skew	Recover from WAL backups, sync system clocks
Label Inconsistency	Same metric appears with different labels	Concurrent label updates, caching issues	Implement label validation, clear metadata caches
Query Result Variations	Same query returns different results	Read-write race conditions, index staleness	Add read barriers, refresh indexes before queries

Key Insight: Time-series data consistency problems often have delayed manifestation—samples might be ingested successfully but lost during later compaction or corrupted during concurrent reads. Always verify data through the complete storage lifecycle.

⚠ Pitfall: Assuming WAL Guarantees Consistency

While the write-ahead log provides durability guarantees, it doesn't protect against logic errors in compaction or querying. A correctly written WAL can still result in missing data if the compaction process has bugs or if queries don't properly handle block boundaries.

Storage Corruption Detection and Recovery

Storage corruption in time-series systems can occur at multiple levels: file system corruption, block-level data corruption, or index inconsistencies. Early detection and recovery procedures are critical for maintaining data integrity.

Corruption Detection Strategy:

Detection Method	Implementation	Trigger Conditions	Recovery Actions
Block Checksums	Compute checksums on write, verify on read	Checksum mismatch during block access	Mark block as corrupted, attempt recovery from WAL
Index Validation	Periodic consistency checks between index and blocks	Scheduled validation, startup checks	Rebuild index from storage blocks
Sample Count Validation	Track expected vs. actual sample counts	Query result discrepancies	Compare with ingestion metrics, identify missing blocks
Temporal Consistency	Verify timestamp ordering within blocks	Out-of-order samples detected	Re-sort block contents, check compaction logic
Cross-Reference Validation	Compare WAL entries with storage blocks	Samples in WAL missing from blocks	Replay missing WAL entries

Storage Recovery Procedures:

The `StorageEngine` should implement comprehensive recovery mechanisms that can handle various corruption scenarios:

- 1. WAL-Based Recovery:** Replay write-ahead log entries to reconstruct missing or corrupted blocks
- 2. Block Reconstruction:** Rebuild corrupted blocks from available data sources

3. **Index Rebuilding:** Regenerate series indexes from existing storage blocks
4. **Partial Recovery:** Recover what data is possible and mark irretrievable data as missing
5. **Backup Integration:** Restore from external backups when local recovery fails

Alerting Issues

Alerting system problems are often the most critical because they directly impact incident response and system reliability. Alert failures can range from alerts not firing when they should (false negatives) to excessive alert noise (false positives).

Alerts Not Firing

The most dangerous alerting problem is alerts failing to fire when conditions are met, as this can lead to undetected outages or performance degradation.

Alert Evaluation Investigation Process:

Investigation Step	Diagnostic Actions	Key Information to Gather	Resolution Path
Verify Alert Rule Configuration	Review <code>AlertRule</code> definition syntax	Rule expression, threshold values, duration settings	Fix syntax errors, validate threshold logic
Check Alert Evaluation Loop	Monitor <code>AlertEvaluator</code> execution	Evaluation frequency, rule processing time, errors	Restart evaluator, fix evaluation logic
Validate Data Availability	Confirm metrics exist for alert queries	Query results, series existence, timestamp alignment	Fix metric ingestion or query problems
Examine Alert State Management	Review state transitions and timing	Current state, state change history, duration tracking	Reset alert state, fix state machine logic
Test Query Execution	Run alert query manually	Query results, execution errors, performance	Optimize query or fix data source

Common Alert Evaluation Failures:

Failure Mode	Symptoms	Root Cause	Debug Approach	Fix Strategy
Query Returns No Data	Alert never transitions from Inactive	Metric name mismatch, missing labels	Execute alert query manually	Correct metric selector syntax
Threshold Logic Error	Alert fires at wrong values	Incorrect comparison operators, type mismatches	Log threshold comparisons	Fix operator logic (>, <, ==)
Duration Requirements Not Met	Alert stays in Pending state	Insufficient duration for sustained condition	Check state transition timing	Adjust duration requirements
Clock Skew Issues	Timestamps don't align with evaluation	System clock differences, timezone problems	Compare timestamps across components	Synchronize system clocks
Resource Exhaustion	Alert evaluator stops processing	Memory/CPU limits exceeded during evaluation	Monitor evaluator resource usage	Scale alert evaluation capacity

Decision: Alert Evaluation Reliability Strategy

- **Context:** Alert failures can have severe operational consequences, requiring high reliability guarantees
- **Options Considered:**
 1. Single-threaded evaluation with checkpointing
 2. Distributed evaluation with consensus
 3. Redundant evaluators with leader election
- **Decision:** Single-threaded evaluation with comprehensive checkpointing and fast failure detection
- **Rationale:** Simpler architecture with fewer failure modes while maintaining reliability through rapid detection and recovery
- **Consequences:** Single point of failure but faster recovery and easier debugging

Alert State Machine Debugging:

The alert state machine manages transitions between `AlertStateInactive`, `AlertStatePending`, `AlertStateFiring`, and `AlertStateResolved`. State transition problems often indicate timing issues or logic errors.

Current State	Expected Trigger	Expected Next State	Common Failures	Debug Actions
Inactive	Threshold exceeded	Pending	Query returns no data, threshold comparison error	Verify query results, log comparison values
Pending	Duration requirement met	Firing	Timer not advancing, state not persisted	Check duration tracking, verify state storage
Firing	Condition no longer met	Resolved	Notification delivery blocking state updates	Separate notification from state management
Resolved	Condition met again	Pending	Rapid state flapping, insufficient hysteresis	Add state change delays, implement hysteresis

Notification Delivery Problems

Even when alerts fire correctly, notification delivery failures can prevent teams from receiving critical alerts. Notification problems often involve external services and network connectivity issues.

Notification Pipeline Debugging:

Pipeline Stage	Common Failures	Detection Methods	Resolution Approaches
Message Formatting	Template errors, missing data	Template compilation errors, malformed messages	Fix template syntax, validate data availability
Channel Selection	Wrong channels chosen, disabled channels	Routing logs, channel configuration	Update routing rules, enable required channels
External Service Integration	API failures, authentication errors	HTTP response codes, service error logs	Fix credentials, handle API rate limits
Network Connectivity	DNS resolution, firewall blocks	Network timeouts, connection errors	Check network configuration, update DNS
Rate Limiting	Too many notifications sent	Rate limit exceeded errors	Implement backoff, batch notifications

Pitfall: Ignoring Notification Delivery Confirmations

Many alert systems send notifications without verifying delivery success. Network failures, API outages, or configuration errors can silently prevent notifications from reaching their destination. Always implement delivery confirmation and retry logic.

Notification Channel Health Monitoring:

Channel Type	Health Check Method	Failure Detection	Recovery Actions
Email SMTP	SMTP connection test, authentication validation	SMTP errors, timeout responses	Retry with exponential backoff, try alternate SMTP servers
Slack Webhook	HTTP POST test to webhook URL	HTTP 4xx/5xx responses, network timeouts	Refresh webhook URLs, check Slack app permissions
PagerDuty API	API key validation, service availability	API authentication errors, service unavailable	Rotate API keys, use backup notification channels
Custom Webhook	HTTP endpoint availability test	Connection refused, DNS resolution failures	Health check endpoints, implement circuit breakers

Alert Flapping and Noise Reduction

Alert flapping occurs when conditions rapidly oscillate around threshold values, causing alerts to fire and resolve repeatedly. This creates noise that can mask genuine problems and lead to alert fatigue.

Flapping Detection and Mitigation:

Flapping Pattern	Detection Method	Root Cause	Mitigation Strategy
Rapid Fire/Resolve Cycles	State changes within short time windows	Threshold too close to normal values	Increase threshold gap, add hysteresis
Periodic Oscillation	Regular pattern of state changes	Scheduled processes, batch jobs	Adjust evaluation timing, use trend analysis
Network-Induced Flapping	Correlates with network events	Network partitions, intermittent connectivity	Increase evaluation duration, use circuit breakers
Load-Based Flapping	Correlates with traffic patterns	Resource saturation during peak load	Use percentile thresholds, add capacity buffers

Key Insight: Alert flapping is often a symptom of poorly chosen thresholds rather than system instability. Effective alerting requires thresholds that account for normal system variation while still detecting genuine problems promptly.

Implementation Guidance

This debugging section requires sophisticated tooling and instrumentation to make problems visible when they occur. The following implementation provides structured debugging capabilities across all system components.

Technology Recommendations

Debugging Component	Simple Option	Advanced Option
Logging Framework	Go <code>slog</code> with JSON output	Structured logging with correlation IDs
Metrics Collection	Prometheus client library	Custom metrics with detailed labels
Performance Profiling	Go <code>net/http/pprof</code>	Continuous profiling with flame graphs
Health Checking	HTTP endpoints with JSON status	Comprehensive health dashboard
Error Tracking	File-based error logs	Centralized error aggregation

Recommended File Structure

```
project-root/
  cmd/
    metrics-server/main.go      ← main entry point with debug flags
    debug-tools/
      cardinality-analyzer/main.go   ← tool for cardinality analysis
      storage-inspector/main.go     ← tool for examining storage blocks
      alert-tester/main.go          ← tool for testing alert rules
  internal/
    debugging/
      health_manager.go           ← centralized health checking
      performance_monitor.go      ← performance metrics collection
      debug_handlers.go           ← HTTP handlers for debug endpoints
    diagnostics/
      ingestion_diagnostics.go    ← ingestion-specific debugging
      storage_diagnostics.go      ← storage-specific debugging
      alert_diagnostics.go        ← alerting-specific debugging
  tools/
    validate-config.go           ← configuration validation
    generate-test-metrics.go     ← synthetic metric generation
```

Infrastructure Starter Code

Complete Health Management System:

GO

```
// internal/debugging/health_manager.go

package debugging

import (
    "context"
    "encoding/json"
    "fmt"
    "log/slog"
    "net/http"
    "sync"
    "time"
)

// HealthStatus represents the overall health state of a component

type HealthStatus int

const (
    HealthStatusHealthy HealthStatus = iota
    HealthStatusDegraded
    HealthStatusUnhealthy
)

func (s HealthStatus) String() string {
    switch s {
    case HealthStatusHealthy:
        return "healthy"
    case HealthStatusDegraded:
        return "degraded"
    case HealthStatusUnhealthy:
        return "unhealthy"
    default:
        return "unknown"
    }
}
```

```

// HealthCheck represents a single health check result

type HealthCheck struct {

    Name      string      `json:"name"`
    Status    HealthStatus `json:"status"`
    LastChecked time.Time   `json:"last_checked"`
    Message   string      `json:"message"`
}

// HealthCheckFunc defines the signature for health check functions

type HealthCheckFunc func(ctx context.Context) error

// HealthManager coordinates health checks across all system components

type HealthManager struct {

    checks map[string]*HealthCheck
    funcs  map[string]HealthCheckFunc
    mu     sync.RWMutex
    logger *slog.Logger
}

func NewHealthManager(logger *slog.Logger, interval time.Duration) *HealthManager {
    hm := &HealthManager{
        checks: make(map[string]*HealthCheck),
        funcs:  make(map[string]HealthCheckFunc),
        logger: logger,
    }

    // Start periodic health check execution
    go hm.runPeriodicChecks(interval)

    return hm
}

func (hm *HealthManager) RegisterCheck(name string, checkFunc HealthCheckFunc) {

```

```
hm.mu.Lock()

defer hm.mu.Unlock()

hm.funcs[name] = checkFunc

hm.checks[name] = &HealthCheck{
    Name:      name,
    Status:    HealthStatusUnhealthy,
    LastChecked: time.Time{},
    Message:   "not yet checked",
}

}

func (hm *HealthManager) RunChecks(ctx context.Context) {

    hm.mu.RLock()
    funcs := make(map[string]HealthCheckFunc)
    for name, fn := range hm.funcs {
        funcs[name] = fn
    }
    hm.mu.RUnlock()

    for name, checkFunc := range funcs {
        hm.runSingleCheck(ctx, name, checkFunc)
    }
}

func (hm *HealthManager) runSingleCheck(ctx context.Context, name string, checkFunc HealthCheckFunc) {

    start := time.Now()
    err := checkFunc(ctx)
    duration := time.Since(start)

    hm.mu.Lock()
    check := hm.checks[name]
    check.LastChecked = time.Now()
```

```
if err != nil {
    check.Status = HealthStatusUnhealthy
    check.Message = fmt.Sprintf("check failed: %v (took %v)", err, duration)
    hm.logger.Error("health check failed",
        slog.String("check", name),
        slog.String("error", err.Error()),
        slog.Duration("duration", duration))
} else {
    // Consider slow checks as degraded
    if duration > 5*time.Second {
        check.Status = HealthStatusDegraded
        check.Message = fmt.Sprintf("check slow but successful (took %v)", duration)
    } else {
        check.Status = HealthStatusHealthy
        check.Message = fmt.Sprintf("check successful (took %v)", duration)
    }
}
hm.mu.Unlock()
}

func (hm *HealthManager) GetOverallStatus() HealthStatus {
    hm.mu.RLock()
    defer hm.mu.RUnlock()

    if len(hm.checks) == 0 {
        return HealthStatusUnhealthy
    }

    overallStatus := HealthStatusHealthy
    for _, check := range hm.checks {
        if check.Status == HealthStatusUnhealthy {
```

```
        return HealthStatusUnhealthy

    }

    if check.Status == HealthStatusDegraded {

        overallStatus = HealthStatusDegraded

    }

}

return overallStatus
}

func (hm *HealthManager) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    hm.mu.RLock()

    checks := make([]*HealthCheck, 0, len(hm.checks))

    for _, check := range hm.checks {

        checks = append(checks, check)

    }

    hm.mu.RUnlock()

    overallStatus := hm.GetOverallStatus()

    response := map[string]interface{}{
        "status": overallStatus.String(),
        "checks": checks,
        "timestamp": time.Now(),
    }

    w.Header().Set("Content-Type", "application/json")

    // Set HTTP status based on health

    switch overallStatus {

    case HealthStatusHealthy:

        w.WriteHeader(http.StatusOK)

    case HealthStatusDegraded:

        w.WriteHeader(http.StatusServiceUnavailable)

    case HealthStatusUnhealthy:

        w.WriteHeader(http.StatusInternalServerError)

    }

}
```

```
case HealthStatusDegraded:

    w.WriteHeader(http.StatusOK) // 200 but degraded

case HealthStatusUnhealthy:

    w.WriteHeader(http.StatusServiceUnavailable)

}

json.NewEncoder(w).Encode(response)

}

func (hm *HealthManager) runPeriodicChecks(interval time.Duration) {

    ticker := time.NewTicker(interval)

    defer ticker.Stop()

    for range ticker.C {

        ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)

        hm.RunChecks(ctx)

        cancel()

    }

}
```

Complete Performance Monitoring System:

```
// internal/debugging/performance_monitor.go                                GO

package debugging

import (
    "runtime"
    "sync"
    "time"
)

// PerformanceMetrics tracks system performance indicators

type PerformanceMetrics struct {

    // Ingestion metrics

    IngestRate      float64   `json:"ingest_rate_per_sec"`
    IngestLatencyP99 time.Duration `json:"ingest_latency_p99"`
    ValidationErrors int64     `json:"validation_errors"`

    // Storage metrics

    StorageWriteRate float64   `json:"storage_write_rate_per_sec"`
    StorageWriteLatency time.Duration `json:"storage_write_latency"`
    CompactionDuration time.Duration `json:"last_compaction_duration"`

    // Query metrics

    QueryRate      float64   `json:"query_rate_per_sec"`
    QueryLatencyP99 time.Duration `json:"query_latency_p99"`
    SlowQueryCount int64     `json:"slow_query_count"`

    // Alert metrics

    AlertEvaluationRate float64   `json:"alert_eval_rate_per_sec"`
    AlertsActive       int64     `json:"alerts_active"`
    NotificationFailures int64   `json:"notification_failures"`

    // System metrics

    MemoryUsage      uint64    `json:"memory_usage_bytes"`
}
```

```
GoroutineCount      int       `json:"goroutine_count"`

GCDuration         time.Duration `json:"gc_duration"`

// Timestamps

Timestamp          time.Time `json:"timestamp"`

mu                sync.RWMutex

}

func NewPerformanceMonitor() *PerformanceMetrics {
    pm := &PerformanceMetrics{
        Timestamp: time.Now(),
    }

    // Start periodic system metrics collection
    go pm.collectSystemMetrics()

    return pm
}

func (pm *PerformanceMetrics) collectSystemMetrics() {
    ticker := time.NewTicker(10 * time.Second)
    defer ticker.Stop()

    for range ticker.C {
        pm.mu.Lock()

        // Memory statistics

        var memStats runtime.MemStats
        runtime.ReadMemStats(&memStats)
        pm.MemoryUsage = memStats.Alloc
        pm.GCDuration = time.Duration(memStats.PauseTotalNs)

        // Goroutine count
    }
}
```

```
pm.GoroutineCount = runtime.NumGoroutine()

pm.Timestamp = time.Now()
pm.mu.Unlock()
}

}

func (pm *PerformanceMetrics) RecordIngestMetric(latency time.Duration, success bool) {
pm.mu.Lock()
defer pm.mu.Unlock()

// Update ingestion rate (simplified - real implementation would use sliding window)
pm.IngestRate++

// Update latency percentile (simplified - real implementation would use histogram)
if latency > pm.IngestLatencyP99 {
pm.IngestLatencyP99 = latency
}

if !success {
pm.ValidationErrors++
}

}

func (pm *PerformanceMetrics) GetSnapshot() *PerformanceMetrics {
pm.mu.RLock()
defer pm.mu.RUnlock()

snapshot := *pm
return &snapshot
}
```

Core Logic Skeleton Code

Ingestion Diagnostics Framework:

```
// internal/diagnostics/ingestion_diagnostics.go          GO

package diagnostics

import (
    "context"
    "fmt"
    "log/slog"
    "time"
)

type IngestionDiagnostics struct {

    logger *slog.Logger

    // Add references to ingestion components
}

func NewIngestionDiagnostics(logger *slog.Logger) *IngestionDiagnostics {
    return &IngestionDiagnostics{
        logger: logger,
    }
}

// DiagnoseMetricIngestion performs comprehensive ingestion pipeline analysis

func (id *IngestionDiagnostics) DiagnoseMetricIngestion(ctx context.Context, metricName string, timeRange time.Duration) error {

    // TODO 1: Check if metrics are reaching the ingestion endpoint
    // - Review HTTP access logs for metric submission requests
    // - Verify request payloads are valid JSON
    // - Check response codes (200 vs 4xx vs 5xx)

    // TODO 2: Validate metric format and content
    // - Parse submitted metrics against expected schema
    // - Check metric names, types, labels, and values
    // - Identify validation rule violations
}
```

```
// TODO 3: Trace metric flow through validation pipeline

// - Check if metrics pass cardinality limits

// - Verify label format and allowed values

// - Identify metrics rejected during validation


// TODO 4: Verify storage writes are successful

// - Check WAL entries for submitted metrics

// - Verify samples appear in storage blocks

// - Identify storage write failures or delays


// TODO 5: Generate diagnostic report with findings

// - Summarize ingestion health for the specified time range

// - List specific failures and their root causes

// - Recommend remediation steps for identified issues


return fmt.Errorf("ingestion diagnostics not implemented")

}

// CheckCardinalityExplosion analyzes label combinations for cardinality issues

func (id *IngestionDiagnostics) CheckCardinalityExplosion(ctx context.Context) error {

    // TODO 1: Query current series count per metric

    // - Group series by metric name

    // - Count unique label combinations

    // - Identify metrics with excessive cardinality


    // TODO 2: Analyze label contribution to cardinality

    // - For high-cardinality metrics, examine individual labels

    // - Calculate unique value count per label

    // - Identify labels with excessive diversity


    // TODO 3: Project cardinality growth trends

    // - Calculate new series creation rate
```

```
// - Estimate storage and memory impact

// - Predict when cardinality limits will be exceeded


// TODO 4: Generate cardinality optimization recommendations

// - Suggest label consolidation opportunities

// - Recommend cardinality limit adjustments

// - Identify metrics suitable for sampling


return fmt.Errorf("cardinality analysis not implemented")

}
```

Storage Diagnostics Framework:

```
// internal/diagnostics/storage_diagnostics.go                                GO

package diagnostics

import (
    "context"
    "fmt"
    "log/slog"
    "time"
)

type StorageDiagnostics struct {
    logger *slog.Logger
    // Add references to storage components
}

// DiagnoseQueryPerformance analyzes slow query execution

func (sd *StorageDiagnostics) DiagnoseQueryPerformance(ctx context.Context, queryString string) error {
    // TODO 1: Parse and analyze query complexity
    // - Break down query into component parts (selectors, functions, time range)
    // - Estimate series count and sample count requirements
    // - Identify potentially expensive operations (regex, aggregations)

    // TODO 2: Examine storage block access patterns
    // - Determine which blocks need to be read for the query
    // - Check for block overlaps requiring merge operations
    // - Identify hot vs cold data access patterns

    // TODO 3: Profile query execution phases
    // - Measure time spent in parsing, planning, execution
    // - Track memory allocation during query processing
    // - Identify bottleneck phases in the query pipeline

    // TODO 4: Analyze index utilization efficiency
```

```
// - Check if label indexes are being used effectively

// - Identify full scans that could benefit from better indexes

// - Measure index hit ratio for the query


// TODO 5: Generate performance optimization recommendations

// - Suggest query restructuring for better performance

// - Recommend index improvements or additions

// - Identify opportunities for result caching


return fmt.Errorf("query performance diagnostics not implemented")

}

// ValidateStorageConsistency checks for data corruption or inconsistencies

func (sd *StorageDiagnostics) ValidateStorageConsistency(ctx context.Context) error {

    // TODO 1: Verify WAL and storage block consistency

    // - Compare WAL entries with corresponding storage blocks

    // - Check that all WAL entries have been persisted to blocks

    // - Identify any samples that exist in WAL but not in blocks


    // TODO 2: Validate block internal consistency

    // - Verify block checksums match stored data

    // - Check timestamp ordering within blocks

    // - Validate sample count matches block metadata


    // TODO 3: Check index consistency with actual data

    // - Verify series index entries match storage blocks

    // - Check that all series in blocks have index entries

    // - Validate label indexes match series labels


    // TODO 4: Analyze compaction impact on data integrity

    // - Compare pre- and post-compaction sample counts

    // - Verify compaction preserved data accuracy
```

```
// - Check for any data loss during compaction operations

    return fmt.Errorf("storage consistency validation not implemented")
}

}
```

Language-Specific Debugging Hints

Go-Specific Debugging Techniques:

- Use `go tool pprof` to analyze CPU and memory profiles during high load periods
- Enable `GODEBUG=gctrace=1` to monitor garbage collection impact on performance
- Use `go tool trace` to analyze goroutine scheduling and coordination issues
- Implement custom `slog.Handler` to add correlation IDs to log entries
- Use `sync/atomic` for lock-free performance counters in hot paths

Performance Profiling Integration:

```
import _ "net/http/pprof" GO

// Add to main HTTP server

go func() {

    log.Println(http.ListenAndServe("localhost:6060", nil))

}()
```

Memory Leak Detection:

```
// Add to health checks GO

func checkMemoryGrowth(ctx context.Context) error {

    var m1, m2 runtime.MemStats

    runtime.ReadMemStats(&m1)

    runtime.GC()

    runtime.ReadMemStats(&m2)

    if m2.Alloc > m1.Alloc*2 {

        return fmt.Errorf("potential memory leak detected")

    }

    return nil

}
```

Milestone Checkpoint

After implementing debugging infrastructure:

1. **Run Health Checks:** `curl http://localhost:8080/health` should return comprehensive health status
2. **Generate Test Load:** Use synthetic metrics to trigger various failure modes
3. **Verify Diagnostic Tools:** Each diagnostic function should provide actionable insights
4. **Test Error Recovery:** Inject failures and verify system recovery behavior

Expected Debug Output Example:

```
{  
    "status": "degraded",  
    "checks": [  
        {  
            "name": "ingestion_pipeline",  
            "status": "healthy",  
            "last_checked": "2024-01-15T10:30:00Z",  
            "message": "check successful (took 45ms)"  
        },  
        {  
            "name": "storage_engine",  
            "status": "degraded",  
            "last_checked": "2024-01-15T10:30:00Z",  
            "message": "compaction behind schedule (took 3.2s)"  
        }  
    "timestamp": "2024-01-15T10:30:00Z"  
}
```

Future Extensions

Milestone(s): All milestones (future extensions build upon the complete metrics and alerting system to address scalability, reliability, and advanced functionality requirements)

The Evolution Principle: Building for Tomorrow's Challenges

Think of system extensions like urban planning for a growing city. When you design the initial road network, you don't just consider current traffic patterns—you plan for wider roads, additional lanes, and subway systems that might be needed as the population

grows. Similarly, our metrics and alerting system needs to accommodate future growth in data volume, user requirements, and operational complexity without requiring a complete architectural overhaul.

The key insight is that **extensibility isn't about predicting the future perfectly**—it's about creating flexible architectural joints and extension points that can accommodate unforeseen requirements. Just as a well-designed building has load-bearing walls that can support additional floors, our system has architectural foundations that can support distributed deployment, advanced analytics, and new data sources.

This section explores two categories of extensions: **scalability enhancements** that address growing data volumes and user loads, and **feature enhancements** that add new capabilities while leveraging the existing architectural foundation.

Current Architecture's Extension Points

Before diving into specific enhancements, it's important to understand how our current design accommodates future growth. The architecture includes several deliberate extension points:

Component Interface Abstraction: All major components implement well-defined interfaces (`TimeSeriesStorage`, `QueryProcessor`, `NotificationManager`). This allows swapping implementations without changing dependent components. For example, replacing the single-node storage engine with a distributed storage cluster requires only implementing the same `TimeSeriesStorage` interface.

Configuration-Driven Behavior: The `Config` structure and its nested configurations (`StorageConfig`, `AlertingConfig`) allow runtime behavior modification without code changes. New storage backends, notification channels, and query execution strategies can be enabled through configuration updates.

Plugin Architecture Foundations: The `NotificationChannel` system demonstrates a plugin pattern that can be extended to other components. New channel types (like Microsoft Teams or custom webhooks) can be added by implementing the notification interface without modifying core alerting logic.

Modular Component Design: The `ComponentCoordinator` manages component lifecycles independently, making it straightforward to add new components (like a machine learning inference engine) that integrate with existing data flows through well-defined interfaces.

Scalability Enhancements

Mental Model: From Corner Store to Global Supply Chain

Think of scalability enhancements like evolving from a small corner store to a global supply chain. The corner store serves its neighborhood well with simple operations: one cashier, local suppliers, and customers who walk in. But as demand grows beyond the neighborhood, you need multiple locations, regional distribution centers, sophisticated inventory management, and coordination between stores. The fundamental business logic remains the same—buying and selling goods—but the operational complexity increases dramatically.

Our current metrics system is like that efficient corner store: it handles significant load on a single machine with straightforward operations. Scalability enhancements transform it into a distributed system that can serve global monitoring needs while preserving the same core functionality and user experience.

Distributed Storage Architecture

The most critical scalability enhancement involves distributing the storage layer across multiple nodes to handle data volumes that exceed single-machine capacity. This transformation requires careful consideration of data partitioning, replication, and consistency guarantees.

Decision: Horizontal Partitioning Strategy

- **Context:** Single-node storage limits both data volume and query throughput. Time-series data has natural partitioning characteristics by metric name and time ranges.
- **Options Considered:** Hash-based partitioning by series ID, time-based partitioning by timestamp ranges, hybrid partitioning combining both approaches
- **Decision:** Hybrid partitioning with time-based primary partitioning and hash-based secondary partitioning
- **Rationale:** Time-based partitioning aligns with query patterns (most queries are time-range based) and enables efficient data lifecycle management. Hash-based secondary partitioning distributes load evenly across nodes within time windows.
- **Consequences:** Enables horizontal scaling and efficient range queries, but requires coordination for cross-partition queries and adds complexity for global operations.

The distributed storage architecture introduces several new components that extend our existing `StorageEngine`:

Component	Responsibility	Interface Changes
<code>PartitionManager</code>	Maps time ranges and series to storage nodes	Extends <code>TimeSeriesStorage</code> with partition awareness
<code>ReplicationCoordinator</code>	Manages data replication across nodes for durability	Adds replication configuration to <code>StorageConfig</code>
<code>ConsistencyManager</code>	Coordinates read/write consistency across replicas	Introduces consistency level parameters to storage operations
<code>ShardingRouter</code>	Routes queries to appropriate partition nodes	Extends <code>QueryProcessor</code> with distributed query planning

The distributed storage maintains backward compatibility by implementing the same `TimeSeriesStorage` interface, but internally coordinates operations across multiple nodes. Write operations become more complex as they must handle replication and consistency requirements:

1. **Write Coordination:** When `WriteSamples` is called, the `PartitionManager` determines which nodes should store each sample based on the series ID and timestamp
2. **Replication Protocol:** The `ReplicationCoordinator` ensures each write is replicated to the configured number of nodes before acknowledging success
3. **Consistency Management:** The `ConsistencyManager` enforces consistency levels (immediate, eventual, or quorum-based) based on the operation requirements

Query processing becomes more sophisticated as queries may span multiple partitions:

1. **Query Planning:** The `ShardingRouter` analyzes queries to determine which partitions contain relevant data
2. **Parallel Execution:** Subqueries are executed in parallel against relevant partition nodes
3. **Result Merging:** Results from multiple nodes are merged and deduplicated to produce the final response
4. **Cross-Partition Aggregation:** Aggregation functions are decomposed into node-local operations and global merge operations

High Availability and Fault Tolerance

Distributed deployment introduces new failure modes that require sophisticated fault tolerance mechanisms. The system must continue operating even when individual nodes fail, network partitions occur, or storage becomes temporarily unavailable.

Node Failure Detection and Recovery: The system implements a gossip-based failure detection protocol where nodes periodically exchange heartbeat information. When a node failure is detected, the `ReplicationCoordinator` triggers automatic

failover to replica nodes and initiates replication repair to restore the desired replication factor.

Split-Brain Prevention: Network partitions can create scenarios where multiple node groups believe they're the authoritative cluster. The system implements a quorum-based approach where operations require acknowledgment from a majority of nodes to prevent conflicting writes during partitions.

Data Consistency During Failures: The `ConsistencyManager` implements configurable consistency levels that balance availability and consistency based on operational requirements. During partial failures, the system can operate in degraded mode with reduced consistency guarantees rather than becoming completely unavailable.

Failure Scenario	Detection Method	Recovery Action	Impact on Operations
Single node failure	Gossip heartbeat timeout	Failover to replica, start replication repair	Temporary query latency increase
Network partition	Quorum loss detection	Minority partition becomes read-only	Write operations blocked in minority
Storage corruption	Checksum validation	Restore from replica, mark node for rebuild	Automatic transparent recovery
Coordinator failure	Leader election timeout	Promote new coordinator	Brief coordination pause

Load Balancing and Query Distribution

As query load increases, the system needs intelligent load balancing to distribute query processing across available resources. This involves both request routing and resource management at multiple levels.

Query Router Enhancement: The existing `QueryProcessor` is extended with a `DistributedQueryRouter` that implements several load balancing strategies. Round-robin distribution works well for uniform queries, while least-connections balancing is better for queries with varying execution times. Resource-aware routing considers CPU and memory utilization on target nodes.

Query Result Caching Distribution: The `QueryCache` becomes a distributed cache shared across query processor instances. This prevents redundant query execution when the same queries are submitted to different nodes. Cache invalidation becomes more complex as it must coordinate across multiple nodes when underlying data changes.

Adaptive Resource Allocation: The system monitors query execution patterns and automatically adjusts resource allocation. Long-running aggregation queries are routed to dedicated high-memory nodes, while simple point queries are handled by fast-response nodes. This prevents resource contention between different query types.

Auto-Scaling Infrastructure

Modern deployment environments support automatic scaling based on resource utilization and demand patterns. The enhanced system integrates with container orchestration platforms to automatically adjust capacity.

Demand-Based Scaling: The system exports internal metrics about query queue lengths, ingestion rates, and resource utilization. These metrics drive auto-scaling policies that add storage or query processing capacity when thresholds are exceeded. Scaling decisions consider both current load and historical patterns to anticipate demand.

Graceful Node Addition and Removal: When new nodes join the cluster, the `PartitionManager` gradually rebalances data to include the new capacity. Similarly, when nodes are removed (either manually or due to scaling down), their data is redistributed to remaining nodes before the node shuts down. This ensures continuous availability during scaling operations.

Geographic Distribution: For global deployments, the system supports multi-region deployment with intelligent routing to minimize latency. Users are automatically routed to their nearest region, while cross-region replication ensures data availability for disaster recovery.

Feature Enhancements

Mental Model: From Basic Calculator to Scientific Workstation

Think of feature enhancements like evolving from a basic calculator to a scientific workstation. The basic calculator handles arithmetic operations perfectly for simple tasks, but scientists need advanced functions like statistical analysis, graphing capabilities, and programmable operations. However, the fundamental mathematical principles remain the same—it's the sophistication of available operations that increases.

Our current metrics system provides solid foundational capabilities: collecting data, storing it efficiently, and providing basic visualization and alerting. Feature enhancements add advanced analytical capabilities, intelligent automation, and richer user experiences while building on the same core data model and architectural patterns.

Advanced Query Functions and Analytics

The current query engine provides basic aggregation functions like sum, average, and percentiles. Advanced analytics requires more sophisticated mathematical operations, statistical analysis, and time-series specific functions.

Statistical Functions Enhancement: The `AggregationFunction` interface is extended with advanced statistical operations that provide deeper insights into metric behavior patterns.

Function Category	New Functions	Use Cases
Trend Analysis	<code>trend()</code> , <code>linear_regression()</code> , <code>correlation()</code>	Identifying performance degradation patterns
Anomaly Detection	<code>zscore()</code> , <code>mad()</code> , <code>seasonal_decompose()</code>	Detecting unusual metric behavior
Forecasting	<code>holt_winters()</code> , <code>arima()</code> , <code>exponential_smoothing()</code>	Capacity planning and predictive alerting
Signal Processing	<code>moving_average()</code> , <code>exponential_decay()</code> , <code>fourier_transform()</code>	Noise reduction and frequency analysis

Advanced Query Language Features: The query parser is enhanced to support more complex expressions including mathematical operations, conditional logic, and subqueries. This enables sophisticated metric analysis without requiring external data processing tools.

For example, advanced queries can compute complex business metrics:

- Service level calculations: `(successful_requests / total_requests) * 100`
- Capacity utilization trends: `trend(cpu_usage[7d]) > 0.1` (detecting increasing utilization)
- Comparative analysis: `current_latency / baseline_latency[1w:offset 1w] > 1.5` (comparing to previous week)

Query Optimization for Complex Analytics: Advanced analytical functions often require processing large amounts of historical data. The query execution engine implements several optimizations specifically for analytical workloads:

1. **Incremental Computation:** Many statistical functions can be computed incrementally, updating results as new data arrives rather than recomputing from scratch
2. **Parallel Processing:** Complex queries are decomposed into parallel operations that can execute simultaneously across multiple CPU cores

3. **Intermediate Result Caching:** Expensive intermediate computations (like moving averages) are cached and reused across multiple queries
4. **Data Skipping:** Query planning identifies time ranges and series that don't affect query results, avoiding unnecessary data access

Machine Learning Integration

Machine learning capabilities transform reactive monitoring into proactive intelligence. Instead of just reporting what happened, the system can predict what will happen and automatically adapt to changing conditions.

Decision: Embedded vs. External ML Integration

- **Context:** Machine learning models can be embedded within the metrics system or integrated with external ML platforms. Each approach has different performance, complexity, and flexibility characteristics.
- **Options Considered:** Fully embedded ML engine, external ML service integration, hybrid approach with embedded inference and external training
- **Decision:** Hybrid approach with lightweight embedded inference and external model training
- **Rationale:** Embedded inference provides low-latency predictions for real-time alerting, while external training leverages specialized ML platforms for model development and heavy computation
- **Consequences:** Enables real-time ML-powered features with acceptable complexity, but requires model deployment and synchronization infrastructure

Anomaly Detection Engine: The system includes an embedded anomaly detection engine that continuously learns normal behavior patterns for each metric and identifies deviations. This goes beyond simple threshold-based alerting to detect subtle behavioral changes that might indicate underlying issues.

Component	Responsibility	Integration Point
AnomalyDetector	Identifies unusual metric patterns	Integrates with AlertEvaluator for ML-powered alerts
ModelRegistry	Manages trained models and versioning	Extends Config with model configuration
FeatureExtractor	Prepares metric data for ML inference	Integrates with QueryProcessor for feature computation
PredictionEngine	Generates forecasts for capacity planning	Provides new query functions for predictive analytics

Predictive Alerting: Traditional alerts fire after problems occur. Predictive alerting uses machine learning models to identify conditions that historically lead to incidents, enabling proactive intervention. For example, the system might learn that specific combinations of CPU usage, memory pressure, and request rate patterns typically precede service outages.

Adaptive Thresholds: Static alert thresholds often generate false alarms because they don't account for natural variations in system behavior (daily cycles, seasonal patterns, gradual growth trends). Machine learning models automatically adjust alert thresholds based on historical patterns, reducing alert fatigue while maintaining sensitivity to genuine issues.

Automated Root Cause Analysis: When incidents occur, machine learning models analyze correlation patterns across multiple metrics to suggest potential root causes. This accelerates incident response by helping engineers focus their investigation on the most likely culprits.

Enhanced Visualization and Dashboard Features

The current dashboard system provides basic line charts and panel layouts. Enhanced visualization adds sophisticated chart types, interactive analysis capabilities, and intelligent dashboard automation.

Advanced Chart Types: The chart renderer is extended with additional visualization options that better suit different types of metric analysis:

Chart Type	Best For	Key Features
Heatmaps	Distribution visualization	Shows metric value distributions over time
Scatter Plots	Correlation analysis	Reveals relationships between different metrics
Network Diagrams	Service dependency mapping	Visualizes service interactions and bottlenecks
Geographic Maps	Location-based metrics	Shows metric values across geographic regions
Sankey Diagrams	Flow analysis	Tracks resource flows through system components

Interactive Analysis Tools: Enhanced dashboards include interactive tools that enable ad-hoc analysis without requiring manual query construction:

- **Drill-down Navigation:** Click on chart elements to automatically generate detailed views of the selected time range or metric dimensions
- **Comparative Analysis:** Select multiple time ranges or metric variants for side-by-side comparison
- **Annotation Support:** Add contextual annotations to charts (deployment markers, incident notes, configuration changes)
- **Real-time Collaboration:** Multiple users can collaborate on dashboard analysis with shared cursors and comment threads

Intelligent Dashboard Automation: Machine learning analyzes dashboard usage patterns to provide intelligent recommendations:

1. **Automated Panel Suggestions:** Based on the metrics being viewed, the system suggests additional related metrics that might provide useful context
2. **Dynamic Layout Optimization:** Dashboard layouts automatically adapt based on the importance and correlation of displayed metrics
3. **Smart Alerting Integration:** Dashboards automatically highlight panels related to active alerts and suggest relevant diagnostic queries
4. **Usage-Based Optimization:** Frequently accessed dashboard elements are optimized for faster loading, while rarely used panels are deprioritized

Advanced Time Navigation: Enhanced time navigation tools support complex analysis scenarios:

- **Multi-timeline Comparison:** View the same metrics across different time periods simultaneously
- **Event Correlation:** Automatically align timelines with significant events (deployments, alerts, configuration changes)
- **Intelligent Time Selection:** Machine learning suggests optimal time ranges based on the type of analysis and historical patterns

Integration Ecosystem Expansion

Modern observability requires integration with a broad ecosystem of tools and services. Enhanced integration capabilities make the metrics system a central hub in the observability infrastructure.

Extended Data Source Support: The `MetricsIngestor` is enhanced to collect data from additional sources beyond traditional application metrics:

Data Source Category	Examples	Integration Method
Infrastructure Logs	Application logs, system logs, audit logs	Log parsing and metric extraction
Application Tracing	Distributed trace data, span metrics	Trace analysis and latency metrics
Business Events	User actions, transaction data, revenue metrics	Event stream processing
External APIs	Cloud service metrics, third-party SaaS data	Scheduled polling and API integration
IoT and Sensor Data	Temperature, pressure, location data	MQTT and sensor protocol support

Bi-directional Integration: The system not only collects data but also exports insights to other tools in the observability stack. This includes pushing alert information to incident management systems, sending capacity forecasts to auto-scaling controllers, and providing metric data to business intelligence platforms.

Workflow Integration: Enhanced notification channels integrate with workflow automation tools to trigger automated responses to alerts. For example, alerts about disk space can automatically trigger cleanup scripts, or performance degradation alerts can initiate auto-scaling actions.

Security and Compliance Enhancements

Enterprise deployment requires sophisticated security and compliance capabilities that go beyond basic authentication and authorization.

Audit Trail and Compliance Reporting: All system interactions are logged with comprehensive audit trails that support compliance with regulations like SOX, GDPR, and HIPAA. The audit system tracks who accessed what data, when changes were made, and what decisions were taken based on metric data.

Data Privacy and Anonymization: The system includes data privacy features that can automatically anonymize or pseudonymize sensitive metric data based on configurable policies. This enables analytics on sensitive data while protecting individual privacy.

Multi-tenancy and Isolation: Enhanced multi-tenancy support provides strict data isolation between different teams or customers sharing the same infrastructure. Each tenant has separate namespaces for metrics, dashboards, and alert rules with no possibility of cross-tenant data access.

Advanced Authentication Integration: The system integrates with enterprise identity providers (LDAP, Active Directory, SAML, OAuth) and supports advanced authentication features like multi-factor authentication, certificate-based authentication, and just-in-time access provisioning.

Performance and Scale Optimizations

Beyond horizontal scaling, numerous performance optimizations can significantly improve system efficiency and user experience.

Intelligent Data Compression: Advanced compression algorithms are tailored specifically for time-series data patterns, achieving better compression ratios than generic algorithms. The system automatically selects optimal compression strategies based on data characteristics and access patterns.

Query Result Streaming: Large query results are streamed to clients rather than being buffered entirely in memory. This enables analysis of massive datasets without overwhelming system resources or client applications.

Adaptive Sampling and Resolution: The system automatically adjusts metric collection frequency and storage resolution based on data importance and access patterns. Critical metrics maintain high resolution, while less important metrics are sampled at lower frequencies to save resources.

Edge Caching and CDN Integration: Dashboard assets and frequently accessed query results are cached at edge locations to minimize latency for global users. Intelligent cache invalidation ensures data freshness while maximizing cache hit rates.

Implementation Guidance

Technology Recommendations

The scalability and feature enhancements require careful technology choices that balance functionality, performance, and operational complexity:

Component	Simple Option	Advanced Option	Rationale
Distributed Storage	etcd + file-based sharding	Apache Cassandra / ScyllaDB	etcd for coordination, dedicated TSDB for heavy workloads
Message Queue	Go channels + persistence	Apache Kafka / NATS Streaming	Channels for simple cases, external queue for reliability
Machine Learning	Basic statistical functions	TensorFlow Serving / MLflow	Start simple, integrate ML platforms as needed
Service Discovery	Static configuration	Consul / etcd	Static config for small deployments, service discovery for dynamic environments
Load Balancing	HTTP reverse proxy	HAProxy / Envoy	Standard reverse proxy sufficient for most cases

Enhanced Architecture Structure

The extended system maintains the same core organization while adding new components for advanced features:

```

project-root/
  cmd/
    server/main.go           ← enhanced with feature flags
    migrate/main.go          ← database migration tool
  internal/
    coordinator/
      coordinator.go        ← enhanced with distributed coordination
      cluster_manager.go    ← NEW: cluster membership and coordination
      health_checker.go    ← enhanced with distributed health checking
    storage/
      engine.go              ← enhanced with partitioning awareness
      distributed/
        partition_manager.go
        replication_coordinator.go
        consistency_manager.go
      compaction/
        distributed_compaction.go ← NEW: cross-node compaction coordination
    query/
      processor.go           ← enhanced with advanced functions
      distributed_router.go   ← NEW: distributed query routing
      ml/
        anomaly_detector.go
        prediction_engine.go
        model_registry.go
    dashboard/
      server.go               ← enhanced with new chart types
      charts/
        heatmap.go             ← NEW: advanced visualization components
        network_diagram.go
        geographic_map.go
    alerting/
      evaluator.go            ← enhanced with ML-powered alerting
      adaptive_thresholds.go ← NEW: ML-based threshold adjustment
    integrations/
      log_parser.go           ← NEW: external system integrations
      trace_collector.go
      workflow_triggers.go
  pkg/
    ml/
      features.go             ← NEW: machine learning utilities
      inference.go             ← feature extraction for ML models
      inference.go             ← lightweight inference engine
  deployments/
    kubernetes/
      storage-cluster.yaml
      query-processors.yaml
      dashboard-service.yaml
    docker-compose/
      distributed-stack.yml   ← multi-node development environment
  migrations/
    001_initial_schema.sql
    002_add_partitioning.sql ← NEW: schema migration scripts

```

Distributed Storage Infrastructure

The distributed storage enhancement requires several new components that coordinate data placement and access across multiple nodes:

GO

```
// ClusterManager coordinates distributed operations across storage nodes

type ClusterManager struct {

    nodeID      string

    nodes       map[string]*NodeInfo

    coordinator *PartitionManager

    replication *ReplicationCoordinator

    gossip      *GossipProtocol

    logger      *slog.Logger

    stopCh     chan struct{}`

    mu         sync.RWMutex

}

// NewClusterManager creates a distributed coordination manager

func NewClusterManager(nodeID string, seedNodes []string, logger *slog.Logger) *ClusterManager {

    // TODO 1: Initialize gossip protocol for node discovery and failure detection

    // TODO 2: Create partition manager for data placement decisions

    // TODO 3: Initialize replication coordinator for data durability

    // TODO 4: Set up periodic maintenance tasks (failure detection, rebalancing)

    // TODO 5: Start gossip protocol and join cluster using seed nodes

}

// DistributedStorageEngine extends StorageEngine with cluster awareness

type DistributedStorageEngine struct {

    localEngine    *StorageEngine

    clusterManager *ClusterManager

    router        *ShardingRouter

    config        *DistributedStorageConfig

    logger        *slog.Logger

}

// WriteSamples distributes writes across appropriate nodes with replication

func (d *DistributedStorageEngine) WriteSamples(ctx context.Context, samples []Sample) error {

    // TODO 1: Group samples by target partition using consistent hashing
```

```
// TODO 2: For each partition, determine primary and replica nodes  
  
// TODO 3: Execute writes to primary nodes in parallel  
  
// TODO 4: Await replication confirmation based on consistency level  
  
// TODO 5: Return error if minimum replicas not achieved within timeout  
  
}
```

Machine Learning Integration Components

The ML integration provides intelligent analysis capabilities while maintaining simple deployment for basic use cases:

```
// AnomalyDetector identifies unusual patterns in metric data

type AnomalyDetector struct {
    models      map[string]*AnomalyModel
    features    *FeatureExtractor
    threshold   float64
    sensitivity float64
    logger     *slog.Logger
    mu         sync.RWMutex
}

// DetectAnomalies analyzes metric samples for unusual behavior

func (a *AnomalyDetector) DetectAnomalies(ctx context.Context, seriesID string, samples []Sample) ([]Anomaly, error) {

    // TODO 1: Extract features from sample data (trend, seasonality, variance)
    // TODO 2: Load or create model for this series
    // TODO 3: Compute anomaly score based on deviation from expected pattern
    // TODO 4: Apply threshold to determine which samples are anomalous
    // TODO 5: Return anomaly objects with confidence scores and explanations
}

// PredictionEngine generates forecasts for capacity planning

type PredictionEngine struct {

    models      map[string]*ForecastModel
    extractor   *FeatureExtractor
    cache       *PredictionCache
    logger     *slog.Logger
    mu         sync.RWMutex
}

// GenerateForecast predicts future values for a metric series

func (p *PredictionEngine) GenerateForecast(ctx context.Context, seriesID string, horizon time.Duration) (*Forecast, error) {

    // TODO 1: Check cache for recent forecast for this series
    // TODO 2: Retrieve historical data for model training/inference
}
```

```
// TODO 3: Extract time-series features (seasonality, trend, level)  
// TODO 4: Apply appropriate forecasting model (Holt-Winters, ARIMA, etc.)  
// TODO 5: Generate forecast with confidence intervals and cache result  
}
```

Advanced Visualization Components

Enhanced visualization requires sophisticated chart rendering and interactive analysis tools:

GO

```
// HeatmapRenderer creates heatmap visualizations for distribution analysis

type HeatmapRenderer struct {
    canvas      Canvas
    colorScale  ColorScale
    aggregator *DistributionAggregator
}

// RenderHeatmap generates heatmap visualization from metric data

func (h *HeatmapRenderer) RenderHeatmap(ctx context.Context, data []TimeSeries, config HeatmapConfig) (*ChartResult, error) {
    // TODO 1: Aggregate data into time/value buckets based on config
    // TODO 2: Calculate bucket densities and statistical measures
    // TODO 3: Apply color mapping based on density distribution
    // TODO 4: Render chart with proper axes, legends, and interactivity
    // TODO 5: Return chart data with metadata for client rendering
}

// InteractiveAnalyzer provides drill-down and exploration capabilities

type InteractiveAnalyzer struct {
    queryProcessor QueryProcessor
    correlator     *MetricCorrelator
    annotator     *AnnotationManager
    logger        *slog.Logger
}

// GenerateRelatedQueries suggests additional metrics for analysis

func (i *InteractiveAnalyzer) GenerateRelatedQueries(ctx context.Context, baseQuery string) ([]SuggestedQuery, error) {
    // TODO 1: Parse base query to extract metric names and labels
    // TODO 2: Find correlated metrics based on historical patterns
    // TODO 3: Generate contextually relevant query variations
    // TODO 4: Rank suggestions by relevance and usefulness
    // TODO 5: Return formatted queries with explanations
}
```

Milestone Checkpoints for Extensions

After implementing scalability enhancements, verify the following capabilities:

Distributed Storage Checkpoint:

- Start 3-node cluster: `go run cmd/server/main.go --cluster-mode --node-id=node1 --seed-nodes=node2:8080,node3:8080`
- Verify node discovery: `curl http://localhost:8080/api/cluster/nodes` should show all 3 nodes
- Test data distribution: ingest metrics and verify they're stored across multiple nodes
- Test failover: stop one node and verify queries still work with slight latency increase
- Expected behavior: System continues operating with one node failure, automatically rebalances data

Machine Learning Checkpoint:

- Enable anomaly detection: add `ml.anomaly_detection.enabled=true` to config
- Ingest normal baseline data for 24 hours with consistent patterns
- Inject anomalous data points (values 3x higher than normal)
- Verify anomaly alerts: `curl http://localhost:8080/api/alerts` should show ML-generated alerts
- Expected behavior: System learns normal patterns and alerts on deviations without manual thresholds

Advanced Visualization Checkpoint:

- Create heatmap dashboard: add heatmap panel to existing dashboard configuration
- Verify interactive features: click on heatmap cells to drill down to detailed time series
- Test correlation analysis: select multiple metrics and generate correlation matrix
- Expected behavior: Rich visualizations render correctly with interactive navigation

Common Enhancement Pitfalls

⚠ Pitfall: Premature Distributed Deployment Many teams attempt distributed deployment before understanding their actual scale requirements. Distributed systems add significant operational complexity—multiple failure modes, complex debugging, consistency challenges. **Why it's wrong:** The coordination overhead can actually reduce performance for small-scale deployments. **How to fix:** Implement and validate single-node optimizations first. Only distribute when single-node capacity is genuinely exceeded (>100GB/day ingestion or >1000 queries/second).

⚠ Pitfall: Over-Complex ML Integration Teams often integrate sophisticated ML platforms for simple use cases that could be solved with basic statistical methods. **Why it's wrong:** Complex ML infrastructure requires specialized expertise, adds deployment dependencies, and introduces model training/versioning overhead. **How to fix:** Start with simple statistical functions (moving averages, standard deviation thresholds) and only add ML complexity when clear value is demonstrated.

⚠ Pitfall: Feature Creep in Visualization Enhanced dashboards can become overwhelmingly complex with too many chart types, interactive features, and configuration options. **Why it's wrong:** Complex interfaces reduce usability and increase maintenance burden. **How to fix:** Follow progressive disclosure principles—start with simple, common visualizations and add advanced features only when specific use cases are identified.

⚠ Pitfall: Ignoring Backward Compatibility Enhancements often break existing configurations, APIs, or data formats in pursuit of new features. **Why it's wrong:** Forces users to rewrite configurations and dashboards during upgrades, creating adoption friction. **How to fix:** Design extensions that enhance existing interfaces rather than replacing them. Provide migration tools and maintain compatibility layers for deprecated features.

⚠ Pitfall: Inadequate Performance Testing Enhanced features often have different performance characteristics that aren't validated under realistic load. **Why it's wrong:** New features may work well in development but degrade system performance in

production. **How to fix:** Implement performance benchmarks for each enhancement and validate them with realistic data volumes and query patterns before deployment.

Integration Testing Strategy

Comprehensive integration testing ensures enhancements work correctly with existing functionality:

Distributed System Testing: Use containerized environments to simulate multi-node deployments with realistic network latency and failure scenarios. Test network partitions, node failures, and data consistency under various failure modes.

ML Model Validation: Create synthetic datasets with known anomaly patterns to validate ML model accuracy. Test model performance degradation over time and automatic retraining capabilities.

Cross-Feature Integration: Test how advanced visualizations perform with ML-generated data, how distributed queries work with enhanced analytics functions, and how notification integrations handle increased alert volume from ML-powered alerting.

Performance Regression Testing: Establish performance baselines before implementing enhancements and validate that new features don't degrade core system performance. Pay particular attention to query latency impact from advanced analytics functions.

The extensibility design ensures these enhancements integrate seamlessly with the existing architecture while providing clear upgrade paths for organizations with growing monitoring requirements.

Glossary

Milestone(s): All milestones (this comprehensive terminology reference supports understanding across the entire metrics and alerting system)

The Dictionary Metaphor: Building a Shared Language

Think of this glossary as a specialized dictionary for our metrics and alerting system. Just as doctors, lawyers, and engineers each have their own professional vocabulary that enables precise communication within their field, our system has specific terminology that carries exact technical meanings. When we say "cardinality explosion," every team member should immediately understand we're talking about a performance problem caused by too many unique label combinations, not just "lots of data."

This shared vocabulary serves three critical purposes: it eliminates ambiguity in technical discussions, it helps new team members quickly understand system concepts, and it ensures consistent naming across code, documentation, and monitoring. Just as a medical dictionary defines "tachycardia" as "heart rate above 100 beats per minute" rather than just "fast heartbeat," our glossary provides precise definitions that translate directly to measurable system behaviors.

The terms in this glossary fall into several categories: data model concepts that define what we store and how, architectural patterns that describe how components interact, operational procedures that define how the system behaves, and troubleshooting vocabulary that helps identify and resolve issues. Each definition includes not just what the term means, but why it matters and how it impacts system design decisions.

Core Data Model Terms

Abstract Syntax Tree (AST): A hierarchical representation of a parsed query structure that captures the logical organization of operations, functions, and data references without concern for the specific textual syntax used to express them. The AST enables query optimization, validation, and execution planning by providing a standardized internal representation that separates query semantics from parsing details.

Alert Instance: A specific occurrence of an alert rule evaluation that represents the current state, value, and notification history for a particular combination of alert rule and time series labels. Each instance tracks its lifecycle through states like pending, firing, and resolved, maintaining timestamps for state changes and a log of notification attempts to enable proper alert management and debugging.

Alert Rule: A configuration that defines the conditions under which alerts should be triggered, specifying a metric query expression, comparison operator, threshold value, evaluation interval, and notification settings. Alert rules serve as templates that generate alert instances when their conditions are met, enabling automated monitoring of system health and business metrics.

Alert State: An enumerated value representing the current lifecycle phase of an alert instance, including inactive (condition not met), pending (condition met but duration requirement not satisfied), firing (condition met for required duration), resolved (condition no longer met), and silenced (notifications suppressed). State transitions follow a defined state machine that ensures proper notification behavior and prevents alert flapping.

Alert State Machine: A finite state automaton that governs transitions between alert lifecycle phases, defining valid state changes based on evaluation results, duration requirements, and administrative actions. The state machine ensures consistent alert behavior, prevents invalid transitions like jumping directly from inactive to firing, and manages notification timing to avoid spam while ensuring critical alerts are communicated promptly.

Cardinality: The number of unique time-series combinations in a dataset, calculated as the product of all possible label value combinations for each metric name. High cardinality can severely impact query performance and memory usage, as each unique combination requires separate storage and indexing, making cardinality management crucial for system scalability.

Cardinality Explosion: A performance-degrading condition where the number of unique time-series combinations grows exponentially due to high-variability labels like user IDs, request IDs, or timestamps being used as label values. This explosion can overwhelm storage systems, consume excessive memory during queries, and make the system unresponsive, requiring careful label design and cardinality monitoring.

Dashboard: A structured collection of visualization panels that displays metric data in charts, graphs, and tables, organized according to a saved configuration that specifies layout, queries, time ranges, and refresh intervals. Dashboards serve as the primary interface for observing system behavior and investigating issues, providing both overview perspectives and detailed drill-down capabilities.

Histogram Buckets: Predefined ranges that group metric observations into discrete intervals for distribution analysis, enabling calculation of percentiles, averages, and counts across different value ranges. Each bucket maintains a count of observations falling within its range, allowing statistical analysis of latency distributions, request sizes, and other measured quantities without storing individual sample values.

Labels: Key-value pairs attached to metrics that provide dimensional metadata enabling filtering, grouping, and aggregation across different facets of the data. Labels transform flat metric streams into multi-dimensional datasets, allowing queries like "CPU usage for service=web, environment=production" while requiring careful management to prevent cardinality explosion.

Metric: A named measurement that captures quantitative information about system behavior, application performance, or business processes over time. Metrics include metadata like type (counter, gauge, histogram), labels for dimensional filtering, and a stream of timestamped sample values that enable trend analysis and alerting.

Panel: An individual visualization component within a dashboard that displays metric data as charts, graphs, tables, or other visual formats according to configured queries, time ranges, and display options. Panels can be resized, repositioned, and configured independently, allowing flexible dashboard layouts that match specific monitoring needs.

Sample: A single timestamped measurement value representing the state of a metric at a specific point in time, consisting of a numeric value and a timestamp that enables time-series analysis. Samples are the atomic units of metric data, and their efficient storage and retrieval determines system performance and query capabilities.

Series ID: A unique identifier for a specific metric-labels combination computed as a hash of the metric name and sorted label pairs, enabling efficient storage indexing and query optimization. Series IDs allow the system to quickly locate related samples without expensive string comparisons and provide a stable reference for time-series data across compaction and retention operations.

Time-Series Data: Sequences of timestamped measurements that capture how values change over time, enabling trend analysis, forecasting, and anomaly detection. Time-series data exhibits temporal locality patterns that can be exploited for efficient storage compression and query optimization.

Storage and Query Terms

Block-Based Storage: An organizational strategy that groups time-series samples into immutable, time-windowed storage blocks that can be efficiently compressed, indexed, and queried. This approach enables effective compaction strategies, optimizes disk I/O patterns, and supports retention policies by allowing entire blocks to be deleted when they exceed configured age limits.

Compaction: The process of merging and downsampling stored data blocks to improve query performance, reduce storage overhead, and maintain system efficiency over time. Compaction combines multiple small blocks into larger ones, applies compression algorithms, and can downsample high-resolution data to lower resolutions for long-term storage while preserving statistical properties.

Counter Reset: A situation where a counter metric's value decreases, typically indicating a process restart, metric redefinition, or data collection issue. Counter resets require special handling in rate calculations to avoid negative spikes and ensure accurate trend analysis, often involving reset detection algorithms and rate calculation adjustments.

Downsampling: The process of reducing data resolution by aggregating high-frequency samples into lower-frequency summaries while preserving important statistical properties like averages, maximums, and percentiles. Downsampling enables long-term data retention by reducing storage requirements for older data while maintaining sufficient detail for historical analysis.

Query Execution Pipeline: A multi-phase process that transforms query strings into results through parsing, query planning, storage access, data aggregation, and result formatting. The pipeline enables optimization opportunities at each stage, supports caching strategies, and provides clear error isolation for debugging query performance issues.

Retention Policy: Automated rules that manage data lifecycle by deleting or downsampling data that exceeds configured age limits, preventing unlimited storage growth while preserving important historical information. Retention policies must balance storage costs with analytical needs, often implementing tiered strategies that keep high-resolution recent data and low-resolution historical data.

Streaming Aggregation: A processing technique that computes aggregation functions incrementally as data flows through the system, without requiring the entire dataset to be buffered in memory. This approach enables real-time analytics on large datasets and reduces memory pressure during query execution, particularly important for high-cardinality queries.

Timestamp Alignment: The process of synchronizing samples from multiple time series to common time intervals to enable mathematical operations and comparisons between series. Alignment handles cases where different metrics are sampled at different frequencies or times, using interpolation or alignment functions to create consistent time grids.

Write-Ahead Log (WAL): A durability mechanism that persists metric samples to disk before acknowledging writes, ensuring data survival across system crashes and enabling recovery by replaying logged operations. The WAL provides atomicity guarantees for batch writes and supports consistent recovery semantics essential for reliable metric storage.

Query and Analysis Terms

Aggregation Function: A mathematical operation that combines multiple time-series values into summary statistics like sum, average, maximum, minimum, or percentiles across specified dimensions or time ranges. Aggregation functions enable dashboard visualizations and alert evaluations that operate on grouped data rather than individual samples.

Cache Entry: A stored query result with associated metadata including the original query, result data, creation timestamp, access statistics, and expiration information. Cache entries enable query result reuse to improve dashboard performance and reduce storage system load for frequently accessed data.

Counter Semantics: The behavioral rules governing counter-type metrics, which represent monotonically increasing cumulative values that can only increase or reset to zero. Counter semantics require special handling for rate calculations, reset detection, and ensuring that derived metrics like queries per second correctly handle counter resets.

Gauge Semantics: The behavioral rules for gauge-type metrics, which represent point-in-time values that can increase or decrease freely, such as memory usage, queue lengths, or temperature readings. Gauge semantics allow direct value comparisons and mathematical operations without the reset-handling complexity required for counters.

Label Explosion: A condition where the number of unique label combinations grows uncontrollably, typically caused by using high-cardinality values like user IDs, request IDs, or IP addresses as label values. Label explosion can severely degrade query performance and exhaust system memory, requiring careful label design and monitoring.

Lexical Analysis: The process of breaking query strings into meaningful tokens like metric names, operators, functions, and literal values, serving as the first phase of query parsing. Lexical analysis handles syntax elements like whitespace, quotes, and escape sequences while identifying token types that guide subsequent parsing phases.

Query Result Caching: A performance optimization technique that stores expensive query results in memory with associated cache keys, expiration times, and invalidation rules to avoid recomputing identical queries. Caching reduces storage system load and improves dashboard responsiveness, particularly for complex aggregation queries over large time ranges.

Operational and Monitoring Terms

Adaptive refresh rates: Dynamically adjusted update frequencies for dashboard panels based on system performance, data velocity, and user interaction patterns. Adaptive refresh rates prevent system overload during high-traffic periods while ensuring timely updates for critical monitoring scenarios.

Alert Flapping: Rapid oscillation between firing and resolved states caused by threshold values that are too close to normal fluctuations in metric values. Flapping creates notification storms that desensitize users to real problems and can be prevented through proper threshold selection, hysteresis, and duration requirements.

Backpressure: A flow control mechanism that prevents system overload by slowing down or rejecting input when downstream components cannot keep up with the incoming data rate. Backpressure protects system stability during traffic spikes but must be carefully implemented to avoid cascading delays.

Circuit Breaker: A fault tolerance pattern that prevents cascade failures by automatically stopping requests to failing services after a configured failure threshold is exceeded. Circuit breakers provide fast-fail behavior during outages and can automatically recover when the downstream service becomes healthy again.

Dashboard Templating: A variable substitution system that enables creation of reusable parameterized dashboards where users can select different services, environments, or time ranges without creating separate dashboard configurations. Templating reduces configuration maintenance and enables consistent monitoring across similar components.

Duration-Based Evaluation: An alert evaluation strategy that requires conditions to persist for a specified time period before transitioning to firing state, preventing transient spikes from triggering false alerts. Duration-based evaluation filters noise but introduces delay in alert notification, requiring balanced configuration.

Exponential Backoff: A retry delay strategy where the wait time increases exponentially with each failed attempt, helping to reduce system load during outages while still attempting recovery. Exponential backoff prevents retry storms that could worsen system problems during failures.

Graceful Degradation: System design that maintains partial functionality during failures rather than complete unavailability, such as displaying cached dashboard data when query systems are unavailable. Graceful degradation improves user experience during

outages and maintains basic monitoring capabilities.

Grid Layout System: A responsive positioning framework for organizing dashboard panels that automatically adjusts to different screen sizes and supports drag-and-drop rearrangement. Grid layouts provide consistent visual organization while enabling flexible dashboard customization.

Health Checking: Periodic verification of component availability and functionality through automated tests that validate both internal state and external dependencies. Health checks enable automated failure detection, load balancer routing decisions, and system health dashboards.

Horizontal Partitioning: A data distribution strategy that splits datasets across multiple storage nodes based on partitioning keys like metric name hash or time ranges. Horizontal partitioning enables system scaling by distributing load and storage requirements across multiple machines.

Incremental Updates: A data transmission optimization that sends only new or changed information rather than complete datasets, reducing network bandwidth and improving dashboard responsiveness. Incremental updates require careful state management to ensure data consistency and handle missed updates.

Message Templating: A customizable formatting system for notification content that allows channel-specific message generation using variables from alert context like metric values, labels, and timestamps. Templates enable consistent notification formatting while supporting different communication channels.

Notification Routing: A rule-based system that determines which communication channels receive specific alert notifications based on factors like alert severity, time of day, escalation policies, and team assignments. Proper routing ensures the right people receive timely notifications without overwhelming any individual or channel.

Panel Subscription: A client-side registration system that tracks which dashboard panels a user is viewing and their required refresh rates, enabling efficient server-side update delivery through WebSocket connections. Subscriptions prevent unnecessary data transmission for off-screen panels.

Pull-Based Collection: A metrics gathering approach where the monitoring system actively scrapes metric data from application endpoints at regular intervals, providing centralized control over collection timing and enabling service discovery integration. Pull-based collection simplifies application configuration but requires network connectivity and service registration.

Push-Based Collection: A metrics transmission method where applications actively send metric data to the monitoring system, enabling immediate data delivery and supporting scenarios where pull-based access is not feasible. Push-based collection provides lower latency but requires applications to handle delivery failures and retries.

WebSocket Connections: Persistent bidirectional communication channels between dashboard clients and servers that enable real-time data updates without polling overhead. WebSocket connections support efficient live dashboard updates but require connection management and reconnection handling for reliability.

Error Handling and Reliability Terms

Cascade Failure: A failure propagation pattern where problems in one system component trigger failures in dependent components, potentially leading to complete system unavailability. Cascade failures can be prevented through circuit breakers, timeouts, bulkheads, and other fault isolation techniques.

Correlation IDs: Unique identifiers attached to requests that enable tracing of operations across multiple system components and log aggregation for debugging distributed system interactions. Correlation IDs are essential for troubleshooting complex failures that span multiple services.

Delivery Confirmation: Verification mechanisms that ensure alert notifications were successfully transmitted to their intended destinations, including retry logic for failed deliveries and escalation procedures for persistent failures. Delivery confirmation is critical for ensuring that important alerts reach their intended recipients.

Error Injection: A testing technique that deliberately introduces failures into system components to validate error handling code paths and system resilience. Error injection helps identify weak points in fault tolerance and ensures that error handling code actually works as intended.

Notification Queue: A background processing system that manages alert notification delivery with features like retry logic, rate limiting, and priority handling to ensure reliable message delivery while preventing notification storms. Queues provide delivery guarantees and help manage notification volume during incident situations.

Rate Limiting: A throttling mechanism that prevents notification storms by limiting the number of alerts that can be sent through specific channels within defined time windows. Rate limiting protects communication channels from overload while ensuring critical alerts still get delivered.

State Persistence: Durability mechanisms that ensure alert states, dashboard configurations, and other critical system data survive system restarts and failures. State persistence enables proper alert lifecycle management and prevents data loss during system maintenance or unexpected outages.

Testing and Development Terms

Integration Testing: Testing approach that validates component interactions using real dependencies and network communication to ensure the complete system works correctly end-to-end. Integration testing catches issues that unit tests miss, such as serialization problems, network timeouts, and protocol mismatches.

Milestone Validation Checkpoints: Structured verification procedures that confirm major system capabilities are working correctly after each development phase, including specific test cases, expected behaviors, and success criteria. Checkpoints help ensure progress is solid before moving to more complex features.

Mock Dependencies: Test doubles that simulate the behavior of external components like databases, network services, and file systems, enabling isolated unit testing and controlled failure scenario testing. Mocks allow testing error conditions that would be difficult or dangerous to reproduce with real systems.

Performance Validation: Testing procedures that verify system behavior under realistic load conditions, including throughput measurement, latency analysis, and resource utilization monitoring. Performance validation ensures the system can handle production workloads and identifies scalability bottlenecks.

Test Fixtures: Reusable test data sets, configuration files, and utility functions that provide consistent testing environments and reduce test setup complexity. Fixtures enable reliable test reproduction and make it easier to create comprehensive test coverage.

Test Harness: Infrastructure for setting up complete system instances in test environments, including component coordination, configuration management, and cleanup procedures. Test harnesses enable integration testing and provide controlled environments for manual testing and debugging.

Unit Testing: Testing individual components in isolation with mocked dependencies to validate specific functionality, error conditions, and edge cases. Unit testing provides fast feedback during development and ensures individual components work correctly before integration.

Advanced and Extension Terms

Anomaly Detection: Machine learning techniques that identify unusual patterns in metric data that deviate significantly from historical behavior, enabling proactive alerting for problems that don't have known thresholds. Anomaly detection can identify subtle issues that traditional threshold-based alerting would miss.

Auto-Scaling: Automatic adjustment of system capacity based on demand metrics like CPU usage, memory pressure, or queue lengths, enabling cost optimization while maintaining performance. Auto-scaling requires careful configuration to avoid oscillation and must integrate with deployment systems.

Consistency Level: Guarantees about data consistency across replicas in distributed systems, ranging from eventual consistency (data will be consistent eventually) to strong consistency (all reads return the most recent write). Consistency levels involve trade-offs between performance and data accuracy.

Feature Extraction: The process of deriving meaningful variables from raw metric data for machine learning analysis, such as trend calculations, seasonal decomposition, and statistical summaries. Feature extraction transforms time-series data into formats suitable for classification and prediction algorithms.

Gossip Protocol: A decentralized communication method for sharing cluster membership information and failure detection where each node periodically exchanges state information with a subset of other nodes. Gossip protocols provide eventual consistency and fault tolerance without requiring central coordination.

Heatmap Visualization: Color-coded matrix displays that show data density, value distributions, or correlation patterns across two-dimensional parameter spaces. Heatmaps are particularly useful for showing request latency distributions over time or error rates across different services.

Horizontal Scaling: System architecture that handles increased load by adding more servers rather than upgrading existing hardware, enabling theoretically unlimited capacity growth. Horizontal scaling requires careful design of data partitioning, load balancing, and coordination protocols.

Model Registry: Centralized management system for trained machine learning models, including version control, deployment tracking, and performance monitoring. Model registries enable systematic management of ML-powered features like anomaly detection and forecasting.

Multi-Tenancy: System design that supports multiple isolated user groups on shared infrastructure while maintaining data isolation, performance guarantees, and security boundaries. Multi-tenancy reduces operational overhead while requiring careful resource management and access control.

Predictive Alerting: Alert systems that use forecasting models to identify potential future problems before they occur, enabling proactive intervention rather than reactive response. Predictive alerting can prevent outages but requires sophisticated modeling and careful false positive management.

Quorum-Based Approach: Decision-making systems that require agreement from a majority of participants to proceed with operations, preventing split-brain scenarios in distributed systems. Quorum systems provide strong consistency guarantees but require careful configuration to handle network partitions.

Replication Factor: The number of copies of each data item stored across different nodes in a distributed system, providing durability and availability at the cost of storage overhead and consistency complexity. Higher replication factors improve fault tolerance but increase coordination requirements.

Split-Brain Prevention: Architectural patterns and protocols that avoid scenarios where network partitions cause different parts of a distributed system to make conflicting decisions. Prevention techniques include quorum systems, witness nodes, and careful leader election protocols.

Performance and Optimization Terms

Compression Optimization: Tailoring data compression algorithms to specific data patterns found in time-series metrics, such as delta encoding for counter values and dictionary encoding for repetitive label values. Optimized compression significantly reduces storage requirements and improves query performance.

Edge Caching: Performance optimization that stores frequently accessed data closer to users or applications to reduce latency and backend load. Edge caching is particularly effective for dashboard data and metric metadata that are accessed repeatedly.

Load Balancing: Distribution of incoming requests across multiple servers to optimize resource utilization, minimize response time, and prevent individual server overload. Load balancing requires careful selection of algorithms and health checking to ensure optimal distribution.

Query Optimization: Techniques for improving query performance through better execution planning, index utilization, result caching, and algorithm selection. Query optimization is crucial for maintaining dashboard responsiveness and alert evaluation performance as data volumes grow.

Acronyms and Abbreviations

ADR: Architecture Decision Record - a structured format for documenting important design decisions with context, options, rationales, and consequences.

API: Application Programming Interface - the defined methods and data formats that components use to communicate with each other.

AST: Abstract Syntax Tree - the hierarchical representation of parsed queries used for optimization and execution.

HTTP: HyperText Transfer Protocol - the communication protocol used for web-based interfaces and REST APIs.

JSON: JavaScript Object Notation - the data serialization format used for configuration files, API responses, and data exchange.

REST: Representational State Transfer - an architectural style for designing web APIs using standard HTTP methods.

SLA: Service Level Agreement - formal commitments about system availability, performance, and reliability.

SLI: Service Level Indicator - specific metrics that measure system performance against defined objectives.

SLO: Service Level Objective - target values or ranges for service level indicators that define acceptable performance.

WAL: Write-Ahead Log - durability mechanism that records intended operations before executing them to enable recovery.

YAML: YAML Ain't Markup Language - human-readable data serialization format commonly used for configuration files.

Implementation Guidance

The terminology in this glossary directly maps to specific implementation patterns and naming conventions used throughout the codebase. Each term has been carefully chosen to reflect industry standards while maintaining consistency with our specific architectural decisions.

A. Naming Convention Mapping

The glossary terms serve as the foundation for consistent naming across all system components. When implementing any feature, developers should use these exact terms in variable names, function names, and documentation to maintain clarity and searchability.

Concept Category	Implementation Pattern	Example Usage
Data Structures	Direct mapping to struct/type names	<code>AlertState</code> , <code>MetricType</code> , <code>Labels</code>
Operations	Verb-noun pattern using glossary terms	<code>ValidateCardinality()</code> , <code>CompactBlocks()</code>
Constants	ALL_CAPS with underscore separation	<code>ALERT_STATE_FIRING</code> , <code>METRIC_TYPE_COUNTER</code>
Configuration	Nested structures following glossary hierarchy	<code>StorageConfig.RetentionPeriod</code>

B. Documentation Standards

Every public function, method, and type should include documentation that references appropriate glossary terms to ensure consistent understanding across the codebase. This creates a self-reinforcing system where code documentation strengthens terminology understanding.

C. Error Message Consistency

Error messages throughout the system should use glossary terminology to help users quickly understand problems and find relevant documentation. For example, "cardinality explosion detected" is more helpful than "too many time series" because it points to specific glossary entries and troubleshooting procedures.

D. Monitoring and Alerting Vocabulary

The operational procedures and alert definitions should use glossary terms in their names and descriptions, creating consistency between system behavior and documentation. This makes it easier to correlate alerts with architectural documentation and debugging procedures.

E. Team Communication Guidelines

During code reviews, design discussions, and incident response, team members should use glossary terminology to ensure precise communication. When someone uses a term differently than the glossary definition, it signals a need to either update the glossary or clarify the usage to maintain consistency.

The glossary serves as both a reference document and a communication contract that enables precise technical discussions and consistent system implementation across the entire metrics and alerting platform.