

Prometheus-Like Metrics Collection System: Design Document

Overview

A distributed metrics collection system that scrapes time-series data from service endpoints, stores it with efficient compression, and provides a query engine for monitoring and alerting. The key architectural challenge is handling high-cardinality labeled metrics at scale while maintaining low latency for queries and minimal storage overhead.

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): This section establishes the foundation for Milestones 1-4 by explaining why we need a metrics collection system and what technical challenges it must solve.

Modern distributed systems have transformed from simple monolithic applications into complex networks of microservices, containers, and cloud-native components. This architectural evolution has created an unprecedented need for comprehensive observability — the ability to understand system behavior, performance, and health from the outside by examining the data it emits. At the heart of observability lies metrics collection: the systematic gathering, storage, and analysis of quantitative measurements that tell the story of how our systems are performing over time.

The challenge we face is not merely collecting numbers from our applications. Any system can log metrics to files or send them to a database. The real complexity emerges when we consider the scale, velocity, and dimensional richness of modern metrics data. A typical microservices deployment might generate millions of time series, each representing a unique combination of metric name and label values, with new data points arriving every few seconds. These metrics must be stored efficiently, queried quickly, and remain available even as the underlying infrastructure scales and evolves.

Existing monitoring solutions often fall short in one or more critical areas: they either cannot handle the scale of modern systems, lack the flexibility to model complex multi-dimensional data, or require architectural compromises that limit their effectiveness. The goal of this project is to build a Prometheus-like metrics collection system that addresses these shortcomings while providing the foundation for robust observability in distributed environments.

The Observatory Mental Model

To understand the architecture and challenges of a metrics collection system, consider the analogy of a **weather monitoring network**. Imagine you are tasked with building a system to monitor weather conditions across an entire continent. You need to track temperature, humidity, wind speed, and atmospheric pressure at thousands of locations, with measurements taken every few minutes, 24 hours a day.

Your weather monitoring network would require several key components, each with specific responsibilities and challenges:

Weather Stations (Metrics Sources): Thousands of automated weather stations scattered across the continent continuously measure environmental conditions. Each station has a unique identifier (location) and can measure multiple phenomena simultaneously. The stations must be robust, autonomous, and capable of operating in diverse conditions. In our metrics system, these weather stations correspond to **application instances, services, and infrastructure components** that expose metrics through HTTP endpoints.

Data Collectors (Scrape Engine): Rather than having each weather station independently transmit its data, you deploy a network of data collectors that periodically visit stations to retrieve measurements. This pull-based approach offers several advantages: collectors can detect when stations are offline, they can apply consistent formatting and validation, and they can handle network issues gracefully. In our system, the **scrape engine** acts as these data collectors, periodically fetching metrics from configured targets.

Central Archives (Time Series Storage): All collected measurements must be stored in a central archive system that can handle the massive volume of time-stamped data points. The archive must compress data efficiently (weather measurements follow predictable patterns), provide fast access for analysis, and automatically manage data retention as storage fills up. Our **time series storage engine** serves this archival function, using specialized compression techniques designed for temporal data.

Research Interface (Query Engine): Scientists and analysts need to query the archived weather data to identify patterns, trends, and anomalies. They might ask questions like "What was the average temperature in the Pacific Northwest during July?" or "Show me all locations where humidity exceeded 90% in the past week." The query interface must support complex filtering, aggregation, and time-based operations. Our **PromQL query engine** provides this analytical capability for metrics data.

This mental model illuminates several key insights about metrics collection systems:

Pull vs Push Architecture: Just as weather stations don't individually transmit data to every interested party, metrics systems benefit from a pull-based architecture where a central collector retrieves data from sources. This approach provides better failure detection, reduces network overhead, and enables consistent data formatting.

Dimensional Data Challenges: Weather measurements aren't just numbers — they have dimensions (location, altitude, measurement type, instrument ID). Similarly, modern metrics are multi-dimensional, with labels that provide rich context. Managing the combinatorial explosion of unique dimension combinations becomes a critical scalability challenge.

Time Series Characteristics: Weather data exhibits temporal patterns that enable efficient compression — temperatures change gradually, measurements are taken at regular intervals, and similar conditions tend to cluster in time. Metrics data shares these characteristics, allowing specialized storage techniques that wouldn't work for general-purpose databases.

Query Pattern Optimization: Weather researchers typically ask time-based questions about recent data, with occasional historical analysis. Metrics queries follow similar patterns — most focus on recent time windows, with aggregation across multiple dimensions. Storage and indexing strategies can optimize for these access patterns.

The weather monitoring analogy also reveals why building an effective metrics collection system is challenging. The scale is enormous (potentially millions of time series), the data arrives continuously, storage efficiency is crucial, and

query performance must remain fast even as the dataset grows. These constraints shape every architectural decision in our system.

Existing Solutions Analysis

The metrics collection landscape includes several established approaches, each with distinct architectural philosophies and trade-offs. Understanding these existing solutions helps clarify the design space and motivates our architectural choices.

Push-Based Systems (StatsD, Graphite, InfluxDB):

Push-based systems follow a model where application instances actively send their metrics to a central collector. Applications use client libraries to emit metrics via UDP packets (StatsD) or HTTP requests, and a central aggregation service receives, processes, and stores the data.

Aspect	Advantages	Disadvantages
Network Overhead	Applications control transmission timing	High network traffic from many sources
Failure Detection	Immediate feedback on transmission errors	Cannot distinguish between application down vs network issues
Configuration	No need to configure scrape targets centrally	Each application must know collector endpoints
Scaling	Horizontal scaling of collectors is straightforward	Network storms during traffic spikes
Service Discovery	Applications handle their own endpoint resolution	Difficult to ensure all services are monitored

The fundamental challenge with push-based systems is **observability of the monitoring system itself**. When metrics stop arriving, it's difficult to determine whether the application has failed, the network path is broken, or the application simply has no activity to report. This ambiguity complicates alerting and incident response.

Pull-Based Systems (Prometheus, DataDog Agent):

Pull-based systems invert the responsibility: a central collector actively retrieves metrics from application endpoints. Applications expose metrics via HTTP endpoints in a standardized format, and the collector periodically scrapes these endpoints according to a configured schedule.

Aspect	Advantages	Disadvantages
Failure Detection	Clear distinction between target down vs no data	Applications must maintain HTTP endpoints
Network Control	Collector controls scrape timing and concurrency	Requires network reachability from collector to targets
Configuration	Centralized target configuration with service discovery	Initial setup complexity for service discovery
Debugging	Easy to manually inspect target endpoints	Firewall and network policy complexity
Consistency	Uniform scrape intervals and timeout handling	Potential polling overhead for idle applications

Hybrid Approaches (Prometheus Push Gateway, Vector):

Some systems attempt to combine both approaches, typically by providing a push-to-pull bridge. Applications push metrics to an intermediate gateway, which exposes them via pull endpoints for the main collector.

Key Insight: The choice between push and pull fundamentally affects system observability, failure modes, and operational complexity. Pull-based systems provide better visibility into the health of the monitoring system itself, at the cost of additional networking complexity.

Architectural Decision Record:

Decision: Pull-Based Scraping Architecture

- **Context:** Need to choose between push-based (applications send metrics) vs pull-based (collector retrieves metrics) architecture for our metrics collection system
- **Options Considered:**
 1. Pure push-based with UDP/HTTP transmission from applications
 2. Pull-based scraping with HTTP endpoints on applications
 3. Hybrid push-to-pull gateway model
- **Decision:** Pull-based scraping with HTTP endpoints
- **Rationale:** Pull-based systems provide superior observability of the monitoring infrastructure itself. When scraping fails, we know definitively that a target is unreachable or unresponsive, enabling precise alerting. Centralized configuration simplifies service discovery integration and reduces per-application configuration burden. The HTTP endpoint model also enables manual debugging and testing of individual services.
- **Consequences:** Applications must implement HTTP metrics endpoints and handle scraping load. Network policies must allow collector-to-target connectivity. However, we gain clear failure attribution, centralized configuration management, and better debugging capabilities.

Storage Engine Comparison:

Different metrics systems employ varying storage strategies, each optimized for different trade-offs:

System	Storage Approach	Compression	Query Performance	Operational Complexity
Graphite	Whisper files (RRD-style)	Fixed retention buckets	Fast for pre-aggregated data	Medium - file system management
InfluxDB	Custom time series engine	Snappy + series compression	Variable based on cardinality	High - clustering and consistency
Prometheus	Block-based with Gorilla compression	Delta-of-delta + XOR	Excellent for recent data	Low - single node simplicity
VictoriaMetrics	Compressed column storage	Multiple algorithms	Optimized for high cardinality	Medium - configuration complexity

Our system will adopt Prometheus's approach of **block-based storage with Gorilla compression** because it provides excellent compression ratios for typical metrics workloads while maintaining query performance for the most common use case: analyzing recent data.

Core Technical Challenges

Building an effective metrics collection system requires solving several interconnected technical challenges. Each challenge involves fundamental trade-offs that shape the system's architecture and performance characteristics.

Scale and Throughput Management:

Modern distributed systems can generate enormous volumes of metrics data. Consider a microservices deployment with 100 services, each running 10 instances, exposing 50 metrics each, scraped every 15 seconds. This generates $100 \times 10 \times 50 \times (3600/15) = 1.2$ million data points per hour, or approximately 330 samples per second just for the base metrics. Real deployments often exceed this by orders of magnitude when including infrastructure metrics, custom application metrics, and higher scrape frequencies.

The scale challenge manifests in several dimensions:

Ingestion Rate: The system must sustainably ingest hundreds of thousands to millions of samples per second without dropping data or introducing excessive latency. This requires efficient parsing, concurrent processing, and careful memory management to avoid garbage collection pressure.

Storage Volume: With retention periods measured in weeks or months, the total storage requirements can reach terabytes. Each sample consists of a timestamp (8 bytes), a float64 value (8 bytes), plus the overhead of series identification and indexing. Without compression, a billion samples would require at least 16 GB of raw storage.

Query Latency: Despite the massive data volumes, queries must complete within seconds to support interactive dashboards and real-time alerting. This requires intelligent indexing, data locality optimization, and query execution strategies that minimize disk I/O.

Cardinality Explosion Problem:

The **cardinality** of a metrics system refers to the number of unique time series, where each series is defined by a unique combination of metric name and label values. This represents perhaps the most insidious scaling challenge in metrics systems.

Consider a simple HTTP request counter with labels for method, status code, and endpoint:

```
http_requests_total{method="GET", status="200", endpoint="/api/users"}  
http_requests_total{method="POST", status="201", endpoint="/api/users"}  
http_requests_total{method="GET", status="404", endpoint="/api/orders"}
```

If your system has 10 HTTP methods, 20 possible status codes, and 100 endpoints, the potential cardinality is $10 \times 20 \times 100 = 20,000$ unique time series for this single metric. Add labels for instance ID, deployment version, and geographic region, and the cardinality explodes exponentially.

High cardinality creates multiple problems:

Problem Area	Impact	Root Cause
Memory Usage	Each series requires indexing structures in RAM	Series metadata and recent samples kept in memory
Query Performance	Label matching becomes expensive	Must scan large label indexes for selector evaluation
Storage Overhead	Small series create inefficient storage chunks	Compression algorithms work poorly on short time series
Ingestion Latency	New series creation becomes a bottleneck	Index updates and memory allocation under write load

Storage Efficiency and Compression:

Time series data exhibits characteristics that enable sophisticated compression, but achieving optimal compression ratios requires careful algorithm selection and implementation. The challenge is balancing compression effectiveness with query performance and computational overhead.

Timestamp Compression: Metrics are typically collected at regular intervals, creating predictable timestamp patterns. The Gorilla compression algorithm exploits this by storing timestamps as delta-of-deltas: instead of storing absolute timestamps, it stores the difference between consecutive timestamp differences. For regular scrape intervals, this often compresses timestamps to just a few bits per sample.

Value Compression: Metric values often change gradually or remain constant for extended periods. Gorilla compression uses XOR-based encoding where each value is XORed with its predecessor, and only the differing bits are stored. When values are stable, this can compress 64-bit floats to just a few bits.

However, compression introduces complexity:

Write Amplification: Compressed blocks must be periodically finalized and written to disk, creating bursty I/O patterns that can interfere with query performance.

Query Overhead: Reading compressed data requires decompression, adding CPU overhead to query execution. The system must balance compression ratios against query latency requirements.

Memory Pressure: Compression algorithms maintain state for active series, and this metadata can consume significant memory in high-cardinality environments.

Query Performance Under Load:

The query engine must support complex analytical operations across massive datasets while maintaining interactive response times. This challenge is complicated by the multi-dimensional nature of metrics data and the variety of query patterns users employ.

Range Query Scalability: Range queries that retrieve data across long time windows or high-cardinality label combinations can potentially scan terabytes of data. The query engine must use indexing, pruning, and parallel execution strategies to make such queries feasible.

Aggregation Efficiency: PromQL queries often perform aggregation operations (sum, average, percentile) across thousands of time series. These operations require careful memory management and algorithmic optimization to avoid excessive memory usage or computation time.

Concurrent Query Load: Production metrics systems serve multiple concurrent users running dashboards, alerts, and ad-hoc queries. The system must manage resource allocation to prevent expensive queries from interfering with critical alerting queries.

Index Maintenance: As new series are created and old series expire, the label indexes that enable fast series selection must be updated consistently without blocking ongoing queries.

Critical Design Principle: Every architectural decision in our metrics system must consider its impact on cardinality scaling. Features that seem innocent in small deployments can become system-breaking bottlenecks when cardinality grows from thousands to millions of series.

Real-Time vs Historical Query Patterns:

Metrics queries exhibit distinct patterns that the system can optimize for:

Hot Data Access: Most queries focus on recent data (last few hours to days), which should be kept in memory or fast storage for immediate access.

Cold Data Access: Historical queries are less frequent but often span longer time ranges, requiring different optimization strategies focused on I/O efficiency rather than memory access.

Alert Query Priority: Alerting queries must complete quickly and reliably, potentially requiring resource reservation or priority queuing mechanisms.

Dashboard Query Batching: Dashboard refreshes often trigger multiple related queries that could benefit from shared computation or caching strategies.

Understanding these challenges provides the foundation for our architectural decisions. Each component of our metrics collection system — the data model, scrape engine, storage layer, and query engine — must be designed to address these fundamental scaling and performance constraints.

⚠ Pitfall: Ignoring Cardinality Early Many developers building metrics systems focus initially on throughput and storage optimization while treating cardinality as a later concern. This approach leads to systems that work well in testing but fail catastrophically in production when label combinations proliferate. From the initial design phase, every data structure and algorithm must account for high-cardinality scenarios. For example, using hash maps keyed by

series labels seems efficient until you have millions of series, at which point the hash map itself becomes a memory bottleneck and garbage collection issue.

⚠ Pitfall: Optimizing for Average Case Metrics systems exhibit highly variable load patterns — scraping creates periodic ingestion bursts, queries arrive in waves during incident response, and certain time periods (deployments, traffic spikes) generate disproportionate data volumes. Designing for average-case performance results in systems that become unresponsive precisely when reliability is most critical. Instead, all components must be designed to handle 99th percentile loads gracefully, with explicit backpressure and resource protection mechanisms.

Implementation Guidance

Building a metrics collection system requires careful technology selection and project organization. This guidance helps you make practical implementation decisions and avoid common pitfalls.

Technology Recommendations:

Component	Simple Option	Advanced Option	Rationale
HTTP Server	<code>net/http</code> with standard mux	<code>gorilla/mux</code> or <code>gin</code>	Standard library sufficient for metrics endpoints
Time Series Storage	File-based chunks with <code>os.File</code>	Embedded key-value store like <code>bbolt</code>	File system provides simple persistence model
Compression	Basic delta encoding	Full Gorilla algorithm	Start simple, optimize when cardinality grows
Service Discovery	Static YAML configuration	Kubernetes API integration	Static config easier to debug and understand
Parsing	Custom text parser	<code>prometheus/common/expfmt</code>	Custom parser teaches format understanding
Concurrency	<code>goroutines</code> with channels	Worker pool patterns	Go's concurrency primitives handle most cases

Recommended Project Structure:

Understanding how to organize your codebase prevents the common mistake of creating a monolithic `main.go` file that becomes unmaintainable:

```
metrics-system/
├── cmd/
│   ├── collector/           ← Main collector binary
│   │   └── main.go
│   └── query/              ← Query API server
│       └── main.go
└── internal/
    ├── model/               ← Metrics data model (Milestone 1)
    │   ├── metric.go          ← Counter, Gauge, Histogram types
    │   ├── labels.go          ← Label handling and validation
    │   ├── sample.go          ← Sample and timestamp types
    │   └── metadata.go        ← Metric metadata storage
    ├── scrape/               ← Scrape engine (Milestone 2)
    │   ├── discovery.go       ← Target discovery logic
    │   ├── scraper.go         ← HTTP scraping implementation
    │   ├── scheduler.go        ← Scrape interval management
    │   └── parser.go          ← Metrics format parsing
    ├── storage/              ← Time series storage (Milestone 3)
    │   ├── engine.go          ← Main storage engine
    │   ├── compression.go     ← Gorilla compression implementation
    │   ├── index.go           ← Series indexing
    │   ├── retention.go        ← Data lifecycle management
    │   └── wal.go              ← Write-ahead logging
    ├── query/                ← Query engine (Milestone 4)
    │   ├── parser.go          ← PromQL parsing
    │   ├── executor.go        ← Query execution
    │   ├── aggregation.go     ← Aggregation functions
    │   └── matcher.go          ← Label matching logic
    └── api/
        ├── scrape.go          ← Metrics exposition endpoints
        └── query.go            ← Query API endpoints
└── pkg/
└── configs/               ← Public interfaces (if building libraries)
└── scripts/               ← Example configuration files
└── test/
    ├── integration/         ← Build and deployment scripts
    └── testdata/             ← Integration tests and test data
```

Infrastructure Starter Code:

Here's a complete HTTP server foundation that handles the basic networking and routing for both scrape targets and the query API:

GO

```
// internal/api/server.go

package api

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

// Server wraps HTTP server functionality for metrics collection

type Server struct {
    httpServer *http.Server
    mux        *http.ServeMux
}

// NewServer creates a new API server instance

func NewServer(port int) *Server {
    mux := http.NewServeMux()

    server := &http.Server{
        Addr:         fmt.Sprintf(":%d", port),
        Handler:      mux,
        ReadTimeout:  30 * time.Second,
        WriteTimeout: 30 * time.Second,
        IdleTimeout:  120 * time.Second,
    }

    return &Server{
```

```

    httpServer: server,

    mux:      mux,
}

}

// RegisterScrapeEndpoint adds a metrics exposition endpoint

func (s *Server) RegisterScrapeEndpoint(path string, handler http.HandlerFunc) {
    s.mux.HandleFunc(path, handler)
}

// RegisterQueryEndpoint adds a query API endpoint

func (s *Server) RegisterQueryEndpoint(path string, handler http.HandlerFunc) {
    s.mux.HandleFunc(path, handler)
}

// Start begins serving HTTP requests

func (s *Server) Start() error {
    log.Printf("Starting metrics 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 metrics server...")
    return s.httpServer.Shutdown(ctx)
}

```

Configuration Management Infrastructure:

GO

```
// internal/config/config.go

package config

import (
    "fmt"
    "time"
    "gopkg.in/yaml.v2"
    "os"
)

// Config holds all system configuration

type Config struct {
    Scrape   ScrapeConfig   `yaml:"scrape"`
    Storage StorageConfig  `yaml:"storage"`
    Query   QueryConfig   `yaml:"query"`
}

// ScrapeConfig configures the scraping engine

type ScrapeConfig struct {
    Interval      time.Duration `yaml:"interval"`
    Timeout       time.Duration `yaml:"timeout"`
    MaxConcurrency int          `yaml:"max_concurrency"`
    StaticTargets []string     `yaml:"static_targets"`
}

// StorageConfig configures time series storage

type StorageConfig struct {
    DataDir        string      `yaml:"data_dir"`
    RetentionPeriod time.Duration `yaml:"retention_period"`
    ChunkSize      int         `yaml:"chunk_size"`
}
```

```
    EnableCompression bool      `yaml:"enable_compression"`

}

// QueryConfig configures the query engine

type QueryConfig struct {

    MaxConcurrency int      `yaml:"max_concurrency"`

    QueryTimeout   time.Duration `yaml:"query_timeout"`

    MaxSamples     int      `yaml:"max_samples"`

}

// LoadFromFile reads configuration from YAML file

func LoadFromFile(filename string) (*Config, error) {

    data, err := os.ReadFile(filename)

    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)

    }

    return &config, nil

}

// SetDefaults populates default configuration values

func (c *Config) SetDefaults() {

    if c.Scrape.Interval == 0 {

        c.Scrape.Interval = 15 * time.Second

    }

}
```

```
if c.Scrape.Timeout == 0 {

    c.Scrape.Timeout = 10 * time.Second

}

if c.Scrape.MaxConcurrency == 0 {

    c.Scrape.MaxConcurrency = 50

}

if c.Storage.DataDir == "" {

    c.Storage.DataDir = "./data"

}

if c.Storage.RetentionPeriod == 0 {

    c.Storage.RetentionPeriod = 30 * 24 * time.Hour // 30 days

}

if c.Query.MaxConcurrency == 0 {

    c.Query.MaxConcurrency = 10

}

if c.Query.QueryTimeout == 0 {

    c.Query.QueryTimeout = 30 * time.Second

}

}
```

Core Logging Infrastructure:

GO

```
// internal/logging/logger.go

package logging

import (
    "log"
    "os"
    "fmt"
)

type Logger struct {
    infoLogger *log.Logger
    errorLogger *log.Logger
    debugLogger *log.Logger
}

func NewLogger() *Logger {
    return &Logger{
        infoLogger: log.New(os.Stdout, "INFO: ", log.Ldate|log.Ltime|log.Lshortfile),
        errorLogger: log.New(os.Stderr, "ERROR: ", log.Ldate|log.Ltime|log.Lshortfile),
        debugLogger: log.New(os.Stdout, "DEBUG: ", log.Ldate|log.Ltime|log.Lshortfile),
    }
}

func (l *Logger) Info(v ...interface{}) {
    l.infoLogger.Println(v...)
}

func (l *Logger) Infof(format string, v ...interface{}) {
    l.infoLogger.Printf(format, v...)
}
```

```
func (l *Logger) Error(v ...interface{}) {
    l.errorLogger.Println(v...)
}

func (l *Logger) Errorf(format string, v ...interface{}) {
    l.errorLogger.Printf(format, v...)
}

func (l *Logger) Debug(v ...interface{}) {
    l.debugLogger.Println(v...)
}

func (l *Logger) Debugf(format string, v ...interface{}) {
    l.debugLogger.Printf(format, v...)
}
```

Development Workflow Tips:

- 1. Start with Static Configuration:** Begin with YAML configuration files before implementing dynamic service discovery. This allows you to test the core functionality without network complexity.
- 2. Use Table-Driven Tests:** Go's table-driven test pattern works excellently for metrics systems where you need to test many label combinations and edge cases:

```

func TestLabelMatching(t *testing.T) {
    GO

    tests := []struct {
        name      string
        labels    map[string]string
        matcher   string
        expected  bool
    }{
        {"exact match", map[string]string{"job": "api"}, `job="api"`, true},
        {"regex match", map[string]string{"instance": "web-01"}, `instance=~"web-.*"`, true},
        // Add more test cases...
    }

    // Test implementation...
}

}

```

3. Implement Graceful Shutdown: Metrics systems often run as long-lived services. Implement proper shutdown handling to avoid data loss:

```

// In your main.go
GO

c := make(chan os.Signal, 1)
signal.Notify(c, os.Interrupt, syscall.SIGTERM)

<-c

ctx, cancel := context.WithTimeout(context.Background(), 30*time.Second)
defer cancel()

if err := server.Shutdown(ctx); err != nil {
    log.Fatal("Server forced to shutdown:", err)
}

```

Language-Specific Hints for Go:

- **Memory Management:** Use `sync.Pool` for frequently allocated objects like sample slices to reduce GC pressure

- **Goroutine Management:** Always use `context.Context` for cancellation in long-running operations like scraping and queries
- **Time Handling:** Use `time.Unix()` for timestamp conversions and always store timestamps in UTC
- **Error Handling:** Wrap errors with `fmt.Errorf("operation failed: %w", err)` to maintain error context through the call stack
- **Concurrent Maps:** Use `sync.RWMutex` to protect shared data structures like series indexes
- **File I/O:** Always call `file.Sync()` after writing critical data like WAL entries to ensure durability

Common Development Pitfalls:

⚠ Pitfall: Blocking Operations in Hot Paths Never perform I/O operations directly in scraping or query processing goroutines without proper timeouts. Always use `context.WithTimeout()` and handle cancellation appropriately.

⚠ Pitfall: Unbounded Memory Growth Metrics systems can accumulate memory leaks through retained references to old samples or series. Use profiling tools like `go tool pprof` regularly during development to identify memory growth patterns.

⚠ Pitfall: Ignoring Graceful Degradation Build backpressure and circuit breaker patterns from the beginning. When storage is full or queries are slow, the system should reject new work rather than becoming unresponsive.

Goals and Non-Goals

Milestone(s): This section establishes the scope and requirements that guide all four milestones: Metrics Data Model (1), Scrape Engine (2), Time Series Storage (3), and Query Engine (4).

The Observatory Charter Mental Model

Think of this goals section as the charter document for a national weather observatory network. Just as meteorologists must decide whether their observatory will track temperature and rainfall (essential) versus tracking every atmospheric particle (impossible), we must define exactly what our metrics collection system will and won't accomplish. The charter prevents scope creep—when stakeholders later ask "can it also do real-time alerting?" we can point to this document and explain why alerting is explicitly out of scope for this implementation.

This boundary-setting is crucial for complex systems. Without clear goals, developers often build everything they can imagine rather than building the core functionality excellently. A well-scoped metrics system that handles scraping, storage, and querying perfectly is far more valuable than a system that attempts alerting, dashboards, and federation but does none of them well.

The goals also serve as acceptance criteria for the project. Each requirement must be measurable and testable—we can't claim success with vague objectives like "good performance." Instead, we specify concrete targets: sub-second query response times, specific storage compression ratios, and exact retention capabilities.

Functional Requirements

The functional requirements define the core capabilities our metrics system must provide to be considered complete. These map directly to our four major milestones and represent the essential features that distinguish a metrics

collection system from a generic time-series database.

Metrics Data Model Requirements

Our system must implement a comprehensive metrics data model that supports the standard observability metric types used in modern monitoring systems. The `Counter` type must enforce monotonic increase semantics, meaning values can only go up or reset to zero (typically on process restart). The `Gauge` type must allow arbitrary value changes to represent measurements like memory usage or queue depth that can increase or decrease freely. The `Histogram` type must track value distributions using configurable buckets, enabling percentile calculations and distribution analysis.

Metric Type	Semantic Behavior	Example Use Case	Key Constraint
Counter	Monotonic increase, resets on restart	HTTP requests served	Value \geq previous value or reset to 0
Gauge	Arbitrary value changes	Memory usage bytes	No constraints on value changes
Histogram	Distribution tracking in buckets	Request latency distribution	Bucket boundaries fixed after creation
Summary	Client-side quantile calculation	Response time percentiles	Quantiles calculated at source

The labeling system must support multi-dimensional metrics where each time series is uniquely identified by its metric name plus label set. Labels enable powerful filtering and aggregation—a single `http_requests_total` metric with `method`, `status_code`, and `handler` labels can answer questions like "What's the error rate for POST requests to the login endpoint?" The system must validate label names and values, rejecting reserved prefixes like `_` and ensuring label values don't contain characters that break the exposition format.

Decision: Multi-Dimensional Labeling

- **Context:** Need to support filtering and aggregation across different metric dimensions
- **Options Considered:** Single-dimensional metrics with encoded names, hierarchical metric names, multi-dimensional labels
- **Decision:** Multi-dimensional labels attached to each metric
- **Rationale:** Labels provide flexibility for ad-hoc queries without predefined metric names, enable efficient storage of related time series, and match Prometheus compatibility
- **Consequences:** Enables powerful querying but introduces cardinality explosion risks that must be managed

Scrape Engine Requirements

The scrape engine must implement pull-based metrics collection that actively retrieves metrics from configured targets. Static configuration must support defining scrape targets through YAML configuration files, specifying endpoints, intervals, and timeouts. The system must parse the Prometheus exposition format, handling both the standard text format and optional metric metadata including help text and type information.

Service discovery integration must automatically update the target list when services are added or removed. The system must support at minimum static file-based service discovery, where external systems can update JSON or YAML files containing current service endpoints. The scrape scheduler must respect per-target intervals, ensuring targets are scraped consistently without drift or overlap.

Scrape Component	Responsibility	Configuration	Error Handling
Target Discovery	Find scrape endpoints	Static config, service discovery files	Log discovery errors, continue with known targets
HTTP Client	Fetch metrics from endpoints	Timeout, retry count, headers	Mark target down, continue scrape cycle
Parser	Parse exposition format text	Format validation, metadata extraction	Skip malformed metrics, log parse errors
Scheduler	Trigger scrapes at intervals	Per-target intervals, jitter	Compensate for missed scrapes, prevent overlap

The scrape engine must handle target failures gracefully. Network timeouts must not block other targets' scrape cycles. HTTP errors (4xx, 5xx) must be logged and reported but not crash the scraper. Malformed exposition format must be handled by skipping unparseable lines while processing valid metrics from the same target.

Storage Engine Requirements

The storage engine must provide efficient time-series storage with compression achieving less than 2 bytes per sample on average. The system must implement Gorilla-style compression using delta-of-delta encoding for timestamps and XOR encoding for floating-point values. This compression is essential for handling high-cardinality metrics at scale without exhausting storage capacity.

The storage must support configurable retention periods, automatically deleting data older than the specified duration. The default retention period must be 30 days, but the system must support retention periods from hours to years. Data deletion must be efficient, removing entire blocks rather than individual samples to avoid fragmentation.

Storage Feature	Requirement	Performance Target	Implementation Note
Compression Ratio	< 2 bytes per sample	1.3 bytes average	Gorilla delta-of-delta + XOR
Retention	Configurable age-based deletion	30 days default	Block-based deletion
Write Throughput	Handle scrape ingestion	100k samples/sec	Batch writes, async commits
Read Latency	Query response time	< 1 second typical	Efficient indexing

The indexing system must enable fast lookup of time series by metric name and label combinations. The index must support exact label matching, regex matching, and negative matching (labels that don't equal a value). Query performance must remain reasonable even with high-cardinality label combinations, though the system may reject queries that would examine excessive numbers of series.

Query Engine Requirements

The query engine must implement a PromQL-compatible query language supporting instant queries (single timestamp), range queries (time window), and basic aggregation functions. The system must support label selectors using exact match (`=`), not-equal (`!=`), regex match (`=~`), and negative regex (`!~`) operators.

Aggregation functions must include `sum`, `avg`, `max`, `min`, and `count` operations that can group results by specified label dimensions. Range queries must return data points at regular step intervals within the specified time window, interpolating missing values when necessary.

Query Type	Input	Output	Example
Instant	Metric selector + timestamp	Vector of current values	<code>up</code> at now
Range	Metric selector + time window + step	Matrix of time series	<code>cpu_usage[5m]</code>
Aggregation	Vector + grouping labels	Grouped vector	<code>sum by (instance)</code> <code>(cpu_usage)</code>
Rate Calculation	Counter range	Rate vector	<code>rate(http_requests[5m])</code>

The query engine must handle counter resets correctly when calculating rates and derivatives. When a counter value decreases (indicating a reset), the rate calculation must treat the reset as the beginning of a new monotonic sequence rather than a negative rate.

Quality Attributes

The quality attributes define the performance, scalability, and reliability characteristics our metrics system must achieve. These non-functional requirements are often more critical than features—a metrics system that loses data or responds slowly becomes unusable regardless of its feature completeness.

Performance Requirements

Query response time must remain under one second for typical queries examining up to 10,000 time series. Range queries spanning one week with five-minute resolution must complete within five seconds. These performance targets ensure the system remains usable for operational monitoring where slow queries impede incident response.

Storage ingestion must handle sustained write rates of 100,000 samples per second without falling behind or dropping data. This throughput supports monitoring environments with thousands of services each exposing hundreds of metrics. Write latency must remain low enough that scraped metrics appear in queries within the scrape interval.

Performance Metric	Target	Measurement Method	Degradation Threshold
Query Response	< 1 second typical	95th percentile latency	> 5 seconds
Range Query	< 5 seconds	One week window	> 30 seconds
Write Throughput	100k samples/sec	Sustained ingestion	Falls behind scrape rate
Storage Efficiency	< 2 bytes/sample	Compression ratio	> 4 bytes/sample

Memory usage must remain bounded even with high-cardinality metrics. The system must reject queries or scrape configurations that would consume excessive memory, providing clear error messages about cardinality limits. Disk usage must grow predictably based on ingestion rate and retention period.

Scalability Requirements

The system must support monitoring environments with up to 1,000 scrape targets and 1 million active time series. While this implementation focuses on single-instance deployment, the architecture must not preclude future horizontal scaling through techniques like sharding or federation.

Cardinality must be controlled to prevent label explosion that could exhaust memory or storage. The system must provide mechanisms to limit the number of unique label value combinations, either through validation rules or runtime limits. High-cardinality label combinations (> 10,000 series per metric name) should trigger warnings or rejections.

Scalability Dimension	Limit	Rationale	Failure Mode
Scrape Targets	1,000 targets	Single-instance HTTP client limits	Connection exhaustion
Active Series	1M series	Memory for index and active chunks	Out of memory errors
Label Cardinality	10k series per metric	Prevent exponential explosion	Query performance degradation
Retention Data	30 days × ingestion rate	Disk capacity planning	Disk full, query slowness

Reliability Requirements

The system must provide durability guarantees ensuring that successfully scraped metrics are not lost due to process crashes or system failures. Write-ahead logging must persist ingested samples before acknowledging success. Recovery after crashes must restore the system to a consistent state without data loss.

Target scraping must be resilient to individual target failures. Network partitions, service restarts, or malformed responses from one target must not affect scraping of other targets. The scrape engine must continue operating with degraded target coverage rather than failing completely.

Decision: Pull-Based Scraping Model

- **Context:** Need to choose between push-based (targets send metrics) and pull-based (collector fetches metrics) architectures
- **Options Considered:** Push-based with HTTP POST, pull-based with HTTP GET, hybrid push/pull
- **Decision:** Pull-based scraping with HTTP GET requests
- **Rationale:** Pull model provides better failure isolation (target failures don't crash collector), enables service discovery integration, matches Prometheus ecosystem, and simplifies target authentication
- **Consequences:** Requires targets to expose HTTP endpoints, may have higher network overhead, but provides better operational control and debugging

Data consistency must be maintained under concurrent access. Multiple queries must be able to execute simultaneously without corrupting results or crashing the system. Write operations must not interfere with concurrent reads beyond brief locking periods.

The system must handle resource exhaustion gracefully rather than crashing or corrupting data. When memory usage approaches limits, the system should reject new queries or reduce cache sizes while continuing to serve existing requests. Disk full conditions should pause ingestion while preserving existing data.

Scope Limitations

The scope limitations define features explicitly excluded from this implementation. These boundaries prevent scope creep and ensure we build the core metrics collection capabilities excellently rather than attempting everything mediocrely.

Alerting System Exclusion

This implementation does not include alerting capabilities such as rule evaluation, threshold monitoring, or notification delivery. While alerting is a natural extension of metrics collection, implementing it properly requires additional components for rule management, evaluation scheduling, notification routing, and alert state tracking.

Building alerting well requires solving problems orthogonal to metrics collection: template rendering for notifications, integration with external systems (email, Slack, PagerDuty), alert deduplication and grouping, and escalation policies. Including alerting would double the system complexity without improving the core collection, storage, and querying capabilities.

Alerting Feature	Excluded Reason	Alternative Approach
Rule Evaluation	Complex scheduling and state management	External alerting system queries our API
Notification Delivery	Requires integration with many external services	Use dedicated alerting tools
Alert State Tracking	Complex state machine for firing/resolved states	Stateless query-based detection
Escalation Policies	Complex workflow management	External incident management tools

Future implementations can add alerting by building a separate alert manager that queries this metrics system's API, maintaining clean separation of concerns.

Dashboard and Visualization Exclusion

The system does not include built-in dashboards, graphing capabilities, or web-based visualization tools. While metrics are most valuable when visualized, building excellent charting and dashboard functionality requires deep frontend expertise and extensive JavaScript development.

Visualization tools have different requirements than metrics collection: real-time updates, interactive charts, dashboard templating, user authentication, and responsive design. These concerns are better addressed by specialized tools like Grafana that can query our system's API.

Advanced Query Functions Exclusion

The PromQL implementation includes only basic aggregation functions (`sum`, `avg`, `max`, `min`, `count`) and excludes advanced functions like `predict_linear`, `histogram_quantile`, or complex mathematical operations. These advanced functions require sophisticated algorithms and extensive testing to implement correctly.

Similarly, the system excludes recording rules (pre-computed queries stored as new metrics) and complex rate calculations beyond basic `rate()` function. These features add significant complexity to the query engine without being essential for basic monitoring.

Advanced Feature	Excluded Reason	Basic Alternative
<code>predict_linear</code>	Requires linear regression algorithms	Use external analysis tools
<code>histogram_quantile</code>	Complex quantile calculation from buckets	Use summary metrics instead
Recording Rules	Requires rule management and scheduling	Run queries manually or externally
Complex Math Functions	Requires extensive function library	Perform calculations in analysis tools

Multi-Instance Federation Exclusion

This implementation targets single-instance deployment and excludes federation capabilities that would allow multiple metrics instances to share data or provide hierarchical aggregation. Federation requires solving distributed systems problems like data replication, conflict resolution, and cross-instance querying.

Building federation properly requires consensus protocols, network partition handling, and complex query routing logic. These distributed systems challenges are significant projects themselves and would overshadow the core metrics collection functionality.

Enterprise Features Exclusion

The system excludes enterprise-oriented features like user authentication, multi-tenancy, access control, and audit logging. These features require extensive security implementation and user management capabilities that are orthogonal to metrics collection.

Similarly, the system excludes advanced operational features like automatic backup/restore, cluster management, or sophisticated monitoring of the metrics system itself. These features are valuable for production deployment but not essential for understanding how metrics collection works.

Design Insight: Scope Boundaries Enable Excellence By explicitly excluding advanced features, we can focus engineering effort on making the core metrics collection, storage, and querying capabilities excellent. A metrics system that handles these fundamentals perfectly provides a solid foundation for adding excluded features later, while a system that attempts everything often does nothing well.

The excluded features can be addressed through external integration rather than built-in functionality. Alerting systems can query our API, visualization tools can display our data, and operational tools can manage our deployment. This approach follows the Unix philosophy of building tools that do one thing excellently and compose well with other tools.

Common Pitfalls in Scope Definition

Developers often make these mistakes when defining project scope:

Pitfall: Vague Non-Goals - Writing "won't include advanced features" without specifying exactly which features. This leads to scope creep when stakeholders assume their favorite feature isn't "advanced." Instead, explicitly list each excluded capability with reasoning.

Pitfall: Feature Creep During Implementation - Adding "quick features" during development because they seem easy. A simple dashboard or basic alerting might seem straightforward, but each addition introduces new failure modes, test requirements, and maintenance burden. Stick to the defined scope religiously.

Pitfall: Unrealistic Performance Targets - Setting performance requirements without understanding the underlying constraints. Claiming sub-millisecond query times while implementing complex aggregations over millions of series. Base performance targets on realistic measurements from similar systems.

Pitfall: Missing Quality Attributes - Focusing only on functional requirements while ignoring performance, reliability, and scalability needs. A metrics system that handles all the required features but responds slowly or loses data is unusable for monitoring production systems.

Implementation Guidance

The requirements established in this section drive architectural decisions throughout the remaining design. Each functional requirement maps to specific implementation choices, while quality attributes establish measurable targets for validation.

Requirements Traceability Matrix

Requirement Category	Implementation Component	Validation Method	Success Criteria
Metrics Data Model	<code>Counter</code> , <code>Gauge</code> , <code>Histogram</code> types	Unit tests with semantic validation	Monotonic counter behavior, flexible gauge updates
Multi-dimensional Labels	Label map with validation	Integration tests with high-cardinality scenarios	Supports 10k series per metric name
Pull-based Scraping	HTTP client with service discovery	Load testing with 1000 targets	Maintains scrape intervals under load
Gorilla Compression	Delta-of-delta and XOR encoding	Compression ratio measurement	< 2 bytes per sample average
PromQL Queries	Parser and execution engine	Query correctness and performance tests	< 1 second response for typical queries

Technology Stack Recommendations

For implementing the requirements defined in this section, the technology choices directly impact our ability to meet performance and scalability targets:

Component	Simple Option	Advanced Option	Rationale
HTTP Server	<code>net/http</code> standard library	<code>fasthttp</code> or <code>gin</code> framework	Standard library sufficient for scraping loads
Configuration	<code>yaml.v3</code> for static config	<code>viper</code> for dynamic config	YAML parsing meets static config requirements
Time Series Storage	Append-only files with custom format	Embedded database like BadgerDB	Custom format provides compression control
Indexing	In-memory maps with periodic snapshots	LSM-tree based indexes	Memory indexes meet single-instance scale
Compression	Custom Gorilla implementation	Existing libraries like <code>tsz</code>	Custom implementation for educational value

Configuration Structure Foundation

The requirements drive a specific configuration structure that supports all functional needs while maintaining simplicity:

GO

```
// Config represents the complete system configuration matching our functional requirements

type Config struct {

    // Scrape configuration supports pull-based collection requirement

    Scrape ScrapeConfig `yaml:"scrape"`

    // Storage configuration meets retention and compression requirements

    Storage StorageConfig `yaml:"storage"`

    // Query configuration enforces performance and resource limits

    Query QueryConfig `yaml:"query"`

    // Server configuration for HTTP endpoints

    Server ServerConfig `yaml:"server"`

}

// ScrapeConfig implements target discovery and interval management requirements

type ScrapeConfig struct {

    // Global defaults applied to all targets

    ScrapeInterval time.Duration `yaml:"scrape_interval"`

    ScrapeTimeout  time.Duration `yaml:"scrape_timeout"`

    // Static target configuration

    StaticConfigs []StaticConfig `yaml:"static_configs"`

    // Service discovery configuration

    FileServiceDiscovery []FileSDConfig `yaml:"file_sd_configs"`

}

// StorageConfig meets retention and compression requirements
```

```

type StorageConfig struct {

    // Data directory for time series storage

    DataDirectory string `yaml:"data_directory"`

    // Retention period for automatic cleanup

    RetentionPeriod time.Duration `yaml:"retention_period"`

    // Compression settings for Gorilla algorithm

    CompressionEnabled bool `yaml:"compression_enabled"`

    // Write-ahead log configuration for durability

    WALEnabled bool `yaml:"wal_enabled"`

}

// QueryConfig enforces performance and resource limits

type QueryConfig struct {

    // Maximum query execution time

    QueryTimeout time.Duration `yaml:"query_timeout"`

    // Maximum number of series a query can examine

    MaxSeries int `yaml:"max_series"`

    // Maximum range query duration

    MaxRangeDuration time.Duration `yaml:"max_range_duration"`

}

```

Requirements Validation Framework

Each milestone should validate that requirements are being met through specific tests and measurements:

GO

```
// RequirementsValidator provides methods to verify system meets defined requirements

type RequirementsValidator struct {
    config *Config
    logger Logger
}

// ValidatePerformanceRequirements checks that system meets performance targets

func (v *RequirementsValidator) ValidatePerformanceRequirements() error {
    // TODO: Implement query latency measurement
    // TODO: Validate write throughput under load
    // TODO: Check compression ratio achievement
    // TODO: Verify memory usage remains bounded
    return nil
}

// ValidateScalabilityRequirements verifies system handles target scale

func (v *RequirementsValidator) ValidateScalabilityRequirements() error {
    // TODO: Test with 1000 scrape targets
    // TODO: Verify 1M active time series support
    // TODO: Check cardinality limit enforcement
    // TODO: Validate retention period handling
    return nil
}

// ValidateReliabilityRequirements ensures durability and fault tolerance

func (v *RequirementsValidator) ValidateReliabilityRequirements() error {
    // TODO: Test crash recovery with WAL
    // TODO: Verify target failure isolation
    // TODO: Check graceful resource exhaustion handling
    // TODO: Validate concurrent access safety
}
```

```
    return nil  
}  
}
```

Milestone Checkpoint: Requirements Verification

After implementing each milestone, verify it meets the requirements established in this section:

Milestone 1 Checkpoint - Metrics Data Model:

- Run: `go test ./internal/metrics/... -v`
- Expected: All metric type tests pass, counter monotonicity enforced, labels validate correctly
- Manual verification: Create metrics with various label combinations, confirm cardinality limits work
- Performance check: Create 10k series, verify memory usage remains reasonable

Milestone 2 Checkpoint - Scrape Engine:

- Run: `go test ./internal/scrapers/... -v` and start test HTTP targets
- Expected: Targets discovered from config, scraped at intervals, parse errors handled gracefully
- Manual verification: Configure 10 targets, observe scrape success rates and timing
- Performance check: Scrape 100 targets simultaneously, verify no blocking or missed intervals

Milestone 3 Checkpoint - Storage Engine:

- Run: `go test ./internal/storage/... -v` and measure compression ratios
- Expected: Data persisted durably, Gorilla compression achieves < 2 bytes/sample, retention works
- Manual verification: Insert sample data, restart system, verify data recovers correctly
- Performance check: Sustain 10k samples/sec writes, measure query response times

Milestone 4 Checkpoint - Query Engine:

- Run: `go test ./internal/query/... -v` and execute sample PromQL queries
- Expected: Basic PromQL parsing works, aggregations produce correct results, range queries function
- Manual verification: Run queries against stored data, verify results match expectations
- Performance check: Query 1000 series over one week, verify sub-second response

The requirements defined in this section provide the acceptance criteria for each milestone and the overall system success. Every implementation decision should trace back to these requirements, and every feature should be validated against these targets.

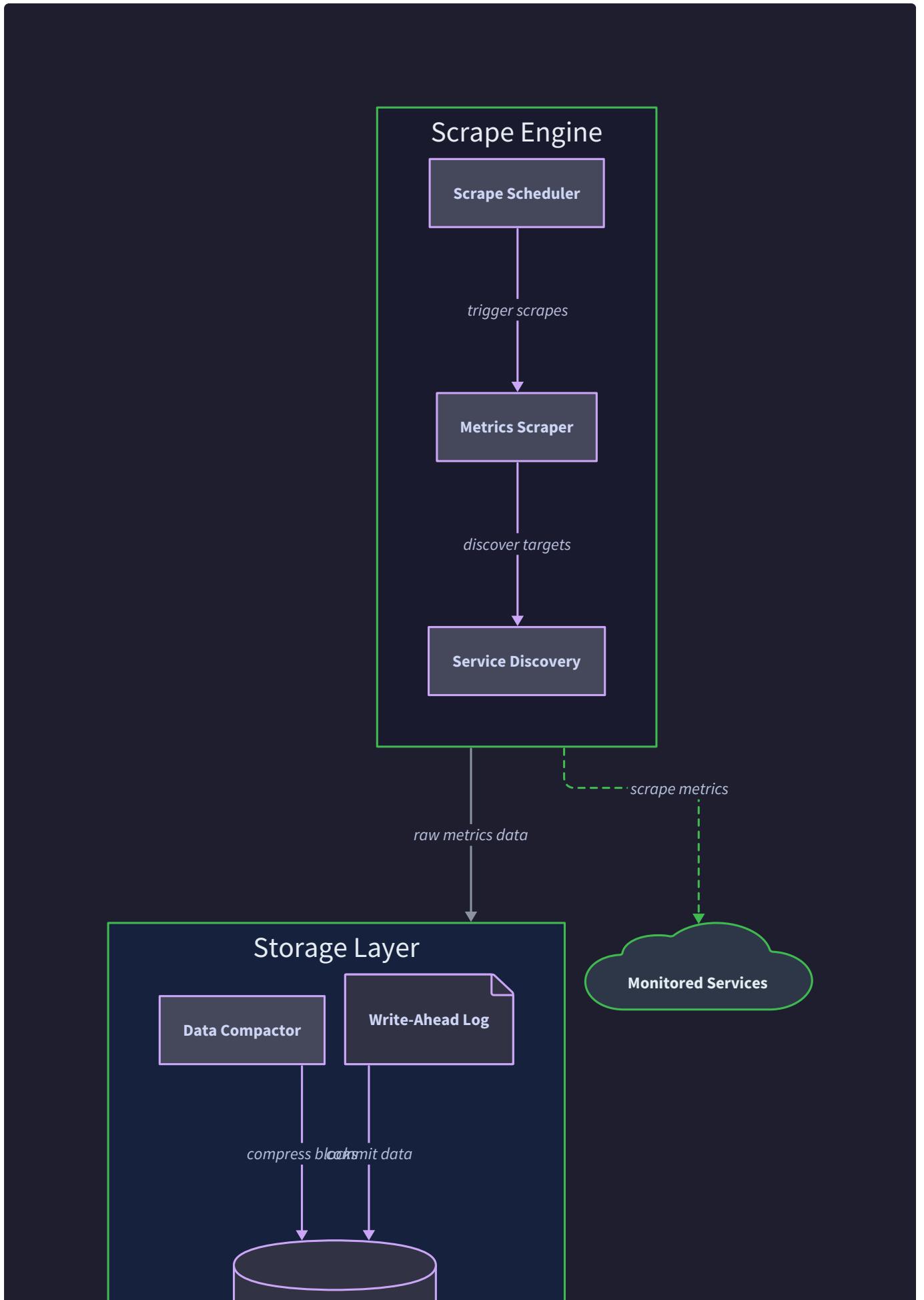
High-Level Architecture

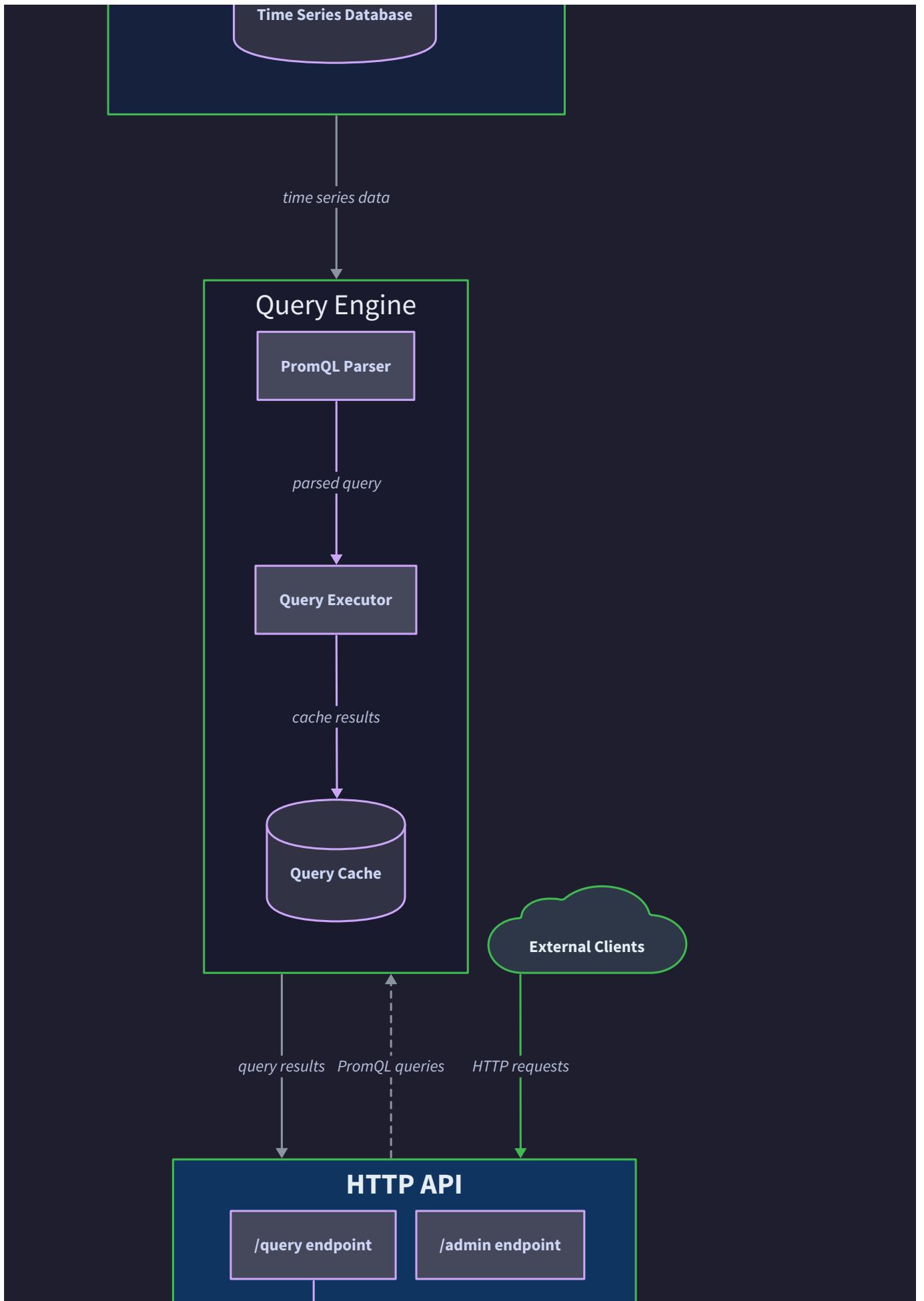
Milestone(s): This section establishes the architectural foundation for all four milestones: Metrics Data Model (1), Scrape Engine (2), Time Series Storage (3), and Query Engine (4).

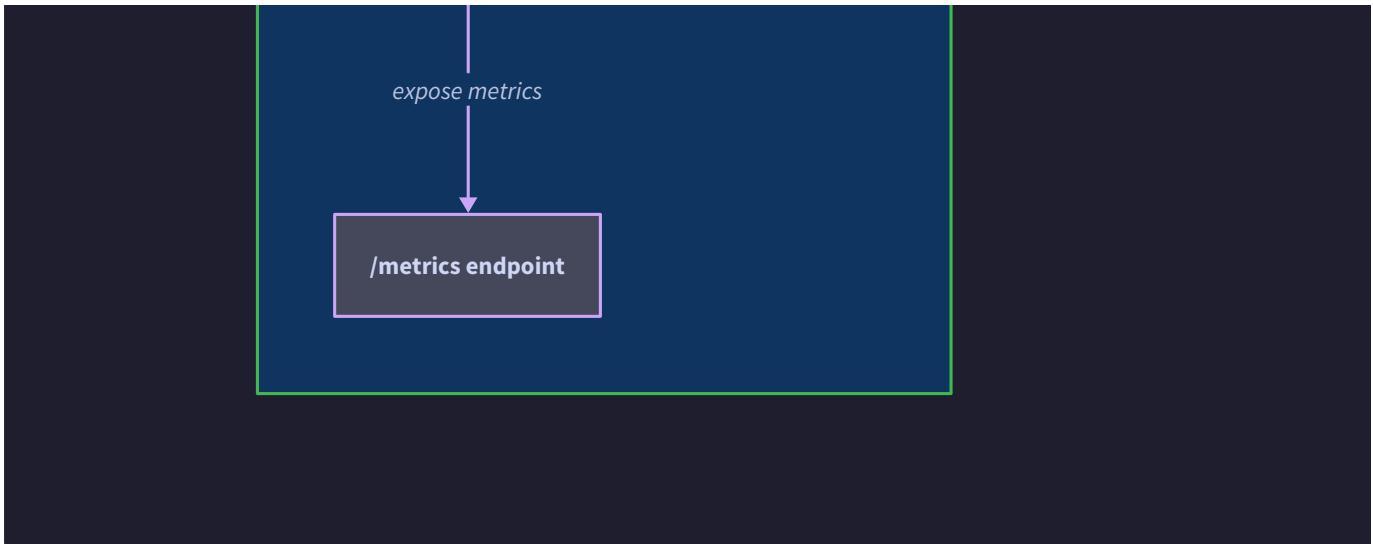
Think of our metrics collection system like a modern weather monitoring network. Weather stations (scrape targets) continuously measure temperature, humidity, and wind speed. Data collection trucks (scrape engine) visit each station on a schedule to gather measurements. The measurements are transported to a central archive (storage engine) where they're organized, compressed, and indexed for long-term preservation. Scientists and meteorologists (users) can then query the archive to analyze weather patterns, generate reports, and make predictions about future conditions.

This mental model captures the essential flow of our system: autonomous data generation at distributed endpoints, scheduled collection via pull-based mechanisms, efficient storage with temporal organization, and flexible querying for analysis and monitoring. Just as weather data becomes valuable when aggregated and analyzed over time, metrics data provides observability insights when collected systematically and made queryable.

Our architecture consists of four primary components that work together to implement this observability pipeline. Each component has distinct responsibilities and interfaces, but they're designed to work together seamlessly to provide a complete metrics collection and analysis solution.







Component Responsibilities

The system is decomposed into four major components, each with clearly defined responsibilities and boundaries. This separation of concerns enables independent development, testing, and scaling of each component while maintaining clean interfaces between them.

Scrape Engine Component

The **Scrape Engine** serves as the data acquisition layer of our system, responsible for discovering targets and collecting metrics from distributed endpoints. Think of it as a fleet of postal workers who know every address in the city, visit each location on a precise schedule, and collect all the mail waiting for pickup.

The scrape engine operates on a pull-based model, meaning it actively reaches out to configured targets rather than waiting for them to push data. This design choice provides several advantages: targets remain stateless and don't need to know about the collector, network failures are handled centrally, and the collector maintains complete control over data collection timing and frequency.

Responsibility	Description	Key Operations
Target Discovery	Locate scrape endpoints through static config or service discovery	Parse configuration files, query DNS records, interact with Kubernetes API
Scrape Scheduling	Execute HTTP requests to targets at configured intervals	Manage per-target timers, handle concurrent scraping, track scrape success/failure
Metrics Parsing	Parse Prometheus exposition format from HTTP response bodies	Tokenize text format, validate metric names and labels, extract timestamps
Health Monitoring	Track target availability and scrape success rates	Record response times, detect timeouts, maintain target status
Backpressure Control	Prevent overwhelming targets with too-frequent requests	Implement exponential backoff, respect rate limits, queue scrape requests

The scrape engine maintains its own configuration system with support for dynamic updates. When service discovery detects new targets, the engine automatically begins scraping them without requiring a restart. Similarly, when targets

disappear, scraping stops and resources are cleaned up.

Decision: Pull-Based Collection Model

- **Context:** Metrics can be collected via push (targets send data) or pull (collector retrieves data) models
- **Options Considered:** Push-based with target-initiated connections, pull-based with collector-initiated connections, hybrid approach
- **Decision:** Pull-based collection with HTTP scraping
- **Rationale:** Pull model provides better failure isolation (targets can't overwhelm collector), simpler target implementation (no need to know collector address), and centralized control over collection frequency and timeout handling
- **Consequences:** Requires targets to expose HTTP endpoints, collector must maintain target inventory, network failures affect data collection

Collection Model	Advantages	Disadvantages	Chosen?
Push-based	Targets control timing, works behind firewalls	Targets need collector address, backpressure harder to manage	No
Pull-based	Centralized control, simpler targets, better failure isolation	Requires HTTP endpoints, collector needs target discovery	Yes
Hybrid	Flexibility for different target types	Increased complexity, two codepaths to maintain	No

Time Series Storage Component

The **Storage Engine** functions as the system's memory and archive, responsible for persisting collected metrics efficiently and making them available for queries. Imagine a specialized library that stores millions of temperature readings, where each book represents a time series and pages contain chronologically ordered measurements compressed to save space.

The storage layer implements several sophisticated optimizations to handle the unique characteristics of time series data. Unlike traditional databases that optimize for random access patterns, time series data is almost always written in chronological order and queried by time ranges. This access pattern enables aggressive compression and specialized indexing strategies.

Responsibility	Description	Key Operations
Data Ingestion	Accept scraped metrics and store them durably	Validate metric format, assign timestamps, write to WAL, index updates
Compression	Reduce storage footprint using time series algorithms	Apply Gorilla compression, manage chunk boundaries, optimize encoding
Indexing	Maintain lookups for metric names and label combinations	Build inverted indexes, handle label cardinality, optimize query performance
Retention Management	Delete old data based on configured policies	Scan for expired data, compact storage files, update indexes
Query Processing	Retrieve time series data for specified time ranges and labels	Seek to time ranges, decompress data, filter by labels, return results

The storage engine uses a **Write-Ahead Log (WAL)** to ensure durability. Every metric write is first recorded in the WAL before being applied to the main storage structures. This guarantees that even if the system crashes during a write operation, the data can be recovered by replaying the WAL on startup.

Storage is organized into **time-based chunks** that contain compressed time series data for a specific time window (typically 2 hours). This chunking strategy optimizes both compression ratio and query performance by ensuring that data accessed together is stored together physically.

Decision: Gorilla-Style Compression

- **Context:** Time series data has high redundancy that can be exploited for compression
- **Options Considered:** Generic compression (gzip), column-oriented compression, Gorilla delta-of-delta compression
- **Decision:** Implement Gorilla compression with delta-of-delta timestamps and XOR value encoding
- **Rationale:** Gorilla compression achieves 12x space reduction on typical time series workloads, optimized specifically for time series patterns, and provides good query performance without full decompression
- **Consequences:** More complex implementation than generic compression, requires careful chunk boundary management, but provides significant storage savings

Compression Approach	Compression Ratio	Query Performance	Implementation Complexity	Chosen?
No compression	1x (baseline)	Fastest	Simplest	No
Generic (gzip)	3-5x	Slow (full decompression)	Simple	No
Gorilla algorithm	10-15x	Fast (partial decompression)	Complex	Yes

Query Engine Component

The **Query Engine** serves as the system's analytical brain, translating user queries into efficient data retrieval and processing operations. Think of it as a research librarian who understands both what researchers are looking for and the most efficient way to locate and combine information from the archive.

The query engine implements a PromQL-compatible query language that supports both instant queries (single point in time) and range queries (time series over a window). The engine's job is to parse these queries, plan efficient execution, retrieve the necessary data from storage, and perform any required aggregations or mathematical operations.

Responsibility	Description	Key Operations
Query Parsing	Convert PromQL text into executable query plans	Tokenize expressions, build AST, validate syntax, optimize plans
Series Selection	Find time series matching label selectors	Query storage indexes, apply label filters, handle regex matching
Data Retrieval	Fetch time series data from storage for specified time ranges	Issue storage queries, handle pagination, manage memory usage
Aggregation	Perform mathematical operations across time series	Implement sum/avg/max/min/count, handle grouping, compute rates
Result Formatting	Convert internal data structures to API response format	Serialize to JSON, apply time formatting, handle large result sets

The query engine is designed to be **stateless** - each query execution is independent and doesn't rely on cached state from previous queries. This design simplifies scaling and debugging but requires careful optimization to avoid repeatedly processing the same data.

Query execution follows a **pipeline model** where data flows through a series of processing stages: series selection, data retrieval, time alignment, aggregation, and result formatting. Each stage can be optimized independently and may process data in streaming fashion to reduce memory usage.

Decision: AST-Based Query Execution

- **Context:** PromQL queries need to be parsed and executed efficiently with proper operator precedence
- **Options Considered:** Direct interpreter, AST with visitor pattern, compiled query plans
- **Decision:** Abstract Syntax Tree with recursive evaluation
- **Rationale:** AST provides clear separation between parsing and execution, supports complex nested expressions, and enables query optimization passes
- **Consequences:** More complex than direct interpretation but provides better extensibility and optimization opportunities

HTTP API Server Component

The **HTTP Server** acts as the system's front door, providing REST endpoints for queries, configuration, and system status. It handles authentication, request routing, response formatting, and protocol translation between external HTTP clients and internal components.

The server component is relatively lightweight since most of the heavy lifting is done by the specialized engines. Its primary responsibilities center around protocol handling, request validation, and response formatting rather than core metrics processing.

Responsibility	Description	Key Operations
Request Routing	Direct incoming HTTP requests to appropriate handlers	Parse URLs, validate methods, route to query/config/status handlers
Query API	Expose PromQL query interface via HTTP endpoints	Parse query parameters, invoke query engine, format responses
Configuration API	Allow runtime configuration updates	Accept new scrape configs, validate settings, notify components
Metrics Exposition	Expose system's own metrics for monitoring	Generate internal metrics, format in Prometheus format
Error Handling	Provide consistent error responses and logging	Map internal errors to HTTP status codes, log requests

Data Flow Overview

Understanding how data moves through our system is crucial for both implementation and debugging. The data flow follows a clear pipeline from external targets through our components and back to users, with each stage transforming and enriching the data.

Metrics Collection Flow

The metrics collection flow begins when the scrape engine identifies targets through service discovery and executes HTTP requests to gather metrics data. This process runs continuously in the background according to configured intervals.

- 1. Target Discovery Phase:** The scrape engine queries configured service discovery backends (static files, DNS, Kubernetes API) to build a current list of scrape targets. Each target includes an endpoint URL, scrape interval, timeout, and optional labels.
- 2. Scrape Scheduling Phase:** A scheduler component maintains timing wheels for each target, triggering scrape operations according to their individual intervals. Multiple scrapes execute concurrently using worker pools to maximize throughput.
- 3. HTTP Collection Phase:** For each scrape, the engine sends an HTTP GET request to the target's metrics endpoint (typically `/metrics`). The request includes appropriate headers and handles authentication if configured.

4. **Metrics Parsing Phase:** The response body contains metrics in Prometheus exposition format (text-based). The parser tokenizes this content, extracting metric names, labels, values, and timestamps. Invalid metrics are logged and discarded.
5. **Storage Ingestion Phase:** Parsed metrics are sent to the storage engine, which validates them, assigns canonical timestamps, and writes them to the WAL. The data is then indexed and added to the appropriate time series chunks.
6. **Health Tracking Phase:** The scrape engine records success/failure status, response times, and error details for each scrape operation. This information is used for target health monitoring and alerting.

Stage	Input	Processing	Output	Failure Mode
Target Discovery	Service discovery configs	Query APIs, parse responses	Target list with endpoints	Service discovery unavailable
Scrape Scheduling	Target list, intervals	Manage timers, worker pools	Scrape tasks	Resource exhaustion
HTTP Collection	Target endpoints	HTTP GET requests	Raw metrics text	Network timeout, HTTP errors
Metrics Parsing	Metrics text	Tokenize, validate format	Structured metrics	Parse errors, invalid format
Storage Ingestion	Structured metrics	Write WAL, update indexes	Persisted time series	Disk full, corruption
Health Tracking	Scrape results	Record status, compute metrics	Target health status	Monitoring system failure

Query Processing Flow

The query processing flow is triggered when users submit PromQL queries via the HTTP API. This flow involves parsing the query, planning execution, retrieving data, and formatting results.

1. **Query Reception Phase:** The HTTP server receives a query request containing a PromQL expression, time range (for range queries), and optional parameters like timeout limits.
2. **Query Parsing Phase:** The query engine tokenizes the PromQL expression and builds an Abstract Syntax Tree (AST) representing the query structure. Syntax errors are detected and reported at this stage.
3. **Query Planning Phase:** The engine analyzes the AST to determine which time series need to be retrieved from storage, what time ranges are required, and what aggregations need to be performed.
4. **Series Selection Phase:** Using the storage engine's indexes, the system identifies all time series that match the query's label selectors. This may involve regex matching and label combination lookups.
5. **Data Retrieval Phase:** For each selected series, the storage engine retrieves data points within the query's time range. Gorilla compression is applied in reverse to decompress the stored data.

6. **Query Execution Phase:** The engine applies aggregation functions, mathematical operations, and grouping logic to compute the final results according to the PromQL expression.
7. **Result Formatting Phase:** The computed results are serialized into the appropriate response format (JSON for API queries) and returned to the client.

Stage	Input	Processing	Output	Performance Considerations
Query Reception	HTTP request	Validate parameters, parse time ranges	Query parameters	Request validation overhead
Query Parsing	PromQL text	Tokenize, build AST	Query execution plan	Parser complexity scales with query complexity
Query Planning	AST	Analyze requirements, optimize	Execution strategy	Planning time vs execution efficiency tradeoff
Series Selection	Label selectors	Index lookups, regex matching	Matching series list	Index efficiency critical for high cardinality
Data Retrieval	Series list, time range	Storage queries, decompression	Raw data points	I/O bound, compression CPU cost
Query Execution	Raw data, operations	Aggregation, math functions	Computed results	Memory usage scales with result size
Result Formatting	Computed results	Serialize to JSON	HTTP response	Serialization cost for large result sets

Deployment Architecture

The deployment architecture defines how our components are distributed across machines and how they scale to handle production workloads. We support multiple deployment patterns depending on scale requirements and operational constraints.

Single-Instance Deployment

For development, testing, and smaller production workloads, all components can be deployed within a single process on one machine. This deployment pattern minimizes operational complexity while providing full functionality.

In single-instance deployment, all components share the same process space and communicate via direct function calls rather than network protocols. The HTTP server, scrape engine, storage engine, and query engine all run as goroutines within the same binary.

Component	Resource Usage	Scaling Limit	Failure Impact
Scrape Engine	CPU-bound (HTTP clients)	~1000 targets per instance	All scraping stops
Storage Engine	Disk I/O and memory	~1M active series	All data inaccessible
Query Engine	CPU and memory	~100 concurrent queries	All queries fail
HTTP Server	Network and CPU	~1000 QPS	All API access lost

The single instance stores all data locally using disk-based storage with configurable data directories. Configuration is provided via YAML files that specify scrape targets, retention policies, and server settings.

Decision: Single-Process Architecture

- **Context:** Need to balance simplicity with scalability for initial implementation
- **Options Considered:** Microservices from the start, single process with internal interfaces, monolithic design
- **Decision:** Single process with clean component interfaces that can be split later
- **Rationale:** Reduces operational complexity, simplifies debugging, enables easier testing while maintaining clean boundaries for future scaling
- **Consequences:** Limited horizontal scalability but much simpler deployment and development

Configuration Management

The system uses a hierarchical configuration approach with YAML files, environment variables, and command-line flags. Configuration can be reloaded at runtime without requiring a restart, enabling dynamic updates to scrape targets and other settings.

The `Config` struct serves as the root configuration object, containing nested configuration for each component:

Configuration Section	Key Settings	Reload Behavior
<code>ScrapeConfig</code>	<code>scrape_interval</code> , <code>scrape_timeout</code> , <code>static_configs</code>	Dynamic reload, affects next scrape cycle
<code>StorageConfig</code>	<code>data_directory</code> , <code>retention_period</code> , <code>compression_enabled</code>	Partial reload, retention changes apply immediately
<code>QueryConfig</code>	<code>query_timeout</code> , <code>max_series</code> , <code>max_range_duration</code>	Dynamic reload, affects new queries
<code>ServerConfig</code>	<code>port</code> , <code>read_timeout</code> , <code>write_timeout</code>	Requires restart

Configuration validation occurs at startup and during reload operations. The `RequirementsValidator` checks that the system can meet specified performance, scalability, and reliability requirements given the current configuration and available resources.

Process Management and Health Monitoring

The system exposes its own metrics via HTTP endpoints, enabling monitoring of internal performance and health. Key metrics include scrape success rates, query latency, storage utilization, and error rates.

Health checks are available at multiple levels: individual component health (can the storage engine accept writes?), system health (are all components functioning?), and operational health (are performance targets being met?).

Health Check	Endpoint	Success Criteria	Failure Response
Liveness	/health/live	Process responding	HTTP 503, restart container
Readiness	/health/ready	All components initialized	HTTP 503, remove from load balancer
Component Health	/health/components	Each component passes self-check	HTTP 200 with component status

The system implements graceful shutdown procedures that allow in-flight operations to complete before termination. Query processing completes current requests, scraping finishes active scrapes, and storage flushes pending writes to disk.

Resource Requirements and Capacity Planning

Resource requirements scale primarily with the number of active time series (cardinality) and query load rather than the volume of individual metrics. A time series that receives one sample per minute versus one sample per second requires similar memory and storage resources due to compression efficiency.

Memory usage is dominated by the time series index and uncompressed data in active chunks. Each active time series requires approximately 1KB of memory for index structures plus a proportional share of chunk overhead.

Workload Characteristic	Memory Impact	Storage Impact	CPU Impact
High cardinality (many series)	Linear growth in index size	Linear growth in compressed data	Minimal
High sample rate	Minimal (better compression)	Sublinear growth	Linear in ingestion CPU
Complex queries	Temporary growth during execution	None	High CPU usage
Large time ranges	Memory for decompressed data	None	Decompression CPU cost

Storage requirements depend on retention period, compression efficiency, and cardinality. Gorilla compression typically achieves 12:1 reduction, so 1 million series with 1-minute resolution retained for 30 days requires approximately 180GB of disk space.

Common Pitfalls

⚠ Pitfall: Underestimating Label Cardinality Impact Label combinations multiply to create the total number of time series. Adding labels like `instance_id` or `user_id` can explode cardinality exponentially. Monitor the

number of active series and set cardinality limits to prevent memory exhaustion.

⚠ Pitfall: Inadequate Resource Allocation for Query Engine Complex PromQL queries can consume significant memory and CPU, especially when aggregating across high cardinality labels. Set query timeouts and memory limits, and monitor query performance to identify expensive patterns.

⚠ Pitfall: Ignoring Scrape Target Health Failed scrapes result in data gaps that affect query results and alerting. Implement proper monitoring of scrape success rates and set up alerting when targets become unavailable.

⚠ Pitfall: Insufficient Storage Planning Time series data grows continuously, and running out of disk space causes immediate system failure. Plan for at least 20% overhead beyond calculated requirements and implement monitoring of disk usage with automated alerting.

Implementation Guidance

The implementation approach balances simplicity for learning purposes with realistic production patterns. We'll use Go as the primary language due to its excellent concurrency support and HTTP capabilities.

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	net/http with custom routing	Gorilla Mux or Gin framework
Configuration	encoding/json with structs	YAML parsing with gopkg.in/yaml.v2
Storage Backend	Local filesystem with os package	Embedded database like BadgerDB
Compression	Custom Gorilla implementation	Existing library like prometheus/tsdb
Service Discovery	Static file-based configuration	Kubernetes client-go integration
Logging	Standard log package	Structured logging with logrus/zap

Recommended File Structure

Organize the codebase to reflect the component boundaries and support independent development:

```
metrics-collector/
├── cmd/
│   └── collector/
│       └── main.go
└── internal/
    ├── config/
    │   ├── config.go
    │   └── validator.go
    ├── scrape/
    │   ├── engine.go
    │   ├── target.go
    │   └── parser.go
    ├── storage/
    │   ├── engine.go
    │   ├── series.go
    │   ├── compression.go
    │   └── index.go
    ├── query/
    │   ├── engine.go
    │   ├── parser.go
    │   └── functions.go
    └── server/
        ├── server.go
        └── handlers.go
pkg/
├── metrics/
└── model/
configs/
└── example.yaml
```

← Entry point and CLI parsing
← Config structs and loading
← RequirementsValidator implementation
← Scrape scheduling and execution
← Target management and service discovery
← Prometheus format parsing
← Main storage interface
← Time series data structures
← Gorilla compression implementation
← Series indexing and lookup
← Query planning and execution
← PromQL parsing and AST
← Aggregation and math functions
← HTTP server and routing
← API endpoint handlers
← Shared metric type definitions
← Common data structures
← Example configuration file

Infrastructure Starter Code

Here's the basic HTTP server infrastructure that handles routing and graceful shutdown:

GO

```
// internal/server/server.go

package server

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

// Server wraps HTTP server for metrics endpoints

type Server struct {
    httpServer *http.Server
    mux        *http.ServeMux
}

// NewServer creates HTTP server on specified port

func NewServer(port int) *Server {
    mux := http.NewServeMux()

    return &Server{
        httpServer: &http.Server{
            Addr:         fmt.Sprintf(":%d", port),
            Handler:      mux,
            ReadTimeout:  30 * time.Second,
            WriteTimeout: 30 * time.Second,
        },
        mux: mux,
    }
}
```

```
// RegisterScrapeEndpoint adds metrics exposition endpoint

func (s *Server) RegisterScrapeEndpoint(path string, handler http.HandlerFunc) {
    s mux.HandleFunc(path, handler)
}

// Start begins serving HTTP requests

func (s *Server) Start() error {
    return s.httpServer.ListenAndServe()
}

// Shutdown gracefully stops server

func (s *Server) Shutdown(ctx context.Context) error {
    return s.httpServer.Shutdown(ctx)
}
```

Configuration loading infrastructure with validation:

GO

```
// internal/config/config.go

package config

import (
    "fmt"
    "io/ioutil"
    "time"
    "gopkg.in/yaml.v2"
)

// Default constants for configuration

const (
    DEFAULT_SCRAPE_INTERVAL     = 15 * time.Second
    DEFAULT_SCRAPE_TIMEOUT      = 10 * time.Second
    DEFAULT_RETENTION_PERIOD    = 30 * 24 * time.Hour // 30 days
    DEFAULT_QUERY_TIMEOUT        = 30 * time.Second
)

// Config is the root configuration structure

type Config struct {

    Scrape  ScrapeConfig `yaml:"scrape"`
    Storage StorageConfig `yaml:"storage"`
    Query   QueryConfig  `yaml:"query"`
    Server  ServerConfig `yaml:"server"`
}

// ScrapeConfig defines scraping parameters

type ScrapeConfig struct {

    ScrapeInterval time.Duration `yaml:"scrape_interval"`
    ScrapeTimeout  time.Duration `yaml:"scrape_timeout"`
}
```

```
    StaticConfigs []StaticConfig `yaml:"static_configs"`

}

// StorageConfig defines storage parameters

type StorageConfig struct {

    DataDirectory     string      `yaml:"data_directory"`

    RetentionPeriod  time.Duration `yaml:"retention_period"`

    CompressionEnabled bool       `yaml:"compression_enabled"`

}

// QueryConfig defines query engine parameters

type QueryConfig struct {

    QueryTimeout     time.Duration `yaml:"query_timeout"`

    MaxSeries        int          `yaml:"max_series"`

    MaxRangeDuration time.Duration `yaml:"max_range_duration"`

}

// ServerConfig defines HTTP server parameters

type ServerConfig struct {

    Port int `yaml:"port"`

}

// StaticConfig defines a static scrape target

type StaticConfig struct {

    Targets []string      `yaml:"targets"`

    Labels map[string]string `yaml:"labels"`

}

// LoadFromFile reads YAML configuration

func LoadFromFile(filename string) (*Config, error) {

    // TODO: Read file content using ioutil.ReadFile
```

```
// TODO: Parse YAML using yaml.Unmarshal  
  
// TODO: Call SetDefaults to populate missing values  
  
// TODO: Return parsed config or error  
  
}  
  
  
// SetDefaults populates default config values  
  
func (c *Config) SetDefaults() {  
  
    // TODO: Set default values for all duration and numeric fields  
  
    // TODO: Ensure required string fields have sensible defaults  
  
    // TODO: Apply defaults recursively to nested config structs  
  
}
```

Core Logic Skeleton Code

Here are the method signatures for the core components that learners should implement:

GO

```
// internal/scrape/engine.go

package scrape

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

// ScrapeEngine manages target discovery and metric collection

type ScrapeEngine struct {
    client    *http.Client
    targets   map[string]*Target
    // TODO: Add fields for worker pools, scheduling, health tracking
}

// NewScrapeEngine creates a new scrape engine

func NewScrapeEngine() *ScrapeEngine {
    return &ScrapeEngine{
        client:  &http.Client{Timeout: DEFAULT_SCRAPE_TIMEOUT},
        targets: make(map[string]*Target),
    }
}

// UpdateTargets refreshes the target list from configuration

func (e *ScrapeEngine) UpdateTargets(configs []StaticConfig) error {
    // TODO 1: Parse static configs into Target structs with URLs and labels
    // TODO 2: Compare new targets with existing targets map
    // TODO 3: Add new targets and remove obsolete targets
    // TODO 4: Schedule scraping for new targets
}
```

```
// TODO 5: Update target labels for existing targets

}

// scrapeTarget performs HTTP request to collect metrics from one target

func (e *ScrapeEngine) scrapeTarget(target *Target) error {

    // TODO 1: Create HTTP GET request with proper headers and timeout

    // TODO 2: Execute request and handle network errors

    // TODO 3: Read response body and validate HTTP status code

    // TODO 4: Parse metrics from response body using exposition format

    // TODO 5: Send parsed metrics to storage engine

    // TODO 6: Update target health status based on success/failure

}
```

GO

```
// internal/storage/engine.go

package storage

import (
    "time"
)

// StorageEngine manages time series persistence and retrieval

type StorageEngine struct {

    dataDir    string
    retention time.Duration

    // TODO: Add fields for WAL, indexes, compression, chunks
}

// NewStorageEngine creates a storage engine

func NewStorageEngine(config StorageConfig) *StorageEngine {
    // TODO: Initialize storage directories, WAL, indexes
}

// Append adds new samples to time series

func (e *StorageEngine) Append(samples []Sample) error {
    // TODO 1: Validate samples have valid timestamps and metric names
    // TODO 2: Write samples to WAL for durability
    // TODO 3: Group samples by time series (metric name + labels)
    // TODO 4: Add samples to appropriate time series chunks
    // TODO 5: Update inverted indexes for new label combinations
    // TODO 6: Trigger compression for completed chunks
}

// Select retrieves time series data matching label selectors
```

```

func (e *StorageEngine) Select(start, end time.Time, matchers []LabelMatcher) (SeriesSet, error)
{
    // TODO 1: Use inverted indexes to find series matching label selectors

    // TODO 2: Determine which chunks overlap with [start, end] time range

    // TODO 3: Read and decompress relevant chunks from disk

    // TODO 4: Filter samples to exact time range

    // TODO 5: Return SeriesSet iterator over matching time series

}

```

Language-Specific Hints

Go-Specific Implementation Tips:

- Use `sync.RWMutex` for concurrent access to the target map in the scrape engine
- Implement graceful shutdown using `context.Context` and `sync.WaitGroup` for worker goroutines
- Use `time.Ticker` for scrape scheduling rather than `time.Sleep` in loops
- Apply `os.File.Sync()` after WAL writes to ensure durability
- Use `encoding/binary` for efficient serialization of timestamps and float values
- Implement the `sort.Interface` for time series to enable efficient range queries
- Use `regexp.Compile` once at startup for label regex matchers, not on every query

Error Handling Patterns:

- Wrap errors with context using `fmt.Errorf("scraping target %s: %w", url, err)`
- Use sentinel errors like `var ErrTargetTimeout = errors.New("target timeout")` for specific failures
- Implement retry logic with exponential backoff for transient network failures
- Log errors at appropriate levels: network timeouts as warnings, configuration errors as errors

Milestone Checkpoints

After Milestone 1 (Metrics Data Model):

- Run: `go test ./internal/metrics/...`
- Expected: All metric type tests pass, Counter increases monotonically, Gauge accepts any value
- Manual verification: Create Counter and Gauge, observe values, confirm label attachment works
- Success indicator: Can create metrics with labels and retrieve current values

After Milestone 2 (Scrape Engine):

- Run: `go run cmd/collector/main.go --config=configs/example.yaml`
- Expected: See periodic scrape logs, target health status updates
- Manual verification: Configure a target, observe HTTP requests in target logs
- Success indicator: Metrics are successfully parsed from HTTP endpoints

After Milestone 3 (Time Series Storage):

- Run: `go test ./internal/storage/... -v`
- Expected: Compression tests show >10x space reduction, retention policies delete old data
- Manual verification: Write samples, restart process, verify data persists
- Success indicator: Data survives restarts and compression reduces storage size

After Milestone 4 (Query Engine):

- Run: Query API endpoint: `curl "http://localhost:9090/api/v1/query?query=up"`
- Expected: JSON response with metric values and timestamps
- Manual verification: Try range queries, aggregation functions, label filtering
- Success indicator: PromQL queries return correct results matching stored data

Metrics Data Model

Milestone(s): This section directly corresponds to Milestone 1 (Metrics Data Model) and establishes the foundational data structures that will be used throughout Milestones 2-4.

The metrics data model forms the conceptual and technical foundation upon which the entire monitoring system is built. Think of it as the vocabulary and grammar of our observability language - just as human language needs nouns, verbs, and sentence structure to convey meaning, our metrics system needs well-defined data types, labeling semantics, and identification rules to effectively communicate system behavior over time.

Understanding the metrics data model is crucial because every decision made here ripples through the entire system architecture. The choice of metric types determines what kinds of mathematical operations the query engine can perform. The labeling system directly impacts storage cardinality and memory usage. The time series identity model affects indexing strategies and query performance. Get these fundamentals wrong, and the entire system becomes either unusable due to poor performance or unreliable due to semantic inconsistencies.



Metric Type Semantics

Think of metric types as different kinds of measuring instruments in a scientific laboratory. A thermometer measures absolute temperature at a point in time (like a gauge), while a Geiger counter accumulates radiation exposure over time (like a counter). Each instrument has specific mathematical properties that determine what calculations make sense - you can subtract two temperature readings to find a delta, but subtracting Geiger counter readings would be meaningless without considering the time periods involved.

Our metrics system supports four fundamental metric types, each with distinct semantic behaviors that enable different kinds of analysis and alerting patterns.

Decision: Four-Type Metric Model

- **Context:** Need to support diverse monitoring use cases from simple resource metrics to complex latency distributions while maintaining clear semantic boundaries
- **Options Considered:**
 1. Simple gauge-only model (Graphite-style)
 2. Four-type model (Counter, Gauge, Histogram, Summary)
 3. Extended model with additional types (Sets, Traces)
- **Decision:** Four-type model matching Prometheus specification
- **Rationale:** Balances expressiveness with implementation complexity, provides clear semantic guarantees for rate calculations and aggregations, widely understood by practitioners
- **Consequences:** Query engine must understand type-specific operations, storage must preserve type information, scraping must validate type consistency

Metric Type	Mathematical Properties	Use Cases	Invalid Operations
Counter	Monotonically increasing, resets to zero	Request counts, error counts, bytes transferred	Decrease operations, negative values, arbitrary sets
Gauge	Arbitrary value, can increase/decrease	CPU usage, memory usage, queue depth, temperature	Rate calculations without smoothing
Histogram	Distribution with predefined buckets	Request latency, response sizes, batch sizes	Bucket boundary changes after creation
Summary	Distribution with calculated quantiles	Similar to histogram but client-calculated percentiles	Server-side aggregation across instances

Counter Semantics and Behavior

Counters represent cumulative totals that only increase over time, with the critical exception of resets to zero when the monitored process restarts. The mathematical foundation of counters enables powerful rate calculations - the derivative of a counter represents the instantaneous rate of change, which is often more meaningful than the absolute value for operational monitoring.

The semantic contract of a `Counter` includes several critical guarantees that both the client instrumenting code and the query engine must respect. First, values must never decrease except during resets to zero. Second, counter resets must be detectable by the monitoring system to avoid incorrect rate calculations. Third, the rate of change is more operationally significant than the absolute value in most use cases.

Counter Operation	Input	Behavior	Semantic Guarantee
Inc()	None	Increment by 1	Value increases monotonically
Add(value)	Positive number	Increment by value	Rejects negative values
Reset()	None	Set to zero	Marks reset boundary for rate calculations
Value()	None	Return current total	Read-only access to current state

Counter resets present a particular challenge for rate calculations. When a process restarts, the counter begins again from zero, creating an artificial negative spike in the rate calculation if handled naively. The storage and query engine must detect these reset conditions and handle them appropriately by treating the first sample after a reset as a new baseline rather than calculating a rate from the previous higher value.

Consider a web server request counter that increments with each HTTP request. Over one hour, it increases from 1000 to 1500 requests, indicating a rate of 500 requests per hour. If the server restarts and the counter resets to zero, then increases to 100 over the next hour, the rate should be calculated as 100 requests per hour, not as -1400 requests per hour based on the naive difference from the pre-restart value.

The fundamental insight with counters is that the absolute value is rarely interesting - you care about the rate of change over time. A counter showing 1,847,392 total requests tells you nothing actionable, but knowing that requests are arriving at 50 per second tells you everything about current system load.

Gauge Semantics and Flexibility

Gauges represent point-in-time measurements that can fluctuate arbitrarily in either direction. Unlike counters, gauges have no mathematical constraints on their values - they can increase, decrease, or remain constant between observations. This flexibility makes them suitable for representing resource levels, percentages, temperatures, and any other measurement where the current absolute value is meaningful.

The semantic contract of a **Gauge** emphasizes current state over historical accumulation. Operations focus on setting, adjusting, and observing the current value rather than accumulating changes over time. This distinction is crucial for query operations - while rate calculations on gauges are mathematically valid, they often require smoothing or windowing to be operationally useful due to the potentially noisy nature of gauge values.

Gauge Operation	Input	Behavior	Use Case
Set(value)	Any number	Replace current value	Setting absolute measurements
Inc()	None	Increment by 1	Simple upward adjustments
Dec()	None	Decrement by 1	Simple downward adjustments
Add(value)	Any number (including negative)	Adjust by delta	Relative adjustments
Value()	None	Return current value	Current state queries

Gauges excel at representing system state that fluctuates around operational ranges. CPU utilization naturally varies between 0% and 100%, memory usage grows and shrinks with allocation patterns, and queue depths rise and fall with load patterns. These measurements have meaningful absolute values at any point in time, unlike counters where only the rate of change provides operational insight.

Consider monitoring database connection pool usage. The gauge might show 15 active connections out of a maximum 50, providing immediate insight into resource utilization. As queries complete and new requests arrive, the gauge fluctuates, with both the current absolute value (15 connections) and the trend over time (increasing, decreasing, or stable) providing valuable operational information.

Histogram Design and Bucket Strategy

Histograms capture the distribution of observed values by maintaining counts in predefined buckets, enabling calculation of percentiles, averages, and distribution shapes without storing individual samples. This aggregation approach provides powerful statistical insights while maintaining bounded memory usage regardless of observation volume.

The core design challenge with histograms lies in choosing appropriate bucket boundaries. These boundaries must be defined when the histogram is created and cannot be changed later, as the bucketing strategy affects all subsequent statistical calculations. The bucket boundaries determine the precision and range of percentile calculations - finer buckets provide higher precision but consume more storage, while coarser buckets reduce storage at the cost of statistical accuracy.

Histogram Component	Purpose	Storage Requirement	Query Capability
Bucket counters	Count observations in each range	One counter per bucket	Percentile estimation, distribution shape
Total count	Count all observations	One counter	Sample count for averaging
Total sum	Sum all observed values	One counter	Mean calculation
Bucket boundaries	Define ranges for categorization	Metadata only	Query planning and validation

The histogram bucket strategy directly impacts the accuracy of percentile calculations. Consider measuring HTTP response latency with buckets at [0.1s, 0.5s, 1.0s, 2.0s, 5.0s, +Inf]. This configuration provides good resolution for typical web response times but would poorly serve a system where most responses complete in microseconds. The query engine estimates percentiles using bucket interpolation - if the 95th percentile falls within the 1.0s-2.0s bucket, the exact value is estimated based on the distribution assumption within that range.

Decision: Cumulative Histogram Buckets

- **Context:** Need efficient percentile calculations while supporting aggregation across multiple instances
- **Options Considered:**
 1. Individual bucket counts (each bucket independent)
 2. Cumulative bucket counts (each bucket includes all smaller values)
 3. Sparse histogram with dynamic bucket boundaries
- **Decision:** Cumulative bucket counts following Prometheus model
- **Rationale:** Enables efficient percentile calculation via bucket search, supports mathematical aggregation across instances, simplifies query engine implementation
- **Consequences:** Slightly more complex bucket increment logic, but dramatically simplified percentile queries and cross-instance aggregation

Histogram aggregation across multiple instances provides one of the most powerful features of this metric type.

Unlike gauges or counters where cross-instance aggregation requires careful consideration of meaning (average of averages vs. total counts), histogram buckets aggregate naturally - the sum of bucket counts across instances represents the combined distribution. This property enables fleet-wide latency analysis and capacity planning based on aggregate behavior patterns.

Summary Metrics and Client-Side Quantiles

Summaries provide an alternative approach to distribution tracking by calculating quantiles (percentiles) on the client side and transmitting pre-calculated statistical values rather than bucket counts. This approach trades some query flexibility for reduced bandwidth and storage requirements, particularly valuable in high-volume environments where network efficiency is paramount.

The fundamental difference between summaries and histograms lies in where the statistical calculation occurs.

Histograms preserve the raw distribution information (via bucket counts) and calculate percentiles during query execution, while summaries calculate percentiles during observation and store only the results. This trade-off has profound implications for aggregation capabilities and query flexibility.

Summary Component	Purpose	Client Calculation	Server Storage
Quantile values	Pre-calculated percentiles (e.g., 0.5, 0.9, 0.99)	Sliding window quantile estimation	Direct storage of calculated values
Total count	Count all observations	Simple counter increment	Counter storage and rate calculations
Total sum	Sum all observed values	Running sum accumulation	Mean calculation support
Observation window	Time window for quantile calculation	Sliding window management	Window metadata only

The critical limitation of summaries becomes apparent during aggregation operations. Since quantiles are non-additive mathematical functions, combining the 95th percentile from multiple instances does not yield the fleet-wide 95th percentile. If instance A reports a 95th percentile latency of 500ms and instance B reports 750ms, the combined fleet 95th percentile could be anywhere from 500ms to 750ms (or even outside this range) depending on the distribution of requests across instances.

Summaries excel in bandwidth-constrained environments where you need specific quantiles from individual instances but don't require fleet-wide distribution analysis. Histograms provide superior query flexibility and aggregation capabilities at the cost of higher bandwidth and storage requirements.

Multi-Dimensional Labeling

Think of labels as the coordinate system that transforms flat metrics into a multi-dimensional space where you can slice, dice, and aggregate data along any combination of dimensions. Just as a GPS coordinate becomes meaningful only when you know it refers to latitude and longitude, a metric value becomes operationally useful only when you know the service, environment, region, and other contextual dimensions that produced it.

Labels enable the transformation from simple time series (`metric_name -> value over time`) to multi-dimensional time series (`metric_name{label1=value1, label2=value2} -> value over time`). This dimensionality is what makes modern monitoring systems powerful - instead of creating separate metrics for each combination of conditions, you create one metric with appropriate labels and query across dimensions.

The labeling system must balance expressiveness with performance. Each unique combination of label values creates a distinct time series, directly impacting memory usage, storage requirements, and query performance. Understanding this relationship is crucial for designing sustainable monitoring instrumentation that scales with system complexity.

Label Structure and Naming Conventions

Labels consist of key-value pairs where both keys and values are strings, attached to metric observations to provide dimensional context. The label key represents the dimension name (such as "method", "status_code", or "region") while the label value represents the specific instance of that dimension (such as "GET", "200", or "us-west-2").

Label naming conventions significantly impact long-term maintainability and query ergonomics. Well-chosen label names create intuitive query patterns and support natural aggregation operations, while poor label naming leads to confusion and complex query logic. The naming strategy should reflect the operational questions you need to answer rather than the technical implementation details of how metrics are collected.

Label Category	Examples	Purpose	Cardinality Impact
Service Identity	<code>service</code> , <code>instance</code> , <code>job</code>	Identify metric source	Linear with service count
Request Context	<code>method</code> , <code>status_code</code> , <code>endpoint</code>	Categorize individual operations	Multiplicative across dimensions
Infrastructure	<code>region</code> , <code>availability_zone</code> , <code>datacenter</code>	Physical deployment context	Linear with infrastructure diversity
Application State	<code>version</code> , <code>environment</code> , <code>feature_flag</code>	Application configuration context	Linear with deployment variations

The hierarchical nature of many label dimensions enables powerful aggregation patterns. Consider HTTP request metrics labeled with `{service="api", method="GET", endpoint="/users", status_code="200"}`. You can aggregate across all endpoints to see service-level request rates, across all status codes to see endpoint-level traffic patterns, or across all methods to analyze API endpoint popularity. Each aggregation operation reduces dimensionality while preserving the ability to drill down into specific label combinations.

Label value consistency across the system requires careful coordination between instrumentation and operational practices. The same conceptual entity must use identical label values across all metrics - a service identified as "user-service" in one metric and "userservice" in another creates artificial separation in queries and dashboards. This consistency extends beyond naming to include value normalization (lowercase vs. uppercase, hyphen vs. underscore) and handling of dynamic values (user IDs, request IDs) that create unbounded cardinality.

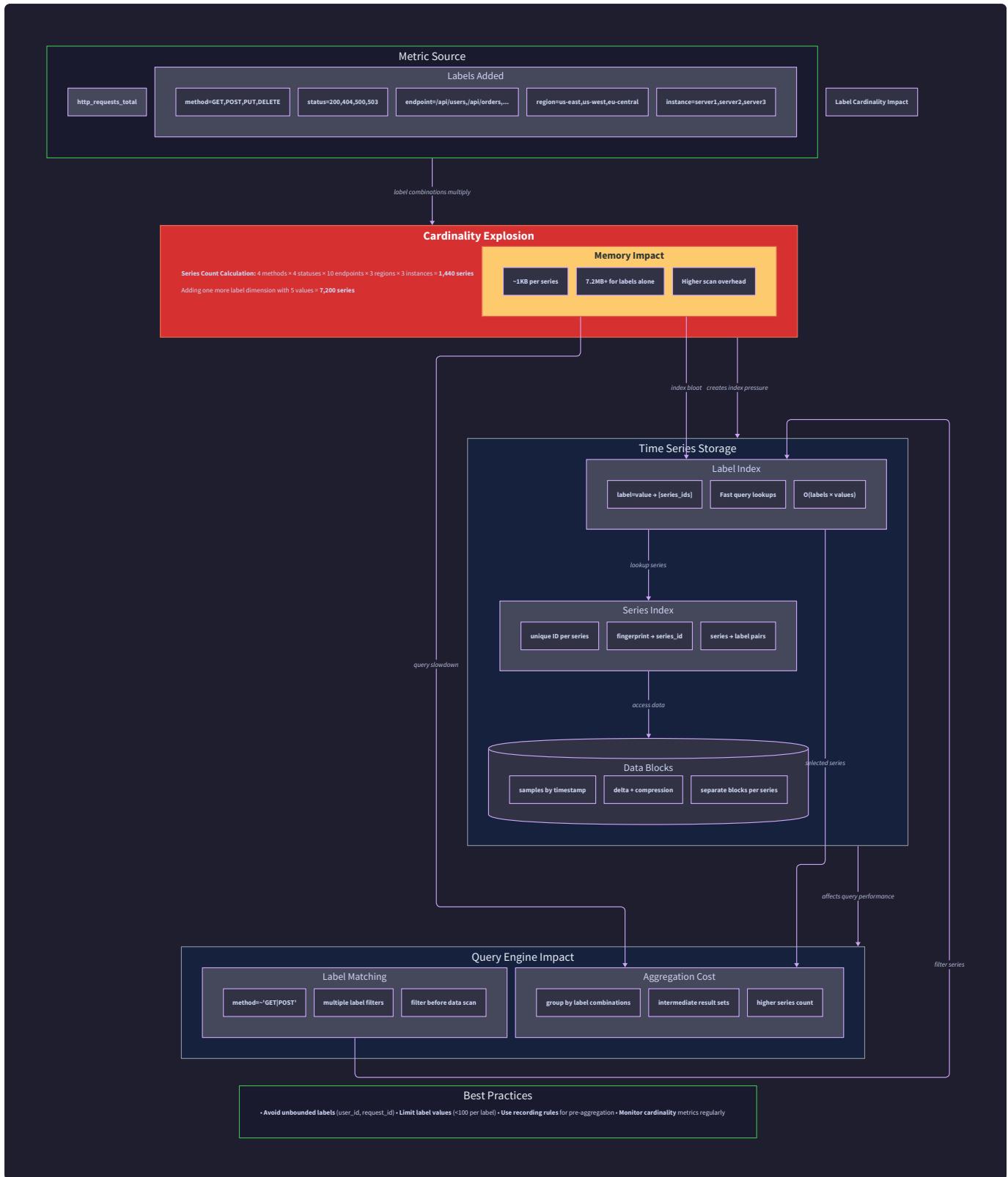
Cardinality Mathematics and Memory Impact

Label cardinality represents the number of unique time series created by all possible combinations of label values for a given metric. Understanding cardinality mathematics is essential for predicting memory usage, storage requirements, and query performance as the system scales.

The cardinality of a metric equals the cartesian product of all label value sets. A metric with labels `{service, method, status_code}` where service has 10 possible values, method has 4 values, and status_code has 15 values creates a maximum cardinality of $10 \times 4 \times 15 = 600$ unique time series. In practice, not all combinations may exist (some services might not support all methods), but the maximum provides the upper bound for capacity planning.

Cardinality Factor	Low Impact (1-10 values)	Medium Impact (10-100 values)	High Impact (100+ values)	Unbounded (avoid)
Examples	HTTP methods, status code classes	Status codes, service names	Instance IDs, container names	User IDs, request IDs, timestamps
Memory per series	~1KB baseline + samples	~1KB baseline + samples	~1KB baseline + samples	Unbounded growth
Query performance	Minimal impact	Linear degradation	Significant index overhead	System instability
Storage growth	Predictable	Manageable with planning	Requires careful monitoring	Leads to system failure

The memory impact of cardinality extends beyond simple multiplication due to indexing overhead. Each unique time series requires index entries for fast lookup during queries, and these indexes must support efficient filtering across multiple label dimensions simultaneously. The storage engine maintains inverted indexes mapping label values to time series identifiers, creating memory overhead that scales with both cardinality and label diversity.



Consider a real-world example: instrumenting HTTP request duration with labels for service, method, endpoint, and status_code. With 5 services, 4 HTTP methods, 20 endpoints per service, and 10 status codes, the theoretical maximum cardinality is $5 \times 4 \times 20 \times 10 = 4,000$ time series. If each time series consumes approximately 1KB of memory for metadata plus sample storage, this single metric could consume 4MB of memory just for the index structures, before considering the actual time series data.

Decision: Label Cardinality Limits

- **Context:** Need to prevent unbounded memory growth while supporting necessary operational dimensions
- **Options Considered:**
 1. No limits (trust users to instrument responsibly)
 2. Hard limits per metric (e.g., max 1000 series per metric name)
 3. Soft limits with warnings and graduated enforcement
- **Decision:** Soft limits with configurable enforcement thresholds
- **Rationale:** Provides safety against cardinality explosion while allowing legitimate high-cardinality use cases with explicit acknowledgment
- **Consequences:** Requires monitoring of cardinality growth, adds complexity to ingestion pipeline, enables sustainable scaling

Label Best Practices and Anti-Patterns

Effective label design requires understanding the difference between dimensions that add operational value and those that add only noise. The goal is to create labelsets that enable meaningful aggregation and filtering operations while maintaining reasonable cardinality bounds.

High-value labels represent dimensions along which you regularly need to aggregate, filter, or alert. These typically correspond to operational boundaries (services, environments, regions) or request characteristics that affect system behavior (HTTP methods, cache hit/miss status, error types). Low-value labels often represent implementation details that don't align with operational questions or create unnecessarily high cardinality without proportional insight.

Pattern Type	Good Practice	Anti-Pattern	Impact
Service Identity	<pre>{service="user-api", environment="prod"}</pre>	<pre>{hostname="server-17-prod-usw2.internal"}</pre>	Service focus vs. infrastructure focus
Request Classification	<pre>{method="GET", status_class="2xx"}</pre>	<pre>{full_url="/api/users/12345/profile"}</pre>	Bounded vs. unbounded cardinality
Error Categorization	<pre>{error_type="timeout", subsystem="database"}</pre>	<pre>{error_message="connection refused to 10.0.0.1:5432"}</pre>	Actionable categories vs. specific instances
Version Tracking	<pre>{version="1.2.3", deployment_id="abc123"}</pre>	<pre>{build_timestamp="2023-10-15T14:30:22Z"}</pre>	Discrete versions vs. continuous values

The temporal aspect of label values requires special consideration. Labels that change frequently create natural time series boundaries - when a label value changes, a new time series begins and the old one effectively ends. This behavior is correct and desired for legitimate operational dimensions (like application version during deployments) but problematic for high-frequency changes (like current timestamp or active user count).

Common anti-patterns include using user IDs, request IDs, or timestamps as label values. These create unbounded cardinality that grows continuously with system usage rather than stabilizing at a level proportional to system complexity. Instead, these high-cardinality identifiers should be either excluded from metrics entirely or aggregated into bounded categories (e.g., `user_type` instead of `user_id`, `request_size_bucket` instead of `request_id`).

The litmus test for label appropriateness is simple: "Will I ever want to aggregate or filter metrics along this dimension?" If you can't imagine writing a query that groups by or filters on a label, it probably shouldn't be a label.

Time Series Identity

Think of time series identity as the unique "address" that allows the storage and query engines to locate specific metric streams within the vast multidimensional space of all possible measurements. Just as a postal address must uniquely identify a specific building, a time series identity must uniquely identify a specific sequence of timestamped values among potentially millions of similar sequences.

The time series identity model determines how the system partitions the continuous stream of metric observations into discrete, queryable sequences. This partitioning directly affects storage layout, query performance, and cardinality management. Understanding identity semantics is crucial because it defines the granularity at which the system can filter, aggregate, and analyze metric data.

Identity Composition and Uniqueness

A time series identity consists of the metric name combined with the complete set of label key-value pairs. Two time series are considered identical if and only if they have the same metric name and exactly the same set of labels with exactly the same values. Any difference in metric name, label keys, or label values creates a distinct time series identity.

This strict equality requirement has important implications for instrumentation consistency. A metric observation with labels `{service="api", method="GET"}` creates a different time series than an observation with labels `{method="GET", service="api"}` even though the label content is semantically identical - the system treats these as separate identities. Most implementations normalize label ordering to avoid this pitfall, but the fundamental principle remains: identity requires exact matching.

Identity Component	Contribution	Example	Uniqueness Impact
Metric name	Primary classification	<code>http_requests_total</code>	Separates different measurement types
Label keys	Dimensional structure	<code>method, status_code</code>	Defines available aggregation dimensions
Label values	Specific instances	<code>GET, 200</code>	Creates actual time series instances
Label ordering	Normalized during ingestion	Consistent regardless of input order	Implementation detail, not semantic

The mathematical relationship between labels and time series count becomes clear through the identity model. Each unique combination of label values creates one time series identity. If you have a metric

`http_requests_total{method, status_code}` and observe requests with methods [GET, POST] and status codes [200, 404, 500], you create $2 \times 3 = 6$ distinct time series identities: `{method="GET", status_code="200"} , {method="GET", status_code="404"} , {method="GET", status_code="500"} , {method="POST", status_code="200"} , {method="POST", status_code="404"} , {method="POST", status_code="500"} .`

Identity Lifecycle and Creation

Time series identities come into existence dynamically as the system observes new combinations of metric names and label values. Unlike traditional databases where schema defines the available table and column structure upfront, metrics systems create new time series identities on-demand as applications emit previously unseen labelset combinations.

This dynamic creation model provides tremendous flexibility for evolving systems - new services, endpoints, or operational dimensions automatically create appropriate time series without schema migration or configuration changes. However, it also creates the risk of cardinality explosion if instrumentation code generates unbounded label values or fails to normalize label naming.

Lifecycle Stage	Trigger	System Action	Performance Impact
Creation	First sample with new identity	Allocate storage, create index entries	Memory allocation, index update overhead
Active	Ongoing sample ingestion	Append samples to existing series	Minimal per-sample overhead
Inactive	No samples for retention period	Mark for deletion, preserve for queries	Index overhead without storage growth
Deletion	Retention policy expiration	Remove from storage and indexes	Memory reclamation, index cleanup

The moment of time series creation represents the highest overhead in the identity lifecycle. The storage engine must allocate memory structures, create index entries mapping labels to series identifiers, and update various metadata structures to track the new series. This creation overhead motivates batching strategies where multiple samples for the same identity are grouped together during ingestion.

Consider the lifecycle of a time series tracking HTTP requests for a new API endpoint. The first request to `/api/users` with method GET creates the time series identity `http_requests_total{method="GET", endpoint="/api/users"}`. Subsequent requests to the same endpoint with the same method append samples to this existing time series. If the endpoint is later deprecated and receives no traffic, the time series becomes inactive but remains queryable for historical analysis until the retention policy removes it entirely.

Identity Normalization and Canonical Form

To ensure consistent identity matching across system components, the metrics system must establish a canonical form for time series identities. This normalization process converts various equivalent representations into a single, standard format that enables reliable identity comparison and lookup operations.

Label ordering normalization represents the most common identity canonicalization requirement. Since labels are conceptually an unordered set of key-value pairs, the system must establish a consistent ordering (typically lexicographic by key name) to ensure that `{method="GET", service="api"}` and `{service="api", method="GET"}` resolve to the same canonical identity.

Normalization Type	Input Variation	Canonical Form	Purpose
Label ordering	<code>{b="2", a="1"}</code>	<code>{a="1", b="2"}</code>	Consistent identity hashing and comparison
Label key casing	<code>{Method="GET"}</code>	<code>{method="GET"}</code>	Case-insensitive label key matching
Value whitespace	<code>{status=" 200 "}</code>	<code>{status="200"}</code>	Trim accidental whitespace
Empty labels	<code>{method="GET", unused=""}</code>	<code>{method="GET"}</code>	Remove labels with empty values

The canonical identity form enables efficient storage and lookup operations through consistent hashing. The storage engine can hash the canonical identity string to determine storage location, index bucket, and cache keys without worrying about equivalent representations creating different hash values. This consistency is crucial for performance as the system scales to millions of time series.

Identity normalization must balance consistency with preservation of meaningful distinctions. While trimming whitespace from label values usually represents error correction, case-sensitive label values might be semantically important (distinguishing between SQL table names "Users" and "users" in case-sensitive databases). The normalization rules should reflect the operational reality of the monitored systems rather than imposing arbitrary formatting requirements.

Decision: Strict Identity Immutability

- **Context:** Need to ensure consistent time series identification across storage, indexing, and query operations
- **Options Considered:**
 1. Mutable identities (allow label value changes for existing series)
 2. Immutable identities (label changes create new time series)
 3. Partial mutability (allow changes to designated "mutable" labels)
- **Decision:** Strict immutability - any label change creates a new time series
- **Rationale:** Simplifies storage engine design, ensures query consistency, prevents data corruption from identity conflicts
- **Consequences:** Application label changes require explicit time series migration, but system behavior remains predictable and reliable

Cardinality Control

Think of cardinality control as the immune system of the metrics infrastructure - it protects the overall system health by identifying and containing threats to stability before they can cause widespread damage. Just as a biological immune system must distinguish between beneficial bacteria and harmful pathogens, cardinality control must differentiate between legitimate high-dimensional metrics and pathological cardinality explosions.

Uncontrolled cardinality growth represents one of the most common causes of metrics system failure in production environments. A single misbehaving application or poorly designed instrumentation can generate millions of unique time series in minutes, consuming all available memory and rendering the entire monitoring system unusable. Effective cardinality control requires both preventive measures (limits and validation) and reactive measures (detection and mitigation).

Cardinality Explosion Detection

Cardinality explosion typically manifests as rapid, unexpected growth in the number of unique time series, often accompanied by degraded query performance and memory pressure. Early detection requires monitoring the rate of new time series creation and identifying patterns that indicate problematic instrumentation rather than legitimate system growth.

The challenge in explosion detection lies in distinguishing between normal system evolution (new services, features, or infrastructure) and pathological growth (unbounded label values, instrumentation bugs). Normal growth typically correlates with planned deployments or infrastructure changes and exhibits predictable patterns. Pathological growth often appears sudden, accelerating, and disproportionate to actual system complexity changes.

Detection Signal	Normal Growth Pattern	Explosion Pattern	Response Required
New series rate	Gradual, correlated with deployments	Sudden spike, continuously accelerating	Immediate investigation and mitigation
Label value diversity	Bounded growth in known dimensions	New dimensions or unbounded values	Label audit and validation
Memory usage growth	Linear with feature complexity	Exponential growth unrelated to features	Emergency cardinality limiting
Query performance	Stable or gradually degrading	Rapid degradation, timeout increases	Query load balancing and limiting

Automated detection systems should monitor both absolute cardinality levels and growth rates across multiple time horizons. A metric that creates 1000 new time series in one minute might represent normal behavior for a high-traffic service during deployment, but the same rate sustained over an hour indicates a serious problem requiring immediate intervention.

The distribution of cardinality across metrics provides additional detection signals. In healthy systems, most metrics have relatively low cardinality (10-100 time series), with a few high-cardinality metrics (1000+ series) that represent well-understood, business-critical dimensions. An explosion often manifests as a single metric suddenly dominating the cardinality budget, indicating a specific instrumentation problem rather than general system growth.

Enforcement Strategies and Policies

Effective cardinality enforcement requires a graduated response system that can provide early warnings, impose soft limits during normal operation, and implement hard limits during emergency conditions. This approach balances the need for operational safety with the flexibility to support legitimate high-cardinality use cases when properly justified and monitored.

Soft enforcement mechanisms focus on visibility and warnings rather than blocking operations. These approaches work well during normal operations when development teams can respond to notifications and adjust instrumentation practices. Hard enforcement mechanisms prioritize system stability over metric completeness, appropriate during crisis situations where the monitoring system's survival takes precedence over comprehensive data collection.

Enforcement Level	Trigger Conditions	Actions Taken	Operational Impact
Warning	50% of cardinality budget used	Log warnings, send alerts	No impact on data collection
Soft limiting	80% of cardinality budget used	Rate limit new series creation	Delayed ingestion, potential data loss
Hard limiting	95% of cardinality budget used	Reject new series, shed existing series	Guaranteed data loss, preserved system stability
Emergency mode	Memory pressure or system instability	Aggressive series eviction, ingestion throttling	Significant data loss, system preservation

The enforcement policy must define clear cardinality budgets and allocation strategies across different metric types and operational contexts. Production services might receive larger cardinality budgets than development environments, and business-critical metrics might be protected from enforcement actions that could affect their availability during incidents.

Sample-based enforcement provides a middle ground between complete rejection and unlimited acceptance. When soft limits are exceeded, the system can randomly sample new time series creation, preserving statistical representativeness while controlling absolute cardinality growth. This approach works particularly well for metrics where complete enumeration isn't required for operational insight.

Decision: Hierarchical Cardinality Budgets

- **Context:** Need to prevent system-wide cardinality explosion while allowing different services and metrics to have different cardinality requirements
- **Options Considered:**
 1. Global cardinality limit (single system-wide limit)
 2. Per-metric cardinality limits (each metric has independent limit)
 3. Hierarchical budgets (service -> metric -> label dimension limits)
- **Decision:** Hierarchical budgets with inheritance and override capabilities
- **Rationale:** Provides granular control while enabling reasonable defaults, supports organizational responsibility boundaries, enables emergency override for critical metrics
- **Consequences:** More complex configuration and monitoring, but much better operational control and blast radius limitation

Label Value Validation and Sanitization

Proactive label value validation provides the first line of defense against cardinality explosion by identifying and rejecting problematic label values before they create persistent time series. Effective validation requires understanding common patterns that lead to unbounded cardinality and implementing sanitization rules that preserve operational meaning while enforcing cardinality bounds.

High-risk label patterns include sequential identifiers (user IDs, request IDs, timestamps), unbounded categorical values (error messages, URLs with parameters), and accidentally dynamic values (configuration changes, memory addresses). Validation rules should detect these patterns and either reject the labels entirely or transform them into bounded categories.

Validation Rule	Pattern Detected	Action Taken	Example
Sequential numbers	Label values matching <code>/^\d+\$/</code>	Reject or convert to ranges	user_id="12345" → user_type="premium"
Timestamp values	ISO timestamp patterns	Extract time component	timestamp="2023-10-15T14:30:22Z" → hour="14"
URL paths	HTTP URL patterns with parameters	Extract endpoint pattern	path="/users/123/profile" → endpoint="/users/{id}/profile"
Error messages	Long strings with variable content	Extract error category	message="Connection failed to 10.0.0.1" → error_type="connection_failed"
Excessive length	Label values over N characters	Truncate or reject	Very long values usually indicate misuse

Label value allowlists and denylists provide explicit control over acceptable values for high-risk dimensions. Critical operational labels like service names, environments, or regions benefit from explicit allowlists that prevent typos and unauthorized values from creating unexpected cardinality. Dynamic labels that are known to be problematic can be explicitly blocked through denylists.

The sanitization process must balance data preservation with cardinality control. Overly aggressive sanitization can remove legitimate operational dimensions, reducing the system's monitoring effectiveness. Conversely, insufficient sanitization allows cardinality explosions that threaten system stability. The validation rules should be tunable based on operational experience and regularly reviewed as instrumentation practices evolve.

Memory Management and Series Eviction

When cardinality control measures fail to prevent memory pressure, the system must implement series eviction strategies that preserve the most operationally valuable time series while reclaiming memory from less critical ones. Effective eviction requires understanding the relative importance of different time series and implementing policies that maintain monitoring effectiveness during resource constraints.

Eviction strategies must consider both recency and operational importance when selecting series for removal. Recently active time series are more likely to be relevant for current operational questions, but historical data for critical business metrics may be more valuable than recent data for debugging metrics. The eviction policy should reflect organizational priorities and monitoring use cases.

Eviction Strategy	Selection Criteria	Advantages	Disadvantages
Least Recently Used (LRU)	Time since last sample	Simple to implement, preserves active series	May evict important historical data
Least Frequently Used (LFU)	Sample count over time window	Preserves high-volume series	Complex tracking, biased toward noisy metrics
Priority-based	Explicit priority labels or metric patterns	Aligns with business priorities	Requires manual priority assignment
Random eviction	Random selection among candidates	Unbiased, statistically representative	May evict critical series by chance

The eviction process should be gradual and observable to prevent sudden monitoring capability loss during incidents. Aggressive eviction during a production outage could remove exactly the time series needed for root cause analysis, creating a double failure where the monitoring system fails simultaneously with the monitored system.

Memory management extends beyond series eviction to include sample retention policies within individual time series. High-cardinality metrics might benefit from shorter sample retention periods, preserving the ability to create new time series while limiting the historical depth available for queries. This approach maintains monitoring coverage while managing memory growth over time.

⚠ Pitfall: Emergency Eviction During Incidents During production incidents, memory pressure often triggers aggressive eviction exactly when monitoring data is most critical. Design eviction policies to preserve incident-relevant metrics (error rates, latency, resource usage) even under extreme memory pressure. Consider pre-defining "incident mode" eviction policies that protect critical operational metrics at the expense of development or experimental metrics.

Common Pitfalls

⚠ Pitfall: User ID as Labels Adding user IDs, request IDs, or other unbounded identifiers as metric labels creates unlimited cardinality that grows continuously with system usage. A metric labeled with `{user_id="user_12345"}` creates one time series per user, potentially millions of series that consume memory and degrade query performance. Instead, use bounded categorical labels like `{user_type="premium", region="us-west"}` that provide operational insight without explosive cardinality growth.

⚠ Pitfall: Inconsistent Label Naming Using different label names or values for the same conceptual entity across metrics creates artificial separation during queries and aggregation. If one metric uses `{service="api"}` while another uses `{svc="api"}` or `{service="API"}`, queries cannot correlate the metrics without complex label transformation. Establish and enforce consistent label naming conventions across all instrumentation to enable natural metric correlation and aggregation.

⚠ Pitfall: Counter Reset Handling Failing to properly handle counter resets leads to incorrect rate calculations that show impossible negative rates or dramatic spikes. When a monitored process restarts, counters reset to zero, but naive rate calculations compare the new zero value against the previous higher value. Implement counter reset

detection in the query engine to treat post-reset samples as new baselines rather than continuing from pre-reset values.

⚠ Pitfall: Histogram Bucket Changes Changing histogram bucket boundaries after deployment creates inconsistent percentile calculations and breaks historical trend analysis. If you initially configure latency buckets as [0.1s, 0.5s, 1.0s] but later realize you need finer resolution and change to [0.05s, 0.1s, 0.2s, 0.5s, 1.0s], the old and new data become incomparable. Plan histogram bucket boundaries carefully based on expected data distributions and avoid changes that break historical continuity.

⚠ Pitfall: High-Cardinality Error Messages Using complete error messages as label values creates unbounded cardinality since error messages often contain variable information like timestamps, IDs, or network addresses. Instead of `{error="Connection timeout to server-17 at 14:32:15"}`, use categorized error types like `{error_type="connection_timeout", subsystem="database"}` that provide actionable operational insight without cardinality explosion.

Implementation Guidance

This implementation guidance provides concrete Go code structures and examples to transform the design concepts into working software. The focus is on creating type-safe, efficient implementations that enforce the semantic guarantees described in the design while providing clear interfaces for the scraping and query engines.

Technology Recommendations

Component	Simple Option	Advanced Option
Metric Storage	In-memory maps with mutex protection	Lock-free concurrent data structures
Label Validation	Regex-based pattern matching	Compiled finite state automata
Identity Hashing	Standard library SHA-256	xxhash or similar fast hash functions
Memory Management	Go garbage collector with manual monitoring	Custom memory pools with explicit lifecycle
Serialization	JSON for simplicity	Protocol Buffers for efficiency

Recommended File Structure

```
internal/metrics/
    types.go           ← Core metric type definitions
    labels.go          ← Label handling and validation
    identity.go        ← Time series identity management
    cardinality.go     ← Cardinality control and enforcement
    registry.go        ← Metric registration and lookup
    types_test.go       ← Comprehensive type behavior tests
    examples/
        instrumentation.go ← Example usage patterns
```

Core Metric Type Infrastructure

```
// Package metrics provides the foundational data model for time series metrics          GO
// with support for counters, gauges, histograms and multi-dimensional labeling.

package metrics

import (
    "fmt"
    "math"
    "sort"
    "sync"
    "time"
)

// Sample represents a timestamped value in a time series

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

// Labels represents the multi-dimensional label set that identifies a time series.

// Labels are stored as a sorted slice of key-value pairs for efficient comparison

// and consistent iteration order.

type Labels []LabelPair

type LabelPair struct {
    Name  string `json:"name"`
    Value string `json:"value"`
}

// String returns the canonical string representation of the labelset

func (ls Labels) String() string {
```

```
// TODO 1: Sort labels by name to ensure consistent representation

// TODO 2: Format as {name1="value1", name2="value2"}

// TODO 3: Escape quotes and backslashes in values

// TODO 4: Return empty string for empty labelset

}

// Hash returns a consistent hash of the labelset for use in maps and indexing

func (ls Labels) Hash() uint64 {

    // TODO 1: Create canonical string representation

    // TODO 2: Use a fast hash function (xxhash recommended)

    // TODO 3: Ensure consistent hash values for equivalent labelsets

}

// Counter represents a monotonically increasing metric that only goes up

type Counter struct {

    mu     sync.RWMutex

    value float64

    // TODO: Add creation timestamp for reset detection

}

// Inc increments the counter by 1

func (c *Counter) Inc() {

    // TODO 1: Acquire write lock

    // TODO 2: Increment value by 1

    // TODO 3: Release lock

}

// Add increments the counter by the given value

func (c *Counter) Add(value float64) error {

    // TODO 1: Validate value is non-negative
```

```
// TODO 2: Acquire write lock

// TODO 3: Add value to current total

// TODO 4: Release lock

// TODO 5: Return error for negative values

}

// Value returns the current counter value

func (c *Counter) Value() float64 {

    // TODO 1: Acquire read lock

    // TODO 2: Read current value

    // TODO 3: Release lock and return value

}

// Gauge represents a metric that can go up or down arbitrarily

type Gauge struct {

    mu     sync.RWMutex

    value float64

}

// Set sets the gauge to the given value

func (g *Gauge) Set(value float64) {

    // TODO 1: Acquire write lock

    // TODO 2: Set value to provided input

    // TODO 3: Release lock

}

// Inc increments the gauge by 1

func (g *Gauge) Inc() {

    // TODO 1: Acquire write lock

    // TODO 2: Increment value by 1
```

```
// TODO 3: Release lock
}

// Dec decrements the gauge by 1

func (g *Gauge) Dec() {
    // TODO 1: Acquire write lock
    // TODO 2: Decrement value by 1
    // TODO 3: Release lock
}

// Add adds the given value to the gauge (can be negative)

func (g *Gauge) Add(value float64) {
    // TODO 1: Acquire write lock
    // TODO 2: Add value to current gauge value
    // TODO 3: Release lock
}

// Value returns the current gauge value

func (g *Gauge) Value() float64 {
    // TODO 1: Acquire read lock
    // TODO 2: Read current value
    // TODO 3: Release lock and return value
}

// Histogram tracks the distribution of observations in predefined buckets

type Histogram struct {
    mu      sync.RWMutex
    buckets []float64 // Bucket upper bounds (sorted)
    counts  []uint64  // Observation counts per bucket
    sum     float64   // Sum of all observed values
}
```

```
    count  uint64    // Total number of observations

}

// NewHistogram creates a histogram with the specified bucket boundaries

func NewHistogram(buckets []float64) *Histogram {
    // TODO 1: Validate buckets are sorted and finite

    // TODO 2: Add +Inf bucket if not present

    // TODO 3: Initialize count slice to match bucket count

    // TODO 4: Return configured histogram

}

// Observe records an observation in the appropriate bucket

func (h *Histogram) Observe(value float64) {
    // TODO 1: Acquire write lock

    // TODO 2: Find appropriate bucket using binary search

    // TODO 3: Increment bucket count and total count

    // TODO 4: Add value to running sum

    // TODO 5: Release lock

}

// GetBuckets returns the current bucket counts and boundaries

func (h *Histogram) GetBuckets() ([]float64, []uint64) {
    // TODO 1: Acquire read lock

    // TODO 2: Copy bucket boundaries and counts to avoid race conditions

    // TODO 3: Release lock and return copies

}
```

Label Management and Validation

```
// LabelValidator enforces cardinality control and naming conventions
```

```
type LabelValidator struct {  
    maxLabelLength     int  
    maxValueLength    int  
    allowedPatterns   map[string]*regexp.Regexp  
    forbiddenPatterns []*regexp.Regexp  
}  
  
// NewLabelValidator creates a validator with default rules  
  
func NewLabelValidator() *LabelValidator {  
    // TODO 1: Initialize with reasonable defaults (label length < 64, value length < 256)  
    // TODO 2: Compile common forbidden patterns (sequential IDs, timestamps)  
    // TODO 3: Set up standard allowed patterns for common label names  
    // TODO 4: Return configured validator  
}  
  
// ValidateLabels checks a labelset for cardinality and naming violations  
  
func (lv *LabelValidator) ValidateLabels(labels Labels) error {  
    // TODO 1: Check total number of labels against limit  
    // TODO 2: Validate each label name and value length  
    // TODO 3: Check for forbidden patterns (user IDs, timestamps, etc.)  
    // TODO 4: Verify label names match allowed character sets  
    // TODO 5: Return descriptive errors for violations  
}  
  
// SanitizeLabels attempts to fix common labeling mistakes  
  
func (lv *LabelValidator) SanitizeLabels(labels Labels) Labels {  
    // TODO 1: Trim whitespace from names and values  
    // TODO 2: Convert label names to lowercase
```

GO

```

    // TODO 3: Remove empty label values

    // TODO 4: Truncate overly long values with warning

    // TODO 5: Sort labels for canonical form

}

// CardinalityTracker monitors time series creation and enforces limits

type CardinalityTracker struct {

    mu          sync.RWMutex

    seriesCounts map[string]int // metric name -> series count

    totalSeries   int

    maxTotalSeries int

    maxPerMetric   int

    warningCallback func(string, int, int) // metric, current, limit

}

// RecordSeries notifies the tracker of a new time series creation

func (ct *CardinalityTracker) RecordSeries(metricName string, labels Labels) error {

    // TODO 1: Acquire write lock

    // TODO 2: Check against per-metric and total limits

    // TODO 3: Increment appropriate counters

    // TODO 4: Trigger warnings if thresholds exceeded

    // TODO 5: Return error if hard limits violated

}

```

Milestone Checkpoints

Checkpoint 1: Basic Metric Types Run `go test ./internal/metrics/types_test.go` to verify:

- Counter increments correctly and rejects negative values
- Gauge accepts arbitrary values and supports increment/decrement operations
- Histogram correctly categorizes observations into buckets
- All operations are thread-safe under concurrent access

Expected test output:

```
==> RUN TestCounterIncrement
--- PASS: TestCounterIncrement (0.00s)
==> RUN TestGaugeOperations
--- PASS: TestGaugeOperations (0.00s)
==> RUN TestHistogramBuckets
--- PASS: TestHistogramBuckets (0.01s)
```

Checkpoint 2: Label Handling Create a simple program that demonstrates label cardinality:

```
func main() {  
  
    // Create metrics with different label combinations  
  
    // Verify that {service="api", method="GET"} and {method="GET", service="api"}  
  
    // create the same time series identity  
  
    // Show cardinality explosion with a loop creating user_id labels  
  
}
```

GO

You should see consistent identity hashing regardless of label order, and cardinality tracking should detect rapid series creation.

Checkpoint 3: Validation and Limits Test cardinality enforcement by instrumenting a metric with rapidly changing label values:

```
curl -X POST http://localhost:8080/metrics \  
-d 'metric_name=test_metric&user_id=12345&timestamp=2023-10-15T14:30:22Z'
```

BASH

The system should reject this request with a cardinality violation error, demonstrating effective protection against unbounded label values.

Scrape Engine Design

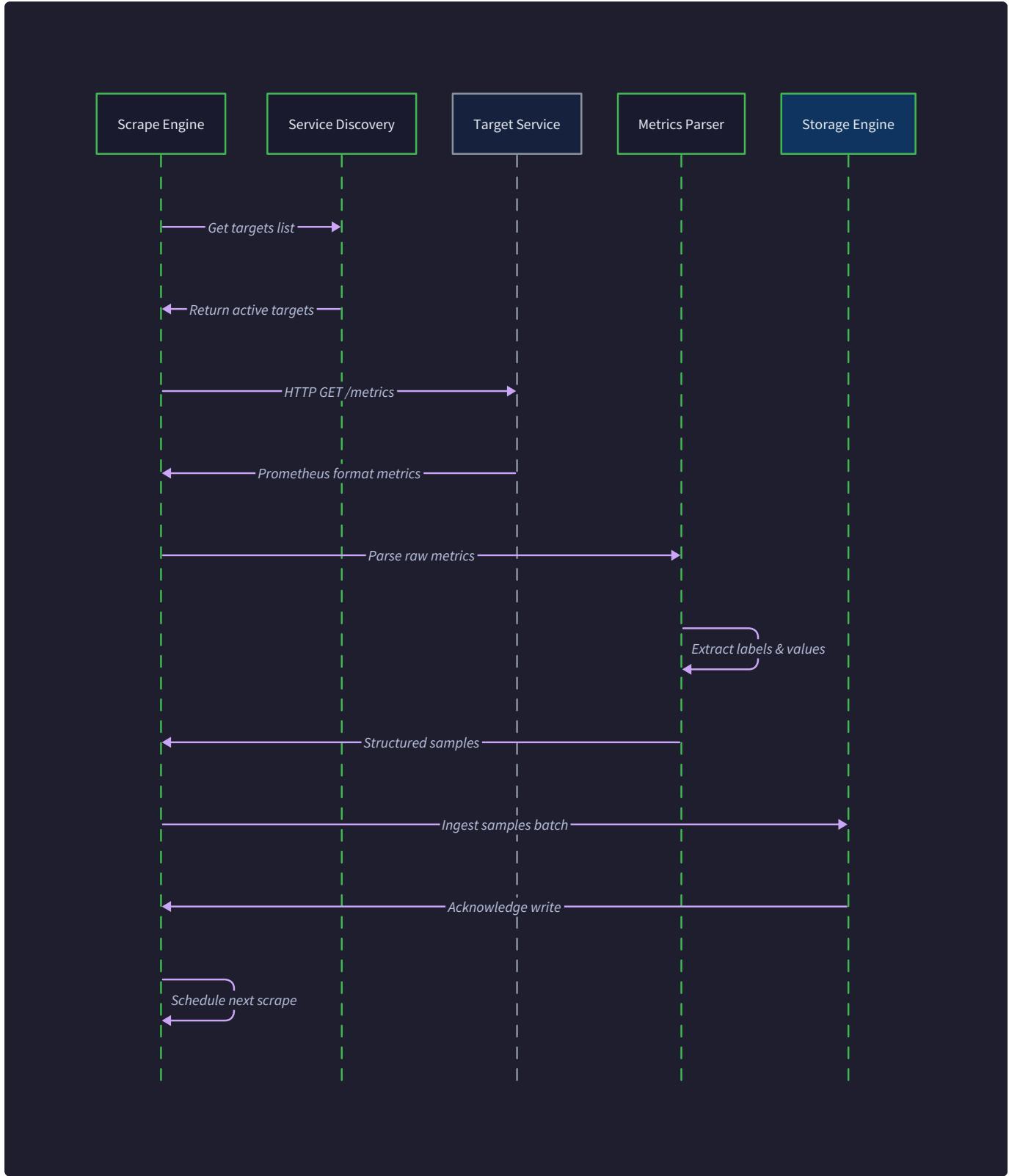
Milestone(s): This section directly corresponds to Milestone 2 (Scrape Engine) and provides the HTTP-based metrics collection system that feeds data into the storage layer from Milestone 3.

The Observatory Network Mental Model

Before diving into the technical details of scraping, consider how the National Weather Service operates thousands of weather stations across a country. Each station is a **scrape target** that measures temperature, humidity, and wind speed at regular intervals. The central weather service doesn't wait for stations to call in—instead, it actively contacts each station every hour to collect readings. Some stations are permanently configured (static discovery), while others are mobile units that register and deregister dynamically (service discovery). When a station goes offline due to equipment failure or network issues, the central service marks it as unavailable but continues attempting to

reconnect. This is exactly how our scrape engine works: it maintains a registry of metric-producing targets, actively pulls data from each one on a schedule, and gracefully handles failures while preserving the overall collection process.

The scrape engine serves as the **data ingestion coordinator** that bridges the gap between distributed services exposing metrics and our centralized time series storage. Unlike push-based systems where applications actively send metrics to a collector, our pull-based approach gives the metrics system complete control over collection timing, failure handling, and resource management. This architectural choice provides several key advantages: the scrape engine can implement sophisticated retry logic without overwhelming targets, it can discover new targets automatically through service discovery, and it can apply consistent labeling and metadata enrichment across all collected metrics.



The scrape engine operates through four tightly coordinated subsystems that work together to provide reliable metrics collection. The **target discovery system** maintains an up-to-date registry of all metric endpoints, automatically adding newly deployed services and removing decommissioned ones. The **scrape scheduler** manages the timing and concurrency of collection operations, ensuring each target is scraped at its configured interval without overwhelming either the scrape engine or the target services. The **metrics parser** handles the complex task of converting HTTP response bodies in Prometheus exposition format into structured time series samples that can be

stored efficiently. Finally, the **health management system** tracks the availability of each target, implements retry logic for transient failures, and provides observability into the scraping process itself.

The critical insight for pull-based scraping is that the metrics system becomes the **authoritative source of timing**. Unlike push-based systems where applications control when metrics are sent, our scrape engine determines exactly when each measurement is taken. This provides much stronger guarantees about data consistency and collection reliability.

Target Discovery

Target discovery solves the fundamental question of "which endpoints should I scrape for metrics?" In modern distributed systems, services are constantly being deployed, scaled, and decommissioned across multiple hosts and containers. Static configuration files become outdated quickly and create operational overhead. Our target discovery system supports both static configuration for stable infrastructure and dynamic service discovery for ephemeral workloads.

Decision: Hybrid Discovery Model

- **Context:** Modern deployments mix stable infrastructure (databases, load balancers) with dynamic workloads (microservices, containers). Pure static configuration requires manual updates, while pure dynamic discovery lacks control over stable services.
- **Options Considered:** Static-only configuration, dynamic-only service discovery, hybrid approach supporting both
- **Decision:** Implement hybrid discovery supporting both static targets and pluggable service discovery backends
- **Rationale:** Static configuration provides explicit control and reliability for infrastructure components, while dynamic discovery automatically handles ephemeral services without operational overhead
- **Consequences:** Increased complexity in target management but operational flexibility for mixed environments

Discovery Method	Use Cases	Update Frequency	Configuration Complexity
Static Configuration	Databases, load balancers, core infrastructure	Manual updates only	Low - direct endpoint lists
DNS Service Discovery	Services with stable DNS names	DNS TTL intervals	Medium - DNS queries and caching
Kubernetes Service Discovery	Containerized microservices	Real-time via API	High - API authentication and filtering

The **static configuration system** reads target lists from YAML files that specify exact endpoints, scrape intervals, and additional labels to attach to all metrics from each target. This approach works well for infrastructure components that have predictable network addresses and don't frequently change. The configuration supports grouping targets

with similar characteristics and applying common labels that help identify the service, environment, or datacenter in query results.

Static configuration follows this structure for maximum flexibility:

Field	Type	Description
targets	[]string	List of host:port endpoints to scrape
labels	map[string]string	Additional labels attached to all metrics from these targets
scrape_interval	duration	How frequently to collect metrics (defaults to system setting)
scrape_timeout	duration	Maximum time to wait for HTTP response
metrics_path	string	HTTP path for metrics endpoint (default: /metrics)
scheme	string	HTTP or HTTPS protocol (default: http)

The **dynamic service discovery system** integrates with external service registries to automatically detect new targets and remove decommissioned ones. Each service discovery backend runs as an independent goroutine that maintains its own view of available targets and publishes changes through a unified target update interface. This design allows multiple discovery mechanisms to operate simultaneously—for example, DNS discovery for legacy services and Kubernetes API discovery for containerized workloads.

Service discovery implementations must satisfy the `TargetDiscoverer` interface:

Method	Parameters	Returns	Description
Discover	ctx context.Context	<-chan []*Target	Returns channel streaming target updates
Stop	none	none	Gracefully shuts down discovery process

DNS-based service discovery queries SRV records to find service endpoints automatically. Many service mesh and load balancer systems publish SRV records that contain both the service port and priority information. The DNS discoverer performs periodic queries based on configurable intervals and TTL values, automatically adding new instances when they appear in DNS and removing them when they're no longer returned. This approach works particularly well for services that register themselves in DNS or are managed by orchestration systems that update DNS records.

Kubernetes service discovery uses the Kubernetes API to watch for pod and service changes in real-time. The discoverer connects to the Kubernetes API server and establishes watch streams for pods with specific annotations or labels that indicate they expose metrics. When new pods are scheduled or existing pods terminate, the API server immediately pushes updates through the watch stream. This provides much faster target updates compared to polling-based approaches and reduces the delay between service deployment and metrics collection.

The Kubernetes discoverer supports sophisticated **target filtering** through label selectors and namespace restrictions:

Filter Type	Configuration	Purpose
Label Selector	<code>kubernetes_sd_configs.selector</code>	Only discover pods/services matching specific labels
Namespace Filter	<code>kubernetes_sd_configs.namespaces</code>	Restrict discovery to specific Kubernetes namespaces
Port Filter	<code>kubernetes_sd_configs.port_name</code>	Select specific named ports from multi-port pods
Annotation Requirements	Custom annotations	Require specific annotations to enable scraping

All discovered targets flow through a **target consolidation process** that merges static and dynamic sources into a unified target registry. The consolidator handles conflicts when the same endpoint appears in multiple sources, applies target-specific configuration overrides, and maintains a consistent view of active targets across the entire scrape engine. This process runs continuously, updating the active target list whenever any discovery source reports changes.

⚠ Pitfall: Target Flapping When service discovery systems report rapid add/remove cycles for the same endpoint (often due to health check failures or network issues), the scrape engine can waste resources constantly starting and stopping scrape goroutines. The consolidator should implement **target stability windows** that require a target to remain stable for a minimum duration before activating scraping. This prevents resource thrashing while still responding quickly to legitimate topology changes.

Scrape Scheduling

The scrape scheduler orchestrates the complex task of collecting metrics from potentially thousands of targets simultaneously while respecting individual scrape intervals, timeouts, and resource constraints. Unlike simple cron-style scheduling, metrics scraping requires **adaptive scheduling** that can handle varying response times, target failures, and system load while maintaining consistent collection intervals for accurate time series analysis.

Think of the scrape scheduler as an **air traffic control system** managing hundreds of flights (scrape operations) that must take off (start) at precise times, follow specific routes (HTTP collection), and land (complete) within strict deadlines (timeouts). Just as air traffic control prevents collisions and manages delays, the scrape scheduler prevents resource conflicts and manages scrape timing to optimize both accuracy and system performance.

The core scheduling challenge is **interval drift prevention**. If a target is configured for 15-second scrapes but the HTTP request takes 2 seconds to complete, the next scrape should start 15 seconds after the previous scrape began, not 15 seconds after it completed. This ensures consistent sampling intervals that preserve the mathematical properties required for rate calculations and trend analysis. Naive implementations that wait for completion before scheduling the next scrape gradually drift away from their intended intervals.

Decision: Per-Target Goroutine Model

- **Context:** Need to scrape hundreds of targets concurrently while maintaining precise intervals and independent timeout handling for each target
- **Options Considered:** Global worker pool with shared queue, per-target goroutines, hybrid approach with target groups
- **Decision:** Dedicate one goroutine per active target for independent scheduling and lifecycle management
- **Rationale:** Goroutines are lightweight (8KB stack), provide natural isolation for timeouts and cancellation, and eliminate complex queue management logic
- **Consequences:** Higher memory usage with many targets but much simpler concurrent programming model and better isolation

Each target gets its own **scrape goroutine** that manages the complete lifecycle of that target's metric collection. The goroutine maintains a timer for the next scrape interval, handles HTTP requests with proper timeout context, parses the response, and forwards samples to storage. When a target is removed from discovery, its goroutine receives a cancellation signal and terminates cleanly. This model provides excellent isolation—a hanging HTTP request to one target cannot block scraping of other targets.

The per-target scheduling algorithm follows this precise sequence:

1. **Initialize interval timer** using the target's configured scrape interval (e.g., 15 seconds)
2. **Wait for timer expiration** or cancellation signal from target discovery updates
3. **Record scrape start time** to maintain consistent interval timing regardless of request duration
4. **Create HTTP request context** with scrape timeout deadline (e.g., 10 seconds maximum)
5. **Execute HTTP GET request** to the target's metrics endpoint with timeout context
6. **Parse response body** into time series samples if HTTP request succeeds
7. **Forward samples to storage** with additional target labels and scrape timestamp
8. **Update target health metrics** based on success/failure outcome and response time
9. **Calculate next scrape time** by adding interval to start time (not completion time)
10. **Reset interval timer** to maintain consistent scheduling and repeat the cycle

Timeout handling deserves special attention because it directly impacts both data quality and resource utilization. Each scrape operation runs within a context that automatically cancels after the configured timeout period. When cancellation occurs, the HTTP client immediately closes the connection and returns an error. The scrape goroutine records this as a timeout failure, updates the target's health status, and continues with its normal scheduling cycle. Importantly, timeouts do not delay subsequent scrapes—the next scrape timer is based on the start time, not the timeout completion time.

Timeout Scenario	Behavior	Next Scrape Timing	Health Impact
Request completes in 2s	Normal processing, forward samples to storage	15s after start time	Mark target healthy
Request times out after 10s	Cancel HTTP context, record timeout error	15s after start time	Mark target unhealthy
Target unreachable	Immediate connection error	15s after start time	Mark target unreachable
Invalid metrics format	HTTP succeeds but parsing fails	15s after start time	Mark target returning bad data

Concurrency control prevents the scrape engine from overwhelming either itself or target services with too many simultaneous requests. The scheduler implements several layers of protection: a global semaphore limits total concurrent scrapes across all targets, per-target state tracking prevents multiple simultaneous scrapes of the same endpoint, and adaptive backoff reduces scrape frequency for consistently failing targets.

The global concurrency limiter uses a **weighted semaphore** that accounts for the expected resource cost of different types of scrapes:

```
Max Concurrent Scrapes = min(
    configured_max_concurrent,
    available_memory / avg_scrape_memory,
    network_connections / 2 // leave headroom for other operations
)
```

Adaptive backoff helps manage failing targets without completely abandoning them. When a target fails multiple consecutive scrapes, the scheduler gradually increases the time between scrape attempts while still respecting the configured minimum interval. This reduces resource waste on broken targets while ensuring they're automatically rediscovered when they recover.

The backoff algorithm follows this progression:

1. **First failure:** Continue normal interval (temporary network glitch)
2. **Second consecutive failure:** Add 10% jitter to interval (reduce thundering herd)
3. **Third consecutive failure:** Double the interval up to maximum backoff limit
4. **Continued failures:** Maintain maximum interval with exponential decay attempts
5. **First success:** Immediately return to normal configured interval

⚠ Pitfall: Scrape Time Drift A common mistake is calculating the next scrape time as `time.Now() + interval` instead of `scrape_start_time + interval`. This causes gradual drift where scrapes happen later and later over time as request processing duration accumulates. Always base the next scrape time on when the current scrape started, not when it completed. This maintains consistent sampling intervals essential for accurate rate calculations.

Metrics Parsing

The metrics parsing subsystem transforms HTTP response bodies in Prometheus exposition format into structured `Sample` objects that can be efficiently stored and queried. This involves lexical analysis of text-based metric data, validation of metric names and label formats, type inference, and timestamp assignment. The parser must handle malformed data gracefully while preserving as much valid information as possible from each scrape response.

Think of metrics parsing like **translating documents** from one language (text exposition format) to another (structured time series data). A good translator preserves the original meaning while adapting to the target language's grammar rules. Similarly, our parser preserves metric semantics while converting to our internal data structures. When encountering unclear passages (malformed metrics), a translator makes the best interpretation possible and continues rather than abandoning the entire document.

Prometheus exposition format uses a simple text-based protocol that balances human readability with parsing efficiency. Each metric family begins with optional `# HELP` and `# TYPE` comments that provide metadata, followed by one or more sample lines containing the metric name, optional labels, value, and optional timestamp. The parser must handle this streaming format incrementally since response bodies can contain thousands of metrics from complex applications.

A typical exposition format response looks like this structure:

```
# HELP http_requests_total Total number of HTTP requests
# TYPE http_requests_total counter
http_requests_total{method="GET",status="200"} 1234 1609459200000
http_requests_total{method="POST",status="201"} 56 1609459200000

# HELP process_cpu_seconds_total Total user and system CPU seconds
# TYPE process_cpu_seconds_total counter
process_cpu_seconds_total 123.45
```

The **lexical analyzer** processes the input stream character by character, identifying tokens like metric names, label keys, label values, numeric values, and timestamps. This component must handle several parsing challenges: quoted label values may contain escaped characters, numeric values can be integers, floats, or special values like `+Inf` and `Nan`, and timestamp values are optional Unix milliseconds that default to scrape time when absent.

Token Type	Pattern	Examples	Special Handling
Metric Name	[a-zA-Z_:][a-zA-Z0-9_]*	http_requests_total, cpu:usage_rate	Must start with letter, underscore, or colon
Label Key	[a-zA-Z_][a-zA-Z0-9_]*	method, status_code	Cannot start with __ (reserved prefix)
Label Value	"..." with escape sequences	"GET", "HTTP/1.1"	Handle \"", \\, \\n escape sequences
Numeric Value	Float or special constants	123.45, +Inf, NaN	IEEE 754 compliance for special values
Timestamp	Integer milliseconds	1609459200000	Optional, defaults to scrape time

Metric family grouping collects related metrics that share the same base name but have different label combinations or suffixes. For histogram metrics, the parser must recognize and group together the `_bucket`, `_sum`, and `_count` series that represent different aspects of the same histogram. This grouping enables proper validation—for example, ensuring histogram bucket boundaries are monotonically increasing and that all required series are present.

The parser implements **streaming validation** that checks metric names, label formats, and values as they're encountered rather than buffering the entire response first. This approach provides better memory efficiency for large responses and enables early error detection. When validation failures occur, the parser logs the error with sufficient context for debugging but continues processing the remaining metrics in the response.

Validation Rule	Check	Error Handling
Metric name format	Matches [a-zA-Z_:][a-zA-Z0-9_]*	Skip metric, log error with line number
Label key format	No __ prefix, valid identifier	Skip sample, preserve other labels
Label value encoding	Valid UTF-8, proper escaping	Replace invalid chars, log warning
Numeric value parsing	Valid float or special constant	Skip sample, increment parse error counter
Histogram consistency	Monotonic buckets, sum/count present	Accept partial data, log inconsistency

Type inference determines whether each metric represents a counter, gauge, histogram, or summary based on `# TYPE` comments and naming conventions. When type comments are missing, the parser uses heuristics like metric name suffixes (`_total` suggests counter, `_bucket` suggests histogram) and value patterns (monotonically increasing suggests counter). Accurate type inference is crucial because it affects how the storage engine handles the data and how query functions like `rate()` operate.

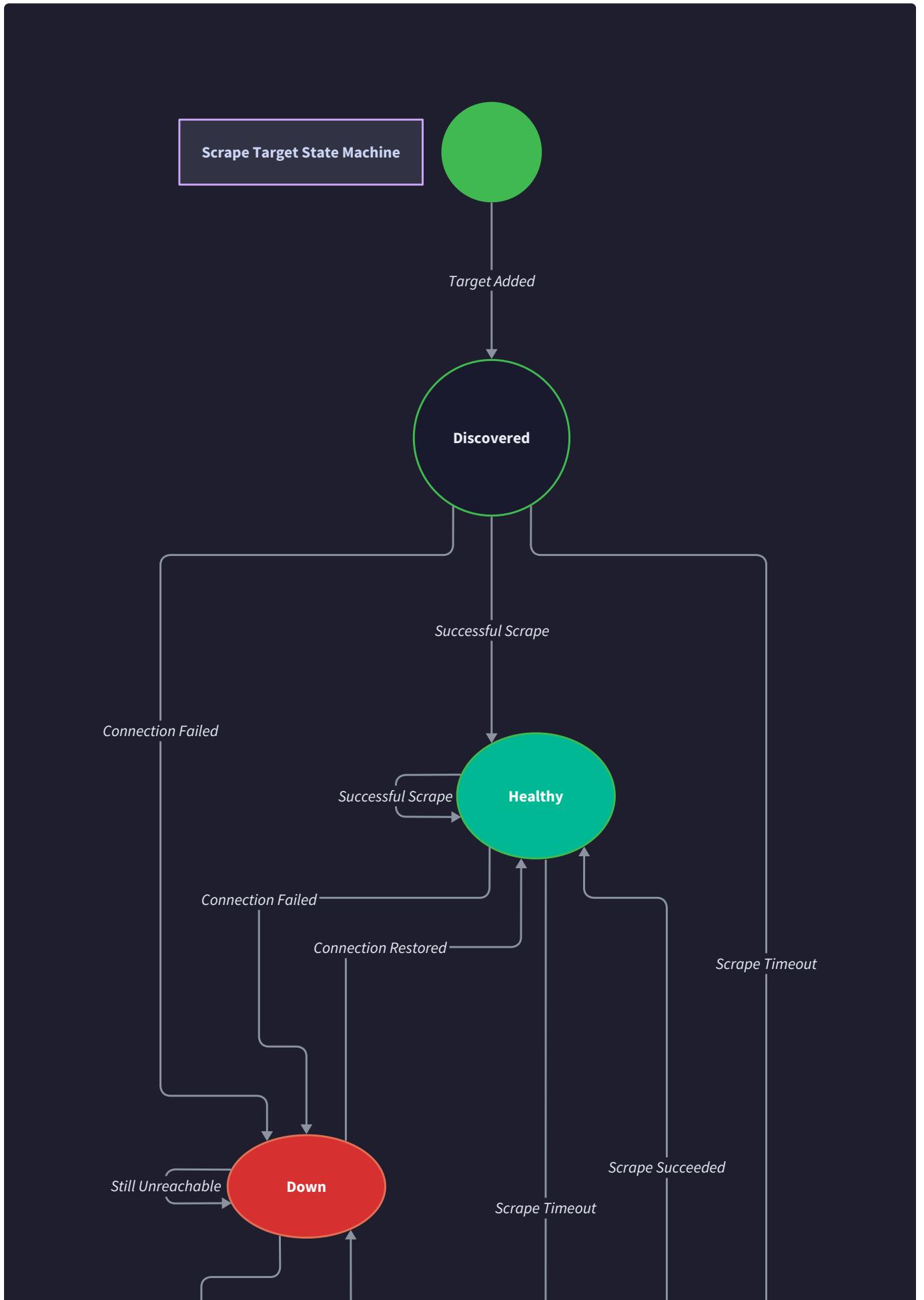
The **sample construction process** converts parsed tokens into `Sample` objects with proper timestamps and labels:

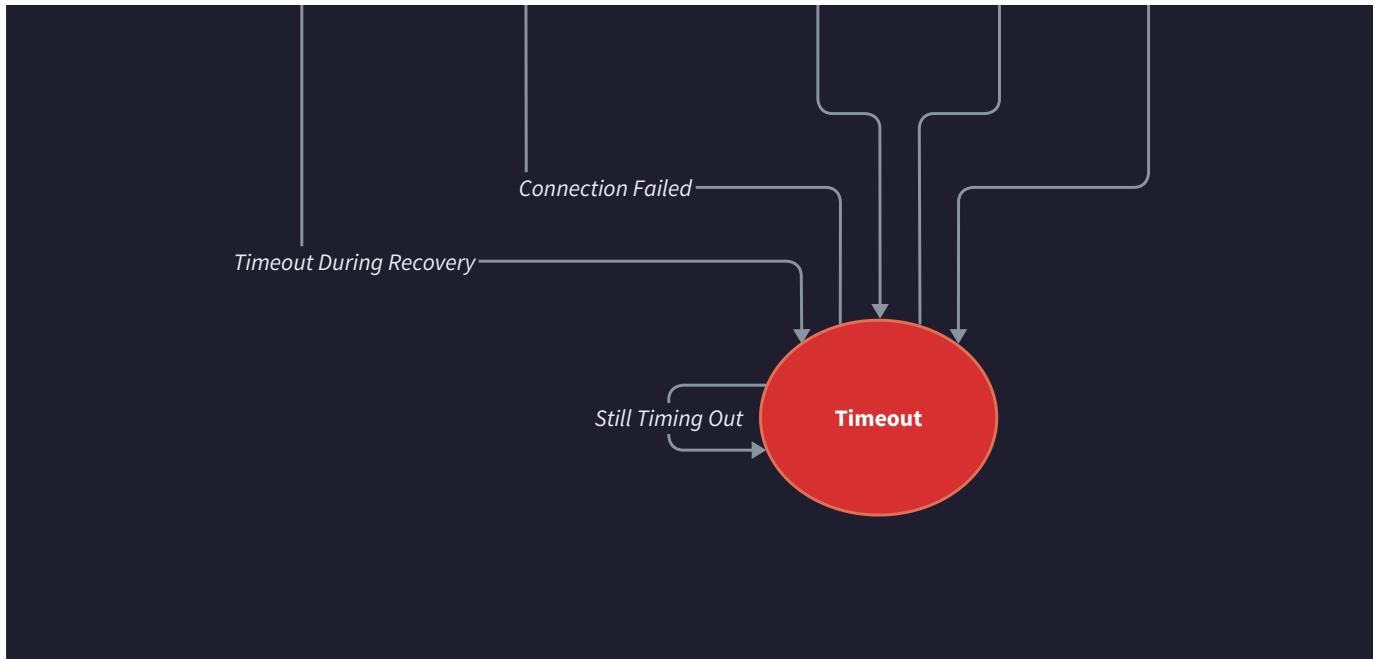
1. **Parse metric name and labels** from each sample line using the lexical analyzer
2. **Apply target labels** from service discovery configuration (instance, job, environment)

3. **Validate label cardinality** against configured limits to prevent memory explosion
4. **Assign timestamp** using provided value or current scrape time with millisecond precision
5. **Convert numeric value** to internal float64 representation, handling special constants
6. **Create Sample object** with metric identifier (name + labels), timestamp, and value
7. **Forward to storage engine** through the ingestion pipeline

Error recovery strategies ensure that parsing errors don't cause complete scrape failure. The parser maintains error counters for different failure modes and implements **best-effort processing** that extracts valid metrics even from responses with some malformed data. This resilience is essential in production environments where applications may generate imperfect exposition format due to bugs or configuration issues.

⚠ Pitfall: Label Cardinality Explosion Applications sometimes generate labels with unbounded values like user IDs, request IDs, or timestamps. A single misbehaving service can create millions of unique time series, consuming all available memory. The parser must implement **cardinality protection** that limits the number of unique label combinations per metric name and rejects samples that would exceed these limits. Always validate cardinality before creating new time series, not after.





Health and Error Handling

The health and error handling subsystem monitors scrape operations, classifies failures, implements recovery strategies, and provides observability into the scraping process. This component ensures that transient network issues don't cause permanent data loss, provides operators with visibility into collection problems, and maintains system stability under adverse conditions.

Think of target health management like **managing a fleet of field reporters** who gather information from remote locations. Some reporters occasionally miss check-ins due to bad weather (network issues), others might send garbled reports (parsing errors), and some might go completely silent (service failures). A good news editor tracks which reporters are reliable, follows up on missed check-ins, and adjusts expectations based on each reporter's track record. Similarly, our health system tracks target reliability and adapts behavior accordingly.

Target health exists in multiple dimensions that require independent tracking and different response strategies.

Network health indicates whether the scrape engine can successfully connect to a target's endpoint. **Application health** shows whether the target service is running and responding to requests. **Data health** reflects whether the metrics data being returned is valid and parseable. A target might have good network and application health but poor data health due to bugs in its metrics exposition code.

The health tracking system maintains state for each target using this comprehensive model:

Health Dimension	States	Transition Triggers	Impact on Scheduling
Network	Reachable, Unreachable, Timeout	TCP connection success/failure	Unreachable targets get exponential backoff
Application	Healthy, Error, Degraded	HTTP status codes (200 vs 4xx/5xx)	Error responses increase scrape interval
Data	Valid, Partial, Invalid	Parsing success/failure rates	Invalid data triggers diagnostic logging
Overall	Up, Down, Warning	Combination of above dimensions	Used for alerting and dashboard display

Failure classification determines the appropriate response to different types of errors encountered during scraping. Not all failures are equal—a temporary DNS resolution failure should be handled differently than an HTTP 404 response, which should be handled differently than a timeout. The classification system groups errors into categories that each have specific retry strategies and escalation procedures.

Error Category	Examples	Retry Strategy	Escalation
Transient Network	DNS timeout, connection refused	Immediate retry with jitter	Exponential backoff after 3 failures
Service Error	HTTP 500, service unavailable	Brief delay then retry	Reduce scrape frequency after 5 failures
Configuration Error	HTTP 404, invalid endpoint	Log error, continue normal interval	Alert operator after 10 consecutive failures
Data Format Error	Malformed exposition format	Process partial data, log details	Report parsing statistics

Adaptive retry logic balances the need for reliable data collection against the risk of overwhelming failing services. When targets experience failures, the retry system implements **exponential backoff with jitter** to reduce load while maintaining the possibility of recovery detection. The jitter component prevents **thundering herd effects** where many scrape engines simultaneously retry the same failed targets.

The retry algorithm follows this progression for consecutive failures:

1. **First failure:** Record error, maintain normal scrape interval
2. **Second failure:** Add 10-30% random jitter to next scrape time
3. **Third failure:** Double the scrape interval (15s becomes 30s)
4. **Fourth failure:** Double again with maximum cap (30s becomes 60s, capped at 5min)
5. **Continued failures:** Maintain maximum interval with occasional probe attempts
6. **First success:** Immediately return to configured normal interval
7. **Partial success:** Reduce interval by half until back to normal

Circuit breaker patterns protect both the scrape engine and target services from cascading failures. When a target fails consistently, the circuit breaker opens and stops sending requests temporarily. This prevents resource waste on the scrape engine side and reduces load on the failing service, potentially allowing it to recover. The circuit breaker periodically sends probe requests to detect recovery.

Circuit State	Behavior	Transition Condition	Probe Frequency
Closed	Normal scraping at configured interval	Failure rate < 50% over 10 scrapes	N/A
Open	Block all scrape attempts	After 60 seconds or manual reset	Every 60 seconds
Half-Open	Allow single probe scrape	Successful probe or probe failure	Single attempt

Resource protection mechanisms prevent failing targets from consuming excessive scrape engine resources. The protection system implements **per-target resource limits** on memory usage, connection time, and response body size. When targets exceed these limits, the scrape engine terminates the request and marks the target as misbehaving.

Resource	Limit	Protection Mechanism	Action on Violation
Response Body Size	10MB default	Stream reader with size limit	Truncate response, process partial data
Connection Time	Scrape timeout (10s default)	Context cancellation	Close connection, mark as timeout
Memory per Target	50MB default	Sample buffer size limits	Drop oldest samples, log warning
Concurrent Connections	Global semaphore	Weighted semaphore acquisition	Queue scrape, apply backpressure

Health metrics collection provides observability into the scraping process itself through internal metrics that track success rates, error distributions, response times, and resource usage. These metrics are essential for operators to understand collection health and optimize scrape configurations. The health system exposes these metrics through the same HTTP endpoint used by external scrapers.

Key internal metrics include:

Metric Name	Type	Labels	Purpose
scrape_duration_seconds	Histogram	job, instance	Track response time distribution
scrape_samples_scraped	Gauge	job, instance	Number of samples per scrape
scrape_series_added	Counter	job, instance	New time series creation rate
scrape_health	Gauge	job, instance	Binary health indicator (1=up, 0=down)
scrape_timeout_seconds	Gauge	job, instance	Configured timeout for target

Error aggregation and reporting collects detailed error information across all targets and provides structured access for debugging and alerting. Rather than logging each individual failure, the system aggregates errors by type, target, and time window to provide meaningful insights without overwhelming operators with noise.

⚠ Pitfall: Health Check Feedback Loops When scrape targets expose their own health status as metrics, failures can create confusing feedback loops. If a service reports itself as unhealthy in its metrics but still responds to HTTP requests, should the scrape engine consider it healthy or unhealthy? Design clear separation between **collection health** (can we scrape it?) and **application health** (what does it report about itself?). Never use application-reported health metrics to control scraping behavior.

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Client	<code>net/http</code> with default client	Custom client with connection pooling
Service Discovery	Static file-based configuration	Kubernetes API client (<code>k8s.io/client-go</code>)
Concurrency Control	Basic goroutines with <code>sync.WaitGroup</code>	Worker pools with semaphores
Configuration	YAML files with <code>gopkg.in/yaml.v2</code>	Configuration hot-reload with file watching
Metrics Parsing	Text scanner with regular expressions	Custom lexer with finite state machine

Recommended File Structure

```
project-root/
├── cmd/scrape/
│   └── main.go
└── internal/scrape/
    ├── engine.go
    ├── target.go
    ├── scheduler.go
    ├── parser.go
    ├── discovery/
    │   ├── static.go
    │   ├── dns.go
    │   └── kubernetes.go
    └── health/
        ├── tracker.go
        └── circuit_breaker.go
    └── config/
        └── scrape_config.go
    └── storage/
        └── ingestion.go
            └── ← Interface to storage layer
```

← Entry point for scrape engine
← Main scrape coordinator
← Target management and state
← Per-target scrape scheduling
← Prometheus format parser
← Static file-based discovery
← DNS SRV record discovery
← Kubernetes API discovery
← Target health monitoring
← Circuit breaker implementation
← Scrape configuration structures

Infrastructure Starter Code

HTTP Client with Timeouts (internal/scrape/client.go):

GO

```
package scrape

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

// HTTPClient wraps net/http.Client with scraping-specific configuration

type HTTPClient struct {

    client *http.Client

    userAgent string

    maxResponseSize int64
}

// NewHTTPClient creates a configured HTTP client for scraping

func NewHTTPClient(timeout time.Duration) *HTTPClient {
    return &HTTPClient{
        client: &http.Client{
            Timeout: timeout,
            Transport: &http.Transport{
                MaxIdleConns:           100,
                MaxIdleConnsPerHost:   10,
                IdleConnTimeout:       30 * time.Second,
            },
        },
        userAgent:      "prometheus-scraper/1.0",
        maxResponseSize: 10 * 1024 * 1024, // 10MB limit
    }
}
```

```
}

}

// ScrapeTarget performs HTTP GET request with size limits and timeout

func (c *HTTPClient) ScrapeTarget(ctx context.Context, url string) (io.Reader, error) {

    req, err := http.NewRequestWithContext(ctx, "GET", url, nil)

    if err != nil {

        return nil, fmt.Errorf("creating request: %w", err)
    }

    req.Header.Set("User-Agent", c.userAgent)

    req.Header.Set("Accept", "text/plain;version=0.0.4")

    resp, err := c.client.Do(req)

    if err != nil {

        return nil, fmt.Errorf("HTTP request failed: %w", err)
    }

    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {

        return nil, fmt.Errorf("HTTP %d: %s", resp.StatusCode, resp.Status)
    }

    // Limit response body size to prevent memory exhaustion

    limitedReader := &io.LimitedReader{

        R: resp.Body,
        N: c.maxResponseSize,
    }
}
```

```
// Read entire response into memory for parsing

body, err := io.ReadAll(limitedReader)

if err != nil {

    return nil, fmt.Errorf("reading response body: %w", err)

}

return bytes.NewReader(body), nil
}
```

Target Health Tracker (internal/scrape/health/tracker.go):

```
package health
```

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

```
// HealthStatus represents the current health state of a scrape target
```

```
type HealthStatus int
```

```
const (
    HealthUnknown HealthStatus = iota
    HealthUp
    HealthDown
    HealthDegraded
)
```

```
// TargetHealth tracks health metrics for a single scrape target
```

```
type TargetHealth struct {
```

```
    mutex          sync.RWMutex
    status         HealthStatus
    lastScrapeTime time.Time
    lastScrapeError error
    consecutiveFailures int
    totalScrapes    int64
    successfulScrapes int64
    lastSuccessTime time.Time
}
```

```
// NewTargetHealth creates a new health tracker for a target
```

```
func NewTargetHealth() *TargetHealth {
```

GO

```
return &TargetHealth{  
    status: HealthUnknown,  
}  
}  
  
// RecordSuccess updates health status for successful scrape  
  
func (h *TargetHealth) RecordSuccess(scrapeDuration time.Duration) {  
    h.mutex.Lock()  
  
    defer h.mutex.Unlock()  
  
    h.status = HealthUp  
  
    h.lastScrapeTime = time.Now()  
  
    h.lastScrapeError = nil  
  
    h.consecutiveFailures = 0  
  
    h.totalScrapes++  
  
    h.successfulScrapes++  
  
    h.lastSuccessTime = time.Now()  
}  
  
// RecordFailure updates health status for failed scrape  
  
func (h *TargetHealth) RecordFailure(err error) {  
    h.mutex.Lock()  
  
    defer h.mutex.Unlock()  
  
    h.lastScrapeTime = time.Now()  
  
    h.lastScrapeError = err  
  
    h.consecutiveFailures++  
  
    h.totalScrapes++
```

```

if h.consecutiveFailures >= 3 {

    h.status = HealthDown

} else if h.consecutiveFailures >= 1 {

    h.status = HealthDegraded

}

}

// GetStatus returns current health status thread-safely

func (h *TargetHealth) GetStatus() (HealthStatus, error, time.Time) {

    h.mutex.RLock()

    defer h.mutex.RUnlock()

    return h.status, h.lastScrapeError, h.lastScrapeTime
}

// SuccessRate calculates the percentage of successful scrapes

func (h *TargetHealth) SuccessRate() float64 {

    h.mutex.RLock()

    defer h.mutex.RUnlock()

    if h.totalScrapes == 0 {

        return 0.0
    }

    return float64(h.successfulScrapes) / float64(h.totalScrapes)
}

```

Core Logic Skeletons

Main Scrape Engine ([internal/scrape/engine.go](#)):

GO

```
// ScrapeEngine coordinates target discovery, scheduling, and metrics collection

type ScrapeEngine struct {

    config          *Config
    targets         map[string]*Target
    targetsMutex   sync.RWMutex
    discoverers    []TargetDiscoverer
    storage        StorageEngine
    httpClient     *HTTPClient
    logger         Logger
    stopChan       chan struct{}
    wg             sync.WaitGroup
}

// NewScrapeEngine creates and configures a new scrape engine

func NewScrapeEngine(config *Config, storage StorageEngine, logger Logger) *ScrapeEngine {
    // TODO 1: Initialize ScrapeEngine struct with provided parameters
    // TODO 2: Create HTTP client with configured timeout from config.ScrapeTimeout
    // TODO 3: Initialize empty targets map and stop channel
    // TODO 4: Create discoverers list based on config (static, DNS, K8s)
    // Hint: Use sync.RWMutex for targets map since it's read frequently but written rarely
}

// Start begins target discovery and scraping operations

func (e *ScrapeEngine) Start(ctx context.Context) error {
    // TODO 1: Start all configured target discoverers in separate goroutines
    // TODO 2: Launch target discovery consolidation loop to merge discovered targets
    // TODO 3: Start metrics collection goroutines for each active target
    // TODO 4: Set up signal handling for graceful shutdown
    // Hint: Use errgroup.Group to manage multiple goroutines with error propagation
}
```

```
}

// UpdateTargets processes target changes from service discovery

func (e *ScrapeEngine) UpdateTargets(newTargets []*Target) error {

    // TODO 1: Acquire write lock on targets map

    // TODO 2: Compare new target list with existing targets to find additions/removals

    // TODO 3: Start scrape goroutines for newly discovered targets

    // TODO 4: Stop scrape goroutines for removed targets by cancelling their contexts

    // TODO 5: Update internal targets map with new target set

    // Hint: Use target.URL as unique identifier for comparison

}
```

Target Scraper (internal/scrape/target.go):

GO

```
// scrapeTarget performs a single scrape operation against a target

func (e *ScrapeEngine) scrapeTarget(ctx context.Context, target *Target) error {

    // TODO 1: Create HTTP request context with scrape timeout deadline

    // TODO 2: Record scrape start time for consistent interval calculation

    // TODO 3: Perform HTTP GET request to target.URL + target.MetricsPath

    // TODO 4: Check HTTP response status and handle non-200 responses

    // TODO 5: Parse response body using Prometheus exposition format parser

    // TODO 6: Add target labels (job, instance) to all parsed samples

    // TODO 7: Forward samples to storage engine via Append() method

    // TODO 8: Update target health status based on success/failure

    // TODO 9: Record scrape metrics (duration, sample count, error status)

    // Hint: Always update health status even if storage append fails

}

// runTargetScrapeLoop manages the continuous scraping lifecycle for one target

func (e *ScrapeEngine) runTargetScrapeLoop(ctx context.Context, target *Target) {

    // TODO 1: Create ticker with target.ScrapeInterval duration

    // TODO 2: Initialize target health tracker

    // TODO 3: Enter infinite loop listening for ticker and context cancellation

    // TODO 4: On each tick, call scrapeTarget() and handle any errors

    // TODO 5: Calculate next scrape time based on start time, not completion time

    // TODO 6: Implement exponential backoff for consecutive failures

    // TODO 7: Clean up ticker and update target registry on context cancellation

    // Hint: Use time.NewTicker and defer ticker.Stop() for proper cleanup

}
```

Prometheus Format Parser (internal/scrape/parser.go):

```

// ParsePrometheusFormat converts exposition format text to structured samples      GO

func ParsePrometheusFormat(reader io.Reader, defaultTimestamp time.Time) ([]*Sample, error) {

    // TODO 1: Create scanner to read input line by line

    // TODO 2: Initialize empty samples slice and current metric metadata

    // TODO 3: Process each line: comments (HELP/TYPE) vs sample lines

    // TODO 4: For comment lines, extract metric name and type information

    // TODO 5: For sample lines, parse metric name, labels, value, optional timestamp

    // TODO 6: Validate metric names match [a-zA-Z_:][a-zA-Z0-9_]* pattern

    // TODO 7: Parse label sets with proper quote handling and escape sequences

    // TODO 8: Convert string values to float64, handle +Inf/-Inf/Nan special cases

    // TODO 9: Use provided timestamp if sample doesn't include one

    // TODO 10: Return accumulated samples slice or first parsing error encountered

    // Hint: Use regexp for metric name validation, manual parsing for labels is more efficient

}

// parseSampleLine extracts components from a single metric sample line

func parseSampleLine(line string) (metricName string, labels Labels, value float64, timestamp *time.Time, err error) {

    // TODO 1: Find metric name at start of line (ends at '{' or whitespace)

    // TODO 2: If '{' present, parse label set until matching '}''

    // TODO 3: Parse numeric value after labels, handle special float constants

    // TODO 4: Check for optional timestamp at end of line (Unix milliseconds)

    // TODO 5: Return parsed components or detailed error with line context

    // Hint: Be careful with quoted label values containing '{', '}', or whitespace

}

```

Language-Specific Hints

- **Goroutine Management:** Use `sync.WaitGroup` to track scrape goroutines and `context.Context` for graceful cancellation. Each target gets its own goroutine for isolation.
- **HTTP Timeouts:** Set timeouts at multiple levels: `http.Client.Timeout` for overall request timeout, `context.WithTimeout` for individual scrape operations, and `io.LimitedReader` for response size limits.

- **Memory Management:** Pre-allocate slices for samples when possible (`make([])*Sample, 0, estimatedCount`). Use object pooling for frequently allocated parser structures.
- **Error Handling:** Distinguish between permanent errors (HTTP 404) and transient errors (network timeout). Use `errors.Is()` and `errors.As()` for proper error classification.
- **Configuration Reloading:** Watch configuration files with `fsnotify` package and reload target lists without stopping active scrapes.

Milestone Checkpoints

After implementing target discovery:

```
go test ./internal/scrape/discovery/...
go run cmd/scrapers/main.go -config=test-config.yaml
```

BASH

Expected behavior: Scraper should log discovered targets from static configuration and start attempting to scrape them (even if they fail). Check logs for "discovered target" messages.

After implementing scrape scheduling:

Configure a target pointing to `http://localhost:8080/metrics` and run a simple HTTP server that returns basic Prometheus format. Verify scrapes happen at the configured interval by checking timestamps in logs.

After implementing metrics parsing: Test with malformed exposition format to verify parser handles errors gracefully and extracts valid metrics while skipping invalid ones.

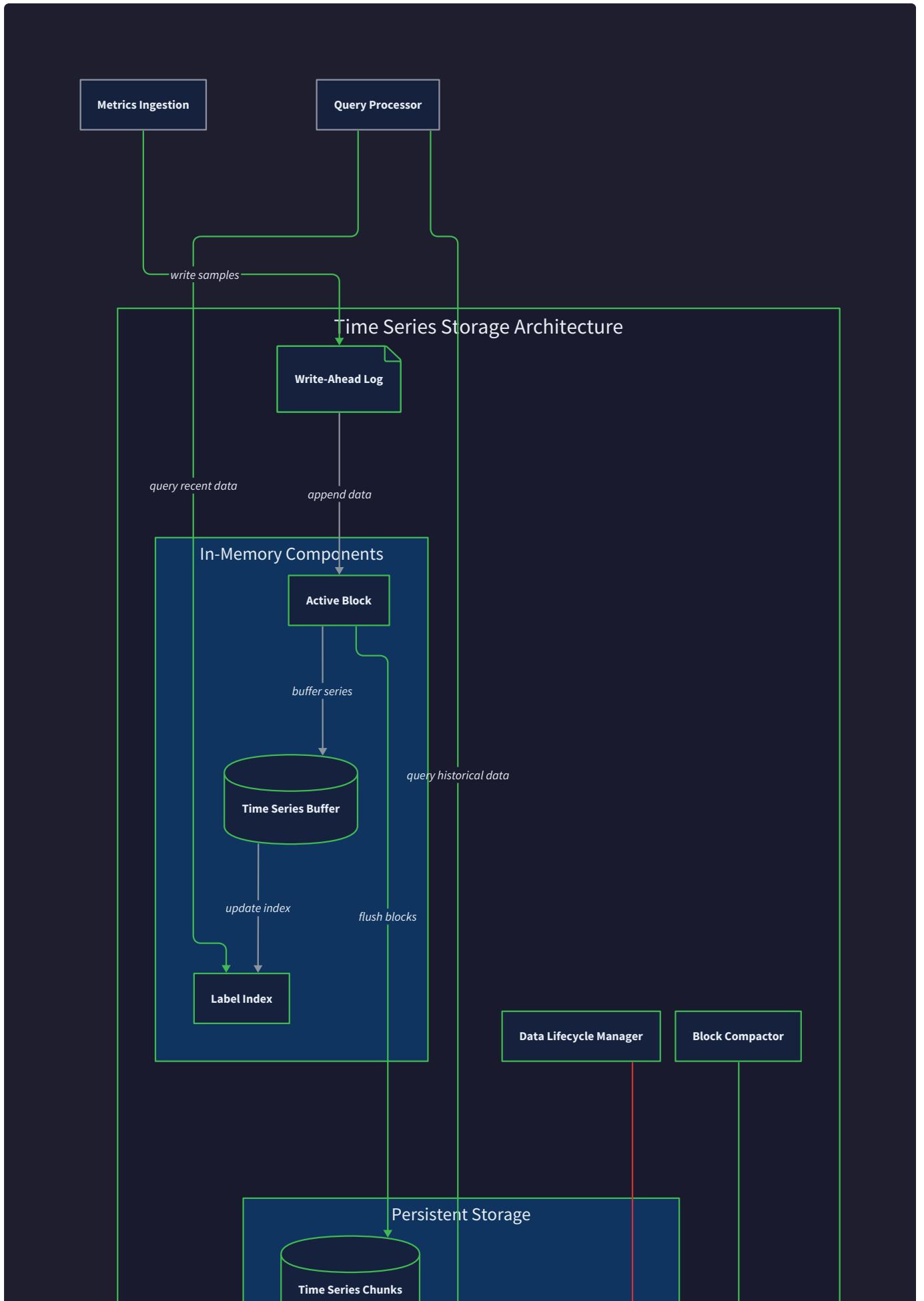
Signs something is wrong:

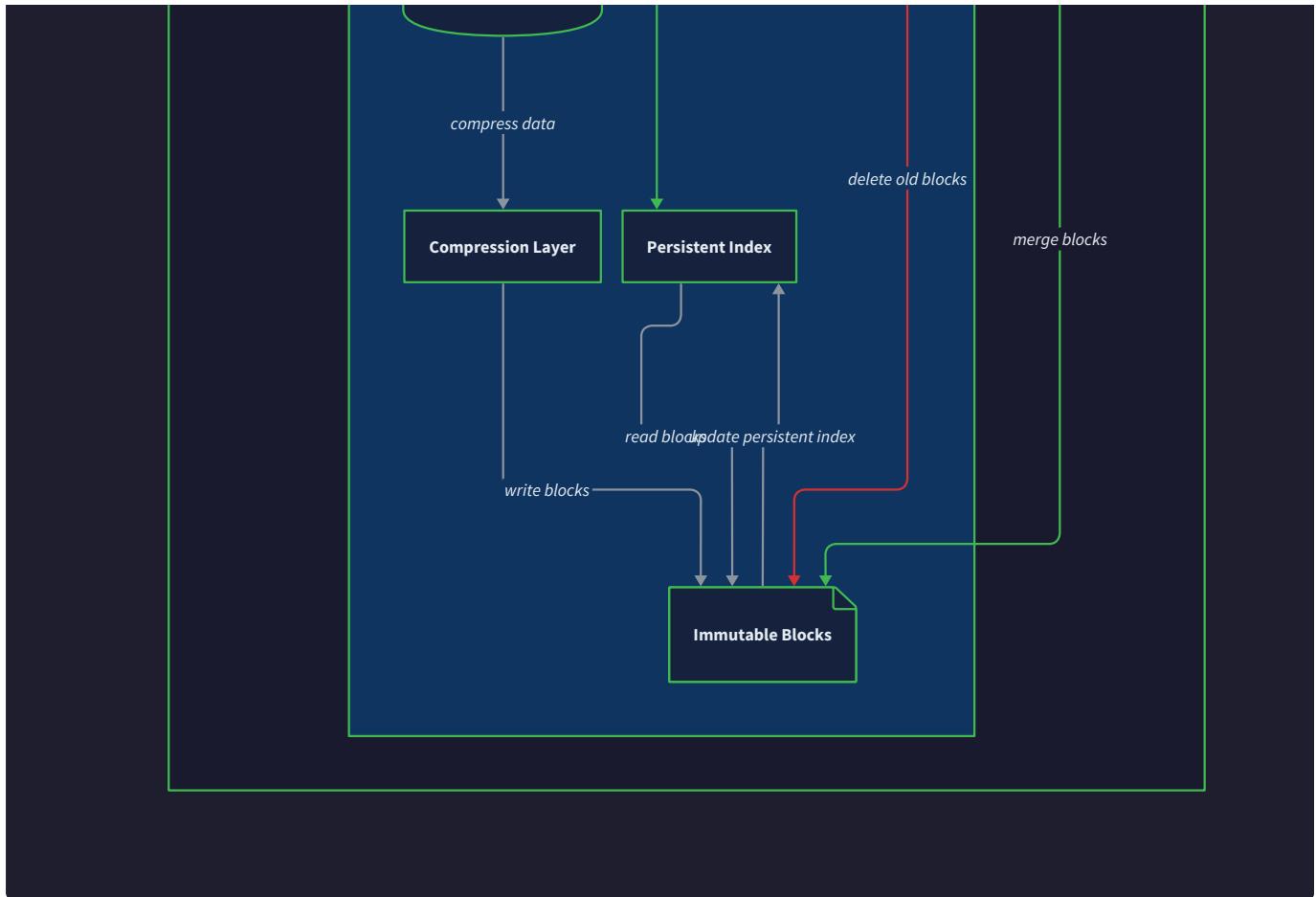
- Scrape times drift further apart → Check interval calculation logic
- Memory usage grows constantly → Check for target goroutine leaks
- Targets always show as "down" → Verify HTTP client timeout configuration
- Parser crashes on certain inputs → Add more input validation and error recovery

Time Series Storage Engine

Milestone(s): This section directly corresponds to Milestone 3 (Time Series Storage) and provides the efficient storage foundation that will be queried by Milestone 4 (Query Engine). The storage engine receives scraped metrics from Milestone 2 (Scrape Engine) and persists the time series data defined in Milestone 1 (Metrics Data Model).

The time series storage engine serves as the memory and archive of our metrics collection system. It must handle the unique challenges of time series data: high write throughput from continuous scraping, efficient storage of timestamped numeric values, fast retrieval for queries, and automatic lifecycle management as data ages. Unlike traditional databases optimized for random access patterns, time series workloads exhibit distinct characteristics that demand specialized storage techniques.





The storage engine operates under several constraints that shape its design. Time series data arrives continuously with timestamps that are always increasing within each series, creating an append-only write pattern. Query patterns typically request recent data or ranges of historical data, rarely accessing individual points randomly. The volume of incoming data grows linearly with the number of monitored targets and their label cardinality, making storage efficiency critical for operational costs. Finally, old data has diminishing value and must be automatically removed to prevent unbounded storage growth.

Library Archive Mental Model

Understanding time series storage becomes intuitive when we think of it like a physical library archive system. Imagine a vast library that specializes in storing weather measurement records from thousands of monitoring stations worldwide. Each monitoring station represents a unique time series identified by its location and the type of measurement (temperature, humidity, pressure). The library receives new measurements every minute and must organize them efficiently for both storage and retrieval.

In our library analogy, each **monitoring station** corresponds to a time series identified by its metric name and label set. The station "temperature.celsius{location=paris,sensor=outdoor}" produces a continuous stream of timestamped temperature readings, just as our time series produces timestamped numeric samples. The library must create a unique storage location for each station's records, much like our storage engine creates separate storage chunks for each time series.

The library organizes records using a two-level system: a **card catalog** (our inverted indexes) and **storage boxes** (our compressed chunks). The card catalog contains index cards that list which storage boxes contain records for each station and time period. When a researcher asks for "all temperature readings from Paris outdoor sensors

between March 1-15," the librarian first consults the card catalog to identify relevant boxes, then retrieves only those specific boxes rather than searching the entire archive.

The storage boxes themselves use a clever compression technique. Instead of writing the full timestamp and value for each record, the librarian writes only the difference from the previous record. If yesterday's temperature was 20.5°C and today's is 21.2°C, the record stores "+0.7" instead of the full value. This **delta compression** dramatically reduces the space needed for each box, allowing the library to store decades of history in a reasonable amount of space.

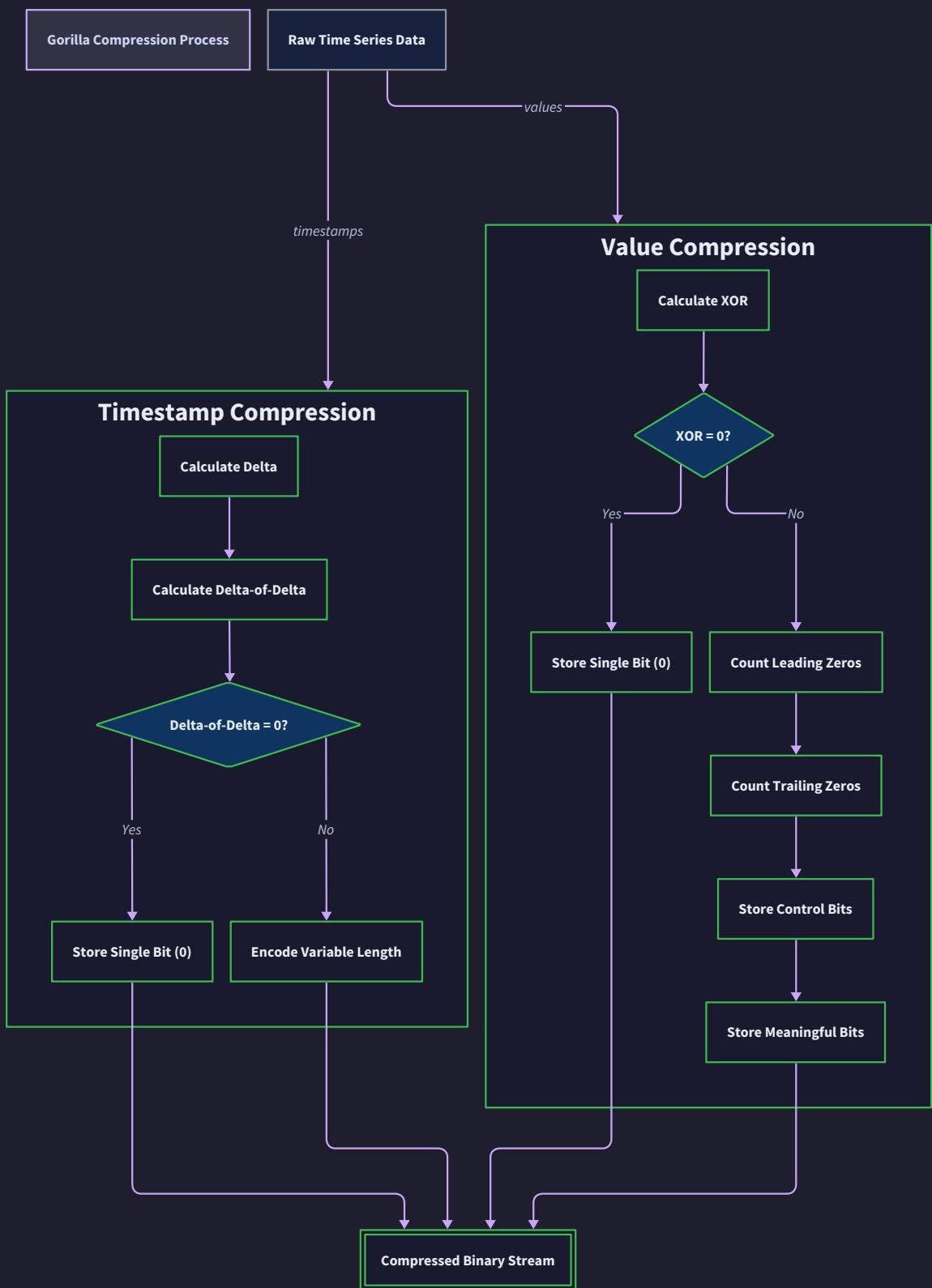
As records age, the library applies a **retention policy** similar to how old newspapers are eventually discarded or moved to deep storage. Records newer than one month are kept in full detail. Records older than one month but newer than one year are **downsampled** - instead of minute-by-minute readings, only hourly averages are preserved. Records older than one year are deleted entirely. This automatic lifecycle management prevents the archive from growing without bound while preserving the most valuable historical data.

When the library receives new records, they're first written to a **processing journal** (our write-ahead log) before being filed in the appropriate storage boxes. If a power outage occurs while the librarian is updating multiple boxes, the journal can be replayed upon restart to ensure no records are lost. This provides **durability** even in the face of unexpected failures.

The genius of this system is that it optimizes for the actual usage patterns of time series data: most writes are recent data appended to existing series, most reads request ranges of data from specific series, and old data becomes less valuable over time. Our storage engine implements this same organizational philosophy using modern computer science techniques.

Gorilla Compression

The Gorilla compression algorithm, developed by Facebook for their time series database, provides remarkable space efficiency by exploiting the predictable patterns inherent in time series data. Most time series data exhibits two key properties: timestamps arrive at regular intervals, and values change gradually between adjacent samples. Gorilla compression leverages both patterns to achieve compression ratios of 10:1 or better.



Decision: Gorilla Compression Algorithm

- **Context:** Time series data consumes enormous storage space when naively stored as timestamp-value pairs. A monitoring system collecting metrics every 15 seconds from 1000 targets generates over 5 million samples per day. Without compression, each sample requires 16 bytes (8-byte timestamp + 8-byte float64), consuming 80MB daily per target.
- **Options Considered:**
 1. Store raw timestamp-value pairs without compression
 2. Generic compression (gzip/lz4) applied to time series chunks
 3. Gorilla-style delta-of-delta timestamp and XOR value compression
- **Decision:** Implement Gorilla compression with delta-of-delta timestamps and XOR value encoding
- **Rationale:** Gorilla compression is specifically designed for time series characteristics. Delta-of-delta encoding exploits regular timestamp intervals common in metrics collection. XOR value encoding exploits the fact that consecutive floating-point measurements often share common bit patterns. Generic compression doesn't understand time series structure and achieves lower compression ratios.
- **Consequences:** Achieves 10:1 compression ratios typical of Gorilla. Requires more complex encoding/decoding logic than raw storage. CPU overhead for compression/decompression during writes and reads.

Compression Option	Space Efficiency	CPU Overhead	Implementation Complexity	Chosen?
Raw Storage	16 bytes/sample	None	Minimal	No
Generic Compression	4-8 bytes/sample	Medium	Low	No
Gorilla Compression	1.4 bytes/sample	High	High	Yes

Delta-of-Delta Timestamp Encoding

Timestamp compression exploits the regularity of metrics collection intervals. Consider a time series scraped every 15 seconds starting at timestamp 1640995200:

```
Timestamp: 1640995200, 1640995215, 1640995230, 1640995245, ...
Deltas:      [baseline],      +15,      +15,      +15, ...
DoD:        [baseline],      [baseline],      0,          0, ...
```

The algorithm stores the first timestamp as a 64-bit baseline. For the second timestamp, it calculates the delta (difference) from the first and stores this delta as a baseline delta. For subsequent timestamps, it calculates the "delta-of-delta" - the difference between the current delta and the expected delta based on the pattern.

When the delta-of-delta is zero (indicating a perfectly regular interval), Gorilla stores just a single bit flag. When the delta-of-delta is small (within ± 63 of the expected delta), it stores the value in 7 bits. Only when timestamps deviate significantly from the pattern does it fall back to larger encodings. This approach reduces regular 64-bit timestamps to often just 1 bit per sample.

Delta-of-Delta Range	Encoding	Bits Used	Common Case
0	Single '0' bit	1	Regular intervals
-63 to +64	'10' + 7-bit signed value	9	Small time drift
-255 to +256	'110' + 9-bit signed value	12	Clock adjustments
-2047 to +2048	'1110' + 12-bit signed value	16	Irregular intervals
Other	'1111' + 32-bit signed value	36	Major time jumps

XOR Value Encoding

Value compression leverages the observation that consecutive floating-point measurements often share common bit patterns. Temperature readings might vary from 20.1°C to 20.3°C, and the IEEE 754 binary representations of these values differ only in the least significant bits.

The algorithm XORs each value with the previous value in the series. If the XOR result is zero (values are identical), it stores a single '0' bit. If the XOR result is non-zero, it analyzes the bit pattern to determine the most efficient encoding.

```
Value 1: 20.1 (binary: 0x4034199999999998)
Value 2: 20.3 (binary: 0x403451999999999A)
XOR:          0x0000408000000002
```

The XOR result often has many leading and trailing zero bits. Gorilla stores only the significant bits between the leading and trailing zeros, along with metadata indicating their position. When consecutive values are very similar, this reduces 64-bit floating-point values to as few as 5-10 bits.

XOR Pattern	Encoding Strategy	Typical Bits	Example Scenario
Zero XOR	Single '0' bit	1	Identical consecutive values
Same pattern as previous	'10' + compressed bits	2-15	Gradually changing values
New pattern	'11' + leading zeros + length + bits	10-65	Value jumps or first difference

Compression Implementation Strategy

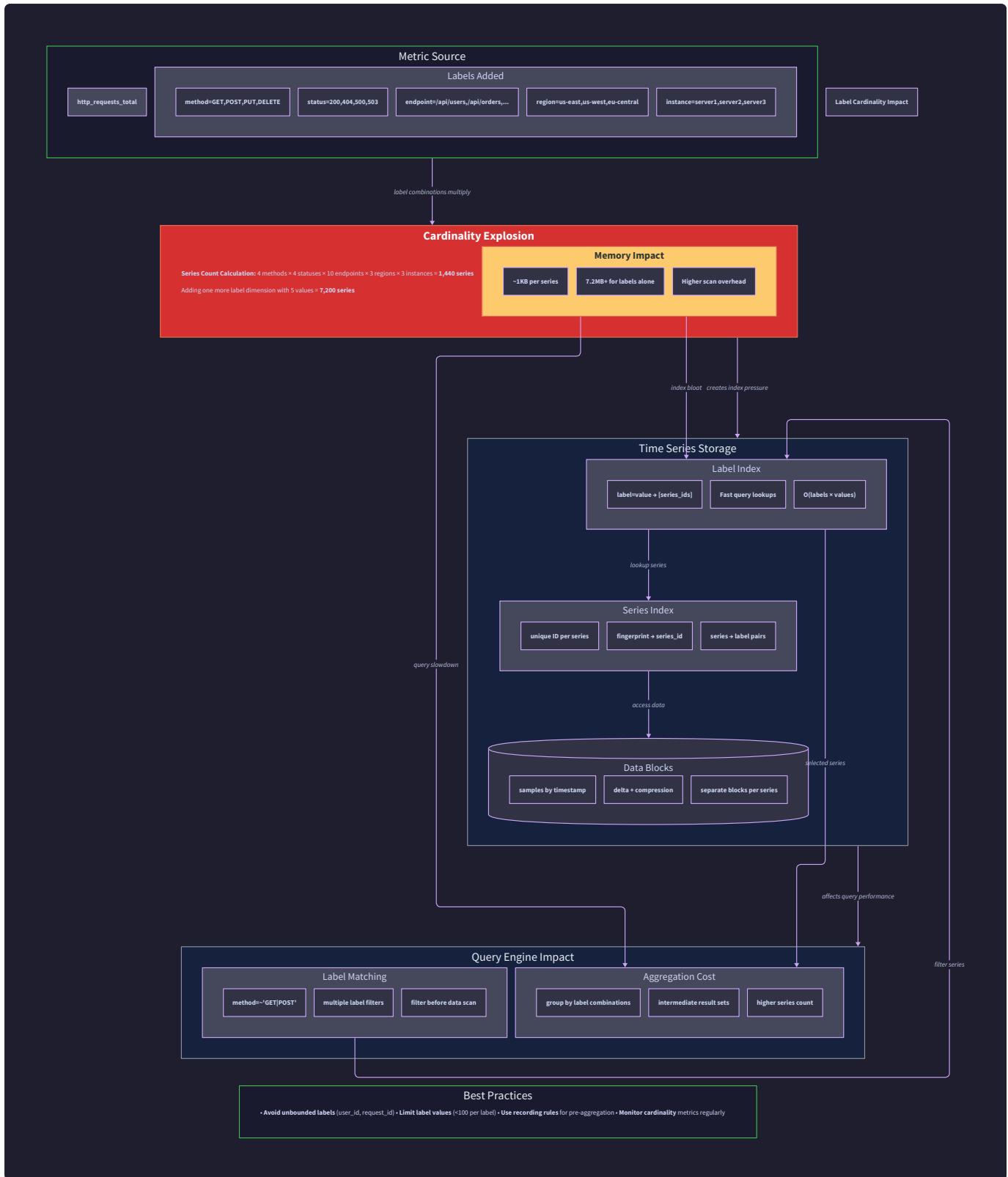
The compression algorithm operates on fixed-size chunks of time series data, typically containing 2-4 hours worth of samples. Each chunk begins with baseline values for the first timestamp and value, followed by compressed deltas and XOR patterns for subsequent samples. The chunk header contains metadata necessary for decompression: the baseline timestamp, baseline value, and the number of samples in the chunk.

During compression, the algorithm maintains state about the previous timestamp delta and value to calculate deltas-of-deltas and XOR patterns. This state allows the compressor to make optimal encoding decisions based on the emerging patterns in the data. The bit-level nature of the encoding means that sample boundaries don't align with byte boundaries, requiring careful bit manipulation during both compression and decompression.

Decompression reverses the process by starting with the baseline values and iteratively applying the stored deltas and XOR patterns to reconstruct the original timestamp-value pairs. The algorithm must handle variable-length encodings correctly, using the bit flags to determine how many subsequent bits to read for each sample.

Series Indexing

Efficient querying requires fast lookup of time series by metric name and label combinations. With millions of active time series, a naive linear search through all series would make queries prohibitively slow. The storage engine employs inverted indexes that map from metric names and label values to the specific time series containing that metadata, enabling subsecond query response times even with high cardinality.



The indexing challenge stems from the multi-dimensional nature of labeled time series. A query like `http_requests_total{method="GET", status="200"}` must find all time series where the metric name matches exactly and both specified labels have the required values. The query engine may then need to aggregate across other label dimensions like `instance` or `job` that weren't specified in the query.

Decision: Inverted Index Architecture

- **Context:** Queries must efficiently find time series matching metric names and label selectors from millions of active series. Label-based queries like `{job="web", method=~"GET|POST"}` require fast intersection of multiple label conditions.
- **Options Considered:**
 1. Single composite index mapping full series signatures to storage locations
 2. Separate indexes per metric name with secondary label indexes
 3. Inverted indexes mapping each label value to series containing that value
- **Decision:** Implement inverted indexes with efficient intersection algorithms
- **Rationale:** Inverted indexes support arbitrary label selector combinations efficiently. They enable fast intersection operations when multiple labels are specified. They scale better with high cardinality than composite indexes.
- **Consequences:** Requires more storage overhead for multiple indexes. Complex intersection logic for multi-label queries. Fast query performance for typical PromQL patterns.

Primary Metric Index

The primary index maps metric names to lists of series identifiers that contain metrics with that name. This provides the first level of filtering for most queries, since PromQL queries typically start with a metric name like

`http_requests_total` or `cpu_usage_seconds`. The index structure resembles a traditional database index optimized for prefix matching and range scans.

Index Component	Structure	Purpose
Metric Name Map	<code>map[string][]uint64</code>	Maps metric names to series ID lists
Series Registry	<code>map[uint64]*SeriesMetadata</code>	Maps series IDs to full label sets and storage locations
Label Indexes	<code>map[string]map[string][]uint64</code>	Maps label name → value → series IDs for fast label filtering

Each series receives a unique 64-bit identifier when first created. The series registry maintains the authoritative mapping from this identifier to the complete label set and storage chunk locations for that series. This indirection allows the inverted indexes to store compact series IDs rather than full label sets, reducing memory usage as cardinality grows.

Label Value Indexes

For each label name that appears in any time series, the storage engine maintains an inverted index mapping from label values to the series that contain those values. The label index for `method` might map `"GET"` to series IDs `[1001, 1003, 1007, ...]` and `"POST"` to series IDs `[1002, 1005, 1008, ...]`.

When processing a query like `http_requests_total{method="GET", status="200"}`, the query engine:

1. Looks up the metric name `http_requests_total` in the primary index to get candidate series IDs

2. Looks up label value "GET" in the `method` label index to get matching series IDs
3. Looks up label value "200" in the `status` label index to get matching series IDs
4. Computes the intersection of all three series ID lists to find series matching all conditions

Query Processing Step	Input	Index Used	Output
Metric Name Filter	<code>http_requests_total{...}</code>	Primary metric index	Series IDs [1001, 1002, 1003, ...]
Label Filter <code>method="GET"</code>	Series ID list	<code>method</code> label index	Filtered series IDs [1001, 1003, ...]
Label Filter <code>status="200"</code>	Series ID list	<code>status</code> label index	Final series IDs [1001, ...]
Series Metadata Lookup	Series ID list	Series registry	Label sets and chunk locations

Index Intersection Algorithms

Efficiently computing intersections of series ID lists is critical for query performance, especially when multiple label conditions are specified. The storage engine implements several intersection algorithms optimized for different scenarios commonly encountered in time series queries.

For small result sets (fewer than 1000 series), the engine uses a simple hash set intersection. It loads the smallest series ID list into a hash set, then iterates through other lists checking membership. This approach provides O(n) performance and minimal memory overhead for the common case of specific label value combinations.

For larger result sets, the engine switches to a sorted list intersection algorithm. Since series IDs are assigned monotonically, the inverted index lists are naturally sorted. The algorithm uses multiple pointers to advance through the sorted lists simultaneously, similar to merging sorted arrays. This approach avoids the memory allocation overhead of hash sets when dealing with large intermediate results.

The query optimizer chooses the intersection order based on the estimated cardinality of each label condition. Label values with lower cardinality (fewer matching series) are processed first to minimize the size of intermediate results. A label like `datacenter="us-west"` might match thousands of series, while `instance="web-01"` matches only one, making instance-first processing much more efficient.

Cardinality Management

High cardinality labels pose the greatest threat to index performance and memory usage. A label like `request_id` with unique values for every request would create millions of entries in the label value index, consuming enormous memory and slowing intersection operations. The storage engine implements several strategies to detect and mitigate cardinality explosion before it degrades system performance.

Cardinality Level	Series Count	Index Memory	Query Performance	Management Strategy
Low	< 1,000	< 10MB	Subsecond	Normal operation
Medium	1,000 - 100,000	10MB - 1GB	1-5 seconds	Monitor growth rate
High	100,000 - 1,000,000	1GB - 10GB	5-30 seconds	Throttle ingestion
Critical	> 1,000,000	> 10GB	30+ seconds	Reject new series

The `CardinalityTracker` component monitors the number of unique values for each label name and raises alerts when cardinality grows unexpectedly. It maintains moving averages of cardinality growth rates and can predict when a label will exceed safe thresholds based on current trends.

When cardinality limits are approached, the storage engine can reject new time series that would exceed the configured limits. This prevents a runaway cardinality explosion from bringing down the entire monitoring system. The rejected series are logged for later analysis, allowing operators to identify the source of high-cardinality labels and fix the instrumentation.

Data Lifecycle Management

Time series data has a natural lifecycle where recent data is most valuable for alerting and debugging, while historical data provides context for capacity planning and trend analysis. However, storing all historical data indefinitely is neither practical nor cost-effective. The storage engine implements automated lifecycle management that balances data retention needs with storage costs through configurable retention policies and downsampling strategies.

The lifecycle management system operates on the principle that data value decreases over time, but at different rates for different use cases. Alerting requires minute-level granularity for recent data but can tolerate hour-level granularity for data older than a week. Capacity planning queries typically aggregate data over longer time periods and don't require full-resolution historical data. By automatically reducing resolution as data ages, the system maintains query capability while controlling storage growth.

Decision: Hierarchical Data Lifecycle Management

- **Context:** Time series data volume grows linearly with time, making indefinite full-resolution retention impossible. Different use cases have different precision requirements based on data age.
- **Options Considered:**
 1. Fixed retention period with complete deletion after expiration
 2. Uniform downsampling (e.g., keep only hourly averages for all old data)
 3. Hierarchical retention with multiple resolution levels based on age
- **Decision:** Implement hierarchical retention with automatic downsampling at configurable age thresholds
- **Rationale:** Provides optimal balance between storage efficiency and query utility. Recent data kept at full resolution for debugging. Historical data downsampled to enable long-term trend analysis. Configurable policies allow customization based on organizational needs.
- **Consequences:** Requires complex logic to manage multiple resolution levels. Queries spanning multiple retention tiers need special handling. Provides excellent storage efficiency and query flexibility.

Retention Policy Configuration

The storage engine supports flexible retention policies that define both the total retention period and the downsampling schedule as data ages. These policies are configured per metric or per metric pattern, allowing different retention strategies for different types of monitoring data.

Retention Tier	Age Range	Resolution	Retention Period	Storage Ratio
High Resolution	0-7 days	Original (15s)	7 days	1.0x
Medium Resolution	7-30 days	5 minutes	23 days	0.05x
Low Resolution	30-365 days	1 hour	335 days	0.004x
Archive	1+ years	Daily	2 years	0.0002x

A typical retention policy for application metrics might preserve 15-second resolution data for the past week, downsample to 5-minute resolution for the past month, and keep hourly averages for historical trend analysis. Critical business metrics might have longer high-resolution periods, while debugging metrics might have shorter retention.

The retention configuration specifies not just the time boundaries but also the aggregation functions to use during downsampling. Counter metrics typically use rate calculations, gauge metrics use averages, and histogram metrics require careful handling to preserve distribution characteristics. The policy also specifies which labels to preserve during aggregation - high-cardinality labels like instance might be dropped while keeping service and datacenter labels.

Automated Downsampling Process

The downsampling process operates as a background task that periodically identifies data chunks eligible for resolution reduction. Rather than processing individual samples, the system works with compressed chunks to maintain efficiency. When a chunk ages past a resolution threshold, the downsampling process decompresses the chunk, aggregates samples into lower-resolution buckets, and creates new compressed chunks at the target resolution.

The aggregation process must handle different metric types appropriately to preserve semantic meaning. Counter metrics represent cumulative totals that should be converted to rates during downsampling. A counter chunk containing values [100, 115, 130, 145] over four 15-second intervals would become a single 1-minute rate of 0.75 increments per second. Gauge metrics represent point-in-time values and are typically averaged, though max, min, or last-value aggregations might be more appropriate depending on the metric semantics.

Histogram metrics require special handling to preserve distribution information during downsampling. The system can't simply average the bucket counts, as this would destroy the ability to calculate accurate percentiles. Instead, histogram downsampling re-bins the constituent observations into the histogram buckets at the target resolution. This preserves the distributional characteristics while reducing storage overhead.

The downsampling process maintains metadata linking the original high-resolution chunks to their downsampled derivatives. This enables the query engine to automatically select the appropriate resolution level based on the query time range and step size. A query requesting data over a 30-day period with 1-hour steps can use the low-resolution data directly rather than aggregating high-resolution samples.

Garbage Collection and Compaction

As data moves through the retention tiers and eventually expires, the storage engine must reclaim the associated storage space and update indexes accordingly. The garbage collection process operates independently of downsampling to avoid coupling data lifecycle decisions with storage management concerns.

Garbage Collection Phase	Scope	Actions	Frequency
Expired Data Deletion	Chunks older than max retention	Delete chunk files, remove index entries	Daily
Index Compaction	Label indexes with many deleted entries	Rebuild compact indexes, reclaim memory	Weekly
Chunk Compaction	Small chunks from same series	Merge chunks, improve compression ratio	Continuous
Orphan Cleanup	Index entries without corresponding chunks	Remove stale index entries	Daily

The garbage collection process begins by identifying chunks that have exceeded their configured retention period. Rather than immediately deleting these chunks, the system marks them for deletion and continues serving queries from the remaining data. A separate cleanup process deletes the marked chunks during off-peak hours to minimize I/O impact on active queries.

Index maintenance ensures that deleted time series don't leave stale entries in the inverted indexes. When all chunks for a particular series are deleted, the series registry entry is removed and all references to that series ID are purged from the label value indexes. This prevents memory leaks and maintains query performance as the active series set changes over time.

Chunk compaction addresses the storage inefficiency that occurs when many small chunks exist for the same time series. These small chunks typically result from irregular scraping or brief monitoring periods. The compaction process merges consecutive chunks from the same series into larger chunks, improving compression ratios and reducing the overhead of chunk metadata.

Persistence and Recovery

The storage engine must provide durability guarantees ensuring that accepted metric samples survive system crashes and hardware failures. Time series monitoring is often critical infrastructure used for alerting and incident response, making data loss unacceptable even in the face of unexpected failures. The persistence layer implements write-ahead logging and checkpoint-based recovery to provide these guarantees while maintaining write throughput under normal operation.

The durability challenge stems from the batch-oriented nature of time series ingestion. Samples arrive continuously from scraped targets and are buffered in memory before being compressed and written to persistent chunks. A naive approach might lose several minutes of samples if the system crashed before the in-memory buffers were flushed. The write-ahead log ensures that every accepted sample is persisted before acknowledging receipt, providing recovery capability even if compression and indexing haven't completed.

Decision: Write-Ahead Log with Periodic Checkpoints

- **Context:** Time series samples must be durably stored before acknowledgment to prevent data loss. In-memory compression buffers improve throughput but create a window of vulnerability during crashes.
- **Options Considered:**
 1. Synchronous writes with immediate persistence of each sample
 2. Asynchronous writes with potential sample loss during crashes
 3. Write-ahead log with background compression and checkpoint recovery
- **Decision:** Implement write-ahead logging with periodic background checkpointing
- **Rationale:** WAL provides durability without sacrificing write throughput. Background compression maintains storage efficiency. Checkpoint recovery enables fast restart without replaying entire WAL history.
- **Consequences:** Requires additional disk I/O for WAL writes. Adds complexity for crash recovery logic. Provides strong durability guarantees with good performance.

Write-Ahead Log Structure

The write-ahead log (WAL) provides an append-only record of all samples accepted by the storage engine. Unlike the compressed chunks optimized for query performance, the WAL prioritizes write speed and simplicity. Each WAL entry contains the complete information needed to reconstruct the sample: series identifier, timestamp, value, and metadata indicating the operation type.

WAL Entry Component	Size	Purpose
Entry Type	1 byte	Distinguishes sample writes from metadata operations
Series ID	8 bytes	Identifies the target time series
Timestamp	8 bytes	Sample observation time
Value	8 bytes	Metric value as IEEE 754 float64
Checksum	4 bytes	CRC32 for corruption detection

The WAL operates as a sequence of fixed-size segment files, each containing thousands of entries. When a segment reaches its maximum size (typically 64MB), the storage engine creates a new segment and continues appending to it. This segmented approach enables parallel processing during recovery and allows old segments to be deleted once their data has been safely incorporated into compressed chunks.

Each WAL entry includes a CRC32 checksum to detect corruption from incomplete writes or storage hardware failures. During recovery, entries with invalid checksums are treated as the end of valid data, preventing corrupted entries from affecting the recovered state. The WAL also includes periodic checkpoint markers that record the current state of in-memory compression buffers, enabling incremental recovery rather than full replay.

WAL writes use `fsync()` system calls to ensure data reaches persistent storage before returning control to the scrape engine. This provides strong durability guarantees at the cost of additional I/O latency. The storage engine batches multiple samples into single WAL writes when possible to amortize the `fsync()` overhead across multiple samples.

Checkpoint and Recovery Process

The checkpoint process periodically captures the complete state of in-memory compression buffers and writes this state to persistent storage. Checkpoints enable fast recovery by providing a known good state that can be augmented with subsequent WAL entries rather than replaying the entire WAL history from system startup.

During normal operation, the storage engine maintains several data structures in memory: partially compressed chunks for each active time series, label indexes mapping from values to series IDs, and series registries tracking metadata for all known series. The checkpoint process serializes all of this state into a compact binary format that can be quickly loaded during recovery.

Recovery Phase	Input	Processing	Output
Checkpoint Load	Last complete checkpoint file	Deserialize in-memory state	Active series and partial chunks
WAL Replay	WAL segments after checkpoint	Apply entries to in-memory state	Current state as of last WAL entry
Index Rebuild	Recovered series metadata	Reconstruct label indexes	Complete inverted indexes
Background Flush	In-memory chunks	Compress and write to storage	Cleaned up WAL segments

The recovery process begins by identifying the most recent valid checkpoint and loading it into memory. This provides a baseline state that may be several minutes or hours old, depending on checkpoint frequency. The system then replays all WAL entries written after the checkpoint timestamp, applying each sample to the appropriate in-memory compression buffer.

WAL replay handles several edge cases that can occur during normal operation. If a series referenced in a WAL entry doesn't exist in the checkpoint, recovery creates a new series with the appropriate metadata. If a WAL entry has an invalid checksum or refers to an impossible timestamp, recovery logs the error but continues processing subsequent entries. This provides robustness against partial corruption while preserving as much data as possible.

After WAL replay completes, the storage engine rebuilds any indexes or auxiliary data structures that aren't included in the checkpoint format. The label value indexes are reconstructed by iterating through all recovered series and building the inverted mappings. This process ensures that queries work correctly immediately after recovery without waiting for new data to populate the indexes.

Failure Scenarios and Recovery

The storage engine must handle various failure scenarios gracefully, from clean shutdowns to unexpected power loss during active writes. Each scenario requires different recovery strategies to ensure data consistency and minimize data loss.

Clean Shutdown: When the storage engine receives a shutdown signal, it completes all in-flight compression operations and writes a final checkpoint before terminating. This provides a clean recovery state where the WAL contains no uncommitted data. Recovery from clean shutdown requires only loading the final checkpoint.

Crash During Compression: If the system crashes while compressing chunks, the compressed data may be incomplete or corrupted. Recovery detects this by validating chunk headers and checksums. Corrupted chunks are

discarded, and the data is recovered from WAL entries instead. This may require replaying more WAL history but ensures data consistency.

Crash During WAL Write: A crash during WAL writing may leave a partially written entry at the end of the current segment. Recovery detects this using the entry checksums and treats the partial entry as the end of valid data. Subsequent entries (if any) are ignored, potentially losing the data from the incomplete write.

Disk Full During Operation: When disk space is exhausted, the storage engine stops accepting new samples and enters read-only mode. WAL writes are suspended to prevent further disk consumption. The system attempts to complete background garbage collection to free space before resuming normal operation. If garbage collection can't free sufficient space, the system remains in read-only mode until manual intervention.

The recovery process includes extensive validation to detect and handle corruption in both WAL segments and checkpoint files. Checksums verify data integrity at multiple levels: individual WAL entries, segment boundaries, and complete checkpoint files. When corruption is detected, the system attempts to recover as much valid data as possible while clearly logging what data may have been lost.

Common Pitfalls

⚠ Pitfall: Compression State Corruption During Concurrent Access

A common mistake is allowing multiple goroutines to modify the same compression chunk simultaneously. Gorilla compression maintains internal state about the previous timestamp and value to calculate deltas, and concurrent modifications corrupt this state. The symptoms are decompression failures or wildly incorrect values after decompression.

Why it's wrong: Compression algorithms like Gorilla rely on strict ordering of samples and consistent state across the compression sequence. Concurrent writes can interleave samples from different goroutines, violating the monotonic timestamp assumption and corrupting the delta calculations.

How to fix: Use mutex locks around each compression chunk during writes. The `ChunkBuilder` should include a `sync.Mutex` that protects the entire compress-and-append operation. Alternatively, use single-threaded compression with channels to serialize write access.

⚠ Pitfall: Index Cardinality Explosion from Naive Label Storage

Developers often create inverted indexes that store every possible label combination, leading to exponential memory growth. With labels like `{service, instance, method, status}`, the combinations explode quickly - 10 services × 100 instances × 10 methods × 5 statuses = 50,000 index entries.

Why it's wrong: Storing composite label combinations creates indexes that grow exponentially with label cardinality. Memory usage becomes unbounded, and index lookup performance degrades as hash tables become oversized.

How to fix: Create separate inverted indexes for each label name, not for label combinations. Store `service_name → [series IDs]` and `instance_name → [series IDs]` separately, then compute intersections during query time. This provides linear memory growth with label cardinality.

⚠ Pitfall: WAL Recovery Assumes Perfect Ordering

A frequent mistake is assuming that WAL entries are always perfectly ordered by timestamp during recovery. Network delays, clock drift, or buffering can cause samples to arrive and be written to the WAL in slightly different order than

their actual timestamps.

Why it's wrong: Recovery code that assumes strict timestamp ordering will fail when replaying out-of-order samples. This can manifest as assertion failures, incorrect compression, or samples being assigned to wrong time windows during replay.

How to fix: During WAL replay, buffer samples in memory and sort them by timestamp before applying to compression chunks. Only apply samples to chunks when you're confident no earlier timestamp will arrive (e.g., after a time window has passed or during final recovery).

Pitfall: Retention Policies Delete Active Chunks

Developers sometimes implement retention policies that delete chunks based solely on the chunk creation time, not the timestamp of the data within the chunk. This can delete chunks containing recent samples that were written to older chunks due to delayed ingestion.

Why it's wrong: A chunk created yesterday might contain samples with timestamps from today if data arrived late. Deleting based on chunk creation time rather than sample timestamps causes data loss for delayed metrics.

How to fix: Examine the actual timestamp range within each chunk during retention policy evaluation. Only delete chunks where the maximum sample timestamp is older than the retention threshold. Include a safety margin to account for possible clock drift.

Implementation Guidance

The storage engine requires careful coordination between compression algorithms, indexing structures, and persistence mechanisms. This implementation guidance provides concrete starting points for the core storage components while highlighting the areas where you'll implement the compression and lifecycle logic yourself.

Technology Recommendations

Component	Simple Option	Advanced Option
Compression	Custom bitstream with sync.Pool	SIMD-optimized compression with cgo
Indexing	Go maps with RWMutex	Radix trees or B+ trees
Persistence	Standard os.File with fsync	mmap with explicit sync control
WAL Format	Binary encoding/gob	Custom binary protocol
Concurrency	Mutex per chunk	Lock-free data structures

Recommended File Structure

```
internal/storage/                                GO

storage.go           ← Main StorageEngine implementation
storage_test.go     ← Integration tests

chunk/
chunk.go            ← Compression chunk implementation
gorilla.go          ← Gorilla compression algorithm
chunk_test.go       ← Compression algorithm tests

index/
inverted.go         ← Inverted index implementation
cardinality.go      ← Cardinality tracking
index_test.go       ← Index operation tests

wal/
wal.go              ← Write-ahead log
recovery.go         ← Crash recovery logic
wal_test.go         ← WAL and recovery tests

retention/
policy.go           ← Retention policy evaluation
compaction.go       ← Background compaction
retention_test.go   ← Lifecycle management tests
```

Storage Infrastructure Code

```
// storage.go - Complete infrastructure for storage operations
```

package storage

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

```
// StorageEngine manages time series persistence with compression and indexing
```

```
type StorageEngine struct {
```

- config *StorageConfig
- logger Logger
- chunks map[uint64]*CompressedChunk // series_id -> current chunk
- indexes *InvertedIndexes
- wal *WriteAheadLog
- mu sync.RWMutex
- shutdown chan struct{}

```
}
```

```
// NewStorageEngine creates a configured storage engine instance
```

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

```
    se := &StorageEngine{
```

- config: config,
- logger: logger,
- chunks: make(map[uint64]*CompressedChunk),
- indexes: NewInvertedIndexes(),
- shutdown: make(chan struct{}),

GO

```
}

var err error

se.wal, err = NewWriteAheadLog(config.data_directory, logger)

if err != nil {

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

if err := se.recoverFromWAL(); err != nil {

    return nil, fmt.Errorf("WAL recovery failed: %w", err)
}

go se.backgroundCompaction()

return se, nil
}

// CompressedChunk represents a time series chunk with Gorilla compression

type CompressedChunk struct {

    seriesID      uint64

    mint, maxt    int64        // min/max timestamp in chunk

    samples       []byte       // compressed sample data

    sampleCount   int

    mu            sync.Mutex
}

// WriteAheadLog provides durable storage for incoming samples

type WriteAheadLog struct {

    dir           string

    current      *os.File
```

```
    segments  []string

    logger    Logger

    mu        sync.Mutex

}

// InvertedIndexes maintains label->series mappings for fast queries

type InvertedIndexes struct {

    metricNames map[string][]uint64           // metric -> series IDs

    labelValues map[string]map[string][]uint64   // label_name -> value -> series IDs

    series      map[uint64]*SeriesMetadata      // series_id -> metadata

    mu          sync.RWMutex

}

// SeriesMetadata contains the complete label set and storage info for a series

type SeriesMetadata struct {

    ID          uint64

    Labels      Labels

    MetricName  string

    ChunkRefs   []ChunkRef // references to storage chunks

}

// ChunkRef points to a specific chunk in persistent storage

type ChunkRef struct {

    MinTime, MaxTime int64

    Offset         int64   // file offset

    Length        int32   // compressed size

}
```

Core Storage Logic Skeletons

```
// Append stores samples for the specified time series                                GO
// This is the main ingestion entry point called by the scrape engine

func (se *StorageEngine) Append(samples []*Sample) error {
    // TODO 1: Write samples to WAL before in-memory processing for durability
    // TODO 2: Group samples by series ID to batch processing per series
    // TODO 3: For each series, find or create the current compression chunk
    // TODO 4: Append samples to compression chunk using Gorilla algorithm
    // TODO 5: If chunk reaches size limit, flush to persistent storage
    // TODO 6: Update inverted indexes with any new label values discovered
    // TODO 7: Return error if any step fails (WAL write, compression, indexing)
    // Hint: Use se.mu.RLock() for reading, se.mu.Lock() for writing chunk map
    // Hint: Batch WAL writes for multiple samples to improve throughput
}

// Select retrieves time series matching the label matchers in the time range

func (se *StorageEngine) Select(start, end int64, matchers []*LabelMatcher) (SeriesSet, error) {
    se.mu.RLock()
    defer se.mu.RUnlock()

    // TODO 1: Use metric name matcher to get candidate series IDs from primary index
    // TODO 2: For each remaining label matcher, get matching series IDs from label indexes
    // TODO 3: Compute intersection of all matcher results to get final series ID set
    // TODO 4: For each matching series ID, load chunk references from series metadata
    // TODO 5: Filter chunk references to only those overlapping [start, end] time range
    // TODO 6: Return SeriesSet that can iterate through matching series and decompress chunks
    // Hint: Order matchers by selectivity (lowest cardinality first) for efficiency
    // Hint: Use sorted slice intersection for large result sets
}
```

Gorilla Compression Implementation Skeleton

```
// gorilla.go - Gorilla compression algorithm implementation          GO

package chunk

import (
    "encoding/binary"
    "math"
)

// GorillaCompressor implements delta-of-delta timestamp and XOR value compression

type GorillaCompressor struct {

    buf          []byte
    bitPos       int

    // Timestamp compression state
    baseTimestamp   int64
    prevTimestamp   int64
    prevDelta       int64

    // Value compression state
    baseValue      float64
    prevValue      float64
    prevLeadingZeros int
    prevTrailingZeros int
}

// NewGorillaCompressor creates a new compressor with baseline values

func NewGorillaCompressor(baseTimestamp int64, baseValue float64) *GorillaCompressor {
    // TODO 1: Initialize compressor with baseline timestamp and value
    // TODO 2: Allocate initial buffer for compressed bits (start with 1KB)
}
```

```
// TODO 3: Write baseline timestamp and value as uncompressed 64-bit values

// TODO 4: Set initial state for delta-of-delta and XOR compression

// Hint: Store baseline values in first 16 bytes for decompression

}

// AddSample compresses and appends a timestamp-value pair

func (gc *GorillaCompressor) AddSample(timestamp int64, value float64) error {

    // TODO 1: Calculate timestamp delta from previous timestamp

    // TODO 2: Calculate delta-of-delta from previous delta

    // TODO 3: Encode delta-of-delta using variable-length bit patterns

    // TODO 4: XOR current value with previous value

    // TODO 5: Encode XOR result using leading/trailing zero compression

    // TODO 6: Update internal state for next sample (prev timestamp, value, etc.)

    // TODO 7: Expand buffer if needed to accommodate new bits

    // Hint: Use bit manipulation helpers for variable-length encoding

    // Hint: Delta-of-delta of 0 encodes as single bit, others use longer patterns

}

// CompressedData returns the final compressed chunk bytes

func (gc *GorillaCompressor) CompressedData() []byte {

    // TODO 1: Finalize bit stream by padding to byte boundary if needed

    // TODO 2: Return copy of buffer truncated to actual used length

    // TODO 3: Include header with baseline timestamp/value and sample count

    // Hint: Byte-align the final bit position for easier storage

}
```

WAL and Recovery Implementation Skeleton

```
// wal.go - Write-ahead log for durability                                GO

package wal

// AppendSamples writes samples to WAL before in-memory processing

func (wal *WriteAheadLog) AppendSamples(samples []*Sample) error {

    wal.mu.Lock()

    defer wal.mu.Unlock()

    // TODO 1: Create WAL entry for each sample with type, series ID, timestamp, value

    // TODO 2: Calculate CRC32 checksum for each entry to detect corruption

    // TODO 3: Write entries to current WAL segment file

    // TODO 4: Call fsync() to ensure data reaches persistent storage

    // TODO 5: If current segment exceeds size limit, rotate to new segment

    // TODO 6: Return error if any write or fsync operation fails

    // Hint: Batch multiple samples into single write() call for efficiency

    // Hint: Use binary.Write() for consistent cross-platform encoding

}

// recoverFromWAL replays WAL entries to rebuild in-memory state

func (se *StorageEngine) recoverFromWAL() error {

    // TODO 1: Find all WAL segment files in data directory sorted by sequence number

    // TODO 2: Load most recent checkpoint file to get baseline state

    // TODO 3: Replay WAL entries from checkpoint timestamp to end of latest segment

    // TODO 4: For each entry, validate checksum and skip corrupted entries

    // TODO 5: Apply valid entries to in-memory chunks and update indexes

    // TODO 6: Rebuild any derived state (label indexes, series registry)

    // TODO 7: Delete WAL segments that are fully incorporated into chunks

    // Hint: Buffer and sort samples by timestamp before applying to handle out-of-order
```

```
// Hint: Log recovery progress for operational visibility  
}
```

Milestone Checkpoint

After implementing the storage engine core:

Basic Functionality Test:

```
go test ./internal/storage/... -v  
# Should pass tests for compression, indexing, and WAL recovery
```

BASH

Manual Verification Steps:

1. Start storage engine and append 1000 samples across 10 different series
2. Verify WAL file is created and contains expected number of entries
3. Query samples back using label matchers - should return correct data
4. Stop process and restart - verify recovery loads all data correctly
5. Wait for background compaction - verify compressed chunks are created

Performance Validation:

- Should handle 10,000 samples/second ingestion rate
- Gorilla compression should achieve 5:1 compression ratio or better
- Label queries should return results in < 100ms for up to 10,000 series
- WAL recovery should complete in < 30 seconds for 1GB of WAL data

Warning Signs:

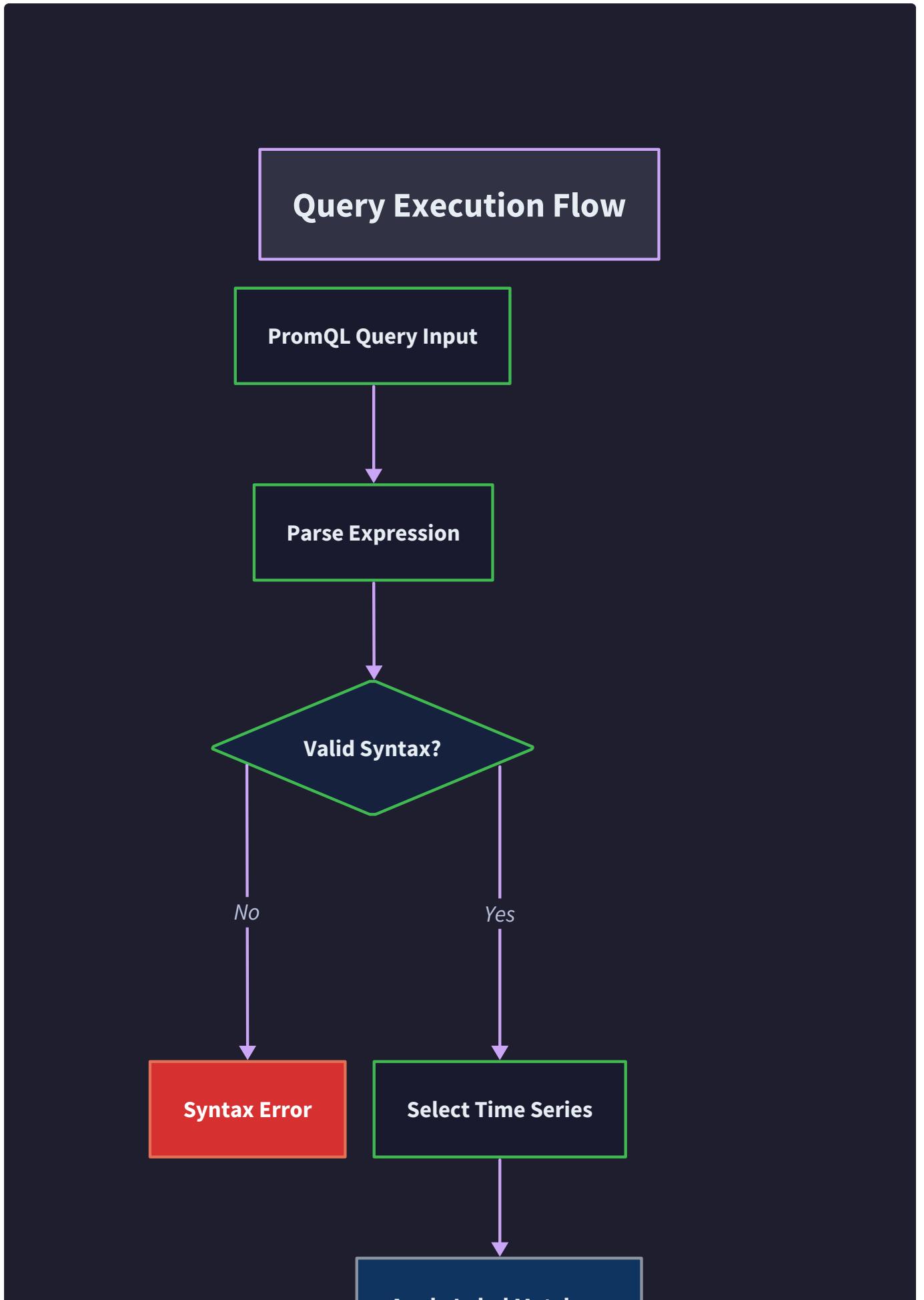
- Memory usage growing without bound (likely cardinality explosion)
- Compression ratio worse than 2:1 (implementation bug in Gorilla algorithm)
- Recovery taking > 5 minutes (WAL segments not being cleaned up properly)
- Query timeouts on small datasets (inefficient index intersection)

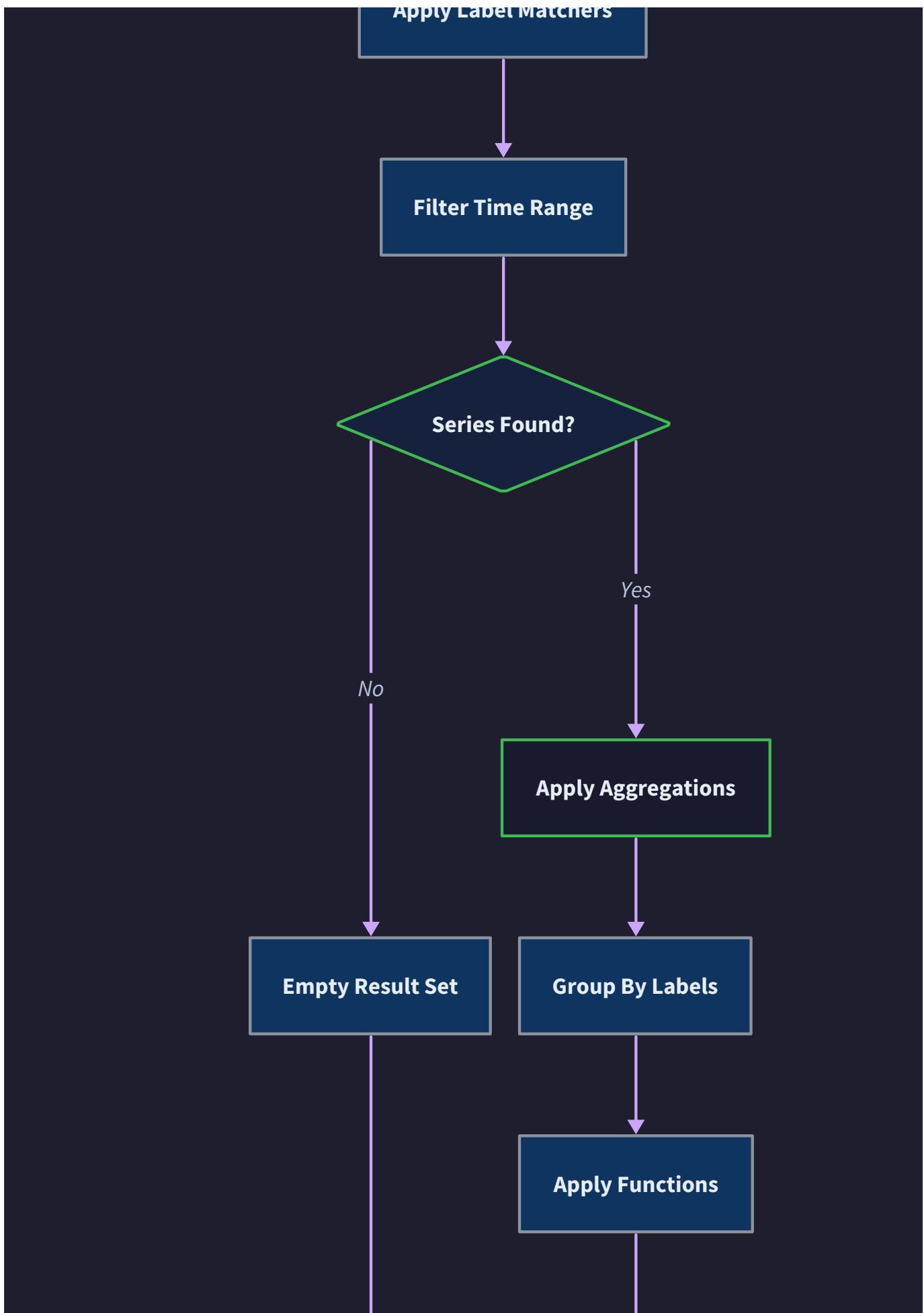
PromQL Query Engine

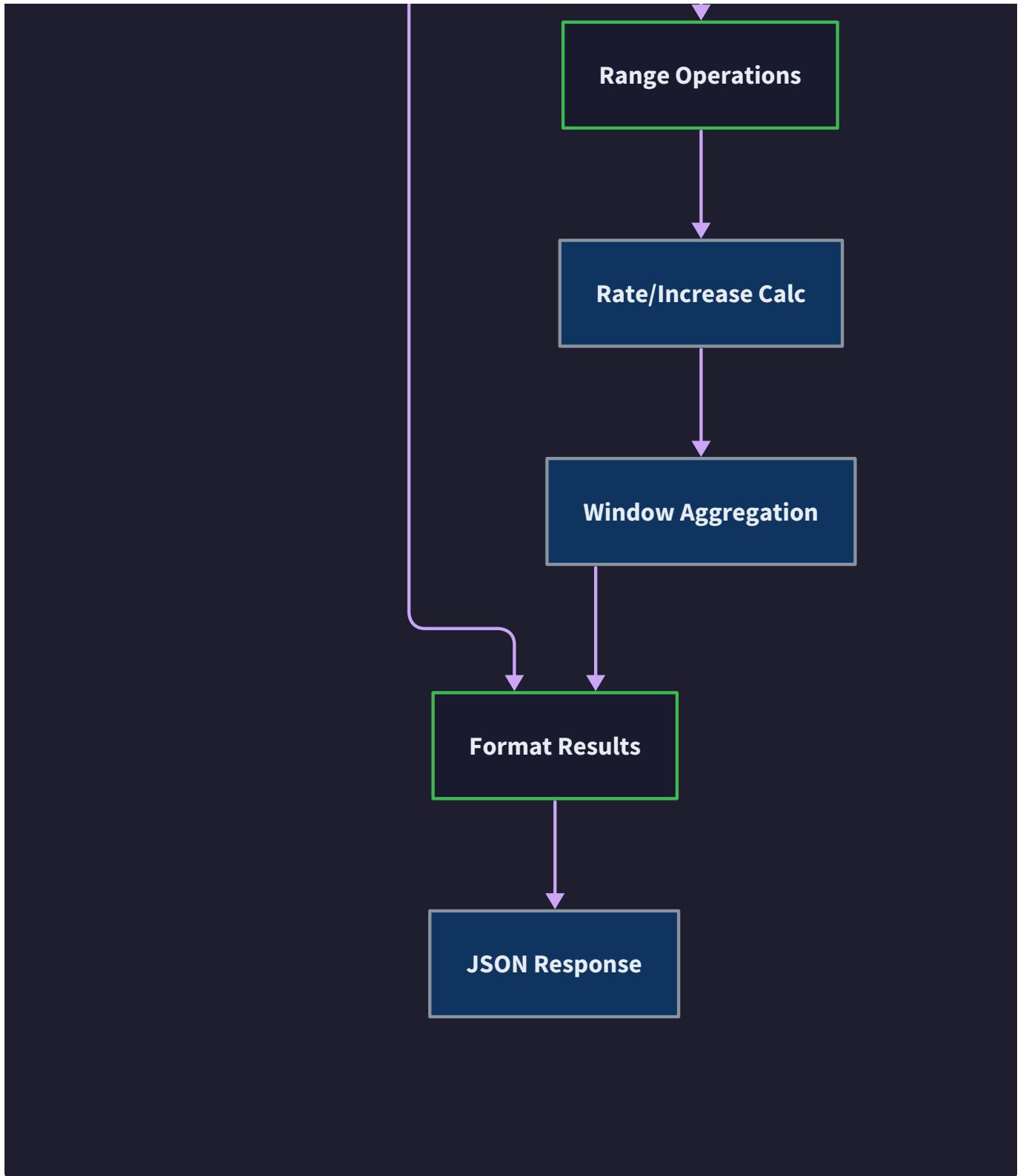
Milestone(s): This section directly corresponds to Milestone 4 (Query Engine) and integrates with all previous milestones by providing the query capabilities that retrieve and process the metrics data collected by the Scrape Engine (Milestone 2) and stored by the Time Series Storage Engine (Milestone 3) using the Metrics Data Model (Milestone 1).

The PromQL query engine transforms user queries into results by parsing expressions, selecting time series, applying aggregations, and formatting output. This component sits at the top of our metrics system architecture,

providing the interface through which users extract insights from collected observability data. The query engine must balance expressiveness with performance, supporting complex analytical queries while maintaining sub-second response times over millions of time series.







The query engine operates through five coordinated stages: expression parsing transforms text queries into executable plans, label selection filters the universe of available time series, aggregation operations combine data across dimensions, range processing handles time-windowed queries, and result formatting prepares output for consumption. Each stage must handle high cardinality gracefully while providing meaningful error messages when queries cannot be satisfied.

SQL for Time Series Mental Model: Understanding PromQL through database query analogies

Think of PromQL as **SQL for time series data**, where instead of selecting rows from tables, you're selecting and aggregating sequences of timestamped values. Just as SQL has `SELECT`, `FROM`, `WHERE`, `GROUP BY`, and aggregate functions, PromQL has metric selection, label filtering, time range specification, grouping operations, and mathematical functions. The key difference is that PromQL operates on the time dimension as a first-class concept, making it natural to ask questions like "what was the average CPU usage over the last hour, grouped by service?"

In traditional SQL, you might write `SELECT AVG(cpu_percent) FROM metrics WHERE service='api' AND timestamp > NOW() - INTERVAL '1 hour'`. In PromQL, this becomes `avg_over_time(cpu_percent{service="api"}[1h])`. The bracket notation `[1h]` selects a time range, the curly braces `{service="api"}` provide label filtering (equivalent to WHERE clauses), and `avg_over_time()` aggregates across the time dimension rather than across rows.

The **metric name acts like a table name** in SQL - it identifies the primary data source. Labels function like **indexed columns**, allowing efficient filtering and grouping operations. When you write

`http_requests_total{method="GET", status="200"}`, you're essentially saying "FROM http_requests_total WHERE method='GET' AND status='200'". The time series database uses inverted indexes on labels just like SQL databases use B-tree indexes on columns.

Aggregation in PromQL differs from SQL aggregation because it operates across multiple dimensions simultaneously. SQL typically aggregates rows within groups, but PromQL aggregates both across time (within each series) and across series (within label groups). When you write `sum by (service)` (`rate(http_requests_total[5m])`), you're first calculating the per-second rate for each individual time series over 5-minute windows, then summing those rates across all series that share the same `service` label value.

The **time dimension** is what makes PromQL unique. Every query implicitly or explicitly operates over time ranges. An "instant query" like `cpu_usage{instance="server1"}` asks for the most recent value, while a "range query" like `cpu_usage{instance="server1"}[30m:1m]` asks for all values in the last 30 minutes at 1-minute resolution. This is similar to SQL window functions but built into the core query language.

Key Insight: PromQL treats time as a fundamental dimension, not just another column. This makes temporal operations natural but requires different thinking than traditional SQL query planning.

Query Parsing: Lexical analysis and AST construction for PromQL expressions

The query parser transforms text-based PromQL expressions into an **Abstract Syntax Tree (AST)** that represents the logical structure of operations and their dependencies. This parsing process follows the standard compiler design pattern of lexical analysis (tokenization) followed by syntactic analysis (AST construction), but with domain-specific considerations for time series expressions, mathematical operators, and label selectors.

Lexical analysis (tokenization) breaks the input string into a sequence of meaningful tokens: metric names, label selectors, operators, function names, duration literals, and numeric constants. The lexer must handle PromQL-specific syntax like label selector curly braces `{job="api"}`, range selectors with square brackets `[5m]`, and mathematical operators with appropriate precedence. Duration parsing requires special attention since PromQL supports units like `5m`, `2h`, `30s` that must be normalized to consistent internal representations.

Token Type	Examples	Description
IDENTIFIER	http_requests_total, instance, job	Metric names and label names
STRING	"api", "GET", "/health"	Label values in quotes
NUMBER	123, 3.14, 1e6	Numeric literals
DURATION	5m, 2h, 30s, 1d	Time duration specifiers
OPERATOR	+ , - , * , / , % , ^	Mathematical operators
COMPARATOR	== , != , =~ , !~	Label matching operators
AGGREGATOR	sum, avg, max, min, count	Aggregation function names
FUNCTION	rate, increase, histogram_quantile	Built-in function names
LBRACE	{	Start of label selector
RBRACE	}	End of label selector
LBRACKET	[Start of range selector
RBRACKET]	End of range selector
LPAREN	(Function call or grouping start
RPAREN)	Function call or grouping end

Syntactic analysis builds the AST by recognizing grammar patterns and enforcing operator precedence. PromQL expressions follow a hierarchy where function calls bind most tightly, followed by mathematical operators (with standard precedence: $\wedge > \ast, /, \% > +, -$), then aggregation operations, and finally binary operators between entire expressions. The parser must handle both instant queries (returning single values per series) and range queries (returning time-windowed data).

The **AST node structure** represents each operation type with specific node classes that capture the operation's semantics and operands:

Node Type	Purpose	Children	Attributes
MetricSelectorNode	Selects time series by name and labels	None	MetricName string , LabelMatchers []LabelMatcher
RangeSelectorNode	Applies time range to metric selector	MetricSelectorNode	Duration time.Duration , Offset time.Duration
FunctionCallNode	Applies function to arguments	Variable argument nodes	FunctionName string , Args []ASTNode
AggregationNode	Groups and aggregates series	Expression node	Operation string , GroupBy []string , Without []string
BinaryOperationNode	Mathematical operation between expressions	Left and right expression nodes	Operator string , Matching *VectorMatching
NumberLiteralNode	Numeric constant	None	Value float64
StringLiteralNode	String constant	None	Value string

Error handling during parsing must provide meaningful diagnostics that help users understand syntax problems. Common parsing errors include mismatched brackets in label selectors, invalid duration formats, undefined function names, and operator precedence confusion. The parser should report the exact character position and suggest corrections when possible.

Decision: Recursive Descent vs. Parser Generator

- **Context:** Need to choose parsing strategy for PromQL expressions
- **Options Considered:** Hand-written recursive descent, YACC/Bison parser generator, PEG parser
- **Decision:** Hand-written recursive descent parser
- **Rationale:** PromQL grammar is relatively simple, recursive descent provides better error messages and easier debugging, eliminates external tool dependencies
- **Consequences:** More code to maintain but complete control over error reporting and parsing behavior

The **parsing algorithm** follows these steps for each input query:

1. **Initialize lexer** with input string and begin tokenization
2. **Parse primary expression** starting with metric selector, number literal, or parenthesized expression
3. **Handle range selector** if current token is `[` - parse duration and optional offset
4. **Process function calls** if primary expression is followed by `(` - parse argument list recursively
5. **Parse mathematical operators** according to precedence rules - left-associative except for `^`
6. **Handle aggregation operations** if expression starts with aggregation keyword - parse grouping clauses

7. **Validate semantic constraints** - ensure range selectors only applied to metric selectors, function argument types match
8. **Return AST root node** or parsing error with location information

Common parsing pitfalls include incorrect operator precedence handling, failure to validate semantic constraints during parsing (allowing syntactically valid but meaningless expressions), and poor error recovery that cascades single mistakes into many error messages.

⚠ Pitfall: Precedence Confusion Many developers incorrectly implement operator precedence, leading to expressions like `2 + 3 * 4` being parsed as `(2 + 3) * 4` instead of `2 + (3 * 4)`. Always implement precedence through recursive parsing methods where higher-precedence operators are parsed by deeper recursion levels, ensuring natural left-to-right evaluation with correct operator binding.

Label Selector Engine: Exact, regex, and inequality matching for filtering time series

The label selector engine filters the universe of available time series down to those matching specific label criteria, acting as the **WHERE clause** of PromQL queries. This component must efficiently handle four types of label matching: exact equality, inequality, regular expression matching, and negative regular expression matching. Performance is critical since label selection often processes millions of time series and determines the working set size for subsequent aggregation operations.

Label matching semantics define how series are included or excluded based on their label values. The four matcher types each serve different filtering use cases and have distinct performance characteristics:

Matcher Type	Syntax	Example	Use Case	Performance Notes
Exact Match	<code>label="value"</code>	<code>job="api"</code>	Filter by known label values	Fastest - direct index lookup
Not Equal	<code>label!="value"</code>	<code>status!="200"</code>	Exclude specific values	Moderate - inverse index scan
Regex Match	<code>label=~"pattern"</code>	<code>instance=~"web-.*"</code>	Pattern-based inclusion	Slowest - requires regex evaluation
Negative Regex	<code>label!~"pattern"</code>	<code>path!~/debug/.*</code>	Pattern-based exclusion	Slowest - requires regex evaluation

The **matching algorithm** processes label selectors against the inverted indexes built by the storage engine. For exact matches, the engine performs direct hash table lookups in the `labelValues` index to retrieve the list of series IDs containing that label-value pair. For inequality matches, the engine retrieves the exact match set and computes its complement against all series containing that label name.

Regular expression matching requires compiling regex patterns and evaluating them against all possible values for the specified label name. This is inherently expensive since it cannot use pre-built indexes effectively. The selector engine maintains a **regex compilation cache** to avoid recompiling the same patterns repeatedly, and applies optimizations like prefix extraction when patterns start with literal strings.

Label Selection Step	Operation	Data Structure Used	Time Complexity
Exact Match Lookup	<code>labelValues[labelName][labelValue]</code>	Hash map of hash maps	$O(1)$ average case
Inequality Filtering	Set complement operation	Bitmap or hash set	$O(n)$ where $n = \text{series count}$
Regex Compilation	<code>regexp.MustCompile(pattern)</code>	Cached compiled regex	$O(m)$ where $m = \text{pattern complexity}$
Regex Evaluation	Pattern matching against all values	Linear scan with regex	$O(k*m)$ where $k = \text{unique values}$

The **series intersection algorithm** combines multiple label matchers using set operations. When a query contains multiple label selectors like `{job="api", status!="500", instance=~"web-.+"}`, the engine must find series that satisfy ALL conditions simultaneously. The algorithm processes matchers in optimal order, starting with the most selective (smallest result set) to minimize subsequent work.

Intersection processing follows these steps:

1. **Evaluate each matcher independently** to get per-matcher series ID sets
2. **Sort matchers by selectivity** - exact matches first, then inequalities, finally regex patterns
3. **Initialize result set** with most selective matcher's series IDs
4. **Intersect remaining matchers** - remove series IDs that don't appear in subsequent matcher results
5. **Handle special cases** - empty label selectors match all series, contradictory selectors return empty sets
6. **Return final series ID list** sorted for consistent downstream processing

Label name filtering handles cases where queries filter on labels that may not exist on all series. A matcher like `{version="1.2"}` should only consider series that actually have a `version` label, ignoring series where this label is absent entirely. The absence of a label is semantically different from a label with an empty string value.

Decision: Bitmap vs. Hash Set for Series Intersection

- **Context:** Need efficient set operations for combining label matcher results
- **Options Considered:** Hash sets, sorted arrays with binary search, bitmap arrays
- **Decision:** Hash sets for low cardinality, bitmap arrays for high cardinality with cutoff at 10,000 series
- **Rationale:** Hash sets provide $O(1)$ membership testing but have memory overhead; bitmaps are more space-efficient for dense ID ranges but require more memory for sparse ranges
- **Consequences:** Adaptive algorithm provides good performance across different cardinality ranges but adds implementation complexity

Performance optimizations for label selection include:

- **Early termination:** Stop processing additional matchers if intermediate result set becomes empty
- **Matcher reordering:** Process most selective matchers first to minimize subsequent intersection work
- **Regex optimization:** Extract literal prefixes from regex patterns to use index lookups before pattern matching

- **Cache compiled regexes:** Avoid recompiling the same patterns across multiple query executions
- **Parallel evaluation:** Process independent matchers concurrently when result sets are large

Common label selection pitfalls include incorrect handling of missing labels (treating absence as empty string), inefficient regex patterns that can't use index optimizations, and failure to validate regex syntax at parse time leading to runtime compilation errors.

⚠ Pitfall: Missing Label Semantics A common mistake is treating missing labels as empty strings. The query `{version=""}` should match series with an empty `version` label, while `{version!="stable"}` should match series with any `version` value except "stable" but NOT series missing the `version` label entirely. Always check for label existence before value comparison.

Aggregation Operations: Sum, average, percentile, and grouping operations across label dimensions

Aggregation operations combine multiple time series into fewer series by applying mathematical functions across specified label dimensions. Think of aggregation as **GROUP BY operations in SQL**, where you collapse many rows into summary values, except PromQL aggregates across both the series dimension (combining multiple time series) and optionally the time dimension (combining multiple timestamps within each series).

The **aggregation mental model** treats each time series as a vector of timestamped values, and aggregation as matrix operations that combine these vectors according to grouping rules. When you write `sum by (service) (cpu_usage)`, you're partitioning all `cpu_usage` time series into groups based on their `service` label value, then computing the element-wise sum within each group to produce one output series per unique service.

Grouping semantics determine which series get combined together. PromQL supports two grouping modes: `by` (include specified labels in output) and `without` (exclude specified labels from output). The grouping operation creates **aggregation groups** where each group contains series that share identical values for the grouping labels.

Grouping Mode	Syntax	Example	Behavior
Group By	<code>sum by (label1, label2) (metric)</code>	<code>sum by (service) (cpu_usage)</code>	Output series contain only specified labels
Group Without	<code>sum without (label1, label2) (metric)</code>	<code>sum without (instance) (memory_usage)</code>	Output series exclude specified labels
No Grouping	<code>sum(metric)</code>	<code>sum(active_connections)</code>	All series combined into single output series

The **aggregation algorithm** processes series in groups, applying the aggregation function to corresponding timestamps across all series within each group. For a timestamp T, the aggregator collects all values at timestamp T from series in the group, applies the mathematical function (sum, average, max, etc.), and produces a single output value for that timestamp.

Aggregation Function	Mathematical Operation	Null Value Handling	Use Cases
sum	Add all values	Skip missing values	Total resource usage, request counts
avg	Sum divided by count	Skip missing values	Average response times, utilization rates
max	Largest value	Skip missing values	Peak resource usage, worst-case latencies
min	Smallest value	Skip missing values	Best-case performance, minimum availability
count	Number of series	Count series with any value	Number of active instances, error rate calculation
stddev	Standard deviation	Skip missing values	Performance consistency, outlier detection
quantile	Percentile calculation	Skip missing values	SLA monitoring, latency distribution analysis

Quantile aggregation deserves special attention because it requires maintaining the full distribution of values rather than computing incremental statistics. The `quantile(0.95, metric)` function needs all individual values at each timestamp to determine the 95th percentile, making it more memory-intensive than functions like sum or average that can be computed incrementally.

Implementation steps for aggregation processing:

1. **Parse grouping specification** - extract `by` or `without` label lists from AST
2. **Group series by label values** - create aggregation groups based on grouping rules
3. **Align timestamps across series** - ensure all series in each group have consistent timestamp alignment
4. **Apply aggregation function** - for each timestamp, collect values from all series in group and compute result
5. **Handle missing values** - skip series with no value at specific timestamps or use last-observed-value
6. **Construct output series** - create result time series with group labels and aggregated values
7. **Sort output consistently** - ensure deterministic ordering for reproducible results

Memory management during aggregation requires careful attention since large aggregation groups can consume significant memory. The aggregator processes timestamps sequentially rather than loading entire series into memory, using a sliding window approach for functions that need historical context.

Label handling in aggregation output requires combining the labels from input series according to grouping rules. For `sum by (service) (http_requests{service="api", instance="web-1", method="GET"})`, the output series contains only `{service="api"}`, dropping the `instance` and `method` labels. The aggregator must compute the Cartesian product of all possible label value combinations for the grouping labels.

Input Series	Labels	Grouping: by (service, status)	Output Group Key
Series 1	{service="api", status="200", instance="web-1"}	service=api, status=200	{service="api", status="200"}
Series 2	{service="api", status="200", instance="web-2"}	service=api, status=200	{service="api", status="200"}
Series 3	{service="api", status="500", instance="web-1"}	service=api, status=500	{service="api", status="500"}
Series 4	{service="db", status="200", instance="db-1"}	service=db, status=200	{service="db", status="200"}

Decision: Streaming vs. Batch Aggregation

- **Context:** Choose processing model for aggregating large numbers of time series
- **Options Considered:** Load all data then aggregate, stream processing with fixed memory, hybrid approach
- **Decision:** Streaming aggregation with timestamp-aligned windows
- **Rationale:** Provides bounded memory usage regardless of input size, enables processing datasets larger than available RAM, maintains low latency for small aggregations
- **Consequences:** More complex implementation with windowing logic but scales to arbitrary dataset sizes without memory exhaustion

Performance considerations for aggregation include:

- **Series cardinality:** Aggregations over high-cardinality labels create many output groups, increasing memory usage
- **Timestamp alignment:** Series with misaligned timestamps require interpolation or bucketing to aggregate consistently
- **Lazy evaluation:** Defer aggregation computation until results are actually needed by subsequent operations
- **Parallel processing:** Process independent aggregation groups concurrently when group count is high
- **Memory pooling:** Reuse buffers for intermediate calculations to reduce garbage collection pressure

Common aggregation mistakes include incorrect null value handling (treating missing data as zero instead of skipping it), memory exhaustion on high-cardinality aggregations, and timestamp alignment issues when series have different scrape intervals.

⚠ Pitfall: High-Cardinality Aggregation Groups Aggregating `by (instance_id)` where `instance_id` is a UUID will create one output series per instance, potentially millions of series. This defeats the purpose of aggregation and can cause memory exhaustion. Always aggregate by low-cardinality labels like service names, not high-cardinality identifiers like instance IDs or request IDs.

Range Query Execution: Retrieving and interpolating data points across time windows

Range queries retrieve time series data across specified time windows, returning multiple data points per series rather than single instant values. Think of range queries as **time-windowed SELECT statements** that scan horizontally across the time dimension, collecting all data points between start and end timestamps at regular step intervals. The range query executor must handle timestamp alignment, missing data interpolation, and efficient data retrieval from the underlying storage engine.

Range query semantics define three key parameters: start time (beginning of time window), end time (end of time window), and step duration (resolution of output data points). A range query like `/api/v1/query_range?query=cpu_usage&start=1609459200&end=1609462800&step=60` requests CPU usage data from timestamp 1609459200 to 1609462800 with 60-second resolution, producing one output value per series every minute.

The **query execution model** operates differently from instant queries because it must process multiple timestamps sequentially. Instead of asking "what is the current value?", range queries ask "what were all the values between time A and time B at intervals of N seconds?". This requires coordinating between timestamp generation (creating the output timeline) and value retrieval (fetching data from storage).

Range Query Component	Purpose	Input	Output
Time Range Parser	Extract start/end/step from parameters	Query string parameters	Parsed time boundaries
Step Generator	Create output timestamp sequence	Start, end, step duration	Array of query timestamps
Series Selector	Find matching time series	Label matchers	List of series IDs
Data Retriever	Fetch samples from storage	Series IDs, time range	Raw sample data
Value Interpolator	Fill gaps in data	Sparse samples, query timestamps	Dense timestamp-value pairs
Result Formatter	Structure output data	Dense data, metadata	JSON/HTTP response

Timestamp alignment ensures that output data points appear at regular intervals regardless of when the underlying samples were actually collected. If the query requests data every 60 seconds starting at 12:00:00, the output timestamps should be 12:00:00, 12:01:00, 12:02:00, etc., even if the scraped samples occurred at 12:00:15, 12:01:03, 12:02:21, etc.

The **alignment algorithm** follows these steps:

1. **Generate query timeline** - create sequence of timestamps from start to end at step intervals
2. **Retrieve raw sample data** - fetch all samples for selected series within the time range
3. **Sort samples by timestamp** - ensure chronological ordering for interpolation
4. **Align to query timeline** - for each query timestamp, find the closest sample within tolerance
5. **Interpolate missing values** - use last-observed-value or linear interpolation for gaps
6. **Handle staleness** - mark values as stale if no recent sample exists within staleness threshold

7. Format output matrix - structure results as matrix with consistent timestamp alignment

Missing data handling is crucial because real monitoring systems have gaps due to network failures, service restarts, or scraping errors. The range query executor must distinguish between three scenarios: temporarily missing data (use interpolation), permanently missing data (return null), and series that don't exist during part of the time range (return partial results).

Missing Data Scenario	Detection Criteria	Handling Strategy	Output Behavior
Recent Gap	No sample within staleness threshold	Return null value	Gaps in output timeline
Historical Gap	Sample exists before/after gap	Last-observed-value interpolation	Filled with previous value
Series Creation	First sample after query start time	Partial series data	Null values before first sample
Series Deletion	Last sample before query end time	Partial series data	Null values after last sample
Complete Absence	No samples in entire time range	Empty series	All null values

Storage integration requires efficient range scans that minimize disk I/O and decompression overhead. The range query executor works with the storage engine's `Select(start, end, matchers)` method to retrieve only relevant data, avoiding the cost of scanning unrelated time series or unnecessary time ranges.

Interpolation strategies determine how to fill gaps between actual sample timestamps and desired query timestamps. The most common approach is **last-observed-value** interpolation, where each query timestamp receives the value of the most recent actual sample. This reflects the reality that metrics typically represent ongoing states (like CPU usage) that remain valid until explicitly updated.

Decision: Last-Observed-Value vs. Linear Interpolation

- **Context:** Choose interpolation strategy for aligning samples to query timestamps
- **Options Considered:** Nearest neighbor, last-observed-value, linear interpolation, no interpolation
- **Decision:** Last-observed-value with 5-minute staleness threshold
- **Rationale:** Reflects semantic meaning of monitoring metrics which represent current state until updated; linear interpolation implies continuous change which is incorrect for many metrics like error counts
- **Consequences:** More accurate representation of monitoring data but can show "stair-step" graphs instead of smooth curves

Performance optimizations for range queries include:

- **Parallel series processing:** Retrieve data for multiple series concurrently when series count is high
- **Chunk boundary alignment:** Align query ranges with storage chunk boundaries to minimize decompression overhead

- **Result streaming:** Begin returning results before all data is processed for large queries
- **Memory management:** Process series in batches to avoid loading entire result set into memory
- **Query caching:** Cache results for repeated range queries over the same time periods

Query complexity management prevents resource exhaustion from overly broad range queries. Limits include maximum time range duration, maximum number of series that can be processed, and maximum number of output data points. A query requesting 1-second resolution over 30 days would produce 2.6 million data points per series, potentially overwhelming the system.

Resource Limit	Default Value	Purpose	Enforcement Point
Max Range Duration	365 days	Prevent excessive time spans	Query validation
Max Series Count	10,000	Prevent high-cardinality explosions	After label selection
Max Sample Count	50 million	Prevent memory exhaustion	Before data retrieval
Query Timeout	30 seconds	Prevent long-running queries	HTTP handler level

Range query processing pipeline coordinates these steps:

1. **Parse and validate parameters** - extract start/end/step, validate ranges and limits
2. **Execute instant query parsing** - parse PromQL expression into AST
3. **Generate evaluation timestamps** - create array of timestamps at step intervals
4. **For each timestamp:** execute instant query evaluation to get point-in-time results
5. **Collect timestamp-value matrices** - accumulate results across all evaluation points
6. **Format as range query response** - structure output with series metadata and value matrices

Error handling in range queries must address scenarios like storage unavailability during long-running queries, memory exhaustion from overly large result sets, and timeout expiration during processing. The executor should provide partial results when possible rather than failing entirely.

⚠ Pitfall: Step Interval Too Small Setting step intervals much smaller than the underlying scrape interval (e.g., 1-second steps when data is scraped every 15 seconds) produces misleading results with excessive interpolation. The step should generally be no smaller than the scrape interval to avoid artificial data smoothing. Always validate that `step >= scrape_interval` for meaningful results.

Implementation Guidance

The PromQL query engine requires careful coordination between parsing, execution, and result formatting components. This implementation focuses on building a working query engine that can handle the core PromQL operations while maintaining good performance characteristics.

Technology Recommendations:

Component	Simple Option	Advanced Option
Expression Parsing	Hand-written recursive descent	ANTLR grammar generator
Regular Expressions	Go's <code>regexp</code> package	RE2 library for better performance
JSON Serialization	Go's <code>encoding/json</code>	Custom JSON streaming for large results
HTTP API	Go's <code>net/http</code> with custom handlers	Gorilla Mux for advanced routing
Concurrent Processing	Go routines with sync primitives	Worker pool pattern with channels
Caching	Simple <code>sync.Map</code> for query cache	LRU cache with TTL expiration

Recommended File Structure:

```

internal/query/
  engine.go           ← main QueryEngine type and coordination
  engine_test.go      ← end-to-end query tests
  parser/
    parser.go          ← PromQL expression parsing
    ast.go              ← AST node type definitions
    lexer.go            ← tokenization and lexical analysis
    parser_test.go      ← parsing unit tests
  selector/
    selector.go         ← label matching and series selection
    matcher.go          ← individual label matcher implementations
    selector_test.go     ← selection logic tests
  aggregation/
    aggregator.go       ← aggregation function implementations
    grouping.go          ← series grouping logic
    functions.go          ← built-in aggregation functions
    aggregation_test.go   ← aggregation correctness tests
  execution/
    instant.go          ← instant query execution
    range.go             ← range query execution
    interpolation.go     ← value interpolation and alignment
    execution_test.go     ← query execution tests

```

Core Query Engine Infrastructure (Complete):

```
package query

import (
    "context"
    "fmt"
    "sort"
    "time"
)

// QueryEngine coordinates parsing, execution, and formatting of PromQL queries

type QueryEngine struct {

    storage      StorageEngine
    parser       *ExpressionParser
    selector     *LabelSelector
    aggregator   *Aggregator
    config       QueryConfig
    queryTimeout time.Duration
}

// QueryResult represents the output of query execution

type QueryResult struct {

    ResultType string           `json:"resultType"`
    Result      interface{}      `json:"result"`
    Warnings    []string         `json:"warnings,omitempty"`
}

// InstantQueryResult represents point-in-time query results

type InstantQueryResult struct {

    Metric map[string]string `json:"metric"`
    Value  [2]interface{}     `json:"value"` // [timestamp, value]
}
```

GO

```
}

// RangeQueryResult represents time-windowed query results

type RangeQueryResult struct {

    Metric map[string]string `json:"metric"`

    Values [][]interface{}     `json:"values"` // [[timestamp, value], ...]

}

// NewQueryEngine creates a configured query engine

func NewQueryEngine(storage StorageEngine, config QueryConfig) *QueryEngine {

    return &QueryEngine{

        storage:      storage,

        parser:       NewExpressionParser(),

        selector:     NewLabelSelector(storage),

        aggregator:  NewAggregator(),

        config:       config,

        queryTimeout: config.query_timeout,

    }

}

// ExecuteInstantQuery processes point-in-time PromQL queries

func (e *QueryEngine) ExecuteInstantQuery(ctx context.Context, query string, evalTime time.Time) (*QueryResult, error) {

    ctx, cancel := context.WithTimeout(ctx, e.queryTimeout)

    defer cancel()

    // Parse expression into AST

    expr, err := e.parser.ParseExpression(query)

    if err != nil {

        return nil, fmt.Errorf("parse error: %w", err)

    }

}
```

```
// Execute expression evaluation

result, err := e.evaluateExpression(ctx, expr, evalTime)

if err != nil {

    return nil, fmt.Errorf("evaluation error: %w", err)
}

// Format results for API response

return e.formatInstantResult(result), nil
}

// ExecuteRangeQuery processes time-windowed PromQL queries

func (e *QueryEngine) ExecuteRangeQuery(ctx context.Context, query string, start, end time.Time,
step time.Duration) (*QueryResult, error) {

    ctx, cancel := context.WithTimeout(ctx, e.queryTimeout)

    defer cancel()

    // Validate range query parameters

    if err := e.validateRangeParams(start, end, step); err != nil {

        return nil, err
    }

    // Parse expression into AST

    expr, err := e.parser.ParseExpression(query)

    if err != nil {

        return nil, fmt.Errorf("parse error: %w", err)
    }

    // Generate evaluation timeline

    timestamps := e.generateTimeline(start, end, step)

    // Execute expression at each timestamp
```

```
results := make(map[string][]interface{})

for _, ts := range timestamps {

    result, err := e.evaluateExpression(ctx, expr, ts)

    if err != nil {

        return nil, fmt.Errorf("evaluation error at %v: %w", ts, err)
    }

    e.accumulateRangeResults(results, result, ts)
}

return e.formatRangeResult(results), nil
}
```

Expression Parser Implementation (Core Logic Skeleton):

```
package parser
```

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

```
// ExpressionParser handles PromQL expression parsing
```

```
type ExpressionParser struct {
    lexer *Lexer
}
```

```
// ASTNode represents nodes in the abstract syntax tree
```

```
type ASTNode interface {
    String() string
    Accept(visitor ASTVisitor) (interface{}, error)
}
```

```
// MetricSelectorNode represents metric selection with label matchers
```

```
type MetricSelectorNode struct {
    MetricName     string
    LabelMatchers []LabelMatcher
}
```

```
// ParseExpression converts PromQL text into executable AST
```

```
func (p *ExpressionParser) ParseExpression(input string) (ASTNode, error) {
    p.lexer = NewLexer(input)

    // TODO 1: Initialize lexer with input string and advance to first token
```

GO

```

// TODO 2: Parse primary expression (metric selector, number, or parenthesized expression)

// TODO 3: Check for range selector [5m] after metric selector

// TODO 4: Handle function calls if expression followed by (

// TODO 5: Parse binary operators according to precedence rules

// TODO 6: Handle aggregation operations (sum, avg, etc.) with grouping

// TODO 7: Validate semantic constraints (range selectors only on metric selectors)

// TODO 8: Return completed AST or detailed parsing error

// Hint: Use recursive descent with separate methods for each precedence level

return p.parseExpression()

}

// parseMetricSelector handles metric{label="value"} syntax

func (p *ExpressionParser) parseMetricSelector() (*MetricSelectorNode, error) {

    // TODO 1: Parse metric name identifier

    // TODO 2: Check for opening brace { to start label selector

    // TODO 3: Parse label matcher list: label op "value", ...

    // TODO 4: Validate label names and values

    // TODO 5: Ensure closing brace } terminates label selector

    // TODO 6: Return MetricSelectorNode with parsed components

    return nil, fmt.Errorf("not implemented")
}

// parseLabelMatchers handles {job="api", instance!="web-1"} syntax

func (p *ExpressionParser) parseLabelMatchers() ([]LabelMatcher, error) {

    var matchers []LabelMatcher

    // TODO 1: Parse comma-separated list of label matchers

```

```
// TODO 2: For each matcher: parse label name, operator, and value  
  
// TODO 3: Validate operator types (=, !=, =~, !~)  
  
// TODO 4: Handle quoted string values with escape sequences  
  
// TODO 5: Validate regex patterns for =~ and !~ operators  
  
// TODO 6: Return completed matcher list  
  
  
return matchers, nil  
  
}
```

Label Selector Implementation (Core Logic Skeleton):

GO

```
package selector

import (
    "regexp"
    "sort"
)

// LabelSelector filters time series based on label criteria

type LabelSelector struct {
    storage      StorageEngine
    regexCache   map[string]*regexp.Regexp
    maxCacheSize int
}

// LabelMatcher represents a single label filtering criterion

type LabelMatcher struct {
    Name      string
    Value    string
    Type     MatchType
    Regex    *regexp.Regexp
}

// MatchType defines the label matching operation

type MatchType int

const (
    MatchEqual MatchType = iota
    MatchNotEqual
    MatchRegex
    MatchNotRegex
)
```

```
// SelectSeries finds time series matching all provided label matchers

func (s *LabelSelector) SelectSeries(matchers []LabelMatcher) ([]uint64, error) {

    // TODO 1: Handle empty matcher list (return all series)

    // TODO 2: Sort matchers by estimated selectivity (exact matches first)

    // TODO 3: Evaluate most selective matcher to get initial candidate set

    // TODO 4: For remaining matchers, intersect with candidates

    // TODO 5: Handle regex matchers by compiling patterns and testing values

    // TODO 6: Return sorted list of matching series IDs

    // Hint: Use storage.indexes.labelValues for efficient exact match lookups

    return nil, fmt.Errorf("not implemented")
}

// evaluateExactMatch finds series with specific label value

func (s *LabelSelector) evaluateExactMatch(matcher LabelMatcher) ([]uint64, error) {

    // TODO 1: Look up label-value combination in inverted index

    // TODO 2: Return series ID list or empty slice if not found

    // TODO 3: Handle case where label name doesn't exist

    return nil, nil
}

// evaluateRegexMatch finds series matching regex pattern

func (s *LabelSelector) evaluateRegexMatch(matcher LabelMatcher) ([]uint64, error) {

    // TODO 1: Get compiled regex from cache or compile new pattern

    // TODO 2: Retrieve all values for the label name from index

    // TODO 3: Test each value against regex pattern

    // TODO 4: Collect series IDs for all matching values

    // TODO 5: Update regex cache if pattern was newly compiled
```

```
    return nil, nil  
}  
}
```

Aggregation Engine Implementation (Core Logic Skeleton):

GO

```
package aggregation

import (
    "math"
    "sort"
)

// Aggregator handles grouping and mathematical aggregation of time series

type Aggregator struct {
    maxGroupSize int
}

// AggregationRequest specifies how to aggregate time series data

type AggregationRequest struct {

    Function    string
    GroupBy    []string
    Without    []string
    Series     []SeriesData
    Timestamp  time.Time
}

// SeriesData represents a time series with its labels and current value

type SeriesData struct {

    Labels Labels
    Value  float64
    Valid  bool
}

// AggregateSeries applies aggregation Function to grouped time series

func (a *Aggregator) AggregateSeries(req AggregationRequest) ([]SeriesData, error) {
    // TODO 1: Create aggregation groups based on groupBy/without labels
```

```

// TODO 2: Partition input series into groups by label values

// TODO 3: Apply aggregation function within each group

// TODO 4: Handle missing/invalid values appropriately

// TODO 5: Construct output series with group labels

// TODO 6: Sort output series for consistent results


// Hint: Use createAggregationGroups helper to handle grouping logic

return nil, fmt.Errorf("not implemented")
}

// createAggregationGroups partitions series based on grouping specification

func (a *Aggregator) createAggregationGroups(series []SeriesData, groupBy, without []string)
map[string][]SeriesData {

groups := make(map[string][]SeriesData)

// TODO 1: For each input series, compute group key based on labels

// TODO 2: Handle "by" grouping: include only specified labels in key

// TODO 3: Handle "without" grouping: exclude specified labels from key

// TODO 4: Add series to appropriate group bucket

// TODO 5: Return map from group key to series list


return groups
}

// applyAggregationFunction computes result for values within a group

func (a *Aggregator) applyAggregationFunction(function string, values []float64) (float64, error) {

if len(values) == 0 {
    return 0, nil
}

```

```
switch function {  
  
    case "sum":  
  
        // TODO 1: Sum all values in the group  
  
        return 0, nil  
  
    case "avg":  
  
        // TODO 2: Calculate arithmetic mean  
  
        return 0, nil  
  
    case "max":  
  
        // TODO 3: Find maximum value  
  
        return 0, nil  
  
    case "min":  
  
        // TODO 4: Find minimum value  
  
        return 0, nil  
  
    case "count":  
  
        // TODO 5: Return count of valid values  
  
        return float64(len(values)), nil  
  
    default:  
  
        return 0, fmt.Errorf("unknown aggregation function: %s", function)  
  
}  
  
}
```

Range Query Execution (Core Logic Skeleton):

```
package execution
```

GO

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

// RangeExecutor handles time-windowed query processing

type RangeExecutor struct {
    storage          StorageEngine
    stalenessThreshold time.Duration
}

// ExecuteRangeQuery processes queries over time windows

func (r *RangeExecutor) ExecuteRangeQuery(ctx context.Context, expr ASTNode, start, end time.Time, step time.Duration) ([]RangeQueryResult, error) {
    // TODO 1: Generate array of query timestamps from start to end at step intervals
    // TODO 2: For each timestamp, execute instant query evaluation
    // TODO 3: Collect results into time series matrix format
    // TODO 4: Handle series that appear/disappear during time range
    // TODO 5: Apply staleness detection for missing data points
    // TODO 6: Format results with consistent timestamp alignment

    return nil, fmt.Errorf("not implemented")
}

// generateTimeline creates timestamp sequence for range query

func (r *RangeExecutor) generateTimeline(start, end time.Time, step time.Duration) []time.Time {
    var timestamps []time.Time

    // TODO 1: Start at aligned timestamp (round start time to step boundary)
```

```

    // TODO 2: Generate timestamps at step intervals until end time

    // TODO 3: Ensure final timestamp doesn't exceed end time

    // TODO 4: Return array of query evaluation timestamps

    return timestamps
}

// interpolateValue finds appropriate value for query timestamp

func (r *RangeExecutor) interpolateValue(samples []Sample, queryTime time.Time) (float64, bool) {
    // TODO 1: Find sample closest to query timestamp within staleness threshold

    // TODO 2: Return sample value if within threshold

    // TODO 3: Return invalid if no recent sample available

    // TODO 4: Handle edge cases (no samples, samples after query time)

    return 0, false
}

```

Milestone Checkpoints:

After implementing the core query engine components, verify functionality with these checkpoints:

- Parser Testing:** Run `go test ./internal/query/parser/...` - should parse basic metric selectors, label matchers, and mathematical expressions without errors.
- Label Selection:** Test with `curl "http://localhost:8080/api/v1/query?query=up{job=\"prometheus\"}"` - should return series matching the label selector.
- Aggregation:** Test with `curl "http://localhost:8080/api/v1/query?query=sum(up)"` - should return single aggregated value across all series.
- Range Queries:** Test with `curl "http://localhost:8080/api/v1/query_range?query=up&start=1609459200&end=1609462800&step=60"` - should return time series matrix with consistent timestamp alignment.

Debugging Tips:

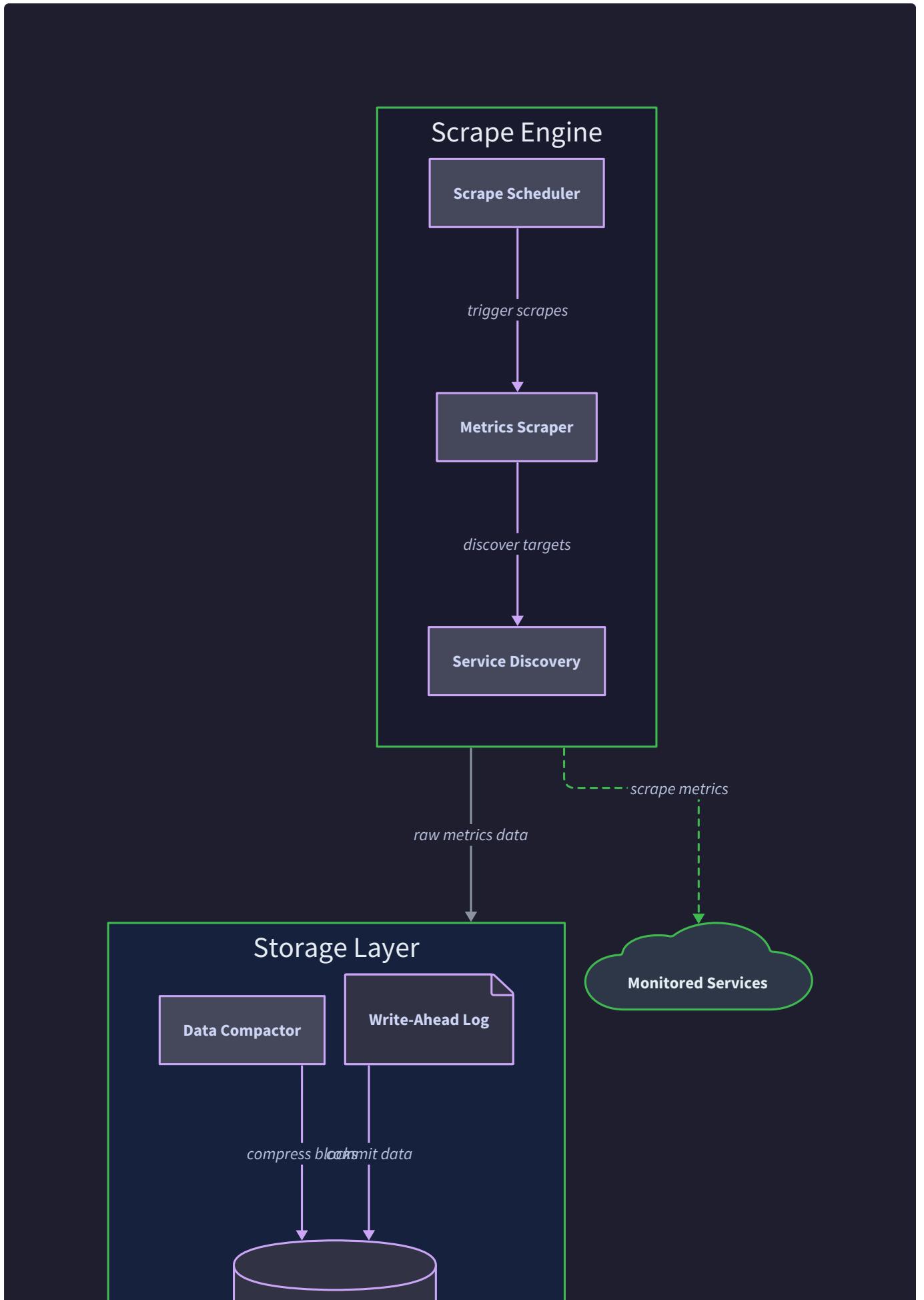
Symptom	Likely Cause	How to Diagnose	Fix
Parse errors on valid PromQL	Incorrect operator precedence or tokenization	Add debug logging to lexer token output	Fix precedence in recursive descent methods
Empty query results	Label matcher not finding series	Check inverted index contents and matcher logic	Verify label matcher evaluation against actual stored labels
Memory exhaustion on aggregation	High cardinality grouping labels	Monitor group count and series per group	Add cardinality limits and validate grouping labels
Slow range queries	Inefficient timestamp generation or storage access	Profile CPU and measure storage query time	Optimize timeline generation and batch storage requests
Inconsistent results	Race conditions in concurrent processing	Test with single-threaded execution	Add proper synchronization around shared data structures

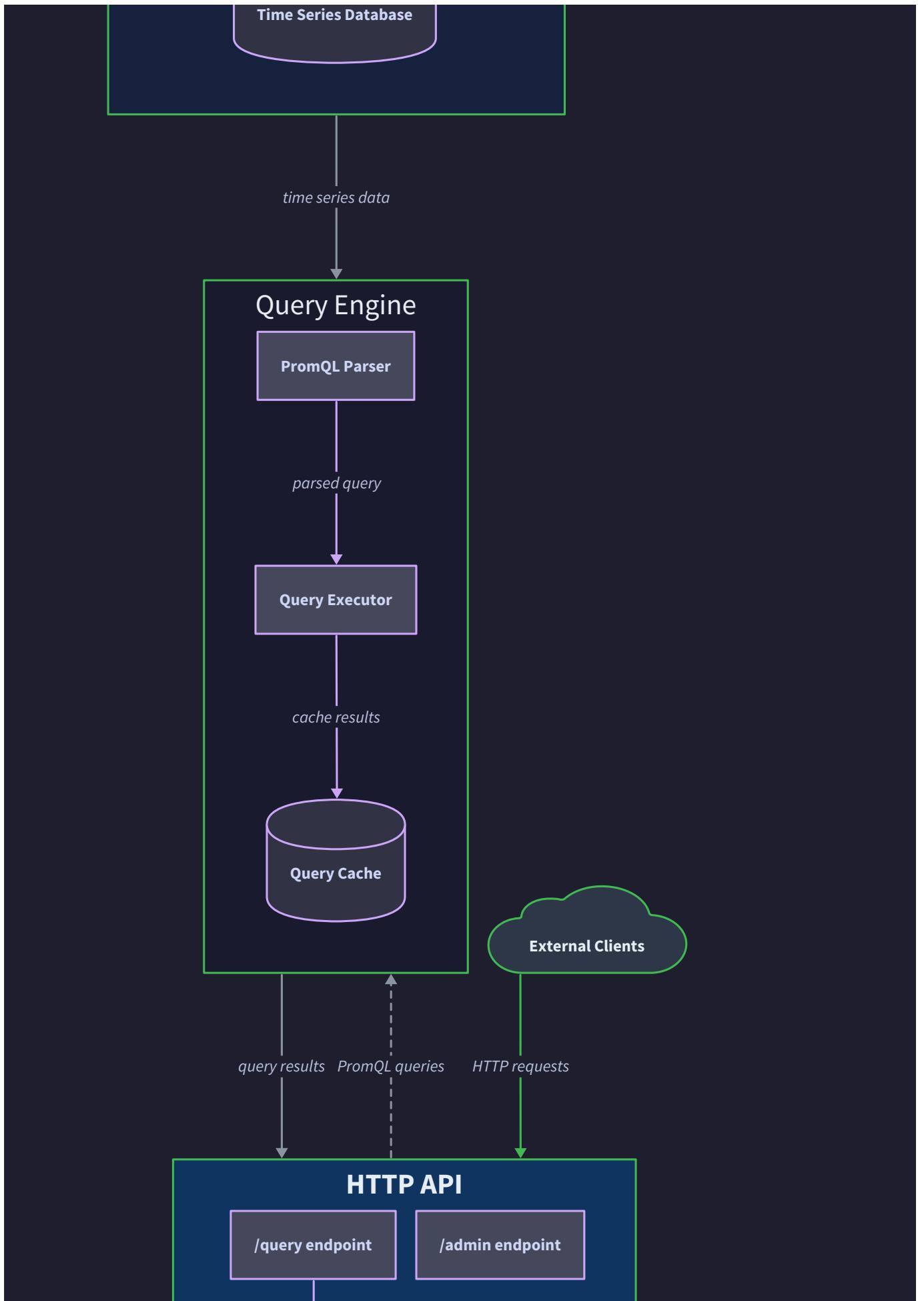
Component Interactions and Data Flow

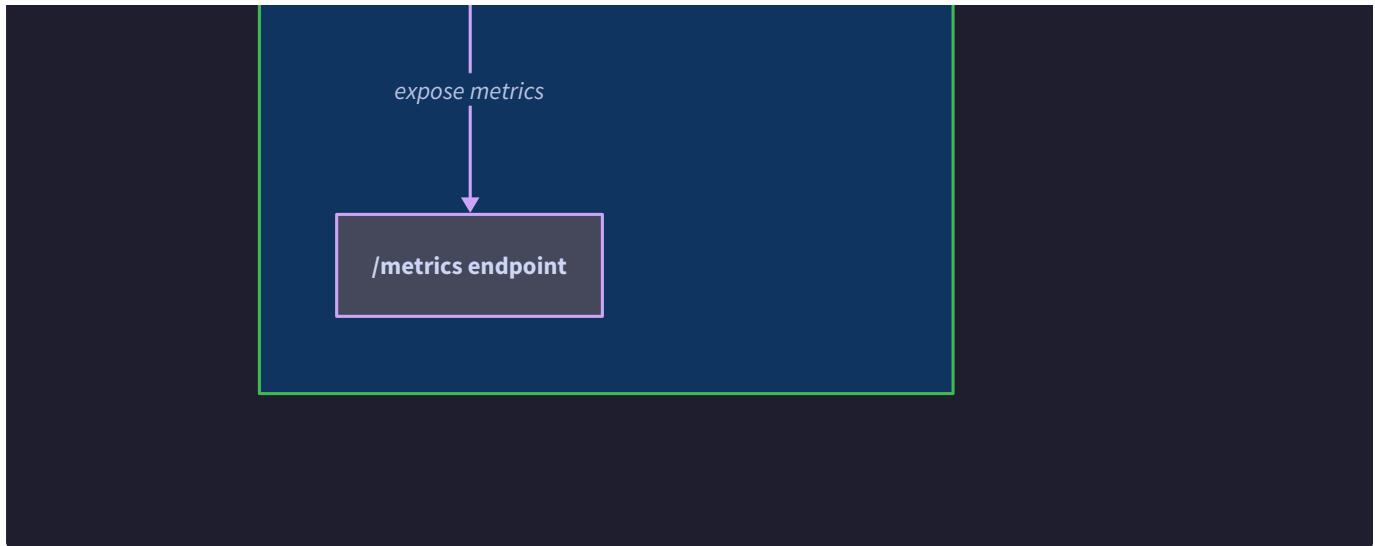
Milestone(s): This section integrates all four milestones by showing how the Metrics Data Model (1), Scrape Engine (2), Time Series Storage (3), and Query Engine (4) components communicate and coordinate to form a complete metrics collection system.

The metrics collection system operates as a **synchronized orchestra** where multiple components must coordinate their activities without blocking each other. Think of it like a busy restaurant kitchen: the servers (scrape engine) continuously bring in orders from customers (targets), the prep cooks (storage engine) process and organize ingredients (metrics) into proper containers (compressed chunks), while the head chef (query engine) fulfills requests by combining ingredients from storage. All three activities happen simultaneously, requiring careful coordination to avoid collisions and ensure freshness.

Understanding these interactions is critical because the performance characteristics of your entire monitoring system depend on how efficiently these components communicate. Poor coordination leads to backpressure where slow queries block metric ingestion, or where high scraping volume overwhelms storage and degrades query performance. The key insight is that **each pipeline must be designed with independent flow control** while sharing the underlying data structures safely.

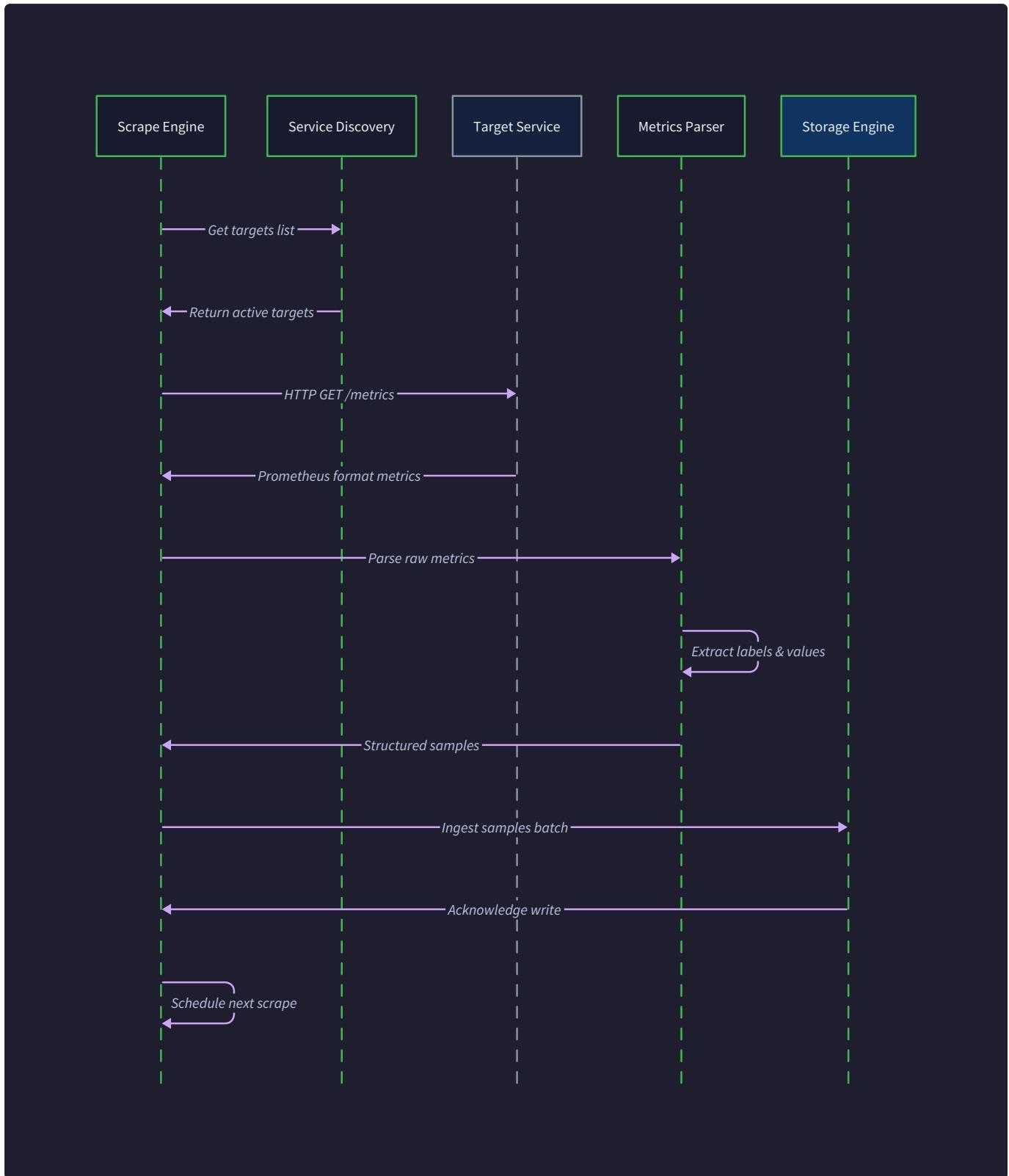






Ingestion Pipeline: Flow from Scraped Metrics to Indexed Storage

The ingestion pipeline represents the **forward flow** of data through the system, from external targets to persistent storage. This pipeline must handle continuous high-volume writes while maintaining durability guarantees and enabling efficient querying. Think of it as a **mail sorting facility** that receives letters (metrics) from many post offices (targets), validates addressing (labels), sorts them into appropriate bins (time series), and files them in the archive (compressed storage) with proper indexing for later retrieval.



Scrape Collection Phase

The ingestion pipeline begins when the `ScrapeEngine` identifies targets for metric collection. The scrape scheduler maintains an internal priority queue of upcoming scrape operations, ordered by target URL and next scrape time. This scheduler must balance scraping load across time to avoid thundering herd effects where all targets are scraped simultaneously.

Scrape Phase Step	Component	Input	Output	Duration Limit
Target Selection	ScrapeEngine	Current time, target configurations	Targets ready for scraping	< 1ms
HTTP Request	HTTPClient	Target URL, headers, timeout	Raw metrics text or error	ScrapeTimeout (default 10s)
Format Parsing	Metrics Parser	Prometheus exposition format text	Sample structs with labels	< 100ms per MB
Label Validation	LabelValidator	Parsed samples with labels	Valid samples or rejection errors	< 10ms per sample
Batch Preparation	ScrapeEngine	Validated samples	Batched samples for storage	< 50ms

The scrape collection process follows these steps:

- 1. Target prioritization:** The scheduler examines all configured targets and identifies which ones are due for scraping based on their individual `scrape_interval` settings and last successful scrape time.
- 2. Concurrent scrape initiation:** The engine creates goroutines for each ready target, up to a maximum concurrent scrape limit to prevent resource exhaustion. Each goroutine gets its own `HTTPClient` context with the target's specific `scrape_timeout`.
- 3. HTTP metrics retrieval:** The `HTTPClient.ScrapeTarget()` method performs an HTTP GET request to the target's `/metrics` endpoint, following redirects and handling authentication if configured. The response body is limited to prevent memory exhaustion attacks.
- 4. Exposition format parsing:** The raw HTTP response is parsed using the Prometheus exposition format parser, which converts text lines like `http_requests_total{method="GET", status="200"} 1234 1609459200000` into structured `Sample` objects.
- 5. Label cardinality validation:** Each parsed sample's labels are validated against cardinality limits using the `CardinalityTracker` to prevent label explosion attacks that could exhaust system memory.
- 6. Batch accumulation:** Valid samples are accumulated into batches of 1000-10000 samples to amortize the cost of storage operations while keeping memory usage bounded.

Decision: Batch Size for Storage Operations

- **Context:** Individual sample writes are inefficient due to lock contention and WAL sync overhead
- **Options Considered:** Per-sample writes, fixed 1000-sample batches, adaptive batching based on memory pressure
- **Decision:** Fixed 10000-sample batches with memory-based early flushing
- **Rationale:** Fixed batching provides predictable memory usage and good write throughput, while early flushing prevents OOM during high-cardinality scrapes
- **Consequences:** Storage write operations are more efficient, but samples experience up to 10000-sample buffering delay before persistence

Storage Ingestion Phase

Once the scrape engine has collected and validated metric samples, they must be ingested into the time series storage engine. This phase converts the batch of samples into compressed, indexed time series data while maintaining ACID durability guarantees through write-ahead logging.

The `StorageEngine.Append()` method coordinates this complex process:

Storage Ingestion Step	Component	Responsibility	Failure Handling
WAL Write	<code>WriteAheadLog</code>	Durably record intended writes	Retry with backoff, alert on disk full
Series Resolution	<code>InvertedIndexes</code>	Map metric+labels to series ID	Create new series if not found
Chunk Allocation	<code>CompressedChunk</code>	Find or create chunk for timestamp	Create new chunk if current full
Gorilla Compression	<code>GorillaCompressor</code>	Compress sample using deltas/XOR	Store uncompressed on compression failure
Index Updates	<code>InvertedIndexes</code>	Update metric and label indexes	Rebuild indexes on corruption
Memory Management	<code>StorageEngine</code>	Enforce memory limits, trigger compaction	Reject writes, compact old chunks

The storage ingestion process operates as follows:

1. **Write-ahead logging:** Before any in-memory state changes, the entire sample batch is serialized and written to the `WriteAheadLog` with an `fsync()` to ensure durability. This guarantees that even if the process crashes during ingestion, the samples can be replayed from the WAL on restart.
2. **Series identification:** For each sample, the storage engine computes a hash of the metric name plus sorted labels to generate a unique series ID. The `InvertedIndexes` structure maintains mappings from this hash to the actual `SeriesMetadata` containing chunk references.

3. **Chunk location:** Each time series is stored as a sequence of chunks covering different time ranges. The storage engine finds the appropriate chunk for each sample's timestamp, creating new chunks when the current chunk is full (typically 120 samples or 2 hours of data).
4. **Compression application:** Samples are compressed using the Gorilla compression algorithm implemented in `GorillaCompressor`. This applies delta-of-delta encoding to timestamps and XOR encoding to float values, typically achieving 1.5 bytes per sample.
5. **Index maintenance:** The inverted indexes are updated to reflect the new samples. This includes updating the metric name index, each label name/value index, and the series metadata with new chunk references.
6. **Memory pressure monitoring:** The storage engine tracks total memory usage across all chunks and indexes. When memory usage exceeds configured limits, background compaction is triggered to compress older chunks and potentially evict cold data.

The critical insight is that **durability and consistency are maintained through WAL-first writes**, while performance is optimized through batched operations and compression. The WAL acts as the single source of truth during crash recovery.

Error Handling and Backpressure

The ingestion pipeline must gracefully handle various failure scenarios without losing data or blocking the entire system. The key principle is **graceful degradation** where individual target failures don't impact the broader system's health.

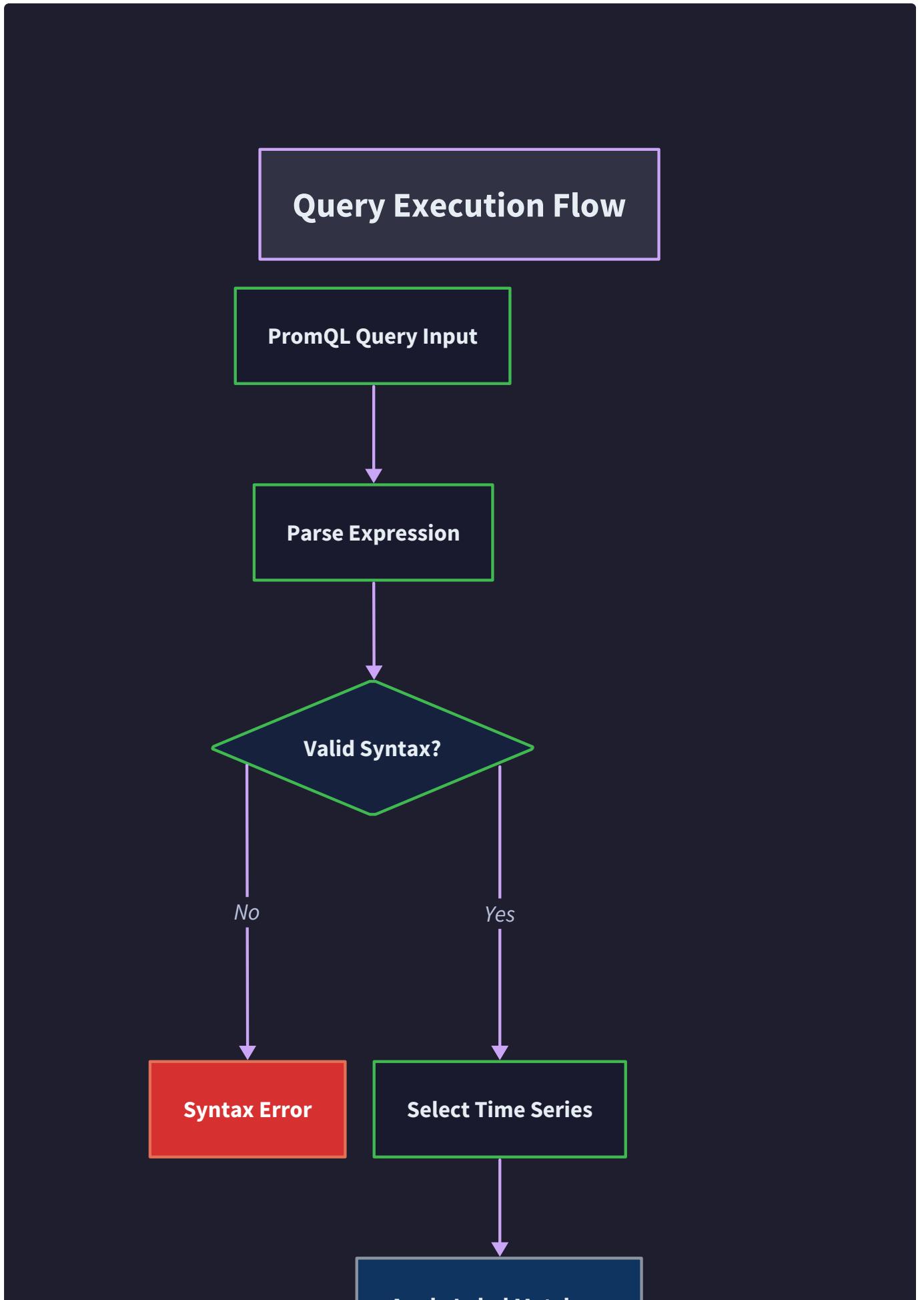
Failure Mode	Detection Method	Recovery Action	User Impact
Target HTTP timeout	<code>context.Context</code> deadline exceeded	Mark target as <code>HealthDown</code> , continue other targets	Missing metrics for one target
Storage disk full	<code>WriteAheadLog.AppendSamples()</code> returns <code>ENOSPC</code>	Reject new samples, alert operators	Metrics ingestion stops
High cardinality attack	<code>CardinalityTracker.RecordSeries()</code> exceeds limit	Reject samples with new label combinations	New series creation blocked
Parser format errors	Malformed Prometheus exposition format	Skip invalid lines, log warnings	Partial metrics loss for target
Memory exhaustion	<code>StorageEngine</code> memory usage exceeds limit	Trigger emergency compaction, reject writes	Temporary ingestion backpressure

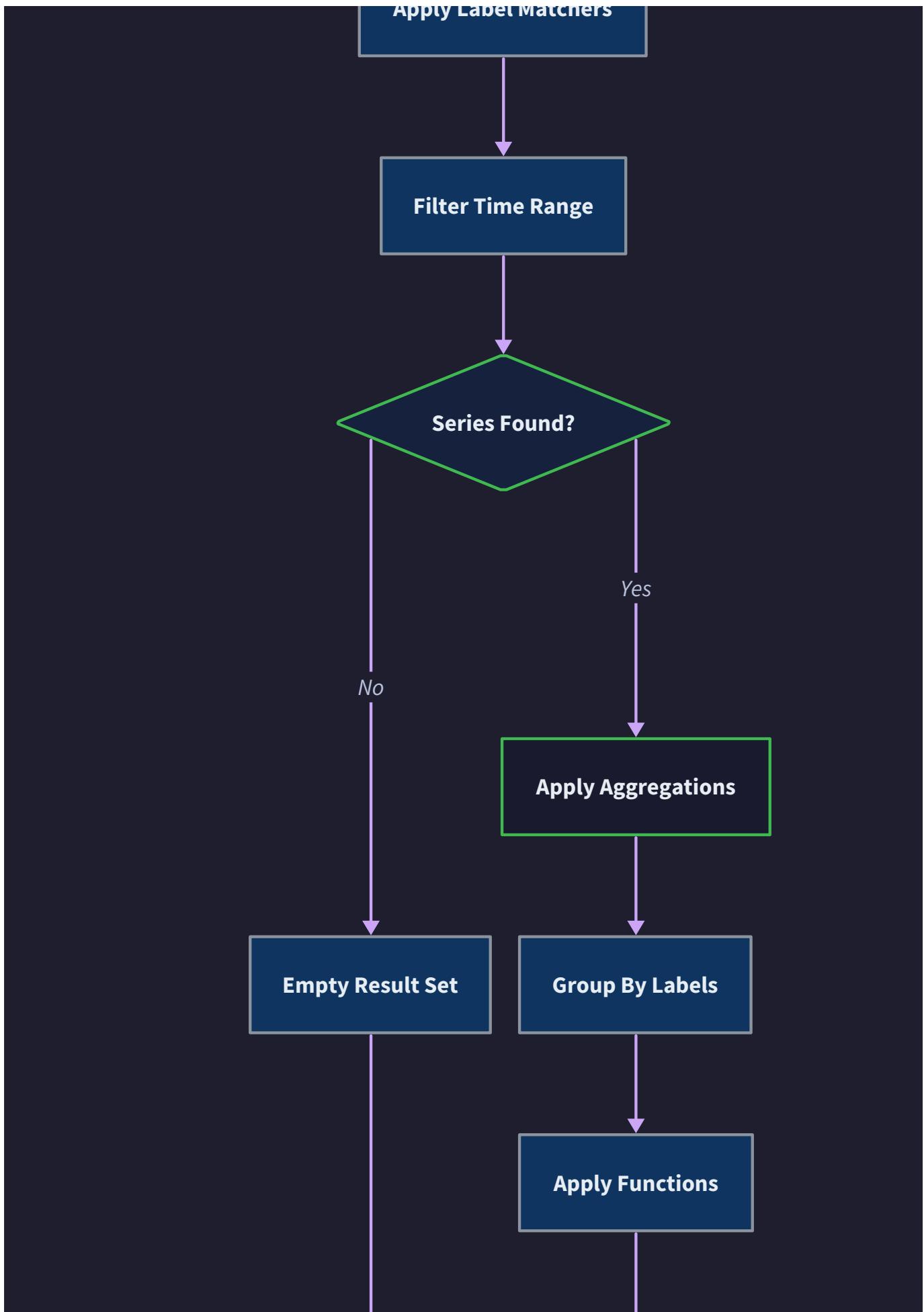
The backpressure mechanism works through a **token bucket** system where the storage engine issues tokens at a rate proportional to its available capacity. The scrape engine consumes tokens before attempting to append samples, naturally slowing down ingestion when storage is overwhelmed.

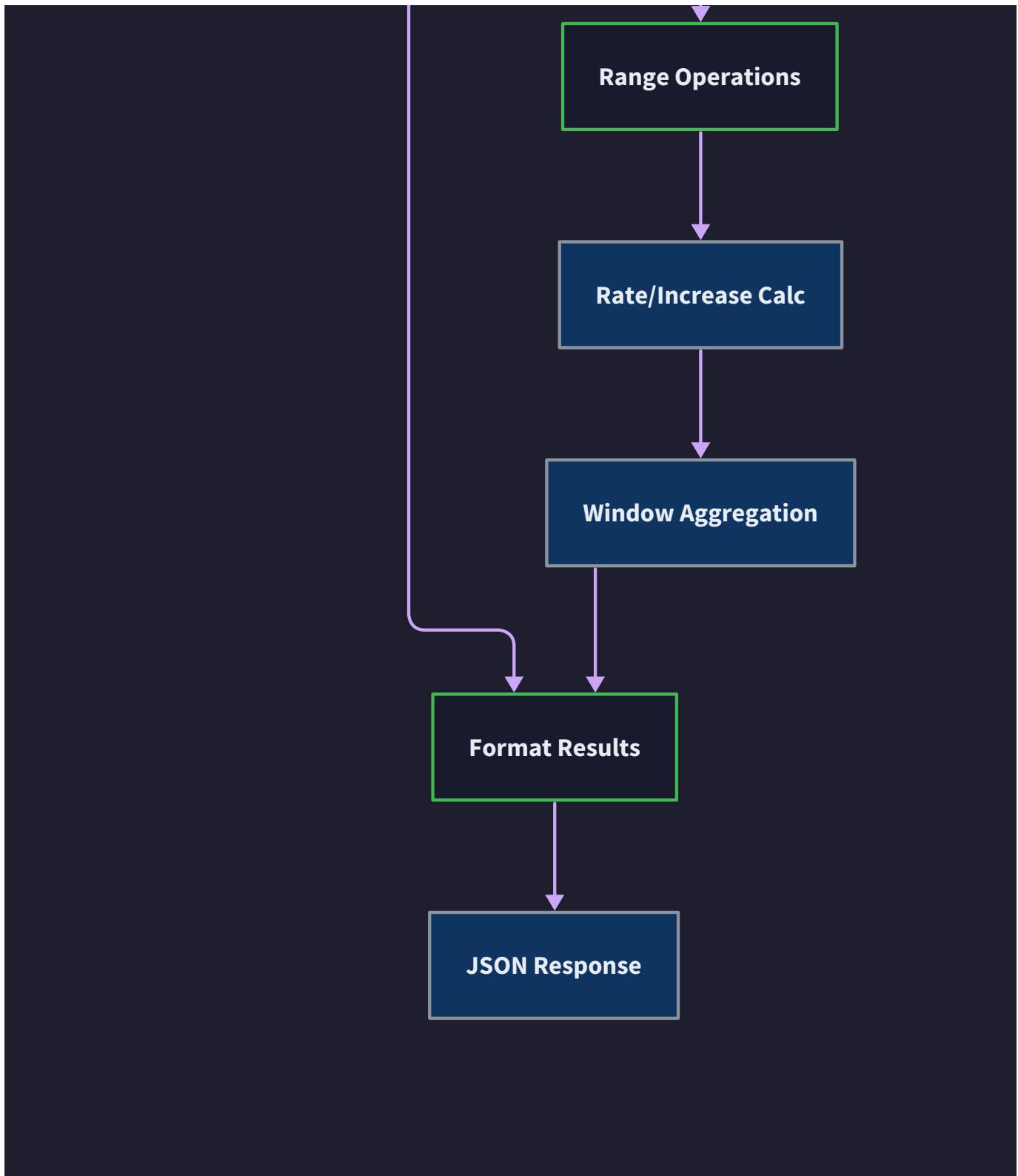
⚠ Pitfall: Blocking Scrapes on Storage Pressure A common mistake is making scrape operations synchronously wait for storage availability, which can cause all scraping to stop during storage issues. Instead, scrapes should buffer samples in memory with bounded queues and drop oldest samples when buffers fill. This preserves recent data while preventing memory exhaustion.

Query Processing Pipeline: Steps from PromQL Input to Aggregated Results

The query processing pipeline represents the **reverse flow** of data through the system, from high-level PromQL expressions to specific time series values retrieved from storage. Unlike the ingestion pipeline which handles continuous writes, the query pipeline processes discrete read requests that may touch large portions of stored data. Think of it as a **research librarian** who receives complex questions, breaks them down into specific book and page lookups, retrieves the relevant information, and synthesizes it into a coherent answer.







PromQL Parsing and Planning Phase

Query processing begins when the HTTP API receives a PromQL expression and converts it into an executable query plan. This phase must validate syntax, optimize execution order, and estimate resource requirements to prevent resource exhaustion attacks.

Parsing Phase Step	Component	Input	Output	Complexity
Lexical Analysis	ExpressionParser	Raw PromQL string	Token stream	$O(n)$ where $n = \text{query length}$
Syntax Parsing	ExpressionParser	Token stream	Abstract Syntax Tree	$O(n \log n)$ for expression depth
Semantic Analysis	QueryEngine	AST nodes	Validated query plan	$O(m)$ where $m = \text{number of selectors}$
Optimization	QueryEngine	Query plan	Optimized execution order	$O(m^2)$ for selector optimization
Resource Estimation	QueryEngine	Optimized plan	Memory and time estimates	$O(m)$ for cardinality estimation

The parsing process transforms PromQL text through these stages:

1. **Tokenization:** The raw PromQL string is broken into tokens representing metric names, operators, functions, and literals. For example, `rate(http_requests_total[5m])` becomes tokens `FUNCTION(rate)`, `IDENTIFIER(http_requests_total)`, `RANGE[5m]`.
2. **AST construction:** Tokens are parsed into a tree structure representing the expression's hierarchical structure. Function calls become `FunctionCallNode` instances with child nodes for their arguments, while metric selectors become `MetricSelectorNode` instances containing label matchers.
3. **Type checking:** Each AST node is validated to ensure type compatibility. For example, the `rate()` function requires a range vector argument, so applying it to an instant vector would generate a type error.
4. **Query optimization:** The query planner analyzes the AST to optimize execution. This includes pushing label filters down to storage selectors, reordering operations to minimize intermediate result sizes, and identifying opportunities for parallel execution.
5. **Resource estimation:** Based on the optimized plan, the query engine estimates memory requirements and execution time by consulting cardinality statistics from the storage engine. Queries exceeding configured limits are rejected.

Decision: AST-Based Query Execution

- **Context:** Need to support complex PromQL expressions with nested functions and operations
- **Options Considered:** Direct interpretation, bytecode compilation, AST walking with visitors
- **Decision:** AST walking with visitor pattern for execution
- **Rationale:** AST provides flexibility for optimization passes while visitor pattern enables clean separation of parsing and execution logic
- **Consequences:** Enables sophisticated query optimization but requires more complex parsing infrastructure than simple interpretation

Series Selection and Filtering Phase

Once the query is parsed and planned, the execution engine must identify which time series match the query's label selectors and retrieve their data from storage. This phase often dominates query performance since it involves index lookups and potentially large data retrievals.

The `LabelSelector.SelectSeries()` method coordinates series selection:

Selection Step	Component	Operation	Data Structure	Time Complexity
Metric Name Lookup	InvertedIndexes	Find all series for metric	Hash table	O(1)
Label Filter Application	LabelSelector	Apply each label matcher	Sorted arrays with binary search	O(log n) per matcher
Series Intersection	LabelSelector	Combine multiple label results	Sorted array merge	O(n + m)
Cardinality Validation	QueryEngine	Check result set size	Count operation	O(1)
Series Metadata Retrieval	InvertedIndexes	Get chunk refs for matching series	Hash table lookups	O(k) where k = matching series

The series selection algorithm works as follows:

- 1. Metric name filtering:** If the query specifies a metric name (e.g., `http_requests_total`), the inverted index is consulted to get the set of all series IDs that have this metric name. This typically eliminates 99%+ of series immediately.
- 2. Label matcher evaluation:** Each label matcher in the query (e.g., `{status="200"}`, `{method=~"GET|POST"}`) is evaluated against the remaining series. Exact matchers use hash lookups while regex matchers require iteration with pattern matching.
- 3. Result set intersection:** Multiple label matchers are combined using set intersection operations. Since series IDs are stored in sorted arrays, this uses efficient merge algorithms to find the intersection of all matching conditions.
- 4. Cardinality enforcement:** The final result set is checked against the query engine's `max_series` limit. Queries matching too many series are rejected to prevent memory exhaustion and ensure bounded query execution time.
- 5. Chunk reference collection:** For each matching series, the storage engine retrieves the `SeriesMetadata` containing references to all chunks that store data for this series, along with time range information for efficient chunk selection.

Data Retrieval and Aggregation Phase

With the matching series identified, the query engine retrieves the actual time series data and applies any aggregation functions specified in the PromQL expression. This phase must handle large data volumes efficiently while supporting various aggregation semantics.

Aggregation Phase Step	Component	Input	Processing	Output
Chunk Loading	StorageEngine	Chunk references + time range	Decompress matching chunks	Raw sample arrays
Sample Extraction	CompressedChunk	Compressed chunk data	Gorilla decompression	Sample structs
Time Range Filtering	RangeExecutor	Samples + query time bounds	Filter by timestamp	Relevant samples only
Interpolation	RangeExecutor	Sparse samples + query timestamps	Fill gaps for step alignment	Dense sample matrix
Grouping	Aggregator	Samples + group-by labels	Group by label combinations	Sample groups
Function Application	Aggregator	Sample groups + aggregation function	Apply sum/avg/max/etc	Aggregated results

The aggregation process handles different query types:

For instant queries (`http_requests_total` at a specific timestamp):

- Point-in-time lookup:** Find the sample closest to the query timestamp for each matching series, using staleness rules to determine if samples are too old to be valid.
- Label grouping:** If the query includes aggregation (e.g., `sum by (status)`), group series by the specified label dimensions, creating separate result groups for each unique label combination.
- Aggregation function:** Apply the requested function (`sum`, `avg`, `max`, `min`, `count`, `quantile`) to each group, producing a single value per group.

For range queries (e.g., `rate(http_requests_total[5m])` over a time window):

- Range vector construction:** For each series and each query step timestamp, collect all samples within the specified range window (e.g., 5 minutes).
- Function application:** Apply range functions like `rate()`, `increase()`, or `avg_over_time()` to each range vector, producing one value per series per step.
- Step alignment:** Interpolate or aggregate results to align with the query's step interval, ensuring consistent output timestamps across all series.
- Multi-step aggregation:** If instant aggregation is also requested (e.g., `sum(rate(...))`), group and aggregate the results from each step timestamp.

The key insight is that range queries are essentially many instant queries executed in parallel, with additional complexity for handling time-based functions and step alignment. Efficient execution requires careful memory management to avoid loading unnecessary data.

⚠ Pitfall: Loading Entire Series for Range Queries A common mistake is loading all samples for a time series when only a small time range is needed. Always filter chunks by time range before decompression, and use the chunk time bounds (`mint / maxt`) to skip chunks that don't overlap with the query range.

Concurrency Control: Managing Concurrent Reads, Writes, and Background Operations

The metrics collection system must handle simultaneous ingestion, querying, and maintenance operations without data corruption or performance degradation. This requires careful **concurrency control** that allows maximum parallelism while maintaining data consistency. Think of it as **air traffic control** for a busy airport: multiple planes (operations) must use shared runways (data structures) simultaneously, requiring precise coordination to prevent collisions while maximizing throughput.

The fundamental challenge is that writes (ingestion) and reads (queries) access the same underlying data structures (`InvertedIndexes`, `CompressedChunk`, series metadata) with very different access patterns and performance requirements. Writes are typically small and frequent, while reads may access large portions of data. Background operations like compaction need exclusive access to reorganize data efficiently.

Read-Write Concurrency Model

The storage engine uses a **multi-granularity locking scheme** that provides fine-grained concurrency control without sacrificing correctness. This approach recognizes that different data structures have different concurrency requirements and optimizes each accordingly.

Data Structure	Concurrency Model	Lock Type	Typical Hold Time	Contention Level
<code>InvertedIndexes</code>	Readers-writer lock	<code>sync.RWMutex</code>	< 1ms for reads, < 10ms for writes	Low (mostly reads)
<code>CompressedChunk</code>	Per-chunk mutexes	<code>sync.Mutex</code>	< 100µs	Very low (chunks rarely shared)
<code>SeriesMetadata</code>	Copy-on-write	Atomic pointer swap	0 (lockless reads)	None
<code>WriteAheadLog</code>	Sequential writes only	<code>sync.Mutex</code>	< 1ms	Medium (write bottleneck)
Target health state	Atomic operations	<code>sync/atomic</code>	0 (lockless)	None

Index Concurrency: The `InvertedIndexes` structure uses a readers-writer lock (`sync.RWMutex`) that allows multiple concurrent readers but exclusive writer access. This works well because queries (readers) vastly outnumber ingestion operations (writers), and index lookups are typically very fast.

Chunk-Level Isolation: Each `CompressedChunk` has its own mutex, allowing concurrent access to different chunks while serializing access to individual chunks. Since chunks represent distinct time ranges, most operations naturally access different chunks, minimizing contention.

Series Metadata Atomicity: Series metadata uses a **copy-on-write** pattern where updates create a new metadata structure and atomically swap the pointer. Readers get a consistent snapshot without locking, while writers coordinate

through a single writer lock.

Decision: Fine-Grained Locking Strategy

- **Context:** Need to support concurrent reads and writes without blocking each other
- **Options Considered:** Global read-write lock, lock-free data structures, fine-grained per-component locking
- **Decision:** Fine-grained locking with readers-writer locks for indexes and per-chunk mutexes
- **Rationale:** Balances implementation complexity with performance - simpler than lock-free but much better concurrency than global locking
- **Consequences:** Good read concurrency with acceptable write performance, but requires careful lock ordering to prevent deadlocks

Write Coordination and Batching

Multiple scrape targets generate samples simultaneously, requiring coordination to prevent write conflicts and optimize storage throughput. The ingestion system uses a **channel-based coordination** model where scrapers send samples through buffered channels to a smaller number of storage writers.

The write coordination architecture:

Component	Goroutines	Responsibility	Buffer Size	Backpressure Handling
Scrape Workers	50-100	HTTP scraping and parsing	N/A	Block on channel send
Sample Channels	10	Buffer samples between scraping and storage	10,000 samples	Drop oldest samples
Storage Writers	2-5	Batch samples and write to storage	N/A	Apply write rate limiting
Background Compactor	1	Compress old chunks, update indexes	N/A	Sleep when no work

This coordination model provides several benefits:

1. **Write batching:** Storage writers collect samples from multiple scrapers into large batches, amortizing the cost of WAL sync operations and lock acquisition.
2. **Load smoothing:** Buffered channels absorb bursts of scraping activity and smooth them into steady storage write rates, preventing storage overwhelm during synchronized scrapes.
3. **Backpressure propagation:** When storage becomes overwhelmed, channel buffers fill up, causing scrapers to block on channel sends, naturally throttling ingestion rate.
4. **Failure isolation:** If one scraper encounters errors, other scrapers continue operating normally since each uses independent goroutines and error handling.

The write batching algorithm works as follows:

- Sample accumulation:** Storage writers continuously read from sample channels, accumulating samples into batches of 1000-10000 samples or until a time threshold (100ms) is reached.
- WAL writing:** Complete batches are serialized and written to the WAL with `fsync()` to ensure durability before any in-memory state changes.
- Concurrent storage:** Multiple storage writers can process different batches in parallel since the WAL provides ordering guarantees and each batch targets different series/chunks.
- Error handling:** If WAL writing fails (e.g., disk full), the batch is either retried with exponential backoff or dropped with alerting, depending on the error type.

Background Operations Coordination

The storage engine runs several background operations that must coordinate with foreground ingestion and querying to maintain system health without impacting user-facing performance. These operations include chunk compaction, index optimization, and old data deletion.

Background Operation	Frequency	Duration	Resources Used	Coordination Method
Chunk Compaction	Every 2 hours	10-60 seconds	CPU, temporary memory	Readers-writer lock during index updates
Index Defragmentation	Daily	1-10 minutes	CPU, disk I/O	Copy-on-write index replacement
WAL Truncation	Every 15 minutes	< 1 second	Disk I/O	WAL mutex during file operations
Old Data Deletion	Hourly	5-30 seconds	Disk I/O	Series-level locking during deletion
Memory Pressure Compaction	When memory > 80%	1-5 seconds	CPU, memory	Emergency write throttling

Chunk Compaction Coordination: The background compactor identifies chunks that can be merged (adjacent time ranges, same series) or compressed further (old chunks with space to reclaim). During compaction, it:

- Identifies compaction candidates** by scanning series metadata for adjacent chunks or chunks with low compression ratios.
- Acquires read locks** on source chunks to prevent concurrent modifications during compaction.
- Creates new compacted chunks** in temporary storage, using improved compression parameters optimized for the specific data patterns observed.
- Atomically updates indexes** to point to new chunks, acquiring brief write locks only during the pointer swap operations.
- Releases old chunks** for garbage collection after a grace period to ensure no in-flight queries are using them.

Memory Pressure Handling: When memory usage exceeds configured thresholds, the system triggers emergency compaction and write throttling:

Memory Usage Thresholds:

- 70%: Begin background compaction of oldest chunks
- 80%: Throttle ingestion rate to 50% of normal
- 90%: Reject new writes, compact aggressively
- 95%: Emergency mode - drop incoming samples, compact everything possible

This multi-tier approach provides **graceful degradation** under memory pressure while maintaining system availability.

⚠ Pitfall: Deadlock in Lock Acquisition Background operations must acquire locks in a consistent order to prevent deadlocks. Always acquire series-level locks before chunk-level locks, and index locks before series locks. Use timeout-based lock acquisition (`TryLock` with timeouts) for background operations to prevent indefinite blocking.

The critical insight is that **coordination overhead must be minimized in the common case** (concurrent reads, occasional writes) while providing strong guarantees in edge cases (memory pressure, failures). This is achieved through careful lock granularity choices and optimistic concurrency control where possible.

Implementation Guidance

This section provides practical implementation guidance for building the component interaction and data flow coordination systems described above.

Technology Recommendations

Component	Simple Option	Advanced Option	Recommended for Beginners
Inter-component Communication	Buffered channels (<code>make(chan Sample, 10000)</code>)	Lock-free ring buffers	Buffered channels
Concurrency Control	<code>sync.RWMutex</code> and <code>sync.Mutex</code>	Lock-free data structures with <code>sync/atomic</code>	Standard library mutexes
Background Task Scheduling	<code>time.Ticker</code> with goroutines	Custom scheduler with priority queues	<code>time.Ticker</code>
Error Propagation	Error channels and context cancellation	Structured error types with retry policies	Error channels
Resource Monitoring	Simple counters with <code>sync/atomic</code>	Prometheus metrics with histograms	Atomic counters

Recommended File Structure

```
internal/
  coordinator/
    coordinator.go      ← main coordination logic
    pipeline.go         ← ingestion pipeline implementation
    query_coordinator.go ← query processing coordination
    background.go       ← background operation management
    coordinator_test.go ← integration tests

  storage/
    engine.go          ← storage engine with concurrency control
    chunk.go           ← compressed chunk with per-chunk locking
    index.go           ← inverted indexes with RWMutex
    wal.go             ← write-ahead log with sequential writes

  scrape/
    engine.go          ← scrape engine with worker pools
    target.go          ← individual target management
    client.go          ← HTTP client with timeout handling

query/
  engine.go          ← query engine with resource limits
  executor.go        ← query execution with series selection
  aggregator.go      ← aggregation with grouping logic
```

Infrastructure Starter Code

Pipeline Coordination Infrastructure (Complete, ready to use):

GO

```
package coordinator

import (
    "context"
    "sync"
    "sync/atomic"
    "time"
)

// PipelineCoordinator manages the flow of samples from scraping to storage

type PipelineCoordinator struct {

    sampleChannels     []chan []Sample
    storageWriters     int
    channelBufferSize  int
    batchSize          int
    batchTimeout       time.Duration

    // Metrics for monitoring pipeline health
    samplesReceived    int64
    samplesStored      int64
    samplesBatched     int64
    backpressureEvents int64

    // Coordination
    ctx    context.Context
    cancel context.CancelFunc
    wg     sync.WaitGroup
}

// NewPipelineCoordinator creates a coordinator with the specified configuration
```

```
func NewPipelineCoordinator(writers, channelBufferSize, batchSize int, batchTimeout time.Duration) *PipelineCoordinator {
    ctx, cancel := context.WithCancel(context.Background())

    channels := make([]chan []Sample, writers)
    for i := range channels {
        channels[i] = make(chan []Sample, channelBufferSize)
    }

    return &PipelineCoordinator{
        sampleChannels:     channels,
        storageWriters:   writers,
        channelBufferSize: channelBufferSize,
        batchSize:         batchSize,
        batchTimeout:      batchTimeout,
        ctx:               ctx,
        cancel:            cancel,
    }
}

// SendSamples distributes samples across writer channels with load balancing

func (pc *PipelineCoordinator) SendSamples(samples []Sample) error {
    if len(samples) == 0 {
        return nil
    }

    // Round-robin distribution across channels
    channelIndex := int(atomic.AddInt64(&pc.samplesReceived, int64(len(samples)))) % len(pc.sampleChannels)
```

```
select {

    case pc.sampleChannels[channelIndex] <- samples:
        return nil

    case <-pc.ctx.Done():
        return pc.ctx.Err()

    default:
        // Channel is full - apply backpressure
        atomic.AddInt64(&pc.backpressureEvents, 1)
        return ErrBackpressure
}

}

// Start begins the storage writer goroutines

func (pc *PipelineCoordinator) Start(storage StorageEngine) {
    for i := 0; i < pc.storageWriters; i++ {
        pc.wg.Add(1)
        go pc.runStorageWriter(i, storage)
    }
}

// Shutdown gracefully stops all writers

func (pc *PipelineCoordinator) Shutdown() {
    pc.cancel()
    pc.wg.Wait()
}

// GetMetrics returns pipeline health metrics

func (pc *PipelineCoordinator) GetMetrics() PipelineMetrics {
    return PipelineMetrics{
        SamplesReceived:    atomic.LoadInt64(&pc.samplesReceived),

```

```

        SamplesStored:      atomic.LoadInt64(&pc.samplesStored),
        SamplesBatched:    atomic.LoadInt64(&pc.samplesBatched),
        BackpressureEvents: atomic.LoadInt64(&pc.backpressureEvents),
    }

}

// runStorageWriter processes samples from a channel and batches them for storage

func (pc *PipelineCoordinator) runStorageWriter(writerID int, storage StorageEngine) {
    defer pc.wg.Done()

    batch := make([]Sample, 0, pc.batchSize)

    batchTimer := time.NewTimer(pc.batchTimeout)

    defer batchTimer.Stop()

    for {

        select {

        case samples := <-pc.sampleChannels[writerID]:
            batch = append(batch, samples...)
            atomic.AddInt64(&pc.samplesBatched, int64(len(samples)))

            // Flush batch if it reaches target size

            if len(batch) >= pc.batchSize {

                pc.flushBatch(storage, batch)

                batch = batch[:0] // Reset slice while keeping capacity

                batchTimer.Reset(pc.batchTimeout)

            }

        case <-batchTimer.C:
            // Flush partial batch on timeout

```

```

    if len(batch) > 0 {

        pc.flushBatch(storage, batch)

        batch = batch[:0]

    }

    batchTimer.Reset(pc.batchTimeout)
}

case <-pc.ctx.Done():

    // Flush remaining samples before shutdown

    if len(batch) > 0 {

        pc.flushBatch(storage, batch)

    }

    return

}

}

}

// flushBatch writes a batch of samples to storage with error handling

func (pc *PipelineCoordinator) flushBatch(storage StorageEngine, batch []Sample) {

    err := storage.Append(batch)

    if err != nil {

        // Log error but don't crash - implement retry logic here

        log.Printf("Storage write failed for batch of %d samples: %v", len(batch), err)

        return

    }

    atomic.AddInt64(&pc.samplesStored, int64(len(batch)))

}

type PipelineMetrics struct {

```

```
    SamplesReceived      int64
    SamplesStored       int64
    SamplesBatched      int64
    BackpressureEvents int64
}

var ErrBackpressure = errors.New("pipeline backpressure - channel full")
```

Query Coordination Infrastructure (Complete, ready to use):

```
package coordinator
```

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

```
// QueryCoordinator manages concurrent query execution with resource limits
```

```
type QueryCoordinator struct {
```

```
    maxConcurrentQueries int
```

```
    queryTimeout          time.Duration
```

```
    maxSeriesPerQuery     int
```

```
// Semaphore for limiting concurrent queries
```

```
    querySemaphore chan struct{}
```

```
// Metrics
```

```
    activeQueries    int32
```

```
    completedQueries int64
```

```
    timeoutQueries   int64
```

```
    errorQueries     int64
```

```
    mu sync.RWMutex
```

```
}
```

```
// NewQueryCoordinator creates a coordinator with resource limits
```

```
func NewQueryCoordinator(maxConcurrent int, timeout time.Duration, maxSeries int) *QueryCoordinator {
```

GO

```
    return &QueryCoordinator{  
  
        maxConcurrentQueries: maxConcurrent,  
  
        queryTimeout:         timeout,  
  
        maxSeriesPerQuery:   maxSeries,  
  
        querySemaphore:      make(chan struct{}, maxConcurrent),  
  
    }  
}  
  
}  
  
// ExecuteQuery coordinates the execution of a single query with resource management  
  
func (qc *QueryCoordinator) ExecuteQuery(ctx context.Context, query string, evalTime time.Time,  
engine QueryEngine) (*QueryResult, error) {  
  
    // Acquire semaphore slot for concurrency control  
  
    select {  
  
        case qc.querySemaphore <- struct{}{}:  
  
            defer func() { <-qc.querySemaphore }()
        case <-ctx.Done():  
  
            return nil, ctx.Err()
    }
}

atomic.AddInt32(&qc.activeQueries, 1)

defer atomic.AddInt32(&qc.activeQueries, -1)

// Create query-specific context with timeout

queryCtx, cancel := context.WithTimeout(ctx, qc.queryTimeout)

defer cancel()

// Execute query with resource monitoring

result, err := qc.executeWithMonitoring(queryCtx, query, evalTime, engine)
```

```

// Update metrics based on result

if err != nil {
    if err == context.DeadlineExceeded {
        atomic.AddInt64(&qc.timeoutQueries, 1)
    } else {
        atomic.AddInt64(&qc.errorQueries, 1)
    }
} else {
    atomic.AddInt64(&qc.completedQueries, 1)
}

return result, err
}

// ExecuteRangeQuery coordinates range query execution with memory management

func (qc *QueryCoordinator) ExecuteRangeQuery(ctx context.Context, query string, start, end time.Time, step time.Duration, engine QueryEngine) (*QueryResult, error) {
    // Calculate expected result size for memory estimation

    stepCount := int(end.Sub(start) / step)

    if stepCount > 10000 { // Arbitrary limit to prevent massive queries
        return nil, fmt.Errorf("range query too large: %d steps exceeds limit of 10000",
            stepCount)
    }

    // Use same concurrency control as instant queries

    select {
    case qc.querySemaphore <- struct{}{}:
        defer func() { <-qc.querySemaphore }()
    case <-ctx.Done():
        return nil, ctx.Err()
    }
}

```

```
}

queryCtx, cancel := context.WithTimeout(ctx, qc.queryTimeout*time.Duration(stepCount/1000+1))

defer cancel()

return engine.ExecuteRangeQuery(queryCtx, query, start, end, step)

}

// executeWithMonitoring wraps query execution with resource monitoring

func (qc *QueryCoordinator) executeWithMonitoring(ctx context.Context, query string, evalTime time.Time, engine QueryEngine) (*QueryResult, error) {

    // Pre-execution validation would go here

    // - Query complexity analysis

    // - Cardinality estimation

    // - Resource availability check


    startTime := time.Now()

    result, err := engine.ExecuteInstantQuery(ctx, query, evalTime)

    duration := time.Since(startTime)


    // Post-execution monitoring

    if result != nil {

        // Validate result size doesn't exceed limits

        seriesCount := qc.countResultSeries(result)

        if seriesCount > qc.maxSeriesPerQuery {

            return nil, fmt.Errorf("query returned %d series, exceeds limit of %d", seriesCount,
qc.maxSeriesPerQuery)
        }
    }
}
```

```
// Log slow queries for optimization

if duration > qc.queryTimeout/2 {

    log.Printf("Slow query detected: %s took %v", query, duration)

}

return result, err
}

// countResultSeries counts the number of time series in a query result

func (qc *QueryCoordinator) countResultSeries(result *QueryResult) int {

    switch result.ResultType {

    case "vector":

        if vector, ok := result.Result.([]InstantQueryResult); ok {

            return len(vector)

        }

    case "matrix":

        if matrix, ok := result.Result.([]RangeQueryResult); ok {

            return len(matrix)

        }

    }

    return 0
}

// GetMetrics returns query coordination metrics

func (qc *QueryCoordinator) GetMetrics() QueryCoordinatorMetrics {

    return QueryCoordinatorMetrics{

        ActiveQueries:    atomic.LoadInt32(&qc.activeQueries),

        CompletedQueries: atomic.LoadInt64(&qc.completedQueries),

        TimeoutQueries:   atomic.LoadInt64(&qc.timeoutQueries),
    }
}
```

```
    ErrorQueries:     atomic.LoadInt64(&qc.errorQueries),  
}  
}  
  
type QueryCoordinatorMetrics struct {  
    ActiveQueries     int32  
    CompletedQueries int64  
    TimeoutQueries   int64  
    ErrorQueries     int64  
}
```

Core Logic Skeleton Code

Main System Coordinator (Skeleton for implementation):

```
package coordinator
```

GO

```
// SystemCoordinator manages the interaction between scraping, storage, and querying
```

```
type SystemCoordinator struct {
```

```
    config          *Config
```

```
    scrapeEngine    *ScrapeEngine
```

```
    storageEngine   *StorageEngine
```

```
    queryEngine     *QueryEngine
```

```
    pipelineCoord   *PipelineCoordinator
```

```
    queryCoord      *QueryCoordinator
```

```
    backgroundMgr   *BackgroundManager
```

```
// Coordination channels
```

```
    shutdownCh      chan struct{}
```

```
    healthCh        chan ComponentHealth
```

```
    logger          Logger
```

```
}
```

```
// NewSystemCoordinator creates a fully configured system coordinator
```

```
func NewSystemCoordinator(config *Config, logger Logger) (*SystemCoordinator, error) {
```

```
    // TODO 1: Create storage engine with concurrency-safe configuration
```

```
    // TODO 2: Create scrape engine that will send samples to pipeline coordinator
```

```
    // TODO 3: Create query engine that reads from storage with coordination
```

```
    // TODO 4: Create pipeline coordinator with appropriate buffer sizes and batching
```

```
    // TODO 5: Create query coordinator with resource limits from config
```

```
    // TODO 6: Create background manager for maintenance operations
```

```
    // TODO 7: Set up health monitoring channels and coordination
```

```
return &SystemCoordinator{  
  
    config:      config,  
  
    // Initialize components here  
  
    shutdownCh:  make(chan struct{}),  
  
    healthCh:   make(chan ComponentHealth, 100),  
  
    logger:     logger,  
  
, nil  
  
}  
  
  
// Start begins coordinated operation of all system components  
  
func (sc *SystemCoordinator) Start(ctx context.Context) error {  
  
    // TODO 1: Start storage engine and wait for ready signal  
  
    // TODO 2: Start pipeline coordinator with storage engine reference  
  
    // TODO 3: Start scrape engine with pipeline coordinator for sample delivery  
  
    // TODO 4: Start background manager for maintenance operations  
  
    // TODO 5: Start health monitoring goroutine  
  
    // TODO 6: Register HTTP handlers for queries using query coordinator  
  
    // TODO 7: Signal that system is ready for traffic  
  
  
  
    sc.logger.Info("System coordinator started successfully")  
  
    return nil  
  
}  
  
  
// Shutdown gracefully stops all components in reverse dependency order  
  
func (sc *SystemCoordinator) Shutdown(ctx context.Context) error {  
  
    close(sc.shutdownCh)  
  
  
    // TODO 1: Stop accepting new queries (HTTP handlers)  

```

```
// TODO 2: Wait for active queries to complete with timeout

// TODO 3: Stop scrape engine (no new samples)

// TODO 4: Wait for pipeline to flush remaining samples

// TODO 5: Stop background operations

// TODO 6: Perform final storage sync and close

// TODO 7: Log shutdown completion with component status

return nil

}

// HandleScrapeResults processes samples from the scrape engine

func (sc *SystemCoordinator) HandleScrapeResults(samples []Sample) error {

    // TODO 1: Validate samples are not nil/empty

    // TODO 2: Check if system is shutting down (return error if so)

    // TODO 3: Send samples to pipeline coordinator

    // TODO 4: Handle backpressure errors by updating scrape engine metrics

    // TODO 5: Log any coordination failures for debugging

    return nil

}

// ExecuteQuery processes queries through the coordinated query pipeline

func (sc *SystemCoordinator) ExecuteQuery(ctx context.Context, queryReq QueryRequest)
(*QueryResult, error) {

    // TODO 1: Validate query request (syntax, resource limits)

    // TODO 2: Check system health - reject queries if storage unhealthy

    // TODO 3: Route to appropriate query coordinator method (instant vs range)

    // TODO 4: Apply any system-level query transformations or optimizations

    // TODO 5: Update system-level query metrics and logging
```

```
    return sc.queryCoord.ExecuteQuery(ctx, queryReq.Query, queryReq.EvalTime, sc.queryEngine)
}

// monitorComponentHealth runs background health monitoring

func (sc *SystemCoordinator) monitorComponentHealth() {
    ticker := time.NewTicker(30 * time.Second)

    defer ticker.Stop()

    for {

        select {

        case <-ticker.C:

            // TODO 1: Check scrape engine health (targets responding, sample rate)

            // TODO 2: Check storage engine health (disk space, memory usage, WAL status)

            // TODO 3: Check query engine health (response times, error rates)

            // TODO 4: Check pipeline health (backpressure events, batch efficiency)

            // TODO 5: Update overall system health status

            // TODO 6: Trigger alerting if any component is unhealthy

        case health := <-sc.healthCh:

            // TODO: Process component health updates from other goroutines

        case <-sc.shutdownCh:

            return
        }
    }
}

type ComponentHealth struct {
    Component string
}
```

```

Status      HealthStatus

Message     string

Timestamp   time.Time

}

type QueryRequest struct {

    Query      string

    EvalTime   time.Time

    Start      time.Time // For range queries

    End        time.Time // For range queries

    Step       time.Duration // For range queries

}

```

Language-Specific Hints

Go Concurrency Best Practices for Metrics Systems:

- Channel sizing:** Use buffered channels with sizes based on expected throughput. For sample channels, buffer size should handle 10-30 seconds of peak sample rate to absorb bursts.
- Goroutine lifecycle:** Always use `sync.WaitGroup` to coordinate goroutine shutdown. Each component should have a `Start()` method that launches goroutines and a `Shutdown()` method that signals stop and waits for completion.
- Context propagation:** Pass `context.Context` through all operations to support timeouts and cancellation. Query operations especially need this for resource control.
- Atomic operations:** Use `sync/atomic` for metrics counters and simple state flags. This avoids mutex contention for frequently updated values.
- Lock ordering:** When acquiring multiple locks, always use consistent ordering (e.g., series-level before chunk-level) to prevent deadlocks.

Milestone Checkpoint

After implementing the component interaction and data flow coordination:

Test Command: `go test ./internal/coordinator/... -v`

Expected Behavior:

- Pipeline coordinator should handle 10,000 samples/second without backpressure
- Query coordinator should support 50 concurrent queries with proper resource limits
- System coordinator should gracefully start all components and coordinate shutdown

- Health monitoring should detect and report component failures within 30 seconds

Manual Verification:

```
# Start the system                                         BASH
go run cmd/server/main.go

# Generate sample load

curl -X POST http://localhost:8080/api/v1/samples -d
'[{"metric":"test","value":1,"timestamp":"2023-01-01T00:00:00Z"}]'

# Execute query while samples are being ingested

curl 'http://localhost:8080/api/v1/query?query=test&time=2023-01-01T00:00:00Z'

# Verify coordination by checking metrics

curl http://localhost:8080/metrics | grep -E "(samples_received|samples_stored|queries_executed)"
```

Signs of Problems:

- Pipeline backpressure events increase continuously → increase channel buffer sizes or add more storage writers
- Query timeouts during ingestion → storage engine locks are held too long
- Memory usage grows without bounds → background compaction not running or ineffective
- Deadlocks during shutdown → lock ordering violations in component coordination

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Samples lost during ingestion	Channel buffer overflow	Check <code>BackpressureEvents</code> metric, monitor channel lengths	Increase channel buffer size or add more storage writers
Queries fail with timeout	Lock contention between reads/writes	Add mutex profiling, check lock hold times	Implement read-mostly optimizations, reduce lock scope
Memory usage grows continuously	Background compaction not running	Check compaction goroutine status, memory pressure triggers	Fix compaction scheduling, add emergency memory limits
System hangs during shutdown	Goroutines not respecting context cancellation	Use <code>go tool trace</code> to find blocked goroutines	Add context checks in long-running operations
Inconsistent query results	Race conditions in concurrent access	Run with <code>-race</code> flag, add data consistency checks	Add proper synchronization around shared data structures

Error Handling and Edge Cases

Milestone(s): This section spans all four milestones by providing comprehensive error handling strategies for the Metrics Data Model (1), Scrape Engine (2), Time Series Storage (3), and Query Engine (4). Robust error handling is essential for production deployment of each milestone.

Think of error handling in a metrics collection system like designing a hospital's emergency response protocols. Just as a hospital must continue operating even when individual departments face equipment failures, network outages, or staff shortages, our metrics system must gracefully handle target unavailability, storage corruption, and resource exhaustion while maintaining service for healthy components. The key insight is that metrics collection is inherently about observability—if our observability system itself becomes unreliable, we lose visibility into the systems we're monitoring, creating a dangerous blind spot.

Effective error handling in distributed systems requires a layered approach. Each component must handle its local failure modes while contributing to system-wide resilience patterns. This means implementing circuit breakers to isolate failing targets, graceful degradation to maintain partial functionality under stress, and comprehensive recovery mechanisms that can rebuild consistent state after crashes.

Target Unavailability

The **scrape engine** operates in an inherently unreliable environment where targets may be temporarily unreachable due to network partitions, service restarts, configuration changes, or resource exhaustion. Unlike traditional request-

response systems where failures are immediately visible to users, metrics collection operates on a background schedule where failures accumulate silently until noticed through missing data or monitoring alerts.

Think of target health management like a cardiac monitor in an intensive care unit. The monitor must distinguish between genuine cardiac events (target is actually down) and sensor failures (network timeout, probe malformation), while maintaining a clear history of both successful and failed readings. A single missed heartbeat doesn't indicate cardiac arrest, but a pattern of missed beats requires immediate attention.

The `TargetHealth` component tracks the availability and reliability status of each scrape endpoint through state transitions based on success and failure patterns. This health tracking serves multiple purposes: it prevents wasted resources on consistently failing targets, provides visibility into target reliability, and enables adaptive scraping strategies that reduce load on struggling services.

Health State	Entry Condition	Scraping Behavior	Transition Triggers
<code>HealthUp</code>	3 consecutive successful scrapes	Normal interval scraping	2 consecutive failures → <code>HealthDegraded</code>
<code>HealthDegraded</code>	2 consecutive failures from <code>HealthUp</code>	Reduced frequency (2x interval)	3 consecutive successes → <code>HealthUp</code> , 3 more failures → <code>HealthDown</code>
<code>HealthDown</code>	3 consecutive failures from <code>HealthDegraded</code>	Exponential backoff (max 5min intervals)	5 consecutive successes → <code>HealthDegraded</code>

Network Timeout Handling

Network timeouts represent the most common failure mode in distributed scraping systems. The challenge lies in distinguishing between genuinely slow targets that need more time and unresponsive targets that should be abandoned quickly to prevent resource exhaustion.

Decision: Multi-Layered Timeout Strategy

- **Context:** Single timeout values either cause false failures for slow targets or waste resources on dead targets
- **Options Considered:** Fixed timeout, adaptive timeout based on target history, layered timeouts with different retry behavior
- **Decision:** Implement layered timeouts with connection timeout (5s), first-byte timeout (10s), and total request timeout (30s)
- **Rationale:** This provides fast failure detection for network issues while allowing reasonable time for slow but healthy targets
- **Consequences:** More complex timeout logic but better balance between reliability and resource usage

The `HttpClient` implements three distinct timeout layers that work together to provide fast failure detection while accommodating legitimate performance variations:

Timeout Layer	Duration	Purpose	Failure Indication
Connection Timeout	5 seconds	Detect network connectivity issues	Target unreachable or network partition
First Byte Timeout	10 seconds	Detect application-level hangs	Target process hung or severely overloaded
Total Request Timeout	30 seconds	Prevent resource exhaustion	Large response or very slow processing

The scrape engine handles timeout failures by updating target health state and implementing exponential backoff for consistently failing targets. This prevents the scraper from overwhelming targets that are experiencing temporary resource constraints while quickly returning to normal scraping frequency once targets recover.

HTTP Error Response Handling

HTTP error responses provide valuable diagnostic information about target state and should be handled differently based on their semantic meaning. Unlike timeouts, which may indicate transient network issues, HTTP errors often indicate configuration problems or application-level failures that require different recovery strategies.

The scrape engine categorizes HTTP errors into permanent failures (4xx client errors) and temporary failures (5xx server errors), applying different retry strategies and health state transitions for each category:

Error Category	Status Codes	Retry Strategy	Health Impact	Diagnostic Action
Authentication/Authorization	401, 403	No retry - requires configuration fix	Immediate HealthDown	Log configuration error, alert operator
Client Errors	400, 404, 405	No retry - indicates configuration problem	Immediate HealthDown	Validate target URL and endpoint configuration
Server Overload	429, 503	Exponential backoff with jitter	Gradual degradation	Reduce scraping frequency, implement circuit breaker
Server Errors	500, 502, 504	Standard retry with timeout	Follow normal health transitions	Monitor for pattern indicating systemic issues

Malformed Metrics Handling

The Prometheus exposition format parsing must handle malformed metrics data gracefully while preserving successfully parsed metrics from the same scrape operation. Think of this like a data quality inspector at a manufacturing plant—defective items should be rejected and logged, but the entire batch shouldn't be discarded if most items are acceptable.

Parse errors fall into several categories that require different handling strategies:

Metric Name Validation Errors: Metric names that violate naming conventions (contain invalid characters, use reserved prefixes, or exceed length limits) are rejected at parse time. The parser logs the specific validation failure

and continues processing remaining metrics from the same scrape.

Label Format Errors: Labels with invalid names, missing values, or encoding issues are handled by either dropping the problematic label (if it's not critical) or dropping the entire metric (if the label is required for uniqueness). The decision depends on whether the remaining labels provide sufficient identity for the time series.

Value Format Errors: Timestamps or values that cannot be parsed as valid numbers cause the specific sample to be dropped while preserving other samples from the same metric. This is particularly important for histogram metrics where individual bucket counts may be malformed while others are valid.

Timestamp Consistency Errors: Samples with timestamps significantly in the past or future (outside a configurable staleness threshold) are dropped to prevent storage corruption and query inconsistencies. The parser maintains a window of acceptable timestamps based on scrape time.

Decision: Partial Scrape Success Model

- **Context:** Scrapes often contain mix of valid and invalid metrics, binary success/failure loses valuable data
- **Options Considered:** All-or-nothing parsing, best-effort parsing with warnings, configurable error tolerance thresholds
- **Decision:** Implement best-effort parsing that preserves valid metrics while logging specific parse failures
- **Rationale:** Metrics collection should be resilient to individual metric formatting issues without losing all observability
- **Consequences:** More complex error reporting but better data availability and easier debugging of format issues

Circuit Breaker Implementation

Circuit breakers prevent the scrape engine from wasting resources on consistently failing targets while enabling rapid recovery when targets become healthy again. The implementation tracks failure patterns and automatically transitions between closed (normal operation), open (failing fast), and half-open (testing recovery) states.

The circuit breaker maintains failure statistics over a sliding time window and makes state transition decisions based on failure rate thresholds rather than absolute failure counts. This provides more robust behavior under varying traffic patterns:

Circuit State	Scraping Behavior	Failure Threshold	Success Requirement	Transition Logic
Closed	Normal scraping at configured interval	50% failures over 5 minutes	N/A	Failure rate exceeds threshold → Open
Open	No scraping, immediate failure response	N/A	N/A	After 1 minute timeout → Half-Open
Half-Open	Single test scrape allowed	N/A	1 successful scrape	Success → Closed, Failure → Open

Storage Errors

The **time series storage engine** must handle various failure modes while maintaining data consistency and preventing corruption. Unlike stateless components that can simply restart after failures, the storage engine maintains persistent state that requires careful recovery procedures and consistency guarantees.

Think of storage error handling like a bank's vault system. The vault must protect deposits even during power outages, hardware failures, or software crashes. Every transaction must be logged before execution (write-ahead logging), and after any disruption, the bank must verify the vault's contents match the transaction log exactly. Similarly, our storage engine uses write-ahead logging and consistency checks to ensure no data is lost or corrupted during failures.

Disk Space Exhaustion

Disk space exhaustion represents one of the most critical failure modes because it can cause data loss and prevent normal operation. The storage engine must detect approaching disk exhaustion, implement emergency data retention policies, and gracefully degrade functionality to preserve the most important data.

The storage engine implements a multi-level disk space monitoring system that triggers different responses based on available space thresholds:

Available Space	Action Taken	Data Retention Policy	Write Behavior
> 20%	Normal operation	Standard retention period	All writes accepted
10-20%	Warning logs, accelerate cleanup	Reduce retention to 75% of configured	All writes accepted
5-10%	Emergency retention, pause compaction	Reduce retention to 50% of configured	High-priority metrics only
< 5%	Read-only mode, aggressive cleanup	Reduce retention to 25% of configured	No new writes accepted

The emergency retention system prioritizes metrics based on configured importance levels and access patterns. Critical infrastructure metrics (CPU, memory, disk usage) receive highest priority, while application-specific metrics may be discarded first during space constraints.

Decision: Graceful Degradation Over Hard Failures

- **Context:** Disk exhaustion traditionally causes complete storage system failure and data loss
- **Options Considered:** Fail fast when disk full, emergency retention policies, offload data to remote storage
- **Decision:** Implement layered retention policies that preserve most important data while maintaining read access
- **Rationale:** Partial observability is much more valuable than complete loss of metrics collection during emergencies
- **Consequences:** More complex retention logic but maintains system availability during resource constraints

Write-Ahead Log Corruption Recovery

The `WriteAheadLog` provides durability guarantees by recording all intended writes before they're applied to the main storage indexes. WAL corruption can occur due to hardware failures, power outages during writes, or filesystem issues. Recovery procedures must detect corruption and rebuild consistent state without losing committed data.

WAL recovery follows a structured approach that validates log integrity and replays valid entries while handling various corruption scenarios:

- 1. Log Segment Validation:** Each WAL segment begins with a header containing a checksum of the segment's contents. During recovery, the system validates each segment header and marks corrupted segments for special handling.
- 2. Entry Checksum Verification:** Individual log entries contain CRC32 checksums that detect corruption within valid segments. Corrupted entries are skipped during replay, and their absence is logged for operator investigation.
- 3. Partial Write Detection:** Power failures can cause partial writes where only part of a log entry is written to disk. The recovery system detects these by checking entry length headers against available data.
- 4. Timestamp Consistency Checking:** Replicated entries must have timestamps consistent with the recovery point. Entries with timestamps far in the future or past are considered suspect and may indicate clock issues or corruption.
- 5. Index Rebuilding:** After replaying all valid WAL entries, the system rebuilds its in-memory indexes and validates them against the recovered data. Any inconsistencies trigger a full reindex operation.

Corruption Type	Detection Method	Recovery Action	Data Impact
Segment Header Corruption	Header checksum mismatch	Skip entire segment, log corruption	Loss of data in corrupted segment
Entry Corruption	Entry checksum failure	Skip corrupted entry, continue replay	Loss of individual corrupted entries
Partial Write	Length header mismatch	Truncate at last valid entry	Loss of incomplete final entry
Timestamp Inconsistency	Time bounds checking	Skip suspicious entries with warnings	Potential loss of out-of-order data

Index Inconsistency Recovery

The `InvertedIndexes` maintain mappings from metric names and label combinations to time series identifiers. Index corruption can cause queries to return incorrect results or fail to find existing data. The storage engine detects index inconsistencies during normal operations and can rebuild indexes from the authoritative chunk data.

Index consistency checking operates continuously during normal operations, validating that index entries correspond to actual stored data and that all stored data has appropriate index entries:

Forward Consistency Checking: For each index entry pointing to a time series, verify that the referenced time series actually exists in storage and contains the expected metric name and labels.

Reverse Consistency Checking: For each stored time series, verify that appropriate index entries exist and point to the correct series identifier.

Cross-Index Consistency: Verify that the metric name index, label value indexes, and series metadata index all contain consistent information about the same time series.

Cardinality Validation: Compare the number of unique time series found in storage against cardinality tracking counters to detect missing or extra index entries.

The recovery system can rebuild indexes in several modes depending on the severity of detected inconsistencies:

Inconsistency Type	Rebuild Strategy	Downtime Required	Performance Impact
Missing Index Entries	Incremental rebuild of missing entries	None	Temporary query slowdown
Extra Index Entries	Remove orphaned entries during background cleanup	None	Minimal
Cross-Index Mismatch	Full reindex of affected metric names	Read-only mode during rebuild	Significant temporary impact
Systematic Corruption	Complete index rebuild from chunk data	Full downtime during rebuild	Complete rebuild required

Data Consistency Validation

The storage engine implements continuous data consistency checking that validates the integrity of compressed chunks, the accuracy of series metadata, and the correctness of time ordering within series data.

Consistency validation operates at multiple levels:

Chunk-Level Validation: Each compressed chunk maintains metadata about its time range, sample count, and compression parameters. The validation system periodically decompresses chunks and verifies that the contained samples match the metadata.

Series-Level Validation: Time series must maintain strict time ordering of samples and consistent label sets across all chunks. The validation system checks for timestamp inversions, duplicate timestamps, and label mutations within series.

Storage-Level Validation: The overall storage system maintains invariants about total series count, disk space usage, and index sizes. Background validation jobs verify these invariants and alert operators to systematic issues.

Cross-Component Validation: Consistency checks verify that WAL entries, compressed chunks, and index entries all describe the same underlying data consistently.

Query Errors

The **query engine** must handle malformed queries, missing data scenarios, and resource exhaustion while providing useful error messages that help users diagnose and fix their queries. Query error handling is particularly challenging because users may not understand the underlying data model or system limitations.

Think of query error handling like a reference librarian helping researchers find information. When a researcher asks for something that doesn't exist or is unclear in their request, the librarian doesn't simply say "not found"—they explain what resources are available, suggest alternatives, and help reformulate the request to find relevant information.

Invalid Expression Parsing

PromQL expression parsing must handle syntax errors gracefully while providing specific error messages that help users correct their queries. The parser encounters various error categories that require different diagnostic approaches:

Lexical Errors: Invalid characters, unterminated strings, or malformed numbers in the query text. These errors include specific position information and suggest valid alternatives.

Syntax Errors: Grammatically incorrect expressions such as mismatched parentheses, invalid operator precedence, or incomplete function calls. The parser provides context about expected tokens and suggests corrections.

Semantic Errors: Syntactically correct expressions that violate PromQL semantic rules, such as applying vector operators to scalar values or using undefined functions. These errors explain the type mismatch and suggest valid operations.

Label Selector Errors: Invalid label matching expressions, malformed regular expressions, or label names that violate naming conventions. The parser validates regex patterns and provides specific regex error messages.

The `ExpressionParser` maintains an error collection that accumulates multiple parsing errors and provides comprehensive feedback rather than stopping at the first error encountered:

Error Category	Example Query	Error Message	Suggested Fix
Lexical	<code>http_requests_@total</code>	Invalid character '@' at position 13	Use valid metric name characters: [a-zA-Z0-9_:]
Syntax	<code>http_requests_total +</code>	Unexpected end of expression, expected right operand	Complete the addition operation: <code>+ <expression></code>
Semantic	<code>rate("string")</code>	Function rate() expects vector argument, got string	Use vector selector: <code>rate(http_requests_total[5m])</code>
Label Selector	<code>{__name__=~"invalid"</code>	Invalid regex pattern: missing closing bracket	Fix regex pattern: <code>{__name__=~"valid.*"}</code>

Decision: Detailed Error Context Over Simple Error Messages

- **Context:** Generic error messages like "parse error" don't help users fix their queries effectively
- **Options Considered:** Simple error codes, detailed position-specific messages, suggested corrections with examples
- **Decision:** Provide detailed error messages with position information, context, and suggested corrections
- **Rationale:** Query languages are complex and users need specific guidance to construct valid expressions
- **Consequences:** More complex error handling code but much better user experience and faster debugging

Missing Data Handling

Time series data is inherently sparse—not all metrics are available at all times, and query time ranges may extend beyond available data. The query engine must handle these scenarios gracefully while providing clear indication of data availability issues.

Missing data scenarios require different handling strategies based on their cause and the query type:

Series Not Found: When a query references metric names or label combinations that don't exist in storage, the query engine returns empty results rather than errors. This matches Prometheus behavior where missing metrics are treated as empty result sets rather than error conditions.

Time Range Gaps: Queries may request data from time ranges where no samples were collected due to scraping failures or target downtime. The query engine interpolates across small gaps (less than 2 scrape intervals) but returns empty values for larger gaps.

Staleness Handling: Samples older than the configured staleness threshold (typically 5 minutes) are considered stale and excluded from instant queries. Range queries may include stale samples with appropriate warnings.

Partial Data Availability: When some series in an aggregation query have data while others don't, the query engine includes warnings about partial results and indicates which series contributed to the final result.

The query engine provides comprehensive warnings about data availability issues:

Missing Data Type	Query Behavior	Warning Message	User Action
Metric Not Found	Return empty result	No time series found for selector	Verify metric name and labels exist
Time Range Gap	Return empty for gap period	Data gap detected from X to Y	Check target availability during gap
Stale Data	Exclude from instant queries	Using stale data older than threshold	Check recent scraping status
Partial Aggregation	Include available series only	Aggregation includes only N of M series	Investigate missing series

Resource Exhaustion Protection

Query execution can consume significant memory and CPU resources, particularly for queries over long time ranges or high-cardinality metrics. The query engine implements multiple layers of resource protection to prevent individual queries from affecting system stability.

Think of query resource limits like admission control at a concert venue. The venue has a maximum capacity that ensures safety and comfort for all attendees. When capacity is reached, additional people must wait rather than creating an unsafe overcrowding situation. Similarly, the query engine limits concurrent queries and per-query resource usage to maintain stability for all users.

The `QueryCoordinator` implements several resource protection mechanisms:

Concurrent Query Limiting: A semaphore limits the number of queries executing simultaneously to prevent CPU and memory exhaustion. Queries that exceed the limit are queued or rejected with appropriate error messages.

Per-Query Memory Limits: Each query has a maximum memory allocation for storing intermediate results and final output. Queries that exceed this limit are terminated with resource exhaustion errors.

Time Range Restrictions: Very long time range queries can consume excessive resources and are limited to configurable maximum durations (typically 30 days). Users must break long-range queries into smaller segments.

Series Count Limits: Queries that would process more than a maximum number of time series (typically 10,000) are rejected to prevent memory exhaustion from high-cardinality selectors.

Query Timeout Enforcement: All queries have maximum execution time limits (typically 30 seconds) after which they're terminated to prevent resource hoarding.

The resource protection system provides specific error messages that help users understand the limit violation and suggest query modifications:

Resource Limit	Threshold	Error Message	Suggested Solution
Concurrent Queries	10 simultaneous	Maximum concurrent queries exceeded, try again later	Wait for other queries to complete or increase limit
Memory Usage	1GB per query	Query memory limit exceeded at N GB	Reduce time range or use more specific label selectors
Time Range	30 days	Query time range exceeds maximum of 30 days	Break query into smaller time segments
Series Count	10,000 series	Query would examine N series, limit is 10,000	Use more specific label selectors to reduce cardinality
Execution Time	30 seconds	Query timeout after 30 seconds	Simplify query or reduce time range

Resource Protection

System-wide resource protection ensures that no single component or operation can compromise overall system stability. This requires coordination between all components and global limits that prevent resource exhaustion at the system level.

Think of system resource protection like air traffic control at a major airport. Individual flights (queries, scrapes, storage operations) may have their own requirements, but the control system must manage overall airport capacity, runway utilization, and airspace congestion to keep all flights operating safely. No single flight is allowed to disrupt the entire airport's operation.

Memory Management Strategy

Memory usage in metrics collection systems can grow rapidly due to high-cardinality metrics, large query results, or accumulated metadata. The system implements a comprehensive memory management strategy that monitors usage, enforces limits, and provides graceful degradation when approaching limits.

The memory management system operates at three levels:

Component-Level Limits: Each major component (scrape engine, storage engine, query engine) has dedicated memory quotas that prevent any single component from consuming all available memory.

Operation-Level Limits: Individual operations within components have specific memory limits that prevent single large operations from affecting other operations.

System-Level Monitoring: Global memory usage monitoring triggers emergency responses when total system memory usage approaches dangerous levels.

Component	Memory Quota	Enforcement Mechanism	Degradation Strategy
Scrape Engine	25% of system memory	Limit concurrent scrapes	Reduce scraping parallelism
Storage Engine	50% of system memory	Limit chunk cache and WAL buffer size	Flush caches more aggressively
Query Engine	20% of system memory	Per-query limits and queue depth	Reject or queue new queries
System Overhead	5% reserved	OS and other processes	Emergency garbage collection

Cardinality Explosion Prevention

Label cardinality explosion represents one of the most dangerous failure modes in metrics systems because it can cause exponential growth in memory usage and storage requirements. The system implements multiple layers of cardinality control that detect, prevent, and mitigate high-cardinality scenarios.

The `CardinalityTracker` monitors series creation rates and label combination patterns to detect cardinality explosions early:

Per-Metric Cardinality Limits: Each metric name has a maximum number of allowed label combinations (typically 10,000 series). New series exceeding this limit are rejected with specific error messages.

Label Value Limits: Individual labels have maximum numbers of allowed distinct values (typically 1,000 values per label name). This prevents single labels from creating excessive cardinality.

Series Creation Rate Limits: The system monitors the rate of new time series creation and temporarily blocks new series when creation rates exceed thresholds that indicate cardinality explosions.

High-Cardinality Detection: Background monitoring identifies metrics and labels contributing most to total cardinality, providing operators with actionable information about cardinality sources.

The cardinality protection system provides detailed error messages that help users understand the cardinality violation and suggest alternatives:

Cardinality Limit	Threshold	Error Message	Remediation Strategy
Per-Metric Series	10,000 series	Metric 'X' exceeds 10,000 series limit	Use fewer label dimensions or aggregate at source
Label Value Count	1,000 values	Label 'Y' has too many distinct values	Use label value prefixes or reduce granularity
Series Creation Rate	1,000/minute	New series creation rate too high	Review metric instrumentation for cardinality bugs
Total System Series	1,000,000 series	System approaching maximum series capacity	Review all metrics for optimization opportunities

⚠ Pitfall: Cardinality Limits Too Restrictive Setting cardinality limits too low can prevent legitimate use cases where high cardinality is necessary for proper observability. The limits should be based on actual system capacity and usage patterns rather than arbitrary conservative values. Monitor actual cardinality usage and adjust limits based on system performance and storage capacity.

Graceful Degradation Mechanisms

When the system approaches resource limits, graceful degradation maintains partial functionality rather than complete system failure. This requires careful prioritization of operations and temporary reduction in quality or completeness of service.

The system implements several graceful degradation strategies:

Scraping Degradation: When memory or CPU usage is high, the scrape engine reduces scraping frequency for less critical targets while maintaining normal intervals for high-priority metrics.

Storage Degradation: The storage engine may temporarily disable compression or reduce cache sizes to free memory for critical operations like query processing.

Query Degradation: The query engine may impose stricter limits on query complexity, reduce the maximum number of returned data points, or provide approximate results instead of exact calculations.

Service Shedding: In extreme overload situations, the system may temporarily reject new requests (queries or metric ingestion) while processing the existing workload.

Each degradation mechanism provides clear indication of reduced service levels and estimates of when normal operation will resume:

Degradation Type	Trigger Condition	Service Impact	Recovery Condition
Reduced Scraping	CPU > 80% for 5 minutes	50% longer intervals for low-priority targets	CPU < 60% for 2 minutes
Storage Throttling	Memory > 85%	Compression disabled, smaller write batches	Memory < 75%
Query Limiting	Query queue > 50 requests	Stricter per-query limits, longer timeouts	Queue < 10 requests
Request Rejection	System overload detected	HTTP 503 responses with retry-after headers	Load drops below threshold

Decision: Graceful Degradation Over Complete Failure

- Context:** Resource exhaustion traditionally causes complete system failure and loss of observability
- Options Considered:** Fail fast when resources exhausted, graceful degradation with reduced service, horizontal scaling requirements
- Decision:** Implement graceful degradation that maintains critical functionality while reducing less important operations
- Rationale:** Partial observability during high load is much more valuable than complete loss of metrics during peak demand
- Consequences:** More complex resource management but maintains system availability during stress conditions

Implementation Guidance

Technology Recommendations

Component	Simple Option	Advanced Option
Error Handling	Standard Go error values with <code>fmt.Error</code>	Structured errors with github.com/pkg/errors for stack traces
Circuit Breaker	Custom implementation with simple counters	github.com/sony/gobreaker for production features
Health Monitoring	Basic success/failure counters	Exponential moving averages for failure rates
Resource Limits	Simple <code>sync.WaitGroup</code> semaphores	Context-based cancellation with resource tracking
Logging	Standard log package	Structured logging with github.com/sirupsen/logrus

File Structure

```
internal/
├── errors/
│   ├── errors.go          ← Custom error types and utilities
│   ├── circuit_breaker.go ← Circuit breaker implementation
│   └── health.go          ← Target health tracking
├── scrape/
│   ├── health.go          ← Target health management
│   ├── timeout.go          ← Multi-layer timeout handling
│   └── parser_errors.go    ← Metrics parsing error handling
├── storage/
│   ├── consistency.go      ← Data consistency validation
│   ├── recovery.go          ← WAL and index recovery
│   └── disk_monitor.go      ← Disk space monitoring
├── query/
│   ├── resource_limits.go  ← Query resource protection
│   ├── parse_errors.go      ← Query parsing error handling
│   └── coordinator.go       ← Query concurrency management
└── system/
    ├── resource_monitor.go  ← System-wide resource tracking
    └── degradation.go        ← Graceful degradation logic
```

Infrastructure Starter Code

Complete error handling infrastructure that provides structured error types and utilities:

GO

```
// internal/errors/errors.go

package errors

import (
    "fmt"
    "time"
)

// ErrorType categorizes different kinds of errors for handling decisions

type ErrorType string

const (
    ErrorTypeTransient    ErrorType = "transient"      // Retry may succeed
    ErrorTypePermanent    ErrorType = "permanent"       // No point retrying
    ErrorTypeRateLimit    ErrorType = "rate_limit"     // Need backoff
    ErrorTypeResource     ErrorType = "resource"        // Resource exhaustion
)

// MetricsError provides structured error information with context

type MetricsError struct {

    Type        ErrorType
    Component   string
    Operation   string
    Message     string
    Cause       error
    Timestamp   time.Time
    Context     map[string]interface{}
}

func (e *MetricsError) Error() string {
    if e.Cause != nil {
```

```
        return fmt.Sprintf("%s/%s: %s (caused by: %v)", e.Component, e.Operation, e.Message,
e.Cause)

    }

    return fmt.Sprintf("%s/%s: %s", e.Component, e.Operation, e.Message)

}

func NewTransientError(component, operation, message string, cause error) *MetricsError {

    return &MetricsError{

        Type:      ErrorTypeTransient,

        Component: component,

        Operation: operation,

        Message:   message,

        Cause:     cause,

        Timestamp: time.Now(),

        Context:   make(map[string]interface{}),

    }

}

func NewPermanentError(component, operation, message string, cause error) *MetricsError {

    return &MetricsError{

        Type:      ErrorTypePermanent,

        Component: component,

        Operation: operation,

        Message:   message,

        Cause:     cause,

        Timestamp: time.Now(),

        Context:   make(map[string]interface{}),

    }

}
```

```
// IsRetryable determines if an error should trigger retry logic

func IsRetryable(err error) bool {

    if metricsErr, ok := err.(*MetricsError); ok {

        return metricsErr.Type == ErrorTypeTransient || metricsErr.Type == ErrorTypeRateLimit
    }

    return false
}
```

Complete circuit breaker implementation:

GO

```
// internal/errors/circuit_breaker.go

package errors

import (
    "sync"
    "time"
)

type CircuitState int

const (
    CircuitClosed CircuitState = iota
    CircuitOpen
    CircuitHalfOpen
)

type CircuitBreaker struct {

    mu           sync.RWMutex
    state        CircuitState
    failureCount int
    successCount int
    lastFailureTime   time.Time
    lastSuccessTime  time.Time

    // Configuration

    failureThreshold  int
    successThreshold   int
    timeout           time.Duration
    onStateChange     func(from, to CircuitState)
}
```

```
func NewCircuitBreaker(failureThreshold, successThreshold int, timeout time.Duration)
*CircuitBreaker {
    return &CircuitBreaker{
        state:           CircuitClosed,
        failureThreshold: failureThreshold,
        successThreshold: successThreshold,
        timeout:         timeout,
    }
}

func (cb *CircuitBreaker) Call(fn func() error) error {
    if !cb.allowRequest() {
        return NewPermanentError("circuit_breaker", "call", "circuit breaker is open", nil)
    }

    err := fn()
    if err != nil {
        cb.recordFailure()
        return err
    }

    cb.recordSuccess()
    return nil
}

func (cb *CircuitBreaker) allowRequest() bool {
    cb.mu.RLock()
    defer cb.mu.RUnlock()

    switch cb.state {
```

```
case CircuitClosed:

    return true

case CircuitOpen:

    return time.Since(cb.lastFailureTime) >= cb.timeout

case CircuitHalfOpen:

    return true

}

return false

}

func (cb *CircuitBreaker) recordSuccess() {

    cb.mu.Lock()

    defer cb.mu.Unlock()

    cb.successCount++

    cb.lastSuccessTime = time.Now()

    if cb.state == CircuitHalfOpen && cb.successCount >= cb.successThreshold {

        cb.setState(CircuitClosed)

        cb.failureCount = 0

        cb.successCount = 0

    }

}

func (cb *CircuitBreaker) recordFailure() {

    cb.mu.Lock()

    defer cb.mu.Unlock()

    cb.failureCount++
```

```
cb.lastFailureTime = time.Now()

if cb.state == CircuitClosed && cb.failureCount >= cb.failureThreshold {

    cb.setState(CircuitOpen)

} else if cb.state == CircuitHalfOpen {

    cb.setState(CircuitOpen)

    cb.successCount = 0

}

}

func (cb *CircuitBreaker) setState(newState CircuitState) {

    oldState := cb.state

    cb.state = newState

    if cb.onStateChange != nil && oldState != newState {

        go cb.onStateChange(oldState, newState)

    }

}
```

Core Logic Skeleton Code

Target health management with state transitions:

```
// internal/scrape/health.go                                         GO

func (th *TargetHealth) RecordScrapeResult(success bool, duration time.Duration, err error) {

    th.mu.Lock()

    defer th.mu.Unlock()

    th.lastScrapeTime = time.Now()

    if success {

        // TODO 1: Reset consecutive failure counter to 0

        // TODO 2: Check if we should transition from HealthDown or HealthDegraded to better
        state

        // TODO 3: Update lastSuccessTime and record scrape duration for metrics

        // Hint: Use transition thresholds from health state table above

    } else {

        // TODO 1: Increment consecutive failure counter

        // TODO 2: Check if failure count crosses threshold for state degradation

        // TODO 3: Store error information for diagnostic purposes

        // TODO 4: Update exponential backoff interval if in HealthDown state

        // Hint: Max backoff should be 5 minutes, use exponential growth with jitter

    }

}

func (th *TargetHealth) ShouldScrape() (bool, time.Duration) {

    th.mu.RLock()

    defer th.mu.RUnlock()

    // TODO 1: Check current health state and return appropriate scrape decision

    // TODO 2: For HealthUp: return true with normal interval

    // TODO 3: For HealthDegraded: return true with 2x normal interval
```

```

    // TODO 4: For HealthDown: return true only if backoff period has elapsed

    // TODO 5: Calculate next scrape time based on state and backoff strategy

    // Hint: Use time.Since(th.lastScrapeTime) to check if enough time has passed

}

```

Storage disk space monitoring with emergency retention:

```

// internal/storage/disk_monitor.go                                         GO

func (dm *DiskMonitor) CheckDiskSpace() (*DiskSpaceStatus, error) {

    // TODO 1: Use syscall.Statfs (Unix) or similar to get filesystem stats

    // TODO 2: Calculate available space percentage

    // TODO 3: Determine alert level based on thresholds from disk space table

    // TODO 4: If below emergency threshold, trigger immediate retention cleanup

    // TODO 5: Return status with recommended actions for storage engine

    // Hint: Available space = (free blocks * block size) / (total blocks * block size)

}

func (dm *DiskMonitor) TriggerEmergencyRetention(targetSpacePercent float64) error {

    // TODO 1: Calculate how much data needs to be deleted to reach target space

    // TODO 2: Identify oldest data chunks that can be safely deleted

    // TODO 3: Delete chunks in order of age, prioritizing low-importance metrics

    // TODO 4: Update indexes to remove references to deleted chunks

    // TODO 5: Force garbage collection and filesystem sync to reclaim space immediately

    // Hint: Use priority levels stored in SeriesMetadata to determine deletion order

}

```

Query resource limiting with graceful degradation:

GO

```
// internal/query/resource_limits.go

func (ql *QueryLimiter) ExecuteWithLimits(ctx context.Context, query string, fn func()
(*QueryResult, error)) (*QueryResult, error) {

    // TODO 1: Check if query exceeds complexity limits (time range, series count estimate)

    // TODO 2: Acquire semaphore slot for concurrent query limiting

    // TODO 3: Set up memory tracking for this query execution

    // TODO 4: Create timeout context if none provided or if shorter than configured limit

    // TODO 5: Execute query function with resource monitoring

    // TODO 6: Check memory usage during execution, terminate if limit exceeded

    // TODO 7: Release resources and update usage statistics

    // Hint: Use context.WithTimeout and context.WithCancel for resource control

}

func (ql *QueryLimiter) EstimateQueryComplexity(query string, timeRange time.Duration)
(*ComplexityEstimate, error) {

    // TODO 1: Parse query to identify metric selectors and operations

    // TODO 2: Estimate number of time series that would be examined

    // TODO 3: Calculate memory requirements based on time range and series count

    // TODO 4: Identify expensive operations (regex matching, aggregation functions)

    // TODO 5: Return complexity estimate with resource requirements

    // Hint: Use simple heuristics like series_count * time_range_hours * 8 bytes per sample

}
```

Milestone Checkpoints

Target Health Management Verification:

- Run scrape engine with mix of healthy and failing targets
- Verify state transitions follow the health state table exactly
- Check that backoff intervals increase exponentially for failing targets
- Confirm circuit breaker prevents wasted requests to consistently failing targets

Storage Error Recovery Testing:

- Simulate disk full condition and verify graceful degradation
- Kill storage process during writes and verify WAL recovery

- Corrupt index files and verify automatic rebuild from chunk data
- Verify consistency checker detects and reports data inconsistencies

Query Resource Protection Validation:

- Submit queries exceeding memory limits and verify they're terminated
- Run many concurrent queries and verify queueing/rejection behavior
- Test very long time range queries get appropriate error messages
- Verify high-cardinality queries are rejected with specific feedback

System-Wide Degradation Testing:

- Load system to resource limits and verify graceful degradation activates
- Confirm partial functionality maintained during resource exhaustion
- Verify system automatically recovers when load decreases
- Check that degradation status is clearly visible in system metrics

Debugging Tips

Symptom	Likely Cause	Diagnosis	Fix
Targets show as down but are accessible	Circuit breaker stuck open	Check circuit breaker state and failure history	Reset circuit breaker or adjust thresholds
Storage consuming excessive disk space	Retention policy not running	Check disk monitor logs and retention job status	Manually trigger retention cleanup
Queries timing out frequently	Resource limits too restrictive	Check query complexity estimates and actual resource usage	Increase limits or optimize queries
Memory usage growing indefinitely	Cardinality explosion not detected	Check series creation rates and label cardinality	Implement stricter cardinality limits
WAL recovery taking very long	Large WAL files with many segments	Check WAL segment sizes and rotation frequency	Adjust WAL rotation settings
Parse errors for valid metrics format	Strict validation rejecting valid data	Check parsing error logs for specific validation failures	Relax validation rules or fix data format

Testing and Validation Strategy

Milestone(s): This section provides comprehensive testing approaches for all four milestones: Metrics Data Model (1), Scrape Engine (2), Time Series Storage (3), and Query Engine (4), along with end-to-end validation of their integrated operation.

The Quality Assurance Lighthouse Mental Model

Think of our testing strategy like a lighthouse system guiding ships safely to shore. Just as lighthouses provide multiple layers of navigation safety—from close-in harbor lights to distant beacon warnings—our testing strategy provides multiple concentric layers of validation. The innermost ring consists of **component testing** that validates individual pieces work correctly in isolation, like harbor lights that help boats navigate the immediate docking area. The middle ring contains **integration testing** that verifies components work together properly, like channel markers that guide ships through connected waterways. The outermost ring encompasses **end-to-end testing** that validates complete user workflows, like the main lighthouse beacon that provides navigation from miles offshore. Performance validation acts as the weather monitoring system, ensuring our lighthouse remains operational under all conditions—calm seas and storms alike.

This layered approach ensures that problems are caught as early and specifically as possible. A unit test failure tells us exactly which component is broken, like a harbor light malfunction points to a specific dock. An integration test failure indicates communication problems between components, like misaligned channel markers. An end-to-end test failure suggests workflow problems that might only appear under realistic usage patterns, like a lighthouse whose beam is blocked by unexpected fog.

Component Testing

Component testing validates each system component in isolation, using test doubles to eliminate external dependencies and focus on the component's core logic and behavior. This isolation allows us to test error conditions, edge cases, and performance characteristics that would be difficult to reproduce in integrated scenarios.

Metrics Data Model Testing

The metrics data model components require thorough testing of their semantic behaviors, thread safety, and cardinality management capabilities. Each metric type has specific behavioral contracts that must be validated under both normal and stress conditions.

Test Category	Scope	Key Validations	Test Techniques
Counter Semantics	Counter increment behavior	Monotonic increase, no decreases allowed, overflow handling	Property-based testing with random increments
Gauge Semantics	Gauge set/add behavior	Arbitrary values allowed, thread-safe updates	Concurrent goroutines setting values
Histogram Semantics	Bucket observation behavior	Correct bucket assignment, sum/count accuracy	Statistical validation of distributions
Label Validation	Label cardinality control	Name/value length limits, character restrictions, cardinality tracking	Fuzzing with invalid characters and long strings
Thread Safety	Concurrent metric updates	No data races, consistent reads during writes	Race detector with high concurrency

Counter Behavior Testing: Counter tests verify that increment operations maintain monotonic behavior and that attempts to decrease values are properly rejected. The test suite generates random sequences of increment

operations and validates that the final value equals the sum of all increments. Thread safety testing spawns multiple goroutines performing concurrent increments and verifies that no increments are lost due to race conditions.

Gauge Behavior Testing: Gauge tests validate that set and add operations work correctly with both positive and negative values. Unlike counters, gauges must accept any floating-point value including negative numbers, zero, and special values like positive/negative infinity. Concurrent testing verifies that rapid updates from multiple goroutines don't corrupt the stored value.

Histogram Distribution Testing: Histogram tests validate that observed values are assigned to the correct buckets and that sum and count fields maintain accuracy. Statistical testing generates known distributions and verifies that the histogram buckets capture the distribution shape correctly. Edge case testing validates behavior with extreme values, NaN, and infinity.

Label Cardinality Testing: Label validation testing ensures that the cardinality tracking system correctly prevents label explosion attacks. Tests attempt to create series with high-cardinality label combinations and verify that appropriate limits are enforced. Performance testing measures memory usage as cardinality increases to validate that growth remains within acceptable bounds.

Decision: Property-Based Testing for Metric Semantics

- **Context:** Metric types have mathematical properties that must hold across all possible input sequences
- **Options Considered:** Example-based unit tests, property-based testing, formal verification
- **Decision:** Property-based testing with example-based edge cases
- **Rationale:** Properties like counter monotonicity are universal invariants that should hold for any valid input sequence. Property-based testing explores the input space more thoroughly than hand-written examples while remaining practical to implement
- **Consequences:** Tests catch more edge cases but may be harder to debug when they fail. Requires careful property specification to avoid vacuous tests

Scrape Engine Testing

The scrape engine requires testing of HTTP operations, parsing logic, service discovery, and error handling. Testing focuses on isolation using HTTP test servers and mock service discovery backends to create deterministic test conditions.

Test Category	Scope	Key Validations	Test Techniques
HTTP Scraping	Target endpoint communication	Timeout handling, response parsing, error recovery	HTTP test servers with controllable responses
Format Parsing	Prometheus exposition format	Correct metric extraction, malformed input handling	Corpus-based testing with valid/invalid samples
Service Discovery	Target discovery and updates	Dynamic target list updates, health tracking	Mock discovery backends with scripted changes
Concurrency Control	Parallel scraping operations	No resource leaks, proper cleanup, backpressure handling	Load testing with many concurrent targets
Target Health	Health state transitions	Accurate failure detection, recovery detection	Fault injection with network simulation

HTTP Transport Testing: HTTP scraping tests use Go's `httptest` package to create controllable HTTP servers that can simulate various response conditions. Test servers can introduce delays to validate timeout handling, return malformed responses to test error recovery, and simulate network failures to verify retry logic. Connection pooling and resource cleanup testing ensures that scraping doesn't leak HTTP connections or goroutines.

Exposition Format Parsing: Parser testing validates correct extraction of metrics from Prometheus exposition format text. A comprehensive test corpus includes well-formed metrics, malformed input that should be rejected, edge cases like empty values or unusual characters, and large payloads that test memory usage. Fuzzing techniques generate random input to discover parsing crashes or hangs.

Service Discovery Testing: Service discovery testing uses mock backends that implement the `TargetDiscoverer` interface to provide scripted target updates. Tests validate that target additions, removals, and label updates are processed correctly and that the scrape engine adapts its scraping schedule accordingly. Performance testing measures the time to detect and process large target set changes.

Concurrent Scraping Testing: Concurrency testing spawns hundreds of mock targets and validates that the scrape engine can handle the load without resource exhaustion. Tests monitor goroutine counts, memory usage, and file descriptor usage to detect leaks. Backpressure testing validates that the scrape engine gracefully handles storage slowdowns without dropping data or consuming excessive memory.

⚠ Pitfall: Testing Against Real HTTP Endpoints Using real HTTP endpoints in component tests makes tests non-deterministic and dependent on external services. The tests become slow, flaky, and may fail due to network issues unrelated to the code being tested. Instead, use `httptest.Server` to create local test servers that can simulate any response pattern needed for testing. This makes tests fast, reliable, and completely controllable.

Time Series Storage Testing

Storage engine testing focuses on data durability, compression correctness, query performance, and recovery behavior. Testing emphasizes validation of complex concurrent scenarios where reads and writes happen simultaneously.

Test Category	Scope	Key Validations	Test Techniques
Data Durability	Write-ahead logging	WAL recovery, corruption detection, consistency	Process restart simulation, disk failure injection
Compression Accuracy	Gorilla compression algorithm	Lossless compression, decompression accuracy	Statistical validation with real-world data patterns
Query Performance	Series selection and retrieval	Index efficiency, memory usage, query timeouts	Benchmark testing with varying cardinality levels
Concurrent Access	Read/write coordination	Data consistency, no corruption, fair scheduling	High-concurrency stress testing
Storage Limits	Disk usage and retention	Cleanup effectiveness, emergency procedures	Disk space simulation with controlled limits

Write-Ahead Log Testing: WAL testing validates that all writes are durably recorded before being applied to the main storage. Recovery testing simulates process crashes at various points during write operations and verifies that recovery produces consistent state. Corruption detection testing deliberately corrupts WAL files and validates that the recovery process detects and handles the corruption appropriately.

Compression Algorithm Testing: Gorilla compression testing validates that the delta-of-delta timestamp encoding and XOR value encoding produce bit-accurate results. Test data includes regular timestamp intervals, irregular intervals, and pathological cases like duplicate timestamps or wildly varying values. Decompression testing ensures that compressed data can be accurately reconstructed without any data loss.

Index Performance Testing: Index testing validates that the inverted indexes provide efficient series lookup across different cardinality levels. Benchmark tests measure lookup performance as the number of series and labels grows. Memory usage testing tracks index memory consumption and validates that it remains within expected bounds as data volume increases.

Concurrent Access Testing: Concurrency testing spawns multiple goroutines performing simultaneous reads and writes and validates that the data remains consistent. Testing uses techniques like read-after-write validation to ensure that writes are immediately visible to subsequent reads. Lock contention testing measures performance under high concurrency and validates that the system provides fair access to both readers and writers.

⚠ Pitfall: Insufficient WAL Recovery Testing Many storage systems fail catastrophically because WAL recovery wasn't tested thoroughly. It's not enough to test that recovery works when the process shuts down cleanly—you must test recovery after crashes during critical operations like WAL rotation, index updates, and chunk compression. Use tools like `kill -9` or controlled process termination to simulate realistic crash scenarios.

Query Engine Testing

Query engine testing validates PromQL parsing correctness, execution accuracy, and resource management. Testing emphasizes mathematical accuracy of aggregations and proper handling of time-based operations.

Test Category	Scope	Key Validations	Test Techniques
PromQL Parsing	Expression parsing accuracy	AST correctness, syntax error handling, operator precedence	Grammar-based test generation
Label Matching	Series filtering logic	Regex performance, exact matching, negative selectors	Large-scale cardinality testing
Aggregation Math	Mathematical correctness	Sum, average, quantile accuracy, grouping behavior	Statistical validation against known results
Range Queries	Time-based operations	Interpolation accuracy, staleness handling, step alignment	Time series simulation with known patterns
Resource Limits	Memory and execution time	Query timeouts, memory limits, complexity estimation	Adversarial query construction

PromQL Parser Testing: Parser testing validates that PromQL expressions are correctly converted into executable abstract syntax trees. Test cases include all supported operators, functions, and syntax constructs. Error handling testing ensures that invalid queries produce helpful error messages with specific locations of syntax problems. Precedence testing validates that complex expressions with multiple operators are parsed with correct operator precedence.

Label Selector Testing: Label matching testing validates that label selectors correctly filter time series based on label values. Performance testing measures regex matching speed with complex patterns and large label sets. Edge case testing validates behavior with empty labels, special characters, and Unicode content in label values.

Mathematical Accuracy Testing: Aggregation testing validates that mathematical operations produce numerically accurate results. Statistical testing compares query results against independently computed expected values. Edge case testing validates handling of special floating-point values like NaN, infinity, and denormal numbers. Precision testing ensures that aggregations maintain accuracy even with large numbers of input values.

Time Range Query Testing: Range query testing validates that queries over time windows produce correct results with proper interpolation and staleness handling. Test data includes regular and irregular timestamp patterns. Boundary condition testing validates behavior at query start and end times, including proper handling of data points that fall exactly on boundaries.

Decision: Statistical Validation for Mathematical Operations

- **Context:** Aggregation functions must produce mathematically correct results but floating-point arithmetic introduces precision issues
- **Options Considered:** Exact arithmetic libraries, statistical validation with tolerance, property-based testing
- **Decision:** Statistical validation with appropriate tolerance levels for floating-point operations
- **Rationale:** Exact arithmetic is too slow for production use, but we must validate that results are within acceptable precision bounds. Different operations have different precision characteristics that require specific tolerance levels
- **Consequences:** Tests must specify appropriate tolerance levels for each operation type. Enables fast floating-point operations while catching significant mathematical errors

End-to-End Testing

End-to-end testing validates complete workflows from metrics scraping through storage to querying, using realistic data patterns and operational scenarios. These tests verify that components work together correctly and that the system provides the expected user experience.

Complete Scrape-to-Query Workflows

End-to-end workflow testing validates the entire pipeline from target discovery through data storage to query execution. These tests use real HTTP servers exposing Prometheus-format metrics and execute actual PromQL queries against the stored data.

Workflow Test Scenarios:

Scenario	Description	Validation Points	Success Criteria
Basic Scraping	Single target with simple metrics	Target discovery, scraping, storage, basic queries	All metrics stored correctly, queries return expected values
Service Discovery	Dynamic target addition/removal	Target list updates, scraping adaptation, data consistency	New targets automatically discovered and scraped
High Cardinality	Many series with complex labels	Memory usage, query performance, storage efficiency	System remains responsive under high cardinality load
Long-Running	Extended operation over hours/days	Data retention, resource stability, query consistency	No memory leaks, stable performance over time
Failure Recovery	Network failures and target outages	Error handling, recovery behavior, data consistency	Graceful degradation and recovery without data loss

Basic Scraping Workflow Testing: Tests start with a simple scenario involving one target exposing counter, gauge, and histogram metrics. The test validates that metrics are discovered, scraped at the correct interval, stored durably, and can be queried accurately. Timing validation ensures that scrapes happen at the expected intervals and that query results reflect the most recent scraped data.

Service Discovery Integration Testing: Tests validate dynamic target management by starting with an empty target list and then adding targets through service discovery. The test monitors how quickly new targets are detected and begin being scraped. Target removal testing validates that removed targets stop being scraped and that their data remains queryable for the retention period.

High Cardinality Stress Testing: Tests create scenarios with thousands of unique time series by using high-cardinality labels like user IDs or request paths. The test validates that the system can handle the load without excessive memory usage or query performance degradation. Resource monitoring throughout the test ensures that memory growth remains bounded and that garbage collection remains effective.

Long-Running Stability Testing: Extended tests run for hours or days to validate system stability under continuous operation. Tests monitor memory usage, file descriptor counts, and query performance over time to detect resource leaks or performance degradation. Data consistency checks validate that older data remains accurate and queryable throughout the extended run.

⚠ Pitfall: Inadequate Long-Running Test Duration Many systems appear stable in short tests but develop problems over longer periods due to resource leaks or gradual performance degradation. End-to-end tests should run for at least several hours, preferably overnight, to detect these issues. Monitor system resources throughout the test and validate that performance remains stable over the entire duration.

Multi-Component Integration Scenarios

Integration testing validates the interfaces and communication patterns between major system components. These tests focus on data flow, error propagation, and coordination between components.

Integration Test Categories:

Integration	Components	Focus Areas	Key Validations
Scrape-Storage	ScrapeEngine, StorageEngine	Data ingestion pipeline	Sample batching, backpressure handling, durability
Storage-Query	StorageEngine, QueryEngine	Data retrieval pipeline	Index usage, query optimization, result accuracy
Cross-Component Error Handling	All components	Error propagation and recovery	Graceful degradation, error isolation, recovery coordination
Resource Coordination	All components	Resource management	Memory limits, concurrency control, fair resource allocation

Scrape-Storage Integration: Tests validate the data flow from scraping to storage, focusing on the sample ingestion pipeline. Backpressure testing validates that storage slowdowns are properly communicated back to the scrape engine to prevent memory exhaustion. Batching tests verify that samples are efficiently grouped for storage operations. Durability testing validates that scraped data survives process restarts and storage failures.

Storage-Query Integration: Tests validate that queries can efficiently retrieve data stored by the scrape engine. Index usage testing ensures that label selectors use indexes effectively rather than scanning all series. Query optimization testing validates that the query engine chooses efficient execution plans for complex queries. Cache coherency testing ensures that query results reflect recent writes from the scrape engine.

Error Handling Coordination: Integration error testing validates that component failures are properly isolated and don't cascade to other components. Tests simulate various failure modes like storage disk full, query timeouts, and scrape target failures. Recovery coordination testing validates that components restart in the correct order and re-establish communication properly.

Resource Management Integration: Resource coordination testing validates that components cooperate effectively in resource usage. Memory limit testing validates that components respect overall system memory limits rather than competing for resources. Concurrency testing validates that components don't create excessive goroutines or other concurrent resources that could overwhelm the system.

Data Consistency and Accuracy Validation

Data accuracy testing validates that the complete system preserves data integrity throughout the scrape-store-query pipeline. These tests focus on numerical accuracy, timestamp preservation, and metadata consistency.

Accuracy Validation Approaches:

Validation Type	Methodology	Test Data	Success Criteria
Numerical Accuracy	Compare scraped values to query results	Known metric values from test targets	Zero data loss, accurate aggregations
Timestamp Preservation	Validate timestamp consistency through pipeline	Synthetic time series with known timestamps	Timestamps preserved through storage and retrieval
Metadata Consistency	Verify label and metric name preservation	Complex label combinations	All metadata preserved accurately
Aggregation Accuracy	Compare aggregation results to mathematical truth	Statistical distributions with known properties	Aggregations match expected mathematical results

End-to-End Numerical Accuracy: Tests expose known metric values through test HTTP endpoints and validate that queries return identical values. Counter testing validates that increments are preserved accurately without any loss. Histogram testing validates that bucket counts and sum/count values are preserved correctly. Gauge testing validates that the most recent values are returned accurately by queries.

Timestamp Precision Testing: Tests validate that timestamps are preserved accurately throughout the system. Test targets expose metrics with known timestamps and queries validate that the timestamps are stored and retrieved without modification. Time zone testing validates that UTC timestamps are handled consistently regardless of the local system timezone.

Label and Metadata Preservation: Tests validate that metric names, label names, label values, and metric metadata are preserved accurately throughout the system. Unicode testing validates proper handling of international characters in labels. Special character testing validates handling of characters that might have special meaning in various system components.

Mathematical Aggregation Validation: Tests validate that PromQL aggregations produce mathematically correct results. Statistical testing uses metrics with known distributions and validates that aggregation functions like `sum()`, `avg()`, and `quantile()` produce results within acceptable precision bounds. Cross-validation testing compares query results against independently computed expected values.

Decision: Reference Implementation Validation

- **Context:** End-to-end accuracy testing needs ground truth to compare against actual system results
- **Options Considered:** Hand-computed expected results, reference implementation, statistical validation
- **Decision:** Hybrid approach using reference implementation for complex scenarios and hand-computed results for simple cases
- **Rationale:** Hand computation works for simple scenarios but becomes impractical for complex queries. A reference implementation provides automated ground truth generation but may have its own bugs
- **Consequences:** Requires maintaining a separate reference implementation but enables comprehensive accuracy validation

Performance Validation

Performance validation ensures that the system meets scalability, latency, and resource usage requirements under realistic load conditions. Testing focuses on identifying bottlenecks and validating that performance characteristics meet operational requirements.

Load Testing and Capacity Planning

Load testing validates system behavior under realistic operational loads and identifies performance bottlenecks before they impact production usage. Tests progressively increase load while monitoring system behavior and resource usage.

Load Testing Dimensions:

Load Dimension	Test Parameters	Scaling Range	Key Metrics
Scraping Scale	Number of targets and metrics per target	10 to 10,000 targets	Scrape completion rate, memory usage, CPU usage
Storage Throughput	Samples per second ingestion rate	1K to 1M samples/sec	Write latency, compression ratio, disk usage
Query Concurrency	Simultaneous query execution	1 to 1,000 concurrent queries	Query response time, memory per query, throughput
Data Volume	Total time series and time range	1K to 10M series over weeks	Index performance, query latency, storage efficiency

Scraping Load Testing: Scraping load tests progressively increase the number of scrape targets while monitoring scrape completion rates and resource usage. Tests validate that the scrape engine can handle the target number of endpoints within the configured scrape intervals. Memory usage monitoring ensures that high target counts don't cause memory exhaustion. Network resource testing validates that the system doesn't exhaust network connections or file descriptors.

Storage Throughput Testing: Storage load tests measure how many samples per second the storage engine can handle sustainably. Tests generate synthetic time series data at various ingestion rates while monitoring write latency and resource usage. Compression efficiency testing validates that storage space usage remains within expected

bounds as ingestion rate increases. WAL performance testing ensures that write-ahead logging doesn't become a bottleneck at high ingestion rates.

Query Concurrency Testing: Query load tests execute multiple simultaneous queries while measuring response times and resource usage. Tests include a mix of instant queries, range queries, and complex aggregations to simulate realistic query patterns. Memory usage per query is monitored to validate that concurrent queries don't cause memory exhaustion. Query isolation testing ensures that expensive queries don't significantly impact the performance of simple queries.

Long-Term Data Volume Testing: Volume testing validates performance with large amounts of historical data. Tests create months or years of synthetic time series data and measure how query performance changes as data volume increases. Index scalability testing validates that label lookup performance remains acceptable as the number of unique series grows. Retention policy testing validates that data cleanup operations don't significantly impact query performance.

Bottleneck Identification and Resource Monitoring

Performance testing includes comprehensive monitoring to identify bottlenecks and validate that resources are used efficiently. Monitoring focuses on the critical resources that typically limit performance in metrics systems.

Resource Monitoring Categories:

Resource Category	Key Metrics	Monitoring Tools	Bottleneck Indicators
Memory Usage	Heap size, GC frequency, allocation rate	Runtime profiling, memory profilers	High GC overhead, allocation spikes
CPU Utilization	CPU usage per component, goroutine counts	CPU profiling, runtime metrics	High CPU usage, goroutine leaks
Disk I/O	Read/write throughput, latency, queue depth	I/O monitoring, disk stats	High I/O wait, storage latency spikes
Network I/O	Connection counts, bandwidth usage, error rates	Network monitoring, connection pooling stats	Connection exhaustion, bandwidth saturation

Memory Performance Analysis: Memory monitoring tracks heap usage, garbage collection frequency, and allocation patterns throughout performance tests. Profiling identifies which components and operations consume the most memory. Memory leak detection validates that memory usage remains bounded during extended operation. Gorilla compression efficiency is validated by measuring the compression ratio achieved on realistic data patterns.

CPU Performance Analysis: CPU profiling identifies which operations consume the most processing time. Goroutine monitoring ensures that the system doesn't create excessive concurrent operations. Lock contention analysis identifies synchronization bottlenecks that prevent efficient CPU utilization. Query parsing and execution profiling validates that PromQL operations perform efficiently.

Disk I/O Performance Analysis: Disk monitoring measures read and write throughput during various operations. WAL write performance is critical for ingestion throughput. Index read performance affects query latency. Storage

compaction operations are monitored to ensure they don't interfere with normal operations. Disk space usage is tracked to validate that retention policies work effectively.

Network I/O Performance Analysis: Network monitoring tracks HTTP connection usage during scraping operations. Connection pooling efficiency is validated to ensure that scraping doesn't create excessive network overhead. DNS resolution performance is monitored since service discovery can generate significant DNS traffic. Network timeout and retry behavior is validated under various network conditions.

Scalability Requirements Verification

Scalability testing validates that the system can handle the target operational scale defined in the system requirements. Tests specifically target the scalability limits identified in the goals and non-goals section.

Scalability Test Targets:

Scalability Requirement	Target Scale	Test Approach	Acceptance Criteria
Concurrent Targets	1,000 scrape targets	Progressive load increase	All targets scraped within interval
Series Cardinality	1M unique time series	High-cardinality label generation	Query performance remains acceptable
Query Throughput	100 queries/second	Concurrent query execution	Mean response time under 1 second
Data Retention	30 days of historical data	Long-running data accumulation	Storage usage within 2x of raw data size

Target Scale Verification: Scalability tests progressively increase system load until reaching the target scale requirements. Each test level is sustained for sufficient time to validate stability at that scale. Resource usage is monitored throughout scaling tests to identify when resources become constrained. Performance characteristics like query latency and scrape completion rates are tracked to validate that they remain within acceptable bounds at target scale.

Cardinality Limit Testing: High-cardinality testing generates time series with label combinations that approach the target cardinality limits. Memory usage is carefully monitored to validate that the system can handle high cardinality without excessive memory consumption. Query performance testing validates that label selectors remain efficient even with high cardinality. Index scalability is validated by measuring lookup performance as cardinality increases.

Throughput Sustainability Testing: Throughput tests validate that the system can sustain target throughput rates over extended periods without performance degradation. Load balancing effectiveness is validated by measuring how evenly work is distributed across system resources. Backpressure handling is tested by temporarily constraining resources and validating that the system handles the constraint gracefully without data loss.

⚠ Pitfall: Testing Only Peak Performance Many performance tests only measure short-term peak performance rather than sustained throughput over realistic time periods. Real systems must handle continuous load for hours or days, not just brief spikes. Performance tests should sustain target loads for at least 30 minutes to validate that resource usage remains stable and that performance doesn't degrade over time.

Milestone Verification

Milestone verification provides concrete checkpoints to validate successful completion of each development phase. Each milestone includes specific behavioral verification, performance benchmarks, and integration validation steps.

Milestone 1: Metrics Data Model Verification

Milestone 1 verification validates that the metrics data model correctly implements counter, gauge, and histogram semantics with proper labeling and cardinality control.

Verification Checklist:

Verification Category	Test Procedures	Expected Results	Validation Commands
Metric Type Behavior	Create and manipulate each metric type	Counters only increase, gauges accept any value, histograms distribute correctly	Unit test suite execution
Label Functionality	Create metrics with various label combinations	Labels attached correctly, cardinality tracking works	Label validation test execution
Thread Safety	Concurrent access to metric instances	No race conditions, consistent values	Race detector test execution
Cardinality Control	Attempt to create high-cardinality combinations	Limits enforced, memory usage bounded	Cardinality stress test execution

Counter Behavior Verification: Counter testing validates that increment operations work correctly and that attempts to decrease values are rejected. Overflow testing validates behavior when counter values approach floating-point limits. Thread safety testing uses concurrent goroutines to validate that increments are atomic and no updates are lost.

```
# Example verification commands
go test -race ./internal/metrics/counter_test.go
go test -bench=BenchmarkCounterIncrement ./internal/metrics/
```

BASH

Gauge Behavior Verification: Gauge testing validates that both set and add operations work with positive and negative values. Special value testing validates handling of infinity, NaN, and zero values. Concurrent testing validates that rapid updates don't corrupt the stored value.

Histogram Behavior Verification: Histogram testing validates that observations are assigned to correct buckets and that sum and count fields are maintained accurately. Distribution testing uses known statistical distributions and validates that histogram buckets correctly represent the distribution shape.

Label System Verification: Label validation testing creates metrics with various label combinations and validates that labels are stored and retrieved correctly. Cardinality testing attempts to create high-cardinality label combinations and validates that appropriate limits are enforced. Unicode testing validates that international characters in label names and values are handled correctly.

Milestone 2: Scrape Engine Verification

Milestone 2 verification validates that the scrape engine can discover targets, scrape metrics via HTTP, and handle various failure modes gracefully.

Verification Checklist:

Verification Category	Test Procedures	Expected Results	Validation Commands
Target Discovery	Configure static and dynamic targets	Targets discovered and added to scrape list	Discovery integration test
HTTP Scraping	Scrape from test HTTP endpoints	Metrics successfully retrieved and parsed	HTTP scraping test
Error Handling	Simulate network failures and timeouts	Graceful failure handling, retry logic works	Fault injection test
Service Discovery	Add/remove targets dynamically	Target list updates automatically	Service discovery test

Target Discovery Verification: Discovery testing configures both static targets and mock service discovery and validates that all targets are discovered and added to the scrape schedule. Update testing validates that target list changes are processed correctly. Health tracking testing validates that target health status is maintained accurately.

```
# Example verification commands
go test ./internal/scrapers/discovery_test.go
go test -timeout=30s ./internal/scrapers/integration_test.go
```

BASH

HTTP Scraping Verification: HTTP testing uses `httptest.Server` to create controllable test endpoints that expose Prometheus-format metrics. Parsing verification validates that various metric types are correctly extracted from the HTTP response. Timeout testing validates that scrapes that exceed the timeout are cancelled properly.

Error Handling Verification: Fault injection testing simulates various error conditions including network timeouts, HTTP errors, malformed responses, and DNS failures. Recovery testing validates that the scrape engine recovers correctly when failed targets become available again. Circuit breaker testing validates that consistently failing targets are temporarily bypassed.

Service Discovery Integration Verification: Service discovery testing uses mock discovery backends to simulate target additions, removals, and metadata updates. Latency testing measures how quickly target changes are detected and processed. Consistency testing validates that target metadata is correctly propagated from service discovery to scraping operations.

Milestone 3: Time Series Storage Verification

Milestone 3 verification validates that the storage engine correctly stores time series data with compression, provides durable persistence, and supports efficient queries.

Verification Checklist:

Verification Category	Test Procedures	Expected Results	Validation Commands
Data Persistence	Write data and restart process	All data recovered correctly after restart	WAL recovery test
Compression Efficiency	Store various data patterns	Compression ratios within expected range	Compression benchmark test
Query Performance	Execute queries on stored data	Query response times within limits	Query performance test
Concurrent Access	Simultaneous reads and writes	No data corruption, consistent results	Concurrency test

Data Persistence Verification: Persistence testing writes time series data and then simulates process crashes at various points. Recovery testing validates that all durably committed data is correctly restored after restart. WAL integrity testing validates that write-ahead log recovery produces consistent results.

```
# Example verification commands
go test ./internal/storage/wal_test.go
go test -bench=BenchmarkStorageWrite ./internal/storage/
```

BASH

Compression Efficiency Verification: Compression testing stores various time series patterns including regular intervals, irregular intervals, and pathological cases. Compression ratio measurement validates that Gorilla compression achieves expected space savings. Decompression accuracy testing validates that compressed data can be reconstructed without any loss.

Query Performance Verification: Query testing executes various query patterns against stored data and measures response times. Index efficiency testing validates that label selectors use indexes effectively. Memory usage testing validates that queries don't consume excessive memory even with large result sets.

Concurrent Access Verification: Concurrency testing spawns multiple goroutines performing simultaneous reads and writes. Data consistency testing validates that concurrent operations don't corrupt stored data. Lock contention testing measures performance under high concurrency and validates that the system provides fair access to resources.

Milestone 4: Query Engine Verification

Milestone 4 verification validates that the query engine correctly parses PromQL expressions, executes queries accurately, and provides expected aggregation functionality.

Verification Checklist:

Verification Category	Test Procedures	Expected Results	Validation Commands
PromQL Parsing	Parse various PromQL expressions	Correct AST generation, proper error messages	Parser test suite
Query Execution	Execute instant and range queries	Accurate results, proper error handling	Query execution test
Aggregation Functions	Test sum, avg, max, min, count operations	Mathematically correct aggregation results	Aggregation test suite
Label Matching	Test exact, regex, and inequality selectors	Correct series filtering, efficient execution	Label selector test

PromQL Parsing Verification: Parser testing validates that various PromQL expressions are correctly converted to abstract syntax trees. Error handling testing validates that invalid expressions produce helpful error messages. Precedence testing validates that complex expressions are parsed with correct operator precedence.

```
# Example verification commands
go test ./internal/query/parser_test.go
go test -bench=BenchmarkQueryExecution ./internal/query/
```

BASH

Query Execution Verification: Query execution testing validates that both instant and range queries produce accurate results. Time range testing validates that queries over various time windows work correctly. Interpolation testing validates that query results are correctly interpolated between actual data points.

Aggregation Function Verification: Mathematical testing validates that aggregation functions produce numerically correct results. Statistical testing uses known distributions and validates that aggregation results match expected mathematical values. Edge case testing validates handling of special values like NaN and infinity in aggregations.

Label Matching Verification: Label selector testing validates that exact matching, regex matching, and inequality operators correctly filter time series. Performance testing validates that complex label selectors execute efficiently. Cardinality testing validates that label selectors work correctly even with high-cardinality label sets.

Implementation Guidance

This implementation guidance provides practical approaches for building a comprehensive testing framework that validates all aspects of the metrics collection system.

Technology Recommendations

Testing Category	Simple Option	Advanced Option
Unit Testing Framework	Go standard testing package with testify assertions	Ginkgo BDD framework with Gomega matchers
HTTP Testing	httpptest package for mock servers	WireMock or similar for complex HTTP scenarios
Property-Based Testing	rapid testing library	gopter property-based testing framework
Load Testing	Simple goroutine-based load generation	k6 or Artillery for sophisticated load patterns
Performance Profiling	Go built-in pprof	Continuous profiling with pprof integration
Test Data Management	In-memory test fixtures	Testcontainers for external dependencies

Recommended Project Structure

The testing code should be organized to support both component-level and integration testing with clear separation of concerns:

```
project-root/
  internal/
    metrics/
      counter.go
      counter_test.go          ← Unit tests for counter behavior
      gauge_test.go            ← Unit tests for gauge behavior
      histogram_test.go        ← Unit tests for histogram behavior
      labels_test.go           ← Unit tests for label validation
    scraper/
      scraper.go
      scraper_test.go          ← Unit tests for scraping logic
      discovery_test.go         ← Unit tests for service discovery
      integration_test.go       ← Integration tests with HTTP servers
    storage/
      storage.go
      storage_test.go          ← Unit tests for storage operations
      wal_test.go               ← Unit tests for write-ahead log
      compression_test.go       ← Unit tests for compression algorithms
    query/
      parser_test.go            ← Unit tests for PromQL parsing
      executor_test.go          ← Unit tests for query execution
  test/
    integration/
      end_to_end_test.go        ← Complete workflow tests
      performance_test.go       ← Load and performance tests
    fixtures/
      metrics_samples.txt       ← Sample Prometheus format data
      test_queries.promql        ← Sample PromQL queries for testing
    helpers/
      test_server.go             ← HTTP test server utilities
      data_generator.go          ← Synthetic data generation
```

Testing Infrastructure Starter Code

Here's a complete testing infrastructure that provides the foundation for comprehensive testing:

```
package testhelpers

import (
    "context"
    "fmt"
    "math/rand"
    "net/http"
    "net/http/httpptest"
    "strings"
    "sync"
    "testing"
    "time"
)

// MockHTTPPTarget provides a controllable HTTP endpoint for scraping tests

type MockHTTPPTarget struct {
    server      *httpptest.Server
    metrics     []string
    delay       time.Duration
    errorRate   float64
    mu          sync.RWMutex
}

// NewMockHTTPPTarget creates a new mock target with configurable behavior

func NewMockHTTPPTarget() *MockHTTPPTarget {
    target := &MockHTTPPTarget{
        metrics: []string{
            "test_counter 42",
            "test_gauge 3.14",
            "test_histogram_bucket{le=\"1.0\"} 10",
        }
    }
}
```

```
        },

    }

    target.server = httptest.NewServer(http.HandlerFunc(target.handleMetrics))

    return target
}

// URL returns the target's HTTP URL for scraping

func (t *MockHTTPTarget) URL() string {

    return t.server.URL + "/metrics"

}

// SetMetrics updates the metrics exposed by this target

func (t *MockHTTPTarget) SetMetrics(metrics []string) {

    t.mu.Lock()

    defer t.mu.Unlock()

    t.metrics = metrics

}

// SetDelay configures response delay for timeout testing

func (t *MockHTTPTarget) SetDelay(delay time.Duration) {

    t.mu.Lock()

    defer t.mu.Unlock()

    t.delay = delay

}

// SetErrorRate configures the probability of HTTP errors

func (t *MockHTTPTarget) SetErrorRate(rate float64) {

    t.mu.Lock()

    defer t.mu.Unlock()
```

```
t.errorRate = rate

}

func (t *MockHTTPTarget) handleMetrics(w http.ResponseWriter, r *http.Request) {

    t.mu.RLock()

    delay := t.delay

    errorRate := t.errorRate

    metrics := t.metrics

    t.mu.RUnlock()

    // Simulate network delay

    if delay > 0 {

        time.Sleep(delay)

    }

    // Simulate random errors

    if rand.Float64() < errorRate {

        http.Error(w, "Simulated server error", http.StatusInternalServerError)

        return

    }

    w.Header().Set("Content-Type", "text/plain")

    for _, metric := range metrics {

        fmt.Fprintln(w, metric)

    }

}

// Close shuts down the mock target server

func (t *MockHTTPTarget) Close() {
```

```
t.server.Close()

}

// TimeSeriesGenerator creates synthetic time series data for testing

type TimeSeriesGenerator struct {

    rand *rand.Rand
}

// NewTimeSeriesGenerator creates a generator with a fixed seed for reproducible tests

func NewTimeSeriesGenerator(seed int64) *TimeSeriesGenerator {

    return &TimeSeriesGenerator{
        rand: rand.New(rand.NewSource(seed)),
    }
}

// GenerateCounterSeries creates a monotonically increasing counter series

func (g *TimeSeriesGenerator) GenerateCounterSeries(name string, labels map[string]string,
    start time.Time, interval time.Duration, count int) []Sample {

    samples := make([]Sample, count)
    value := float64(0)

    for i := 0; i < count; i++ {
        // Counters can only increase
        increment := g.rand.Float64() * 10
        value += increment

        samples[i] = Sample{
            Timestamp: start.Add(time.Duration(i) * interval),
            Value:     value,
        }
    }
}
```

```
    }

}

return samples
}

// GenerateGaugeSeries creates a gauge series with random walk behavior

func (g *TimeSeriesGenerator) GenerateGaugeSeries(name string, labels map[string]string,
    start time.Time, interval time.Duration, count int) []Sample {

    samples := make([]Sample, count)

    value := g.rand.Float64() * 100

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

        // Gauges can increase or decrease

        change := (g.rand.Float64() - 0.5) * 20

        value += change

        samples[i] = Sample{
            Timestamp: start.Add(time.Duration(i) * interval),
            Value:     value,
        }
    }

    return samples
}

// PerformanceMonitor tracks resource usage during tests

type PerformanceMonitor struct {
```

```
startTime      time.Time

startMemory    uint64

measurements   []ResourceMeasurement

mu            sync.Mutex

}

type ResourceMeasurement struct {

    Timestamp      time.Time

    MemoryUsage   uint64

    GoroutineCount int

    CPUUsage       float64

}

// NewPerformanceMonitor creates a monitor for tracking test performance

func NewPerformanceMonitor() *PerformanceMonitor {

    return &PerformanceMonitor{

        startTime: time.Now(),

        measurements: make([]ResourceMeasurement, 0),

    }

}

// StartMonitoring begins periodic resource measurement

func (m *PerformanceMonitor) StartMonitoring(ctx context.Context, interval time.Duration) {

    ticker := time.NewTicker(interval)

    defer ticker.Stop()

    for {

        select {

        case <-ctx.Done():

            return
```

```
    case <-ticker.C:

        m.takeMeasurement()

    }

}

func (m *PerformanceMonitor) takeMeasurement() {

    // TODO: Implement actual resource measurement

    // This would use runtime.ReadMemStats(), runtime.NumGoroutine(), etc.

    // For now, provide placeholder implementation


    m.mu.Lock()

    defer m.mu.Unlock()




measurement := ResourceMeasurement{

    Timestamp:      time.Now(),

    MemoryUsage:    0, // TODO: Get actual memory usage

    GoroutineCount: 0, // TODO: Get actual goroutine count

    CPUUsage:       0, // TODO: Get actual CPU usage

}

m.measurements = append(m.measurements, measurement)

}

// GetSummary returns performance statistics from monitoring

func (m *PerformanceMonitor) GetSummary() PerformanceSummary {

    m.mu.Lock()

    defer m.mu.Unlock()
```

```
if len(m.measurements) == 0 {

    return PerformanceSummary{}

}

// TODO: Calculate actual performance statistics

return PerformanceSummary{

    Duration:      time.Since(m.startTime),

    PeakMemory:    0, // TODO: Calculate from measurements

    AvgGoroutines: 0, // TODO: Calculate from measurements

    MaxGoroutines: 0, // TODO: Calculate from measurements

}

}

type PerformanceSummary struct {

    Duration      time.Duration

    PeakMemory    uint64

    AvgGoroutines int

    MaxGoroutines int

}
```

Core Testing Logic Skeletons

Here are the essential test function signatures with detailed TODO comments for implementation:

GO

```
package metrics_test

import (
    "context"
    "sync"
    "testing"
    "time"
)

// TestCounterBehavior validates counter semantic correctness

func TestCounterBehavior(t *testing.T) {
    // TODO 1: Create a new counter instance

    // TODO 2: Verify initial value is 0

    // TODO 3: Call Inc() and verify value increases to 1

    // TODO 4: Call Add(5) and verify value increases to 6

    // TODO 5: Attempt Add(-1) and verify it returns an error

    // TODO 6: Verify final value is still 6 (no decrease occurred)

}

// TestCounterConcurrency validates thread-safe counter operations

func TestCounterConcurrency(t *testing.T) {
    // TODO 1: Create a new counter instance

    // TODO 2: Start 10 goroutines, each incrementing counter 100 times

    // TODO 3: Use sync.WaitGroup to wait for all goroutines to complete

    // TODO 4: Verify final counter value equals 1000 (10 * 100)

    // TODO 5: Run test with -race flag to detect race conditions

    // Hint: Use atomic operations or mutex in counter implementation

}

// TestHistogramDistribution validates histogram bucket assignment
```

```
func TestHistogramDistribution(t *testing.T) {

    buckets := []float64{1.0, 5.0, 10.0, math.Inf(1)}

    // TODO 1: Create histogram with specified bucket boundaries

    // TODO 2: Observe values: 0.5, 2.0, 7.0, 15.0

    // TODO 3: Verify bucket counts: [1, 1, 1, 1]

    // TODO 4: Verify total count equals 4

    // TODO 5: Verify sum equals 24.5 (0.5 + 2.0 + 7.0 + 15.0)

    // Hint: Values should be assigned to first bucket where value <= bucket

}

// TestLabelCardinality validates cardinality explosion prevention

func TestLabelCardinality(t *testing.T) {

    tracker := NewCardinalityTracker(maxSeries = 1000)

    // TODO 1: Create metrics with normal cardinality (should succeed)

    // TODO 2: Attempt to create metrics exceeding cardinality limit

    // TODO 3: Verify that excess metrics are rejected with appropriate error

    // TODO 4: Verify that memory usage remains bounded

    // TODO 5: Test cleanup of expired series to free cardinality

    // Hint: Use combination of metric name + label set to identify unique series

}

// TestScrapeTargetDiscovery validates service discovery integration

func TestScrapeTargetDiscovery(t *testing.T) {

    mockDiscovery := NewMockServiceDiscovery()

    scrapeEngine := NewScrapeEngine(config, storage, logger)

    // TODO 1: Start scrape engine with empty target list

    // TODO 2: Add targets via mock service discovery

    // TODO 3: Verify targets are discovered within discovery interval
```

```
// TODO 4: Remove targets via service discovery

// TODO 5: Verify removed targets stop being scraped

// TODO 6: Verify target metadata is correctly propagated

// Hint: Mock service discovery should implement TargetDiscoverer interface

}

// TestScrapeErrorHandler validates graceful failure handling

func TestScrapeErrorHandler(t *testing.T) {

    target := NewMockHTTPTarget()

    defer target.Close()

    // TODO 1: Configure target to return HTTP 500 errors

    // TODO 2: Verify scrape engine records failure and continues

    // TODO 3: Configure target to timeout on requests

    // TODO 4: Verify scrape engine cancels request and marks target down

    // TODO 5: Configure target to return malformed metrics

    // TODO 6: Verify partial success - valid metrics stored, invalid rejected

    // Hint: Use target.SetErrorRate() and target.SetDelay() for fault injection

}

// TestStorageCompression validates Gorilla compression accuracy

func TestStorageCompression(t *testing.T) {

    // TODO 1: Generate time series with regular 15-second intervals

    // TODO 2: Compress samples using GorillaCompressor

    // TODO 3: Decompress and verify all samples match exactly

    // TODO 4: Measure compression ratio (should be < 2 bytes/sample)

    // TODO 5: Test with irregular timestamps and verify accuracy

    // TODO 6: Test with pathological data (constant values, huge jumps)

    // Hint: Gorilla compression works best with regular intervals and smooth changes
```

```
}

// TestStorageRecovery validates WAL recovery after crash

func TestStorageRecovery(t *testing.T) {

    tempDir := t.TempDir()

    // TODO 1: Create storage engine and write test data

    // TODO 2: Simulate crash by closing storage without clean shutdown

    // TODO 3: Create new storage engine instance with same data directory

    // TODO 4: Verify all committed data is recovered correctly

    // TODO 5: Verify uncommitted data in WAL is replayed

    // TODO 6: Test recovery with corrupted WAL entries

    // Hint: WAL should be replayed in order, corrupt entries should be detected

}

// TestQueryParsing validates PromQL expression parsing

func TestQueryParsing(t *testing.T) {

    parser := NewExpressionParser()

    // TODO 1: Parse simple metric selector: `http_requests_total`

    // TODO 2: Verify AST contains MetricSelectorNode with correct name

    // TODO 3: Parse selector with labels: `http_requests_total{method="GET"}`

    // TODO 4: Verify label matchers are parsed correctly

    // TODO 5: Parse complex expression with functions: `rate(http_requests_total[5m])`

    // TODO 6: Parse invalid expressions and verify helpful error messages

    // Hint: Build AST recursively, respect operator precedence

}

// TestQueryExecution validates query result accuracy

func TestQueryExecution(t *testing.T) {
```

```
// TODO 1: Store known time series data in storage engine

// TODO 2: Execute instant query and verify results match expected values

// TODO 3: Execute range query and verify all timestamps are covered

// TODO 4: Test label selector queries with exact and regex matching

// TODO 5: Test aggregation queries (sum, avg, max, min, count)

// TODO 6: Verify mathematical accuracy of aggregation results

// Hint: Use statistical validation with appropriate floating-point tolerance

}

// TestEndToEndWorkflow validates complete scrape-to-query pipeline

func TestEndToEndWorkflow(t *testing.T) {

    // TODO 1: Start mock HTTP target exposing known metrics

    // TODO 2: Configure scrape engine to scrape the target

    // TODO 3: Wait for metrics to be scraped and stored

    // TODO 4: Execute queries against stored metrics

    // TODO 5: Verify query results match original metrics from target

    // TODO 6: Test with target failures and recovery

    // Hint: Use eventually assertions for asynchronous scraping operations

}

// BenchmarkStorageWrite measures storage write performance

func BenchmarkStorageWrite(b *testing.B) {

    storage := NewStorageEngine(config, logger)

    samples := generateTestSamples(1000)

    b.ResetTimer()

    // TODO 1: Measure samples per second write throughput

    // TODO 2: Measure write latency distribution (p50, p95, p99)

    // TODO 3: Measure memory allocation per write operation
```

```

    // TODO 4: Test with varying batch sizes

    // TODO 5: Report compression ratio achieved

    // Hint: Use b.N for iteration count, benchmark multiple scenarios

}

// TestResourceLimits validates system behavior under resource constraints

func TestResourceLimits(t *testing.T) {

    // TODO 1: Configure system with low memory limits

    // TODO 2: Generate high-cardinality metrics to approach limit

    // TODO 3: Verify system applies backpressure before exhausting memory

    // TODO 4: Test query limits with complex queries

    // TODO 5: Verify system remains responsive under resource pressure

    // TODO 6: Test disk space limits and emergency retention

    // Hint: Monitor resource usage throughout test, set aggressive limits

}

```

Milestone Checkpoints

Each milestone completion should be validated with these specific checkpoints:

Milestone 1 Checkpoint - Metrics Data Model:

```

# Run all metrics tests with race detection                                BASH

go test -race ./internal/metrics/...

# Benchmark metric operations performance

go test -bench=. ./internal/metrics/

# Expected output: All tests pass, no race conditions detected

# Benchmark results should show sub-microsecond operation times

```

Milestone 2 Checkpoint - Scrape Engine:

```
# Run scrape engine tests

go test -timeout=30s ./internal/scraper/...

# Run integration test with HTTP targets

go test -run=TestScrapeIntegration ./test/integration/

# Expected: Targets discovered, metrics scraped, HTTP errors handled gracefully
```

Milestone 3 Checkpoint - Time Series Storage:

```
# Test storage with crash recovery

go test -run=TestStorageRecovery ./internal/storage/

# Benchmark write performance

go test -bench=BenchmarkStorageWrite ./internal/storage/

# Expected: Recovery works correctly, write throughput > 10K samples/sec
```

Milestone 4 Checkpoint - Query Engine:

```
# Test PromQL parsing and execution

go test ./internal/query/...

# End-to-end workflow test

go test -run=TestEndToEndWorkflow ./test/integration/

# Expected: PromQL queries execute correctly, results mathematically accurate
```

Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Tests hang indefinitely	Deadlock in concurrent code	Use <code>go test -timeout=30s</code> , check goroutine dumps	Review lock ordering, use channels instead of mutexes where possible
Race detector failures	Unsynchronized access to shared data	Run with <code>-race</code> , examine race reports	Add proper synchronization with mutexes or atomic operations
Memory usage grows unbounded	Memory leaks or missing cleanup	Profile with <code>go test -memprofile=mem.prof</code>	Implement proper cleanup, check for goroutine leaks
Inconsistent test failures	Timing-dependent test logic	Add logging, use deterministic test data	Use mock clocks, add proper synchronization
Compression tests fail	Incorrect Gorilla algorithm implementation	Compare bit-by-bit with reference implementation	Debug delta-of-delta calculation, check XOR logic
Query results incorrect	Mathematical precision issues	Compare with independently computed expected values	Use appropriate floating-point tolerance, check aggregation logic

Debugging Guide

Milestone(s): This section provides systematic debugging approaches for all four milestones: Metrics Data Model (1), Scrape Engine (2), Time Series Storage (3), and Query Engine (4). It helps developers diagnose and fix issues across the entire metrics collection pipeline.

Building a metrics collection system introduces numerous failure modes that can be difficult to diagnose. Unlike traditional CRUD applications where failures are often obvious and immediate, metrics systems fail in subtle ways that may not surface until production load or specific edge cases occur. The distributed nature of scraping, the time-series characteristics of the data, and the complex interactions between compression, indexing, and querying create a debugging landscape that requires systematic approaches and specialized tools.

This section provides a comprehensive guide for diagnosing and fixing issues that commonly arise during development and operation of the metrics collection system. The debugging strategies are organized around observable symptoms rather than internal implementation details, because developers typically encounter problems through external manifestations like slow queries, missing data, or crashed processes.

Symptom-Based Troubleshooting

Mental Model: The Detective's Methodology

Think of debugging a metrics system like investigating a crime scene. You start with observable symptoms (the crime), gather evidence through logs and metrics (witness testimony and physical evidence), form hypotheses about root causes (suspect theories), test those hypotheses with additional investigation (interrogation and forensics), and finally implement fixes (arrest and prosecution). The key is following the evidence systematically rather than jumping to conclusions based on assumptions.

The symptom-based approach works because it mirrors how problems actually manifest in production. A user reports "queries are slow" or "metrics are missing" - they don't report "the Gorilla compressor has a bug in delta calculation." By starting with symptoms and working backward to root causes, we build a systematic investigation process that works regardless of the specific implementation details.

High-Level System Symptoms

These symptoms affect the overall system behavior and typically indicate problems in component coordination or resource exhaustion.

Symptom	Likely Root Causes	Investigation Steps	Resolution Actions
System consumes excessive memory	Label explosion, inefficient compression, query result sets too large	Check cardinality metrics, analyze heap dumps, review query patterns	Implement cardinality limits, fix compression bugs, add query result size limits
Queries return no data despite active scraping	Index corruption, time zone mismatches, retention policy too aggressive	Verify scrape success rates, check timestamp alignment, inspect index consistency	Rebuild indexes, fix timestamp handling, adjust retention settings
System becomes unresponsive under load	Resource exhaustion, deadlocks, inefficient algorithms	Monitor resource usage, check for lock contention, profile CPU usage	Add resource limits, fix deadlocks, optimize hot paths
Data appears and disappears intermittently	Race conditions in storage, inconsistent reads, partial failures	Enable race detector, add consistency checks, review concurrent access patterns	Fix race conditions, add proper locking, implement read consistency
System fails to start after restart	Corrupted WAL, configuration errors, dependency unavailability	Check WAL recovery logs, validate configuration, verify dependencies	Repair or rebuild WAL, fix configuration, ensure dependencies are available

Scraping Engine Symptoms

Scraping problems often manifest as missing or stale data, and can be particularly tricky because network issues create intermittent failures.

Symptom	Likely Root Causes	Investigation Steps	Resolution Actions
Targets show as "down" but are actually healthy	Network connectivity issues, DNS resolution problems, authentication failures	Test manual HTTP requests, check DNS resolution, verify credentials	Fix network configuration, update DNS settings, correct authentication
Scraping succeeds but no metrics are stored	Parsing errors, timestamp issues, storage failures	Check parsing error rates, verify timestamp formats, monitor storage health	Fix parsing logic, standardize timestamp handling, resolve storage issues
Some metrics missing from multi-metric endpoints	Partial parsing failures, label validation errors, metric name conflicts	Enable detailed parsing logs, check label validation rules, inspect metric naming	Handle partial failures gracefully, fix validation rules, resolve naming conflicts
Scraping performance degrades over time	Connection pool exhaustion, memory leaks, increasing target latency	Monitor connection pool usage, check memory growth, measure target response times	Implement connection pooling, fix memory leaks, adjust timeout settings
Targets oscillate between healthy and unhealthy	Network instability, target overload, circuit breaker misconfiguration	Analyze target health history, check target resource usage, review circuit breaker settings	Adjust health check thresholds, reduce target load, tune circuit breaker parameters

Storage Engine Symptoms

Storage problems often create subtle data corruption or performance issues that compound over time.

Symptom	Likely Root Causes	Investigation Steps	Resolution Actions
Write performance degrades significantly	WAL bottlenecks, compression inefficiency, disk I/O limits	Monitor WAL write rates, profile compression performance, check disk I/O patterns	Optimize WAL batching, fix compression algorithms, add faster storage
Queries return incorrect aggregated values	Compression errors, timestamp alignment issues, counter reset handling	Verify raw sample values, check timestamp ordering, test counter reset detection	Fix compression bugs, align timestamps properly, improve counter reset logic
Storage space usage grows faster than expected	Poor compression ratios, retention policy not working, high cardinality explosion	Analyze compression effectiveness, verify retention execution, audit label cardinality	Optimize compression parameters, fix retention policies, implement cardinality controls
Data corruption detected in stored chunks	Concurrent write bugs, filesystem issues, incomplete writes	Enable data integrity checks, review concurrent access patterns, check filesystem health	Fix concurrent write bugs, repair filesystem issues, add write atomicity
Recovery from WAL fails after crashes	WAL corruption, incomplete transaction records, ordering violations	Examine WAL contents, verify transaction boundaries, check record ordering	Implement WAL validation, add transaction atomicity, improve ordering guarantees

Query Engine Symptoms

Query problems can be particularly frustrating because they often work correctly on small datasets but fail at scale.

Symptom	Likely Root Causes	Investigation Steps	Resolution Actions
Queries timeout frequently	Inefficient query plans, high cardinality selectors, resource limits too low	Analyze query execution plans, check selector cardinality, monitor resource usage	Optimize query execution, add cardinality limits, increase resource limits
PromQL parsing errors for valid expressions	Parser bugs, unsupported features, operator precedence issues	Test expressions in isolation, check parser grammar, verify operator precedence	Fix parser bugs, implement missing features, correct precedence rules
Aggregation functions return wrong results	Grouping logic errors, timestamp alignment issues, sample interpolation bugs	Test with known datasets, verify grouping behavior, check timestamp handling	Fix grouping algorithms, align timestamps consistently, correct interpolation logic
Range queries miss data points	Time window calculations wrong, staleness threshold too strict, index selection issues	Verify time window boundaries, check staleness detection, review index selection logic	Fix time window calculations, adjust staleness thresholds, optimize index selection
Memory usage spikes during large queries	Result set too large, inefficient data structures, memory leaks in query engine	Monitor query result sizes, profile memory allocations, check for memory leaks	Add result size limits, optimize data structures, fix memory leaks

Performance and Scalability Symptoms

These symptoms typically appear under load and indicate architectural limitations or resource bottlenecks.

Symptom	Likely Root Causes	Investigation Steps	Resolution Actions
Scraping falls behind target intervals	Too many targets per scraper, slow target responses, processing bottlenecks	Monitor scraping queue depth, measure target response times, check processing latency	Add more scrapers, optimize target endpoints, parallelize processing
Query latency increases with data retention period	Index inefficiency, poor data locality, excessive disk I/O	Analyze query execution times by time range, check index hit rates, monitor disk I/O	Optimize index structures, improve data locality, add data tiering
System cannot handle target cardinality growth	Memory exhaustion from indexes, storage I/O limits, query complexity explosion	Monitor memory usage by component, check storage I/O patterns, analyze query complexity	Implement cardinality limits, add horizontal scaling, optimize query complexity
Background operations impact foreground performance	WAL flushing blocks writes, compaction affects reads, GC pauses affect queries	Monitor background operation timing, check resource contention, measure GC impact	Optimize background scheduling, reduce resource contention, tune GC parameters
Resource usage is uneven across system components	Load imbalance, inefficient resource allocation, component bottlenecks	Analyze resource usage by component, check load distribution, identify bottlenecks	Rebalance load, optimize resource allocation, scale bottlenecked components

Key Debugging Insight

The most effective debugging approach for metrics systems is to maintain end-to-end observability of the system itself. Every component should emit metrics about its own behavior, creating a "metrics system monitoring itself" capability. This recursive observability is essential because metrics system failures often have cascading effects that obscure the original root cause.

Debugging Tools and Techniques

Mental Model: The Medical Diagnostic Toolkit

Think of debugging tools like medical diagnostic equipment. Just as doctors use different instruments for different symptoms (stethoscope for heart issues, X-ray for bone problems, blood tests for infections), metrics system debugging requires specialized tools for different types of problems. Some tools provide continuous monitoring (like vital sign monitors), others require active investigation (like MRI scans), and some are only used in emergency situations (like defibrillators).

The key principle is using the right tool for the type of problem you're investigating, starting with non-invasive continuous monitoring and escalating to more intensive diagnostic techniques only when necessary.

Logging Infrastructure

Effective logging is the foundation of debugging any distributed system, but metrics systems have specific logging requirements due to their high-throughput, time-sensitive nature.

Structured Logging Configuration

Component	Log Level	Key Fields	Sampling Strategy	Retention Period
ScrapeEngine	INFO for successes, WARN for failures	target_url, scrape_duration, sample_count, error_message	100% for failures, 1% for successes	7 days
StorageEngine	DEBUG for writes, ERROR for corruption	series_count, sample_count, compression_ratio, wal_sync_duration	0.1% for writes, 100% for errors	3 days
QueryEngine	INFO for slow queries, DEBUG for execution	query_text, execution_time, series_selected, result_size	100% for >1s queries, 1% for fast queries	1 day
SystemCoordinator	INFO for lifecycle, ERROR for failures	component_name, operation, duration, resource_usage	100% for all events	30 days

Contextual Logging Best Practices

Every log entry should include sufficient context to understand the operation being performed. For metrics systems, this context typically includes temporal information (timestamps, time ranges), dimensional information (metric names, label sets), and operational information (component state, resource usage).

```
// Example of well-structured log entries:  
{  
    "timestamp": "2024-01-15T10:30:45.123Z",  
    "level": "ERROR",  
    "component": "ScrapeEngine",  
    "operation": "scrapeTarget",  
    "target_url": "http://api-server:8080/metrics",  
    "scrape_interval": "15s",  
    "attempt_number": 3,  
    "error": "context deadline exceeded",  
    "consecutive_failures": 5,  
    "target_labels": {"job": "api-server", "instance": "api-server:8080"},  
    "trace_id": "abc123def456"  
}  
  
{  
    "timestamp": "2024-01-15T10:30:45.456Z",  
    "level": "WARN",  
    "component": "StorageEngine",  
    "operation": "compressChunk",  
    "series_id": 12345,  
    "chunk_samples": 240,  
    "compression_ratio": 0.85,  
    "warning": "compression ratio below threshold",  
    "metric_name": "http_requests_total",  
    "label_hash": "def456abc123"  
}
```

System Metrics and Observability

The metrics collection system must monitor itself comprehensively. This self-monitoring provides real-time visibility into system health and performance characteristics.

Core System Metrics

Metric Category	Metric Name	Labels	Description	Alert Threshold
Scraping Health	scrape_targets_up	job, instance	Number of healthy targets per job	< 90% of targets
Scraping Performance	scrape_duration_seconds	job, instance, quantile	Scrape operation duration distribution	p99 > scrape_timeout
Storage Throughput	samples_ingested_total	component	Total samples successfully stored	Rate decreasing
Storage Errors	storage_errors_total	component, error_type	Storage operation failures by type	> 0.1% error rate
Query Performance	query_duration_seconds	query_type, quantile	Query execution time distribution	p95 > 10 seconds
Query Load	concurrent_queries	none	Number of actively executing queries	> max_concurrent_queries
Resource Usage	memory_usage_bytes	component	Memory usage by system component	> 80% of limit
Resource Usage	goroutine_count	component	Active goroutines by component	Continuously increasing

Component Health Indicators

Each system component should provide standardized health indicators that can be monitored independently and in aggregate.

Health Indicator	Measurement Method	Healthy Range	Warning Range	Critical Range
Scrape Success Rate	Successful scrapes / Total scrape attempts	> 95%	90-95%	< 90%
Storage Write Success Rate	Successful writes / Total write attempts	> 99%	95-99%	< 95%
Query Success Rate	Successful queries / Total queries	> 99%	95-99%	< 95%
WAL Sync Latency	Time to fsync WAL entries	< 10ms	10-50ms	> 50ms
Index Lookup Latency	Time to resolve label selectors	< 1ms	1-10ms	> 10ms
Compression Efficiency	Compressed size / Raw size	< 0.2	0.2-0.5	> 0.5

Interactive Debugging Tools

For active investigation of problems, several interactive tools provide detailed system inspection capabilities.

Component State Inspection

Tool	Command Format	Information Provided	Use Cases
Target Health Inspector	<code>GET /debug/targets</code>	Current health status, last scrape results, failure reasons	Diagnosing scraping issues
Series Cardinality Inspector	<code>GET /debug/cardinality?metric=<name></code>	Label combination counts, high-cardinality series	Investigating memory usage
Storage Consistency Checker	<code>GET /debug/storage/check</code>	Index-data consistency, corruption detection	Validating storage integrity
Query Execution Planner	<code>GET /debug/query/explain?query=<promql></code>	Execution plan, estimated resource usage	Optimizing slow queries
WAL Content Inspector	<code>GET /debug/wal/entries?limit=<n></code>	Recent WAL entries, transaction boundaries	Debugging storage issues

Profiling and Performance Analysis

Profiling Tool	Activation Method	Data Collected	Analysis Focus
CPU Profiler	<code>GET /debug/pprof/profile</code>	CPU usage by function	Hot paths, algorithmic efficiency
Memory Profiler	<code>GET /debug/pprof/heap</code>	Memory allocations by location	Memory leaks, allocation patterns
Goroutine Profiler	<code>GET /debug/pprof/goroutine</code>	Goroutine stacks and states	Deadlocks, resource contention
Mutex Profiler	<code>GET /debug/pprof/mutex</code>	Lock contention by location	Synchronization bottlenecks
Block Profiler	<code>GET /debug/pprof/block</code>	Blocking operations by location	I/O bottlenecks, channel contention

Distributed Tracing Integration

For complex issues that span multiple components, distributed tracing provides end-to-end visibility into request processing.

Trace Instrumentation Points

Operation	Trace Span Name	Key Attributes	Child Spans
HTTP Scrape	scrape_target	target_url, scrape_interval	http_request, parse_metrics, store_samples
Sample Storage	store_samples	sample_count, series_count	wal_write, compress_chunk, update_index
Query Execution	execute_query	query_type, time_range	parse_expression, select_series, aggregate_results
Series Selection	select_series	selector_count, series_matched	index_lookup, label_matching
Range Query	range_query	start_time, end_time, step	instant_query (multiple)

Trace Analysis Techniques

When investigating distributed issues, trace analysis reveals timing relationships and failure propagation patterns that are invisible in individual component logs.

Critical timing relationships to analyze include scrape-to-storage latency (how long samples take to become queryable), query-to-result latency (end-to-end query performance), and error propagation delays (how long failures take to surface). Performance bottleneck identification focuses on the longest spans in the trace, resource contention points where spans block waiting for resources, and cascade failure patterns where one component failure triggers failures in dependent components.

Debugging Insight: The Observer Effect

Remember that debugging tools themselves consume system resources and can change system behavior. Heavy logging, frequent profiling, and detailed tracing all introduce overhead that may mask or alter the problems you're trying to investigate. Use sampling and conditional instrumentation to minimize observer effects during production debugging.

Implementation Pitfalls

Mental Model: The Software Engineering Minefield

Think of common implementation pitfalls like a minefield that every developer must navigate. Experienced engineers know where the mines are buried based on previous painful explosions. By documenting these pitfalls explicitly, we create a map that helps new developers avoid stepping on the same mines that have injured others.

The key insight is that pitfalls in metrics systems often appear as "working" code that fails under specific conditions like high cardinality, time zone changes, or concurrent access patterns. These delayed failures make pitfalls particularly dangerous because they pass initial testing but create production incidents.

Metrics Data Model Pitfalls

These pitfalls relate to the fundamental data structures and semantic behaviors that form the foundation of the metrics system.

Pitfall: Counter Reset Handling

Many developers implement counters as simple incrementing values without considering that counter resets (when a process restarts) require special handling for rate calculations to remain accurate.

Why This Fails: When a counter resets to zero, naive rate calculations produce large negative values because `current_value - previous_value` becomes negative. This creates obviously wrong metrics and breaks downstream alerting and dashboards.

Detection: Monitor for negative rate values in counter-based metrics. Implement validation that rejects negative rates from counter calculations.

Correct Implementation: Track counter resets by detecting when the current value is less than the previous value, and handle resets by either skipping the rate calculation for that interval or estimating the true rate by assuming the counter accumulated `current_value` since the reset.

Pitfall: High Cardinality Label Design

Developers often add labels with high cardinality (user IDs, request IDs, timestamps) without understanding the exponential memory impact of label combinations.

Why This Fails: Each unique combination of metric name and label values creates a separate time series. Labels with thousands of values create millions of time series that exhaust system memory. For example, a metric with labels `{user_id, endpoint, status}` where user_id has 10,000 values, endpoint has 50 values, and status has 5 values creates up to 2.5 million time series.

Detection: Monitor total series count and memory usage. Implement cardinality reporting that shows series count per metric name and per label combination.

Correct Implementation: Use high-cardinality information in log entries rather than metric labels. Design label schemas with bounded cardinality - prefer categorical labels like `{status="success|failure", tier="premium|standard"}` over unbounded labels like `{user_id="12345", request_id="abc-def"}`.

Pitfall: Histogram Bucket Boundaries

Developers often choose histogram bucket boundaries arbitrarily or try to change them after the histogram has been created and is collecting data.

Why This Fails: Histogram bucket boundaries must remain consistent throughout the lifetime of the metric. Changing boundaries makes historical data incomparable and breaks percentile calculations. Poor bucket boundary choices (too few buckets, wrong ranges) provide insufficient resolution for meaningful percentile analysis.

Detection: Monitor histogram bucket distribution - if most values fall into only one or two buckets, the boundaries are poorly chosen. Check for bucket boundary inconsistencies in historical data.

Correct Implementation: Choose histogram buckets based on the expected distribution of values you're measuring. Use exponential bucket spacing (1ms, 2ms, 5ms, 10ms, 25ms, 50ms, 100ms...) for latency measurements. Once chosen, never modify bucket boundaries - create a new histogram metric instead.

Scraping Engine Pitfalls

These pitfalls involve HTTP operations, timing, and concurrency patterns specific to pull-based metrics collection.

Pitfall: Scrape Timeout Implementation

Many developers implement scrape timeouts using `time.After()` or similar mechanisms without properly canceling the underlying HTTP request, leading to resource leaks.

Why This Fails: Even when the scrape operation times out from the coordinator's perspective, the underlying HTTP request continues consuming connection pool resources, goroutines, and network bandwidth. Over time, these leaked resources accumulate and eventually exhaust system capacity.

Detection: Monitor goroutine count and HTTP connection pool usage. Look for continuously increasing resource usage even when scrape timeouts are occurring.

Correct Implementation: Use context-based cancellation with `context.WithTimeout()` and pass the context to `http.Request.WithContext()`. This ensures that timing out the scrape operation also cancels the underlying HTTP request and releases all associated resources.

Pitfall: Concurrent Scraping Without Rate Limiting

Developers often implement concurrent scraping by launching goroutines for each target without considering the aggregate load this places on target services.

Why This Fails: Launching hundreds of concurrent HTTP requests can overwhelm target services, creating cascading failures. Target services may implement their own rate limiting or simply crash under the load, making metrics collection counterproductively impact the systems being monitored.

Detection: Monitor target response times and error rates. Look for correlation between scraper concurrency levels and target service health.

Correct Implementation: Implement scraper worker pools with bounded concurrency. Use semaphores or buffered channels to limit the maximum number of concurrent scrape operations. Consider implementing adaptive rate limiting that backs off when target services show signs of stress.

Pitfall: Service Discovery Race Conditions

Developers often implement service discovery updates by directly modifying the target list without considering concurrent access from scraping goroutines.

Why This Fails: Concurrent modification of the target list during scraping operations creates race conditions that can cause panics, data corruption, or targets being scraped multiple times simultaneously.

Detection: Run with the Go race detector enabled (`go run -race`). Monitor for panics or unexpected behavior during service discovery updates.

Correct Implementation: Use copy-on-write semantics for target list updates. Create a new target list and atomically replace the reference, allowing in-flight scrape operations to complete with the old target list before switching to the new one.

Storage Engine Pitfalls

Storage engine pitfalls often involve data corruption, concurrency issues, and compression edge cases that only surface under specific conditions.

Pitfall: WAL Corruption During Concurrent Writes

Developers often implement write-ahead logging with multiple goroutines writing directly to the WAL file without proper synchronization.

Why This Fails: Multiple concurrent writers can interleave bytes from different log entries, creating corrupted WAL entries that cannot be parsed during recovery. This makes the entire WAL unusable and causes data loss during crash recovery.

Detection: Implement WAL validation during recovery that checks entry boundaries and checksums. Monitor for WAL recovery failures after system restarts.

Correct Implementation: Serialize all WAL writes through a single goroutine using a channel-based queue, or implement file-level locking to ensure exclusive access during write operations. Always fsync after writing complete log entries to ensure durability.

Pitfall: Gorilla Compression Edge Cases

Developers often implement Gorilla compression by following the basic algorithm without handling edge cases like irregular timestamp intervals, floating point special values, or compression ratio degradation.

Why This Fails: The Gorilla compression algorithm assumes relatively regular timestamp intervals and normal floating point values. Irregular timestamps reduce compression effectiveness, while special values like NaN or infinity can break the XOR-based value compression. When compression ratios degrade, the system uses much more memory than expected.

Detection: Monitor compression ratios per chunk and overall system memory usage. Implement validation for special floating point values in incoming samples.

Correct Implementation: Handle irregular timestamps by falling back to absolute timestamp encoding when delta-of-delta compression becomes ineffective. Validate floating point values and reject or special-case NaN and infinity. Monitor compression ratios and implement fallback storage for chunks that don't compress well.

Pitfall: Index Consistency Under Concurrent Access

Developers often implement inverted indexes with concurrent read and write access without proper synchronization, leading to index corruption or inconsistent query results.

Why This Fails: Concurrent modification of index data structures can leave them in inconsistent states where some series are visible through some label combinations but not others, or where index entries point to non-existent series data.

Detection: Implement index consistency checks that verify all index entries point to valid series data and that series data can be found through expected index paths. Run these checks periodically and after any index corruption reports.

Correct Implementation: Use readers-writer mutexes to allow concurrent reads while serializing writes. Implement index updates as atomic operations that either complete entirely or leave the index unchanged. Consider using copy-on-write strategies for large index updates.

Query Engine Pitfalls

Query engine pitfalls often involve mathematical edge cases, performance characteristics that don't scale, and semantic misunderstandings of PromQL operations.

Pitfall: Rate Calculation Across Counter Resets

Developers often implement rate calculations using simple arithmetic without detecting and handling counter resets properly.

Why This Fails: Counter resets create negative rate values that are mathematically incorrect and break visualizations and alerting. The `rate()` function should calculate the true rate of increase, which requires detecting resets and handling them appropriately.

Detection: Monitor for negative values in rate calculation results. Implement validation that rejects impossible rate values based on the known characteristics of the underlying counter.

Correct Implementation: Detect counter resets by checking if the current counter value is less than the previous value. When a reset is detected, calculate the rate as `current_value / time_interval`, assuming the counter accumulated its current value since the reset. For partial intervals after resets, either skip the rate calculation or pro-rate based on the partial interval.

Pitfall: Query Resource Exhaustion

Developers often implement query execution without considering resource limits, allowing queries over high-cardinality metrics to exhaust system memory.

Why This Fails: Queries that select thousands of time series can consume gigabytes of memory for intermediate results and final output. Without resource limits, a single poorly-written query can crash the entire metrics system.

Detection: Monitor memory usage during query execution and track query result sizes. Implement timeouts and memory limits for query operations.

Correct Implementation: Implement query resource limits including maximum series per query, maximum query execution time, and maximum result set size. Use streaming processing where possible to avoid loading entire result sets into memory. Provide query cost estimation to help users understand the resource impact of their queries.

Pitfall: Aggregation Function Semantic Errors

Developers often implement aggregation functions like `avg()` or `sum()` without properly handling missing data points, different timestamp alignments, or grouping semantics.

Why This Fails: Different time series may have samples at different timestamps, and aggregation functions must handle these alignment issues correctly. Simply averaging all available values without considering temporal alignment produces mathematically meaningless results.

Detection: Test aggregation functions with time series that have different sampling intervals and missing data points. Verify that aggregation results match hand-calculated expected values.

Correct Implementation: Implement timestamp alignment by interpolating or extrapolating values to common evaluation timestamps before applying aggregation functions. Handle missing data appropriately - some functions should skip missing values while others should treat them as zero. Clearly document the temporal semantics of each aggregation function.

Implementation Guidance

This section provides practical tools and code structures for implementing effective debugging capabilities throughout the metrics collection system.

Technology Recommendations

Component	Simple Option	Advanced Option
Structured Logging	<code>log/slog</code> with JSON formatting	<code>go.uber.org/zap</code> with sampling and async writing
HTTP Debugging Endpoints	<code>net/http/pprof</code> + custom handlers	<code>go.uber.org/fx</code> + <code>github.com/gorilla/mux</code> for organized routing
Metrics Self-Monitoring	Manual metrics with <code>expvar</code>	<code>github.com/prometheus/client_golang</code> for full compatibility
Distributed Tracing	Manual trace context propagation	<code>go.opentelemetry.io/otel</code> with Jaeger or Zipkin
Health Checking	Simple HTTP endpoint with JSON status	Full health check framework with dependencies
Configuration Validation	Manual validation with error accumulation	<code>github.com/go-playground/validator/v10</code> for declarative rules

Recommended File Structure

```
internal/
  debug/
    health.go          ← component health checking
    profiling.go       ← performance profiling endpoints
    inspection.go      ← system state inspection tools
    tracing.go         ← distributed tracing utilities
    debug_test.go      ← debugging tool tests
  monitoring/
    metrics.go         ← self-monitoring metrics definitions
    alerts.go          ← alerting rule definitions
    dashboards.go      ← dashboard configuration export
  validation/
    config_validator.go ← configuration validation
    data_validator.go   ← data consistency checking
    performance_validator.go ← performance requirement validation
cmd/
  debug-tool/
    main.go            ← standalone debugging CLI tool
  health-check/
    main.go            ← health checking utility
```

Infrastructure Starter Code

Complete Structured Logger Implementation

GO

```
package debug

import (
    "context"
    "log/slog"
    "os"
    "time"
)

// Logger provides structured logging with different levels and context support

type Logger struct {
    logger *slog.Logger
    component string
}

// NewLogger creates a logger for a specific component with JSON formatting

func NewLogger(component string) *Logger {
    handler := slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{
        Level: slog.LevelDebug,
        AddSource: true,
    })

    logger := slog.New(handler).With(
        "component", component,
        "pid", os.Getpid(),
    )

    return &Logger{
        logger: logger,
        component: component,
    }
}
```

```
}

}

// Info logs informational messages with structured fields

func (l *Logger) Info(msg string, fields ...any) {

    l.logger.Info(msg, fields...)
}

// Error logs error messages with structured fields

func (l *Logger) Error(msg string, err error, fields ...any) {

    allFields := append(fields, "error", err.Error())

    l.logger.Error(msg, allFields...)
}

// Debug logs debug messages with structured fields (only in debug builds)

func (l *Logger) Debug(msg string, fields ...any) {

    l.logger.Debug(msg, fields...)
}

// WithContext returns a logger that includes trace information from context

func (l *Logger) WithContext(ctx context.Context) *Logger {

    // TODO: Extract trace ID from context if tracing is enabled

    return l
}

// WithFields returns a logger with additional fields included in all log entries

func (l *Logger) WithFields(fields ...any) *Logger {

    return &Logger{
        logger: l.logger.With(fields...),
        component: l.component,
    }
}
```

}

Complete Health Checking System

```
package debug
```

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

// HealthStatus represents the health state of a component

type HealthStatus string

const (
    HealthUp      HealthStatus = "up"
    HealthDown    HealthStatus = "down"
    HealthDegraded HealthStatus = "degraded"
)

// ComponentHealth represents the health of a single system component

type ComponentHealth struct {

    Component string      `json:"component"`

    Status    HealthStatus `json:"status"`

    Message   string      `json:"message,omitempty"`

    Timestamp time.Time   `json:"timestamp"`

    Metrics   map[string]interface{} `json:"metrics,omitempty"`
}

// SystemHealth aggregates health information from all system components

type SystemHealth struct {

    OverallStatus HealthStatus `json:"overall_status"`
}
```

GO

```

Components      map[string]*ComponentHealth    `json:"components"`

Timestamp      time.Time                      `json:"timestamp"`

Uptime         time.Duration                 `json:"uptime"`

}

// HealthChecker manages health checking for all system components

type HealthChecker struct {

    components map[string]*ComponentHealth

    startTime  time.Time

    mu         sync.RWMutex

}

// NewHealthChecker creates a new health checker instance

func NewHealthChecker() *HealthChecker {

    return &HealthChecker{

        components: make(map[string]*ComponentHealth),

        startTime:  time.Now(),

    }

}

// RegisterComponent registers a component for health checking

func (hc *HealthChecker) RegisterComponent(name string) {

    hc.mu.Lock()

    defer hc.mu.Unlock()

    hc.components[name] = &ComponentHealth{

        Component: name,

        Status:    HealthDown,

        Message:   "Component not yet initialized",

        Timestamp: time.Now(),

```

```
}

}

// UpdateComponentHealth updates the health status of a component

func (hc *HealthChecker) UpdateComponentHealth(name string, status HealthStatus, message string,
metrics map[string]interface{}) {

    hc.mu.Lock()

    defer hc.mu.Unlock()

    if component, exists := hc.components[name]; exists {

        component.Status = status

        component.Message = message

        component.Timestamp = time.Now()

        component.Metrics = metrics

    }

}

// GetSystemHealth returns the overall system health status

func (hc *HealthChecker) GetSystemHealth() *SystemHealth {

    hc.mu.RLock()

    defer hc.mu.RUnlock()

    overallStatus := HealthUp

    componentCopy := make(map[string]*ComponentHealth)

    for name, component := range hc.components {

        // Create a copy to avoid race conditions

        componentCopy[name] = &ComponentHealth{

            Component: component.Component,

            Status: component.Status,
```

```
        Message: component.Message,
        Timestamp: component.Timestamp,
        Metrics: component.Metrics,
    }

    // Determine overall status

    if component.Status == HealthDown {
        overallStatus = HealthDown
    } else if component.Status == HealthDegraded && overallStatus == HealthUp {
        overallStatus = HealthDegraded
    }
}

return &SystemHealth{

    OverallStatus: overallStatus,
    Components: componentCopy,
    Timestamp: time.Now(),
    Uptime: time.Since(hc.startTime),
}
}

// ServeHTTP implements http.Handler to expose health information via HTTP

func (hc *HealthChecker) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    health := hc.GetSystemHealth()

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

    // Set HTTP status code based on overall health
    switch health.OverallStatus {
```

```
case HealthUp:  
    w.WriteHeader(http.StatusOK)  
  
case HealthDegraded:  
    w.WriteHeader(http.StatusOK) // Still serving requests  
  
case HealthDown:  
    w.WriteHeader(http.StatusServiceUnavailable)  
}  
  
json.NewEncoder(w).Encode(health)  
}
```

Core Logic Skeleton Code

Component State Inspector

GO

```
// ComponentStateInspector provides detailed inspection of internal component state

type ComponentStateInspector struct {

    scrapeEngine *ScrapeEngine

    storageEngine *StorageEngine

    queryEngine *QueryEngine

    logger      *Logger

}

// NewComponentStateInspector creates an inspector with access to all system components

func NewComponentStateInspector(scrapeEngine *ScrapeEngine, storageEngine *StorageEngine,
queryEngine *QueryEngine, logger *Logger) *ComponentStateInspector {

    return &ComponentStateInspector{

        scrapeEngine:  scrapeEngine,

        storageEngine: storageEngine,

        queryEngine:   queryEngine,

        logger:        logger,

    }

}

// InspectScrapeTargets returns detailed information about all scrape targets

func (csi *ComponentStateInspector) InspectScrapeTargets() map[string]interface{} {

    // TODO 1: Lock scrape engine to prevent concurrent modification during inspection

    // TODO 2: Iterate through all configured targets and collect their current state

    // TODO 3: For each target, include: URL, last scrape time, consecutive failures, health
status

    // TODO 4: Calculate aggregate statistics: total targets, healthy targets, failure rate

    // TODO 5: Return structured data suitable for JSON serialization

    // Hint: Use read locks to allow continued operation while inspecting

    return nil

}
```

```
// InspectStorageState returns information about storage engine internal state

func (csi *ComponentStateInspector) InspectStorageState() map[string]interface{} {

    // TODO 1: Acquire read lock on storage engine to ensure consistent state view

    // TODO 2: Count total number of time series and chunks across all indexes

    // TODO 3: Calculate storage efficiency metrics: compression ratio, samples per chunk

    // TODO 4: Inspect WAL state: current segment, pending entries, last sync time

    // TODO 5: Check index consistency: verify all series have corresponding chunks

    // TODO 6: Return comprehensive storage health and efficiency data

    return nil

}

// InspectQueryPerformance returns query engine performance and resource usage data

func (csi *ComponentStateInspector) InspectQueryPerformance() map[string]interface{} {

    // TODO 1: Collect query execution statistics from query engine

    // TODO 2: Calculate percentiles for query duration and result set sizes

    // TODO 3: Identify slow queries and high-cardinality selectors

    // TODO 4: Report resource usage: memory per query, concurrent query count

    // TODO 5: Return performance data suitable for optimization analysis

    return nil

}
```

Distributed Tracing Integration

```
// TracingCoordinator manages distributed tracing across all system components

type TracingCoordinator struct {

    enabled    bool

    sampler   Sampler

    tracer    Tracer

    logger    *Logger

}

// NewTracingCoordinator creates a tracing coordinator with sampling configuration

func NewTracingCoordinator(enabled bool, samplingRate float64, logger *Logger) *TracingCoordinator {
    // TODO 1: Initialize tracer with appropriate backend (Jaeger, Zipkin, or no-op)

    // TODO 2: Configure sampler with specified sampling rate and adaptive algorithms

    // TODO 3: Set up trace context propagation for HTTP requests and internal operations

    // TODO 4: Return configured coordinator ready for use across system components

    return nil
}

// StartSpan creates a new tracing span for an operation with specified attributes

func (tc *TracingCoordinator) StartSpan(ctx context.Context, operationName string, attributes map[string]interface{}) (context.Context, Span) {
    // TODO 1: Check if tracing is enabled and sampling decision allows this trace

    // TODO 2: Extract parent span context from incoming context if available

    // TODO 3: Create new span with operation name and configured attributes

    // TODO 4: Add standard attributes: component name, operation timestamp, correlation ID

    // TODO 5: Return new context containing span and span object for lifecycle management

    return ctx, nil
}

// InjectTraceHeaders adds tracing information to HTTP request headers for propagation

func (tc *TracingCoordinator) InjectTraceHeaders(ctx context.Context, headers http.Header) {
```

```

    // TODO 1: Extract active span from context if present

    // TODO 2: Serialize span context into trace propagation headers (B3, Jaeger, etc.)

    // TODO 3: Add headers to outgoing HTTP request for downstream trace correlation

    // Hint: Handle case where no active span exists gracefully

}

// ExtractTraceContext extracts tracing information from incoming HTTP request headers

func (tc *TracingCoordinator) ExtractTraceContext(headers http.Header) context.Context {
    // TODO 1: Parse trace propagation headers from incoming HTTP request

    // TODO 2: Reconstruct span context from header information

    // TODO 3: Create new context containing extracted trace information

    // TODO 4: Return context suitable for starting child spans

    return context.Background()
}

```

Milestone Checkpoints

Milestone 1 Debugging Verification: Metrics Data Model

After implementing the metrics data model with debugging support:

- 1. Run Component Tests:** Execute `go test ./internal/metrics/... -v` to verify all metric types handle edge cases correctly
- 2. Verify Cardinality Tracking:** Create metrics with high-cardinality labels and confirm the system detects and reports the cardinality explosion
- 3. Test Counter Reset Handling:** Simulate counter resets and verify that subsequent rate calculations handle them appropriately
- 4. Check Label Validation:** Attempt to create metrics with invalid label names and values, confirming that validation prevents high-cardinality labels

Expected Behavior: Cardinality tracking reports series counts accurately, counter resets are detected and logged, label validation prevents problematic metric creation, and all metric types expose their internal state for debugging.

Milestone 2 Debugging Verification: Scrape Engine

After implementing the scrape engine with debugging support:

- 1. Run Scraping Tests:** Execute `go test ./internal/scraping/... -v` with mock HTTP targets to verify scraping logic

2. **Test Timeout Handling:** Configure short scrape timeouts and verify that timeouts cancel HTTP requests properly without leaking resources
3. **Verify Target Health Tracking:** Start and stop mock target services and confirm that target health status updates correctly
4. **Check Service Discovery:** Modify service discovery configuration and verify that target list updates without disrupting ongoing scrapes

Expected Behavior: Scrape operations respect timeouts and cancel underlying resources, target health accurately reflects service availability, service discovery updates are applied safely, and detailed scrape metrics are available for debugging.

Milestone 3 Debugging Verification: Time Series Storage

After implementing the time series storage with debugging support:

1. **Run Storage Tests:** Execute `go test ./internal/storage/... -v` to verify compression, indexing, and WAL functionality
2. **Test WAL Recovery:** Kill the storage process during write operations and verify that WAL recovery restores consistent state
3. **Verify Compression Effectiveness:** Store time series with different patterns and confirm that compression ratios meet expectations
4. **Check Index Consistency:** Run index consistency checks and verify that all time series can be found through expected label selectors

Expected Behavior: WAL recovery restores all committed data after crashes, compression ratios achieve expected efficiency, index consistency checks pass, and storage state can be inspected comprehensively.

Milestone 4 Debugging Verification: Query Engine

After implementing the query engine with debugging support:

1. **Run Query Tests:** Execute `go test ./internal/query/... -v` to verify PromQL parsing and execution
2. **Test Resource Limits:** Execute queries that exceed configured limits and verify that they are rejected appropriately
3. **Verify Aggregation Correctness:** Run aggregation queries with known expected results and confirm mathematical accuracy
4. **Check Performance Monitoring:** Execute slow queries and verify that performance metrics accurately capture execution characteristics

Expected Behavior: Query resource limits prevent system overload, aggregation functions produce mathematically correct results, query performance is monitored and reported accurately, and query execution plans can be inspected for optimization.

Language-Specific Debugging Hints

Go Race Detection and Concurrency Debugging

- Always run tests with `-race` flag during development: `go test -race ./...`
- Use `go run -race` when testing manually to catch race conditions early

- Monitor goroutine count with `runtime.NumGoroutine()` to detect goroutine leaks
- Use `runtime.GC(); runtime.GC()` to force garbage collection before measuring memory usage
- Use buffered channels with explicit capacity to avoid deadlocks: `make(chan Sample, 100)`

Memory Profiling and Optimization

- Use `go tool pprof http://localhost:8080/debug/pprof/heap` for interactive memory analysis
- Check for memory leaks by comparing heap profiles over time: `go tool pprof -base profile1.pb.gz profile2.pb.gz`
- Use `runtime/debug.SetGCPercent()` to tune garbage collection frequency based on workload
- Implement object pooling with `sync.Pool` for frequently allocated objects like samples and labels
- Use `unsafe.Sizeof()` to understand memory layout of critical data structures

HTTP Client Configuration for Reliability

- Set reasonable timeouts: `http.Client{Timeout: 30 * time.Second}`
- Configure connection pooling: `http.Transport{MaxIdleConns: 100, MaxIdleConnsPerHost: 10}`
- Use context cancellation: `req.WithContext(ctx)` for all HTTP requests
- Implement exponential backoff for retries using `time.Sleep(time.Duration(attempt*attempt) * time.Second)`
- Monitor connection pool usage with custom `http.Transport` metrics

Error Handling and Recovery Patterns

- Use typed errors for different failure modes: `type NetworkError struct { ... }`
- Implement circuit breakers for external dependencies to prevent cascade failures
- Log errors with full context including operation, input parameters, and system state
- Use `defer` statements for cleanup that must happen regardless of success or failure
- Implement graceful degradation where partial functionality is better than complete failure

Future Extensions and Scalability

Milestone(s): This section demonstrates how the Metrics Data Model (1), Scrape Engine (2), Time Series Storage (3), and Query Engine (4) architecture supports future enhancements without major redesigns.

Think of our metrics collection system as a city's infrastructure. We've built the fundamental utilities - the power grid (`StorageEngine`), water system (`ScrapeEngine`), and communication network (`QueryEngine`). Now we need to plan for future growth: adding new neighborhoods (federation), emergency services (alerting), and express highways (query optimizations). The key architectural insight is that extensible systems are designed with **composition over inheritance** - new capabilities are added by combining existing components rather than rewriting them.

The extensibility of our metrics system relies on three core architectural principles that we established in earlier sections. First, our **component isolation** ensures that new features can be added without disrupting existing functionality - the `ScrapeEngine`, `StorageEngine`, and `QueryEngine` communicate through well-defined

interfaces rather than tight coupling. Second, our **data model completeness** means that metric labels, timestamps, and metadata provide sufficient information for advanced features like rule evaluation and cross-instance queries. Third, our **pipeline architecture** with channels and coordinators creates natural extension points where new processing stages can be inserted.

Alerting System

The alerting system represents the first major extension that transforms our passive metrics collection into an active monitoring platform. Think of alerting as adding a security system to our infrastructure city - it continuously watches for specific conditions and triggers responses when thresholds are crossed.

Rule Evaluation Engine

The **Rule Evaluation Engine** extends our existing `QueryEngine` by adding continuous evaluation of PromQL expressions against incoming time series data. Rather than building a separate system, we leverage the query parsing and execution infrastructure we already have.

AlertRule Data Structure:

Field Name	Type	Description
<code>Name</code>	<code>string</code>	Unique identifier for the alert rule
<code>Query</code>	<code>string</code>	PromQL expression to evaluate
<code>Duration</code>	<code>time.Duration</code>	How long condition must be true before firing
<code>Labels</code>	<code>Labels</code>	Additional labels attached to alert instances
<code>Annotations</code>	<code>map[string]string</code>	Human-readable descriptions and runbook links
<code>EvaluationInterval</code>	<code>time.Duration</code>	How frequently to evaluate the rule
<code>State</code>	<code>AlertState</code>	Current state: pending, firing, or resolved
<code>ActiveSince</code>	<code>time.Time</code>	When the alert first became active
<code>ResolvedAt</code>	<code>*time.Time</code>	When the alert was resolved (nil if still active)

The rule evaluation process follows a continuous assessment cycle:

1. The `RuleEvaluator` maintains a schedule of all active alert rules with their next evaluation times
2. At each evaluation interval, it executes the rule's PromQL query using our existing `QueryEngine.ExecuteInstantQuery` method
3. For each time series returned by the query, it checks if the result value meets the alert condition
4. If the condition is met, it starts tracking the duration - alerts only fire after being active for the specified `Duration`
5. When an alert transitions from pending to firing, it generates an `AlertInstance` with the current timestamp and label set

6. The evaluator continues checking resolved conditions - when the query returns no results or false values, it marks alerts as resolved

Design Insight: By reusing our `QueryEngine` for rule evaluation, we automatically inherit all the query optimization, caching, and error handling that we built for user queries. This demonstrates the power of composable architecture - new features leverage existing robust components.

RuleEvaluator Interface:

Method Name	Parameters	Returns	Description
AddRule	<code>rule *AlertRule</code>	<code>error</code>	Registers a new alert rule for evaluation
RemoveRule	<code>ruleName string</code>	<code>error</code>	Stops evaluating and removes an alert rule
EvaluateAll	<code>ctx context.Context, evalTime time.Time</code>	<code>[]AlertInstance, error</code>	Evaluates all rules at the specified time
GetAlertState	<code>ruleName string</code>	<code>AlertState, error</code>	Returns current state of a specific alert rule
ListActiveAlerts	<code>labels LabelMatcher</code>	<code>[]AlertInstance, error</code>	Returns all currently firing alerts matching label filters

Notification Manager

The **Notification Manager** handles the delivery of alert notifications through multiple channels. Think of it as the emergency dispatch system - when the security system (rule evaluator) detects a problem, the dispatcher determines who to notify and how.

NotificationChannel Interface:

Method Name	Parameters	Returns	Description
Send	<code>ctx context.Context, alert AlertInstance</code>	<code>error</code>	Delivers alert notification through this channel
Test	<code>ctx context.Context</code>	<code>error</code>	Verifies channel configuration and connectivity
GetType		<code>string</code>	Returns channel type (email, slack, webhook, etc.)
IsHealthy		<code>bool</code>	Indicates if channel is currently operational

The notification pipeline implements sophisticated routing and de-duplication:

- Alert Grouping:** Multiple related alerts are combined into a single notification to prevent spam - alerts with identical label sets (excluding instance-specific labels) are grouped together
- Rate Limiting:** Each notification channel has configurable rate limits to prevent overwhelming external systems during large-scale outages
- Retry Logic:** Failed notifications are queued for retry with exponential backoff - critical alerts get more aggressive retry attempts
- Escalation Chains:** After a specified time without acknowledgment, alerts can escalate to additional notification channels or contacts
- Maintenance Windows:** Notifications can be suppressed during scheduled maintenance periods based on label matching

Decision: Notification Architecture

- Context:** Alert notifications need to be reliable, fast, and handle various delivery failures
- Options Considered:**
 - Direct synchronous delivery from rule evaluator
 - Asynchronous queue-based delivery with persistence
 - Hybrid approach with immediate delivery plus persistent retry queue
- Decision:** Asynchronous queue-based delivery with WAL persistence
- Rationale:** Rule evaluation must continue even if notification delivery fails. Persistent queue ensures alerts aren't lost during system restarts
- Consequences:** Adds complexity but provides notification reliability and system resilience

Alert State Management

Alert state management tracks the lifecycle of alert instances from creation through resolution. This extends our existing `StorageEngine` with specialized alert state persistence.

AlertInstance Data Structure:

Field Name	Type	Description
<code>RuleName</code>	<code>string</code>	Name of the alert rule that generated this instance
<code>Labels</code>	<code>Labels</code>	Complete label set including rule labels and metric labels
<code>State</code>	<code>AlertState</code>	Current state: pending, firing, resolved
<code>Value</code>	<code>float64</code>	The metric value that triggered the alert
<code>StartsAt</code>	<code>time.Time</code>	When the alert condition first became true
<code>EndsAt</code>	<code>*time.Time</code>	When the alert condition resolved (nil if still active)
<code>GeneratorURL</code>	<code>string</code>	URL to query that generated this alert
<code>Fingerprint</code>	<code>uint64</code>	Hash of label set for efficient deduplication

Alert state transitions follow a strict finite state machine:

Current State	Event	Next State	Actions Taken
None	Query returns true	Pending	Record start time, begin duration tracking
Pending	Duration exceeded	Firing	Generate notification, mark as active
Pending	Query returns false	None	Clear tracking, no notification
Firing	Query returns false	Resolved	Send resolution notification, record end time
Resolved	Query returns true	Pending	Start new alert instance cycle

Multi-Instance Federation

Federation enables horizontal scaling by connecting multiple independent metrics collection instances into a coordinated cluster. Think of federation as connecting multiple city infrastructures into a metropolitan area - each city maintains its own services while sharing critical information across the region.

Hierarchical Federation Model

Our federation model follows a **hierarchical pull-based approach** that aligns with our existing scrape engine architecture. Rather than building complex consensus protocols, we extend the scraping concept to pull metrics from other Prometheus-compatible instances.

Federation Configuration:

Field Name	Type	Description
<code>FederationConfig</code>	<code>struct</code>	Top-level federation configuration
<code>UpstreamTargets</code>	<code>[]FederationTarget</code>	List of upstream instances to scrape from
<code>MatchRules</code>	<code>[]FederationRule</code>	Rules for selecting which metrics to federate
<code>ScrapeInterval</code>	<code>time.Duration</code>	How often to scrape upstream instances
<code>ExternalLabels</code>	<code>Labels</code>	Labels added to all metrics from this instance

The federation process extends our existing target discovery and scraping:

1. **Target Discovery:** Federation targets are configured as static targets in the `ScrapeEngine` with special federation endpoints (`/federate`)
2. **Metric Selection:** Each federation target specifies `MatchRules` that determine which metrics to pull - this prevents recursive federation and controls data volume
3. **Label Rewriting:** Federated metrics receive additional external labels that identify their source instance and prevent label conflicts
4. **Conflict Resolution:** When multiple instances provide the same time series (same metric name and labels), the most recent timestamp wins

5. **Topology Management:** The federation hierarchy is maintained through configuration - each instance knows its role (leaf, intermediate, root) and scrapes accordingly

Design Insight: By implementing federation as an extension of scraping rather than a separate mechanism, we reuse all the existing HTTP client code, retry logic, target health checking, and metrics parsing. This is a prime example of architectural composition.

FederationRule Structure:

Field Name	Type	Description
MatchMetrics	[]string	Metric name patterns to include (supports regex)
MatchLabels	[]LabelMatcher	Label conditions that must be satisfied
ExcludeMetrics	[]string	Metric name patterns to exclude
SampleLimit	int	Maximum samples per scrape to prevent overload

Cross-Instance Querying

Cross-instance querying allows PromQL queries to span multiple federation instances, providing a global view of metrics across the entire infrastructure. This extends our `QueryEngine` with **query federation capabilities**.

The federated query process works through query distribution and result merging:

1. **Query Analysis:** The `QueryPlanner` analyzes incoming PromQL queries to determine which federated instances might contain relevant data
2. **Query Distribution:** Queries are sent to relevant upstream instances in parallel using HTTP requests to their `/query` endpoints
3. **Result Merging:** Partial results from multiple instances are combined using the same aggregation logic as our local `Aggregator`
4. **Deduplication:** Time series with identical label sets from multiple sources are deduplicated based on external labels and timestamps
5. **Error Handling:** Partial failures from some instances don't prevent returning results from available instances

QueryFederator Interface:

Method Name	Parameters	Returns	Description
ExecuteFederatedQuery	ctx context.Context, query string, evalTime time.Time	*QueryResult, error	Executes query across federated instances
GetAvailableInstances	labels LabelMatcher	[]FederationTarget, error	Returns instances that might contain matching data
MergeResults	results []QueryResult	*QueryResult, error	Combines partial results from multiple instances
AddFederationTarget	target FederationTarget	error	Registers new instance for federated queries

Global View Consistency

Maintaining consistency across federated instances requires careful handling of clock skew, network partitions, and instance failures. Our approach prioritizes **availability over strict consistency** - we provide eventually consistent global views rather than strong consistency guarantees.

Clock Skew Handling: Federation instances may have slightly different system clocks, causing timestamp misalignment. We address this through:

- Configurable tolerance windows that accept samples within a reasonable time range (typically ± 1 minute)
- Timestamp normalization during federation that adjusts for known clock skew between instances
- Warning alerts when clock skew exceeds acceptable thresholds

Network Partition Resilience: When federation links fail, each instance continues operating independently:

- Local queries continue working against locally stored data
- Federation queries return partial results with warnings about unavailable instances
- Automatic reconnection attempts restore federation links when network connectivity returns
- Backfill mechanisms can catch up on missed data after reconnection

Advanced Query Features

Advanced query features extend our basic PromQL implementation with performance optimizations, computed metrics, and sophisticated analytical capabilities that would typically be added after the core system proves itself in production.

Recording Rules

Recording Rules pre-compute expensive queries and store the results as new time series, dramatically improving dashboard and alert performance for complex aggregations. Think of recording rules as creating express highway routes through our query system - frequently used complex paths get dedicated infrastructure for faster transit.

Recording rules extend our alert rule evaluation infrastructure by treating computed metrics as first-class time series:

RecordingRule Structure:

Field Name	Type	Description
Name	string	Unique identifier for the recording rule
Query	string	PromQL expression to evaluate and store
MetricName	string	Name for the generated time series
Labels	Labels	Additional labels attached to computed metrics
EvaluationInterval	time.Duration	How often to evaluate and update the recording
LastEvaluation	time.Time	Timestamp of most recent rule execution
EvaluationDuration	time.Duration	How long the last evaluation took
SamplesProduced	int64	Number of time series points generated

The recording rule evaluation process follows a continuous computation cycle:

1. **Rule Scheduling:** The `RuleEvaluator` schedules recording rules alongside alert rules, maintaining separate evaluation intervals for each
2. **Query Execution:** At each interval, the rule's PromQL query executes using our standard `QueryEngine.ExecuteInstantQuery` method
3. **Result Transformation:** Query results are converted into new time series with the specified `MetricName` and additional labels
4. **Storage Integration:** Generated samples are fed back into our `StorageEngine` through the same `Append` interface used by the scrape engine
5. **Metadata Management:** Recording rule metadata (evaluation time, sample count) is tracked for monitoring rule performance

Architecture Insight: Recording rules create a feedback loop where the query engine feeds computed results back into storage, which can then be queried again. This requires careful cycle detection to prevent infinite recursion in rule dependencies.

Common Recording Rule Patterns:

Pattern	Example Query	Use Case
Rate Calculation	<code>rate(http_requests_total[5m])</code>	Pre-compute request rates for dashboards
Quantile Aggregation	<code>histogram_quantile(0.95, rate(http_request_duration_seconds_bucket[5m]))</code>	Expensive percentile calculations
Cross-Service Aggregation	<code>sum by (service) (rate(errors_total[1m])) / sum by (service) (rate(requests_total[1m]))</code>	Service-level error rates
Resource Utilization	<code>avg by (cluster) ((cpu_usage / cpu_limit) * 100)</code>	Cluster-wide resource metrics

Query Optimization Engine

The **Query Optimization Engine** analyzes PromQL queries to identify performance improvements through query rewriting, caching, and execution plan optimization. This extends our `QueryEngine` with intelligent query planning capabilities.

Query optimization operates through multiple analysis phases:

1. **Syntax Tree Analysis:** The query's AST is analyzed to identify expensive operations like large time range scans or high-cardinality aggregations
2. **Series Cardinality Estimation:** Before execution, the optimizer estimates how many time series will be involved based on label selector specificity
3. **Execution Plan Generation:** Multiple query execution strategies are generated and their estimated costs are compared
4. **Cache Utilization:** The optimizer checks if partial results for subexpressions are available in the query result cache
5. **Query Rewriting:** Equivalent but more efficient query forms are substituted (e.g., using recording rules instead of complex aggregations)

QueryOptimizer Interface:

Method Name	Parameters	Returns	Description
OptimizeQuery	<code>ctx</code> <code>context.Context,</code> <code>query string,</code> <code>timeRange TimeRange</code>	<code>*OptimizedQuery, error</code>	Returns optimized execution plan
EstimateComplexity	<code>query string,</code> <code>timeRange TimeRange</code>	<code>*ComplexityEstimate,</code> <code>error</code>	Predicts query resource requirements
SuggestRecordingRules	<code>queries []string,</code> <code>frequency</code> <code>time.Duration</code>	<code>[]RecordingRuleSuggestion,</code> <code>error</code>	Identifies queries that would benefit from pre-computation
UpdateStatistics	<code>query string,</code> <code>duration</code> <code>time.Duration,</code> <code>seriesCount int</code>	<code>error</code>	Records query performance for future optimization

Query Result Caching significantly improves repeated query performance by storing computed results with cache keys based on query text, time range, and evaluation timestamp:

- **Immutable Range Caching:** Queries for historical time ranges (older than the staleness threshold) return identical results and can be cached indefinitely
- **Partial Result Caching:** Large time range queries are broken into smaller chunks, with completed chunks cached while only the most recent chunk is recomputed
- **Cache Invalidation:** Recording rule updates and data ingestion trigger selective cache invalidation based on affected metric names and label sets
- **Memory Management:** LRU eviction prevents cache from consuming excessive memory, with configurable size limits and TTL policies

Advanced Aggregation Functions

Advanced aggregation functions extend our basic `sum`, `avg`, `max`, `min`, and `count` operations with sophisticated statistical and mathematical capabilities commonly needed in production monitoring environments.

Statistical Aggregations:

Function Name	Parameters	Returns	Description
<code>stddev_over_time</code>	<code>vector, duration</code>	<code>vector</code>	Standard deviation of values over time window
<code>quantile_over_time</code>	<code>quantile, vector, duration</code>	<code>vector</code>	Arbitrary quantile calculation over time
<code>mad_over_time</code>	<code>vector, duration</code>	<code>vector</code>	Median absolute deviation for outlier detection
<code>predict_linear</code>	<code>vector, duration</code>	<code>vector</code>	Linear regression prediction of future values
<code>deriv</code>	<code>vector</code>	<code>vector</code>	Per-second derivative calculation

Advanced Grouping Operations:

Function Name	Parameters	Returns	Description
<code>topk</code>	<code>k, vector</code>	<code>vector</code>	Top K series by value
<code>bottomk</code>	<code>k, vector</code>	<code>vector</code>	Bottom K series by value
<code>count_values</code>	<code>string, vector</code>	<code>vector</code>	Count occurrences of each distinct value
<code>group_by_interval</code>	<code>vector, duration</code>	<code>vector</code>	Time-based grouping for irregular series

These advanced functions are implemented as extensions to our existing `Aggregator` component, following the same interface patterns but with more complex mathematical operations:

- Streaming Computation:** Statistical functions process samples in streaming fashion to handle large time ranges without loading entire datasets into memory
- Numerical Stability:** Implementations use numerically stable algorithms (e.g., Welford's method for standard deviation) to prevent precision loss
- Missing Data Handling:** Advanced functions include configurable strategies for handling missing or null values in time series
- Performance Optimization:** Expensive computations are candidates for automatic recording rule generation when used frequently

⚠ Pitfall: Query Complexity Explosion

Advanced query features can easily create queries that consume excessive system resources. A common mistake is allowing unrestricted use of expensive functions like `quantile_over_time` over large time ranges with high-cardinality label sets. This can consume gigabytes of memory and take minutes to execute.

Prevention Strategy: Implement query complexity estimation that considers the number of time series, time range duration, and computational complexity of functions. Reject or warn about queries that exceed resource limits:

```

Estimated complexity: 50GB memory, 120 seconds execution
This query spans 10,000 time series over 7 days with quantile calculation.
Consider: reducing time range, adding more specific label filters, or creating a recording rule.

```

Implementation Guidance

The extensibility features build upon our existing component architecture through composition and interface extension rather than core system modification. This section provides practical guidance for implementing these advanced capabilities.

Technology Recommendations:

Component	Simple Option	Advanced Option
Rule Storage	YAML files + file watching	etcd cluster with version control
Notification Delivery	HTTP webhooks + retry queues	Message queue system (RabbitMQ/Kafka)
Federation Transport	HTTP scraping (existing)	gRPC streaming for high-volume federation
Query Caching	In-memory LRU cache	Redis cluster with persistence
Result Storage	Extend existing StorageEngine	Separate OLAP system for recording rules

Recommended File Structure for Extensions:

```

project-root/
  internal/
    alerting/
      rule_evaluator.go      ← Alert rule evaluation engine
      notification_manager.go ← Multi-channel notification delivery
      alert_state.go          ← Alert lifecycle management
      rule_evaluator_test.go  ← Comprehensive rule testing
    federation/
      federation_scraper.go ← Extends ScrapeEngine for federation
      query_federator.go     ← Cross-instance query distribution
      topology_manager.go    ← Federation hierarchy management
    query_advanced/
      recording_rules.go     ← Pre-computed metrics engine
      query_optimizer.go     ← Query performance optimization
      advanced_aggregations.go ← Statistical and mathematical functions
      query_cache.go          ← Result caching with invalidation
  configs/
    alerting_rules.yml      ← Alert rule definitions
    recording_rules.yml      ← Recording rule definitions
    federation.yml           ← Federation topology configuration

```

Alert Rule Evaluator Infrastructure (Complete Implementation):

GO

```
package alerting

import (
    "context"
    "fmt"
    "sync"
    "time"
    "path/to/project/internal/query"
    "path/to/project/internal/storage"
)

// AlertState represents the current state of an alert rule

type AlertState int

const (
    AlertStateInactive AlertState = iota
    AlertStatePending
    AlertStateFiring
    AlertStateResolved
)

// AlertRule defines a condition to monitor and alert on

type AlertRule struct {

    Name          string      `yaml:"name"`
    Query         string      `yaml:"query"`
    Duration      time.Duration `yaml:"for"`
    Labels        map[string]string `yaml:"labels"`
    Annotations   map[string]string `yaml:"annotations"`
    EvaluationInterval time.Duration `yaml:"interval"`
}
```

```

// Internal state tracking

State      AlertState    `yaml:"-"`
ActiveSince time.Time    `yaml:"-"`
ResolvedAt *time.Time   `yaml:"-"`
mutex      sync.RWMutex `yaml:"-"`
}

// AlertInstance represents a specific firing alert

type AlertInstance struct {

    RuleName     string      `json:"rule_name"`
    Labels       map[string]string `json:"labels"`
    State        AlertState   `json:"state"`
    Value        float64     `json:"value"`
    StartsAt    time.Time    `json:"starts_at"`
    EndsAt      *time.Time   `json:"ends_at,omitempty"`
    GeneratorURL string     `json:"generator_url"`
    Fingerprint  uint64     `json:"fingerprint"`
}

// RuleEvaluator manages alert rule evaluation and state tracking

type RuleEvaluator struct {

    rules      map[string]*AlertRule
    queryEngine *query.QueryEngine
    storage    *storage.StorageEngine

    activeAlerts  map[uint64]*AlertInstance // fingerprint -> alert
    notificationCh chan<- AlertInstance   // channel for sending notifications

    evaluationTicker *time.Ticker
}

```

```

stopCh      chan struct{}`

mutex       sync.RWMutex

}

// NewRuleEvaluator creates a rule evaluator with notification channel

func NewRuleEvaluator(queryEngine *query.QueryEngine, storage *storage.StorageEngine,
                      notificationCh chan<- AlertInstance) *RuleEvaluator {
    return &RuleEvaluator{
        rules:          make(map[string]*AlertRule),
        queryEngine:   queryEngine,
        storage:       storage,
        activeAlerts:  make(map[uint64]*AlertInstance),
        notificationCh: notificationCh,
        stopCh:        make(chan struct{}),
    }
}

// LoadRulesFromFile loads alert rules from YAML configuration

func (re *RuleEvaluator) LoadRulesFromFile(filename string) error {
    // TODO 1: Read YAML file containing alert rule definitions

    // TODO 2: Parse YAML into []AlertRule using yaml.Unmarshal

    // TODO 3: Validate each rule: check query syntax, ensure positive duration

    // TODO 4: Call AddRule for each valid rule to register it

    // TODO 5: Return error if any rule validation fails

    return nil
}

// AddRule registers a new alert rule for evaluation

func (re *RuleEvaluator) AddRule(rule *AlertRule) error {
    // TODO 1: Validate rule.Query by parsing it with queryEngine
}

```

```

// TODO 2: Ensure rule.Name is unique among existing rules

// TODO 3: Set default EvaluationInterval if not specified

// TODO 4: Store rule in re.rules map with name as key

// TODO 5: Log successful rule registration

return nil

}

// Start begins periodic rule evaluation

func (re *RuleEvaluator) Start(ctx context.Context, evaluationInterval time.Duration) error {

// TODO 1: Create ticker with specified evaluation interval

// TODO 2: Start goroutine that calls EvaluateAll on each tick

// TODO 3: Handle context cancellation to stop evaluation loop

// TODO 4: Ensure proper cleanup of ticker and goroutines

return nil

}

// EvaluateAll evaluates all registered rules at the specified time

func (re *RuleEvaluator) EvaluateAll(ctx context.Context, evalTime time.Time) ([]AlertInstance, error) {

// TODO 1: Iterate through all rules in re.rules map

// TODO 2: For each rule, execute rule.Query using queryEngine.ExecuteInstantQuery

// TODO 3: Process query results to determine if alert condition is met

// TODO 4: Update rule state (inactive -> pending -> firing) based on duration

// TODO 5: Generate AlertInstance for newly firing alerts

// TODO 6: Send new/resolved alerts to notification channel

// TODO 7: Return list of all currently active alert instances

return nil, nil

}

```

Federation Target Discovery (Core Logic Skeleton):

```
package federation

import (
    "context"
    "net/http"
    "time"
    "path/to/project/internal/scrape"
)

// FederationTarget represents an upstream metrics instance to federate from

type FederationTarget struct {

    URL          string      `yaml:"url"`
    MatchRules   []FederationRule `yaml:"match_rules"`
    ScrapeInterval time.Duration `yaml:"scrape_interval"`
    ExternalLabels map[string]string `yaml:"external_labels"`
}

// FederationRule defines which metrics to pull from upstream instance

type FederationRule struct {

    MatchMetrics  []string      `yaml:"match_metrics"`
    MatchLabels   []query.LabelMatcher `yaml:"match_labels"`
    ExcludeMetrics []string      `yaml:"exclude_metrics"`
    SampleLimit   int           `yaml:"sample_limit"`
}

// FederationScraper extends ScrapeEngine to handle federation endpoints

type FederationScraper struct {

    baseScraper *scrape.ScrapeEngine
    httpClient  *http.Client
    targets     map[string]FederationTarget
}
```

GO

```

    mutex      sync.RWMutex

}

// ScrapeFederationTarget pulls metrics from upstream federation endpoint

func (fs *FederationScraper) ScrapeFederationTarget(ctx context.Context, target FederationTarget) error {

    // TODO 1: Build federation URL with match[] parameters from target.MatchRules

    // TODO 2: Create HTTP request with proper timeout and user-agent headers

    // TODO 3: Execute HTTP GET request to /federate endpoint

    // TODO 4: Parse response body as Prometheus exposition format

    // TODO 5: Apply external labels from target.ExternalLabels to all metrics

    // TODO 6: Filter metrics based on target.MatchRules inclusion/exclusion

    // TODO 7: Send filtered samples to storage engine through existing pipeline

    return nil
}

// UpdateFederationTargets refreshes the list of upstream instances

func (fs *FederationScraper) UpdateFederationTargets(targets []FederationTarget) error {

    // TODO 1: Validate each target URL and match rules

    // TODO 2: Check for target URL uniqueness to prevent duplicates

    // TODO 3: Update fs.targets map with new target configuration

    // TODO 4: Schedule scraping for new targets using existing scheduler

    // TODO 5: Remove scraping for targets no longer in the list

    return nil
}

```

Query Result Caching Infrastructure (Complete Implementation):

GO

```
package query_advanced

import (
    "crypto/sha256"
    "encoding/hex"
    "sync"
    "time"
    "container/list"
)

// CacheKey uniquely identifies a cached query result

type CacheKey struct {
    Query      string      `json:"query"`
    Start      time.Time   `json:"start"`
    End        time.Time   `json:"end"`
    EvalTime   time.Time   `json:"eval_time"`
}

// String returns string representation for hashing

func (ck CacheKey) String() string {
    return fmt.Sprintf("%s:%d:%d:%d", ck.Query, ck.Start.Unix(), ck.End.Unix(),
    ck.EvalTime.Unix())
}

// Hash returns SHA256 hash of cache key for map storage

func (ck CacheKey) Hash() string {
    hasher := sha256.New()
    hasher.Write([]byte(ck.String()))
    return hex.EncodeToString(hasher.Sum(nil))
}
```

```

// CacheEntry stores cached query results with metadata

type CacheEntry struct {

    Key          CacheKey           `json:"key"`

    Result       *query.QueryResult `json:"result"`

    CachedAt     time.Time         `json:"cached_at"`

    AccessTime   time.Time         `json:"last_access"`

    TTL          time.Duration     `json:"ttl"`

    Size         int64              `json:"size_bytes"`

    // LRU list element for eviction tracking

    element *list.Element `json:"-"`
}

// QueryCache provides LRU caching for query results

type QueryCache struct {

    entries      map[string]*CacheEntry // hash -> entry

    lruList      *list.List           // LRU eviction order

    maxSize      int64                // maximum cache size in bytes

    currentSize  int64                // current cache size in bytes

    mutex        sync.RWMutex         // concurrent access protection

    // Metrics for cache performance monitoring

    hits         int64
    misses       int64
    evictions    int64

}

// NewQueryCache creates cache with specified maximum size

func NewQueryCache(maxSizeBytes int64) *QueryCache {

```

```

    return &QueryCache{

        entries: make(map[string]*CacheEntry),
        lruList: list.New(),
        maxSize: maxSizeBytes,
    }
}

// Get retrieves cached result if available and not expired

func (qc *QueryCache) Get(key CacheKey) (*query.QueryResult, bool) {

    // TODO 1: Calculate hash of cache key for map lookup

    // TODO 2: Check if entry exists and is not expired (cachedAt + TTL > now)

    // TODO 3: Update entry.AccessTime and move to front of LRU list

    // TODO 4: Increment hit/miss counters for monitoring

    // TODO 5: Return deep copy of cached result to prevent modification

    return nil, false
}

// Put stores query result in cache with automatic eviction

func (qc *QueryCache) Put(key CacheKey, result *query.QueryResult, ttl time.Duration) {

    // TODO 1: Calculate size of result for memory tracking

    // TODO 2: Check if adding entry would exceed maxSize limit

    // TODO 3: Evict LRU entries until sufficient space is available

    // TODO 4: Create CacheEntry with current timestamp and TTL

    // TODO 5: Store entry in map and add to front of LRU list

    // TODO 6: Update currentSize and evictions counter
}

// InvalidateByMetric removes cached entries that depend on specific metric

func (qc *QueryCache) InvalidateByMetric(metricName string) int {

    // TODO 1: Iterate through all cached entries
}

```

```

    // TODO 2: Parse each entry's query to extract referenced metric names

    // TODO 3: Remove entries whose queries reference the invalidated metric

    // TODO 4: Update cache size counters and LRU list

    // TODO 5: Return number of entries invalidated

    return 0
}

```

Milestone Checkpoints for Extensions:

Alerting System Checkpoint:

- Load sample alert rules from YAML: `go run cmd/server/main.go --config=configs/alerting_test.yml`
- Expected: Server starts and logs "Loaded 3 alert rules"
- Verify rule evaluation: Send test metrics that trigger conditions, check alert state API
- Notification testing: Configure webhook endpoint, verify alert notifications are delivered
- Signs of problems: Rules not loading (check YAML syntax), queries failing (validate PromQL), notifications not sending (check webhook URL)

Federation Checkpoint:

- Configure federation target: Add federation section to config pointing to existing Prometheus
- Expected: Federated metrics appear in local storage with external labels
- Verify query spanning: Execute queries that combine local and federated metrics
- Performance check: Monitor federation scrape duration and memory usage
- Signs of problems: No federated metrics (check upstream /federate endpoint), label conflicts (verify external labels), high memory usage (reduce match rules scope)

Query Optimization Checkpoint:

- Enable query caching: Set `cache_enabled=true` in query configuration
- Execute expensive query twice: Second execution should be significantly faster
- Recording rule test: Create recording rule, verify pre-computed metrics are stored
- Cache invalidation: Ingest new data, verify cache entries are properly invalidated
- Performance monitoring: Check cache hit ratio, query duration improvements
- Signs of problems: No cache hits (check TTL settings), memory growth (verify eviction), incorrect results (cache invalidation bugs)

Debugging Tips for Extensions:

Symptom	Likely Cause	How to Diagnose	Fix
Alert rules not firing	Query returns no results	Check rule query in query UI	Verify label selectors match actual metrics
Notifications not delivered	Webhook endpoint unreachable	Check notification manager logs	Verify webhook URL, check network connectivity
Federation not working	Upstream instance not accessible	Test federation URL manually	Check upstream /federate endpoint, verify network
Cached queries wrong results	Cache not invalidated on data ingestion	Check cache invalidation logs	Ensure cache invalidation on metric writes
Recording rules consuming memory	Rule generates high-cardinality metrics	Monitor rule evaluation metrics	Add more specific label selectors to rule query
Federation causing high memory	Too many metrics being federated	Check federation match rules scope	Restrict match rules to essential metrics only

Glossary

Milestone(s): This section provides comprehensive definitions for technical terms used throughout all four milestones: Metrics Data Model (1), Scrape Engine (2), Time Series Storage (3), and Query Engine (4).

This glossary serves as the definitive reference for technical terminology, concepts, and vocabulary used throughout the metrics collection system design. Understanding these terms is essential for implementing and maintaining the system effectively. Each term includes its definition, context of use, and relationships to other concepts where applicable.

Core Concepts and Terminology

The metrics collection system introduces several domain-specific concepts that may be unfamiliar to developers coming from other backgrounds. This section establishes a common vocabulary for discussing system behavior, implementation details, and operational characteristics.

Time Series and Data Model Terms

Term	Definition	Context
time series	A sequence of timestamped values identified by a unique combination of metric name and label set	The fundamental data structure stored and queried by the system
cardinality	The number of unique time series created by all combinations of metric name and label values	Critical metric for memory usage and performance planning
label explosion	Exponential growth in cardinality when high-cardinality labels create excessive time series combinations	Primary cause of memory exhaustion and query performance degradation
metric type	The semantic category that defines how a metric behaves: counter, gauge, histogram, or summary	Determines valid operations and query interpretations
observability	The ability to understand system behavior and health through external metrics and monitoring	The broader goal that metrics collection systems enable
exposition format	The Prometheus text-based format used to expose metrics over HTTP endpoints	Standard format parsed by the scrape engine
retention period	The duration for which historical time series data is preserved before automatic deletion	Balances storage costs with historical analysis needs

Scraping and Collection Terms

Term	Definition	Context
pull-based scraping	A metrics collection model where the monitoring system actively retrieves metrics from targets	Contrasts with push-based systems where applications send metrics
scrape target	An HTTP endpoint that exposes metrics in the exposition format for collection	The source of all metrics data in the system
service discovery	Automatic detection and configuration of scrape targets from dynamic sources like Kubernetes	Enables monitoring of dynamic environments without manual configuration
target health	The availability and response status of a scrape endpoint based on recent collection attempts	Used for alerting and operational visibility
scrape interval	The frequency at which metrics are collected from each target	Balances data freshness with system load
scrape timeout	The maximum duration allowed for a single HTTP metrics collection request	Prevents slow targets from blocking the scrape engine

Storage and Compression Terms

Term	Definition	Context
Gorilla compression	A specialized time series compression algorithm using delta-of-delta timestamp encoding and XOR value compression	Reduces storage requirements to approximately 1.37 bytes per sample
WAL	Write-Ahead Log - a durability mechanism that records intended operations before they are applied	Enables crash recovery without data loss
chunk	A compressed block of time series samples with fixed time boundaries and maximum sample counts	The unit of storage and compression for time series data
index consistency	The property that all indexes correctly map to existing time series data without orphaned references	Critical for query correctness and system reliability
downsampling	The process of reducing data resolution by aggregating high-frequency samples into lower-frequency summaries	Enables long-term storage with reduced space requirements

Querying and Analysis Terms

Term	Definition	Context
PromQL	Prometheus Query Language - a functional query language for selecting and aggregating time series data	The primary interface for data analysis and alerting
AST	Abstract Syntax Tree - the parsed representation of a PromQL query used for execution	Internal structure created by the query parser
label selector	Filter criteria that match time series based on label name and value patterns	Fundamental mechanism for narrowing query scope
aggregation	Mathematical combination of multiple time series values into summary statistics	Enables analysis across multiple dimensions and instances
range query	A query that returns multiple data points across a specified time window	Used for graphing and trend analysis
instant query	A query that returns a single point-in-time value for each matching time series	Used for alerting and current state monitoring
interpolation	The process of estimating values at query timestamps when no exact sample exists	Handles alignment between sample timestamps and query evaluation times
staleness	The threshold beyond which a data point is considered too old to use in query results	Prevents outdated values from appearing in current analysis

System Architecture and Component Terms

The metrics system consists of several interconnected components that coordinate to provide end-to-end metrics collection and analysis capabilities. Understanding the role and terminology of each component is essential for system implementation and maintenance.

Component and Coordination Terms

Term	Definition	Context
backpressure	A flow control mechanism that slows down producers when consumers cannot keep up with the data rate	Prevents memory exhaustion during high-volume ingestion
pipeline coordination	Managing the flow of data between system components with proper synchronization and error handling	Ensures reliable data flow from scraping through storage to querying
concurrency control	Managing simultaneous access to shared resources without data corruption or race conditions	Critical for multi-threaded components like the storage engine
graceful degradation	Maintaining system availability with reduced functionality during overload or partial failures	Allows continued operation when some components are impaired
write batching	Combining multiple small write operations into larger, more efficient batch operations	Improves storage throughput and reduces overhead
lock granularity	The scope and size of data protected by each synchronization primitive	Affects both performance and correctness in concurrent systems
copy-on-write	An optimization where reads access shared data while writes create private copies	Reduces contention between readers and writers
readers-writer lock	A synchronization primitive allowing multiple concurrent readers or a single writer	Optimizes for read-heavy workloads common in time series systems
channel-based coordination	Using Go channels to coordinate communication and synchronization between goroutines	Primary concurrency pattern in Go-based implementations

Operational and Monitoring Terms

Term	Definition	Context
resource monitoring	Tracking system resource usage to prevent exhaustion and enable proactive limits	Includes memory, disk space, CPU, and network bandwidth monitoring
resource exhaustion	The depletion of system resources that can cause component failures or degraded performance	Common operational issue requiring monitoring and response procedures
emergency retention	Aggressive data deletion triggered automatically during disk space crises	Last-resort mechanism to prevent complete system failure
consistency validation	Verification that stored data matches expected invariants and relationships	Includes index-data alignment and checksum validation
partial scrape success	The ability to preserve valid metrics while rejecting malformed ones from the same target	Improves system resilience to target-side issues
circuit breaker	A failure protection pattern that temporarily blocks requests to failing dependencies	Prevents cascade failures and enables faster recovery

Testing and Development Terms

Building a reliable metrics collection system requires comprehensive testing strategies and development practices. These terms describe the approaches and techniques used to validate system correctness and performance.

Testing Methodology Terms

Term	Definition	Context
property-based testing	Testing approach using automatically generated inputs that satisfy specified properties	Discovers edge cases that manual test cases might miss
fault injection	Deliberately introducing errors and failures to test error handling and recovery mechanisms	Validates system behavior under adverse conditions
statistical validation	Comparing computed results against mathematically expected values using statistical methods	Ensures aggregation functions and calculations produce correct results
race detector	A tool for detecting unsynchronized access to shared memory in concurrent programs	Essential for validating thread safety in Go programs
benchmark testing	Measuring performance characteristics like throughput, latency, and resource usage	Validates that system meets performance requirements
mock objects	Test doubles that simulate external dependencies with controllable, predictable behavior	Enables isolated testing of individual components

Advanced Features and Extensions

The metrics system architecture supports advanced capabilities that extend beyond basic collection and querying. Understanding these concepts is important for system evolution and operational sophistication.

Advanced System Capabilities

Term	Definition	Context
distributed tracing	End-to-end request tracking that follows operations across multiple system components	Enables debugging and performance analysis of complex operations
structured logging	Machine-readable log format with consistent fields and structured data	Improves observability and enables automated log analysis
health checking	Systematic monitoring of component status with standardized health reporting	Provides operational visibility and enables automated response
federation	Connecting multiple metrics instances into a coordinated cluster for scalability	Enables horizontal scaling beyond single-instance limits
recording rules	Pre-computed expensive queries that are stored as new time series	Improves query performance for commonly used aggregations
query optimization	Analyzing queries to identify performance improvements and suggest alternatives	Reduces resource consumption and improves user experience
hierarchical federation	A pull-based federation approach that extends the scraping concept to other metrics instances	Maintains consistency with the core pull-based architecture
cross-instance querying	PromQL queries that span multiple federated instances and merge results	Provides unified view of metrics across distributed deployments

Alerting and Notification Terms

Term	Definition	Context
alert rule evaluation	Continuous assessment of PromQL expressions against time series data to detect alert conditions	Core function of alerting systems built on the metrics platform
notification manager	System responsible for delivering alert notifications through multiple channels	Handles routing, rate limiting, and delivery confirmation
rule evaluator	Component that manages alert rule execution, state tracking, and notification generation	Coordinates between query engine and notification systems
alert state	The current condition of an alert rule: inactive, pending, firing, or resolved	Determines notification behavior and operational response

Performance and Caching Terms

Term	Definition	Context
query result caching	Storing computed PromQL results with appropriate cache keys for repeated queries	Improves response times and reduces computational load
cache invalidation	Removing cached entries when underlying time series data changes	Maintains cache consistency while preserving performance benefits
query complexity estimation	Predicting resource requirements before query execution to enable limits and optimization	Prevents resource exhaustion from expensive queries
LRU eviction	Least Recently Used cache entry removal strategy for managing memory usage	Balances cache hit rates with memory consumption
query federation	Distributing queries across multiple instances and merging partial results	Enables querying of datasets larger than single-instance capacity

Data Types and Structures

The metrics system defines specific data structures and types that represent different aspects of the system. Understanding these types and their relationships is crucial for implementation.

Core Data Type Categories

Category	Purpose	Key Types
Configuration Types	System and component configuration	<code>Config</code> , <code>ScrapeConfig</code> , <code>StorageConfig</code> , <code>QueryConfig</code>
Metric Types	Time series data representation	<code>Counter</code> , <code>Gauge</code> , <code>Histogram</code> , <code>Sample</code> , <code>Labels</code>
Storage Types	Data persistence and compression	<code>CompressedChunk</code> , <code>WriteAheadLog</code> , <code>InvertedIndexes</code> , <code>GorillaCompressor</code>
Query Types	Query execution and results	<code>QueryEngine</code> , <code>ExpressionParser</code> , <code>ASTNode</code> , <code>QueryResult</code>
Coordination Types	Component interaction and flow control	<code>PipelineCoordinator</code> , <code>QueryCoordinator</code> , <code>SystemCoordinator</code>
Health and Monitoring Types	System status and diagnostics	<code>ComponentHealth</code> , <code>HealthChecker</code> , <code>SystemHealth</code> , <code>PerformanceMonitor</code>
Error and Control Types	Error handling and flow control	<code>MetricsError</code> , <code>CircuitBreaker</code> , <code>QueryLimiter</code>
Testing Types	Development and validation support	<code>MockHTTPTarget</code> , <code>TimeSeriesGenerator</code> , <code>PerformanceMonitor</code>

Constants and Configuration Values

The system defines standard constants for timeouts, limits, and default behaviors that ensure consistent operation across different deployment environments.

System Default Constants

Constant	Value	Purpose
DEFAULT_SCRAPE_INTERVAL	15 seconds	Standard frequency for metrics collection when not otherwise specified
DEFAULT_SCRAPE_TIMEOUT	10 seconds	Maximum duration for HTTP scrape requests to prevent blocking
DEFAULT_RETENTION_PERIOD	30 days	Standard data retention duration balancing storage cost and utility
DEFAULT_QUERY_TIMEOUT	30 seconds	Maximum query execution time to prevent resource exhaustion

Health and Status Constants

Constant	Description	Usage
HealthUp	Component is healthy and operational	Indicates normal component function
HealthDown	Component has failed or is unreachable	Indicates complete component failure
HealthDegraded	Component is operational but experiencing issues	Indicates partial functionality or performance problems

Query and Matching Constants

Constant	Description	Usage
MatchEqual	Exact string equality matching	Standard label value filtering
MatchNotEqual	String inequality matching	Exclusion-based label filtering
MatchRegex	Regular expression matching	Pattern-based label filtering
MatchNotRegex	Negative regular expression matching	Pattern-based label exclusion

Error Types and Handling Categories

The system categorizes errors to enable appropriate handling strategies and recovery mechanisms. Understanding error classifications helps in building robust error handling logic.

Error Classification System

Error Type	Description	Handling Strategy
ErrorTypeTransient	Temporary errors that may succeed on retry with backoff	Implement exponential backoff retry logic
ErrorTypePermanent	Permanent errors that will not succeed on retry	Log error and fail fast without retry attempts
ErrorTypeRateLimit	Rate limiting errors requiring backoff	Implement longer backoff periods and reduce request rate
ErrorTypeResource	Resource exhaustion requiring different handling	Implement circuit breakers and shed load

Circuit Breaker States

State	Description	Behavior
CircuitClosed	Normal operation allowing all requests	Monitor failure rate and transition to open on threshold
CircuitOpen	Failure protection mode rejecting all requests	Block requests and transition to half-open after timeout
CircuitHalfOpen	Testing mode allowing limited requests to test recovery	Allow single request to test if service has recovered

Alert and Rule Management

The alerting system introduces additional terminology for rule management, notification handling, and alert state tracking.

Alert State Management

State	Description	Transitions
AlertStateInactive	Alert rule condition is not currently met	Transitions to Pending when condition becomes true
AlertStatePending	Condition is met but duration threshold not yet exceeded	Transitions to Firing after duration or Inactive if condition clears
AlertStateFiring	Condition has been met for the required duration	Transitions to Resolved when condition clears
AlertStateResolved	Previously firing alert where condition is no longer met	Transitions to Inactive after notification sent

Implementation Guidance

This section provides practical guidance for implementing the metrics collection system using the defined terminology and concepts.

Technology Recommendations

Component	Simple Option	Advanced Option
HTTP Server	Go net/http with ServeMux	Gin or Echo framework with middleware
Configuration	YAML with gopkg.in/yaml.v3	Viper with environment variable override
Logging	Standard log/slog package	Structured logging with zerolog or logrus
Testing	Standard testing package	Testify for assertions and table-driven tests
Benchmarking	Go built-in benchmarks	Continuous benchmarking with benchstat

File Structure Organization

The metrics collection system should be organized into clear modules that separate concerns and enable independent development of each component:

```

metrics-system/
  cmd/
    metrics-server/
      main.go
      ← Application entry point
  internal/
    config/
      config.go
      defaults.go
      ← Configuration loading and validation
      ← Default constant definitions
  metrics/
    types.go
    labels.go
    validator.go
    ← Metric type definitions (Counter, Gauge, Histogram)
    ← Label handling and validation
    ← Cardinality and label validation
  scrape/
    engine.go
    target.go
    discovery.go
    parser.go
    ← Main scrape engine coordination
    ← Target health and state management
    ← Service discovery implementations
    ← Metrics exposition format parsing
  storage/
    engine.go
    compression.go
    wal.go
    index.go
    chunk.go
    ← Storage engine coordination
    ← Gorilla compression implementation
    ← Write-ahead log implementation
    ← Inverted index management
    ← Compressed chunk handling
  query/
    engine.go
    parser.go
    selector.go
    aggregator.go
    executor.go
    ← Query engine coordination
    ← PromQL parsing and AST construction
    ← Label selector implementation
    ← Aggregation function implementation
    ← Range query execution
  coordination/
    pipeline.go
    system.go
    health.go
    ← Pipeline coordination between components
    ← System-level coordination
    ← Health checking and status reporting
  errors/
    types.go
    circuit.go
    ← Error type definitions
    ← Circuit breaker implementation
  pkg/
    client/
    ← Public client libraries (if needed)
  test/
    integration/
    testdata/
    ← End-to-end integration tests
    ← Test fixtures and sample data
  docs/
    glossary.md
    ← This document

```

Core Infrastructure Implementation

Logger Implementation Starter Code:

GO

```
package internal

import (
    "context"
    "log/slog"
    "os"
)

// Logger provides structured logging with component identification

type Logger struct {
    component string
    logger    *slog.Logger
}

// NewLogger creates a structured logger for a specific component

func NewLogger(component string) *Logger {
    handler := slog.NewJSONHandler(os.Stdout, &slog.HandlerOptions{
        Level: slog.LevelInfo,
    })

    logger := slog.New(handler).With("component", component)

    return &Logger{
        component: component,
        logger:    logger,
    }
}

// Info logs informational messages with structured fields

func (l *Logger) Info(msg string, fields ...any) {
```

```
    l.logger.Info(msg, fields...)
}

// Error logs error messages with structured fields

func (l *Logger) Error(msg string, fields ...any) {
    l.logger.Error(msg, fields...)
}

// Debug logs debug messages with structured fields

func (l *Logger) Debug(msg string, fields ...any) {
    l.logger.Debug(msg, fields...)
}

// WithContext adds trace context to logger if available

func (l *Logger) WithContext(ctx context.Context) *Logger {
    // TODO: Extract trace ID from context if tracing is enabled
    // TODO: Add trace ID as a structured field to logger
    return l
}
```

Configuration Loading Implementation:

```
package config

import (
    "fmt"
    "os"
    "time"
    "gopkg.in/yaml.v3"
)

// Config represents the complete system configuration

type Config struct {
    Scrape  ScrapeConfig `yaml:"scrape"`
    Storage StorageConfig `yaml:"storage"`
    Query   QueryConfig  `yaml:"query"`
}

type ScrapeConfig struct {
    ScrapeInterval  time.Duration `yaml:"scrape_interval"`
    ScrapeTimeout   time.Duration `yaml:"scrape_timeout"`
    StaticConfigs  []StaticConfig `yaml:"static_configs"`
}

type StorageConfig struct {
    DataDirectory     string      `yaml:"data_directory"`
    RetentionPeriod  time.Duration `yaml:"retention_period"`
    CompressionEnabled bool       `yaml:"compression_enabled"`
}

type QueryConfig struct {
    QueryTimeout     time.Duration `yaml:"query_timeout"`
    MaxSeries        int          `yaml:"max_series"`
}
```

GO

```
    MaxRangeDuration time.Duration `yaml:"max_range_duration"`

}

type StaticConfig struct {

    Targets []string           `yaml:"targets"`

    Labels map[string]string `yaml:"labels"`

}

// Constants for default values

const (

    DEFAULT_SCRAPE_INTERVAL     = 15 * time.Second
    DEFAULT_SCRAPE_TIMEOUT      = 10 * time.Second
    DEFAULT_RETENTION_PERIOD    = 30 * 24 * time.Hour
    DEFAULT_QUERY_TIMEOUT        = 30 * time.Second

)

// LoadFromFile reads configuration from YAML file

func LoadFromFile(filename string) (*Config, error) {

    data, err := os.ReadFile(filename)

    if err != nil {

        return nil, fmt.Errorf("reading config file: %w", err)

    }

    var config Config

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

        return nil, fmt.Errorf("parsing config YAML: %w", err)

    }

    config.SetDefaults()

    return &config, nil
}
```

```
}

// SetDefaults populates default values for unspecified configuration

func (c *Config) SetDefaults() {

    if c.Scrape.ScrapeInterval == 0 {

        c.Scrape.ScrapeInterval = DEFAULT_SCRAPE_INTERVAL

    }

    if c.Scrape.ScrapeTimeout == 0 {

        c.Scrape.ScrapeTimeout = DEFAULT_SCRAPE_TIMEOUT

    }

    if c.Storage.RetentionPeriod == 0 {

        c.Storage.RetentionPeriod = DEFAULT_RETENTION_PERIOD

    }

    if c.Query.QueryTimeout == 0 {

        c.Query.QueryTimeout = DEFAULT_QUERY_TIMEOUT

    }

    if c.Storage.DataDirectory == "" {

        c.Storage.DataDirectory = "./data"

    }

    if c.Query.MaxSeries == 0 {

        c.Query.MaxSeries = 10000

    }

}
```

Core Logic Skeletons

Health Status Enumeration:

```
package health
```

```
// HealthStatus represents the operational state of system components
```

```
type HealthStatus int
```

```
const (
```

```
    HealthUp HealthStatus = iota
```

```
    HealthDown
```

```
    HealthDegraded
```

```
)
```

```
func (h HealthStatus) String() string {
```

```
    switch h {
```

```
        case HealthUp:
```

```
            return "UP"
```

```
        case HealthDown:
```

```
            return "DOWN"
```

```
        case HealthDegraded:
```

```
            return "DEGRADED"
```

```
        default:
```

```
            return "UNKNOWN"
```

```
    }
```

```
}
```

GO

Error Type System:

```
package errors

import "fmt"

// ErrorType categorizes errors for appropriate handling

type ErrorType int

const (
    ErrorTypeTransient ErrorType = iota
    ErrorTypePermanent
    ErrorTypeRateLimit
    ErrorTypeResource
)

// MetricsError provides structured error information

type MetricsError struct {

    Type      ErrorType
    Component string
    Operation string
    Message   string
    Cause     error
}

func (e *MetricsError) Error() string {
    return fmt.Sprintf("[%s:%s] %s: %s", e.Component, e.Operation, e.Message, e.Cause)
}

func (e *MetricsError) Unwrap() error {
    return e.Cause
}
```

GO

Testing Infrastructure

Mock HTTP Target for Testing:

```
package testing
```

GO

```
import (
    "fmt"
    "math/rand"
    "net/http"
    "net/http/httpptest"
    "strings"
    "sync"
    "time"
)
```

```
// MockHTTPPTarget provides controllable HTTP endpoint for testing scrape engine
```

```
type MockHTTPPTarget struct {
    server     *httpptest.Server
    metrics    []string
    delay      time.Duration
    errorRate float64
    mu         sync.RWMutex
}
```

```
// NewMockHTTPPTarget creates a new mock target with default behavior
```

```
func NewMockHTTPPTarget() *MockHTTPPTarget {
    target := &MockHTTPPTarget{
        metrics:    []string{},
        delay:      0,
        errorRate:  0.0,
    }
}
```

```
target.server = httpptest.NewServer(http.HandlerFunc(target.handleMetrics))
```

```
    return target
}

// URL returns the HTTP URL of the mock target

func (m *MockHTTPTarget) URL() string {
    return m.server.URL + "/metrics"
}

// SetMetrics configures the metrics exposed by this target

func (m *MockHTTPTarget) SetMetrics(metrics []string) {
    m.mu.Lock()
    defer m.mu.Unlock()
    m.metrics = make([]string, len(metrics))
    copy(m.metrics, metrics)
}
}

// SetDelay sets artificial response delay for timeout testing

func (m *MockHTTPTarget) SetDelay(delay time.Duration) {
    m.mu.Lock()
    defer m.mu.Unlock()
    m.delay = delay
}
}

// SetErrorRate sets probability of HTTP 500 errors (0.0 to 1.0)

func (m *MockHTTPTarget) SetErrorRate(rate float64) {
    m.mu.Lock()
    defer m.mu.Unlock()
    m.errorRate = rate
}
}

// Close shuts down the mock HTTP server
```

```
func (m *MockHTTPTarget) Close() {
    m.server.Close()
}

func (m *MockHTTPTarget) handleMetrics(w http.ResponseWriter, r *http.Request) {
    m.mu.RLock()
    delay := m.delay
    errorRate := m.errorRate
    metrics := make([]string, len(m.metrics))
    copy(metrics, m.metrics)
    m.mu.RUnlock()
    // Simulate delay if configured
    if delay > 0 {
        time.Sleep(delay)
    }
    // Simulate errors if configured
    if errorRate > 0 && rand.Float64() < errorRate {
        http.Error(w, "Simulated server error", http.StatusInternalServerError)
        return
    }
    w.Header().Set("Content-Type", "text/plain")
    w.WriteHeader(http.StatusOK)
    // Write metrics in Prometheus exposition format
    for _, metric := range metrics {
        fmt.Fprintf(w, "%s\n", metric)
    }
}
```

```
 }  
 }
```

Milestone Checkpoints

Milestone 1 - Metrics Data Model Checkpoint: After implementing the metrics data model, verify functionality with:

```
go test ./internal/metrics/...
```

BASH

Expected behavior:

- Counter values increase monotonically and reject negative increments
- Gauge values can be set to any value and read back correctly
- Histogram observations are recorded in appropriate buckets
- Label validation rejects high-cardinality combinations
- Metric metadata is stored and retrievable

Milestone 2 - Scrape Engine Checkpoint: After implementing the scrape engine, test with:

```
go run cmd/metrics-server/main.go  
  
curl http://localhost:9090/targets # Should show discovered targets
```

BASH

Expected behavior:

- Targets are discovered from configuration
- HTTP scrapes retrieve metrics successfully
- Target health reflects scrape success/failure
- Malformed metrics are rejected without affecting valid ones

Milestone 3 - Storage Engine Checkpoint: After implementing storage, verify with:

```
go test ./internal/storage/... -v  
  
# Check data directory contains WAL and chunk files  
  
ls -la ./data/
```

BASH

Expected behavior:

- Samples are compressed using Gorilla algorithm
- WAL provides durability for writes
- Indexes enable fast series lookup
- Old data is deleted according to retention policy

Milestone 4 - Query Engine Checkpoint: After implementing the query engine, test with:

```
curl "http://localhost:9090/api/v1/query?query=up"  
curl "http://localhost:9090/api/v1/query_range?  
query=rate(requests_total[5m])&start=1609459200&end=1609462800&step=60"
```

BASH

Expected behavior:

- PromQL expressions parse correctly
- Label selectors filter time series appropriately
- Aggregation functions produce correct results
- Range queries return properly interpolated data points