Alerting System: Design Document

Overview

This document outlines the design of an alerting system that transforms raw metric data into actionable notifications. The key challenge is managing the transition from continuous metric streams to discrete, intelligible human alerts while reducing noise through grouping, silencing, and smart routing.

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

1. Context and Problem Statement

> **Milestone(s):** This section provides the foundational context and problem definition for the entire Alerting System project, encompassing all four milestones (Rule Evaluation, Alert Grouping, Silencing & Inhibition, and Notification Routing).

In modern software operations, systems are instrumented to emit a continuous stream of **metrics**—numerical measurements of performance, resource utilization, and business health. While these metrics provide deep visibility, humans cannot effectively monitor thousands of real-time data points. The challenge is transforming this continuous, high-volume stream of data into discrete, actionable, and intelligible notifications for human operators. An alerting system is the critical bridge that performs this transformation, applying logic to detect problems, aggregating related signals, suppressing noise, and routing notifications to the right people through appropriate channels.

Mental Model: The Factory Control Room

Imagine a large industrial factory filled with hundreds of machines, each equipped with numerous sensors measuring temperature, pressure, vibration, and throughput. In the **factory control room**, operators cannot watch every individual gauge in real-time. Instead, they rely on an automated system with three key functions:

1. **The Watchtower Guards (Rule Evaluation):** Automated guards are stationed at key observation points, each tasked with watching a specific set of gauges. Each guard has a simple instruction: "If the boiler temperature gauge reads above 150°C for 5 minutes straight, raise the 'Boiler Overheat' alarm." These guards periodically check their assigned gauges, make comparisons, and decide when to create an alarm signal.
2. **The Mail Sorting Desk (Alert Grouping & Routing):** When alarms are raised, they are sent to a central sorting desk. Rather than delivering each alarm individually to the maintenance team's inbox, the sorter groups related alarms together. All alarms from "Boiler Room A" might be bundled into a single notification. The sorter also decides which team gets which bundle: electrical issues go to the electricians, pressure leaks go to the pipefitters, and critical issues that are unacknowledged get escalated to the shift supervisor.
3. **The "Do Not Disturb" Signs & Circuit Breakers (Silencing & Inhibition):** During scheduled maintenance in Boiler Room B, the team can place a "Do Not Disturb" sign. Alarms from that specific room are temporarily suppressed, preventing unnecessary notifications. Furthermore, if a major "Power Grid Failure" alarm is raised, it automatically trips a circuit breaker that suppresses all subsequent "Low Voltage" alarms from individual machines, as they are just symptoms of the root cause and would only create noise.

This **factory control room** is the mental model for our alerting system. The `Rule Evaluator` is the watchtower guard, the `Grouper` and `Router` form the mail sorting desk, and the `Silencer/Inhibitor` provides the DND signs and circuit breakers. The ultimate goal is to ensure the right person gets the right information at the right time—no more, no less.

The Core Problem: Signal vs. Noise

The core challenge in alerting is **Alert Fatigue**—the phenomenon where operators become desensitized or overwhelmed due to a high volume of low-value alerts, causing them to miss critical signals. Raw metric streams are pure **data**; they lack context, correlation, and prioritization. Without intelligent processing, they produce overwhelming **noise**.

The problem manifests in several dimensions:

1. **Volume & Duplication:** A single underlying issue (e.g., a database failure) can trigger hundreds of related metric anomalies (high latency, error rate spikes, connection pool exhaustion). Sending each as a separate notification floods the on-call engineer with redundant information, obscuring the root cause.
2. **Temporal Bursts & Flapping:** Metrics can be volatile. A CPU spike that lasts for 10 seconds then returns to normal is likely not actionable. Without temporal filtering ("for-duration"), the system generates transient "flapping" alerts that distract operators.
3. **Contextual Irrelevance:** Not all alerts are relevant to all people at all times. A development team doesn't need alerts for the production payment service during their scheduled deployment window. An alert about a test environment failure at 3 AM should not wake up an on-call engineer.
4. **Lack of Prioritization & Escalation:** Critical, unacknowledged alerts can get lost in the shuffle if there is no mechanism to escalate them to a secondary channel or a more senior responder after a timeout.

Our alerting system must therefore act as a **signal processing pipeline**. It must:

- **Filter** raw metrics to detect genuine anomalies (Milestone 1).
- **Aggregate** related anomalies to present a coherent picture (Milestone 2).
- **Suppress** expected or irrelevant noise (Milestone 3).
- **Route** the distilled signal to the appropriate destination with the correct urgency (Milestone 4).

The failure mode is not just technical—it's human. A poorly designed alerting system leads to ignored pages, slower incident response, and burnout among engineering staff.

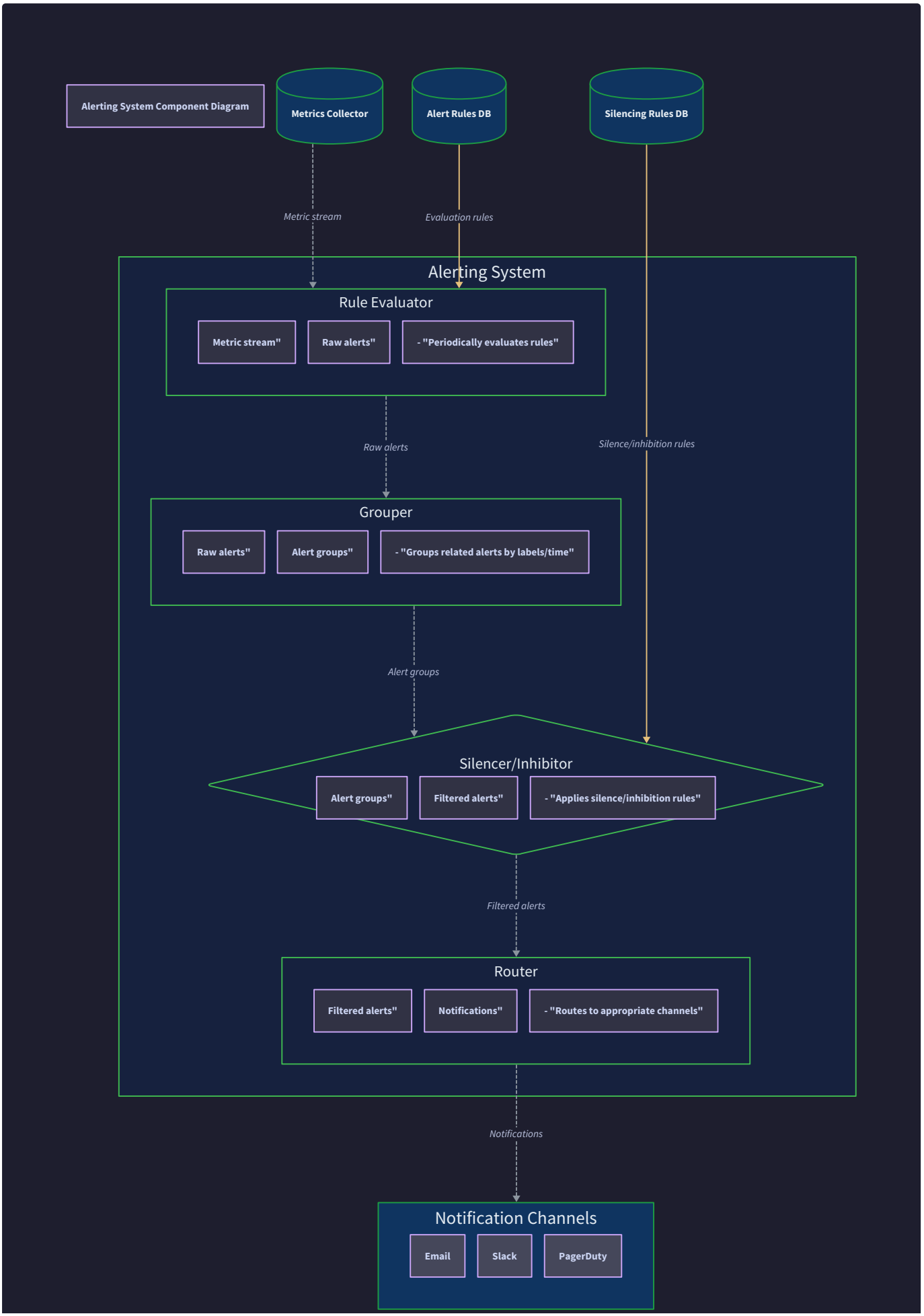
Existing Approaches & Comparison

Several paradigms and existing systems address parts of this problem space. Understanding their strengths and trade-offs informs our design decisions.

> **Decision: Adopting a Prometheus/Alertmanager-Inspired Architecture**
>
> - **Context**: We need a pragmatic, widely understood model for metric-based alerting that balances simplicity with powerful features like grouping, inhibition, and rich routing.
> - **Options Considered**:
>   1. **Simple Threshold Monitor**: A monolithic service that evaluates static thresholds and sends immediate notifications (e.g., Nagios). It's simple but lacks sophisticated noise reduction.
>   2. **Event Correlation Engine (CEP)**: A complex system that processes streams to find patterns (e.g., Apache Flink with CEP rules). Extremely powerful for correlation but has high operational and cognitive complexity.
>   3. **Prometheus & Alertmanager Model**: A two-stage pipeline: a time-series database with a flexible query language (PromQL) for rule evaluation, and a separate component (Alertmanager) for post-processing (grouping, silencing, routing). This separates detection logic from notification management.
> - **Decision**: We will architect our system following the core concepts of the **Prometheus & Alertmanager model**. Our Rule Evaluator will play the role of Prometheus's alerting rules, and our subsequent components (Grouper, Silencer/Inhibitor, Router) will form a unified Alertmanager-like processor.
> - **Rationale**: This model is a proven industry standard for cloud-native monitoring. It provides a clear separation of concerns, enabling independent scaling and development of the detection and notification layers. The concepts (labels, matchers, grouping keys) are well-documented and provide a powerful yet understandable abstraction. It hits the sweet spot between the oversimplification of Nagios and the overcomplexity of a full CEP system.
> - **Consequences**: We inherit a design vocabulary (labels, Alertmanager configuration). We must implement a PromQL-like expression evaluator or a compatible facade. The system will be inherently pull-based for metric evaluation, which simplifies state management in the evaluator but requires a reliable metrics source.

Approach	Pros	Cons	Relevance to Our Design
Simple Threshold Monitor (Nagios-style)	Simple to implement and understand. Low latency from detection to notification.	No built-in grouping or deduplication. Poor handling of flapping alerts. Limited routing logic.	We reject this as our primary model due to its lack of noise reduction features. However, we adopt its simplicity for the core rule evaluation *interface*.
Event Correlation Engine (CEP)	Can detect complex multi-metric patterns and sequences. Powerful for reducing noise by correlating root causes.	Very high complexity in rule definition and engine operation. Steep learning curve. Often overkill for common threshold-based alerting.	We acknowledge the need for correlation but address it through simpler means: **alert grouping** (by labels) and **inhibition** (suppressing symptoms when a cause fires).
Prometheus & Alertmanager	Excellent separation of concerns. Rich, declarative configuration for routing and suppression. Powerful label-based data model for flexibility. Industry-standard with vast community knowledge.	Pull-based metric evaluation can introduce latency. Rule evaluation is tied to the metrics database's query capabilities.	**This is our primary inspiration.** We adopt its label/matcher semantics, grouping logic, and routing tree concept. We will simplify where appropriate for the project scope (e.g., implementing a basic PromQL-like evaluator instead of a full query engine).
Push-Based Alert Aggregators (e.g., Grafana OnCall)	Alerts are pushed from various sources, decoupling detection from aggregation. Good for heterogeneous environments.	Requires a reliable push from all alert sources. Can lose the context and richness of the underlying metric time-series.	We are building a more integrated system focused on metrics. Our design is pull-based for evaluation but uses internal push semantics between components (e.g., Evaluator pushes alerts to Grouper).

The component diagram below illustrates the high-level architecture of our Prometheus/Alertmanager-inspired system, showing the flow of data from metrics to human notification.


Alerting System Component Diagram
Metrics Collector
Alert Rules DB
Silencing Rules DB
Metric stream
Evaluation rules
Alerting System
Rule Evaluator
Metric stream"
Raw alerts"
- "Periodically evaluates rules"
Raw alerts
Silence/inhibition rules
Grouper
Raw alerts"
Alert groups"
- "Groups related alerts by labels/time"
Alert groups
Silencer/Inhibitor
Alert groups"
Filtered alerts"
- "Applies silence/inhibition rules"
Filtered alerts
Router
Filtered alerts"
Notifications"
- "Routes to appropriate channels"
Notifications
Notification Channels
Email
Slack
PagerDuty

Implementation Guidance

While this section is primarily conceptual, setting up a clear project structure from the start is crucial. The following recommendations establish a foundation for the code you will write in subsequent milestones.

A. Technology Recommendations Table

Component	Simple Option (Recommended for Learning)	Advanced Option (For Extension)
Language & Runtime	Go (standard library + `net/http`, `time`, `sync`)	Go with additional libraries for structured logging (`slog` or `zap`), advanced scheduling (`cron.v3`)
Metrics Source Client	HTTP client to a REST API returning JSON-formatted metric values (simulating Prometheus's `query_range` API)	Native Prometheus client library (`prometheus/client_golang`) for direct querying
Data Serialization	Go's built-in `encoding/json` for external communication and config files	Protocol Buffers (`google.golang.org/protobuf`) for efficient internal messaging
Concurrent Data Structures	`sync.RWMutex` protecting `map`s for state (alerts, silences, groups)	`sync.Map` for concurrent maps or partitioned locks for higher throughput
Configuration Management	YAML files parsed with `gopkg.in/yaml.v3` for rule and route definitions	Dynamic configuration via etcd/Consul or a database with live reloading

B. Recommended File/Module Structure Organize your project from day one to separate concerns and mirror the architectural components. This structure scales cleanly through all four milestones.

```
alerting-system/
├── cmd/
│   ├── evaluator/          # Entry point for the Rule Evaluator service (Milestone 1)
│   │   └── main.go
│   └── alertmanager/       # Entry point for the Grouper/Silencer/Router service (Milestones 2-4)
│       └── main.go
├── internal/
│   ├── alert/              # Core data types: Alert, Rule, Notification
│   │   ├── alert.go
│   │   ├── rule.go
│   │   └── notification.go
│   ├── evaluator/          # Rule evaluation engine (Milestone 1)
│   │   ├── engine.go       # Main evaluation loop and scheduler
│   │   ├── promql.go       # Simplified PromQL-like expression parser/evaluator
│   │   └── state_manager.go # Manages alert state (pending/firing/resolved)
│   ├── grouper/            # Alert grouping logic (Milestone 2)
│   │   ├── grouper.go      # Group lifecycle and key management
│   │   └── timer.go        # Handles group_wait and group_interval timers
│   ├── silence/            # Silencing and inhibition (Milestone 3)
│   │   ├── silence.go      # Silence matcher and time-based activation
│   │   ├── inhibitor.go    # Inhibition rule evaluation
│   │   └── store.go        # Storage for active silences/inhibition rules
│   ├── router/             # Notification routing (Milestone 4)
│   │   ├── router.go       # Routing tree walk and matcher evaluation
│   │   ├── receiver.go     # Interfaces for different receivers (Slack, PagerDuty, etc.)
│   │   └── rate_limit.go   # Rate limiting per receiver/route
│   └── config/             # Configuration structs and parsers
│       ├── rule.go
│       ├── route.go
│       └── silence.go
├── pkg/
│   └── metricsclient/      # Reusable client for querying the metrics collector
│       └── client.go
├── configs/                # Example configuration files
│   ├── rules.yml
│   └── alertmanager.yml
├── go.mod
└── go.sum
```

C. Infrastructure Starter Code Here is a complete, ready-to-use HTTP client for querying the prerequisite Metrics Collector service. This client abstracts the communication with your metrics source and will be used by the Rule Evaluator.

GO

```go
// pkg/metricsclient/client.go

package metricsclient

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

// MetricQueryResult represents a simplified response from a Prometheus-like query API.
// For this project, we assume the query returns a single instant vector (one data point).
type MetricQueryResult struct {
    Status string `json:"status"`
    Data   struct {
        ResultType string `json:"resultType"`
        Result     []struct {
            Metric map[string]string `json:"metric"`
            Value  []interface{}     `json:"value"` // [timestamp, value]
        } `json:"result"`
    } `json:"data"`
}

// Client communicates with the metrics collector HTTP API.
type Client struct {
    baseURL    string
    httpClient *http.Client
}

// NewClient creates a new metrics client.
func NewClient(baseURL string) *Client {
    return &Client{
        baseURL: baseURL,
        httpClient: &http.Client{
            Timeout: 10 * time.Second,
        },
    }
}

// Query executes a PromQL-like query at the current time.
// It returns the numeric value of the first result, or an error.
func (c *Client) Query(ctx context.Context, query string) (float64, error) {
```

```
    req, err := http.NewRequestWithContext(ctx, "GET", c.baseURL+"/api/v1/query", nil)
    if err != nil {
        return 0, fmt.Errorf("creating request: %w", err)
    }

    q := req.URL.Query()
    q.Add("query", query)
    req.URL.RawQuery = q.Encode()

    resp, err := c.httpClient.Do(req)
    if err != nil {
        return 0, fmt.Errorf("executing request: %w", err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        return 0, fmt.Errorf("unexpected status code: %d", resp.StatusCode)
    }

    var result MetricQueryResult
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        return 0, fmt.Errorf("decoding response: %w", err)
    }

    if result.Status != "success" {
        return 0, fmt.Errorf("query status not successful: %s", result.Status)
    }

    if len(result.Data.Result) == 0 {
        return 0, fmt.Errorf("no data returned from query")
    }

    // Expect the value to be [timestamp, valueAsString]
    if len(result.Data.Result[0].Value) != 2 {
        return 0, fmt.Errorf("unexpected value format in result")
    }

    // The value is returned as a JSON string; convert to float64.
    valStr, ok := result.Data.Result[0].Value[1].(string)
    if !ok {
        return 0, fmt.Errorf("value is not a string")
    }

    var val float64
    if _, err := fmt.Sscanf(valStr, "%f", &val); err != nil {
```

```
        return 0, fmt.Errorf("converting value %q to float: %w", valStr, err)
    }

    return val, nil
}
```

D. Core Logic Skeleton Code For the Rule Evaluator's main scheduling loop (a core concept from Milestone 1), here is a skeleton with TODOs. This will be placed in `internal/evaluator/engine.go`.

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

package evaluator


import (

    "context"

    "sync"

    "time"

    "yourproject/internal/alert"

    "yourproject/pkg/metricsclient"

)


// Engine evaluates a set of alerting rules at a fixed interval.

type Engine struct {

    rules   []*alert.Rule

    client  *metricsclient.Client

    interval time.Duration

    stateMu sync.RWMutex

    // state maps rule ID to a map of alert labels -> alert state.

    state map[string]map[string]*alert.Alert

    stopCh chan struct{}

}


// NewEngine creates a new rule evaluation engine.

func NewEngine(rules []*alert.Rule, client *metricsclient.Client, interval time.Duration) *Engine {

    return &Engine{

        rules:    rules,

        client:   client,

        interval: interval,

        state:    make(map[string]map[string]*alert.Alert),

        stopCh:   make(chan struct{}),

    }

}


// Run starts the evaluation loop in a background goroutine.

func (e *Engine) Run(ctx context.Context) {

    ticker := time.NewTicker(e.interval)

    defer ticker.Stop()


    for {

        select {

        case <-ticker.C:

            e.evaluateAllRules(ctx)

        case <-e.stopCh:
```

```
            return
        case <-ctx.Done():
            return
        }
    }
}

// Stop halts the evaluation loop.
func (e *Engine) Stop() {
    close(e.stopCh)
}

// evaluateAllRules iterates through all rules and evaluates each one.
func (e *Engine) evaluateAllRules(ctx context.Context) {
    for _, rule := range e.rules {
        // TODO 1: Execute the rule's query expression using the metrics client.
        //   - Call e.client.Query(ctx, rule.Expression)
        //   - Handle query errors (log them, set alert state to inactive?)
        // TODO 2: Compare the query result value with the rule's threshold using the rule's operator (>, <, etc.).
        //   - Implement a comparator function that takes (value, threshold, operator).
        // TODO 3: Based on the comparison result and the rule's `For` duration, determine the new alert state.
        //   - Access the current state for this rule/label combination from e.state.
        //   - Apply state transition logic: inactive -> pending -> firing -> resolved.
        //   - Manage timestamps for when the condition started being true.
        // TODO 4: If the alert state has changed (e.g., to firing or resolved), update e.state and send the alert to the next processing stage (e.g., a channel).
        //   - Create an alert.Alert struct with all necessary fields (labels, annotations, startsAt, endsAt).
        //   - Emit the alert via a callback or channel for the Grouper to consume.
    }
}
```

E. Language-Specific Hints

- **Concurrency**: Use `sync.RWMutex` to protect shared state maps. For the evaluation loop, a read lock is sufficient when checking state, but a write lock is needed when updating.
- **Time**: Use `time.Now()` for timestamps. For managing `For` durations, store a `time.Time` when the condition first became true and compare it with the current time.
- **JSON**: Use struct tags (`json:"field_name"`) when unmarshaling configuration files and API responses.
- **Channels**: Use buffered channels to decouple the Rule Evaluator from the Grouper to prevent backpressure from blocking evaluation.

2. Goals and Non-Goals

> **Milestone(s):** This section establishes the scope and boundaries for the entire Alerting System project, providing the foundation for decisions made in all four milestones (Rule Evaluation, Alert Grouping, Silencing & Inhibition, and Notification Routing).

Establishing clear boundaries is essential before designing any complex system. Imagine designing an emergency broadcast network for a city—you must decide whether it handles only fire alarms, or also includes medical emergencies, weather alerts, and public announcements. Setting these parameters early prevents

scope creep and ensures the team builds what's actually needed rather than chasing endless features.

This section explicitly defines what the system **must achieve** (Goals) and what it **will not do** (Non-Goals). This distinction is critical for several reasons:

1. **Focus Development Effort:** By declaring non-goals, we intentionally limit the problem space, allowing deeper implementation of core functionality rather than superficial coverage of many features.
2. **Set User Expectations:** Documentation and marketing can accurately reflect system capabilities.
3. **Guide Architectural Decisions:** When facing design trade-offs, we can refer back to these goals to determine which option better serves our primary objectives.

2.1 Goals: Functional Requirements

The system must transform raw metric data into actionable human notifications while minimizing **alert fatigue**. The following table enumerates the specific functional capabilities, mapped to their corresponding project milestones.

Functional Goal	Description	Milestone Reference	Success Criteria
Metric-Based Alert Detection	Evaluate numeric conditions against time-series metric data to detect abnormal states.	Milestone 1	System can query a metrics API, evaluate PromQL-like expressions, and trigger alerts when thresholds are breached for specified durations.
Multi-State Alert Management	Track alerts through distinct lifecycle states (pending, firing, resolved) with proper transitions.	Milestone 1	Each alert has a clear state that changes based on rule evaluation results; resolved alerts are properly identified and handled.
Alert Aggregation	Group related alerts together to reduce notification volume through configurable grouping keys.	Milestone 2	Multiple firing alerts sharing the same `alertname` and `cluster` labels appear as a single notification bundle.
Configurable Group Timing	Control notification timing with `group_wait` (initial delay) and `group_interval` (repeat frequency).	Milestone 2	First notification for a new alert group waits 30 seconds to batch related alerts; subsequent updates are sent no more frequently than every 5 minutes.
Time-Based Silencing	Suppress alerts during maintenance windows using label matchers and time ranges.	Milestone 3	Alerts matching `service="database"` and `cluster="prod-east"` generate no notifications between 2:00 AM and 3:00 AM UTC.
Alert Inhibition	Automatically suppress lower-severity alerts when related higher-severity alerts are firing.	Milestone 3	When a `severity="critical"` alert fires for `service="api"` , all `severity="warning"` alerts for the same service are automatically suppressed.
Multi-Channel Notification Routing	Route alerts to appropriate destinations (Slack, PagerDuty, email, webhook) based on label matching.	Milestone 4	Critical alerts go to PagerDuty, warning alerts go to Slack, and all production alerts are also emailed to an ops mailing list.
Routing Hierarchy	Support hierarchical routing configuration where alerts flow through a tree of route rules.	Milestone 4	A root route sends all alerts to email, but child routes can override for specific teams or severity levels.
Rate Limiting	Prevent notification storms by enforcing maximum frequency limits per receiver or route.	Milestone 4	The Slack receiver cannot send more than one notification per minute per unique group of alerts.
Dynamic Alert Metadata	Generate alert annotations and labels using template expressions that reference metric labels and values.	Milestone 1	Alert summary includes the actual metric value: "CPU usage at 95% (threshold: 90%)".
Persistence of Active Alerts	Maintain alert state across process restarts for consistent notification behavior.	Cross-cutting	After a restart, previously firing alerts remain in firing state rather than being forgotten.
Configuration Validation	Validate all user-provided configuration files for syntax and semantic errors at startup.	Cross-cutting	Invalid regular expression in a silence matcher is caught during configuration load, not at runtime.

These goals collectively address the core problem of **signal vs. noise** in monitoring systems. By implementing threshold evaluation, intelligent grouping, strategic suppression, and targeted routing, the system elevates truly actionable signals while filtering out irrelevant noise.

2.1.1 Operational Goals (Quality Attributes)

Beyond functional requirements, the system must meet specific operational characteristics to be usable in production environments.

Operational Goal	Description	Measurement Criteria
Low Latency Evaluation	Rule evaluation must complete within a predictable time window to avoid missing metric data windows.	95th percentile of evaluation cycle duration < rule evaluation interval (e.g., < 15s for 15s intervals).
High Throughput	Process thousands of alert rules and tens of thousands of active alert instances concurrently.	Successfully evaluate 10,000 rules with 50,000 active alerts within a 15-second evaluation window.
Configuration Agility	Support dynamic configuration changes without service interruption or state loss.	Adding a new silence rule takes effect within 30 seconds without restarting the process.
Observability	Provide extensive internal metrics and logs for debugging alert processing pipeline issues.	All state transitions, grouping decisions, and notification attempts are logged with structured context.
Resource Efficiency	Use memory and CPU resources proportionally to the active workload, not total configured rules.	Memory usage scales linearly with number of firing alerts, not total configured rules or historical alerts.

> **Key Design Insight:** The operational goal of **configuration agility** directly influences our architecture. We will design components to accept configuration updates at runtime (hot reloading) rather than requiring process restarts, which is essential for operational environments where alerting cannot be disabled for maintenance.

2.2 Non-Goals: Explicit Limitations

Equally important to what the system *will* do is what it *will not* do. These non-goals prevent scope creep and clarify that certain expected features from similar systems (like Prometheus Alertmanager) are intentionally omitted from the initial implementation.

Non-Goal	Rationale	Alternative/Workaround
Historical Alert Analysis	The system focuses on real-time alert detection, not retrospective analysis of past alerts.	Alert history can be stored in external systems (databases, SIEM tools) via webhook notifications.
Graphical User Interface	Building a web UI for alert management is out of scope for the core alerting engine.	Configuration and alert inspection will be done via configuration files, CLI tools, and API endpoints.
Alert Correlation Engine	The system will not automatically discover relationships between unrelated alerts using machine learning or complex event processing.	Correlation can be implemented at the metric level (via rule expressions) or through explicit inhibition rules.
Multi-Tenancy with Isolation	The system does not provide built-in tenant isolation, access control, or quota management for shared deployments.	For multi-team use, deploy separate instances per team or implement isolation at the infrastructure layer.
Long-Term State Storage	The system is not designed as a permanent archive for alert state; it maintains only active and recently resolved alerts.	Important alert history should be forwarded to external systems designed for long-term storage.
Metric Collection	The system consumes metrics from external collectors; it does not scrape, collect, or store time-series data itself.	Integrate with the existing Metrics Collector project (prerequisite) or any Prometheus-compatible metrics API.
Alert Acknowledgment System	Users cannot manually acknowledge alerts to temporarily suppress notifications (different from silences).	Use silences for planned suppression; for manual intervention, rely on external ticketing systems integrated via webhooks.
Complex Escalation Policies	Beyond simple routing, the system does not implement multi-level escalation chains (e.g., "page after 15 minutes if no response").	Implement escalation logic in external systems (PagerDuty, OpsGenie) that receive notifications via webhooks.
Distributed High Availability	The initial design runs as a single instance; clustering and automatic failover are not provided out-of-the-box.	For reliability, run multiple independent instances behind a load balancer with shared nothing architecture.
Custom Notification Templates per Receiver	All receivers of the same type (e.g., all Slack receivers) use the same message template format.	Custom formatting can be implemented in external systems via webhooks or in the notification channel configuration.

> **Design Principle:** The distinction between **silences** and **acknowledgments** is crucial. Silences are time-based suppressions for known maintenance windows, while acknowledgments are manual human interventions for unexpected incidents. By omitting acknowledgments, we simplify the state management model and avoid the complexity of tracking human responses within the alerting engine itself.

2.2.1 Boundary Decisions (What Stays Out)

Several features commonly found in commercial alerting systems are intentionally excluded based on two primary criteria:

1. **Complexity vs. Value:** The implementation complexity vastly outweighs the value for our target users (small to medium operations teams).
2. **Externalization:** The functionality can be better provided by specialized external systems.

> **Decision: Scope Boundary - External Integration Over Built-in Complexity**
>
> - **Context:** We need to decide which advanced features to implement natively versus delegate to external systems.
> - **Options Considered:**
>   1. **Built-in Everything:** Implement full-featured acknowledgment, escalation, and correlation within the alerting engine.
>   2. **External Integration Focus:** Provide webhooks and APIs to integrate with external incident management tools.
> - **Decision:** External integration focus, with rich webhook payloads and OpenAPI specification.
> - **Rationale:** Incident management (acknowledgment, escalation, collaboration) is a separate concern from alert detection and routing. Specialized tools like PagerDuty, OpsGenie, and ServiceNow already excel at this. By providing clean integration points, we leverage existing ecosystems rather than duplicating functionality.
> - **Consequences:** Users must deploy additional tools for complete incident management, but the alerting system remains focused, maintainable, and interoperable.

Boundary Area	Our System's Responsibility	External System's Responsibility
Incident Lifecycle	Detect and route alerts	Acknowledge, escalate, resolve, post-mortem
Team Scheduling	Route to configured receivers	Manage on-call rotations, overrides, and schedules
Alert Enrichment	Basic label/annotation templating	Add runbooks, documentation links, team context
Alert Deduplication	Group identical alerts over time	Detect and correlate related incidents across systems

This clear boundary allows the alerting system to excel at its core competency—efficiently processing metric data into notification events—while leveraging the broader ecosystem for incident management workflow.

2.3 Trade-off Framework

When evaluating future feature requests or design changes, we will use this framework to determine alignment with our stated goals and non-goals.

A feature proposal should be ACCEPTED if it:

1. Directly advances one or more stated functional goals
2. Does not violate any stated non-goals
3. Maintains or improves operational goals (latency, throughput, etc.)
4. Can be implemented without disproportionate complexity relative to value

A feature proposal should be REJECTED if it:

1. Primarily addresses a stated non-goal (e.g., "add a GUI for alert management")
2. Significantly compromises operational goals without compelling justification
3. Creates tight coupling with specific external systems (beyond standard integrations)
4. Introduces new state management complexity that could be handled externally

This framework will guide the architecture decisions throughout the detailed design sections that follow.

---

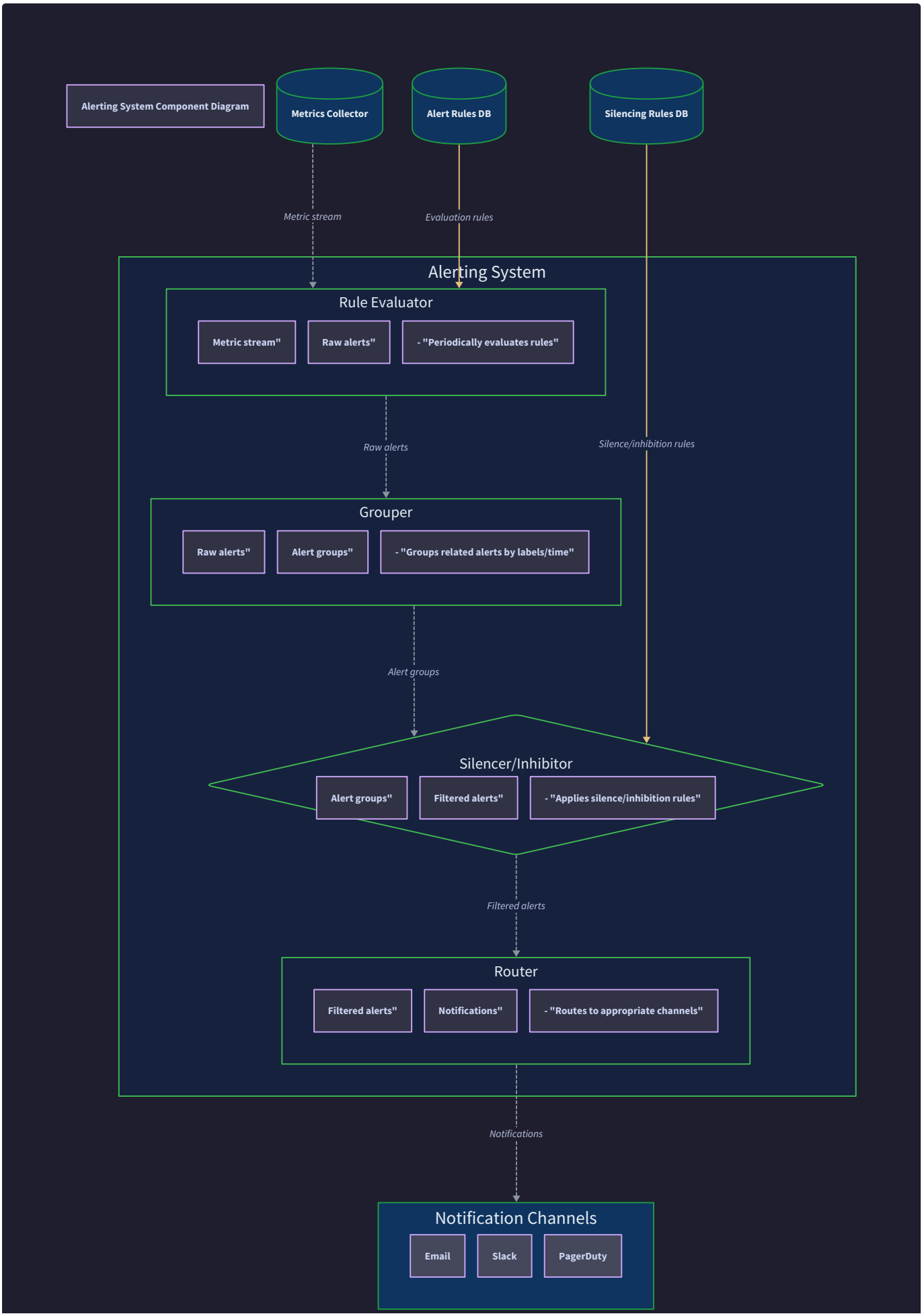
3. High-Level Architecture

> **Milestone(s):** This section provides the overarching architectural blueprint for the entire Alerting System, corresponding directly to all four milestones (Rule Evaluation, Alert Grouping, Silencing & Inhibition, and Notification Routing).

The Alerting System transforms raw, continuous metric streams into discrete, actionable notifications for human operators. Its core challenge is managing the **signal vs. noise** ratio by filtering irrelevant data, grouping related events, suppressing during known conditions, and routing intelligently to the right responders. This section presents the bird's-eye view of the system's four main logical components, their responsibilities, and how they communicate to achieve this goal.

Component Overview & Responsibilities

Think of the system as an **Industrial Factory Control Room**. Raw metrics are like thousands of sensor readings flowing from machines on the factory floor (temperature, pressure, vibration). The control room's job is not to show every reading to the operator, but to intelligently detect anomalies, correlate related issues, suppress alarms during scheduled maintenance, and escalate critical problems to the correct technician's pager or console. Each component in our architecture plays a distinct role in this control room.


Alerting System Component Diagram
Metrics Collector
Alert Rules DB
Silencing Rules DB
Metric stream
Evaluation rules
Alerting System
Rule Evaluator
Metric stream"
Raw alerts"
- "Periodically evaluates rules"
Raw alerts
Silence/inhibition rules
Grouper
Raw alerts"
Alert groups"
- "Groups related alerts by labels/time"
Alert groups
Silencer/Inhibitor
Alert groups"
Filtered alerts"
- "Applies silence/inhibition rules"
Filtered alerts
Router
Filtered alerts"
Notifications"
- "Routes to appropriate channels"
Notifications
Notification Channels
Email
Slack
PagerDuty

The system is composed of four sequential, decoupled components, each responsible for a specific transformation in the alert lifecycle. Data flows unidirectionally: from raw metric evaluation, through aggregation and suppression, to final notification delivery.

1. Rule Evaluator (The Watchtower)

Mental Model: Imagine a watchtower guard overlooking the factory floor, equipped with a list of checklists (`Rule` objects). Each checklist says: "Every 15 seconds, check gauge X. If its reading is above 100 for 5 minutes straight, raise a red flag with label `machine=A` and message 'Overheating'." The guard periodically performs these checks, and when conditions are met, creates an alert object and passes it down to the sorting desk.

Responsibilities:

- **Periodic Evaluation:** Executes all configured alert rules at a fixed interval (e.g., every 15 seconds) by querying the metrics backend.
- **Threshold & Duration Logic:** Evaluates PromQL-like expressions, compares results to thresholds using operators (>, <, etc.), and enforces the `for_duration` by managing alert state transitions (`inactive` → `pending` → `firing` → `resolved`).
- **Alert Creation:** Instantiates `Alert` objects with the appropriate `Labels` , `Annotations` , `StartsAt` , and `State` when a rule condition is met.
- **State Management:** Maintains the state of each unique alert (identified by its labels) across evaluation cycles to handle pending periods and resolution.

Key Data Held:

- Configuration: List of all active `Rule` objects.
- Runtime State: A map from alert fingerprint (hash of labels) to the current in-memory `Alert` object (state, start time).

Interface (Primary Methods):

Method	Parameters	Returns	Description
`Run`	`ctx context.Context`	(none)	Starts the main evaluation loop, periodically calling `evaluateAllRules` .
`Stop`	(none)	(none)	Stops the evaluation loop and cleans up resources.
`evaluateAllRules`	`ctx context.Context`	(none)	Internal method that iterates through all rules, queries metrics, and updates alert states.

2. Grouper (The Mail Sorting Desk)

Mental Model: After the guard raises flags, they are sent to a mail sorting desk. The sorter's job is to bundle related letters (alerts) going to the same neighborhood (notification group) to avoid sending a separate mail truck for each letter. Alerts are sorted into bundles based on their address labels (`alertname` , `cluster` , `service`). The sorter also waits a short time (`group_wait`) to see if more letters for the same bundle arrive before dispatching the truck, and controls how often re-notifications are sent if the bundle persists (`group_interval`).

Responsibilities:

- **Aggregation:** Groups incoming `Alert` objects together based on a configurable set of grouping labels (the **group key**), creating `Group` objects that contain multiple alerts.
- **Deduplication & Batching:** Ensures that multiple firing alerts belonging to the same logical issue (e.g., 100 instances of "HighCPU" on the same service) result in a single notification batch.
- **Timer Management:** Implements `group_wait` (delay initial notification for a group to collect more alerts) and `group_interval` (control frequency of repeat notifications for a continuing group).
- **Lifecycle Management:** Creates groups when the first alert for a new group key arrives, adds subsequent alerts, and cleans up empty groups when all contained alerts resolve.

Key Data Held:

- Runtime State: A map from group key (string hash) to `Group` object, which contains the list of current alerts in that group and active timers.
- Configuration: Grouping labels, `group_wait` , `group_interval` .

Interface (Primary Methods):

Method	Parameters	Returns	Description
`Process`	`alert *alert.Alert`	(none)	Ingests a new or updated alert, calculates its group key, and adds it to the appropriate group, managing group timers.
`FlushGroup`	`groupKey string`	`Notification`	Generates a consolidated notification message for all alerts in the specified group, intended for sending to the router.

3. Silencer & Inhibitor (The "Do Not Disturb" Sign and Circuit Breaker)

Mental Model: Two separate but related mechanisms. The **Silencer** is like placing a "Do Not Disturb" sign on a machine undergoing maintenance—any alarms from that machine are temporarily ignored, regardless of severity. The **Inhibitor** acts like a circuit breaker in an electrical panel: when a major fire alarm (`source` alert) is triggered, it automatically cuts power to (inhibits) the less critical "smoke detector low battery" alarms (`target` alerts) to avoid distracting noise while the primary emergency is handled.

Responsibilities (Silencer):

- **Label-Based Suppression:** Matches incoming alerts against a set of active `Silence` objects based on label matchers (e.g., `service="api", environment="staging"`). If a match is found, the alert is marked as silenced and does not proceed to notification.
- **Time-Based Activation:** Honors silence `StartsAt` and `EndsAt` times, automatically activating and expiring silences.
- **Storage & Cleanup:** Provides an interface to create, list, and delete silences, and garbage collects expired ones.

Responsibilities (Inhibitor):

- **Dependency Suppression:** Uses `InhibitionRule` configurations to suppress `target` alerts (matched by label matchers) when a `source` alert (matched by different matchers) is currently in a firing state.
- **Graph Evaluation:** Evaluates inhibition relationships—potentially a directed graph—to determine if an alert should be suppressed based on the current set of firing alerts.
- **State Awareness:** Consistently queries or is notified of the current set of firing alerts from the Rule Evaluator to apply inhibition logic.

Key Data Held (Silencer):

- Configuration/State: Set of active `Silence` objects with matchers and time windows. **Key Data Held (Inhibitor):**
- Configuration: List of `InhibitionRule` objects (`source_matchers` , `target_matchers`).
- Runtime State: Current set of firing alerts (or a reference to query them).

Interface (Primary Methods - Combined Check):

Method	Parameters	Returns	Description
`CheckSuppressed`	`alert *alert.Alert`	`bool, string`	Returns `true` and a reason (e.g., "silence-id-123", "inhibited-by-alert-X") if the alert should be suppressed by any active silence or inhibition rule.

4. Notification Router (The Airport Luggage Router)

Mental Model: Picture an airport baggage handling system. Bags (alerts) arrive on a conveyor belt. Each bag has tags (labels). The router reads the tags and sends the bag to the correct carousel (notification channel) based on rules: "Bags with tag `priority=critical` go to the PagerDuty carousel; bags with tag `team=frontend` go to the Slack #frontend-alerts channel." The system can also wait to accumulate multiple bags for the same destination before sending out a cart (batching via grouping), and has rate limiters to prevent flooding any single carousel.

Responsibilities:

- **Route Matching:** Walks a tree of `Route` configurations, matching alert labels against route `matchers` to select the appropriate `Receiver` (e.g., Slack, PagerDuty, Email).
- **Receiver Integration:** Formats alert data according to each receiver's API (JSON payloads for webhooks, specific fields for PagerDuty, etc.) and dispatches the notification via HTTP.
- **Rate Limiting & Retries:** Applies per-receiver or per-route rate limiting to prevent notification storms. Implements exponential backoff and retry logic for failed deliveries.
- **Escalation Integration (Future):** Could integrate with on-call schedules to route critical alerts to the currently on-call person.

Key Data Held:

- Configuration: Routing tree (root `Route` with nested children), receiver definitions.
- Runtime State: Rate limiters per receiver/route, retry queues for failed notifications.

Interface (Primary Methods):

Method	Parameters	Returns	Description
`Route`	`notification Notification`	`error`	Accepts a notification (which may contain multiple grouped alerts), walks the routing tree to find matching receivers, and dispatches the notification.
`Send`	`ctx context.Context` , `receiver Receiver` , `notification Notification`	`error`	Internal method that formats the notification for the specific receiver and handles the HTTP call with retries.

> **Key Design Insight:** The unidirectional, pipeline-based architecture (Evaluator → Grouper → Silencer/Inhibitor → Router) provides clear separation of concerns and makes the system easier to reason about, test, and extend. Each component can be developed, scaled, and configured independently, as long as it adheres to the data contracts (the `Alert` and `Notification` structures) between them.

Recommended File/Module Structure

Organizing the codebase with clear boundaries from the start prevents spaghetti code and `scope creep`. The following Go module structure mirrors the architectural components, placing related types, interfaces, and implementations together. We use `internal/` to prevent external consumption of our packages, enforcing encapsulation.

```
alerting-system/
├── cmd/
│   └── alertmanager/          # Main application entry point
│       └── main.go            # Parses config, wires up components, starts the engine
├── internal/
│   ├── alert/                 # Core data types (Alert, Rule, etc.) used across components
│   │   ├── types.go
│   │   ├── state.go           # State transition logic
│   │   └── fingerprint.go     # Hashing functions for alert/group keys
│   ├── evaluator/             # Milestone 1: Rule Evaluator component
│   │   ├── engine.go          # Main Engine struct with Run, Stop, evaluateAllRules
│   │   ├── query.go           # Client wrapper for querying metrics API
│   │   └── template.go        # Rendering alert annotations from templates
│   ├── grouper/               # Milestone 2: Grouper component
│   │   ├── grouper.go         # Core grouping logic (Process, FlushGroup)
│   │   ├── group.go           # Group struct and lifecycle methods
│   │   └── timer.go           # Manages group_wait and group_interval timers
│   ├── silence/               # Milestone 3: Silencer sub-component
│   │   ├── silencer.go        # CheckSuppressed logic for silences
│   │   ├── store.go           # Creates, lists, deletes silences (in-memory or persisted)
│   │   └── matcher.go         # Label matching engine (shared with inhibitor & router)
│   ├── inhibition/            # Milestone 3: Inhibitor sub-component
│   │   ├── inhibitor.go       # CheckSuppressed logic for inhibition rules
│   │   └── rule.go            # InhibitionRule struct and evaluation
│   ├── router/                # Milestone 4: Notification Router component
│   │   ├── router.go          # Tree-walking Route method
│   │   ├── route.go           # Route struct and matching logic
│   │   ├── receiver/          # Receiver implementations
│   │   │   ├── receiver.go    # Interface
│   │   │   ├── slack.go
│   │   │   ├── pagerduty.go
│   │   │   └── webhook.go
│   │   └── limiter.go         # Rate limiting logic
│   └── metricsclient/         # Shared HTTP client for querying metrics API (from Prerequisites)
│       └── client.go          # Client struct with Query method
├── pkg/
│   └── api/                   # Public REST API for creating silences, viewing alerts (optional)
│       ├── silence.go
│       └── alert.go
├── configs/
│   └── config.yaml.example    # Example configuration file
├── go.mod
└── README.md
```

Rationale for Structure:

- **`internal/alert`:** Centralizes the core data models. This prevents circular dependencies and ensures all components speak the same language. The `fingerprint` utility is here because it's a pure function on `Alert` labels.
- **Component-specific directories (`evaluator/`, `grouper/`, etc.):** Each corresponds to a milestone and a logical component. They can depend on `internal/alert` and shared utilities but not on each other horizontally, maintaining the pipeline flow.
- **Shared `metricsclient`:** Encapsulates the interaction with the external metrics collector system (the prerequisite project). This promotes reusability and makes mocking easier for tests.
- **Separate `silence` and `inhibition`:** While they both suppress alerts, their semantics, configuration, and storage needs are different enough to warrant separate packages. They can both be used by a higher-level coordinator.
- **`router/receiver/` subpackage:** Isolates the integration details of each notification channel. Adding a new channel (e.g., Microsoft Teams) means adding one file here without touching core routing logic.
- **`pkg/api` (optional):** If you choose to expose an administrative HTTP API (for creating silences, checking status), this public package can be used by external tools. It's separate because it's not core to the pipeline's operation.

This structure scales well. As the system grows (e.g., adding persistent storage for silences, a UI, or advanced templating), new packages can be added under `internal/` without disrupting the core flow.

> **Design Principle:** "A place for everything, and everything in its place." A clear module structure is the first line of defense against architectural decay. It makes the system navigable for new developers and enforces logical boundaries that prevent unintended coupling.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option (Recommended for Learning)	Advanced Option (Production-Ready)
HTTP Client (Metrics)	Standard `net/http` Client with JSON decoding.	Generated client from OpenAPI spec, with connection pooling and observability (metrics, traces).
Scheduling (Evaluator)	`time.Ticker` in a goroutine.	Dedicated scheduler library (e.g., `github.com/robfig/cron/v3`) for complex intervals.
Timer Management (Grouper)	`time.AfterFunc` for each group's timers.	Centralized timer wheel (e.g., `github.com/rcrowley/go-timers`) for efficiency with many groups.
Matcher Engine	Custom parsing of simple equality (`key=value`) and regex (`key=~regex`) matchers.	Use a well-tested library like Prometheus's `github.com/prometheus/common/model` `LabelMatcher`.
Rate Limiting	Token bucket implemented with `golang.org/x/time/rate`.	Distributed rate limiter using Redis for multi-instance deployment.
Configuration	YAML parsed with `gopkg.in/yaml.v3`.	Configuration live-reloading with `fsnotify` for **hot reloading**.

B. Infrastructure Starter Code

Here is a complete, ready-to-use HTTP client for querying the metrics collector API (prerequisite project). Place this in `internal/metricsclient/client.go`.

```go
package metricsclient

import (
    "context"
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
    "time"
)

// MetricQueryResult mirrors the expected response from the metrics collector API.
type MetricQueryResult struct {
    Status string `json:"status"`
    Data   struct {
        ResultType string `json:"resultType"`
        Result     []struct {
            Metric map[string]string `json:"metric"`
            Value  []interface{}     `json:"value"` // [timestamp, value]
        } `json:"result"`
    } `json:"data"`
}

// Client is a configured HTTP client for the metrics API.
type Client struct {
    baseURL    string
    httpClient *http.Client
}

// NewClient creates a new metrics client with a given base URL and timeout.
func NewClient(baseURL string, timeout time.Duration) *Client {
    return &Client{
        baseURL: baseURL,
        httpClient: &http.Client{
            Timeout: timeout,
        },
    }
}

// Query executes a PromQL-like query against the metrics API and returns the first scalar value.
// It expects the API endpoint at /api/v1/query?query=<query> returning a JSON response.
// For simplicity, this function assumes the query returns a single instant vector or scalar.
// Returns the numeric value and any error.
```

```go
func (c *Client) Query(ctx context.Context, query string) (float64, error) {
    // Construct the request URL
    u, err := url.Parse(c.baseURL + "/api/v1/query")
    if err != nil {
        return 0, fmt.Errorf("invalid base URL: %w", err)
    }
    q := u.Query()
    q.Set("query", query)
    u.RawQuery = q.Encode()

    // Create HTTP request
    req, err := http.NewRequestWithContext(ctx, "GET", u.String(), nil)
    if err != nil {
        return 0, fmt.Errorf("creating request: %w", err)
    }
    req.Header.Set("Accept", "application/json")

    // Execute request
    resp, err := c.httpClient.Do(req)
    if err != nil {
        return 0, fmt.Errorf("executing query: %w", err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        return 0, fmt.Errorf("unexpected status code: %d", resp.StatusCode)
    }

    // Decode JSON response
    var result MetricQueryResult
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        return 0, fmt.Errorf("decoding response: %w", err)
    }

    // Check status field
    if result.Status != "success" {
        return 0, fmt.Errorf("query failed with status: %s", result.Status)
    }

    // Extract the numeric value from the first result.
    // This is a simplified implementation. In reality, you need to handle
    // different result types (vector, scalar, matrix) and multiple results.
    if len(result.Data.Result) == 0 {
```

```
        return 0, fmt.Errorf("no data returned from query")
    }
    // Value is [timestamp, value]. We expect value to be a string-encoded float.
    if len(result.Data.Result[0].Value) != 2 {
        return 0, fmt.Errorf("unexpected value format in result")
    }
    valStr, ok := result.Data.Result[0].Value[1].(string)
    if !ok {
        return 0, fmt.Errorf("value is not a string")
    }

    var value float64
    if _, err := fmt.Sscanf(valStr, "%f", &value); err != nil {
        return 0, fmt.Errorf("parsing float from %q: %w", valStr, err)
    }
    return value, nil
}
```

C. Core Logic Skeleton Code (Rule Evaluator Engine)

This skeleton, to be placed in `internal/evaluator/engine.go`, outlines the core loop. The learner will fill in the detailed logic per the algorithm steps described in the Milestone 1 section.

```go
package evaluator

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

    "yourproject/internal/alert"
    "yourproject/internal/metricsclient"
)

// Engine is the main rule evaluation engine.
type Engine struct {
    rules   []*alert.Rule
    client  *metricsclient.Client
    interval time.Duration

    stateMu sync.RWMutex
    // Map from rule name + label fingerprint to the active alert state.
    state map[string]map[string]*alert.Alert

    stopCh chan struct{}
}

// NewEngine creates a new evaluation engine with the given rules, metrics client, and evaluation interval.
func NewEngine(rules []*alert.Rule, client *metricsclient.Client, interval time.Duration) *Engine {
    return &Engine{
        rules:   rules,
        client:  client,
        interval: interval,
        state:   make(map[string]map[string]*alert.Alert),
        stopCh:  make(chan struct{}),
    }
}

// Run starts the periodic rule evaluation loop. It blocks until Stop is called.
func (e *Engine) Run(ctx context.Context) {
    ticker := time.NewTicker(e.interval)
    defer ticker.Stop()

    log.Printf("Evaluation engine started with interval %v", e.interval)

    for {
```

```
        select {
        case <-ticker.C:
            e.evaluateAllRules(ctx)
        case <-e.stopCh:
            log.Println("Evaluation engine stopping")
            return
        case <-ctx.Done():
            log.Println("Evaluation engine cancelled via context")
            return
        }
    }
}

// Stop signals the evaluation loop to exit.
func (e *Engine) Stop() {
    close(e.stopCh)
}

// evaluateAllRules iterates through all rules, queries metrics, and updates alert states.
func (e *Engine) evaluateAllRules(ctx context.Context) {
    // TODO 1: Iterate over e.rules (the list of configured alert rules).
    // TODO 2: For each rule, call e.client.Query with the rule's Expression.
    //   - Handle query errors (log them, set alert state to inactive?).
    // TODO 3: Compare the returned float64 value against the rule's Threshold using the rule's Operator.
    //   - Operators: >, <, >=, <=, ==, !=.
    // TODO 4: Determine if the rule condition is "firing" (true) or "not firing" (false).
    // TODO 5: For each unique set of labels this rule produces (from the query result or static labels),
    //   create or update an alert in e.state.
    //   - Use a fingerprint (hash) of the labels as the key in the per-rule map.
    // TODO 6: Implement alert state transition logic:
    //   - If condition is true and alert was inactive -> move to pending, record StartsAt.
    //   - If condition is true and alert is pending for >= rule.ForDuration -> move to firing.
    //   - If condition is false and alert was firing -> move to resolved, set EndsAt.
    //   - If condition is false and alert was pending -> move to inactive (clear from state).
    // TODO 7: For any alert that is now in firing or resolved state, render its Annotations using
    //   template rendering (replace {{ $labels.xxx }} with actual label values).
    // TODO 8: Pass the updated alert (if firing or resolved) to the next component (e.g., via a channel).
    //   - This will be connected to the Grouper in the main application wiring.
    // Hint: Use e.stateMu to protect concurrent access to e.state.
    // Hint: Keep track of the previous state to avoid sending duplicate updates.
}
```

D. Language-Specific Hints (Go)

- **Concurrency:** Use `sync.RWMutex` for protecting the `state` map in the `Engine`. Readers (like the Inhibitor checking firing alerts) can use `RLock`, while the evaluator loop uses `Lock`.
- **Timers:** For the Grouper's `group_wait` and `group_interval`, use `time.AfterFunc`. Remember to `Stop()` the previous timer when resetting it.
- **Hashing:** For creating fingerprint hashes of label sets (for alert deduplication and group keys), use a stable method: sort the label keys, then use a hash like `fnv64a`. You can implement this in `internal/alert/fingerprint.go`.
- **Configuration:** Use struct tags for YAML parsing: `` `yaml:"for_duration"` ``. Use `time.ParseDuration` to convert strings like "5m" to `time.Duration`.
- **HTTP Retries:** For the Router, use a helper with `backoff.Retry` from `github.com/cenkalti/backoff/v4` for robust retry logic with exponential backoff.

E. Milestone Checkpoint (After High-Level Setup) Before diving into component details, ensure your project skeleton builds and the basic wiring compiles.

1. **Command:** `go build ./cmd/alertmanager`
2. **Expected Output:** No errors. A binary `alertmanager` should be created.
3. **Verify:** Create a minimal `main.go` that instantiates the `Engine` with a dummy `metricsclient.Client` and a single test `Rule`. Run it to see the logging from the `Run` method (you might not see evaluation yet if `evaluateAllRules` is empty).
4. **Signs of Trouble:**
   - `undefined: alert.Rule` → Check that `internal/alert/types.go` defines the `Rule` struct with the correct fields.
   - `cannot use client (type *metricsclient.Client) as type *http.Client` → Ensure your `metricsclient.NewClient` returns the correct type and is imported properly.

Proceed to the next sections (Component Design) to flesh out the detailed logic for each component, following the skeletons and guided by the mental models and ADRs.

4. Data Model

> **Milestone(s):** All four milestones (Rule Evaluation, Alert Grouping, Silencing & Inhibition, Notification Routing) depend on the data structures defined in this section. These types form the foundational language that components use to communicate about alerts, rules, silences, and routing.

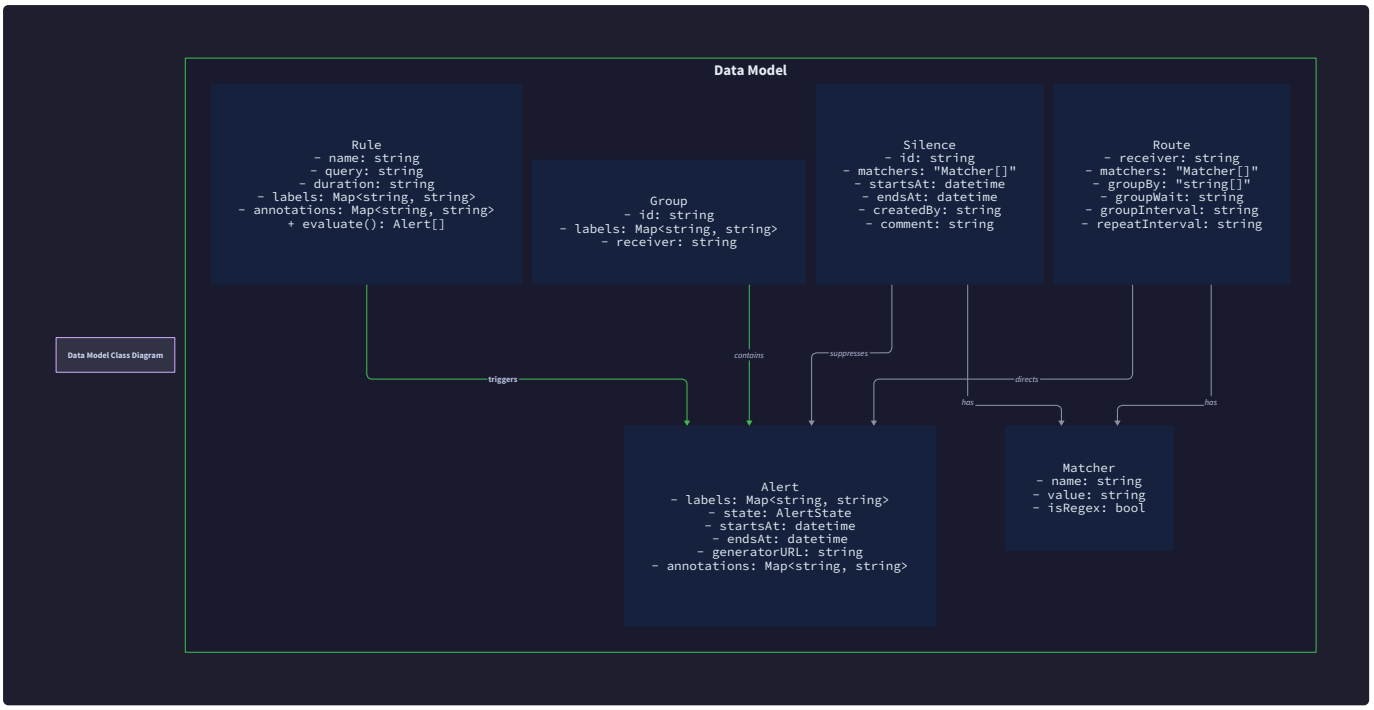
Mental Model: The Factory's Paperwork System

Imagine a factory's control room uses standardized forms to track every event. Each type of form has specific fields that capture different aspects of the factory's operation. The **Alert** is an incident report filed when a sensor reading crosses a threshold. The **Rule** is the checklist that defines when to file such a report. The **Group** is a folder that bundles related incident reports for batch processing. The **Silence** is a red "QUIET HOURS" stamp that temporarily suppresses reports from certain machines. The **Route** is a routing slip attached to the folder, directing it to the appropriate department (maintenance, management, etc.).

This section defines the exact structure of these "forms"—the data types that flow through our alerting system. Just as a factory's efficiency depends on clear, consistent paperwork, our system's reliability depends on well-defined data structures that all components understand.

Core Types and Structures

The following tables define the primary data structures used throughout the alerting system. Each field is documented with its type and purpose. These structures are serialized as JSON for configuration files, API communication, and internal state persistence. The relationships between these types are visualized in the class diagram below.



Alert

An `Alert` represents a single instance of a rule firing. It is the core unit of information that flows through the system after rule evaluation, carrying metadata about the condition that triggered it, its current lifecycle state, and where it originated. Every alert is uniquely identified by its set of labels (a fingerprint or hash is typically computed from them for deduplication).

Field Name	Type	Description
`Labels`	`map[string]string`	**Identifying dimensions.** A set of key-value pairs that uniquely identify the alert and provide context for grouping and routing. Common labels include `alertname` (the rule name), `severity` (critical, warning), `instance` (the source machine), `job` (the service), and `cluster`. Labels are immutable for the lifetime of an alert instance; changing a label creates a new alert.
`Annotations`	`map[string]string`	**Human-readable context.** Supplementary information that does not identify the alert but provides details for notifications. These are often templated from the rule (e.g., `description: "CPU on {{ $labels.instance }} is at {{ $value }}%"`, `summary: "High CPU usage"`). Unlike labels, annotations can change while the alert is active without creating a new alert.
`StartsAt`	`time.Time`	**Alert inception.** The timestamp when the alert condition was first met. For alerts with a `for_duration`, this is set when the condition enters the "pending" state. For immediate alerts, this equals the firing time.
`EndsAt`	`time.Time`	**Alert expiration.** The timestamp when the alert is no longer firing. Initially set to `StartsAt` plus a default timeout (e.g., 5 minutes). Updated each evaluation cycle while the condition remains true. When the condition clears, `EndsAt` is set to the current time (marking resolution). The system uses this to garbage-collect old, resolved alerts.
`State`	`string`	**Lifecycle phase.** One of: `"pending"` (condition met but `for_duration` not yet elapsed), `"firing"` (condition met for required duration), or `"resolved"` (condition no longer true). This drives the notification logic (pending alerts typically don't notify).
`GeneratorURL`	`string`	**Link to source.** A URL pointing back to the dashboard or query interface that generated this alert, allowing operators to quickly investigate the underlying metrics. Often constructed from the rule's configuration.

Rule

A `Rule` defines the condition that, when met, creates one or more `Alert` instances. It encapsulates the logic of *what* to monitor, *when* to trigger, and *how* to label the resulting alerts. Rules are configured by users and loaded by the Rule Evaluator.

Field Name	Type	Description
`Name`	`string`	**Unique identifier.** A human-readable name for the rule (e.g., `"HighCPUUsage"`). This becomes the value of the `alertname` label on all alerts generated by this rule. Must be unique within a rules file.
`Expression`	`string`	**Condition logic.** A PromQL-like expression that evaluates to a numeric value or a time series set. The evaluator queries this against the metrics backend. Examples: `"up{job=\"api\"} == 0"`, `"rate(http_requests_total[5m]) > 100"`.
`ForDuration`	`time.Duration`	**Stability requirement.** The duration an alert condition must remain true before transitioning from `pending` to `firing`. A value of `0s` means immediate firing. This prevents flapping alerts from causing notifications.
`Labels`	`map[string]string`	**Static label assignments.** A set of labels to attach to every alert generated by this rule. These are merged with any labels extracted from the query result (e.g., labels from the time series). Common uses: setting `severity`, `team`, or `service`.
`Annotations`	`map[string]string`	**Template definitions.** A set of annotation templates that will be rendered for each alert. Templates can reference labels and the query value (e.g., `{{ $value }}`). These provide the human-readable content in notifications.
`Operator`	`string`	**Comparison operator.** The operator used to compare the query result against the `Threshold`. Must be one of: `">"`, `"<"`, `">="`, `"<="`, `"=="`, `"!="`. While this could be embedded in the `Expression`, separating it simplifies parsing for threshold-based rules.
`Threshold`	`float64`	**Comparison value.** The numeric value against which the query result is compared using the `Operator`. For binary expressions (e.g., `up == 0`), this field may be ignored as the comparison is part of the expression itself.

MetricQueryResult

This structure represents the raw response from a query to the metrics collector backend (e.g., the project from prerequisite). It is used internally by the `Client` to parse the HTTP response before converting it into a simple numeric value for threshold comparison.

Field Name	Type	Description
`Status`	`string`	**Query execution status.** Either `"success"` or `"error"`. An error status indicates the query was malformed or the backend failed.
`Data`	`struct`	**The result payload.** Contains the actual result of the query.
`Data.ResultType`	`string`	**Type of result.** For alert rules, this is typically `"vector"` (instant query) or `"scalar"`.
`Data.Result`	`[]struct`	**List of result items.** Each item represents a time series that matched the query.
`Data.Result[].Metric`	`map[string]string`	**Labels of the time series.** The dimensional labels identifying this specific series (e.g., `{job="api", instance="10.0.0.1:8080"}`).
`Data.Result[].Value`	`[]interface{}`	**The sample value.** A two-element array `[timestamp, value]` where `value` is a string representation of a float. The timestamp is in seconds since epoch.

Client

The `Client` is a thin wrapper around an HTTP client, responsible for communicating with the external metrics collector API. It encapsulates the details of constructing the request, handling errors, and parsing the JSON response into a `MetricQueryResult`.

Field Name	Type	Description
`baseURL`	`string`	**Base address of metrics API.** The protocol, host, and port of the metrics collector (e.g., `"http://localhost:9090"`).
`httpClient`	`*http.Client`	**Configured HTTP client.** Used to make requests. Can be configured with timeouts, TLS settings, etc.

Engine

The `Engine` is the core of the Rule Evaluator component. It holds all active rules, manages the periodic evaluation loop, and maintains the current state of all alerts (which alerts are pending, firing, or resolved). It uses the `Client` to fetch metric data.

Field Name	Type	Description
`rules`	`[]*alert.Rule`	**Loaded rule definitions.** The list of all rules to evaluate on each cycle.
`client`	`*metricsclient.Client`	**Metrics API client.** Used to execute the query in each rule's `Expression`.
`interval`	`time.Duration`	**Evaluation frequency.** How often to run the evaluation loop (e.g., `15s`).
`stateMu`	`sync.RWMutex`	**Concurrency control.** Protects concurrent access to the `state` map from the evaluation loop and any read APIs (e.g., a status UI).
`state`	`map[string]map[string]*alert.Alert`	**Active alert state.** A two-level map: the first key is the rule name, the second key is a fingerprint (hash) of the alert's identifying labels. This allows quick lookup and updates of alerts per rule.
`stopCh`	`chan struct{}`	**Loop control.** A channel used to signal the evaluation loop to stop gracefully.

Group

A `Group` is a collection of related `Alert` instances that are batched together for notification. It is created dynamically by the Grouper component based on configurable grouping keys (like `alertname` and `cluster`). The group manages timers for the `group_wait` and `group_interval` batching logic.

Field Name	Type	Description
`key`	`string`	**Deterministic group identifier.** A string derived from the sorted values of the grouping labels (e.g., `"alertname=HighCPU&cluster=prod"`). All alerts with the same grouping label values belong to the same group.
`alerts`	`[]*alert.Alert`	**Alerts in this group.** The current set of firing and/or pending alerts that share the group key. The list is updated as alerts are added, resolved, or expire.
`timers`	`map[string]*time.Timer`	**Active timers for the group.** Typically holds two timers: one for `group_wait` (initial delay before first notification) and one for `group_interval` (delay between repeat notifications). Keys might be `"wait"` and `"interval"`.

Silence

A `Silence` defines a time window during which alerts matching certain criteria are suppressed and will not trigger notifications. It is like a temporary mute rule applied to the alert stream. Silences are created by operators for planned maintenance or to suppress known false positives.

Field Name	Type	Description
`ID`	`string`	**Unique silence identifier.** A UUID or other unique string generated when the silence is created. Used to reference, update, or delete the silence.
`Matchers`	`[]Matcher`	**Matching criteria.** A list of matchers that define which alerts are suppressed. An alert is silenced if it matches **all** matchers (AND logic). Example: `[{Name: "alertname", Value: "HighCPU", IsRegex: false}, {Name: "cluster", Value: "staging", IsRegex: false}]`.
`StartsAt`	`time.Time`	**Activation time.** When the silence becomes active. Can be in the future for scheduled silences.
`EndsAt`	`time.Time`	**Expiration time.** When the silence ends. After this time, alerts will no longer be suppressed by this silence.
`CreatedBy`	`string`	**Creator identity.** Username or system identifier of who created the silence, for auditing.
`Comment`	`string`	**Reason for silence.** A human-readable explanation (e.g., "Database maintenance window").

InhibitionRule

An `InhibitionRule` suppresses certain alerts (targets) when other alerts (sources) are firing. It is used to prevent redundant notifications: for example, if a cluster-level alert fires, you might want to suppress all instance-level alerts within that cluster. Inhibition is a form of dynamic, alert-driven suppression.

Field Name	Type	Description
`SourceMatchers`	`[]Matcher`	**Source alert criteria.** Matchers that identify which firing alerts trigger the inhibition (the "cause").
`TargetMatchers`	`[]Matcher`	**Target alert criteria.** Matchers that identify which alerts should be suppressed (the "effect").
`EqualLabels`	`[]string`	**Label equality constraint.** An optional list of label names. For an alert to be inhibited, there must be a source alert that matches `SourceMatchers` **and** that shares identical values for all labels listed in `EqualLabels` with the target alert. This ensures inhibition is scoped (e.g., only inhibit alerts in the same `cluster`).

Route

A `Route` is a node in the routing tree that determines where alerts are sent. Each route can have child routes, forming a hierarchy. The router walks this tree, checking matchers at each node to decide whether to send the notification to the route's receiver and/or continue to child routes.

Field Name	Type	Description
`Receiver`	`string`	**Destination receiver name.** The name of the `Receiver` configuration to which notifications matching this route should be sent. If empty, this route does not send notifications itself (may only have children).
`Matchers`	`[]Matcher`	**Route matching criteria.** A list of matchers that an alert must satisfy to enter this route. If empty, all alerts match.
`Continue`	`bool`	**Continue to sibling routes?** If `true`, after processing this route (sending to its receiver and checking children), the routing walk will continue to evaluate subsequent sibling routes. If `false`, processing stops for this branch of the tree. This controls whether alerts can match multiple routes.
`Routes`	`[]*Route`	**Child routes.** A list of sub-routes that inherit the matching context of their parent. Used to create a hierarchical routing policy (e.g., first match by `team`, then by `severity`).
`GroupWait`	`time.Duration`	**Initial batching delay.** How long to wait before sending the first notification for a new group of alerts in this route. This allows time for more alerts to arrive and be batched together.
`GroupInterval`	`time.Duration`	**Repeat notification interval.** How long to wait before sending a new notification for an existing group that has already been notified, but has new alerts or updated alert states.

Receiver

A `Receiver` defines a concrete notification channel endpoint, such as a Slack webhook, PagerDuty integration, or email SMTP server. It contains the configuration necessary to format and deliver a notification.

Field Name	Type	Description
`Name`	`string`	**Unique receiver identifier.** Used in `Route` configurations to reference this receiver (e.g., `"slack-critical"`).
`Type`	`string`	**Channel type.** Determines the integration logic to use. Common values: `"slack"`, `"pagerduty"`, `"email"`, `"webhook"`.
`Config`	`json.RawMessage`	**Type-specific configuration.** A flexible JSON blob that is parsed according to the `Type`. For Slack, this might include `webhook_url` and `channel`. For email, it would include `smtp_host`, `to`, `from`, etc. Using `json.RawMessage` allows deferred, type-specific parsing.

Notification

A `Notification` is the packaged unit sent to a receiver. It contains a batch of alerts (a group) that share the same grouping key and are destined for the same receiver. This structure is what the Grouper passes to the Router, and what the Router eventually sends to a receiver's integration code.

Field Name	Type	Description
`GroupKey`	`string`	**Identifier for the alert group.** The same as `Group.key`, used for deduplication and tracking notification state in the receiver.
`Alerts`	`[]*alert.Alert`	**Alerts in this batch.** The list of alerts being notified. Typically includes both firing and recently resolved alerts (for sending resolution notices).
`Receiver`	`string`	**Intended receiver name.** The name of the `Receiver` that should process this notification.

Matcher (Implied Type)

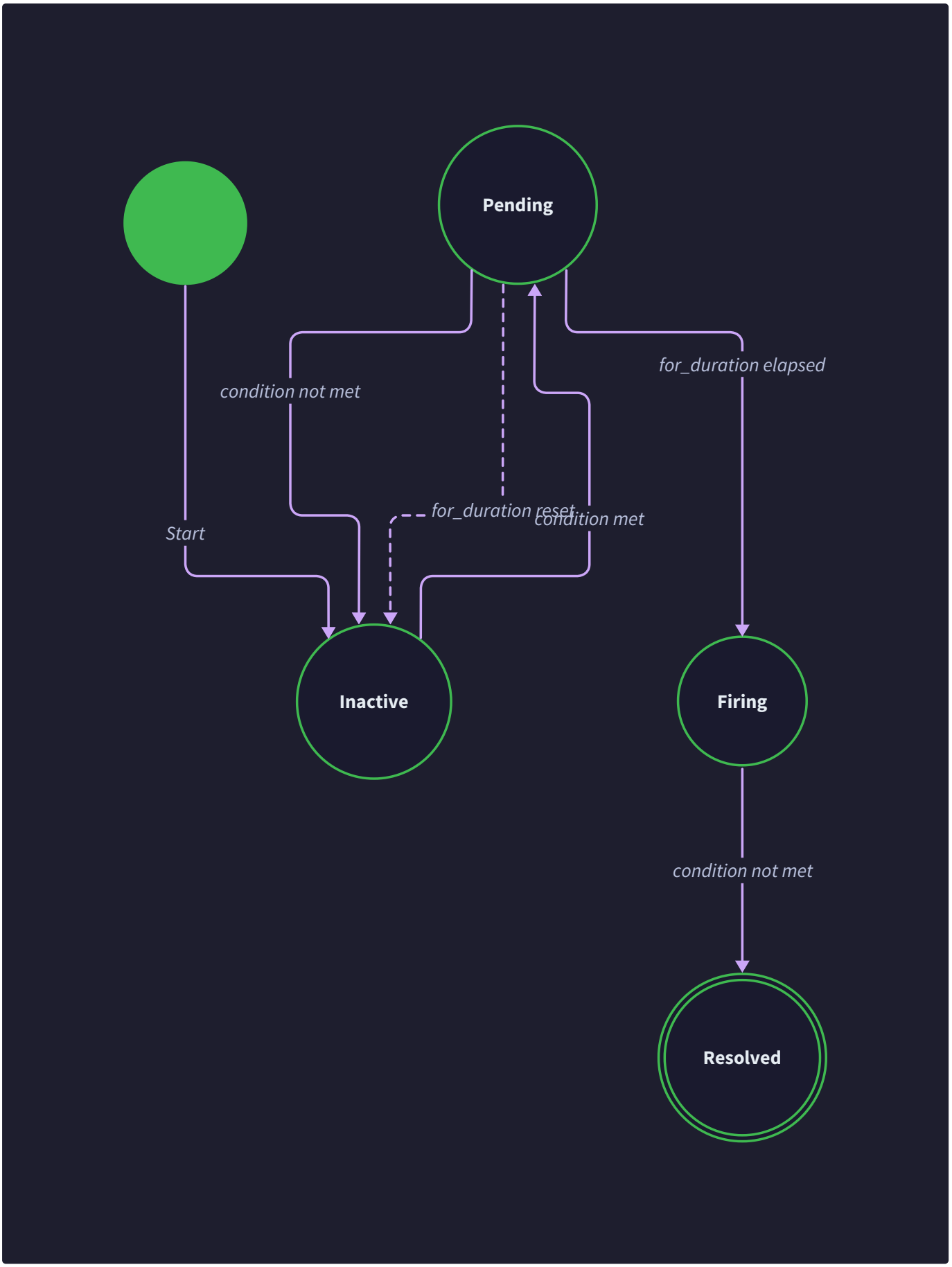
While not listed in the initial naming conventions, the `Matcher` type is central to silencing, inhibition, and routing. It is typically defined as:

```go
type Matcher struct {
    Name    string // Label name to match (e.g., "alertname")
    Value   string // Value to match against
    IsRegex bool   // If true, Value is a regex pattern; if false, exact equality match
}
```

A matcher evaluates whether a given label set contains a label with key `Name` whose value either exactly equals `Value` (if `IsRegex` is false) or matches the regular expression `Value` (if `IsRegex` is true). If the label key does not exist, the matcher fails (does not match).

Alert State Transitions

An alert's `State` field is not static; it evolves based on the continuous evaluation of its rule condition and the passage of time. Understanding this state machine is critical for implementing correct behavior in the Rule Evaluator and for knowing when notifications should be sent. The state transitions are depicted visually in the diagram below.



The following table details every possible transition. The **Current State** is the alert's state at the beginning of an evaluation cycle. The **Event** is what the Rule Evaluator observes about the rule's condition for this specific alert instance (identified by its unique labels). The **Next State** is the state after processing the event, and **Actions Taken** describes any side effects, such as updating timestamps or triggering notifications.

Current State	Event	Next State	Actions Taken
inactive (or non-existent)	Rule condition is **MET** for this alert's labels.	`pending`	1. Create a new `Alert` struct with `Labels` from rule and query result. 2. Set `State` to `"pending"`. 3. Set `StartsAt` to current time. 4. Set `EndsAt` to `StartsAt` plus a default timeout (e.g., 5 minutes). 5. Store the alert in the engine's state map.
`pending`	Rule condition is **MET** and `ForDuration` has **elapsed** (i.e., `current_time - StartsAt >= ForDuration`).	`firing`	1. Update `State` to `"firing"`. 2. Update `EndsAt` to current time plus default timeout (resetting the TTL). 3. **Mark for notification:** The alert is now eligible to be passed to the Grouper for notification routing.
`pending`	Rule condition is **NOT MET** (even briefly).	`inactive`	1. Remove the alert from the engine's state map entirely (or mark as resolved and garbage collect). 2. **Important:** The `for_duration` timer is reset. If the condition becomes true again later, a new `pending` period starts from scratch.
`firing`	Rule condition is **MET** (ongoing).	`firing`	1. Update `EndsAt` to current time plus default timeout (keep-alive). This signals that the alert is still active. 2. The alert remains in the Grouper; if `group_interval` elapses, a repeat notification may be sent.
`firing`	Rule condition is **NOT MET**.	`resolved`	1. Update `State` to `"resolved"`. 2. Set `EndsAt` to current time. 3. **Mark for resolution notification:** The alert (now in resolved state) is sent to the Grouper. Receivers may send a "resolved" message. 4. Schedule garbage collection: After a grace period (e.g., 1 hour), remove the resolved alert from state.
`resolved`	Garbage collection grace period **elapses**.	`inactive` (removed)	1. Remove the alert from the engine's state map to free memory.

> **Key Insight:** The transition from `pending` to `firing` only occurs after the condition holds *continuously* for the entire `ForDuration`. A single evaluation cycle where the condition is false resets the timer. This "for_duration" logic is the primary defense against **alert flapping** caused by noisy metrics that bounce around a threshold.

Common Pitfalls: State Management

⚡ Pitfall: Forgetting to Reset the For-Duration Timer

- **Description:** When an alert is in `pending` and the condition becomes false, the implementor might incorrectly leave the alert in `pending` state, hoping it will resume. This violates the principle that `ForDuration` requires *continuous* truth.
- **Why It's Wrong:** It would allow an alert to fire after sporadic, non-continuous true conditions, defeating the purpose of the stability check and causing false alerts.
- **Fix:** Always delete the alert from state (or move to `inactive`) the moment the condition evaluates to false for a `pending` alert.

⚡ Pitfall: Not Extending `EndsAt` for Firing Alerts

- **Description:** Failing to update `EndsAt` on each evaluation cycle while an alert is `firing`.
- **Why It's Wrong:** The alert will appear to have expired (`EndsAt` is in the past) even though it's still firing. This can cause premature garbage collection or confusion in UIs.
- **Fix:** On every cycle where the condition remains true for a `firing` alert, set `EndsAt = currentTime + defaultTimeout`.

⚡ Pitfall: Immediate Deletion of Resolved Alerts

- **Description:** Removing a `resolved` alert from state as soon as its condition clears.
- **Why It's Wrong:** The Grouper and Router need to know an alert is resolved in order to send a resolution notification. Immediate deletion makes this impossible.
- **Fix:** Keep resolved alerts in state with `State: "resolved"` and a recent `EndsAt`. Implement a separate, slower garbage collection loop that cleans up old resolved alerts after a configured retention period.

Implementation Guidance

This section bridges the design to actual Go code, providing starter structures and skeletons for the core data types.

Technology Recommendations Table

Component	Simple Option	Advanced Option
Data Serialization	Standard `encoding/json` package for marshaling/unmarshaling structs.	`json-iterator/go` for faster JSON processing, or `protobuf` for internal gRPC communication.
Time Handling	`time.Time` and `time.Duration` from the standard library. Use `time.Now().UTC()` for consistency.	Consider `github.com/benbjohnson/clock` for testable time-dependent code.
Concurrent Maps	`sync.RWMutex` protecting a standard `map`.	`sync.Map` for highly concurrent read-heavy workloads with stable keys.
Hashing/Group Key	Sorted label concatenation (e.g., `fmt.Sprintf`) or `sha256` hash of canonical JSON.	`xxhash` or `fnv` for faster, non-cryptographic hashing of group keys.

Recommended File/Module Structure

Place the core data type definitions in a central package, typically named `alert` or `models`. This keeps them isolated and importable by all components.

```
project-root/
  cmd/alertmanager/main.go          # Main entry point
  internal/alert/                    # Core data types package
    types.go                         # Alert, Rule, Group, etc. definitions
    matcher.go                       # Matcher type and evaluation logic
  internal/metricsclient/            # Metrics API client
    client.go                        # Client struct and Query method
  internal/engine/                   # Rule Evaluator component
    engine.go                        # Engine struct and loop
  internal/grouper/                  # Grouping component
    grouper.go                       # Grouper logic
  internal/silencer/                 # Silencing & Inhibition component
    silencer.go
    inhibitor.go
  internal/router/                   # Notification Router component
    router.go
    receivers/                       # Receiver implementations
      slack.go
      email.go
      webhook.go
```

Infrastructure Starter Code

Here is a complete, ready-to-use implementation for the `Matcher` type and its evaluation logic, which is a prerequisite for multiple components. Place this in `internal/alert/matcher.go`.

```go
package alert

import (
    "regexp"
    "strings"
)

// Matcher defines a rule to match a label value.
type Matcher struct {
    Name    string `json:"name"`              // Label name to match
    Value   string `json:"value"`             // Value to match against (exact or regex pattern)
    IsRegex bool   `json:"isRegex,omitempty"` // If true, Value is a regular expression
}

// Match returns true if the given label set satisfies the matcher.
func (m *Matcher) Match(labels map[string]string) bool {
    v, ok := labels[m.Name]
    if !ok {
        // Label name not present -> no match.
        return false
    }

    if !m.IsRegex {
        // Exact equality match.
        return v == m.Value
    }

    // Regex match. Compile regex once and cache for performance.
    // (In production, you'd use a sync.Map to cache compiled regexes.)
    re, err := regexp.Compile(m.Value)
    if err != nil {
        // If the regex is invalid, treat it as no match (or log error).
        return false
    }
    return re.MatchString(v)
}

// String returns a human-readable representation of the matcher.
func (m *Matcher) String() string {
    if m.IsRegex {
        return m.Name + "=~" + m.Value
    }
    return m.Name + "=" + m.Value
```

```
}


// ParseMatcher parses a matcher from a string like "alertname=HighCPU" or "cluster=~prod-.*".
func ParseMatcher(s string) (*Matcher, error) {
    var m Matcher
    parts := strings.SplitN(s, "=", 2)
    if len(parts) != 2 {
        return nil, fmt.Errorf("bad matcher format: %s", s)
    }
    m.Name = parts[0]

    // Check if it's a regex matcher (starts with '=~').
    if strings.HasPrefix(parts[1], "~") {
        m.IsRegex = true
        m.Value = strings.TrimPrefix(parts[1], "~")
    } else {
        m.Value = parts[1]
    }
    return &m, nil
}
```

Core Logic Skeleton Code

Below is the skeleton for the core `Alert` and `Rule` types, with TODOs guiding the implementation of key methods. Place this in `internal/alert/types.go`.

```go
package alert

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

// Alert represents a single firing/resolved alert.
type Alert struct {
    Labels       map[string]string `json:"labels"`
    Annotations  map[string]string `json:"annotations,omitempty"`
    StartsAt     time.Time         `json:"startsAt"`
    EndsAt       time.Time         `json:"endsAt"`
    State        string            `json:"state"` // "pending", "firing", "resolved"
    GeneratorURL string            `json:"generatorURL,omitempty"`
}

// Fingerprint returns a unique string identifier for the alert based on its labels.
// This is used as a key in maps and for deduplication.
func (a *Alert) Fingerprint() string {
    // TODO 1: Extract label keys into a slice and sort them alphabetically.
    // TODO 2: For each sorted key, append "key=value" to a builder.
    // TODO 3: Return the concatenated string (or a hash of it).
    // Hint: Use strings.Builder for efficient concatenation.
    // Example: For labels {b:"2", a:"1"} -> "a=1,b=2"
    return ""
}

// IsActive returns true if the alert is currently pending or firing.
func (a *Alert) IsActive() bool {
    // TODO: Return true if State is "pending" or "firing".
    return false
}

// Rule defines an alerting rule.
type Rule struct {
    Name         string            `json:"name"`
    Expression   string            `json:"expression"`
    ForDuration  time.Duration     `json:"for,omitempty"`
    Labels       map[string]string `json:"labels,omitempty"`
```

```
    Annotations  map[string]string `json:"annotations,omitempty"`
    Operator     string            `json:"operator,omitempty"`   // ">", "<", etc.
    Threshold    float64           `json:"threshold,omitempty"`
}

// Validate checks if the rule has required fields and valid operator.
func (r *Rule) Validate() error {
    // TODO 1: Check that Name is non-empty.
    // TODO 2: Check that Expression is non-empty.
    // TODO 3: If Operator is set, verify it's one of the allowed operators.
    // TODO 4: Ensure ForDuration is non-negative.
    return nil
}

// RenderAnnotations fills in template placeholders in the rule's annotations
// using the alert's labels and the query value.
func (r *Rule) RenderAnnotations(alertLabels map[string]string, value float64) map[string]string {
    rendered := make(map[string]string)
    for k, tpl := range r.Annotations {
        // TODO: Implement simple template rendering.
        // Replace {{ $labels.<name> }} with value from alertLabels.
        // Replace {{ $value }} with fmt.Sprint(value).
        // You can use strings.ReplaceAll or a simple regex.
        rendered[k] = tpl // Placeholder; replace with rendered version.
    }
    return rendered
}
```

Language-Specific Hints

- **Time and Duration:** Always use `time.Time` and `time.Duration`. When serializing to JSON, `time.Time` defaults to RFC3339 format. For custom JSON formatting, you can implement `MarshalJSON` / `UnmarshalJSON` methods.
- **Maps are Reference Types:** Be cautious when passing maps (like `Labels`) between functions. If you need to modify a map without affecting the original, make a copy: `copied := make(map[string]string); for k, v := range original { copied[k] = v }`.
- **Concurrency:** The `Engine.state` map is accessed concurrently by the evaluation loop and potentially by API handlers. Always use `stateMu.RLock()` for reads and `stateMu.Lock()` for writes.
- **JSON RawMessage:** The `Receiver.Config` field uses `json.RawMessage` to delay parsing. When implementing a specific receiver, unmarshal this into a concrete config struct: `var slackConfig SlackConfig; json.Unmarshal(receiver.Config, &slackConfig)`.

5.1 Component: Rule Evaluator

> **Milestone(s):** This section details the design and implementation of the Rule Evaluator, corresponding to **Milestone 1: Alert Rule Evaluation**.

The Rule Evaluator is the foundational component of the alerting system. It acts as the primary sensor, continuously monitoring the health of the system by evaluating user-defined rules against live metric data. Its core responsibility is to transform continuous, noisy metric streams into discrete, semantically meaningful alert events.

Mental Model: The Watchtower Guard

Imagine a guard stationed in a watchtower overlooking a factory floor filled with gauges and dials (metrics). Each gauge has a red line drawn at a critical threshold. The guard's duty is to periodically scan a list of instructions (alert rules). Each instruction says: "Check Gauge X. If its needle is above the red line for at least 5 consecutive scans, then raise a red flag (alert) with a specific message."

The guard's process is methodical:

1. **Scheduled Patrols:** The guard performs a patrol (evaluation cycle) at a fixed interval, say every 15 seconds.
2. **Reading Gauges:** For each instruction, the guard reads the current value of the specified gauge by querying the factory's instrument panel (metrics API).
3. **Condition Evaluation:** The guard compares the reading to the threshold using the specified operator (e.g., greater than).
4. **Stateful Judgment:** The guard doesn't panic at the first blip. If the condition is met, they note the time and move the alert to a "pending" state. Only if the condition *persists* for the entire "for" duration (e.g., 5 minutes) does the guard officially raise the "firing" alert flag.
5. **Flag Management:** Once raised, the flag remains until the gauge returns to normal. When it does, the guard lowers the flag and marks the alert as "resolved."

This component embodies that guard. It is stateful, patient (to avoid noise), and precise, ensuring that only sustained anomalies become alerts.

Interface & Algorithm

The Rule Evaluator's primary interface is its `Engine`, which orchestrates the periodic evaluation of all configured `Rule` objects.

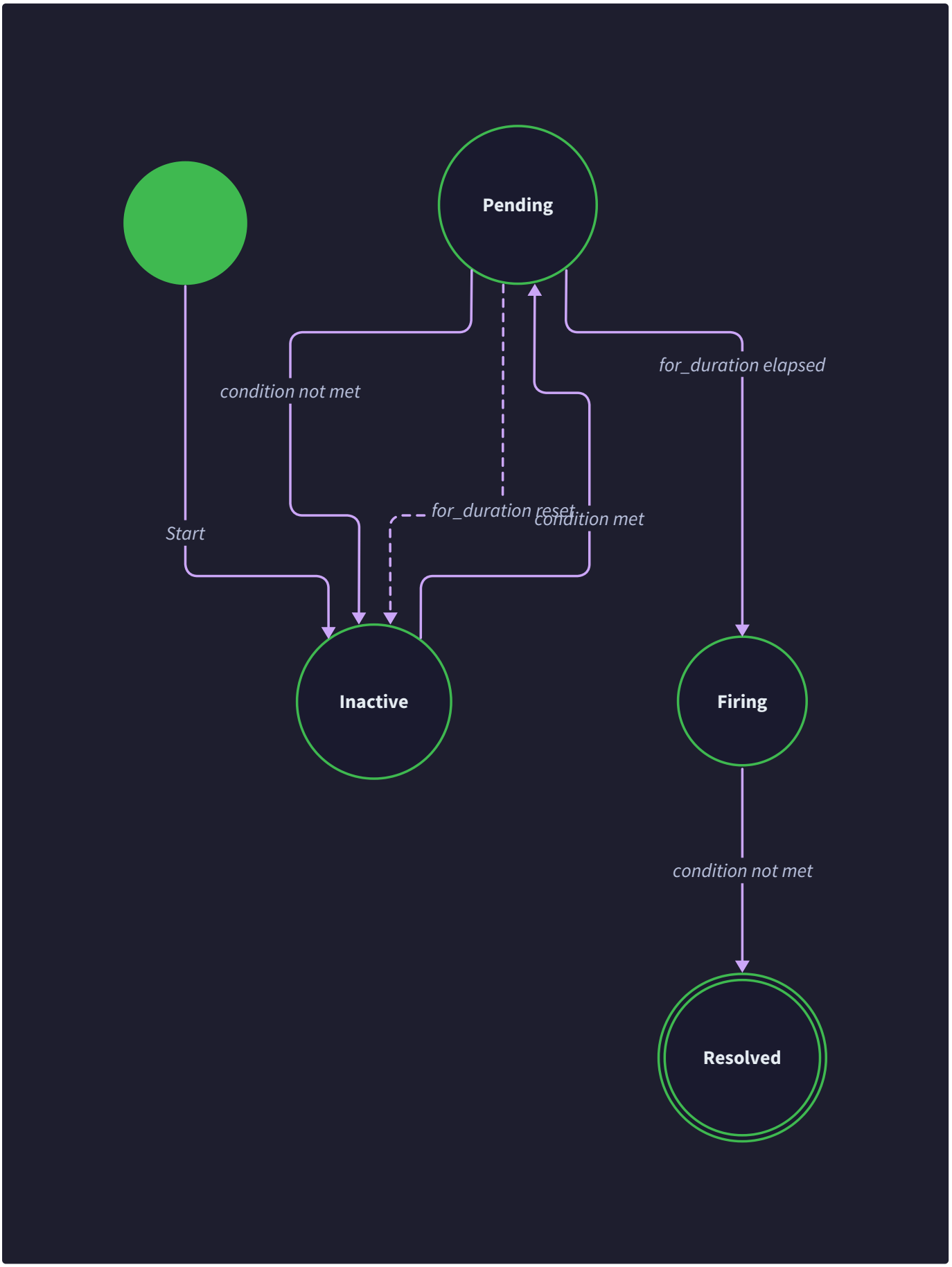
Core Data Structures & Methods:

The following table details the primary interface and data structures for the Rule Evaluator, as defined in the system's Data Model (Section 4).

Method / Type	Parameters / Fields	Returns / Type	Description
`Engine.Run(ctx context.Context)`	`ctx` : Context for cancellation.	-	Starts the main evaluation loop. It runs `evaluateAllRules` at the configured `interval` until `Stop()` is called or the context is cancelled.
`Engine.Stop()`	-	-	Signals the evaluation loop to stop and waits for the current cycle to complete.
`Engine.evaluateAllRules(ctx context.Context)` (internal)	`ctx` : Context for the evaluation cycle.	-	Iterates through all loaded `Rule` objects, calling `evaluateRule` for each one. Manages concurrency and error reporting for the cycle.
`Client.Query(ctx context.Context, query string)`	`ctx` : Context for the HTTP request. `query` : A PromQL-like expression string.	`(float64, error)`	Executes the query against the external Metrics Collector API. Returns the scalar numeric result of the evaluation. This is the "reading the gauge" operation.
`Rule` (struct)			Defines a single alert rule.
	`Name string`	-	Unique identifier for the rule (e.g., `"HighCPUUtilization"`).
	`Expression string`	-	The PromQL-like query that returns a numeric value to evaluate (e.g., `"avg(rate(container_cpu_usage_seconds_total[5m]))"`).
	`ForDuration time.Duration`	-	The duration the condition must hold before the alert transitions from `pending` to `firing` . A value of `0` means immediate firing.
	`Labels map[string]string`	-	Key-value pairs attached to any alert fired by this rule (e.g., `severity: "warning"`). These are static for the rule.
	`Annotations map[string]string`	-	Human-readable information (e.g., `summary` , `description`) that can contain templates populated with label values and the query result.
	`Operator string`	-	The comparison operator (`>` , `<` , `>=` , `<=` , `==` , `!=`).
	`Threshold float64`	-	The numeric value to compare the query result against.
`Alert` (struct)			Represents an individual alert instance.
	`Labels map[string]string`	-	A composite of the rule's static `Labels` and any labels from the metric result itself (e.g., `instance` , `job`). Uniquely identifies the alert.
	`Annotations map[string]string`	-	The rendered annotation templates, filled with concrete values.
	`StartsAt time.Time`	-	The timestamp when the alert's condition was first met (entered `pending` state).
	`EndsAt time.Time`	-	For a `resolved` alert, the time the condition stopped being true. For a `firing` alert, this is typically set far in the future.
	`State string`	-	One of: `"inactive"` , `"pending"` , `"firing"` , `"resolved"` .
	`GeneratorURL string`	-	A URL linking back to the rule definition or the metrics query for this alert.

State Transition Algorithm:

The heart of the evaluator is managing an alert's lifecycle. The following state machine governs this, triggered on each evaluation cycle for each unique alert (identified by its label set).


Pending
condition not met
for_duration elapsed
for_duration reset
condition met
Start
Inactive
Firing
condition not met
Resolved

Current State	Event / Condition Check	Next State	Actions Taken
`inactive`	Rule expression evaluates to `TRUE` (condition met).	`pending`	Create a new `Alert` object. Set `Labels` (rule + metric labels). Set `StartsAt` to current time. Call `Rule.RenderAnnotations()` to populate `Annotations`. Store alert in engine's `state` map.
`pending`	Condition is `TRUE` and `(current_time - StartsAt) >= ForDuration`.	`firing`	Update `Alert.State = "firing"`. Mark alert for output to the next component (Grouper).
`pending`	Condition is `FALSE`.	`inactive`	Remove the alert from the engine's `state` map. It is considered "forgotten"; no resolved notification is sent.
`firing`	Condition is `FALSE`.	`resolved`	Set `Alert.State = "resolved"`. Set `EndsAt` to current time. Mark alert for output (so a resolved notification can be sent). The alert will be removed from state after a grace period.
`firing`	Condition remains `TRUE`.	`firing`	No state change. The alert continues to be "firing". It may be re-output (idempotently) on each cycle.
`resolved`	(After a cleanup period, e.g., 1 hour)	`inactive`	Remove the alert from the engine's `state` map.

Step-by-Step Evaluation Algorithm for a Single Rule:

For each rule, the engine follows this precise sequence:

1. **Execute Query:** Call `Client.Query(ctx, rule.Expression)`. This returns a single numeric `float64` value representing the result of the PromQL expression at the current evaluation time.
2. **Compare to Threshold:** Apply the `rule.Operator` and `rule.Threshold` to the query result. The outcome is a boolean: `conditionMet`.
3. **For Each Unique Alert Series:** A single rule expression can return multiple time series (e.g., CPU usage per instance). The engine must process each unique series (identified by its metric labels) as a separate alert. It iterates through each series in the query result.
4. **Generate Alert Key:** For the current series, combine the rule's static `Labels` with the metric's labels (e.g., `instance="host:9100"`) to form the complete `Alert.Labels` set. The `Alert.Fingerprint()` of this label set is the unique key used to look up the alert's previous state in the engine's internal `state` map.
5. **Lookup State:** Check the `state` map using the fingerprint key. If no entry exists, the alert is `inactive`.
6. **Apply State Machine:** Follow the logic in the state transition table above, using `conditionMet`, the alert's current state, and the `ForDuration`.
7. **Handle State Changes:**
   - If transitioning *to* `pending` or `firing`: Create or update the `Alert` object, render annotations via `Rule.RenderAnnotations(...)`, and add it to an output batch for this cycle.
   - If transitioning *to* `resolved`: Update the `Alert` object and add it to the output batch.
   - If staying in `firing`: The existing alert may be re-added to the output batch to ensure downstream components see it as still active.
8. **Output:** After all rules are evaluated, the batch of new, updated, and resolved `Alert` objects is sent to the next component in the pipeline (the Grouper).

> **Key Insight:** The `ForDuration` is a critical anti-noise mechanism. It requires a condition to be *persistently* true, preventing transient spikes from generating alerts. The engine must track the `StartsAt` time for each pending alert to enforce this.

ADR: Pull vs. Push Metric Evaluation

> **Decision: Use a Pull Model for Metric Evaluation**
>
> - **Context:** The Rule Evaluator needs to assess system health by querying metric data. We must choose between periodically pulling data from a central metrics store (like Prometheus) or having metrics pushed into the alerting system from external agents.
> - **Options Considered:**
>   1. **Pull Model (Active Querying):** The Rule Evaluator acts as an HTTP client, periodically executing queries against a pre-existing Metrics Collector's query API.
>   2. **Push Model (Passive Reception):** External agents or the Metrics Collector itself would send metric samples to a listening endpoint on the Rule Evaluator, which would then evaluate rules against a sliding window of stored data.
> - **Decision:** Adopt the **Pull Model** using an HTTP client to query the Metrics Collector.
> - **Rationale:**
>   - **Leverages Prerequisite:** The project explicitly lists a completed Metrics Collector project as a prerequisite. This collector likely already exposes a Prometheus-compatible query API (`/api/v1/query`), making integration straightforward and reducing scope.
>   - **Decoupling & Simplicity:** The alerting system remains stateless with respect to metric history. It does not need to implement complex time-series storage, aggregation, or retention policies. It delegates all data management to the specialized collector.
>   - **Deterministic Evaluation:** Each evaluation cycle queries for the *exact* moment of evaluation, providing a consistent and repeatable view of the system state for threshold checking. Push models can introduce timing and ordering complexities.
>   - **Established Pattern:** This mirrors the widely-understood and proven Prometheus alerting architecture, making the system more intuitive for operators familiar with industry standards.
> - **Consequences:**
>   - **Introduces a Dependency:** The alerting system cannot function without a reachable and healthy Metrics Collector.
>   - **Evaluation Latency:** There is a delay between a metric changing and the next evaluation cycle detecting it. This is acceptable for most monitoring scenarios where alerting on sub-second latency is not required.
>   - **Load on Collector:** The evaluation frequency and complexity of queries contribute to the load on the Metrics Collector API. This must be considered during capacity planning.

The following table compares the two approaches:

Option	Pros	Cons	Chosen?
Pull Model	Simple, stateless evaluator. Leverages existing metrics infrastructure. Clear, time-bound queries.	Adds load to metrics API. Single point of failure (metrics collector). Inherits latency of poll interval.	**Yes**
Push Model	Potentially lower latency for alert detection. Evaluator can manage its own data window.	Requires implementing time-series storage and aggregation. Complex state management. Increases scope significantly ("scope creep").	No

Common Pitfalls & Mitigations

Implementing a robust Rule Evaluator involves navigating several subtle traps that can lead to unreliable or noisy alerting.

⚠️ Pitfall 1: Alert Flapping from Noisy Metrics

- **Description:** A metric value oscillates rapidly around the threshold (e.g., CPU at 74%, 76%, 74%, 76%). With a short or zero `ForDuration`, this causes the alert to constantly transition between `pending` / `firing` and `inactive` / `resolved`, generating a storm of alert and resolve notifications.
- **Why it's Wrong:** This causes **alert fatigue** and makes it impossible for operators to discern real, sustained issues from system noise.
- **Mitigation:** The primary tool is the `ForDuration`. Setting it to a sensible value (e.g., 2-5 minutes for infrastructure alerts) acts as a low-pass filter. For more advanced scenarios, one could implement hysteresis (e.g., alert at >75%, but only resolve when <70%).

⚠️ Pitfall 2: For_Duration Reset on Brief Normalcy

- **Description:** An alert is in `pending` for 4 minutes (of a required 5-minute `ForDuration`). On the next evaluation, the metric briefly dips to normal for one cycle, causing the state to revert to `inactive` and the `StartsAt` timer to be discarded. When the metric goes bad again, the 5-minute clock restarts from zero.
- **Why it's Wrong:** This can indefinitely delay a genuinely persistent problem from becoming a `firing` alert, defeating the purpose of the `ForDuration`.
- **Mitigation:** The state machine design must be precise. The transition from `pending` to `inactive` should only happen when the condition is `FALSE`. The timer (`StartsAt`) is not reset unless the alert is completely forgotten. Some systems add a "grace period" or implement a sliding window ("condition true for 5 out of the last 6 minutes") for more robustness.

⚠️ Pitfall 3: Template Rendering Errors Crash the Loop

- **Description:** The `Rule.RenderAnnotations` function uses Go's `text/template`. If a template references a label `{{ .Labels.instance }}` that doesn't exist in a particular metric series, it will panic, crashing the entire evaluation goroutine.

- **Why it's Wrong:** A single misconfigured rule can halt all alerting for the entire system.
- **Mitigation:** Execute template rendering within a deferred `recover()` block for each alert, or use the `text/template` package's option to define a custom error handler that returns a safe default string. Log the error and render the annotation with a placeholder (e.g., `<template error>`) to allow the alert to proceed, ensuring the system remains operational.

⚠️ Pitfall 4: Memory Leak from Unbounded Alert State

- **Description:** Every unique alert series (unique label set) that enters `pending` or `firing` state is stored in the engine's internal `state` map. If a rule matches a high-cardinality metric (e.g., per-request error codes), the map can grow infinitely, consuming all available memory.
- **Why it's Wrong:** The alerting system becomes its own cause of an outage.
- **Mitigation:** Implement active garbage collection. Periodically scan the `state` map and remove entries where the alert is `resolved` and the `EndsAt` time is older than a cleanup threshold (e.g., 1 hour). Consider cardinality limits per rule as a configuration safeguard.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
HTTP Client for Metrics	Standard `net/http` client with timeout context.	Client with connection pooling, circuit breaker (e.g., `sony/gobreaker`), and structured logging.
Scheduling	`time.Ticker` in a goroutine loop.	Dedicated scheduler library (e.g., `robfig/cron/v3`) for more complex evaluation schedules per rule.
Concurrency	`sync.RWMutex` for protecting the `state` map.	Using `sync.Map` for high read/write parallelism or sharding the state by rule name.
Template Rendering	Standard `text/template`.	`text/template` with custom functions for formatting, or a more secure sandboxed template engine.

B. Recommended File/Module Structure

Place the Rule Evaluator within the `internal` directory to enforce encapsulation.

```
alerting-system/
├── cmd/
│   └── alertengine/
│       └── main.go                # Service entry point, loads config, starts Engine.
├── internal/
│   ├── alert/                     # Core data types (Alert, Rule, etc.)
│   │   ├── alert.go
│   │   ├── rule.go
│   │   └── matcher.go             # Used later for silencing/routing.
│   ├── metricsclient/             # Client for querying the Metrics Collector.
│   │   ├── client.go
│   │   └── client_test.go
│   └── ruleengine/                # The Rule Evaluator component.
│       ├── engine.go              # Main Engine type and lifecycle methods.
│       ├── evaluator.go           # Core evaluation logic (evaluateAllRules, evaluateRule).
│       └── state.go               // Internal state map management and garbage collection.
├── configs/
│   └── rules.example.yaml         # Example rule configuration file.
└── go.mod
```

C. Infrastructure Starter Code

1. Metrics Client (`internal/metricsclient/client.go`): This is a complete, ready-to-use client for querying a Prometheus-compatible API.

```go
package metricsclient

import (
    "context"
    "encoding/json"
    "fmt"
    "net/http"
    "net/url"
    "time"
)

// MetricQueryResult mirrors the Prometheus query API response structure.
type MetricQueryResult struct {
    Status string `json:"status"`
    Data   struct {
        ResultType string `json:"resultType"`
        Result     []struct {
            Metric map[string]string `json:"metric"`
            Value  []interface{}     `json:"value"` // [timestamp, value]
        } `json:"result"`
    } `json:"data"`
}

// Client wraps the HTTP client and base URL.
type Client struct {
    baseURL    string
    httpClient *http.Client
}

// NewClient creates a new metrics client with a default HTTP client.
func NewClient(baseURL string) *Client {
    return &Client{
        baseURL: baseURL,
        httpClient: &http.Client{
            Timeout: 30 * time.Second,
        },
    }
}

// Query executes a PromQL instant query and returns the first scalar result.
// It expects a query that returns a single time series with a single value.
func (c *Client) Query(ctx context.Context, query string) (float64, error) {
    u, err := url.Parse(c.baseURL + "/api/v1/query")
```

```
    if err != nil {
        return 0, fmt.Errorf("invalid base URL: %w", err)
    }

    q := u.Query()
    q.Set("query", query)
    // Query for the current evaluation time.
    q.Set("time", time.Now().Format(time.RFC3339))
    u.RawQuery = q.Encode()

    req, err := http.NewRequestWithContext(ctx, "GET", u.String(), nil)
    if err != nil {
        return 0, fmt.Errorf("creating request: %w", err)
    }

    resp, err := c.httpClient.Do(req)
    if err != nil {
        return 0, fmt.Errorf("executing query: %w", err)
    }
    defer resp.Body.Close()

    if resp.StatusCode != http.StatusOK {
        return 0, fmt.Errorf("unexpected status code: %d", resp.StatusCode)
    }

    var result MetricQueryResult
    if err := json.NewDecoder(resp.Body).Decode(&result); err != nil {
        return 0, fmt.Errorf("decoding response: %w", err)
    }

    if result.Status != "success" {
        return 0, fmt.Errorf("query status not success: %s", result.Status)
    }

    // Expect a single result of type "vector" or "scalar".
    if len(result.Data.Result) == 0 {
        return 0, nil // No data is not an error; it's equivalent to a zero value.
    }

    // For simplicity, we take the first result's value.
    // In a full implementation, you'd iterate through all results.
    val := result.Data.Result[0].Value[1]
    switch v := val.(type) {
    case string:
```

```
        var f float64
        if _, err := fmt.Sscanf(v, "%f", &f); err != nil {
            return 0, fmt.Errorf("parsing float from string %q: %w", v, err)
        }
        return f, nil
    case float64:
        return v, nil
    default:
        return 0, fmt.Errorf("unexpected value type: %T", val)
    }
}
```

D. Core Logic Skeleton Code

1. Engine Skeleton (`internal/ruleengine/engine.go`):

```go
package ruleengine

import (
    "context"
    "sync"
    "time"
    "alerting-system/internal/alert"
    "alerting-system/internal/metricsclient"
)

// Engine is the central coordinator for rule evaluation.
type Engine struct {
    rules   []*alert.Rule
    client  *metricsclient.Client
    interval time.Duration

    stateMu sync.RWMutex
    // Map key: Alert.Fingerprint(), value: the current Alert object.
    state   map[string]*alert.Alert

    stopCh chan struct{}
}

// NewEngine creates a new rule evaluation engine.
func NewEngine(rules []*alert.Rule, client *metricsclient.Client, interval time.Duration) *Engine {
    return &Engine{
        rules:   rules,
        client:  client,
        interval: interval,
        state:   make(map[string]*alert.Alert),
        stopCh:  make(chan struct{}),
    }
}

// Run starts the periodic evaluation loop. It blocks until Stop() is called or ctx is cancelled.
func (e *Engine) Run(ctx context.Context) {
    ticker := time.NewTicker(e.interval)
    defer ticker.Stop()

    for {
        select {
        case <-ticker.C:
            // TODO 1: Start a new evaluation cycle. Call e.evaluateAllRules(ctx)
            // TODO 2: Handle any non-critical errors from evaluation (log them, don't panic).
```

```
        case <-e.stopCh:
            return
        case <-ctx.Done():
            return
        }
    }
}

// Stop signals the engine to stop evaluating rules.
func (e *Engine) Stop() {
    close(e.stopCh)
}

// evaluateAllRules is the internal method called on each tick.
func (e *Engine) evaluateAllRules(ctx context.Context) error {
    // TODO 1: Create a slice to collect alerts for output in this cycle.
    // TODO 2: Iterate through e.rules. For each rule, call e.evaluateRule(ctx, rule).
    // TODO 3: For each alert returned from evaluateRule, add it to the output slice.
    // TODO 4: Send the output slice to the next component (e.g., via a channel). This will be connected to the Grouper.
    // TODO 5: Perform garbage collection on the state map (remove old resolved alerts).
    return nil
}

// evaluateRule evaluates a single rule and returns any alerts that changed state.
func (e *Engine) evaluateRule(ctx context.Context, rule *alert.Rule) ([]*alert.Alert, error) {
    var outputAlerts []*alert.Alert

    // TODO 1: Call e.client.Query(ctx, rule.Expression) to get the metric result (float64).
    // TODO 2: Apply the rule.Operator and rule.Threshold to determine if conditionMet is true/false.
    // TODO 3: Process the result. (Note: The starter client returns a single value. Extend this to handle multiple series from result.Data.Result).
    // TODO 4: For each unique metric series in the result (identified by its labels):
    //   a. Combine rule.Labels and metric labels to form the final alertLabels.
    //   b. Generate a fingerprint key from alertLabels (use alert.Fingerprint()).
    //   c. Take a read lock on e.stateMu and look up the existing alert state by key.
    //   d. Apply the state transition logic (refer to the state table):
    //        - If inactive and conditionMet -> create new pending alert.
    //        - If pending, conditionMet, and ForDuration elapsed -> transition to firing.
    //        - If pending and !conditionMet -> delete from state (inactive).
    //        - If firing and !conditionMet -> transition to resolved.
    //        - If firing and conditionMet -> keep firing (re-output).
    //   e. For any state change (new, to firing, to resolved), create/update the Alert object.
    //        - Set StartsAt, EndsAt, State, Labels, Annotations.
```

```
    //        - Render annotations using rule.RenderAnnotations(alertLabels, queryResult).
    //   f. If the alert is new or changed, add it to outputAlerts.
    //   g. Update e.state map with the new alert state (acquire write lock).
    // TODO 5: Return the slice of outputAlerts.
    return outputAlerts, nil
}
```

2. Alert Helper Methods (`internal/alert/alert.go`):

```go
package alert

import (
    "crypto/md5"
    "encoding/hex"
    "fmt"
    "sort"
    "strings"
    "time"
)

// Fingerprint generates a unique string identifier for an alert based on its labels.
// It sorts the label keys, concatenates key=value pairs, and returns an MD5 hash.
func (a *Alert) Fingerprint() string {
    if len(a.Labels) == 0 {
        return ""
    }
    var keys []string
    for k := range a.Labels {
        keys = append(keys, k)
    }
    sort.Strings(keys)

    var sb strings.Builder
    for i, k := range keys {
        if i > 0 {
            sb.WriteByte('\xff') // Use a separator not allowed in label values.
        }
        sb.WriteString(k)
        sb.WriteByte('=')
        sb.WriteString(a.Labels[k])
    }
    hash := md5.Sum([]byte(sb.String()))
    return hex.EncodeToString(hash[:])
}

// IsActive returns true if the alert is in a state that should be considered "active"
// (i.e., pending or firing). Resolved and inactive alerts return false.
func (a *Alert) IsActive() bool {
    return a.State == "pending" || a.State == "firing"
}
```

3. Rule Helper Methods (`internal/alert/rule.go`):

```go
package alert

import (
    "bytes"
    "fmt"
    "text/template"
    "time"
)

// Validate checks that the rule has all required fields and valid settings.
func (r *Rule) Validate() error {
    if r.Name == "" {
        return fmt.Errorf("rule name is required")
    }
    if r.Expression == "" {
        return fmt.Errorf("rule expression is required")
    }
    // Add validation for operator (must be one of allowed set).
    allowedOps := map[string]bool{">": true, "<": true, ">=": true, "<=": true, "==": true, "!=": true}
    if !allowedOps[r.Operator] {
        return fmt.Errorf("invalid operator: %s", r.Operator)
    }
    if r.ForDuration < 0 {
        return fmt.Errorf("ForDuration cannot be negative")
    }
    return nil
}

// RenderAnnotations fills in the annotation templates using the provided label set and query value.
func (r *Rule) RenderAnnotations(alertLabels map[string]string, value float64) (map[string]string, error) {
    rendered := make(map[string]string, len(r.Annotations))
    data := struct {
        Labels map[string]string
        Value  float64
    }{
        Labels: alertLabels,
        Value:  value,
    }

    for key, tmplStr := range r.Annotations {
        tmpl, err := template.New(key).Parse(tmplStr)
        if err != nil {
```

```
            return nil, fmt.Errorf("parsing template for annotation %q: %w", key, err)
        }
        var buf bytes.Buffer
        // Execute the template safely. Consider adding a custom error handler to the template.
        if err := tmpl.Execute(&buf, data); err != nil {
            // Instead of failing, you could log and set a default.
            return nil, fmt.Errorf("executing template for annotation %q: %w", key, err)
        }
        rendered[key] = buf.String()
    }
    return rendered, nil
}
```

E. Language-Specific Hints

- **Concurrency:** Use `sync.RWMutex` for the `state` map. Acquire a read lock (`RLock()`) when looking up state in `evaluateRule`, and upgrade to a write lock (`Lock()`) only when you need to insert or update an alert. This allows multiple rules to be evaluated concurrently with minimal contention.
- **Time:** Use `time.Now().UTC()` for consistent timestamps. When checking `ForDuration`, use `time.Since(alert.StartsAt) >= rule.ForDuration`.
- **Templates:** The `text/template` package is powerful. Consider defining custom template functions (using `template.FuncMap`) for common operations like formatting durations or rounding numbers.
- **Error Handling:** In `evaluateAllRules`, if one rule's evaluation fails (e.g., due to a network error querying metrics), log the error but continue evaluating the remaining rules. Do not let one bad rule halt the entire system.

F. Milestone Checkpoint

After implementing the Rule Evaluator, you should be able to verify its core functionality.

1. **Unit Tests:** Run the rule engine tests.

```
    go test ./internal/ruleengine/... -v
```
BASH

`Expected: Tests for state transitions (`inactive` -> `pending` -> `firing` -> `resolved`) pass, particularly around `ForDuration` l`

2. **Integration Test:**
    - Start a mock Metrics Collector that serves a simple HTTP endpoint returning a fixed numeric value.
    - Configure a rule (e.g., `expression="some_metric"`, `operator=">"`, `threshold=10`, `ForDuration=10s`).
    - Start the `Engine`.
    - **Verify:** If the mock returns `15`, after 10 seconds you should see log output indicating a `firing` alert was generated. If you then change the mock to return `5`, you should see a `resolved` alert.
3. **Signs of Trouble:**
    - **No alerts after condition is met:** Check the `Client.Query` is returning the expected value. Verify the `conditionMet` logic and the `ForDuration` timer.
    - **Alerts fire immediately ignoring ForDuration:** Ensure you are tracking `StartsAt` and comparing it with `time.Since()` correctly. The transition from `pending` to `firing` must check the duration.
    - **High CPU/Memory:** You may have a goroutine leak if `Stop()` isn't working, or a memory leak if resolved alerts aren't being cleaned up from the `state` map.

5.2 Component: Grouper

> **Milestone(s):** This section details the design and implementation of the Grouper component, corresponding to **Milestone 2: Alert Grouping**.

The Grouper is the system's noise reduction engine. It transforms a stream of discrete alerts into consolidated notification bundles, dramatically reducing the cognitive load on human operators. By treating each alert as a data point and grouping related ones, it enables operators to see patterns and root causes instead of being overwhelmed by individual events. This component is pivotal in transforming raw technical signals into actionable operational intelligence.

Mental Model: The Mail Sorting Desk

Imagine a massive post office processing millions of letters. If every letter was delivered individually, delivery vans would make constant trips to the same neighborhoods, overwhelming both the drivers and the recipients. Instead, the postal service uses a **sorting desk**:

1. **Incoming Letters (Alerts)**: Each letter arrives with an address (labels like `alertname=HighCPU`, `cluster=prod`, `service=api`).
2. **Sorting Bins (Groups)**: The sorter examines each letter's address and places it into a bin designated for a specific neighborhood. Letters for "Downtown Prod API" go in one bin, while letters for "West Coast Monitoring" go in another.
3. **Bundling for Delivery**: Instead of delivering each letter immediately, the sorter waits a short time (`group_wait`) to collect more letters heading to the same neighborhood. Once the wait timer expires or the bin becomes full, all letters for that neighborhood are bundled together into a single package.
4. **Scheduled Pickups**: After the first delivery, if more letters arrive for that same neighborhood, the sorter doesn't send another van immediately. Instead, they schedule periodic pickups (`group_interval`) to deliver accumulated letters in batches, avoiding constant trips.

This model illustrates the core benefits: **reduced delivery trips** (fewer notifications), **contextual bundling** (related alerts stay together), and **batching efficiency** (operators get comprehensive updates rather than constant interruptions). The Grouper is precisely this sorting desk for alerts.

Interface & Algorithm

The Grouper exposes a minimal interface for ingesting alerts and manages the complex lifecycle of alert groups internally. Its primary responsibility is to maintain a collection of `Group` objects, each representing a logical bundle of related alerts.

Core Data Structures

Type	Fields	Description
`Group`	`key string`	A deterministic identifier for this group, derived from the grouping label values.
	`alerts []*alert.Alert`	The current set of alerts belonging to this group. Maintained as a slice for ordered iteration.
	`timers map[string]*time.Timer`	Active timers for this group, keyed by timer type ("wait", "interval"). Required for managing notification timing.
`Notification`	`GroupKey string`	The key of the group being notified. Allows the receiver to correlate notifications.
	`Alerts []*alert.Alert`	The consolidated list of alerts in the group at the moment of notification generation.
	`Receiver string`	The name of the receiver (e.g., "slack-prod-team") this notification is destined for.

Interface Specification

Method	Parameters	Returns	Description
`Grouper.Process(alert *alert.Alert)`	`alert`: The alert to ingest (may be new, updated, or resolved).	(none)	The primary entry point. Determines the alert's group based on configured grouping labels, adds it to the appropriate `Group`, and manages group lifecycle timers.
`Grouper.FlushGroup(groupKey string)`	`groupKey`: The identifier of the group to flush.	(none)	Triggers immediate notification generation for the specified group. Used when a group's wait timer expires or when an alert resolves and the group becomes empty.
`Grouper.GetGroupKeys()`	(none)	`[]string`	Returns all active group keys. Useful for debugging and administrative UIs.
`Grouper.GetGroup(key string)`	`key`: The group identifier.	`*Group`, `bool`	Retrieves a specific group by its key, with a boolean indicating if it was found.

Group Lifecycle Algorithm

The Grouper's core algorithm is triggered every time `Grouper.Process` is called with a new or updated alert. The following numbered steps describe the procedure:

1. **Calculate Group Key**: Extract the values of the **grouping labels** (configurable, e.g., `["alertname", "cluster", "service"]`) from the incoming alert's `Labels` map. Sort the label names alphabetically, retrieve their values, and concatenate them into a deterministic string (e.g., `"HighCPU|prod|api"`). This string becomes the alert's group key.
2. **Locate or Create Group**: Using the calculated group key, look up the corresponding `Group` in an internal map. If it doesn't exist, create a new `Group` object with an empty alerts slice and initialize its timers map.
3. **Update Group Membership**:
    - If the alert is **new** (no matching `Fingerprint` in the group's alerts), append it to the group's `alerts` slice.

- If the alert **already exists** in the group (matched by `Fingerprint`), update the existing entry with the new alert's state, `StartsAt`, `EndsAt`, and annotations.
- If the alert is **resolved** (state is `"resolved"`), mark it as such within the group but do not immediately remove it; it remains in the group until a flush occurs.

4. **Manage Group Timers**:
   - **Initial Wait Timer (`group_wait`)**: If this is the **first alert** added to a newly created group, start a timer for the configured `group_wait` duration (e.g., 30 seconds). When this timer fires, it calls `Grouper.FlushGroup(key)`.
   - **Subsequent Interval Timer (`group_interval`)**: If the group already exists and its initial wait timer has already fired (meaning at least one notification has been sent), manage the interval timer. On each new alert arrival, **reset** the interval timer. This ensures that after the first notification, subsequent alerts are batched and sent at most every `group_interval` (e.g., 5 minutes).
5. **Handle Immediate Flush Conditions**: Certain events trigger an immediate flush, bypassing timers:
   - If an alert's state changes to `"resolved"` and this resolves the **last active alert** in the group, flush immediately to send a resolution notification.
   - If a group exceeds a configurable maximum size (e.g., 100 alerts), flush immediately to prevent memory exhaustion.
6. **Notification Generation (in `FlushGroup`)**:
   - Collect all alerts in the group. Filter out resolved alerts if they've been resolved for longer than a configurable retention period (e.g., 5 minutes).
   - If the filtered alert list is empty, delete the group and clean up its timers.
   - Otherwise, construct a `Notification` object containing the group key, the list of alerts (including resolved ones if within retention), and the receiver determined by the routing layer.
   - Pass the `Notification` to the downstream `Router.Route` method for delivery.
   - After flushing, if the group still contains active alerts, restart the `group_interval` timer for the next batch.

Configuration Parameters

The Grouper's behavior is controlled by a set of parameters, typically defined within a `Route` object:

Parameter	Type	Default	Description
`GroupBy`	`[]string`	`["alertname"]`	List of label names used to calculate the group key. Alerts sharing the same values for these labels are grouped together.
`GroupWait`	`time.Duration`	`30s`	Time to buffer alerts in a new group before sending the first notification. Allows time for related alerts to arrive.
`GroupInterval`	`time.Duration`	`5m`	Minimum time between sending notifications for the same group. Subsequent alerts reset the timer, implementing batching.
`RepeatInterval`	`time.Duration`	`4h`	(Note: This is often a router-level parameter for re-sending firing alerts, not directly used by the Grouper's core algorithm but relevant for the overall notification timing).

ADR: Group Key Generation Strategy

Decision: Deterministic String Concatenation with Label Sorting

- **Context**: The Grouper must partition incoming alerts into stable, deterministic groups. The grouping key must be efficiently computable, comparable, and must not change for the same set of label values, even if the alert arrives in a different order. We need a method that balances performance, simplicity, and correctness.
- **Options Considered**:
  1. **Hash-based (e.g., MD5, SHA1 of concatenated values)**: Compute a cryptographic hash of the sorted label key-value pairs.
  2. **Direct String Concatenation with Delimiters**: Sort label keys, then join their values with a delimiter (e.g., `|`).
  3. **Serialized JSON Object**: Serialize the sorted label map to a JSON string and use that as the key.
- **Decision**: **Direct String Concatenation with Delimiters**.
- **Rationale**:
  - **Simplicity & Readability**: The resulting key is human-readable (e.g., `HighCPU|prod|api`), which aids immensely in debugging and logging. Hash-based keys are opaque.
  - **Performance**: String concatenation is significantly faster than computing cryptographic hashes. While performance is not the primary bottleneck, it matters at high alert volumes.
  - **Adequate Uniqueness**: The combination of label values for the chosen `GroupBy` labels is naturally unique for distinct groups. We don't need the collision resistance of a cryptographic hash.
  - **Stability**: Sorting the label keys ensures the same key is generated regardless of the order labels appear in the alert.
- **Consequences**:
  - The key may contain special characters if label values do. We must choose a delimiter that does not appear in valid label values (Prometheus labels restrict characters to `[a-zA-Z0-9_]` and the delimiter `|` is safe).
  - Key length grows linearly with the number of grouping labels and their values. This is acceptable as `GroupBy` typically uses few labels (2-3).
  - The key is not cryptographically secure, but this is irrelevant for grouping.

Option	Pros	Cons	Chosen?
Hash-based (SHA1)	- Extremely low collision probability. - Fixed-length output.	- Opaque, hard to debug. - Slightly more CPU intensive. - Overkill for the use case.	❌
Direct String Concatenation	**- Human-readable, excellent for debugging. - Very fast computation. - Naturally unique for intended scope.**	**- Potential for delimiter collisions (mitigated by choice of `|`). - Variable length.**	✅
JSON Serialization	- Self-describing format. - Built-in library support.	- Verbose, longer keys. - Slightly slower than simple concatenation. - Still requires sorting for stability.	❌

The algorithm for key generation is:

1. Extract the values for each label name in the `GroupBy` list from the alert's `Labels`.
2. Sort the `GroupBy` label names alphabetically.
3. For each sorted label name, retrieve its value (use empty string if label is missing).
4. Join all values with the delimiter `|`.

Example: For `GroupBy = ["alertname", "cluster", "service"]` and alert labels `{cluster="prod", alertname="HighCPU", service="api", instance="host1"}`, the sorted keys are `["alertname", "cluster", "service"]`, values are `["HighCPU", "prod", "api"]`, and the resulting group key is `"HighCPU|prod|api"`.

Common Pitfalls & Mitigations

⚠️ Pitfall: Orphaned Alerts Due to Dynamic Label Changes

- **Description**: An alert's labels change after it's been placed in a group (e.g., due to a template rendering error correction or a dynamic label update). This changes its group key, but the alert remains in the old group. The new group key may not receive the alert, causing it to be "orphaned" – it never appears in notifications or disappears from view.
- **Why it's wrong**: Alerts become invisible to operators, defeating the purpose of alerting. The system state becomes inconsistent: the alert exists but is not grouped correctly.
- **Mitigation**: In `Grouper.Process`, after calculating the new group key, check if the alert's `Fingerprint` already exists in any group (maintain a fingerprint-to-group-key map). If it does and the group key differs, **move** the alert from the old group to the new group. This requires updating both group's alert lists and timers.

⚠️ **Pitfall: Excessive `group_wait` Delaying Critical Alerts**

- **Description**: Setting a long `group_wait` (e.g., 10 minutes) to batch more alerts can dangerously delay the first notification for a critical, standalone alert.
- **Why it's wrong**: For a severe incident represented by a single alert, operators lose precious response time waiting for the batching window to expire.
- **Mitigation**: Implement **dual-wait strategies** inspired by Prometheus Alertmanager: Use a short `group_wait` for groups with high-severity alerts (e.g., severity=critical). This can be achieved by configuring different `group_wait` values in routing trees based on label matchers (e.g., a route matching `severity=critical` has `group_wait: 0s`). Alternatively, implement an immediate flush if any alert in the group has a severity label indicating criticality.

⚠️ **Pitfall: Memory Leak from Stale Empty Groups**

- **Description**: When all alerts in a group resolve, the group may be flushed and cleared. If the cleanup logic is incomplete (e.g., timers not stopped, group not removed from the internal map), empty groups accumulate, causing a memory leak.
- **Why it's wrong**: Over time, the system consumes increasing memory without benefit, potentially leading to out-of-memory crashes.
- **Mitigation**: In the `FlushGroup` method, after sending a notification, explicitly check if the group's alert list is empty (or contains only resolved alerts beyond the retention period). If so, **stop and remove all associated timers** from the `Group.timers` map, **delete** the group from the main groups map, and **remove** all fingerprint-to-group-key mappings for alerts that were in this group.

⚠️ **Pitfall: Notification Storm from Rapid-Fire Timer Resets**

- **Description**: With the default algorithm, each new alert in an existing group resets the `group_interval` timer. Under a high alert rate, this can perpetually delay the next notification, causing a backlog. When the alert rate finally slows, a massive batch notification is sent, overwhelming the receiver.
- **Why it's wrong**: Defeats the purpose of `group_interval` as a rate limiter and can cause a notification storm when the timer finally fires.
- **Mitigation**: Implement a **non-resettable interval timer**. Once the `group_interval` timer is started, do not reset it on new alerts. Instead, track that new alerts have arrived. When the timer fires, flush all accumulated alerts and restart the timer only if the group still has active alerts. This guarantees at least `group_interval` seconds between notifications, providing predictable rate limiting.

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Timer Management	`time.Timer` with `Reset()` (Go standard library). Simple but requires careful state management to avoid race conditions and reset pitfalls.	Dedicated timer wheel or heap (e.g., using `container/heap` with custom priority queue). More complex but scales better to tens of thousands of groups with efficient timer operations.
Concurrent Access	`sync.RWMutex` protecting the map of groups. Straightforward and sufficient for moderate load.	Partitioned sharding (array of mutexes and maps based on group key hash). Reduces lock contention for high-throughput systems.
Group Key Storage	Store as plain string in `Group.key`. Fast and simple.	Interned strings (using a `sync.Map` or custom interning cache). Reduces memory footprint when many groups share similar label values.

B. Recommended File/Module Structure

```
alerting-system/
├── cmd/
│   └── alertmanager/          # Main entry point
├── internal/
│   ├── alert/                 # Core data types (Alert, Rule, etc.)
│   │   └── alert.go
│   ├── grouper/               # Grouper component
│   │   ├── grouper.go         # Main Grouper struct and public interface
│   │   ├── group.go           # Group struct and methods
│   │   ├── timers.go          # Timer management utilities
│   │   └── grouper_test.go    # Unit tests
│   ├── router/                # Router component (Milestone 4)
│   └── metricsclient/         # Client for querying metrics (Milestone 1)
└── pkg/
    └── notification/          # Notification struct and sender interfaces (shared)
```

C. Infrastructure Starter Code

Below is a complete, reusable implementation for a safe timer manager that avoids common `time.Timer` reset pitfalls (resetting a timer that has already fired can cause panics or unpredictable behavior).

```go
// internal/grouper/timers.go

package grouper


import (

    "sync"

    "time"

)


// timerManager safely manages a timer that can be reset or stopped.

// It encapsulates the complexity of timer state transitions.

type timerManager struct {

    timer    *time.Timer

    duration time.Duration

    callback func()

    mu       sync.Mutex

    stopped  bool

}


// newTimerManager creates a new timerManager. The timer is initially stopped.

// duration is the timer interval, callback is the function to call when the timer fires.

func newTimerManager(duration time.Duration, callback func()) *timerManager {

    tm := &timerManager{

        duration: duration,

        callback: callback,

        stopped:  true,

    }

    return tm

}


// startOrReset restarts the timer with its configured duration.

// If the timer is already running, it is stopped and a new timer is created.

// If the timer was stopped, it is started.

// This method is safe to call from multiple goroutines.

func (tm *timerManager) startOrReset() {

    tm.mu.Lock()

    defer tm.mu.Unlock()


    if tm.stopped {

        tm.timer = time.AfterFunc(tm.duration, tm.callback)

        tm.stopped = false

    } else {

        // Stop the existing timer and drain the channel if it fired.

        if !tm.timer.Stop() {
```

```
            // If the timer already fired, we need to drain the channel
            // to prevent the callback from executing later.
            <-tm.timer.C
        }
        tm.timer.Reset(tm.duration)
    }
}

// stop stops the timer if it is running. It ensures the timer channel is drained
// to prevent the callback from executing.
func (tm *timerManager) stop() {
    tm.mu.Lock()
    defer tm.mu.Unlock()

    if !tm.stopped {
        if !tm.timer.Stop() {
            <-tm.timer.C
        }
        tm.stopped = true
    }
}

// isStopped returns whether the timer is currently stopped.
func (tm *timerManager) isStopped() bool {
    tm.mu.Lock()
    defer tm.mu.Unlock()
    return tm.stopped
}
```

D. Core Logic Skeleton Code

The following skeleton provides the main Grouper implementation with detailed TODO comments mapping to the algorithm steps described earlier.

```go
// internal/grouper/grouper.go

package grouper


import (

    "sort"

    "strings"

    "sync"

    "time"


    "github.com/yourorg/alerting-system/internal/alert"

)


// Grouper aggregates alerts into groups based on configurable labels.

type Grouper struct {

    // config holds grouping parameters (GroupBy, GroupWait, GroupInterval).

    config Config


    // groups maps groupKey -> *Group.

    groups map[string]*Group

    // fingerprintToKey maps alert fingerprint -> current groupKey.

    // Used to detect and handle label changes that move an alert between groups.

    fingerprintToKey map[string]string


    mu sync.RWMutex


    // flushFunc is called when a group is ready to send a notification.

    // In production, this would be set to call the Router.

    flushFunc func(groupKey string, alerts []*alert.Alert)

}


// Config holds grouping configuration.

type Config struct {

    GroupBy        []string      // Labels to group by (e.g., ["alertname", "cluster"])

    GroupWait      time.Duration // Time to wait before sending first notification

    GroupInterval  time.Duration // Minimum time between notifications for the same group

    Retention      time.Duration // How long to keep resolved alerts in a group before removing

}


// NewGrouper creates a new Grouper with the given configuration and flush callback.

func NewGrouper(config Config, flushFunc func(string, []*alert.Alert)) *Grouper {

    return &Grouper{

        config:           config,

        groups:           make(map[string]*Group),

        fingerprintToKey: make(map[string]string),
```

```
        flushFunc:        flushFunc,
    }
}

// Process ingests a new or updated alert into the grouping system.
// This is the main entry point.
func (g *Grouper) Process(a *alert.Alert) {
    // TODO 1: Calculate the group key for this alert.
    //   - Extract values for each label in g.config.GroupBy from a.Labels.
    //   - Sort the GroupBy label names alphabetically.
    //   - For each sorted label name, get its value (empty string if missing).
    //   - Join all values with "|" to form the key.

    // TODO 2: Check if this alert (by fingerprint) already belongs to a different group.
    //   - Lock the mutex for writing (g.mu.Lock()).
    //   - oldKey, exists := g.fingerprintToKey[a.Fingerprint()]
    //   - If exists and oldKey != newKey, the alert has moved. Call g.moveAlert(a, oldKey, newKey).

    // TODO 3: Find or create the group for newKey.
    //   - group, ok := g.groups[newKey]
    //   - If not ok, create a new Group with g.createGroup(newKey).

    // TODO 4: Update the group with this alert.
    //   - Call group.UpsertAlert(a) to add or update the alert in the group's internal slice.

    // TODO 5: Update the fingerprint mapping.
    //   - g.fingerprintToKey[a.Fingerprint()] = newKey

    // TODO 6: Manage group timers based on the group's state and the alert.
    //   - If the group was just created (first alert), start the group_wait timer.
    //   - If the group already exists and has already sent its first notification, manage the group_interval timer.
    //   - Use the timerManager from timers.go for safe timer operations.

    // TODO 7: Check for immediate flush conditions.
    //   - If the alert is resolved and it's the last active alert in the group, call g.FlushGroup(newKey) immediately.
    //   - If the group size exceeds a limit (e.g., 100), call g.FlushGroup(newKey) immediately.

    // TODO 8: Unlock the mutex (g.mu.Unlock()).
}

// createGroup initializes a new Group and starts its initial group_wait timer.
func (g *Grouper) createGroup(key string) *Group {
    // TODO: Create a new Group struct with the given key.
    // TODO: Initialize its alerts slice and timers map.
    // TODO: Start a timerManager for the group_wait duration.
```

```
    //   - The callback should call g.FlushGroup(key).
    //   - Store this timer in the group's timers map under "wait".
    // TODO: Return the group.
}

// moveAlert handles an alert changing groups (due to label change).
func (g *Grouper) moveAlert(a *alert.Alert, oldKey, newKey string) {
    // TODO 1: Remove the alert from the old group.
    //   - oldGroup := g.groups[oldKey]
    //   - oldGroup.RemoveAlert(a.Fingerprint())

    // TODO 2: If the old group is now empty, clean it up (stop timers, delete from map).
    //   - Call g.cleanupGroupIfEmpty(oldKey)

    // TODO 3: Update the fingerprint mapping to point to the new key.
    //   - g.fingerprintToKey[a.Fingerprint()] = newKey
}

// FlushGroup generates a consolidated notification for a group and sends it via flushFunc.
func (g *Grouper) FlushGroup(key string) {
    // TODO 1: Lock the mutex (g.mu.Lock()).
    // TODO 2: Retrieve the group by key.
    // TODO 3: If group doesn't exist, unlock and return.
    // TODO 4: Filter alerts: remove resolved alerts older than g.config.Retention.
    // TODO 5: If after filtering there are no alerts, delete the group and clean up.
    //   - Call g.cleanupGroup(key)
    //   - Unlock and return.
    // TODO 6: If there are alerts, call g.flushFunc(key, filteredAlerts) to send the notification.
    // TODO 7: After flushing, if the group still has active alerts, restart the group_interval timer.
    //   - Ensure the group_interval timer is set (not reset on new alerts, see pitfall mitigation).
    // TODO 8: Unlock the mutex (g.mu.Unlock()).
}

// cleanupGroup stops all timers and removes a group entirely.
func (g *Grouper) cleanupGroup(key string) {
    // TODO 1: Stop all timers in the group (both wait and interval).
    // TODO 2: Delete the group from g.groups map.
    // TODO 3: For each alert that was in this group, delete its entry from fingerprintToKey.
}

// GetGroupKeys returns all active group keys (for debugging/admin UI).
func (g *Grouper) GetGroupKeys() []string {
    g.mu.RLock()
```

```
    defer g.mu.RUnlock()
    keys := make([]string, 0, len(g.groups))
    for k := range g.groups {
        keys = append(keys, k)
    }
    return keys
}

// calculateGroupKey computes the deterministic group key for an alert.
func (g *Grouper) calculateGroupKey(a *alert.Alert) string {
    // TODO: Implement as described in step 1 of the algorithm.
    // Hint: Use sort.Strings on a copy of g.config.GroupBy.
    // Hint: Use strings.Builder for efficient concatenation.
}
```

```go
// internal/grouper/group.go

package grouper

import (
    "github.com/yourorg/alerting-system/internal/alert"
)

// Group holds a set of alerts that share the same grouping labels.
type Group struct {
    key    string
    alerts []*alert.Alert
    timers map[string]*timerManager
    // Add other fields as needed (e.g., lastNotificationTime).
}

// UpsertAlert adds a new alert or updates an existing one in the group.
func (g *Group) UpsertAlert(a *alert.Alert) {
    // TODO 1: Find if an alert with the same fingerprint already exists.
    // TODO 2: If exists, replace it with the new alert (update).
    // TODO 3: If not, append to g.alerts.
}

// RemoveAlert removes an alert by fingerprint.
func (g *Group) RemoveAlert(fingerprint string) bool {
    // TODO: Find the alert by fingerprint and remove it from g.alerts slice.
    // Return true if removed, false if not found.
}

// HasActiveAlerts returns true if any alert in the group is pending or firing.
func (g *Group) HasActiveAlerts() bool {
    // TODO: Iterate through g.alerts, call a.IsActive().
}
```

E. Language-Specific Hints

- **Map Thread Safety**: Use `sync.RWMutex` for protecting the `groups` map. For read-heavy operations (like calculating group keys), an `RWMutex` allows multiple concurrent readers.
- **Slice Management**: When removing an element from the middle of a slice (`Group.alerts`), use the trick `slice[i] = slice[len(slice)-1]; slice = slice[:len(slice)-1]` if order doesn't matter, for O(1) removal. If order matters (e.g., for deterministic notification output), use `slice = append(slice[:i], slice[i+1:]...)`.
- **Timer Channels**: Always drain a timer's channel after stopping it (`if !timer.Stop() { <-timer.C }`) to prevent a fired timer from executing its callback later unexpectedly.
- **String Building**: For `calculateGroupKey`, use `strings.Builder` for efficient string concatenation, especially when the number of grouping labels or length of values is large.

F. Milestone Checkpoint

After implementing the Grouper, you can verify its behavior with a simple test:

1. **Create a test configuration**: `GroupBy = ["alertname", "env"]`, `GroupWait = 2s`, `GroupInterval = 5s`.

2. **Simulate alert flow**: Write a test that calls `g.Process` with alerts having labels `{alertname="HighCPU", env="prod", instance="host1"}` and `{alertname="HighCPU", env="prod", instance="host2"}`.
3. **Expected behavior**:
   - Both alerts should land in the same group (key `"HighCPU|prod"`).
   - A notification should be sent after ~2 seconds containing both alerts.
   - If a third alert for the same group arrives 1 second after the first notification, it should not trigger an immediate second notification; it should be batched and sent after the next `group_interval` (approximately 5 seconds after the first notification).
4. **Run the test**: Execute `go test ./internal/grouper/... -v` and observe the timing of flush callbacks.

Signs of problems:

- **Alerts appearing in separate groups**: Check `calculateGroupKey` logic, especially label sorting and handling of missing labels.
- **Notifications sent immediately without waiting**: Verify the `group_wait` timer is being started when the first alert arrives in a new group.
- **Multiple notifications sent in rapid succession**: Ensure the `group_interval` timer is not being reset on each new alert (use the non-resettable strategy from the pitfall mitigation).

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Alerts disappear from notifications after label update.	Orphaned alert: label change moved it to a new group, but old group still holds it.	Log the fingerprint and old/new group key in `Process`. Check `fingerprintToKey` map.	Implement the `moveAlert` logic to transfer alerts between groups.
Memory usage grows indefinitely even when alerts resolve.	Stale groups not cleaned up.	Add metrics to count number of groups and alerts per group. Check if empty groups persist.	Ensure `cleanupGroup` is called when a group becomes empty after flushing.
Critical alert delayed by `group_wait`.	Configuration uses same `group_wait` for all alerts.	Check the route configuration; `group_wait` should be set per-route.	Implement routing tree (Milestone 4) where high-severity routes have `group_wait: 0s`.
Notifications stop entirely after first flush for a group.	`group_interval` timer not restarted after flush.	Add logging to timer start/stop events. Verify the interval timer exists after flush.	In `FlushGroup`, if group still has active alerts after sending, restart the interval timer.

5.3 Component: Silencer & Inhibitor

> **Milestone(s):** This section details the design and implementation of the Silencer and Inhibitor components, corresponding to **Milestone 3: Silencing & Inhibition**. These components are responsible for suppressing alerts—either temporarily during planned maintenance via silences, or automatically when higher-priority alerts fire via inhibition—to reduce alert noise and prevent operator fatigue.

Mental Model: The "Do Not Disturb" Sign and Circuit Breaker

Before diving into technical details, consider two intuitive analogies for how these components manage alert suppression.

The "Do Not Disturb" Sign (Silencing): Imagine a hotel room door with a "Do Not Disturb" sign. Housekeeping staff, despite their normal schedule, will skip cleaning that room while the sign is displayed. The sign has a clear start time (when it's hung) and an end time (when it's removed or expires). Similarly, a **silence** is a declarative rule that says, "For this specific set of alerts (defined by matching labels), do not send any notifications within this time window." It's a proactive, time-bounded suppression for planned activities like maintenance windows, deployments, or testing.

The Circuit Breaker (Inhibition): In an electrical panel, a main circuit breaker can trip and cut power to a set of downstream branch circuits. If the main overloads, there's no need to also alert about the individual lights and outlets losing power—the root cause is known. An **inhibition rule** acts similarly: when a "source" alert (e.g., `severity=critical`, `component=network`) fires, it automatically suppresses any "target" alerts (e.g., `severity=warning`, `component=api`) that match a related set of labels. This prevents a cascade of redundant, lower-severity notifications when a root-cause issue is already known and being addressed.

Together, these mechanisms allow the system to distinguish between **planned suppression** (silences) and **automatic, dependency-based suppression** (inhibition), both critical for reducing noise.

Interface & Algorithm

The Silencer and Inhibitor are two distinct but conceptually similar components. They expose interfaces for creating suppression rules and checking whether a given `Alert` should be suppressed. While they could be combined, keeping them separate simplifies responsibility: the Silencer manages user-created, time-based suppressions, while the Inhibitor manages system-configured, alert-triggered suppressions.

Core Interfaces

The primary interaction for other components (notably the Grouper or Router) is to query whether an alert is suppressed before sending a notification.

Method Name	Parameters	Returns	Description
`Silencer.CheckSuppressed`	`alert *alert.Alert`	`bool`	Returns `true` if the alert matches any active, non-expired silence. An alert is considered suppressed for its entire active (`pending` or `firing`) lifetime if a matching silence exists at the time of the check.
`Inhibitor.CheckSuppressed`	`alert *alert.Alert`	`bool`	Returns `true` if the alert is inhibited by any active source alert. Inhibition is evaluated dynamically each time an alert's state is processed.
`Silencer.AddSilence`	`silence *Silence`	`error`	Adds a new silence to the active set. Validates the silence (e.g., `EndsAt` after `StartsAt` , matchers are valid) and stores it. Starts a background timer to expire it at `EndsAt` .
`Silencer.RemoveSilence`	`id string`	`error`	Removes a silence by its ID, effectively ending suppression early.
`Inhibitor.AddRule`	`rule *InhibitionRule`	`error`	Adds a new inhibition rule to the active set. Validates the rule (e.g., matchers are valid, no self-inhibition loops).
`Inhibitor.RemoveRule`	`id string`	`error`	Removes an inhibition rule by its identifier.

Data Structures

The components operate on the following core data types, defined in the Data Model section but detailed here for completeness.

Type Name	Fields	Description
`Silence`	`ID string` `Matchers []Matcher` `StartsAt time.Time` `EndsAt time.Time` `CreatedBy string` `Comment string`	A time-bounded suppression rule. `Matchers` define which alerts are suppressed. `StartsAt` and `EndsAt` define the active window. The `ID` is a unique identifier (e.g., UUID).
`InhibitionRule`	`SourceMatchers []Matcher` `TargetMatchers []Matcher` `EqualLabels []string`	A rule that suppresses target alerts when a source alert is firing. `SourceMatchers` identify the "cause" alerts. `TargetMatchers` identify the "effect" alerts to suppress. `EqualLabels` is an optional list of label names; if provided, the source and target alerts must have identical values for these labels for inhibition to apply (e.g., to inhibit only alerts from the same `cluster`).
`Matcher`	`Name string` `Value string` `IsRegex bool`	A label matching condition. If `IsRegex` is `false` , it performs an exact string match (`label[Name] == Value`). If `true` , `Value` is a regular expression that must match the label value.

Algorithm: Checking for Suppression

The algorithm for determining if an alert is suppressed involves evaluating all active silences and inhibition rules. This check is performed **after** an alert is grouped and before a notification is sent (or alternatively, before the alert is added to a group to avoid processing overhead). Here is the step-by-step logic:

1. **Silence Check:**
    1. For each active `Silence` (where `currentTime` is between `StartsAt` and `EndsAt`):
    2. Evaluate the alert's `Labels` against the silence's `Matchers` using the `Matcher.Match()` method.
    3. If **all** matchers in the silence match the alert's labels, the alert is suppressed. Return `true` immediately.
    4. If no matching silence is found, proceed to the inhibition check.
2. **Inhibition Check:**
    1. For each active `InhibitionRule` :
    2. First, verify there exists at least one **active** (`State == "firing"`) source alert in the system whose labels match the rule's `SourceMatchers` . This requires the Inhibitor to have access to the current set of firing alerts (typically provided by the Rule Evaluator or a shared state store).
    3. If a matching source alert exists, evaluate the candidate alert's labels against the rule's `TargetMatchers` .
    4. If the alert matches the target matchers, **and** (if `EqualLabels` is specified) the alert and the source alert have identical values for all labels listed in `EqualLabels` , then the alert is inhibited. Return `true` .

5. If no inhibition rule applies, the alert is not suppressed. Return `false`.

> **Design Insight:** The order of checks (Silence first, then Inhibition) is intentional. A silence is a strong, explicit user intent to suppress—it should override any automatic inhibition logic. If an alert is silenced, we skip the inhibition check entirely.

Algorithm: Managing Silence Expiry

Silences have a finite lifetime. To avoid scanning all silences on every check, we use a timer-based expiry mechanism.

1. **On Silence Creation (`AddSilence`):**
    1. Validate the silence. Generate a unique `ID` if not provided.
    2. Store the silence in a map keyed by `ID` and also in a time-ordered index (e.g., a min-heap by `EndsAt`).
    3. If `StartsAt` is in the future, schedule a timer to activate the silence at that time (or alternatively, treat it as inactive until `currentTime >= StartsAt`).
    4. Schedule a timer to expire the silence at `EndsAt` . When the timer fires, remove the silence from the active maps.
2. **Periodic Cleanup:**
    1. As a safety net, run a background goroutine every minute to scan for silences where `EndsAt` is in the past (missed timer events) and remove them.
    2. This also cleans up any silences with `EndsAt` set to a very distant future (effectively permanent) if they are manually deleted.

ADR: Label Matching Engine Design

> **Decision: Use a Flexible Matcher Syntax with Equality and Regex Support**
>
> - **Context:** Both silences and inhibition rules need to specify which alerts they apply to based on labels. We require a expressive yet simple syntax for users to define matches. The system must match alerts efficiently, as checks happen frequently in the hot path.
> - **Options Considered:**
>   1. **Exact Equality Only:** Matchers are simple `key=value` pairs. The alert must have the exact label.
>   2. **Equality + Regex:** Matchers support `key=value` (exact) and `key=~<regex>` (regular expression match).
>   3. **Full PromQL-like Matchers:** Support equality, regex, negation (`key!=value` , `key!~<regex>`), and set membership (`key in (value1, value2)`).
> - **Decision:** We chose **Option 2: Equality + Regex**. This provides the essential flexibility needed for most use cases (e.g., `cluster=~prod-.*` , `severity=critical`) without the complexity of implementing and maintaining a full matcher grammar.
> - **Rationale:**
>   - **Pragmatic Flexibility:** Regex matching covers a wide array of practical grouping scenarios (all production clusters, any service starting with "api-"). Exact equality handles the majority of precise matches.
>   - **Implementation Simplicity:** The parsing logic for `key=value` and `key=~regex` is straightforward. Evaluating a regex is more expensive than string equality, but the number of active silences/rules is typically low (< 1000), and we can optimize by pre-compiling regexes.
>   - **Alignment with Precedent:** This is the model used by Prometheus Alertmanager and is well-understood by the community. It strikes a balance between power and complexity.
> - **Consequences:**
>   - **Positive:** Users have a familiar, sufficiently powerful matching language. Implementation is manageable.
>   - **Negative:** Does not support negated matchers or set operations directly. Users must work around this with regex (e.g., `key=~((?!value).)*` for negation is complex and error-prone). If needed, these could be added in a future extension.

Option	Pros	Cons	Chosen?
Exact Equality Only	Very fast evaluation, simple parsing and storage.	Inflexible; cannot match groups of alerts with regex patterns (e.g., all `prod-*` clusters).	No
Equality + Regex	Good balance of power and complexity. Covers most real-world use cases. Regex can be pre-compiled for performance.	Regex evaluation is slower than exact match. No native negation or set operations.	**Yes**
Full PromQL-like Matchers	Maximum expressiveness. Familiar to Prometheus users.	Complex parser and evaluation engine. Higher performance overhead for negation and set checks. Risk of scope creep.	No

Common Pitfalls & Mitigations

⚠️ Pitfall: Inhibition Loops (A inhibits B, B inhibits A)

- **Description:** Two inhibition rules are configured such that alert A inhibits alert B, and alert B also inhibits alert A. When both are firing, they could theoretically suppress each other, leading to indeterminate behavior or both being incorrectly suppressed.

- **Why it's wrong:** This creates a logical deadlock and violates the intended hierarchical "source suppresses target" model. It can cause critical alerts to be unexpectedly silenced.
- **How to fix:** During rule addition (`Inhibitor.AddRule`), perform a cycle detection check. Treat rules as edges in a directed graph (source -> target). Use a simple graph traversal (e.g., depth-first search) to ensure adding a new rule does not create a cycle. If a cycle is detected, reject the rule with an error.

⚠️ Pitfall: Overly Broad or Incorrect Matchers

- **Description:** A silence with matchers `severity=~"warning|critical"` might be intended to suppress all high-severity alerts, but if the `severity` label is missing on some alerts, the regex match will fail (no label != label with empty value). Conversely, a typo like `cluster=prod` instead of `cluster=production` will silently fail to match intended alerts.
- **Why it's wrong:** Suppression fails silently, leading to unexpected notifications during maintenance. Operators lose trust in the system.
- **How to fix:**
  1. Provide a "preview" API that returns which currently firing alerts would be matched by a proposed silence.
  2. Log a warning when a new silence is created that matches zero currently firing alerts (potential typo).
  3. Clearly document matcher semantics: a matcher requires the label to **exist** and match the value. Use `.*` regex to match any value if the label must be present.

⚠️ Pitfall: Race Condition Between Silence Creation and Alert Evaluation

- **Description:** An alert fires at time T. A silence that would match it is created at time T+1ms but with a `StartsAt` time of T-1h (backdated). Should the alert be suppressed? The system might have already sent a notification before the silence was processed.
- **Why it's wrong:** Inconsistent state: the alert may be "stuck" in a firing, unsuppressed state even though a matching backdated silence exists, leading to duplicate notifications if the alert re-fires.
- **How to fix:** Define a clear, consistent policy. We recommend that silences only affect alerts that become active **after** the silence is **processed by the system**. When a silence is added, evaluate all **currently active** alerts against it. If they match, update their suppressed status immediately. This ensures atomic state transitions. Do not apply backdated silences to alerts that have already resolved.

⚠️ Pitfall: Forgetting to Clean Up Expired Silences

- **Description:** Expired silences remain in memory because the timer-based cleanup failed (e.g., a panic) or was not implemented. The `CheckSuppressed` function continues to iterate over thousands of expired entries, degrading performance.
- **Why it's wrong:** Memory leak and performance degradation over time.
- **How to fix:** Implement the dual cleanup strategy: **timer-driven** for efficiency (immediate removal at `EndsAt`) and **periodic sweeping** (e.g., every 5 minutes) as a safety net to remove any expired silences the timers missed. Always use the periodic sweep in addition to timers.

Implementation Guidance

This subsection provides concrete code guidance for building the Silencer and Inhibitor components in Go.

A. Technology Recommendations

Component	Simple Option	Advanced Option
Matcher Parsing	Custom parser for `key=value` and `key=~regex` strings using `strings.Split` and `regexp.Compile`.	Use a parsing library (e.g., `github.com/prometheus/prometheus/pkg/labels`) for full PromQL matcher compatibility.
Storage (Active Silences)	In-memory map + slice sorted by `EndsAt` . Simple and fast for single-instance deployments.	Persistent storage with a database (SQLite, PostgreSQL) for silences to survive restarts; required for high-availability setups.
Time Management	`time.Timer` for each silence's expiry. Suitable for a moderate number of silences (< 10k).	A single priority queue (min-heap) of silences keyed by `EndsAt` , polled by a single goroutine using `time.Tick` . More scalable for many silences.
Concurrent Access	`sync.RWMutex` protecting the in-memory maps.	Partitioned locks (sharding) by silence ID hash for higher concurrency.

For the initial implementation, we recommend the **Simple Option** for each category to focus on the core logic.

B. Recommended File/Module Structure

Extend the project structure established in previous milestones:

```
alerting-system/
├── cmd/
│   └── alertmanager/          # Main application entry point
├── internal/
│   ├── alert/                 # Core data types (Alert, Rule, etc.)
│   ├── metricsclient/         # Client for querying metrics (from Milestone 1)
│   ├── evaluator/             # Rule Evaluator (Milestone 1)
│   ├── grouper/               # Grouper (Milestone 2)
│   ├── silencer/              # This component: Silencer
│   │   ├── silencer.go        # Main Silencer type and logic
│   │   ├── matcher.go         # Matcher parsing and evaluation
│   │   └── silencer_test.go
│   ├── inhibitor/             # This component: Inhibitor
│   │   ├── inhibitor.go       # Main Inhibitor type and logic
│   │   └── inhibitor_test.go
│   └── router/                # Notification Router (Milestone 4)
└── pkg/
    └── api/                   # HTTP API for creating silences, etc.
```

C. Infrastructure Starter Code

1. Matcher Parsing and Evaluation: This is a prerequisite. Provide a complete, reusable `Matcher` implementation.

```go
// internal/silencer/matcher.go

package silencer


import (

    "regexp"

    "strings"

)


// Matcher defines a label matching rule.

type Matcher struct {

    Name    string

    Value   string

    IsRegex bool

    regex   *regexp.Regexp // compiled regex, used if IsRegex is true

}


// ParseMatcher parses a matcher string in the form "key=value" or "key=~regex".

// Returns an error if the syntax is invalid or the regex fails to compile.

func ParseMatcher(s string) (*Matcher, error) {

    // Split on the first '=', '!=', '=~', or '!~'.

    // For simplicity, we handle only '=' and '=~' per our ADR.

    var op string

    if idx := strings.Index(s, "=~"); idx != -1 {

        op = "=~"

    } else if idx := strings.Index(s, "="); idx != -1 {

        op = "="

    } else {

        return nil, fmt.Errorf("invalid matcher format: must contain '=' or '=~'")

    }



    parts := strings.SplitN(s, op, 2)

    if len(parts) != 2 {

        return nil, fmt.Errorf("invalid matcher format")

    }

    name := parts[0]

    value := parts[1]



    m := &Matcher{

        Name:    name,

        Value:   value,

        IsRegex: op == "=~",

    }
```

```
    if m.IsRegex {
        re, err := regexp.Compile(value)
        if err != nil {
            return nil, fmt.Errorf("invalid regex %q: %v", value, err)
        }
        m.regex = re
    }
    return m, nil
}

// Match returns true if the given label set satisfies the matcher.
func (m *Matcher) Match(labels map[string]string) bool {
    v, ok := labels[m.Name]
    if !ok {
        // Label is missing -> cannot match.
        return false
    }
    if m.IsRegex {
        return m.regex.MatchString(v)
    }
    return v == m.Value
}

// String returns a human-readable representation of the matcher.
func (m *Matcher) String() string {
    if m.IsRegex {
        return m.Name + "=~" + m.Value
    }
    return m.Name + "=" + m.Value
}
```

2. Timer Manager for Silence Expiry: Reuse the generic `timerManager` defined in the Grouper component (Milestone 2) to handle scheduling silence expiry.

GO

```go
// internal/silencer/timer_manager.go (or reuse from internal/grouper)
// This is a copy of the timerManager from Milestone 2, provided for completeness.
package silencer

import (
    "sync"
    "time"
)

type timerManager struct {
    timer    *time.Timer
    duration time.Duration
    callback func()
    mu       sync.Mutex
    stopped  bool
}

func newTimerManager(duration time.Duration, callback func()) *timerManager {
    tm := &timerManager{
        duration: duration,
        callback: callback,
    }
    tm.startOrReset()
    return tm
}

func (tm *timerManager) startOrReset() {
    tm.mu.Lock()
    defer tm.mu.Unlock()
    if tm.stopped {
        return
    }
    if tm.timer != nil {
        tm.timer.Stop()
    }
    tm.timer = time.AfterFunc(tm.duration, tm.callback)
}

func (tm *timerManager) stop() {
    tm.mu.Lock()
    defer tm.mu.Unlock()
    tm.stopped = true
    if tm.timer != nil {
```

```
        tm.timer.Stop()
        tm.timer = nil
    }
}

func (tm *timerManager) isStopped() bool {
    tm.mu.Lock()
    defer tm.mu.Unlock()
    return tm.stopped
}
```

D. Core Logic Skeleton Code

1. Silencer Core Logic: Implement the main `Silencer` type and its `CheckSuppressed` method.

```go
// internal/silencer/silencer.go

package silencer


import (

    "sync"

    "time"

    "github.com/your-org/alerting-system/internal/alert"

)


// Silencer holds active silences and provides methods to check suppression.

type Silencer struct {

    mu          sync.RWMutex

    silences    map[string]*Silence            // ID -> Silence

    byExpiry    []*Silence                     // sorted slice by EndsAt, for cleanup

    // A map for faster lookup? Could index by common label keys, but for simplicity we scan.

    // For many silences, consider a more sophisticated index.

}


// NewSilencer creates a new empty Silencer.

func NewSilencer() *Silencer {

    return &Silencer{

        silences: make(map[string]*Silence),

        byExpiry: make([]*Silence, 0),

    }

}


// CheckSuppressed returns true if the given alert matches any active silence.

// An active silence is one where current time is within [StartsAt, EndsAt].

func (s *Silencer) CheckSuppressed(a *alert.Alert) bool {

    s.mu.RLock()

    defer s.mu.RUnlock()


    now := time.Now()

    // TODO 1: Iterate over all silences in s.silences.

    // TODO 2: For each silence, check if now is within [silence.StartsAt, silence.EndsAt].

    // TODO 3: If yes, evaluate all matchers in silence.Matchers against a.Labels using Matcher.Match().

    // TODO 4: If all matchers match, return true (suppressed).

    // TODO 5: If no matching silence found, return false.

    return false

}


// AddSilence adds a new silence and schedules its expiry.

func (s *Silencer) AddSilence(silence *Silence) error {
```

```
    // TODO 1: Validate silence: EndsAt after StartsAt, matchers are valid.
    // TODO 2: Generate an ID (e.g., UUID) if silence.ID is empty.
    // TODO 3: Acquire write lock (s.mu.Lock()).
    // TODO 4: Store silence in s.silences map and insert into s.byExpiry maintaining sorted order.
    // TODO 5: Schedule a timerManager for expiry at silence.EndsAt. Callback should call s.expireSilence(silence.ID).
    // TODO 6: If silence.StartsAt is in the future, optionally schedule activation (or handle in CheckSuppressed).
    // TODO 7: (Optional) Evaluate currently active alerts against this new silence and update their state if matched.
    return nil
}

// expireSilence is called by the timer callback to remove an expired silence.
func (s *Silencer) expireSilence(id string) {
    s.mu.Lock()
    defer s.mu.Unlock()
    // TODO: Remove silence from s.silences and s.byExpiry.
}

// periodicCleanup is a safety net to remove expired silences missed by timers.
func (s *Silencer) periodicCleanup(interval time.Duration) {
    ticker := time.NewTicker(interval)
    defer ticker.Stop()
    for now := range ticker.C {
        s.mu.Lock()
        // TODO: Remove any silences from s.byExpiry where silence.EndsAt < now.
        s.mu.Unlock()
    }
}
```

2. Inhibitor Core Logic: The Inhibitor needs access to the current set of firing alerts. We assume a simple method to get them (e.g., from the Evaluator). We'll define an interface for this dependency.

```go
// internal/inhibitor/inhibitor.go

package inhibitor


import (

    "sync"

    "github.com/your-org/alerting-system/internal/alert"

    "github.com/your-org/alerting-system/internal/silencer" // for Matcher

)


// FiringAlertsGetter is an interface for components that can provide the current set of firing alerts.

// The Evaluator can implement this.

type FiringAlertsGetter interface {

    GetFiringAlerts() []*alert.Alert

}


// InhibitionRule defines a single inhibition rule.

type InhibitionRule struct {

    ID              string

    SourceMatchers  []silencer.Matcher

    TargetMatchers  []silencer.Matcher

    EqualLabels     []string

}


// Inhibitor checks if an alert should be inhibited based on active rules and firing alerts.

type Inhibitor struct {

    mu            sync.RWMutex

    rules         map[string]*InhibitionRule

    alertsGetter  FiringAlertsGetter

}


// NewInhibitor creates a new Inhibitor.

func NewInhibitor(getter FiringAlertsGetter) *Inhibitor {

    return &Inhibitor{

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

        alertsGetter: getter,

    }

}


// CheckSuppressed returns true if the alert is inhibited by any active rule.

func (i *Inhibitor) CheckSuppressed(a *alert.Alert) bool {

    i.mu.RLock()

    defer i.mu.RUnlock()
```

```
    firingAlerts := i.alertsGetter.GetFiringAlerts()

    // TODO 1: Iterate over all rules in i.rules.

    // TODO 2: For each rule, check if there exists at least one alert in firingAlerts that matches rule.SourceMatchers.

    //   - Use a helper function matchAll(matchers, labels) that returns true if all matchers match.

    // TODO 3: If a matching source alert exists, then check if the input alert 'a' matches rule.TargetMatchers.

    // TODO 4: If target matches, check EqualLabels: for each label name in rule.EqualLabels, ensure a.Labels[label] equals the source alert's label value.

    // TODO 5: If all conditions satisfied, return true (inhibited).

    // TODO 6: If no rule inhibits, return false.

    return false

}


// AddRule adds a new inhibition rule, performing cycle detection.

func (i *Inhibitor) AddRule(rule *InhibitionRule) error {

    // TODO 1: Validate rule: matchers are valid, source and target matchers are not empty.

    // TODO 2: Check for self-inhibition (source and target matchers are identical). Reject if true.

    // TODO 3: Perform cycle detection: build a directed graph of existing rules and the new rule, ensure no cycles.

    // TODO 4: Acquire write lock and store rule in i.rules.

    return nil

}


// Helper function to match a set of matchers against labels.

func matchAll(matchers []silencer.Matcher, labels map[string]string) bool {

    for _, m := range matchers {

        if !m.Match(labels) {

            return false

        }

    }

    return true

}
```

E. Language-Specific Hints

- **Time Management:** Use `time.Now().UTC()` for consistency when comparing with `StartsAt` and `EndsAt`. Store all times in UTC in the data structures.
- **Regular Expressions:** Pre-compile regexes in the `Matcher` constructor (`regexp.Compile`) and store the compiled `*regexp.Regexp` . This avoids re-compiling on every match check.
- **Concurrency:** Use `sync.RWMutex` for the main maps. The `CheckSuppressed` method uses `RLock()` for concurrent reads, while `AddSilence` and `expireSilence` use `Lock()` .
- **Sorting Silences by Expiry:** Maintain a slice `byExpiry` sorted by `EndsAt` . Use `sort.Slice` to insert new silences in order. The periodic cleanup can then efficiently trim expired silences from the beginning of the slice.
- **Cycle Detection:** For inhibition rules, implement a simple depth-first search (DFS) on the rule graph. Each rule is a node; an edge exists from rule A to rule B if A's target matchers could match an alert that also matches B's source matchers (this is a conservative approximation). For simplicity, you can start by checking for direct cycles (rule source matches its own target) and expand later.

F. Milestone Checkpoint

After implementing the Silencer and Inhibitor, you should be able to:

1. **Create a Silence via API:** Start the system and use a `curl` command to create a silence.

```bash
curl -X POST http://localhost:9093/api/v1/silences \
  -H 'Content-Type: application/json' \
  -d '{
    "matchers": [{"name": "severity", "value": "warning", "isRegex": false}],
    "startsAt": "2023-10-01T10:00:00Z",
    "endsAt": "2023-10-01T12:00:00Z",
    "createdBy": "test",
    "comment": "Silence all warnings for maintenance"
  }'
```

`Expected response: `{"silenceId":"<generated-id>"}`.`

2. **Verify Suppression:** Trigger a warning alert (via the Rule Evaluator) that matches the silence. Observe that no notification is sent (check logs of the Router). The alert should still appear in the internal state as `firing` but be marked as suppressed.
3. **Test Inhibition:** Configure an inhibition rule (e.g., via a config file) where a `critical` alert inhibits `warning` alerts with the same `service` label. Fire a critical alert for `service=api`. Then fire a warning alert for the same service. The warning alert should not generate a notification.
4. **Run Unit Tests:**

```bash
go test ./internal/silencer/... -v
go test ./internal/inhibitor/... -v
```

`Tests should cover matcher parsing, silence lifecycle (active/expired), and inhibition logic with and without `EqualLabels`.`

G. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Alert is not being suppressed by a silence that should match.	1. Silence is not active (wrong time window). 2. Matcher syntax error (typo). 3. Label name or value case mismatch.	1. Check silence `StartsAt` / `EndsAt` vs current time in logs. 2. Use the API to fetch the silence and verify matchers. 3. Compare exact label strings from the alert and silence.	1. Adjust silence timing. 2. Correct the matcher string. 3. Ensure label consistency (use consistent case).
All alerts are unexpectedly suppressed.	A silence with overly broad matchers (e.g., `severity=~".*"`) is active.	List all active silences via API. Look for matchers that match everything.	Remove or adjust the overly broad silence.
Inhibition is not working; both source and target alerts generate notifications.	1. Source alert is not in `firing` state. 2. `EqualLabels` condition not met (labels differ). 3. Inhibition rule not loaded.	1. Verify source alert state. 2. Check label values for the specified `EqualLabels`. 3. Check logs for rule parsing errors on startup.	1. Ensure source alert condition holds for its `for_duration`. 2. Adjust labels or rule configuration. 3. Fix rule configuration file.
Memory usage grows over time and never decreases.	Expired silences are not being cleaned up.	Check logs for errors in the timer callback or periodic cleanup. Add metrics to track number of active vs expired silences.	Ensure `expireSilence` is called and correctly removes entries. Verify the periodic cleanup goroutine is running.

5.4 Component: Notification Router

> **Milestone(s):** This section details the design and implementation of the Notification Router component, corresponding to **Milestone 4: Notification Routing**. The Router is the system's final processing stage that directs alerts to appropriate human operators via configured channels like Slack, PagerDuty, or email. It transforms the internal `Alert` objects into external notifications, applies routing rules based on labels, manages rate limits, and handles retries for failed deliveries.

Mental Model: The Airport Luggage Router

Imagine a major international airport's baggage handling system. Thousands of suitcases arrive from incoming flights every hour. Each bag has a destination tag attached (a barcode or RFID) containing metadata: the passenger's final destination city, flight number, and sometimes special handling instructions (like "fragile" or "priority"). The luggage routing system scans each bag's tag and makes a critical decision: which conveyor belt (carousel) should this bag be sent to? Bags headed for New York go to Carousel A, bags for London to Carousel B, and so on. Some bags with "priority" tags might be routed to a special fast-track carousel for first-class passengers. Crucially, the system doesn't send each bag individually—it waits a few minutes to collect all bags going to the same destination, then releases them together onto the appropriate carousel. It also has safety limits: if a particular carousel gets jammed, the system temporarily stops sending more bags there to prevent a pileup.

In our alerting system, the **Notification Router** serves this exact purpose. Each `Alert` arriving from the Grouper is like a suitcase, with its **labels** acting as the destination tag. The Router examines these labels to determine which "conveyor belt" (notification channel) the alert should be sent to. The **routing rules** are like the airport's routing configuration: "all bags with `destination=NYC` go to Carousel A" becomes "all alerts with `team=platform` and `severity=critical` go to the PagerDuty receiver". The **grouping** logic from the previous component provides the "wait and batch" behavior—alerts destined for the same channel are held briefly and sent as a consolidated notification. **Rate limiting** prevents "jams" by ensuring we don't overwhelm any single channel with too many notifications too quickly. Finally, the **receivers** (Slack, PagerDuty, email) are the actual conveyor belts that deliver the bags (notifications) to their final recipients.

Interface & Algorithm

The Router's primary responsibility is to accept a `Notification` (which contains a group of alerts) and deliver it to the appropriate external systems based on a configurable routing tree. It must support multiple receiver types, handle retries for transient failures, and enforce rate limits per receiver.

Core Data Structures

The Router operates on several key data structures defined in the system's data model:

Data Structure	Fields	Description
`Route`	`Receiver string`	Name of the receiver to send matching alerts to
	`Matchers []Matcher`	List of label matchers that must all match for the route to be selected
	`Continue bool`	If true, continue evaluating child routes after this match
	`Routes []*Route`	Child routes for hierarchical routing
	`GroupWait time.Duration`	Override for initial delay before first notification (inherits from parent if not set)
	`GroupInterval time.Duration`	Override for minimum time between notifications (inherits from parent if not set)
`Receiver`	`Name string`	Unique identifier for this receiver
	`Type string`	One of: `"slack"`, `"pagerduty"`, `"email"`, `"webhook"`
	`Config json.RawMessage`	Receiver-specific configuration (API tokens, URLs, templates)
`Notification`	`GroupKey string`	The grouping key that identifies which group these alerts belong to
	`Alerts []*Alert`	The list of alerts to be sent in this notification batch
	`Receiver string`	Name of the receiver this notification should be sent to

Router Interface

The Router exposes a minimal interface for processing notifications. The primary method `Route` is called by the Grouper when it's time to send a notification batch.

Method	Parameters	Returns	Description
`Router.Route(notification Notification)`	`notification Notification`	(none)	Main entry point. Walks the routing tree to find matching receivers and dispatches the notification. This method is synchronous for simplicity; actual sending happens in goroutines.
`Router.Send(ctx context.Context, receiver Receiver, notification Notification)`	`ctx context.Context`, `receiver Receiver`, `notification Notification`	`error`	Formats and sends a notification to a specific receiver. Implements retry logic, rate limiting, and error handling for each receiver type.

Routing Algorithm

The routing process follows a deterministic tree-walk algorithm. The routing configuration is structured as a tree where each node is a `Route`. The root route contains the default configuration (like default `GroupWait` and `GroupInterval`) and potentially some top-level matchers. When a `Notification` arrives, the Router performs a depth-first walk of this tree:

1. **Start at the root route** with the notification's alert group.
2. **Check matchers**: For the current route node, evaluate whether all `Matchers` in its `Matchers` slice match the **common labels** of the notification's alerts. Common labels are those labels that have the same value across every alert in the group. If any matcher fails, skip this route and all its children.
3. **If matched**:
   - **Send to receiver**: If the route has a `Receiver` set (not empty), call `Router.Send` with the receiver configuration. This triggers the actual notification delivery.
   - **Check continue flag**: If `Continue` is `false`, stop processing for this notification—no child routes are evaluated. This is how you create exclusive routes. If `Continue` is `true`, proceed to evaluate child routes.
4. **Process children**: Recursively apply steps 2-3 to each child route in `Routes`.
5. **Default route**: The root route should have a `Receiver` set and `Continue` set to `false` as a catch-all default. If no routes match during the walk, the notification is sent to this default receiver.

> **Design Insight:** Using common labels for matching (rather than individual alert labels) ensures that all alerts in a notification group are routed consistently. If one alert in a group had `team=platform` and another had `team=database`, the common label `team` wouldn't exist for that group, so a route matching on `team=platform` wouldn't match the group.

Rate Limiting Implementation

Each receiver has its own rate limiter to prevent notification storms. The rate limiter uses a token bucket algorithm:

- **Bucket capacity**: Maximum number of notifications that can be sent in a burst (e.g., 10).
- **Refill rate**: Notifications per second (e.g., 1 notification per 30 seconds for low-volume channels).
- **Per-receiver buckets**: Each receiver type (Slack, PagerDuty, etc.) gets its own limiter, identified by receiver name.

When `Router.Send` is called:

1. Check the rate limiter for the specific receiver.
2. If tokens are available, consume one and proceed immediately.
3. If no tokens are available, wait until the next refill (with a maximum wait timeout, after which the notification is dropped with a warning log).
4. Failed sends (due to network errors) **do not** refund tokens—this prevents retry loops from bypassing rate limits.

Retry Logic

Network failures are inevitable. The Router implements exponential backoff with jitter for retries:

1. **Initial attempt** with no delay.
2. **First retry** after 1 second ± random jitter (up to 100ms).
3. **Second retry** after 2 seconds ± jitter.
4. **Third retry** after 4 seconds ± jitter.
5. **Maximum 3 retries** before marking the notification as permanently failed.

Permanently failed notifications are logged as errors but not retried further. A monitoring metric tracks failed notification counts per receiver.

ADR: Routing Tree vs. Linear Rule List

> **Decision: Tree-Based Routing Configuration**
>
> **Context**: We need a flexible way to route alerts to different notification channels based on their labels. The configuration must support hierarchical matching (e.g., "all critical alerts go to PagerDuty, but critical database alerts additionally get an email to the DBA team"), allow for exclusive routing ("only page for critical alerts, never email"), and be intuitive to configure. The system will have dozens to hundreds of routing rules in production.
>
> **Options Considered**:
>
> 1. **Linear List of Rules**: A flat list of `(matchers, receiver)` pairs evaluated in order. First match wins.
> 2. **Tree-Based Routing**: A nested tree structure where each node can have matchers, a receiver, and child nodes. Evaluation follows depth-first traversal with a `continue` flag controlling whether to check child nodes after a match.
> 3. **Tag-Based Subscription**: Each receiver "subscribes" to tags/labels it's interested in. All matching receivers get the notification.
>
> **Decision**: We implement **Tree-Based Routing**, modeled after Prometheus Alertmanager's approach.
>
> **Rationale**:
>
> - **Hierarchical organization**: Real-world routing logic often follows a tree structure (e.g., "all alerts → team-specific alerts → service-specific alerts"). A tree naturally represents this hierarchy.
> - **Explicit control via `continue`**: The `continue` flag provides clear control over whether child routes should be evaluated, preventing unexpected duplicate notifications.
> - **Inheritance of timing parameters**: Child routes can inherit `GroupWait` and `GroupInterval` from parent routes, reducing configuration duplication.
> - **Proven pattern**: This is the same model used by Prometheus Alertmanager, which has been battle-tested in thousands of production deployments. Developers familiar with Alertmanager will immediately understand our configuration format.
> - **Performance**: For typical configurations with 10-100 routes, tree traversal is O(n) same as linear list, but with better logical grouping.
>
> **Consequences**:
>
> - **Configuration complexity**: Users must understand tree traversal semantics, particularly the `continue` flag.
> - **Implementation complexity**: We need to implement recursive tree walking and proper inheritance of timing parameters.
> - **Powerful expressiveness**: Can represent complex routing policies that would require awkward workarounds in a linear list.

Option	Pros	Cons	Chosen?
Linear List of Rules	Simple to implement and understand. Easy to debug (just follow the list).	No natural way to express hierarchies. Difficult to manage timing parameter inheritance. Can lead to rule ordering confusion.	No
Tree-Based Routing	Natural hierarchical organization. Explicit control with `continue` flag. Inherits timing parameters. Industry-proven pattern.	More complex to implement. Configuration requires understanding tree traversal.	**Yes**
Tag-Based Subscription	Decouples senders and receivers. Easy to add new receivers without modifying routing logic.	Can cause notification explosion (one alert to many receivers). No built-in exclusivity. Difficult to manage rate limits per logical route.	No

Common Pitfalls & Mitigations

⚠️ Pitfall: Missing Default Route

- **Description**: Forgetting to configure a default receiver at the root route, or setting `Continue=true` on a route that should be the catch-all. This results in notifications that match no routes being silently dropped.
- **Why it's wrong**: Critical alerts might go unnoticed because they fall through the cracks. This is a silent failure—the system appears to work (alerts are evaluated and grouped) but no human ever sees them.
- **Fix**: Always validate routing configuration on startup. Ensure the root route has a `Receiver` set. Consider requiring a default receiver in configuration validation. Log a warning if a notification passes through the entire tree without being sent to any receiver.

⚠️ Pitfall: Duplicate Notifications from Misconfigured `continue`

- **Description**: Setting `Continue=true` on a route with a receiver, then having child routes that also match the same alerts. This causes the same alert group to be sent to multiple receivers, creating duplicate notifications.

- **Why it's wrong**: Alert fatigue increases as operators receive the same alert through multiple channels (e.g., both Slack and PagerDuty). It wastes resources and confuses incident response.
- **Fix**: Understand the routing tree semantics: when a route matches and has a receiver, if `Continue=false`, processing stops for that branch. Use `Continue=true` only when you intentionally want alerts to go to multiple receivers (e.g., "page the on-call AND post to team Slack"). Document this clearly in configuration examples.

⚠️ **Pitfall: Overly Strict Rate Limiting Blocks Critical Alerts**

- **Description**: Setting rate limits too strictly (e.g., "1 notification per hour") on a critical channel like PagerDuty. During an incident, multiple related alerts might be triggered in quick succession, but the rate limiter delays or drops them.
- **Why it's wrong**: The most important alerts during an outage might be delayed or never delivered due to rate limiting, defeating the purpose of alerting.
- **Fix**: Implement **priority channels** exempt from rate limiting (or with much higher limits). Use separate receivers for critical vs. non-critical alerts. Consider implementing **burst capacity** in the token bucket algorithm to allow short bursts of alerts during incidents. Monitor dropped/delayed notification metrics.

⚠️ **Pitfall: Not Handling Receiver Configuration Changes at Runtime**

- **Description**: The Router loads receiver configuration (API tokens, webhook URLs) once at startup. If credentials expire or URLs change, the system must be restarted to pick up new configuration.
- **Why it's wrong**: In production, restarting the alerting system might cause missed alerts during the restart window. Operators need to be able to update Slack webhooks or PagerDuty integration keys without downtime.
- **Fix**: Implement **hot reloading** of configuration. Watch the configuration file for changes and reload routing rules and receiver configurations safely. Ensure in-flight notifications complete with old configuration before switching.

⚠️ **Pitfall: No Retry Logic for Transient Network Failures**

- **Description**: Implementing `Router.Send` as a single HTTP request that gives up immediately if the remote service is temporarily unavailable or slow to respond.
- **Why it's wrong**: Network glitches, temporary DNS failures, or brief outages of the notification service (Slack, PagerDuty) would cause alerts to be lost permanently.
- **Fix**: Implement the exponential backoff retry logic described earlier. Distinguish between transient failures (network timeouts, 5xx HTTP status codes) and permanent failures (invalid configuration, 4xx errors). For permanent failures, fail fast and log clearly.

Implementation Guidance

This section provides concrete implementation guidance for building the Notification Router in Go.

Technology Recommendations Table

Component	Simple Option	Advanced Option	Recommendation
HTTP Client for Webhooks	Standard library `net/http`	Custom client with connection pooling, timeouts, and metrics	Use `net/http` with properly configured `http.Client` timeouts. Keep it simple.
Rate Limiting	Custom token bucket implementation	Third-party library like `golang.org/x/time/rate`	Use `golang.org/x/time/rate` —it's battle-tested and provides a well-designed token bucket.
Configuration Format	JSON flat file	YAML with schema validation	Use YAML for human-friendliness. Implement struct tags for parsing.
Template Rendering	Standard library `text/template`	Custom template engine with cached compilation	Use `text/template` with caching of compiled templates per receiver.
Retry Logic	Simple loop with `time.Sleep`	Exponential backoff with jitter using `github.com/cenkalti/backoff/v4`	Implement basic exponential backoff yourself for learning; use `backoff` library for production robustness.

Recommended File/Module Structure

```
project-root/
  cmd/
    alerting-server/
      main.go                    # Entry point, loads configuration
  internal/
    alert/                       # Core data types (Alert, Rule, etc.)
      types.go
    router/                      # Notification Router component
      router.go                  # Main Router struct and Route method
      sender.go                  # Send method and receiver implementations
      rate_limiter.go           # Rate limiting wrapper
      config.go                  # Configuration parsing structs
      templates.go               # Template rendering utilities
    grouper/                     # From previous milestone
    silencer/                    # From previous milestone
  configs/
    example-config.yaml          # Example routing configuration
```

Infrastructure Starter Code

Here's complete, working code for common infrastructure pieces that are prerequisites but not the core learning focus:

1. Configuration Parsing Structures (save as `internal/router/config.go`):

```go
package router

import (
    "time"
    "github.com/yourproject/internal/silencer" // Matcher type is defined in silencer package
)

// Route represents a node in the routing tree
type Route struct {
    Receiver      string             `yaml:"receiver,omitempty"`
    Matchers      []silencer.Matcher `yaml:"matchers,omitempty"`
    Continue      bool               `yaml:"continue,omitempty"`
    Routes        []*Route           `yaml:"routes,omitempty"`
    GroupWait     *time.Duration     `yaml:"group_wait,omitempty"`
    GroupInterval *time.Duration     `yaml:"group_interval,omitempty"`
}

// Receiver defines a notification channel
type Receiver struct {
    Name   string          `yaml:"name"`
    Type   string          `yaml:"type"` // slack, pagerduty, email, webhook
    Config json.RawMessage `yaml:"config"`
}

// Config is the top-level router configuration
type Config struct {
    Receivers []Receiver `yaml:"receivers"`
    Route     Route      `yaml:"route"`
}

// LoadConfig reads and parses router configuration from a YAML file
func LoadConfig(path string) (*Config, error) {
    data, err := os.ReadFile(path)
    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 YAML: %w", err)
    }

    // Validate: at least one receiver, root route has receiver
```

```
    if len(config.Receivers) == 0 {
        return nil, errors.New("at least one receiver must be configured")
    }
    if config.Route.Receiver == "" {
        return nil, errors.New("root route must have a receiver")
    }

    return &config, nil
}
```

2. Rate Limiter Wrapper (save as `internal/router/rate_limiter.go`):

```go
package router

import (
    "context"
    "sync"
    "time"
    "golang.org/x/time/rate"
)

// RateLimiter wraps a token bucket limiter with a mutex for safe concurrent access
type RateLimiter struct {
    limiter *rate.Limiter
    mu      sync.RWMutex
}

// NewRateLimiter creates a rate limiter with n events per second and burst size b
func NewRateLimiter(eventsPerSecond float64, burst int) *RateLimiter {
    return &RateLimiter{
        limiter: rate.NewLimiter(rate.Limit(eventsPerSecond), burst),
    }
}

// Wait blocks until the limiter allows an event or context is cancelled
func (rl *RateLimiter) Wait(ctx context.Context) error {
    rl.mu.RLock()
    defer rl.mu.RUnlock()
    return rl.limiter.Wait(ctx)
}

// SetRate updates the rate limit (for hot reloading)
func (rl *RateLimiter) SetRate(eventsPerSecond float64) {
    rl.mu.Lock()
    defer rl.mu.Unlock()
    rl.limiter.SetLimit(rate.Limit(eventsPerSecond))
}
```

3. Template Rendering Utility (save as `internal/router/templates.go`):

```go
package router

import (
    "bytes"
    "html/template"
    "sync"
    "text/template"
    "github.com/yourproject/internal/alert"
)

// TemplateCache caches compiled templates to avoid re-parsing
type TemplateCache struct {
    mu       sync.RWMutex
    templates map[string]*template.Template
}

// NewTemplateCache creates an empty template cache
func NewTemplateCache() *TemplateCache {
    return &TemplateCache{
        templates: make(map[string]*template.Template),
    }
}

// Render executes a template with the given alert data
func (tc *TemplateCache) Render(templateText string, data map[string]interface{}) (string, error) {
    tc.mu.RLock()
    tpl, exists := tc.templates[templateText]
    tc.mu.RUnlock()

    if !exists {
        // Parse and cache the template
        tpl, err := template.New("notification").Parse(templateText)
        if err != nil {
            return "", fmt.Errorf("parsing template: %w", err)
        }

        tc.mu.Lock()
        tc.templates[templateText] = tpl
        tc.mu.Unlock()
    }

    var buf bytes.Buffer
```

```
    if err := tpl.Execute(&buf, data); err != nil {
        return "", fmt.Errorf("executing template: %w", err)
    }

    return buf.String(), nil
}

// BuildNotificationData creates the data structure for template rendering
func BuildNotificationData(alerts []*alert.Alert, groupKey string) map[string]interface{} {
    // Common labels across all alerts in the group
    commonLabels := make(map[string]string)
    if len(alerts) > 0 {
        // Start with all labels from first alert
        for k, v := range alerts[0].Labels {
            commonLabels[k] = v
        }
        // Remove labels that differ in other alerts
        for _, a := range alerts[1:] {
            for k, v := range commonLabels {
                if a.Labels[k] != v {
                    delete(commonLabels, k)
                }
            }
        }
    }

    return map[string]interface{}{
        "GroupKey":      groupKey,
        "Alerts":        alerts,
        "CommonLabels":  commonLabels,
        "CommonAnnotations": getCommonAnnotations(alerts),
        "ExternalURL":   "", // Could be configured globally
    }
}
```

Core Logic Skeleton Code

Now, here's skeleton code for the core Router logic that you need to implement:

1. Main Router Structure and Route Method (save as `internal/router/router.go`):

```go
package router

import (
    "context"
    "fmt"
    "sync"
    "time"
    "github.com/yourproject/internal/alert"
)

// Router directs notifications to receivers based on a routing tree
type Router struct {
    config      *Config
    receivers   map[string]Receiver          // receiver name -> Receiver config
    rateLimiters map[string]*RateLimiter     // receiver name -> rate limiter
    templateCache *TemplateCache
    mu          sync.RWMutex                 // protects config, receivers, rateLimiters
}

// NewRouter creates a new router with the given configuration
func NewRouter(config *Config) (*Router, error) {
    r := &Router{
        config:      config,
        receivers:   make(map[string]Receiver),
        rateLimiters: make(map[string]*RateLimiter),
        templateCache: NewTemplateCache(),
    }

    // Build receiver lookup map
    for _, recv := range config.Receivers {
        r.receivers[recv.Name] = recv
        // Initialize rate limiter: default 1 notification per 30 seconds, burst of 10
        r.rateLimiters[recv.Name] = NewRateLimiter(1.0/30.0, 10)
    }

    return r, nil
}

// Route processes a notification by walking the routing tree
func (r *Router) Route(notification alert.Notification) {
    // TODO 1: Start tree walk from the root route (r.config.Route)
    // TODO 2: For the current route, check if all matchers match the notification's common labels
```

```
    //         (Use matchAll function from silencer package)
    // TODO 3: If matched:
    //   a) If route has a receiver, call r.sendToReceiver with receiver name and notification
    //   b) If route.Continue is false, return (stop processing this notification)
    //   c) If route.Continue is true, continue to child routes
    // TODO 4: Recursively process child routes (if any)
    // TODO 5: If we reach the end without sending (should not happen with proper config), log a warning
}

// sendToReceiver sends a notification to a specific receiver with rate limiting
func (r *Router) sendToReceiver(receiverName string, notification alert.Notification) {
    // TODO 1: Look up the receiver configuration from r.receivers map
    // TODO 2: Apply rate limiting: create a context with timeout (e.g., 10 seconds)
    //         Call rateLimiter.Wait(ctx) for this receiver
    // TODO 3: If rate limit wait times out, log warning and drop notification
    // TODO 4: Otherwise, call r.Send with the receiver config and notification
}

// Send formats and sends a notification to a receiver (implements retry logic)
func (r *Router) Send(ctx context.Context, receiver Receiver, notification alert.Notification) error {
    // TODO 1: Based on receiver.Type, delegate to appropriate sender method:
    //   - sendToSlack, sendToPagerDuty, sendToEmail, sendToWebhook
    // TODO 2: Implement retry logic with exponential backoff:
    //   maxRetries := 3
    //   for attempt := 0; attempt <= maxRetries; attempt++ {
    //     err := sendWithCurrentAttempt()
    //     if err == nil { return nil } // Success
    //     if isPermanentError(err) { return err } // Don't retry
    //     if attempt == maxRetries { return err } // Out of retries
    //     // Calculate backoff: time.Duration(math.Pow(2, float64(attempt))) * time.Second
    //     // Add jitter: ±10% random variation
    //     // Sleep for backoff duration
    //   }
    // TODO 3: For permanent failures or after max retries, return error
}

// Helper: walkRoute recursively walks the routing tree
func (r *Router) walkRoute(route *Route, notification alert.Notification) {
    // TODO: Implement recursive tree walk as described in Route() method steps 2-4
}
```

2. Receiver-Specific Sender Methods (save as `internal/router/sender.go`):

```go
package router

import (
    "context"
    "encoding/json"
    "fmt"
    "net/http"
    "time"
    "github.com/yourproject/internal/alert"
)

// SlackConfig holds Slack webhook configuration
type SlackConfig struct {
    WebhookURL string `json:"webhook_url"`
    Channel    string `json:"channel,omitempty"`
    Username   string `json:"username,omitempty"`
    IconEmoji  string `json:"icon_emoji,omitempty"`
}

// sendToSlack sends a notification to Slack via webhook
func (r *Router) sendToSlack(ctx context.Context, config json.RawMessage, notification alert.Notification) error {
    // TODO 1: Parse config into SlackConfig struct
    // TODO 2: Build Slack message payload using template rendering
    //         Use r.templateCache.Render() with a default Slack template
    // TODO 3: Make HTTP POST to webhook URL with JSON payload
    // TODO 4: Check response status code; 200-299 = success
    // TODO 5: Return error for non-2xx responses or network errors
}

// PagerDutyConfig holds PagerDuty integration configuration
type PagerDutyConfig struct {
    IntegrationKey string `json:"integration_key"`
    Severity       string `json:"severity,omitempty"` // critical, error, warning, info
}

// sendToPagerDuty sends a notification to PagerDuty
func (r *Router) sendToPagerDuty(ctx context.Context, config json.RawMessage, notification alert.Notification) error {
    // TODO 1: Parse config into PagerDutyConfig
    // TODO 2: Build PagerDuty event payload according to their API
    // TODO 3: Determine event action: "trigger" for firing alerts, "resolve" for resolved
    // TODO 4: Make HTTP POST to PagerDuty Events API v2
    // TODO 5: Handle response and return appropriate error
}
```

```
// Helper: makeHTTPRequest is a generic HTTP client with timeout and retry (used by sendToSlack, sendToPagerDuty, sendToWebhook)
func (r *Router) makeHTTPRequest(ctx context.Context, method, url string, body []byte, headers map[string]string) (*http.Response, error)
{
    // TODO: Create HTTP request with context
    // TODO: Set headers (Content-Type: application/json)
    // TODO: Use http.Client with timeout (e.g., 10 seconds)
    // TODO: Send request and return response/error
}
```

Language-Specific Hints

- **HTTP Client Configuration**: Always set reasonable timeouts on your `http.Client`: `Timeout: 30*time.Second` for overall request, and consider setting `Transport` with `IdleConnTimeout` and `MaxIdleConnsPerHost` for connection pooling.
- **JSON Marshaling**: Use `json.Marshal` for creating request bodies. For Slack/PagerDuty payloads, define structs that match their API specifications.
- **Context Propagation**: Pass `context.Context` through all network calls to support cancellation and deadlines. Use `context.WithTimeout` for rate limiting waits.
- **Concurrency Safety**: The Router will be called concurrently from multiple goroutines (as different alert groups flush). Use `sync.RWMutex` for protecting configuration maps that might be hot-reloaded.
- **Error Wrapping**: Use `fmt.Errorf("... %w", err)` to wrap errors with context, making debugging easier.
- **Logging**: Use structured logging with fields (receiver name, notification group key, alert counts) to make logs searchable.

Milestone Checkpoint

After implementing the Notification Router, you should be able to:

1. **Test Configuration Loading**: Create a test YAML configuration with a root route and a Slack receiver. Run a test that loads it:

```bash
go test ./internal/router -run TestLoadConfig
```

Expected: No errors, configuration parsed successfully.

2. **Test Tree Walking**: Create a unit test with a simple routing tree and verify notifications are routed correctly:

```bash
go test ./internal/router -run TestRoute
```

Expected: Notifications with labels matching route matchers are sent to the correct receiver.

3. **Integration Test**: Start the full alerting system with a mock metrics endpoint that triggers an alert. Configure a webhook receiver that logs to a local HTTP server (use `net/http/httptest`). Verify the notification arrives with correct format.

```bash
go run cmd/alerting-server/main.go --config test-config.yaml
```

Expected: When the alert fires, you should see an HTTP POST to your test server containing the notification payload.

4. **Verify Rate Limiting**: Send 15 notifications in rapid succession to a receiver with rate limit 1/30s burst 10. The first 10 should go immediately, the next 5 should be delayed/spaced out.

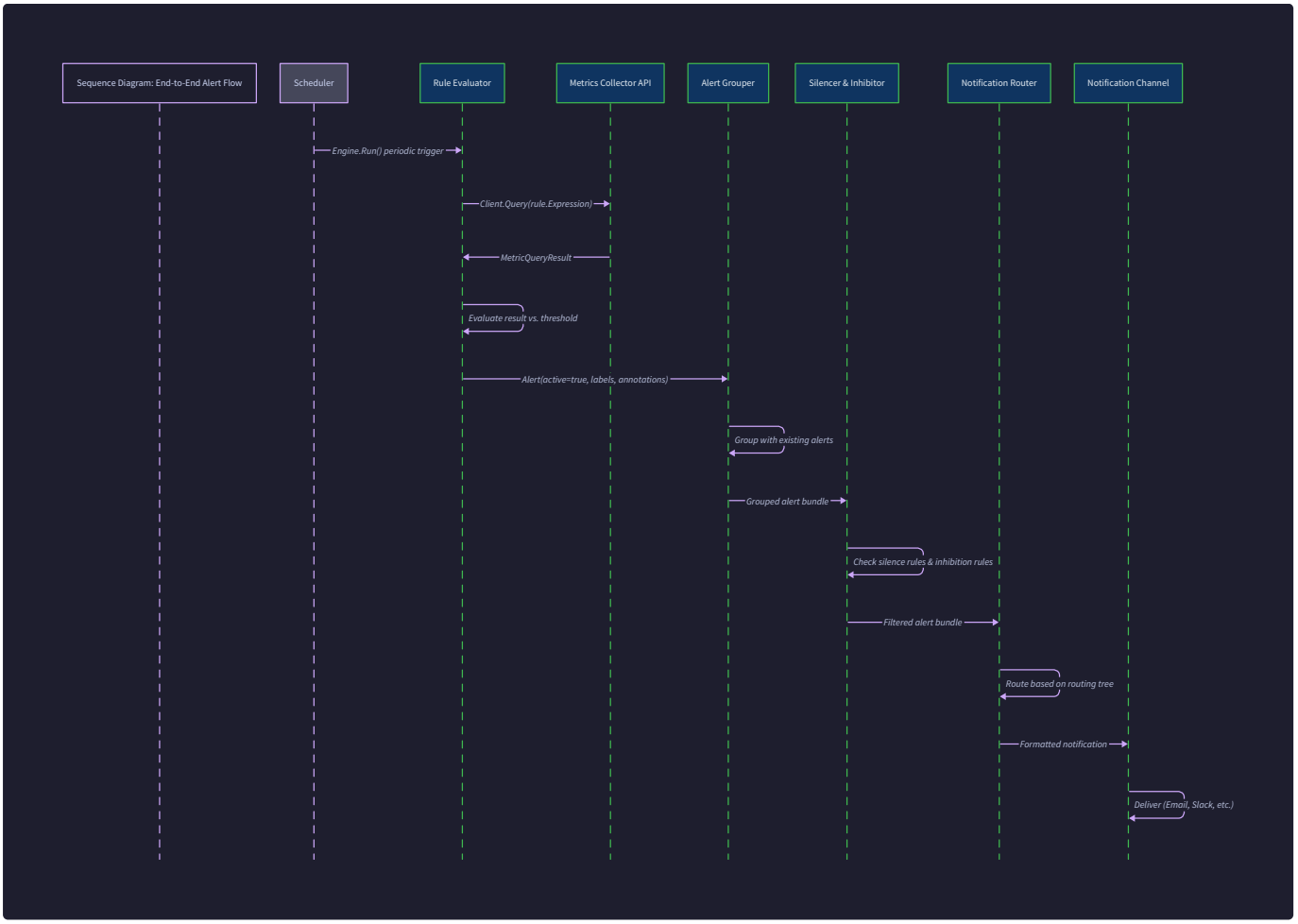
Signs of Problems:

- Notifications not being sent: Check that the Grouper's `flushFunc` is connected to `Router.Route`.
- Duplicate notifications: Verify `continue` flag logic in routing tree.
- "Receiver not found" errors: Ensure receiver names in routes match receiver names in configuration.
- Timeout errors: Check network connectivity and increase HTTP client timeouts if needed.

6. Interactions and Data Flow

> **Milestone(s):** This section describes the end-to-end flow of an alert through the entire system, encompassing all four milestones (Rule Evaluation, Alert Grouping, Silencing & Inhibition, Notification Routing). It explains how components interact and what data structures are exchanged.

The Alerting System transforms raw metric data into actionable human notifications through a carefully orchestrated pipeline. Understanding the **end-to-end flow** is critical for debugging, performance tuning, and ensuring alerts reach their intended destinations reliably. This pipeline operates like an assembly line where each station (component) performs specific transformations, with quality gates (silencing/inhibition) and sorting facilities (grouping/routing) along the way.



Sequence: Alert Lifecycle

Think of the alert lifecycle as a **package's journey through a sophisticated postal system**. A package (alert) originates at a sender (Rule Evaluator), gets sorted into bundles with similar packages (Grouping), passes through customs checks that might block it (Silencing/Inhibition), and finally gets routed to different delivery trucks based on destination tags (Routing). The entire journey must handle packages that change state (pending → firing → resolved) and might need to be recalled.

Step 1: Metric Query and Initial Alert Creation (Rule Evaluator)

The journey begins when the `Engine.Run()` method's periodic evaluation loop triggers. For each configured `Rule`:

1. **Metric Query Execution**: The `Engine` calls `Client.Query()` with the rule's `Expression` (e.g., `http_requests_total{job="api"} > 100`). This sends an HTTP request to the Metrics Collector API and receives a `MetricQueryResult`.
2. **Result Evaluation**: The engine extracts the numeric value from the result and compares it against the rule's `Threshold` using the specified `Operator` (>, <, etc.). If the comparison returns `false`, any previously firing alert for this metric combination transitions to resolved state (if the alert had been firing).
3. **Duration Logic Application**: When the comparison returns `true`:
   - If `ForDuration` is zero, the alert immediately transitions to **firing** state.
   - If `ForDuration` is positive and this is the first time the condition became true, the alert enters **pending** state with `StartsAt` set to current time.
   - If the condition remains true for the entire `ForDuration`, the alert transitions from pending to firing.
   - If the condition becomes false during the pending period, the pending alert is discarded and the `ForDuration` timer resets.
4. **Alert Object Creation**: For each unique metric combination (identified by its label set) that triggers firing, the engine creates an `Alert` struct:
   - **Labels**: Copied from the metric labels, plus rule-specific labels from `Rule.Labels`
   - **Annotations**: Generated by calling `Rule.RenderAnnotations()` with the metric labels and query result value
   - **StartsAt**: Set to the time the condition first became true (for pending) or when it completed `ForDuration` (for firing)
   - **EndsAt**: Initially empty for firing alerts; set to current time when resolved
   - **State**: Set to either `"pending"` or `"firing"`

- **GeneratorURL**: A URL pointing back to the metrics query for investigation

Example Walkthrough: Consider a rule `HighRequestRate` with expression `rate(http_requests_total[5m]) > 100`, `ForDuration: 2m`, and labels `{severity="critical"}`. At evaluation time `T0`, the query returns 95 → no alert. At `T1` (1 minute later), query returns 120 → alert enters pending state with `StartsAt = T1`. At `T2` (another minute later, total 2 minutes), condition still true (value 130) → alert transitions to firing with `StatesAt = T1` (maintaining original detection time). Labels become `{job="api", instance="01", severity="critical"}`.

Step 2: Group Aggregation (Grouper)

Firing and resolved alerts flow from the Rule Evaluator to the Grouper via `Grouper.Process(alert)`. This is where similar alerts are bundled:

1. **Group Key Calculation**: `Grouper.calculateGroupKey(alert)` extracts values for the configured `GroupBy` labels (e.g., `["alertname", "cluster"]`), sorts them, and creates a deterministic hash. If an alert lacks a grouping label, that dimension is treated as empty (all alerts without that label group together).
2. **Group Lookup and Creation**: The grouper checks its `groups` map for the calculated key:
   - If the group doesn't exist, `Grouper.createGroup(key)` initializes a new `Group` with empty alerts list and starts the `group_wait` timer (e.g., 30 seconds).
   - If the alert was previously in a different group (detected via `fingerprintToKey` map), `Grouper.moveAlert()` removes it from the old group and cleans up that group if empty.
3. **Alert Upsertion**: `Group.UpsertAlert(alert)` adds or updates the alert in the group's alerts slice. For a firing alert replacing a resolved one (same fingerprint), the resolved alert is removed. For a resolved alert replacing a firing one, the firing alert is marked resolved but kept in the group (to track resolution).
4. **Timer Management**:
   - **First Notification Delay**: The `group_wait` timer, started when the group is created, ensures the system waits to collect potentially related alerts before sending the first notification. When this timer fires, it calls the `flushFunc` (which routes to Step 4).
   - **Repeat Notification Interval**: After the first notification, a `group_interval` timer starts. When this fires, if there are still active alerts in the group, another notification is sent (updating recipients about ongoing issues).
   - **Timer Reset Logic**: When a new alert arrives in an existing group, the `group_interval` timer is reset if it's running, preventing premature repeated notifications while the situation is still evolving.

Example Walkthrough: Two alerts fire for `HighRequestRate` from different instances but same cluster. Both have labels `{alertname="HighRequestRate", cluster="prod-us", instance="01"}` and `{alertname="HighRequestRate", cluster="prod-us", instance="02"}`. With `GroupBy: ["alertname", "cluster"]`, both share the same group key. The first alert creates the group and starts 30s `group_wait`. The second alert arrives 10s later and joins the same group. After 30s total, the group contains both alerts and triggers its first notification.

Step 3: Suppression Checks (Silencer & Inhibitor)

Before a group can send notifications, each individual alert must pass through suppression gates:

1. **Silence Matching**: For each alert in the group, `Silencer.CheckSuppressed(alert)` evaluates all active `Silence` objects (those where current time is between `StartsAt` and `EndsAt`). Each silence's `Matcher` list is checked against the alert's labels using `matchAll()`. If **any** silence matches, the alert is suppressed.
2. **Inhibition Checking**: Next, `Inhibitor.CheckSuppressed(alert)` evaluates all configured `InhibitionRule` objects:
   - For each rule, check if any **source alerts** (alerts matching `SourceMatchers`) are currently firing (queried via `alertsGetter`).
   - If a firing source exists, verify the `EqualLabels` constraint: the source and target alerts must have identical values for all labels listed in `EqualLabels`.
   - If both conditions hold, the target alert is inhibited.
3. **Group Filtering**: After checking all alerts in a group:
   - If **all** alerts in the group are suppressed (by silence or inhibition), the entire group notification is canceled—no notification is sent.
   - If **some** alerts are suppressed but others aren't, the group proceeds with only the non-suppressed alerts. The suppressed alerts remain in the group (their state is preserved) but don't appear in the notification.
   - If the group becomes empty due to suppression, it's cleaned up via `Grouper.cleanupGroup()`.

Example Walkthrough: A silence is active with matchers `{cluster="prod-us", severity="warning"}`. Our `HighRequestRate` alert has `severity="critical"`, so it doesn't match and passes through. Simultaneously, an inhibition rule exists where source=`{severity="critical"}`, target=`{severity="warning"}`, `EqualLabels: ["cluster"]`. Since our alert has `severity="critical"`, it acts as a source that would inhibit warning alerts in the same cluster, but isn't inhibited itself.

Step 4: Notification Routing and Delivery

When a group's timer fires or an immediate flush is triggered, the filtered alerts proceed to routing:

1. **Notification Object Creation**: A `Notification` struct is created with:
   - `GroupKey`: The group's identifier
   - `Alerts`: The filtered list of non-suppressed alerts (both firing and recently resolved)

- `Receiver`: Initially empty; populated during routing walk

2. **Routing Tree Walk**: `Router.Route(notification)` traverses the routing tree starting from the root `Route`:
   - At each route node, check if **all** of the route's `Matchers` match the **common labels** across all alerts in the notification (labels with identical values across all alerts).
   - If matchers match:
     - If the route has a `Receiver` configured, call `Router.Send()` to dispatch the notification to that receiver (subject to rate limiting).
     - If `Continue` is `true`, continue checking child routes recursively.
     - If `Continue` is `false`, stop further routing for this notification at this branch.
   - If matchers don't match, skip this route and its children.
3. **Receiver-Specific Processing**: For each matched receiver:
   - **Rate Limiting**: `RateLimiter.Wait()` blocks if the receiver has exceeded its configured rate limit (implemented via token bucket).
   - **Template Rendering**: Alert annotations and labels are templated using `TemplateCache.Render()` to create human-readable messages tailored to the receiver type (Slack, email, etc.).
   - **Delivery with Retry**: `Router.Send()` uses exponential backoff to retry failed deliveries up to a maximum number of attempts. Failed notifications after all retries are logged but dropped (to prevent system backlog).
4. **Resolution Notifications**: When all alerts in a group transition to resolved state (via subsequent `Grouper.Process()` calls with resolved alerts), the group's `group_interval` timer fires one final time with only resolved alerts. This sends a resolution notification to the same receivers that got the original firing notification.

Example Walkthrough: Our notification with two `HighRequestRate` alerts (both `severity="critical", cluster="prod-us"`) reaches the router. The routing tree has: root → match `severity="critical"` → receiver="pagerduty-primary", continue=false. Another branch: root → match `cluster="staging"` → receiver="slack-dev". The notification matches the first route, sends to PagerDuty with critical severity, and stops (continue=false). The PagerDuty integration key is retrieved from `Receiver.Config`, and an incident is created.

Step 5: Cleanup and State Maintenance

Background processes maintain system health:

1. **Expired Silences Cleanup**: Periodically, the `Silencer` sweeps its `byExpiry` sorted list to remove silences with `EndsAt` in the past. Previously suppressed alerts may now trigger notifications on next evaluation cycle.
2. **Empty Group Removal**: When `Group.RemoveAlert()` removes the last alert from a group (due to resolution), `Grouper.cleanupGroup()` stops all timers and removes the group from the `groups` map.
3. **Stale Alert Pruning**: The Rule Evaluator periodically removes resolved alerts from its state map after a configurable retention period to prevent memory growth.

Internal Message Formats

Components communicate primarily through two data structures: the `Alert` (flowing from Rule Evaluator through Grouper to suppression checks) and the `Notification` (flowing from Grouper through Router to receivers). These structures act as **envelopes** carrying both data and metadata through the pipeline.

Alert Structure

The `Alert` is the fundamental unit of concern in the system. It represents a single condition that requires attention, either currently active (firing), developing (pending), or recently resolved.

Field	Type	Description
`Labels`	`map[string]string`	**Identifying dimensions**. Key-value pairs that uniquely identify the alert source and context. Includes metric labels (e.g., `job`, `instance`) plus rule-added labels (e.g., `severity`). Used for grouping, matching, and display.
`Annotations`	`map[string]string`	**Human-readable information**. Supplementary text describing the alert, often templated with label values. Common keys: `summary`, `description`, `runbook_url`. Not used for grouping or matching—only for display in notifications.
`StartsAt`	`time.Time`	**Detection timestamp**. When the alert condition first became true (for pending) or completed the `ForDuration` (for firing). For resolved alerts, this remains as the original detection time.
`EndsAt`	`time.Time`	**Resolution timestamp**. Initially zero for firing alerts. Set to the time when the alert condition stopped being true (transition to resolved). Used to calculate alert duration and for resolution notifications.
`State`	`string`	**Current lifecycle state**. One of: `"inactive"` (condition false, not tracked), `"pending"` (condition true but `ForDuration` not met), `"firing"` (condition true and `ForDuration` met), `"resolved"` (condition became false after firing).
`GeneratorURL`	`string`	**Debugging URL**. A link to the metric query or dashboard that provides context for investigating the alert. Typically points to the metrics collector UI with the rule's expression pre-filled.

> **Design Insight**: The separation between `Labels` (for machine processing) and `Annotations` (for human consumption) is deliberate. Labels must be relatively stable and low-cardinality to enable effective grouping, while annotations can contain rich, variable text without affecting system behavior.

Notification Structure

The `Notification` represents a batch of related alerts destined for one or more receivers. It's the output of the grouping process and the input to the routing system.

Field	Type	Description
`GroupKey`	`string`	**Group identifier**. A deterministic hash derived from the grouping label values (e.g., `sha256(alertname=HighRequestRate&cluster=prod-us)`). Enables receivers to correlate multiple notifications about the same ongoing issue.
`Alerts`	`[]*Alert`	**Alert batch**. The list of alerts belonging to this group that have passed suppression checks. May include both firing and resolved alerts (for resolution notifications). Ordered by `StartsAt` (oldest first) within the notification.
`Receiver`	`string`	**Destination receiver name**. Populated during routing tree walk with the name of the receiver configuration (e.g., `"pagerduty-primary"`, `"slack-ops"`). Used by `Router.Send()` to look up receiver-specific configuration.

Supporting Data Structures in Transit

While `Alert` and `Notification` are the primary carriers, other structures facilitate component interactions:

- **MetricQueryResult**: Returned from `Client.Query()` to the Rule Evaluator. Contains the raw metric data and status code from the metrics collector API.
- **Group Internal State**: Within the Grouper, alerts are stored in `Group.alerts` slice, but the external interface only sees individual alerts flowing in via `Grouper.Process()` and notifications flowing out via the `flushFunc` callback.
- **Matcher Evaluation Context**: When checking silences, inhibitions, or routes, the system evaluates `Matcher` objects against label sets. These matchers are configured in `Silence`, `InhibitionRule`, and `Route` objects but aren't themselves passed between components—they're part of each component's configuration.

Example Data Flow: Consider an alert flowing through the pipeline:

1. **Rule Evaluator → Grouper**: `Alert{Labels: {alertname="HighCPU", instance="01", severity="critical"}, State: "firing", ...}`
2. **Grouper internal**: Alert added to `Group` with key derived from `alertname` and `severity`.
3. **Grouper → Silencer/Inhibitor**: Same `Alert` object passed to suppression checks (by reference).
4. **Grouper → Router**: `Notification{GroupKey: "abc123", Alerts: [Alert{...}], Receiver: ""}`
5. **Router → Receiver**: After routing walk, `Notification.Receiver` set to `"slack-ops"`, then dispatched via HTTP POST to Slack webhook with templated message.

Component Communication Patterns

Components interact through three primary patterns:

1. **Synchronous Method Calls**: The `Engine` calls `Grouper.Process()` synchronously after each rule evaluation cycle. This ensures alerts enter the grouping system immediately upon detection.
2. **Timer-Driven Callbacks**: The Grouper uses `timerManager` objects to schedule future actions (group_wait, group_interval). When timers fire, they invoke the `flushFunc` callback registered by the Router, creating an asynchronous but time-bound flow.
3. **Configuration Injection**: Silences, inhibition rules, and routing configuration are loaded at startup and can be updated dynamically via admin APIs. Components receive these configurations and rebuild internal indices without stopping the data flow.

> **Key Design Principle**: The data flow is **unidirectional**—alerts move forward through the pipeline (Evaluator → Grouper → Router) but never backward. This simplifies reasoning about system state and prevents circular dependencies. However, the Inhibitor component needs read access to currently firing alerts, creating a **read-only feedback loop** from the Grouper's state back to the suppression check point.

Failure Handling in the Data Flow

When components fail or encounter errors, the system employs several strategies to maintain functionality:

Failure Point	Detection	Recovery Strategy
Metric Query Failure	HTTP error or timeout from `Client.Query()`	Log error, skip this evaluation cycle for the rule, keep previous alert state unchanged (stale alerts continue firing).
Grouper Processing Backlog	Channel buffer full or processing delay	Rule Evaluator drops alerts after retries (fail-fast). Better to lose some alerts than cause memory explosion.
Silence/Inhibition Check Error	Matcher evaluation panic (e.g., invalid regex)	Catch panic, log error, treat matcher as non-matching (fail-open to avoid suppressing alerts incorrectly).
Router Delivery Failure	HTTP error from receiver	Exponential backoff retry up to max attempts, then log and drop notification (acknowledged data loss).
Timer Manager Failure	Timer callback panic	Restart timer with jitter, log error, attempt to flush group immediately to prevent indefinite stall.

The system prioritizes **availability over consistency** in notification delivery—it's better to send a duplicate or partial notification than to send none at all during partial failures.

Implementation Guidance

A. Technology Recommendations

Component	Simple Option	Advanced Option
Inter-component Communication	Direct method calls within same process	gRPC with streaming for cross-process communication
Timer Management	`time.Ticker` with manual reset logic	Dedicated timer wheel (e.g., `github.com/RussellLuo/timingwheel`) for scalable timers
Callback Registration	Function pointers/interfaces	Event bus with topic subscription (e.g., `github.com/asaskevich/EventBus`)

B. Recommended File Structure

Place the coordination logic that ties components together in a dedicated package:

```
alertingsystem/
├── cmd/
│   └── server/
│       └── main.go               # Component initialization and wiring
├── internal/
│   ├── engine/                   # Rule Evaluator (Milestone 1)
│   ├── grouper/                  # Grouper (Milestone 2)
│   ├── silencer/                 # Silencer & Inhibitor (Milestone 3)
│   ├── router/                   # Notification Router (Milestone 4)
│   └── coordinator/              # NEW: Component coordination and data flow
│       ├── coordinator.go        # Main pipeline orchestration
│       ├── pipeline.go           # Alert flow definition
│       └── lifecycle.go          # Alert state tracking across components
└── pkg/
    └── alert/                    # Shared data structures
```

C. Pipeline Orchestration Starter Code

Here's complete code for a simple coordinator that wires components together:

```go
// internal/coordinator/coordinator.go

package coordinator

import (
    "context"
    "sync"
    "time"

    "alertingsystem/internal/engine"
    "alertingsystem/internal/grouper"
    "alertingsystem/internal/router"
    "alertingsystem/internal/silencer"
    "alertingsystem/pkg/alert"
)

// Coordinator wires components together and manages the alert pipeline.
type Coordinator struct {
    engine    *engine.Engine
    grouper   *grouper.Grouper
    silencer  *silencer.Silencer
    inhibitor *silencer.Inhibitor
    router    *router.Router

    // Pipeline channels for alert flow
    alertChan chan *alert.Alert

    wg     sync.WaitGroup
    cancel context.CancelFunc
}

// New creates and wires all components.
func New(cfg Config) (*Coordinator, error) {
    // Initialize components (simplified)
    eng := engine.New(cfg.Engine)
    grp := grouper.New(cfg.Grouper)
    sil := silencer.New(cfg.Silencer)
    inh := silencer.NewInhibitor(cfg.Inhibitor)
    rtr := router.New(cfg.Router)

    // Create coordinator
    c := &Coordinator{
        engine:    eng,
```

```
        grouper:   grp,
        silencer:  sil,
        inhibitor: inh,
        router:    rtr,
        alertChan: make(chan *alert.Alert, 1000),
    }

    // Wire grouper flush callback to router
    grp.SetFlushFunc(c.handleGroupFlush)

    // Wire inhibitor to get firing alerts from grouper
    inh.SetAlertsGetter(grp.GetFiringAlerts)

    return c, nil
}

// Run starts all components and the coordination loop.
func (c *Coordinator) Run(ctx context.Context) error {
    ctx, cancel := context.WithCancel(ctx)
    c.cancel = cancel

    // Start components
    c.wg.Add(4)
    go func() {
        defer c.wg.Done()
        c.engine.Run(ctx)
    }()
    go func() {
        defer c.wg.Done()
        c.grouper.Run(ctx)
    }()
    go func() {
        defer c.wg.Done()
        c.silencer.Run(ctx) // Cleans expired silences
    }()

    // Start alert processing pipeline
    go func() {
        defer c.wg.Done()
        c.processAlerts(ctx)
    }()
```

```
    return nil
}

// Stop gracefully shuts down all components.
func (c *Coordinator) Stop() {
    if c.cancel != nil {
        c.cancel()
    }
    c.wg.Wait()
}

// processAlerts reads alerts from engine and passes through pipeline.
func (c *Coordinator) processAlerts(ctx context.Context) {
    for {
        select {
        case <-ctx.Done():
            return
        case alert := <-c.engine.AlertOutput(): // Assume engine has output channel
            // Pass through suppression checks
            if c.silencer.CheckSuppressed(alert) {
                continue // Alert silenced, drop it
            }
            if c.inhibitor.CheckSuppressed(alert) {
                continue // Alert inhibited, drop it
            }

            // Send to grouper
            c.grouper.Process(alert)
        }
    }
}

// handleGroupFlush is called by grouper when a group is ready for notification.
func (c *Coordinator) handleGroupFlush(groupKey string, alerts []*alert.Alert) {
    // Create notification
    notification := &alert.Notification{
        GroupKey: groupKey,
        Alerts:   alerts,
    }

    // Route to appropriate receivers
```

```
    c.router.Route(notification)
}
```

D. Alert Flow Skeleton Code

For the core alert processing logic in the coordinator:

```go
// internal/coordinator/pipeline.go
package coordinator

import (
    "context"
    "log"

    "alertingsystem/pkg/alert"
)

// processAlertThroughPipeline handles a single alert through all stages.
func (c *Coordinator) processAlertThroughPipeline(ctx context.Context, a *alert.Alert) error {
    // TODO 1: Validate alert has required fields (Labels, State, StartsAt)
    // TODO 2: Check if alert is suppressed by any active silence
    // TODO 3: Check if alert is inhibited by any firing source alerts
    // TODO 4: If suppressed or inhibited, log at debug level and return nil (alert processed)
    // TODO 5: Pass alert to grouper via grouper.Process(a)
    // TODO 6: Update metrics counters for alerts processed/filtered
    // TODO 7: Return any unrecoverable error (e.g., grouper backlog full)

    return nil
}

// handleGroupForNotification processes a ready group into notifications.
func (c *Coordinator) handleGroupForNotification(group *grouper.Group) {
    // TODO 1: Extract all alerts from the group
    // TODO 2: Filter out alerts that are currently suppressed (silenced or inhibited)
    // TODO 3: If no alerts remain after filtering, cleanup group and return
    // TODO 4: Create Notification object with filtered alerts
    // TODO 5: Pass notification to router via router.Route(notification)
    // TODO 6: If group has no active alerts (all resolved), schedule group cleanup
    // TODO 7: Restart group_interval timer if group still has active alerts
}
```

E. Language-Specific Hints

- **Go Channels for Flow Control**: Use buffered channels between engine and coordinator to prevent backpressure from slowing down rule evaluation. Monitor channel capacity metrics.
- **Context Propagation**: Pass the same `context.Context` through all components to enable coordinated shutdown when cancellation occurs.
- **Structured Logging**: Use log fields that include `alert_fingerprint`, `group_key`, and `receiver` to trace an alert's journey through logs.

F. Milestone Checkpoint

After implementing the data flow coordination:

1. **Run the integration test**:

```
go test ./internal/coordinator/... -v -count=1
```
BASH

Expected: Tests should pass showing alerts flow from engine → grouper → router.

2. **Manual verification**:
   - Start the system with a simple rule (CPU > 90%).
   - Use `curl` to create a silence that matches the alert labels.
   - Trigger the alert condition (e.g., run a CPU stress test).
   - Verify no notification arrives (silence working).
   - Delete the silence.
   - Verify notification arrives within `group_wait` + processing time.
3. **Debugging tip**: If alerts appear in logs but no notifications arrive, check:
   - Silencer/inhibitor logs for suppression messages
   - Grouper's active groups count via debug endpoint
   - Router's matched routes log

7. Error Handling and Edge Cases

> **Milestone(s):** This section addresses the robustness and fault tolerance of the entire Alerting System, encompassing error scenarios and edge cases across all four milestones (Rule Evaluation, Alert Grouping, Silencing & Inhibition, Notification Routing).

An alerting system must remain operational during infrastructure failures, configuration errors, and unexpected data conditions—precisely when its alerts are most critical. This section details the system's failure modes, strategies for detection and isolation, and mechanisms for graceful degradation and recovery. The design follows the principle of **fail-open**: when components fail, they default to allowing alerts through the pipeline rather than blocking them, preventing silent failures during outages.

7.1 Core Error Handling Philosophy

Think of the alerting system as a **water purification plant**. The raw water (metric data) must be cleaned and delivered even when some filtration stages malfunction. If a filter fails, the plant bypasses it and continues delivering water (perhaps with a warning about reduced purity) rather than shutting down entirely. This fail-open approach ensures that critical alerts still reach operators, though with potentially reduced functionality like grouping or silencing.

The system implements a **unidirectional flow** of alerts through a processing pipeline. Errors in any component should not cause alerts to be lost or trapped indefinitely. Instead, errors are logged, metrics are emitted, and the alert continues downstream with appropriate tagging or default behavior.

7.2 Component Failure Modes and Mitigations

Each component has distinct failure modes. The following tables detail the most critical scenarios, detection strategies, and recovery mechanisms.

7.2.1 Rule Evaluator Failures

The Rule Evaluator's primary failure modes involve metric query failures, expression evaluation errors, and state corruption.

Failure Mode	Detection Strategy	Recovery Mechanism	Impact & Degradation
Metrics API unreachable	HTTP client returns network error or timeout; `Client.Query` returns error.	Retry with exponential backoff (max 3 attempts). If all fail, skip evaluation cycle and log error. Increment error metric.	Alerts for affected rules are not updated until connectivity restored. Existing alert states persist.
PromQL expression returns error (e.g., invalid syntax, division by zero)	`Client.Query` returns error with status code and error message from API.	Mark rule as invalid, skip its evaluation, log error with rule name. Increment `rule_evaluation_errors` counter.	Specific rule becomes inactive; other rules continue evaluating.
Template rendering panic during `Rule.RenderAnnotations`	Recover from panic in evaluation goroutine, capture stack trace.	Log panic with rule name and template, skip annotation rendering for that alert (use empty annotations). Increment `template_panic` counter.	Alert fires but with missing or malformed annotations in notification.
For-duration timer reset due to flapping metric	Engine detects metric value crossing threshold boundary during pending state.	Reset pending timer to zero; alert remains in pending state until condition holds continuously for full duration.	Prevents premature firing from transient spikes.
Memory leak from growing alert state map	Monitor `engine_state_alerts` gauge (count of active alerts). Implement periodic state cleanup.	`Engine.evaluateAllRules` removes resolved alerts from state map after retention period (e.g., 1 hour after resolution).	Prevents unbounded memory growth from long-running alerts.
Deadlock in state mutex	Use `sync.RWMutex` with timeouts via `context.Context` in critical sections (optional).	Log warning if lock acquisition exceeds threshold (e.g., 5 seconds). Implement circuit breaker to skip evaluation if deadlock suspected.	Temporary evaluation skip prevents full system hang.

> **Design Principle: Fail-Open for Metric Queries** When the metrics API is unreachable, the Rule Evaluator skips evaluation rather than blocking or crashing. This ensures that existing alert states persist and the pipeline continues processing (e.g., resolved alerts can still flow through). The alternative—halting all evaluation—would cause complete alert blindness during metric collector outages.

7.2.2 Grouper Failures

The Grouper's failures typically involve timer management issues, group key collisions, and memory leaks.

Failure Mode	Detection Strategy	Recovery Mechanism	Impact & Degradation
Timer callback panic during `flushFunc`	Recover panic in timer goroutine, capture stack trace.	Log panic with group key, discard the affected group (force-flush immediately). Increment `group_timer_panic` counter.	Group notifications may be sent prematurely or duplicated.
Orphaned alerts due to label changes after grouping	`Grouper.Process` detects group key mismatch with alert's current labels.	Call `Grouper.moveAlert` to migrate alert to correct group; clean up old group if empty. Log label change event.	Prevents alerts from being stuck in wrong notification groups.
Group key collision (two distinct label sets produce same hash)	Extremely rare with SHA-256 fingerprint. Detect via label validation in `Grouper.calculateGroupKey`.	Use full label set sorted string representation as fallback key. Log collision event for investigation.	Groups may incorrectly merge, causing unrelated alerts to batch together.
Memory leak from empty groups not cleaned up	`Grouper.cleanupGroup` not called after last alert removed.	Implement periodic sweep (every 10 minutes) that removes groups with `HasActiveAlerts() == false`.	Prevents accumulation of empty group structures.
Long `group_wait` delays critical alerts	Monitor `alert_group_wait_duration` histogram. Implement priority override mechanism.	If alert has `priority: critical` label, bypass group_wait and trigger immediate flush (configurable).	Critical alerts get faster notification at cost of grouping benefits.
Backpressure from slow downstream (Router busy)	`Grouper.FlushGroup` blocks on channel send to Coordinator.	Use buffered channel with size limit; if full, log warning and drop notification (but keep alerts in group for next interval).	Prevents Grouper from stalling entire pipeline; may delay notifications.

ADR: Group Timer Isolation

- **Context**: Timer callbacks execute concurrently and may panic, potentially crashing the entire grouping subsystem.
- **Options Considered**: 1) Let panic propagate and crash process; 2) Recover panic and discard affected group; 3) Use a supervisor goroutine to restart timers.
- **Decision**: Recover panic and discard the affected group (force-flush).
- **Rationale**: Crashing the process is too severe for a single group failure. Restarting timers adds complexity. Discarding the group ensures notifications are still sent (though perhaps with incorrect timing) while keeping the system running.
- **Consequences**: May cause duplicate or mistimed notifications for one group, but preserves overall system availability.

7.2.3 Silencer & Inhibitor Failures

Silencer and Inhibitor failures involve matcher evaluation errors, race conditions, and circular dependencies.

Failure Mode	Detection Strategy	Recovery Mechanism	Impact & Degradation
Matcher syntax error in silence creation	`ParseMatcher` returns error during `Silencer.AddSilence`.	Reject silence creation, return validation error to API caller. Log invalid matcher expression.	Invalid silences never become active; existing silences unaffected.
Inhibition loop (alerts mutually inhibit each other)	Detect cycles in inhibition graph during `Inhibitor.AddRule`.	Reject rule that creates cycle, return validation error. Log cycle detection warning.	Prevents deadlock where all alerts suppress each other and no notifications fire.
Race condition between silence creation and firing alert	Alert checked against silences while silence set is being updated.	Use `sync.RWMutex` in `Silencer.CheckSuppressed` and `AddSilence`. Ensure atomic updates.	Minimal: alert may briefly fire before being silenced, or vice versa.
Expired silences not cleaned up	`Silencer` maintains `byExpiry` slice; cleanup relies on periodic tick.	Run background goroutine every minute to remove expired silences from active map. Log cleanup counts.	Expired silences consume memory but don't affect matching (check includes time bounds).
Regex matcher catastrophic backtracking on large label values	Monitor `matcher_evaluation_duration` histogram; timeout individual match operations.	Implement 100ms timeout per matcher evaluation using `context.WithTimeout`. Fall back to false (no match).	Prevents denial-of-service via malicious label values; may incorrectly match/unmatch.
Silence timezone confusion (start/end times interpreted incorrectly)	Validate `StartsAt` and `EndsAt` during creation (EndsAt > StartsAt).	Store and compare times in UTC only; reject silences with end before start.	Prevents silences that are always active or never active due to timezone errors.

7.2.4 Notification Router Failures

Router failures involve receiver integration errors, rate limiting issues, and configuration problems.

Failure Mode	Detection Strategy	Recovery Mechanism	Impact & Degradation
Receiver integration failure (Slack webhook timeout, email server down)	`Router.Send` returns error after all retries exhausted. HTTP client returns 4xx/5xx status.	Log error with receiver name, increment `notification_failures` counter. Store failed notifications in dead-letter queue for replay.	Alerts not delivered to that channel; other channels continue receiving.
Rate limiter starving critical alerts	Monitor `rate_limiter_wait_duration` histogram; track alerts dropped due to rate limiting.	Implement priority-based rate limiting: alerts with `severity: critical` bypass rate limit or use separate bucket.	Prevents important alerts from being throttled during high-volume incidents.
Missing default route causing unrouted alerts	`Router.Route` reaches leaf node with no matching receiver and `continue=false`.	Log warning with alert labels, increment `unrouted_alerts` counter. Route to fallback "default" receiver if configured.	Alerts may be lost if no fallback; logging provides visibility.
Configuration hot-reload error (invalid YAML/JSON)	Validate entire routing tree before applying; check for cycles, invalid matchers.	Reject entire new configuration, keep previous configuration active. Log validation errors in detail.	Prevents routing table corruption; existing routes continue working.
Notification grouping across different routes	Alert matches multiple routes due to `continue=true` and overlapping matchers.	Each route maintains independent grouping state; alerts may appear in multiple notifications.	Can cause duplicate notifications; documented behavior requiring careful configuration.
Memory growth from retry queue	Monitor `retry_queue_size` gauge; implement maximum queue size and age.	Drop oldest notifications when queue exceeds capacity (e.g., 10,000 items). Log dropped count.	Prevents unbounded memory consumption during prolonged receiver outage.

ADR: Dead-Letter Queue for Failed Notifications

- **Context**: When a receiver fails permanently (e.g., misconfigured webhook), notifications are lost without trace.
- **Options Considered**: 1) Drop failed notifications silently; 2) Retry indefinitely with exponential backoff; 3) Store in persistent queue for manual replay.
- **Decision**: In-memory dead-letter queue with size and age limits, plus logging.
- **Rationale**: Persistent storage adds operational complexity. Silent dropping is unacceptable. In-memory queue provides temporary buffer for transient failures while preventing unbounded growth.
- **Consequences**: Notifications may be lost if process restarts during prolonged outage. Queue size limits prevent memory exhaustion.

7.2.5 Coordinator Failures

The Coordinator orchestrates the pipeline; its failures affect the entire data flow.

Failure Mode	Detection Strategy	Recovery Mechanism	Impact & Degradation
Pipeline stage panic (e.g., Grouper panics on nil alert)	Recover panic in `Coordinator.processAlertThroughPipeline`.	Log panic with stack trace, skip that alert, increment `pipeline_panic` counter. Alert is dropped but pipeline continues.	Single alert lost; other alerts continue flowing.
Channel deadlock (alertChan full with no consumer)	Monitor channel buffer utilization; implement timeout on send.	If `alertChan` send blocks for >5 seconds, log warning and drop alert. Increment `channel_full_drops` counter.	Prevents pipeline from stalling entire system; some alerts may be lost.
Component initialization failure (e.g., Router fails to load config)	`Coordinator.Run` checks error returns from component constructors.	Fail fast: return initialization error, process exits. Requires external restart (e.g., Kubernetes pod restart).	No partial operation; clear failure signal to operator.
Shutdown timeout during graceful termination	`Coordinator.Stop` implements timeout context for component cleanup.	After timeout, log warning and force exit. Increment `shutdown_timeout` counter.	Some in-flight alerts may be lost, but process terminates promptly.
CPU exhaustion from evaluation loop running too frequently	Monitor `evaluation_cycle_duration` vs configured interval.	Implement cycle duration tracking; if evaluation takes longer than interval, skip next cycle and log warning.	Prevents snowballing latency; maintains steady state.

7.3 System-Wide Failure Scenarios

Beyond individual components, the system must handle infrastructure-level failures and edge cases.

7.3.1 Data Corruption and Version Skew

Scenario	Detection	Recovery
Persisted state corruption (silences file, retry queue)	Checksum validation on load; JSON unmarshal errors.	Discard corrupted file, log error, start with empty state. Increment `state_corruption` counter.
Configuration version mismatch (new fields unknown to old code)	Configuration struct uses strict JSON decoding with `DisallowUnknownFields`.	Reject configuration with unknown fields during validation. Provide clear error about version compatibility.
Clock skew between nodes in future cluster deployment	Compare system time with NTP; monitor timestamp anomalies.	Reject silences with creation time too far in future (e.g., >5 minutes). Log clock skew warning.

7.3.2 Resource Exhaustion

Resource	Detection	Recovery
Memory	Monitor Go runtime memory metrics; implement limits via cgroups.	Gradual degradation: drop oldest cached templates, clear resolved alert history, restart process as last resort.
File descriptors	Track open FD count; fail fast on `open()` errors.	Log warning when approaching limit (e.g., 80% of max). Close idle HTTP connections.
CPU	Monitor goroutine count; detect runaway goroutines.	Implement circuit breakers that skip non-critical work (e.g., template rendering, metric queries) under load.
Disk space (for future persistent queues)	Check disk usage before write operations.	Enter degraded mode: stop persisting new state, log errors, rely on in-memory only.

7.3.3 Network Partition and Distributed Coordination

> **Note**: The current design is single-node. Future clustering would introduce additional failure modes.

Scenario	Detection	Recovery (Future)
Split-brain in clustered silence storage	Quorum detection; lease-based leadership.	Only leader accepts writes; followers reject modification requests.
Network partition isolates notification gateway	Health checks between router and gateway fail.	Failover to secondary gateway; queue notifications for later delivery.
Duplicate notifications from multiple active instances	Deduplication IDs in notifications; coordinated tracking.	Store recently sent notification fingerprints; suppress duplicates within time window.

7.4 Error Propagation and Observability

The system employs a layered observability strategy to detect and diagnose failures:

1. **Structured Logging**: Every component logs errors with consistent fields: `component`, `error`, `alert_fingerprint` (if applicable), `rule_name`, `group_key`, etc. Log levels:
   - `ERROR`: Operations failing after retries, corrupted state, panics
   - `WARN`: Recoverable errors, degraded performance, near-limits
   - `INFO`: Normal operations (config changes, silence creation)
   - `DEBUG`: Detailed tracing for debugging
2. **Metrics Exposition**: Prometheus metrics exposed at `/metrics` endpoint:

```promql
# Counter examples
alerting_rule_evaluation_errors_total{rule="..."}
alerting_notification_failures_total{receiver="..."}
alerting_alerts_dropped_total{reason="channel_full"}

# Gauge examples
alerting_active_alerts
alerting_silences_active
alerting_groups_active

# Histogram examples
alerting_evaluation_duration_seconds
alerting_notification_latency_seconds
```

3. **Health Endpoints**: HTTP `/health` returns component status:
    - `200 OK`: All components healthy
    - `503 Service Unavailable`: One or more components degraded (with details in JSON body)
4. **Alerting on the Alerting System**: The system monitors itself via:
    - Rules that fire when error rates exceed thresholds (e.g., `rate(alerting_notification_failures_total[5m]) > 0.1`)
    - Rules detecting resource exhaustion (e.g., `process_resident_memory_bytes > 1GB`)
    - These self-monitoring alerts route to a dedicated "alerting-infra" receiver with higher urgency.

7.5 Graceful Degradation Matrix

When components fail partially, the system degrades functionality in predictable ways:

Failed Component	Degraded Functionality	Preserved Functionality
Metrics API	No new alerts triggered; existing alert states frozen	Silencing, grouping, routing continue working; resolved alerts still processed
Grouper	Alerts bypass grouping, sent individually immediately	Rule evaluation, silencing, routing still function
Silencer	All silences ignored; no alert suppression	Rule evaluation, grouping, routing still function
Inhibitor	Inhibition rules ignored; all alerts pass through	Rule evaluation, grouping, silencing, routing still function
Router (single receiver)	Specific channel (e.g., Slack) fails; alerts not delivered there	Other channels (email, PagerDuty) continue receiving; grouping still occurs
Coordinator channel	Alerts dropped if channel full; pipeline continues	Components continue internal processing; may recover when backpressure eases

7.6 Common Pitfalls in Error Handling

⚠️ Pitfall: Silent Dropping of Alerts

- **Description**: Catching an error and continuing without logging or metrics, causing alerts to disappear silently.
- **Why Wrong**: Operators lose visibility into system failures; critical alerts may be lost without trace.
- **Fix**: Always log errors at appropriate level and increment metrics counters. If an alert must be dropped, log the fingerprint and reason.

⚠️ Pitfall: Blocking Pipeline on Slow Downstream

- **Description**: Synchronous calls to external services (e.g., Slack API) without timeouts or buffers.
- **Why Wrong**: Single slow receiver can stall entire alert pipeline, causing alerts to queue up and delay notifications.
- **Fix**: Use buffered channels between components, implement timeouts on all external calls, and drop notifications rather than blocking indefinitely.

⚠️ Pitfall: Retry Storms

- **Description**: Immediate retry of failed notifications without backoff, overwhelming already-struggling receivers.
- **Why Wrong**: Can exacerbate receiver outages and consume system resources.
- **Fix**: Implement exponential backoff with jitter, maximum retry limits, and circuit breakers that stop retrying after repeated failures.

⚠️ Pitfall: State Corruption Without Recovery

- **Description**: Loading corrupted state (e.g., silences JSON) and crashing or entering undefined behavior.
- **Why Wrong**: System becomes unavailable due to corrupt data that could be recovered from.
- **Fix**: Validate data on load, provide repair mechanisms (e.g., discard corrupt entries), and maintain backups of critical state.

⚠️ Pitfall: Missing Idempotency

- **Description**: Sending duplicate notifications when retrying failed sends or after process restart.
- **Why Wrong**: Causes alert fatigue and confusion about whether an issue is new or repeated.
- **Fix**: Use deduplication keys (e.g., alert fingerprint + timestamp) and track recently sent notifications.

7.7 Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Error Tracking	Structured logging with `log/slog`	Distributed tracing with OpenTelemetry
Metrics Collection	Prometheus client library `github.com/prometheus/client_golang`	Custom metrics aggregator with histogram buckets
Dead-Letter Queue	In-memory ring buffer with `container/ring`	Persistent queue with `github.com/nats-io/nats.go`
Circuit Breaker	Simple counter with reset timeout	`github.com/sony/gobreaker` for stateful circuit breaking
Rate Limiter	`golang.org/x/time/rate` token bucket	Custom priority-based multi-bucket limiter

B. Recommended File Structure

```
alerting-system/
  internal/
    errors/                  # Error types and handling utilities
      types.go               # Custom error types (e.g., ErrSilenceNotFound)
      recovery.go            # Panic recovery utilities
      metrics.go             # Error metric registration
    observability/
      logging.go             # Structured logger setup
      metrics.go             # Metric definitions and registration
      health.go              # Health check endpoint
    coordinator/
      coordinator.go         # Main pipeline with error handling
      deadletter.go          # Dead-letter queue implementation
    # ... other component directories
  pkg/
    api/                     # External API
      middleware/            # Error handling middleware
        recovery.go          # HTTP panic recovery
        logging.go           # Request logging
```

C. Infrastructure Starter Code

Panic Recovery Wrapper for Goroutines:

```go
// internal/errors/recovery.go

package errors


import (

    "log/slog"

    "runtime/debug"

)


// Recoverer captures panics in goroutines and logs them

func Recoverer(component string) {

    if r := recover(); r != nil {

        slog.Error("panic recovered in goroutine",

            "component", component,

            "panic", r,

            "stack", string(debug.Stack()))

    }

}


// Go launches a goroutine with panic recovery

func Go(component string, fn func()) {

    go func() {

        defer Recoverer(component)

        fn()

    }()

}
```

Health Check Structure:

GO

```go
// internal/observability/health.go

package observability


import (

    "encoding/json"

    "net/http"

    "sync"

    "time"

)


type HealthStatus struct {

    mu     sync.RWMutex

    status map[string]string // component -> status

}


func NewHealthStatus() *HealthStatus {

    return &HealthStatus{

        status: make(map[string]string),

    }

}


func (h *HealthStatus) SetComponent(component, status string) {

    h.mu.Lock()

    defer h.mu.Unlock()

    h.status[component] = status

}


func (h *HealthStatus) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    h.mu.RLock()

    defer h.mu.RUnlock()



    overall := "healthy"

    for _, status := range h.status {

        if status != "healthy" {

            overall = "unhealthy"

            break

        }

    }



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

    if overall == "unhealthy" {

        w.WriteHeader(http.StatusServiceUnavailable)

    }
```

```
    json.NewEncoder(w).Encode(map[string]any{
        "status":    overall,
        "timestamp": time.Now().UTC(),
        "components": h.status,
    })
}
```

D. Core Logic Skeleton Code

Coordinator Error Handling Pipeline:

```go
// internal/coordinator/coordinator.go

func (c *Coordinator) processAlertThroughPipeline(ctx context.Context, a *alert.Alert) error {

    // TODO 1: Validate alert is not nil, log warning and return error if nil



    // TODO 2: Check if alert is suppressed by silencer

    //   - Call c.silencer.CheckSuppressed(a)

    //   - If suppressed, log at DEBUG level and return nil (alert processed, no further action)

    //   - If error checking, log WARNING and continue (fail-open)



    // TODO 3: Check if alert is inhibited

    //   - Call c.inhibitor.CheckSuppressed(a)

    //   - If inhibited, log at DEBUG level and return nil

    //   - If error checking, log WARNING and continue (fail-open)



    // TODO 4: Pass alert to grouper

    //   - Call c.grouper.Process(a) with timeout (e.g., 5 seconds)

    //   - If timeout or error, log ERROR and increment metric

    //   - Continue anyway (alert may be lost from grouping but pipeline continues)



    // TODO 5: Return nil to indicate alert was processed (even if with errors)

    //   - All errors should have been logged and metrics incremented

}



func (c *Coordinator) handleGroupFlush(groupKey string, alerts []*alert.Alert) {

    // TODO 1: Recover from any panic in this callback

    //   - Use defer errors.Recoverer("group_flush")



    // TODO 2: Validate inputs (groupKey not empty, alerts not empty)

    //   - Log warning and return if invalid



    // TODO 3: Create notification from alerts

    //   - Generate common labels across all alerts

    //   - Format notification message



    // TODO 4: Route notification

    //   - Call c.router.Route(notification) with timeout

    //   - If error, log ERROR and increment metric

    //   - Optionally add to dead-letter queue for retry



    // TODO 5: Log successful notification at INFO level

    //   - Include groupKey, alert count, receiver names
```

```
}
```

Circuit Breaker for Receiver Integrations:

```go
// internal/router/circuitbreaker.go

type ReceiverCircuitBreaker struct {
    name          string
    failureWindow time.Duration
    failureCount  int
    maxFailures   int
    state         string // "closed", "open", "half-open"
    mu            sync.RWMutex
    lastFailure   time.Time
}

func (cb *ReceiverCircuitBreaker) Allow() bool {
    cb.mu.RLock()
    defer cb.mu.RUnlock()

    // TODO 1: If state is "open", check if reset timeout has elapsed
    //   - If elapsed, transition to "half-open" and allow single request
    //   - Otherwise, return false

    // TODO 2: If state is "half-open", allow request (will determine if should close)

    // TODO 3: If state is "closed", always allow

    return true
}

func (cb *ReceiverCircuitBreaker) RecordSuccess() {
    // TODO: Transition from "half-open" to "closed", reset failure count
}

func (cb *ReceiverCircuitBreaker) RecordFailure() {
    // TODO: Increment failure count, if exceeds maxFailures transition to "open"
    // TODO: Set lastFailure timestamp, schedule state reset
}
```

E. Language-Specific Hints

1. **Error Wrapping**: Use `fmt.Errorf("%w", err)` to wrap errors with context, enabling `errors.Is` and `errors.As` checks upstream.
2. **Context Timeouts**: Always pass `context.Context` to blocking operations and respect cancellation for graceful shutdown.
3. **Structured Logging**: Use `log/slog` with attribute names consistent across components (e.g., `alert_fingerprint`, `rule_name`, `group_key`).
4. **Metrics Registration**: Register Prometheus metrics in `init()` functions to avoid race conditions during startup.
5. **Memory Limits**: Use `runtime/debug.SetMemoryLimit()` in Go 1.19+ to trigger GC more aggressively and prevent OOM kills.

F. Debugging Tips

Symptom	Likely Cause	How to Diagnose	Fix
Alerts not firing but metric shows threshold crossed	Rule evaluation error or metrics API timeout	Check logs for `rule_evaluation_errors`, verify `Client.Query` error handling	Ensure exponential backoff on query failures; validate PromQL expression
Notifications duplicated	Group timer panic causing re-flush, or multiple routes matching	Check `group_timer_panic` metric; examine routing tree for overlapping matchers	Fix timer panic recovery; adjust route matchers or set `continue=false`
Critical alerts delayed by group_wait	No priority override mechanism	Check alert labels for `priority: critical`; examine `alert_group_wait_duration` metric	Implement priority bypass in `Grouper.Process`
Memory usage grows indefinitely	Empty groups not cleaned up, or resolved alerts retained	Monitor `alerting_groups_active` and `alerting_active_alerts` metrics	Implement periodic cleanup sweeps in Grouper and Engine
Notifications stop entirely after receiver failure	Circuit breaker stuck "open", dead-letter queue full	Check circuit breaker state logs; monitor dead-letter queue size	Implement manual reset endpoint for circuit breakers; increase queue capacity
Process crashes on configuration reload	Panic in template parsing or validation	Check panic stack trace in logs; validate template syntax before hot reload	Add validation step before applying new config; recover panics in reload goroutine

8. Testing Strategy

> **Milestone(s):** This section provides a comprehensive testing framework for the entire Alerting System, covering testing methodologies and specific implementation checkpoints for all four milestones.

Testing an alerting system requires verifying both functional correctness under expected conditions and resilience under failure scenarios. The system must be reliable—false negatives (missed alerts) can lead to undetected incidents, while false positives (alert storms) cause **alert fatigue**. Our testing strategy employs multiple complementary approaches, each targeting different risk categories and architectural layers.

Testing Philosophy and Approaches

Think of testing the alerting system as **calibrating a smoke alarm**. You need to verify it detects real smoke (true positives) but doesn't trigger from steam or burnt toast (false positives), responds within the required time (performance), continues working during a power outage (resilience), and clearly communicates the location of the fire (correct routing). Each testing method serves a different calibration purpose.

1. Unit Testing: Component Isolation Unit tests verify individual components in isolation with mocked dependencies. This is the foundation of our testing pyramid, ensuring each piece of logic—expression evaluation, label matching, timer management—behaves correctly in a controlled environment.

2. Integration Testing: Component Interaction Integration tests verify that components work together correctly when connected through defined interfaces. These tests catch interface mismatches, serialization errors, and concurrency issues that unit tests might miss.

3. Scenario-Based Testing: End-to-End Workflows Scenario tests simulate real-world sequences of events that exercise the complete **end-to-end flow**. A scenario might simulate a CPU spike triggering an alert, that alert being grouped with similar alerts, suppressed by a maintenance window, then routed to Slack. These tests validate the system's behavior as users will experience it.

4. Property-Based Testing: Edge Case Discovery Property-based tests (using libraries like `gopter` for Go) automatically generate hundreds of test cases by defining invariants that must always hold. For example: "An alert's group key should always be deterministic—the same labels must produce the same key regardless of ordering." This approach discovers edge cases that manual test design might miss.

5. Chaos Engineering: Failure Resilience Chaos tests deliberately inject failures—network partitions, process crashes, high latency—to verify the system degrades gracefully. While not required for the core milestones, we design for **fail-open** behavior: if the grouping component crashes, alerts should bypass grouping rather than being lost entirely.

Test Organization and Infrastructure

Tests are organized alongside the code they verify, following Go conventions:

```
project-root/
├── cmd/
│   └── alerting/
│       └── main.go
├── internal/
│   ├── engine/                    # Rule Evaluator
│   │   ├── engine.go
│   │   ├── engine_test.go         # Unit tests
│   │   └── integration_test.go    # Integration with metrics client
│   ├── grouper/
│   │   ├── grouper.go
│   │   ├── grouper_test.go
│   │   └── scenario_test.go       # Multi-alert grouping scenarios
│   ├── silencer/
│   │   ├── silencer.go
│   │   ├── inhibitor.go
│   │   └── matcher_test.go        # Property-based tests for matchers
│   ├── router/
│   │   ├── router.go
│   │   ├── receivers/
│   │   │   ├── slack.go
│   │   │   └── slack_test.go
│   │   └── router_integration_test.go
│   └── coordinator/
│       ├── coordinator.go
│       └── coordinator_scenario_test.go # End-to-end scenarios
└── test/
    ├── fixtures/                  # Test data files
    │   ├── sample_rules.yaml
    │   └── test_alerts.json
    └── testutils/                 # Shared test utilities
        ├── mock_metrics.go
        └── test_helpers.go
```

Key Test Scenarios by Component

Component	Test Scenario	What It Verifies	Success Criteria
Rule Evaluator	Threshold crossing	Alert transitions from inactive → pending → firing when metric exceeds threshold for `for_duration`	Alert state changes at correct times
Rule Evaluator	Template rendering	Annotation templates correctly substitute label values and query results	Rendered annotations contain expected values
Grouper	Multi-alert grouping	Alerts with same grouping labels aggregate into single notification	Notification contains all alerts, group key stable
Grouper	Label change during firing	Alert changes grouping labels while active	Alert moves to new group, old group cleans up
Silencer	Time-window silencing	Alert suppressed only during active silence window	Alert appears/disappears at exact start/end times
Inhibitor	Cross-severity suppression	Low-severity alert suppressed when high-severity alert with matching labels fires	Only high-severity alert notified
Router	Tree-based routing	Alert matches multiple route levels with `continue` flags	Alert sent to all matching receivers
Router	Rate limiting	Rapid-fire alerts trigger limited notifications	Notifications respect configured burst and frequency
Coordinator	Pipeline processing	Alert flows through all components in correct sequence	Final notification contains transformed, grouped, routed alert

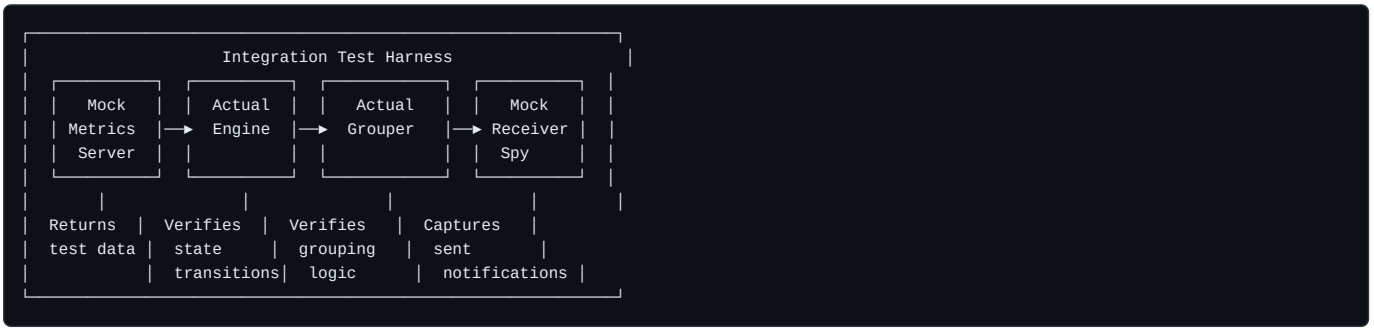
Property-Based Test Invariants

These invariants must hold for all possible inputs:

Invariant	Component	Property
Deterministic Grouping	Grouper	For any two alerts A and B: if `calculateGroupKey(A) == calculateGroupKey(B)`, then A and B share identical values for all grouping labels (after sorting)
Idempotent Processing	All	Processing the same alert twice (identical fingerprint) should not change system state (no duplicate notifications, no duplicate group entries)
Matcher Reflexivity	Silencer	Any matcher `M` should match a label set `L` if and only if `M.Match(L)` returns true, regardless of label map iteration order
Time Monotonicity	Engine	Alert's `EndsAt` should always be ≥ its `StartsAt`
Routing Completeness	Router	Every alert that enters the router either matches at least one route (including default) or is logged as unrouted (never silently dropped)

Integration Test Architecture

Integration tests use a **test harness** that runs the actual components with minimal mocking:



The mock metrics server returns predefined time series data, allowing tests to simulate specific metric conditions. The receiver spy captures all notifications for verification without actually sending emails or Slack messages.

Scenario Testing: Concrete Walkthrough

A comprehensive scenario test follows this sequence:

1. **Setup**: Start test coordinator with test configuration
2. **Initial State**: Verify no active alerts or notifications
3. **Trigger Event**: Mock metrics server returns high CPU value (0.95)
4. **Rule Evaluation**: Engine evaluates CPU rule, creates pending alert
5. **Duration Elapse**: Fast-forward time (test uses mock clock), alert transitions to firing
6. **Grouping**: Grouper receives alert, creates group with `group_wait` timer
7. **Additional Alert**: Second server also reports high CPU, joins same group
8. `group_wait` **Expires**: Group triggers notification
9. **Silence Check**: Notification checked against active silences (none)
10. **Inhibition Check**: Notification checked against inhibition rules (none)
11. **Routing**: Router matches alert to "team-platform" route, sends to Slack receiver spy
12. **Verification**: Assert spy received one notification containing both alerts with correct annotations
13. **Resolution**: Metrics return to normal, alerts resolve, resolved notification sent
14. **Cleanup**: Verify no orphaned timers, memory leaks, or stale state

Milestone Implementation Checkpoints

Each milestone includes specific verification steps to validate progress before moving to the next component. These checkpoints assume you're implementing in the order of milestones.

Milestone 1: Alert Rule Evaluation Checkpoint

Expected Behavior: The rule evaluation engine should periodically query metrics, evaluate threshold conditions with `for_duration` support, and transition alerts through pending → firing → resolved states.

Verification Commands:

```bash
# Run unit tests for the engine component
go test ./internal/engine/... -v -count=1

# Expected output should show:
# ✓ TestThresholdEvaluation (tests >, <, == operators)
# ✓ TestForDurationPendingState (alert stays pending until duration elapsed)
# ✓ TestAlertStateTransitions (inactive→pending→firing→resolved)
# ✓ TestTemplateRendering (annotations contain label values and metric results)
# ✓ TestExpressionParsing (PromQL-like expressions parsed correctly)
```

Manual Verification Steps:

1. **Start the test server**:

```bash
go run cmd/testserver/main.go --config test/fixtures/engine_test_config.yaml
```

2. **Check engine health** (via HTTP endpoint):

```bash
curl http://localhost:8080/health
# Expected: {"engine": "healthy", "evaluation_cycle": "running"}
```

3. **Trigger a test metric spike** using the mock metrics API:

```bash
# This sets CPU metric to 0.95 for 60 seconds
curl -X POST http://localhost:9090/api/v1/test/metrics \
  -H "Content-Type: application/json" \
  -d '{"metric": "cpu_usage", "value": 0.95, "duration": "60s"}'
```

4. **Monitor alert state transitions**:

```bash
# Watch the alert state change every 15 seconds (evaluation interval)
watch -n 1 'curl http://localhost:8080/api/v1/alerts 2>/dev/null | jq .'

# Expected sequence:
# 0s:  [] (no alerts)
# 15s: [{"name": "HighCPU", "state": "pending", "startsAt": "...", "value": 0.95}]
# 45s: [{"name": "HighCPU", "state": "firing", "startsAt": "...", "value": 0.95}]
# (Assuming for_duration: 30s)
```

5. **Verify resolved state** when metric returns to normal:

```bash
# Set metric back to normal
curl -X POST http://localhost:9090/api/v1/test/metrics \
  -d '{"metric": "cpu_usage", "value": 0.45, "duration": "10s"}'

# After next evaluation cycle:
# [{"name": "HighCPU", "state": "resolved", "endsAt": "...", "value": 0.45}]
```

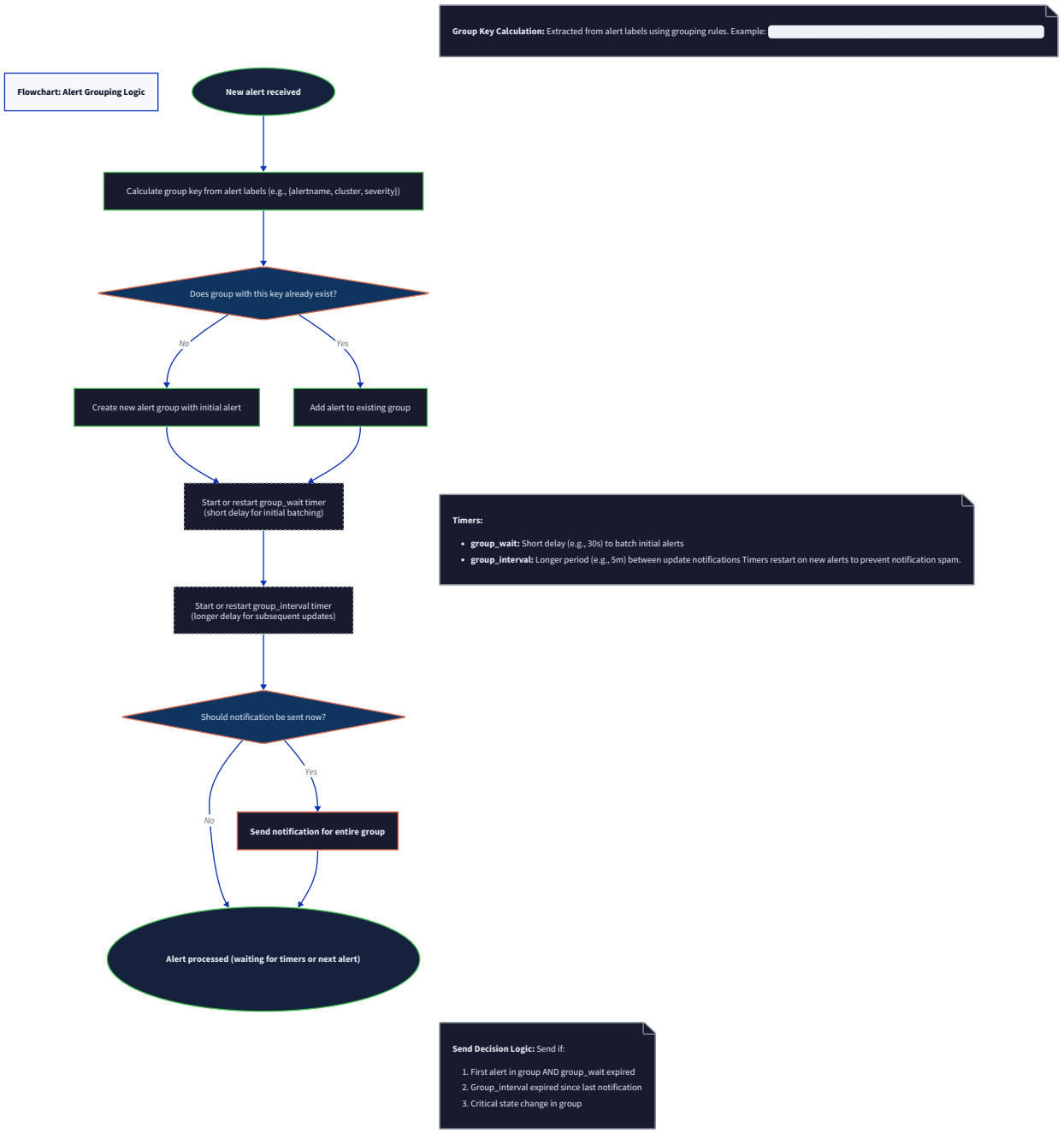
Common Failure Signs:

- Alerts never transition from pending to firing → Check `for_duration` logic and timer implementation

- Alert state flips between firing/resolved rapidly → Missing hysteresis or incorrect threshold comparison
- Template rendering errors crash engine → Missing error handling in `Rule.RenderAnnotations()`

Milestone 2: Alert Grouping Checkpoint

Expected Behavior: Alerts with identical grouping labels aggregate into groups with configurable `group_wait` and `group_interval` timers, reducing notification noise.



Verification Commands:

```bash
# Run grouper tests
go test ./internal/grouper/... -v -count=1

# Expected output includes:
# √ TestGroupKeyDeterministic (same labels → same key)
# √ TestGroupWaitDelay (first notification delayed by group_wait)
# √ TestAlertReGrouping (alert moves when labels change)
# √ TestGroupCleanup (empty groups removed)
# √ TestGroupInterval (repeat notifications spaced correctly)
```

Manual Verification Steps:

1. **Start with grouper integrated**:

```bash
go run cmd/testserver/main.go --config test/fixtures/grouping_test_config.yaml
```

2. **Send multiple alerts with same grouping labels**:

```bash
# Simulate three different servers with high CPU
curl -X POST http://localhost:8080/api/v1/test/inject-alerts \
  -d '[
    {"labels": {"alertname": "HighCPU", "instance": "web-01", "severity": "warning"}, "value": 0.95},
    {"labels": {"alertname": "HighCPU", "instance": "web-02", "severity": "warning"}, "value": 0.96},
    {"labels": {"alertname": "HighCPU", "instance": "web-03", "severity": "warning"}, "value": 0.97}
  ]'
```

3. **Monitor group formation**:

```bash
curl http://localhost:8080/api/v1/groups | jq .


# Expected: One group with key like "HighCPU/warning" containing 3 alerts
# Group should have timers active (group_wait: 30s)
```

4. **Verify single notification after `group_wait`**:

```bash
# Check notification log after 30+ seconds
tail -f /tmp/alerting_test_notifications.log


# Expected: ONE notification containing all 3 alerts
# Notification should have common labels {"alertname": "HighCPU", "severity": "warning"}
```

5. **Test alert regrouping** by changing one alert's labels:

```bash
# Update web-02 to have severity: "critical"
curl -X POST http://localhost:8080/api/v1/test/inject-alerts \
  -d '[{"labels": {"alertname": "HighCPU", "instance": "web-02", "severity": "critical"}, "value": 0.98}]'

# Verify: Now 2 groups:
# Group1: HighCPU/warning with web-01, web-03
# Group2: HighCPU/critical with web-02
```

Common Failure Signs:

- Each alert triggers separate notification → Group key calculation incorrect or `group_wait` timer not working
- Alert disappears when labels change → `Grouper.moveAlert()` not implemented or called
- Memory usage grows indefinitely → `Grouper.cleanupGroup()` not called on empty groups

Milestone 3: Silencing & Inhibition Checkpoint

Expected Behavior: Alerts matching silence matchers are suppressed; inhibition rules suppress target alerts when source alerts fire with matching labels.

Verification Commands:

```bash
# Run silencer and inhibitor tests
go test ./internal/silencer/... -v -count=1

# Expected output includes:
# ✓ TestSilenceTimeWindow (active only between StartsAt/EndsAt)
# ✓ TestMatcherCombinations (AND logic for multiple matchers)
# ✓ TestInhibitionLabelMatching (equalLabels constraint)
# ✓ TestInhibitionLoopPrevention (no mutual inhibition)
# ✓ TestSilenceExpirationCleanup (expired silences removed)
```

Manual Verification Steps:

1. **Start with full silencing/inhibition stack**:

```bash
go run cmd/testserver/main.go --config test/fixtures/silence_test_config.yaml
```

2. **Create a silence for maintenance window**:

```bash
curl -X POST http://localhost:8080/api/v1/silences \
  -H "Content-Type: application/json" \
  -d '{
    "matchers": [{"name": "instance", "value": "web-01"}],
    "startsAt": "2024-01-01T10:00:00Z",
    "endsAt": "2024-01-01T12:00:00Z",
    "createdBy": "test",
    "comment": "Maintenance window"
  }'
```

3. **Verify silence suppresses matching alerts**:

```bash
# Trigger alert for web-01
curl -X POST http://localhost:8080/api/v1/test/inject-alerts \
  -d '[{"labels": {"alertname": "HighCPU", "instance": "web-01"}, "value": 0.99}]'

# Check if alert is suppressed
curl http://localhost:8080/api/v1/alerts?state=suppressed | jq .
# Expected: Alert appears in suppressed list, not in firing list
```

4. **Test inhibition rule**:

```bash
# Create inhibition: critical alerts suppress warning alerts for same instance
curl -X POST http://localhost:8080/api/v1/inhibition-rules \
  -d '{
    "sourceMatchers": [{"name": "severity", "value": "critical"}],
    "targetMatchers": [{"name": "severity", "value": "warning"}],
    "equalLabels": ["instance", "service"]
  }'

# Fire critical alert for web-01
curl -X POST http://localhost:8080/api/v1/test/inject-alerts \
  -d '[{"labels": {"alertname": "HighCPU", "instance": "web-01", "severity": "critical"}, "value": 0.99}]'

# Fire warning alert for same instance
curl -X POST http://localhost:8080/api/v1/test/inject-alerts \
  -d '[{"labels": {"alertname": "HighMemory", "instance": "web-01", "severity": "warning"}, "value": 0.90}]'

# Only critical alert should be firing
curl http://localhost:8080/api/v1/alerts?state=firing | jq .
# Expected: Only HighCPU (critical) appears; HighMemory (warning) inhibited
```

5. **Verify silence expiration**:

```bash
# Set system time forward (or use test endpoint to fast-forward)
curl -X POST http://localhost:8080/api/v1/test/fast-forward -d '{"duration": "2h"}'

# Silence should be expired and removed
curl http://localhost:8080/api/v1/silences | jq .
# Expected: Empty array or silence with active: false
```

Common Failure Signs:

- Alerts not suppressed despite matching silence → Matcher logic incorrect or time window check wrong
- Inhibition causes infinite loop → Missing check for mutual inhibition or incorrect `equalLabels` handling
- Expired silences not cleaned up → Background cleanup goroutine not running or not removing from map

Milestone 4: Notification Routing Checkpoint

Expected Behavior: Alerts route to appropriate receivers (Slack, PagerDuty, email) based on label matchers in a routing tree, with rate limiting and retry logic.

Verification Commands:

```bash
# Run router and receiver tests
go test ./internal/router/... -v -count=1

# Expected output includes:
# √ TestRoutingTreeTraversal (correct route selection with continue flag)
# √ TestRateLimiting (notifications throttled per receiver)
# √ TestReceiverRetryLogic (exponential backoff on failure)
# √ TestDefaultRouteCatchAll (unmatched alerts go to default)
# √ TestNotificationFormatting (correct payload for each receiver type)
```

Manual Verification Steps:

1. **Start with full routing stack**:

```bash
go run cmd/testserver/main.go --config test/fixtures/routing_test_config.yaml
```

2. **Examine routing tree configuration**:

```bash
curl http://localhost:8080/api/v1/routes | jq .

# Expected tree structure:
# - Root (default: "email-admin")
#   - severity=critical (receiver: "pagerduty", continue: false)
#   - team=platform (receiver: "slack-platform", continue: true)
#     - severity=warning (receiver: "slack-platform-warnings")
```

3. **Test route matching** with different alerts:

```bash
# Critical alert should go to PagerDuty only
curl -X POST http://localhost:8080/api/v1/test/inject-alerts \
  -d '[{"labels": {"alertname": "DiskFull", "severity": "critical"}, "value": 0.95}]'

# Check which receivers were triggered
grep "Receiver called" /tmp/alerting_test_receiver.log
# Expected: Only PagerDuty receiver spy called

# Team=platform, severity=warning alert should go to both Slack receivers
curl -X POST http://localhost:8080/api/v1/test/inject-alerts \
  -d '[{"labels": {"alertname": "HighCPU", "team": "platform", "severity": "warning"}, "value": 0.85}]'

# Expected: slack-platform AND slack-platform-warnings both called
```

4. **Test rate limiting**:

```bash
# Send 10 alerts rapidly to same receiver
for i in {1..10}; do
  curl -X POST http://localhost:8080/api/v1/test/inject-alerts \
    -d "[{\"labels\": {\"alertname\": \"TestAlert\", \"severity\": \"warning\"}, \"value\": 0.$i}]" &
done

# Check notification log
cat /tmp/alerting_test_notifications.log | grep "slack" | wc -l
# With rate limit of 5/minute, only 5 should be delivered immediately
# Others should be queued or dropped (depending on configuration)
```

5. **Test retry logic with failing receiver**:

```bash
# Configure a receiver to fail
curl -X POST http://localhost:8080/api/v1/test/receivers/fail \
  -d '{"receiver": "slack-platform", "failCount": 3}'

# Send alert
curl -X POST http://localhost:8080/api/v1/test/inject-alerts \
  -d '[{"labels": {"team": "platform", "severity": "warning"}, "value": 0.95}]'

# Check logs for retry attempts with exponential backoff
tail -f /tmp/alerting_test_router.log
# Expected: Attempt 1 (immediate), Attempt 2 (2s delay), Attempt 3 (4s delay)
```

6. **Verify default route catches unmatched alerts**:

```bash
# Alert with no matching labels
curl -X POST http://localhost:8080/api/v1/test/inject-alerts \
  -d '[{"labels": {"alertname": "UnknownAlert", "custom": "label"}, "value": 0.5}]'

# Should go to default email-admin receiver
grep "email-admin" /tmp/alerting_test_receiver.log | tail -1
# Expected: Notification logged for email-admin
```

Common Failure Signs:

- Alerts not routed anywhere (disappear) → Missing default route or routing tree traversal bug
- Alerts sent to all receivers → `continue` flag not respected or route matching too permissive
- Rate limiting too strict blocks all alerts → Token bucket configured with zero capacity or incorrect burst size
- Retry logic causes infinite loops → Missing circuit breaker or max retry limit

End-to-End Integration Checkpoint

After completing all four milestones, run the full integration test to verify the complete pipeline:

```bash
# Run the comprehensive scenario test
go test ./internal/coordinator/coordinator_scenario_test.go -v -count=1 -timeout 2m

# Expected output narrates the complete alert journey:
# √ Setup test environment
# √ Trigger metric threshold breach
# √ Verify alert created and pending
# √ Wait for_duration, alert fires
# √ Verify grouping with group_wait
# √ Verify silence does NOT match
# √ Verify inhibition does NOT apply
# √ Verify routing to correct receiver
# √ Verify notification format and content
# √ Trigger resolution, verify resolved notification
# √ Cleanup, verify no state leakage
```

Testing Best Practices and Pitfalls

> **Key Insight**: Test alerting systems with the same rigor as production code, but remember that the goal is confidence in reliability, not 100% coverage of every improbable edge case.

Test Data Management:

- Use **fixture files** for complex configurations (YAML for rules, JSON for test alerts)
- **Reset state completely** between tests—lingering alerts from previous tests cause flaky failures
- Use **mock clocks** (`time.Now` override) for time-based tests instead of `time.Sleep`

Concurrency Testing:

- Run tests with `-race` flag to detect data races
- Use `sync.WaitGroup` in tests to coordinate goroutines
- Test concurrent alert processing with deliberately jumbled timing

Common Testing Anti-Patterns to Avoid:

Anti-Pattern	Problem	Better Approach
Sleep-based timing	Tests become slow and flaky	Use mock clocks or callback synchronization
Testing internal state directly	Tests break on refactoring	Test through public APIs and observable behavior
Over-mocking	Tests pass but integration fails	Use real components with mocked external dependencies
Ignoring resolution flow	Only tests firing, not cleanup	Always test the complete lifecycle: fire → notify → resolve → cleanup
Hardcoded timing constants	Tests fail on slower CI machines	Use relative timing with generous tolerances

Debugging Test Failures:

When tests fail, follow this diagnostic checklist:

1. **Check test isolation**: Did previous test leave state that affects this one?
2. **Verify timing assumptions**: Are you assuming synchronous behavior that's actually asynchronous?
3. **Examine logs**: Run tests with `-v` and check component logs for unexpected behavior
4. **Simplify the test**: Remove complexity to isolate the failing component
5. **Reproduce locally**: Run the exact same scenario manually via API calls

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Test Framework	Standard `testing` package + `testify/assert`	`ginkgo` + `gomega` for BDD-style tests
Mocking	Manual interface implementations	`gomock` or `mockery` for generated mocks
HTTP Testing	`net/http/httptest` server	`testcontainers` for real service testing
Property Testing	Manual edge case enumeration	`gopter` for automatic property generation
Concurrency Testing	`-race` flag + manual goroutines	`github.com/benbjohnson/clock` for time mocking
Scenario Testing	Sequential Go tests	`cucumber` + Gherkin for business-readable tests

B. Test Utilities Starter Code

Mock Metrics Server (Complete Implementation):

```go
// testutils/mock_metrics.go

package testutils


import (

    "encoding/json"

    "net/http"

    "sync"

    "time"

)


type MockMetricsServer struct {

    mu       sync.RWMutex

    metrics  map[string]float64

    server   *http.Server

}


func NewMockMetricsServer(addr string) *MockMetricsServer {

    m := &MockMetricsServer{

        metrics: make(map[string]float64),

    }


    mux := http.NewServeMux()

    mux.HandleFunc("/api/v1/query", m.handleQuery)

    mux.HandleFunc("/api/v1/test/set-metric", m.handleSetMetric)


    m.server = &http.Server{

        Addr:    addr,

        Handler: mux,

    }


    return m

}


func (m *MockMetricsServer) Start() error {

    return m.server.ListenAndServe()

}


func (m *MockMetricsServer) Stop() error {

    return m.server.Close()

}


func (m *MockMetricsServer) handleQuery(w http.ResponseWriter, r *http.Request) {

    query := r.URL.Query().Get("query")
```

```
    // Parse query to extract metric name
    // For simplicity, assume query like "cpu_usage"

    m.mu.RLock()
    value, exists := m.metrics[query]
    m.mu.RUnlock()

    response := map[string]interface{}{
        "status": "success",
        "data": map[string]interface{}{
            "resultType": "vector",
            "result": []map[string]interface{}{
                {
                    "metric": map[string]string{"__name__": query},
                    "value":  []interface{}{time.Now().Unix(), value},
                },
            },
        },
    }

    if !exists {
        response["status"] = "error"
        response["error"] = "metric not found"
    }

    w.Header().Set("Content-Type", "application/json")
    json.NewEncoder(w).Encode(response)
}

func (m *MockMetricsServer) handleSetMetric(w http.ResponseWriter, r *http.Request) {
    var req struct {
        Metric string  `json:"metric"`
        Value  float64 `json:"value"`
    }

    if err := json.NewDecoder(r.Body).Decode(&req); err != nil {
        http.Error(w, err.Error(), http.StatusBadRequest)
        return
    }

    m.mu.Lock()
```

```
    m.metrics[req.Metric] = req.Value
    m.mu.Unlock()

    w.WriteHeader(http.StatusOK)
}

// Usage in tests:
func TestEngineWithMockMetrics(t *testing.T) {
    mock := NewMockMetricsServer(":9090")
    go mock.Start()
    defer mock.Stop()

    // Set test metric value
    resp, _ := http.Post("http://localhost:9090/api/v1/test/set-metric",
        "application/json",
        strings.NewReader(`{"metric": "cpu_usage", "value": 0.95}`))
    defer resp.Body.Close()

    // Now engine query will return 0.95
}
```

Receiver Spy for Capturing Notifications:

GO

```go
// testutils/receiver_spy.go

package testutils

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

type ReceiverSpy struct {
    mu          sync.RWMutex
    calls       []NotificationCall
    failPattern map[string]int // receiver -> fail count
}

type NotificationCall struct {
    Receiver   string
    Notification interface{}
    Timestamp  time.Time
}

func NewReceiverSpy() *ReceiverSpy {
    return &ReceiverSpy{
        calls:       make([]NotificationCall, 0),
        failPattern: make(map[string]int),
    }
}

func (s *ReceiverSpy) Send(receiver string, notification interface{}) error {
    s.mu.Lock()
    defer s.mu.Unlock()

    // Check if we should fail this call
    if failCount, ok := s.failPattern[receiver]; ok && failCount > 0 {
        s.failPattern[receiver] = failCount - 1
        return fmt.Errorf("simulated failure for receiver %s", receiver)
    }

    s.calls = append(s.calls, NotificationCall{
        Receiver:    receiver,
        Notification: notification,
        Timestamp:   time.Now(),
    })
```

```
    // Log for manual verification
    data, _ := json.MarshalIndent(notification, "", "  ")
    fmt.Printf("[ReceiverSpy] %s received: %s\n", receiver, string(data))

    return nil
}

func (s *ReceiverSpy) GetCalls(receiver string) []NotificationCall {
    s.mu.RLock()
    defer s.mu.RUnlock()

    var result []NotificationCall
    for _, call := range s.calls {
        if call.Receiver == receiver {
            result = append(result, call)
        }
    }
    return result
}

func (s *ReceiverSpy) Reset() {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.calls = make([]NotificationCall, 0)
    s.failPattern = make(map[string]int)
}

func (s *ReceiverSpy) SetFailPattern(receiver string, count int) {
    s.mu.Lock()
    defer s.mu.Unlock()
    s.failPattern[receiver] = count
}
```

C. Core Test Skeleton for Rule Evaluator

```go
// internal/engine/engine_test.go

package engine

import (
    "context"
    "testing"
    "time"

    "github.com/stretchr/testify/assert"
    "github.com/stretchr/testify/require"
)

func TestThresholdEvaluation(t *testing.T) {
    // Setup
    mockClient := &MockMetricsClient{}
    engine := NewEngine([]*Rule{
        {
            Name:        "HighCPU",
            Expression:  "cpu_usage",
            Operator:    ">",
            Threshold:   0.9,
            ForDuration: 30 * time.Second,
        },
    }, mockClient, 15*time.Second)

    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    go engine.Run(ctx)

    // TODO 1: Set mock to return value 0.95 (above threshold)
    // TODO 2: Wait for evaluation cycle
    // TODO 3: Verify alert is in pending state
    // TODO 4: Fast-forward time past for_duration
    // TODO 5: Verify alert transitions to firing
    // TODO 6: Set mock to return value 0.8 (below threshold)
    // TODO 7: Verify alert transitions to resolved
    // TODO 8: Verify resolved alert has EndsAt set
}

func TestForDurationResets(t *testing.T) {
```

```
    // TODO 1: Create rule with for_duration: 60s
    // TODO 2: Set metric above threshold for 45s, then below for 1s, then above again
    // TODO 3: Verify timer resets - alert should NOT fire after 45+1+15=61s total
    // TODO 4: Wait full 60s uninterrupted, verify alert fires
}

func TestTemplateRendering(t *testing.T) {
    rule := &Rule{
        Name:       "TestRule",
        Expression: "error_rate",
        Labels: map[string]string{
            "severity": "critical",
            "instance": "{{ $labels.instance }}",
        },
        Annotations: map[string]string{
            "description": "Error rate is {{ $value }} on {{ $labels.instance }}",
            "summary":     "High error rate",
        },
    }

    // TODO 1: Call Rule.RenderAnnotations with test labels and value
    // TODO 2: Verify instance label is copied from input labels
    // TODO 3: Verify description includes formatted value
    // TODO 4: Test with missing label - should render empty or error
}
```

D. Language-Specific Hints for Go Testing

1. **Use `t.Parallel()` wisely**: Only for independent tests; avoid for tests sharing global state
2. **Clean up resources**: Use `t.Cleanup()` instead of defer in setup functions
3. **Test helpers**: Use `t.Helper()` to mark helper functions, improving failure output
4. **Table-driven tests**: Ideal for testing multiple input combinations:

```go
func TestMatcher(t *testing.T) {
    tests := []struct {
        name     string
        matcher  Matcher
        labels   map[string]string
        expected bool
    }{
        {
            name: "exact match",
            matcher: Matcher{Name: "severity", Value: "critical", IsRegex: false},
            labels:  map[string]string{"severity": "critical"},
            expected: true,
        },
        // ... more cases
    }

    for _, tt := range tests {
        t.Run(tt.name, func(t *testing.T) {
            actual := tt.matcher.Match(tt.labels)
            assert.Equal(t, tt.expected, actual)
        })
    }
}
```

5. **Concurrent map access**: Use `sync.Map` for test fixtures accessed by multiple goroutines
6. **Timeout handling**: Always add timeout to tests that might hang:

```go
func TestSomethingThatMightHang(t *testing.T) {

    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)

    defer cancel()



    done := make(chan bool)

    go func() {

        // test logic

        done <- true

    }()



    select {

    case <-done:

        // test passed

    case <-ctx.Done():

        t.Fatal("test timed out")

    }

}
```

E. Debugging Tips Table

Symptom	Likely Cause	How to Diagnose	Fix
Alert never fires	Threshold comparison wrong, metric query fails, or `for_duration` timer broken	Check engine logs for query results; add debug logging to threshold check; verify timer callback is called	Fix comparison operator logic; handle query errors gracefully; ensure timer goroutine runs
Alert fires immediately	Missing `for_duration` support or duration set to zero	Check rule configuration; trace state transitions (pending should come before firing)	Implement pending state with timer
Duplicate notifications	Group key changes on each evaluation, or `group_wait` timer restarts incorrectly	Log group key calculation; check if labels change between evaluations; trace timer restarts	Ensure stable group key; only restart timers on first alert
Memory grows over time	Groups not cleaned up when empty, or silences never expire	Monitor group count over time; check cleanup conditions; verify silence expiration scanner runs	Implement group cleanup; run background scanner for expired silences
Infinite retry loop	Missing circuit breaker or max retry limit	Log retry attempts with timestamps; check if failure persists	Add circuit breaker pattern with exponential backoff cap
Race condition flaky tests	Concurrent map access without locking, or goroutine timing dependency	Run with `-race` flag; add more synchronization points; use `sync.WaitGroup` in tests	Add appropriate mutex protection; make tests less timing-dependent

9. Debugging Guide

> **Milestone(s):** This section provides diagnostic techniques and troubleshooting procedures for the entire Alerting System, encompassing all four milestones (Rule Evaluation, Alert Grouping, Silencing & Inhibition, Notification Routing). A systematic approach to debugging is critical for maintaining operational reliability and understanding the complex interactions between components.

Debugging a distributed alerting pipeline requires moving beyond simple print statements. The system's state is distributed across multiple components with complex interactions—timers, concurrent goroutines, and state machines. This guide provides a structured approach: first, a comprehensive symptom-cause-fix table for common implementation bugs, followed by techniques for inspecting system state through built-in observability features.

Mental Model: The Hospital Diagnostic Lab

Think of debugging this system like a hospital diagnostic lab processing patient samples. The **symptoms** (e.g., "no notifications") are like a patient's complaints. The **diagnostic tools** (logs, metrics, inspection endpoints) are like blood tests and imaging scans that reveal internal state. The **root causes** are the underlying diseases—often not where the symptom manifests. A rash (symptom) might stem from a liver problem (root cause). Similarly, a missing notification might originate from a misconfigured silence in a different component, not the router. This mental model emphasizes systematic diagnosis: start with the symptom, use tools to trace the data flow backward, and identify the true root cause.

Symptom-Cause-Fix Reference Table

The following table catalogs common symptoms organized by the component where they typically manifest. However, due to the pipeline architecture, the root cause often lies upstream. Each entry includes specific diagnostic steps to isolate the issue.

Symptom	Likely Component	Root Cause	Diagnostic Steps	Fix
Alerts never transition from Pending to Firing	Rule Evaluator	1. **Incorrect `ForDuration` handling**: The `for` duration timer resets whenever the condition briefly becomes false between evaluation cycles. 2. **Flapping metric**: The metric value oscillates around the threshold, preventing sustained breach. 3. **Evaluation interval mismatch**: The rule's `ForDuration` is shorter than the evaluation interval, causing missed checks.	1. Check engine logs for `pending` alerts and their `StartsAt` timestamp. Use `GetRuleState` endpoint (see below) to see if `for` timer resets. 2. Query the metric directly via the metrics API to see historical volatility. 3. Compare `Engine.interval` configuration to rule's `ForDuration`.	1. Implement hysteresis (require value to drop below threshold by a margin before resetting `for` timer). 2. Adjust threshold or use averaging in the alert expression. 3. Ensure evaluation interval ≤ `ForDuration`/2 for reliable detection.
Alerts fire but no notifications arrive	Router (symptom) / Silencer/Grouper (cause)	1. **Silence match**: An active silence matches the alert's labels. 2. **Inhibition**: A higher-severity firing alert inhibits this one. 3. **Orphaned alert in group**: Alert changed grouping labels but remains in old group, never triggering `group_wait` expiry. 4. **Missing default route**: Alert doesn't match any route and is dropped.	1. Call `Silencer.CheckSuppressed(alert)` in debug console or check `/silences` endpoint. 2. Call `Inhibitor.CheckSuppressed(alert)` and list all firing alerts with `GetFiringAlerts()`. 3. Use `Grouper.GetGroupKeys()` and `Grouper.GetGroup(key)` to find if alert's fingerprint exists in any group. Compare its current group key via `calculateGroupKey(alert)`. 4. Check router configuration for a route with no matchers (`Continue=false`).	1. Review silence matchers and expiration times. 2. Adjust inhibition rule `EqualLabels` or source/target matchers. 3. Implement `Grouper.moveAlert()` to handle label changes and migrate alerts between groups. 4. Add a catch-all default route with a receiver (e.g., "default").
Duplicate notifications for the same alert	Router / Grouper	1. **Routing tree `continue=true` misconfiguration**: Alert matches a parent route that sends a notification, then continues to child routes that also match and send. 2. **Group key collision**: Two different sets of labels produce identical group key due to hash collision or incorrect key calculation. 3. **Timer race condition**: `group_interval` timer fires while a new alert is being added, causing partial flush.	1. Examine routing tree configuration. Trace the alert's path through the tree, checking `Continue` flag at each node. 2. Log group key calculation for alerts. Verify that `calculateGroupKey` uses sorted label values and includes all grouping labels. 3. Inspect logs for flush events and alert additions around the same timestamp. Check `timerManager` for proper mutex protection.	1. Set `Continue=false` on routes that should be terminal, or refine matchers to be mutually exclusive. 2. Use a cryptographic hash (SHA-256) of the canonicalized label string for group key. 3. Ensure `Group.UpsertAlert` and timer callbacks are synchronized with a mutex; use atomic flags to prevent re-entrant flush.
Notifications are delayed beyond `group_wait`	Grouper	1. **Timer starvation**: Goroutine managing timers is blocked on a slow operation (e.g., logging, network I/O). 2. **Clock skew**: System clock jumped forward, causing timers to fire late. 3. **Alert state transition after group formed**: Alert enters group in `resolved` state, making `HasActiveAlerts()` false, which suppresses flush.	1. Check goroutine dump (`pprof`) for blocking calls in timer callback path. 2. Compare system time with NTP source; check for large jumps in logs. 3. Inspect group contents via `GetGroup()` : count of alerts vs. active alerts.	1. Ensure timer callbacks are non-blocking (push to buffered channel for async processing). 2. Use `time.Timer` with monotonic clock where available; monitor system time changes. 3. Modify `HasActiveAlerts()` to return true if group has *any* alerts (not just active) during `group_wait` , or implement separate logic for resolved alerts.
Silences do not suppress expected alerts	Silencer	1. **Matcher syntax error**: Silence uses regex matcher `key=~value` but `value` is not a valid regex.	1. Validate silence matchers at creation time with `ParseMatcher()` and test with sample labels. 2. Check silence activation via `/silences` endpoint; ensure times are UTC.	1. Add validation in `Silencer.AddSilence()` to reject invalid regex patterns. 2. Store and compare all times in

Symptom	Likely Component	Root Cause	Diagnostic Steps	Fix
		2. **Time window misalignment**: Silence `StartsAt` is in future or `EndsAt` is in past due to timezone confusion. 3. **Label missing**: Alert lacks a label that the silence matcher expects, causing match to fail.	3. Compare alert labels with silence matchers; note that missing labels never match equality or regex matchers.	UTC; provide UI clarity on timezone. 3. Consider using negative matchers (`key!=value`) or ensure alerts have consistent label schema.
Inhibition causes unexpected suppression chain (cascade)	Inhibitor	1. **Inhibition loop**: Alert A inhibits alert B, and alert B (through another rule) inhibits alert A, causing mutual suppression. 2. **Overly broad `EqualLabels`** : Inhibition requires matching on too many labels, causing unrelated alerts to inhibit each other. 3. **Stale firing alerts**: Source alert is resolved but inhibition cache hasn't been updated.	1. Draw inhibition rule graph: nodes are alerts, edges are inhibition rules. Check for cycles. 2. Log `EqualLabels` matching; see if labels like `cluster` or `alertname` are correctly scoped. 3. Verify `alertsGetter` returns only currently firing alerts; check inhibition cache TTL.	1. Implement cycle detection when adding inhibition rules. 2. Limit `EqualLabels` to only the labels that truly define the dependency (e.g., `[cluster, service]`). 3. Ensure `Inhibitor` polls `alertsGetter` periodically or subscribes to alert state changes.
Memory usage grows indefinitely	Grouper / Silencer	1. **Memory leak in groups**: Empty groups are not cleaned up after all alerts resolve. 2. **Silence retention**: Expired silences are not removed from memory. 3. **Alert history accumulation**: Resolved alerts are kept indefinitely in engine state map.	1. Monitor `Grouper.groups` map size over time. Check if `cleanupGroup()` is called when `HasActiveAlerts()` becomes false. 2. Check `Silencer.byExpiry` slice; ensure periodic cleanup runs. 3. Inspect `Engine.state` map; check if resolved alerts are pruned after notification.	1. Implement `Grouper.cleanupGroup()` to delete group after `Retention` period elapses and group is empty. 2. Run a goroutine in `Silencer` that periodically removes expired silences from the map. 3. Implement TTL for resolved alerts in engine state, or move them to a separate archive.
Notification storm to a single receiver	Router	1. **Rate limiter misconfigured**: Token bucket rate or burst capacity is too high. 2. **Circuit breaker stuck open**: Receiver failures cause circuit to open, then when it half-opens, a backlog of notifications floods it. 3. **Grouping disabled**: `GroupBy` is empty, causing each alert to be its own group and bypassing `group_wait` batching.	1. Check `RateLimiter` configuration for the receiver: `limit` (events/sec) and `burst` . 2. Examine `ReceiverCircuitBreaker` state logs; check failure counts and reset timeout. 3. Verify `GroupBy` list is not empty in router configuration for the route.	1. Adjust rate limiter to appropriate values (e.g., 1 notification per 30s for critical, 1 per 5min for warning). 2. Implement exponential backoff for circuit breaker reset timeout; queue notifications while open. 3. Set `GroupBy` to at least `[alertname]` or `[service]` to enable batching.
Template rendering produces empty annotations	Rule Evaluator	1. **Missing template data**: Template expects label `{{ .Labels.instance }}` but alert has no `instance` label. 2. **Syntax error in template**: Template contains `{{ .Value	printf "%.2f" }}` but `.Value` is not a float64. 3. **Caching bug**: Template compilation error causes cached nil template, silently failing render.	1. Check template content and alert labels; use debug endpoint to render template with given data. 2. Validate template with `template.Parse()` at rule load time; check `TemplateCache` logs. 3. Inspect `TemplateCache.templates` map; ensure errors are stored and retried.
Alert appears in multiple groups simultaneously	Grouper	1. **Race condition in `moveAlert`** : Alert is being processed while its labels change, causing it to be added to new group before removal	1. Check logs for `moveAlert` calls; look for duplicate `fingerprintToKey` entries. 2. Verify `Alert.Fingerprint()` uses a hash of *all* labels (including `alertname`).	1. Make `moveAlert` atomic: remove from old group, update `fingerprintToKey` , add to new group under same mutex lock. 2. Ensure fingerprint includes all label

Symptom	Likely Component	Root Cause	Diagnostic Steps	Fix
		from old. 2. **Fingerprint collision**: Two distinct alerts have identical label sets (same fingerprint) but different internal IDs, causing duplication. 3. **Concurrent `Process` calls**: Two goroutines call `Grouper.Process()` for same alert simultaneously before group key is stabilized.	3. Add mutex to `Grouper.Process()` or use channel serialization.	key-value pairs (sorted). 3. Protect `Grouper.Process()` with mutex or process alerts via a single goroutine channel.

Techniques for Inspecting System State

Beyond reacting to symptoms, you need proactive tools to examine the system's internal state. The following techniques should be built into the implementation.

1. Structured Logging with Diagnostic Levels

Implement a structured logging library (e.g., `slog` in Go) with log levels that can be adjusted at runtime. Key events to log:

Log Event	Component	Fields to Include	Level
`rule_evaluation`	Engine	`rule`, `result`, `new_state`, `for_remaining`	INFO
`alert_state_change`	Engine	`alert_fingerprint`, `old_state`, `new_state`, `labels`	INFO
`group_created`	Grouper	`group_key`, `group_by_labels`, `first_alert`	DEBUG
`group_flush`	Grouper	`group_key`, `alert_count`, `active_count`, `reason` (timer, new alert)	INFO
`silence_matched`	Silencer	`alert_fingerprint`, `silence_id`, `matchers`	DEBUG
`inhibition_matched`	Inhibitor	`target_alert`, `source_alert`, `rule_id`	DEBUG
`route_matched`	Router	`alert_fingerprints`, `receiver`, `route_matchers`	DEBUG
`notification_sent`	Router	`receiver`, `notification_id`, `alert_count`, `success`	INFO
`notification_failed`	Router	`receiver`, `error`, `retry_count`, `circuit_state`	WARN
`timer_fired`	Grouper	`timer_type` (group_wait, group_interval), `group_key`	DEBUG

> **Design Insight:** Structured logging transforms logs from human-readable text to machine-queryable data. You can query logs for all alerts that matched a specific silence, or calculate the average time between alert firing and notification.

2. HTTP Inspection Endpoints

Embed a lightweight HTTP server (on a configured admin port) exposing read-only diagnostic endpoints. Protect these endpoints with authentication in production.

Endpoint Path	Component	Returns	Purpose
`GET /debug/rules`	Engine	List of all rules with current evaluation state (last result, next evaluation time).	Verify rule configuration and evaluation status.
`GET /debug/rules/{name}/state`	Engine	Map of fingerprint to Alert for all alerts (pending/firing/resolved) for that rule.	Inspect individual rule's alert instances.
`GET /debug/groups`	Grouper	List of active group keys with alert counts and timer statuses.	See how alerts are grouped and if groups are stuck.
`GET /debug/groups/{key}`	Grouper	Detailed group info: alerts list, timers (next fire), created time.	Examine composition of a specific group.
`GET /debug/silences`	Silencer	All silences (active, pending, expired).	Verify silence coverage and timing.
`GET /debug/inhibitions`	Inhibitor	Active inhibition rules and currently inhibited alerts.	Understand suppression dependencies.
`GET /debug/routing-tree`	Router	The configured routing tree with matchers and receivers.	Debug route matching logic.
`GET /debug/receivers`	Router	Receiver configurations and status (circuit breaker, rate limiter).	Check receiver health and throttling.
`POST /debug/render`	Engine	Renders a template with provided JSON data (labels, value).	Test template rendering without creating alerts.
`GET /health`	All	`HealthStatus` map showing component health (e.g., "engine": "ok", "grouper": "degraded").	Overall system health for load balancers.

3. Metrics for Internal Observability

Instrument each component to expose Prometheus metrics. These metrics allow proactive detection of issues.

Metric Name	Type	Labels	Description
`alerting_alerts`	Gauge	`state` (pending, firing, resolved), `rule`	Number of alerts in each state per rule.
`alerting_rule_evaluations_total`	Counter	`rule`, `result` (success, error)	Total rule evaluation attempts.
`alerting_rule_evaluation_duration_seconds`	Histogram	`rule`	Duration of rule evaluation queries.
`alerting_groups`	Gauge	`group_by` (label keys)	Number of active groups.
`alerting_group_alerts`	Gauge	`group_key`	Number of alerts in a specific group.
`alerting_silences`	Gauge	`state` (active, pending, expired)	Number of silences.
`alerting_inhibitions_active`	Gauge	`rule_id`	Number of active inhibition suppressions.
`alerting_notifications_total`	Counter	`receiver`, `outcome` (success, error)	Total notifications sent.
`alerting_notification_duration_seconds`	Histogram	`receiver`	Duration of sending notification.
`alerting_rate_limit_dropped_total`	Counter	`receiver`	Notifications dropped due to rate limiting.
`alerting_circuit_breaker_state`	Gauge	`receiver`, `state` (closed, open, half-open)	Current circuit breaker state.

4. Interactive Debug Console (Optional)

For deep debugging, provide a simple REPL (Read-Eval-Print Loop) that can be attached via a socket or interactive HTTP endpoint. This console can execute predefined debug functions in the running process context:

```
> debug.GetAlert("abcd1234")
Alert{
  Labels: {alertname="HighCPU", instance="web-01"},
  State: "firing",
  GroupKey: "alertname=HighCPU,instance=web-01"
}
> debug.CheckSuppressed("abcd1234")
Silence matched: silence_id="maint-123", comment="Weekly maintenance"
```

PLAINTEXT

Implementation Guidance

A. Technology Recommendations Table

Component	Simple Option	Advanced Option
Logging	`log/slog` with JSON output	Structured logging with OpenTelemetry SDK (logs, traces, metrics correlation)
HTTP Endpoints	`net/http` with `/debug` routes	Dedicated admin server with mTLS authentication and role-based access
Metrics	`prometheus/client_golang` library	OpenTelemetry Metrics exported to multiple backends
Debug Console	Simple HTTP POST endpoint that evaluates scripts in sandbox	Built-in REPL with secure channel (SSH or WebSocket)

B. Recommended File/Module Structure

```
project-root/
  cmd/alerting/
    main.go                         # Entry point, starts Coordinator
  internal/
    alert/                          # Data models (Alert, Rule, etc.)
      types.go
      fingerprint.go                # Alert.Fingerprint() implementation
    engine/                         # Rule Evaluator (Milestone 1)
      engine.go                     # Engine type and main loop
      evaluator.go                  # Threshold evaluation logic
      state_manager.go              # Manages alert state map
      debug.go                      # /debug/rules endpoint handlers
    grouper/                        # Grouper (Milestone 2)
      grouper.go                    # Grouper type and Process method
      group.go                      # Group type and methods
      timers.go                     # timerManager implementation
      debug.go                      # /debug/groups endpoints
    silencer/                       # Silencer & Inhibitor (Milestone 3)
      silencer.go                   # Silencer type and methods
      inhibitor.go                  # Inhibitor type and methods
      matcher.go                    # Matcher parsing and matching
      debug.go                      # /debug/silences, /debug/inhibitions
    router/                         # Notification Router (Milestone 4)
      router.go                     # Router type and Route method
      receivers/                    # Receiver implementations
        slack.go
        pagerduty.go
        webhook.go
      rate_limiter.go               # RateLimiter implementation
      circuit_breaker.go            # ReceiverCircuitBreaker
      debug.go                      # /debug/routing-tree, /debug/receivers
    coordinator/                    # Orchestrates pipeline
      coordinator.go                # Coordinator type and Run
      pipeline.go                   # processAlerts, processAlertThroughPipeline
    debug/                          # Shared debugging utilities
      console.go                    # Interactive debug REPL
      health.go                     # HealthStatus implementation
    metrics/                        # Internal metrics collection
      internal_metrics.go           # Registers and exposes Prometheus metrics
    api/                            # External API (optional)
      server.go
```

C. Infrastructure Starter Code: Health Status Endpoint

Below is a complete, ready-to-use implementation of the `HealthStatus` type that can be used across components.

```go
// internal/debug/health.go

package debug


import (

    "encoding/json"

    "net/http"

    "sync"

    "time"

)


// HealthStatus aggregates health status from components.

type HealthStatus struct {

    mu     sync.RWMutex

    status map[string]string // component -> status ( "ok", "degraded", "error" )

}


// NewHealthStatus creates a new HealthStatus.

func NewHealthStatus() *HealthStatus {

    return &HealthStatus{

        status: make(map[string]string),

    }

}


// SetComponent updates the health status for a component.

func (h *HealthStatus) SetComponent(component, status string) {

    h.mu.Lock()

    defer h.mu.Unlock()

    h.status[component] = status

}


// GetComponent returns the current health status for a component.

func (h *HealthStatus) GetComponent(component string) string {

    h.mu.RLock()

    defer h.mu.RUnlock()

    return h.status[component]

}


// ServeHTTP implements http.Handler for /health endpoint.

func (h *HealthStatus) ServeHTTP(w http.ResponseWriter, r *http.Request) {

    h.mu.RLock()

    statusCopy := make(map[string]string, len(h.status))

    for k, v := range h.status {

        statusCopy[k] = v
```

```
    }
    h.mu.RUnlock()

    // Determine overall status: if any component is "error", overall is "error".
    // If any "degraded", overall is "degraded", else "ok".
    overall := "ok"
    for _, s := range statusCopy {
        if s == "error" {
            overall = "error"
            break
        }
        if s == "degraded" && overall != "error" {
            overall = "degraded"
        }
    }

    w.Header().Set("Content-Type", "application/json")
    resp := map[string]interface{}{
        "status":   overall,
        "time":     time.Now().UTC().Format(time.RFC3339),
        "services": statusCopy,
    }
    json.NewEncoder(w).Encode(resp)
}
```

D. Core Logic Skeleton Code: Diagnostic Rule Evaluation Endpoint

Implement a debug endpoint that shows the current state of a rule and its alerts.

```go
// internal/engine/debug.go

package engine


import (

    "encoding/json"

    "net/http"

    "sync"

)


// DebugServer attaches debug endpoints to an Engine.

type DebugServer struct {

    engine *Engine

    mu     sync.RWMutex

}


// RegisterHandlers registers HTTP handlers for debug endpoints.

func (d *DebugServer) RegisterHandlers(mux *http.ServeMux) {

    mux.HandleFunc("/debug/rules", d.handleListRules)

    mux.HandleFunc("/debug/rules/{name}/state", d.handleRuleState)

}


func (d *DebugServer) handleListRules(w http.ResponseWriter, r *http.Request) {

    d.mu.RLock()

    defer d.mu.RUnlock()


    rulesInfo := make([]map[string]interface{}, 0, len(d.engine.rules))

    for _, rule := range d.engine.rules {

        // TODO 1: For each rule, collect: name, expression, for duration, labels, annotations.

        // TODO 2: Add last evaluation time and result (if available) from engine's internal state.

        // TODO 3: Append to rulesInfo slice.

    }

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

    json.NewEncoder(w).Encode(rulesInfo)

}


func (d *DebugServer) handleRuleState(w http.ResponseWriter, r *http.Request) {

    // TODO 1: Extract rule name from URL path (using gorilla/mux or httprouter).

    // TODO 2: Look up rule by name in engine.rules.

    // TODO 3: Acquire read lock on engine.stateMu.

    // TODO 4: Retrieve state map for this rule from engine.state[rule.Name].

    // TODO 5: Convert map of fingerprint->Alert to JSON, include alert details (labels, state, timestamps).

    // TODO 6: Write JSON response.

}
```

E. Language-Specific Hints

- **Structured Logging with `slog`** : Use `slog.NewJSONHandler` to output logs as JSON. Attach a component field to every log entry: `slog.With("component", "engine")`.
- **Concurrency in Debug Endpoints**: Use `sync.RWMutex` to protect internal state when reading for debug endpoints. Avoid blocking writes (like rule evaluation) on debug reads.
- **Prometheus Metrics**: Register metrics in `init()` function of each component package. Use `promauto` package for automatic registration and lifecycle management.
- **Safe Template Rendering for Debug**: When implementing the `/debug/render` endpoint, sanitize user input and limit execution time to prevent denial-of-service via infinite loops in templates.

F. Milestone Checkpoint: Debugging Capabilities

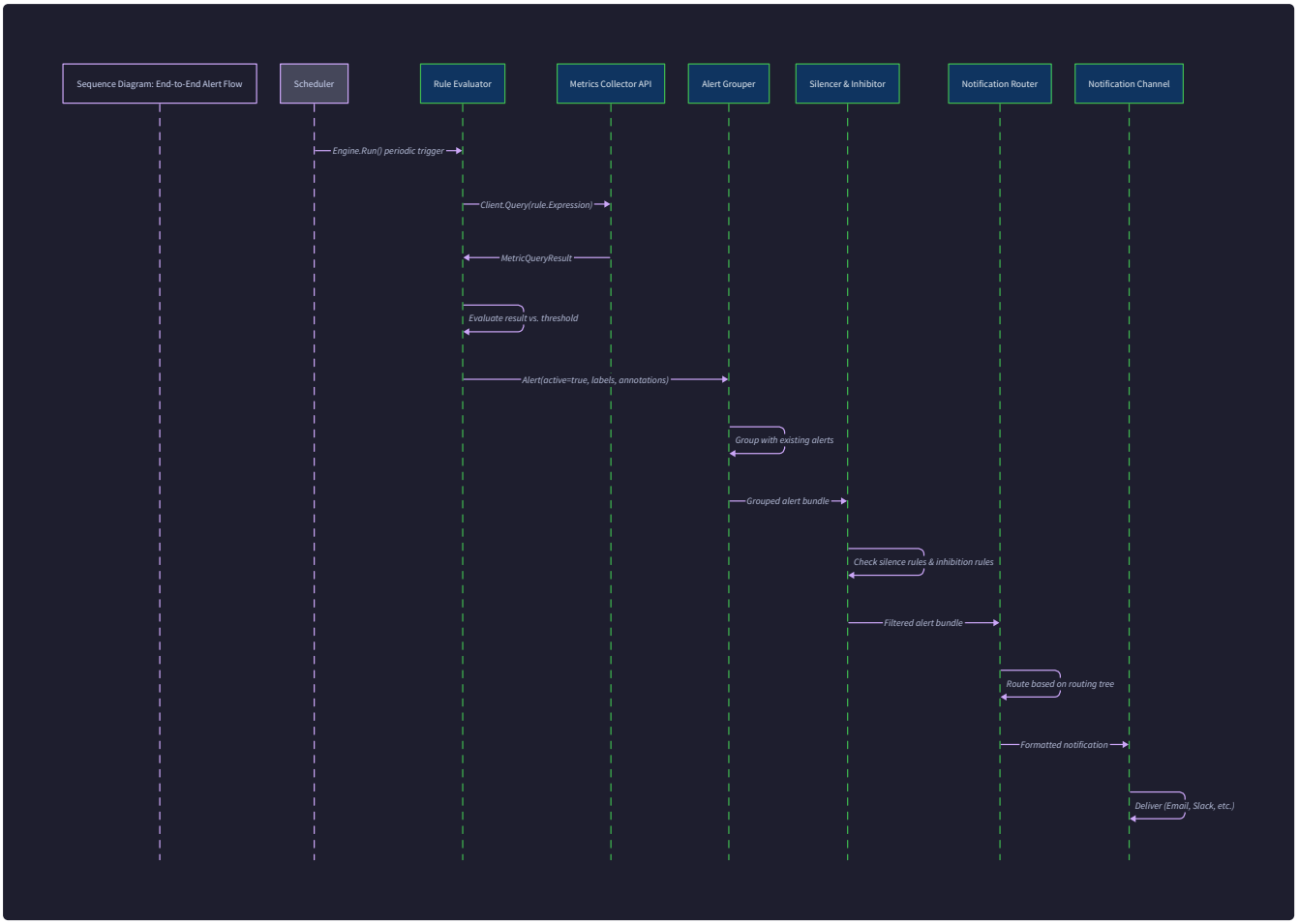
After implementing each milestone, verify the debugging tools work:

1. **Milestone 1 (Rule Evaluator)**: Start the engine with a sample rule. Query `/debug/rules` and see the rule listed with its configuration. Trigger the alert condition, then check `/debug/rules/{name}/state` to see the alert in `pending` state.
2. **Milestone 2 (Grouper)**: Send a few alerts to the grouper via the coordinator. Query `/debug/groups` to see groups formed. Verify that groups have correct alert counts and that timers are running.
3. **Milestone 3 (Silencer & Inhibitor)**: Create a silence via API. Send an alert that matches the silence. Check `/debug/silences` to see the silence as active. Verify the alert does not produce a notification (check logs for `silence_matched`).
4. **Milestone 4 (Notification Router)**: Configure a route to Slack. Trigger an alert. Check `/debug/receivers` to see the receiver status. Verify logs show `notification_sent` with success. Intentionally break the Slack webhook URL and see the circuit breaker transition to `open` state in logs and metrics.

G. Debugging Tips: Structured Approach

When facing an issue, follow this systematic diagnostic workflow:

1. **Check Health Endpoint**: `curl http://localhost:9094/health` to see if any component reports `degraded` or `error`.
2. **Examine Logs**: Filter logs for the alert's fingerprint or rule name. Look for state transitions and component hand-offs.
3. **Inspect Internal State**: Use debug endpoints to see where the alert resides (which rule state, which group, matched silences, etc.).
4. **Trace Data Flow**: Follow the alert's journey using the sequence diagram



as a map. At each step, verify the expected output.

5. **Isolate Component**: If the alert stops at a component (e.g., no notification), test that component in isolation by injecting a test alert directly and observing its output.

10. Future Extensions

> **Milestone(s):** This section explores potential enhancements that could be built upon the foundational Alerting System described in the previous sections. While not part of the current project scope, these extensions represent natural evolution paths that address common production requirements and advanced use cases.

The core system provides robust alerting capabilities, but real-world operational needs often demand additional sophistication. This section outlines potential future extensions that could transform the system from a competent alert manager to a comprehensive observability orchestration platform. Each extension includes mental models for intuition, detailed technical considerations, and architecture decision records for key design choices.

10.1 High Availability Deployment

Mental Model: The Redundant Control Room Imagine having two identical control rooms monitoring the same factory, with shared communication channels and synchronized dashboards. If one control room loses power or suffers equipment failure, the other immediately takes over without missing a single sensor alert. The transition is seamless because both rooms maintain identical understanding of the current state through continuous synchronization.

High availability (HA) ensures the alerting system remains operational even during individual component failures. The single-node architecture described in previous sections works well for small to medium deployments, but production environments at scale require fault tolerance.

Design Considerations for HA:

1. **Shared State Synchronization:** The most challenging aspect is maintaining consistent state (active alerts, silences, inhibition rules) across multiple replicas. When one node receives an alert from the metrics collector, all nodes must agree on whether to trigger notifications.
2. **Leadership Election:** Only one node should evaluate rules at a time to avoid duplicate alert generation. Similarly, only one node should send notifications for each group to prevent notification duplication.
3. **Data Persistence:** State must survive node restarts. If a node crashes and comes back online, it must recover its previous state to avoid missing alerts or sending incorrect resolved notifications.
4. **Client Distribution:** External systems (metrics collectors, webhook senders) must be able to discover and fail over between available nodes.

Decision: State Synchronization Strategy

> **Decision: CRDT-based State Synchronization over Centralized Database**
>
> - **Context**: Multiple alerting nodes need consistent view of alert state, silences, and groupings. Traditional primary-secondary replication introduces single points of failure and complex failover procedures.
> - **Options Considered**:
>   1. **Centralized Database Backend**: All nodes read/write state to a shared database (PostgreSQL, etcd). Provides strong consistency but introduces external dependency and potential latency.
>   2. **Primary-Secondary with Log Shipping**: One active node processes everything; secondaries replicate WAL and take over if primary fails. Simple but has failover delay and wasted secondary capacity.
>   3. **Conflict-Free Replicated Data Types (CRDTs)**: Each node maintains its own state replica; changes merge automatically using mathematical properties that guarantee eventual consistency without coordination.
> - **Decision**: Use CRDTs for alert state and operational data (silences, inhibitions) with periodic anti-entropy synchronization.
> - **Rationale**: CRDTs provide partition tolerance and continuous availability—nodes can operate independently during network partitions. The alerting domain has natural idempotency properties (duplicate alerts merge, later silences override earlier ones) that map well to CRDT semantics. This avoids external dependencies and provides linear scalability.
> - **Consequences**: Eventual consistency means brief periods where nodes may have slightly different views; this is acceptable for alerting where seconds of inconsistency rarely matter. Implementation complexity increases due to need for vector clocks or version vectors.

Option	Pros	Cons	Suitability
Centralized Database	Strong consistency, well-understood patterns	Single point of failure, operational overhead, latency	Environments with existing HA database infrastructure
Primary-Secondary	Simple to implement, clear failure semantics	Wasted resources, failover delay, brain split risk	Small deployments with infrequent node failures
CRDTs	No single point of failure, partition tolerant, scales horizontally	Eventual consistency, implementation complexity, larger state transmission	Cloud-native deployments needing maximum availability

Implementation Approach:

1. **Node Discovery and Membership**: Use gossip protocol or service discovery (Consul, etcd) for nodes to find each other. Each node maintains a membership list and detects node failures via heartbeat.
2. **State Synchronization**:
   - Each node maintains a version vector tracking its knowledge of each data item (alert, silence, etc.)
   - When nodes communicate, they exchange missing updates based on version comparisons
   - Merge conflicts resolved using Last-Write-Wins with tie-breaking by node ID
3. **Rule Evaluation Coordination**:
   - Use distributed lease (via etcd or similar) to elect a leader for rule evaluation
   - Leader evaluates all rules and broadcasts resulting alerts to all nodes
   - If leader fails, new election occurs; brief evaluation gap acceptable
4. **Notification Deduplication**:
   - Each notification group assigned to a specific node via consistent hashing of `GroupKey`
   - Only the responsible node sends notifications for that group
   - If node fails, its groups reassigned via consistent hashing ring

Data Structures for HA:

Type Name	Fields	Description
`ClusterNode`	`ID string`, `Addr string`, `Status string`, `LastSeen time.Time`	Represents a node in the HA cluster
`VersionedAlert`	`Alert Alert`, `Version VectorClock`, `NodeID string`, `Timestamp time.Time`	Alert with version metadata for conflict resolution
`VectorClock`	`Counters map[string]uint64`	Map of node ID to counter, representing knowledge state
`SyncMessage`	`Type string`, `NodeID string`, `Payload []byte`, `VectorClock VectorClock`	Message exchanged between nodes for state synchronization

Common Pitfalls in HA Implementation:

⚠️ **Pitfall: Split-Brain Notification Storms** When network partition occurs and both sides believe they're the active leader, both may evaluate rules and send duplicate notifications.
Why it's wrong: Recipients receive identical alerts multiple times, causing confusion and potentially missing real issues in the noise.
How to fix: Implement fencing tokens or tie-breaking logic (e.g., only nodes with odd-numbered IDs can send during partitions).

⚠️ **Pitfall: Inconsistent Silence Application** Node A applies a silence, but Node B doesn't receive it before processing an alert, resulting in unsuppressed notification.
Why it's wrong: Silences become unreliable during network issues, undermining maintenance windows.
How to fix: Use CRDTs with "remove-wins" semantics for silences and include silence matching in notification deduplication logic.

10.2 Advanced Alert Correlation and AI-Driven Triage

Mental Model: The Expert Diagnostician Imagine not just a control room that displays gauges, but a senior engineer who watches patterns across all instruments, recognizes that "when gauge X shows this pattern and valve Y is in that position, it usually means problem Z is developing." This expert connects seemingly unrelated signals, identifies root causes, and suggests specific remediation steps before operators even recognize there's an issue.

Modern alerting systems often bombard teams with symptom alerts rather than root cause identification. Advanced correlation can reduce noise by grouping related alerts and suggesting probable causes.

Correlation Approaches:

1. **Temporal Correlation**: Alerts occurring within a time window likely share common cause.
2. **Topological Correlation**: Alerts from services in the same dependency chain (e.g., database → API → frontend).
3. **Statistical Correlation**: Machine learning models that learn which alerts typically fire together.
4. **Causal Inference**: Using service mesh or dependency graph to infer root cause from symptom propagation.

Decision: Layered Correlation Strategy

Decision: Three-Layer Correlation Engine with Pluggable Analyzers

- **Context**: Need to reduce alert noise while preserving signal. Different environments have different correlation needs (temporal vs. topological).
- **Options Considered**:
  1. **Fixed Rule-based Correlation**: Predefined rules (e.g., "alerts with same `cluster` label within 60 seconds are correlated"). Simple but inflexible.
  2. **Machine Learning Only**: Train models on historical alert data to find patterns. Powerful but requires large datasets and may produce unexplainable correlations.
  3. **Layered Hybrid Approach**: Multiple correlation strategies applied in sequence, each feeding into the next, with human-overridable results.
- **Decision**: Implement three layers: 1) Rule-based temporal/topological, 2) Statistical anomaly detection, 3) Graph-based causal inference.
- **Rationale**: Layered approach provides immediate value with simple rules while allowing gradual adoption of more sophisticated techniques. Each layer's results are explainable (unlike black-box ML). Operators can see which layer produced which correlation and adjust confidence accordingly.
- **Consequences**: Increased computational complexity; need to design clean interfaces between layers; correlation results become part of alert context that must be propagated through the pipeline.

Correlation Layer	Inputs	Outputs	Implementation Complexity
Rule-based	Alert labels, timestamps, static topology	Correlation groups, suggested root cause	Low (extend existing grouping)
Statistical	Historical alert patterns, firing frequencies	Anomaly scores, unusual co-occurrences	Medium (time series analysis)
Causal Inference	Real-time dependency graph, failure propagation models	Probable root cause node, impact prediction	High (graph algorithms)

Data Enrichment Pipeline:

1. **Alert Enrichment Stage**: After rule evaluation but before grouping, enrich each alert with:
   - Topological context (upstream/downstream services)
   - Historical firing patterns (is this alert unusual for this time?)
   - Similar active alerts (temporal neighbors)
2. **Correlation Engine**: Process enriched alerts through correlation layers:

```
Raw Alert → Enrichment → Rule Correlation → Statistical Analysis → Causal Inference → Correlated Alert Set
```
PLAINTEXT

3. **Notification Enhancement**: Include correlation insights in notifications:
   - "This alert typically occurs with [X, Y, Z] alerts"
   - "Probable root cause: database cluster us-east-1"
   - "Suggested action: check connection pool settings"

Data Structures for Correlation:

Type Name	Fields	Description
`EnrichedAlert`	`Alert Alert`, `Topology []string`, `HistoricalFrequency float64`, `SimilarAlerts []string`	Alert augmented with correlation context
`CorrelationRule`	`Name string`, `Matchers []Matcher`, `TimeWindow time.Duration`, `GroupBy []string`, `OutputAnnotations map[string]string`	Rule defining how to correlate alerts
`CorrelationResult`	`RootCauseAlertID string`, `Confidence float64`, `CorrelatedAlertIDs []string`, `Evidence []string`	Result of correlation analysis
`DependencyGraph`	`Nodes map[string]Node`, `Edges map[string][]Edge`	Service dependency topology for causal analysis

Integration with Existing Pipeline: The correlation engine would sit between the Rule Evaluator and Grouper. Instead of `Coordinator.processAlertThroughPipeline` sending alerts directly to the grouper, it would first send them through the correlation engine, which might:

1. Modify alert annotations with correlation insights
2. Create new "meta-alerts" representing correlated groups
3. Suppress individual alerts in favor of correlated summary alerts

Common Pitfalls in Correlation:

⚠️ **Pitfall: Correlation Creating False Root Causes** Statistical correlation might identify coincidental patterns as causal relationships, leading operators down wrong troubleshooting paths.

Why it's wrong: Wastes investigation time and may cause operators to ignore real issues.
How to fix: Include confidence scores, require minimum support thresholds, and allow operators to flag incorrect correlations for model retraining.

⚠️ **Pitfall: Correlation Delay Masking Urgent Alerts** Waiting for correlation analysis (especially ML inference) could delay critical alerts by seconds.
Why it's wrong: For P0 alerts, every second matters.
How to fix: Implement fast-path for high-severity alerts (bypass correlation), or run correlation asynchronously and send initial notification immediately with follow-up correlation insights.

10.3 Predictive Alerting and Forecasting

Mental Model: The Weather Forecaster Imagine a control room that not only alerts when a pipe bursts, but predicts which pipes are likely to burst based on pressure trends, material fatigue models, and historical failure data. This forecaster provides lead time for preventive maintenance rather than just reacting to failures.

Predictive alerting uses historical metrics and trends to forecast future problems before threshold breaches occur. This shifts operations from reactive to proactive.

Prediction Techniques:

1. **Time Series Forecasting**: ARIMA, Prophet, or LSTM models to predict metric values.
2. **Anomaly Detection**: Statistical models to identify deviations from normal patterns.
3. **Capacity Planning Integration**: Predict when resources will be exhausted based on consumption trends.
4. **Failure Probability Models**: Bayesian networks estimating component failure likelihoods.

Decision: Dual-Threshold Prediction System

> **Decision: Combine Statistical Forecasting with Rule-based Predictions**
>
> - **Context**: Need reliable predictions with explainable reasoning. Pure ML forecasts can be opaque and untrustworthy for critical infrastructure.
> - **Options Considered**:
>   1. **ML-Only Predictions**: Train models on all metrics, use predicted values for alerting. Maximally flexible but requires significant data science expertise.
>   2. **Rule-Based Trends**: Simple linear extrapolation ("if memory grows at X GB/day, will exhaust in Y days"). Simple but misses non-linear patterns.
>   3. **Hybrid Approach**: Use statistical methods to detect anomalies, rule-based methods to extrapolate trends, and combine results with confidence weighting.
> - **Decision**: Implement prediction as a separate rule type with statistical forecasting backend and trend analysis fallback.
> - **Rationale**: Operators need to understand why a prediction was made to trust it. Hybrid approach provides multiple evidence sources and allows gradual improvement of statistical models while maintaining understandable rule-based predictions.
> - **Consequences**: More complex rule evaluation engine; need to store historical metrics for forecasting; prediction accuracy depends on quality of historical data.

Prediction Method	Use Case	Implementation	Output
Linear Extrapolation	Resource exhaustion (disk, memory)	Simple slope calculation	"Will exhaust in X hours at current rate"
Seasonal ARIMA	Periodic workload patterns	Statistical model fitting	"Expected to breach threshold in 2 cycles"
Anomaly Detection	Unusual behavior without clear threshold	Statistical deviation scoring	"Behavior is 3σ from normal pattern"
Survival Analysis	Component failure prediction	Weibull distribution fitting	"90% probability of failure within 30 days"

Prediction Rule Extension: Extend the `Rule` data structure to support predictive rules:

Field	Type	Description
`PredictionMethod`	`string`	"linear", "arima", "anomaly", or "survival"
`ForecastHorizon`	`time.Duration`	How far ahead to predict
`ConfidenceThreshold`	`float64`	Minimum confidence to trigger alert
`TrainingDataDuration`	`time.Duration`	How much historical data to use

Prediction Pipeline:

1. **Data Collection**: Store historical metric values (beyond what metrics collector retains).
2. **Model Training**: Periodically train prediction models based on rule configuration.
3. **Forecast Evaluation**: During rule evaluation cycle, generate predictions and compare to thresholds.
4. **Alert Generation**: Create alerts with special annotations indicating prediction details:
   - `predicted_value`: Forecasted metric value
   - `confidence`: Statistical confidence (0.0-1.0)

- `time_until_breach`: Estimated time until threshold breach
- `recommendation`: Suggested preventive action

Example Predictive Alert Annotation:

```
High memory usage predicted in 8 hours
Current: 4.2GB used of 8GB total (52%)
Trend: Increasing at 0.5GB/hour
Prediction: Will reach 90% threshold (7.2GB) in 6 hours
Confidence: 85%
Recommended action: Scale up memory or reduce cache size
```

Integration Challenges:

1. **Data Retention**: Need to store days/weeks of historical metrics for training.
2. **Model Performance**: Statistical model training can be computationally expensive.
3. **Explainability**: Operators must understand why prediction was made to trust it.
4. **Feedback Loop**: When predictions are correct/incorrect, use this to improve models.

Common Pitfalls in Predictive Alerting:

⚠️ **Pitfall: Cry Wolf with False Predictions** Overly sensitive predictions trigger many alerts that don't materialize, causing operators to ignore all predictions.
Why it's wrong: Destroys trust in the system and wastes investigation time.
How to fix: Start with high confidence thresholds, implement prediction accuracy tracking, and allow operators to adjust sensitivity per rule.

⚠️ **Pitfall: Prediction Delay on Regime Changes** When system behavior fundamentally changes (new deployment, traffic pattern shift), predictions based on old data become inaccurate.
Why it's wrong: Misses real issues or creates false alerts during transitions.
How to fix: Detect regime changes (statistical change point detection), retrain models automatically, and mark predictions as "low confidence" during transitions.

10.4 Advanced Templating and Notification Customization

Mental Model: The Dynamic Newspaper Editor Imagine a newspaper that not only reports facts but rewrites each article for different readers: technical details for engineers, business impact for managers, actionable steps for operators. The editor uses templates that pull different information based on who will read it and through which channel (print, web, mobile).

Current alert templates are static text with simple variable substitution. Advanced templating allows dynamic content generation based on alert context, recipient role, and delivery channel.

Advanced Templating Capabilities:

1. **Conditional Content**: Show/hide sections based on alert severity or labels.
2. **External Data Integration**: Pull information from CMDB, runbooks, or incident history.
3. **Multi-format Rendering**: Generate HTML for email, plain text for SMS, Slack blocks for Slack.
4. **Recipient-aware Content**: Show different information to developers vs. SREs vs. managers.

Decision: Template Engine with Pipeline Processing

> **Decision: Extend Go Templates with Custom Functions and Pipeline Stages**
>
> - **Context**: Need more powerful templating than simple variable substitution, but must remain within Go ecosystem for consistency.
> - **Options Considered**:
>   1. **Custom DSL**: Create domain-specific language for alert templates. Maximum flexibility but new language to learn.
>   2. **JavaScript Templates**: Use embedded JavaScript engine (otto, goja). Familiar to many but adds heavy dependency.
>   3. **Extended Go Templates**: Use standard `text/template` with custom functions and multi-stage processing.
> - **Decision**: Extend Go's template system with custom functions for common operations and implement template pipeline (input → transform → render).
> - **Rationale**: Go templates are already used in the codebase, minimizing new dependencies. The pipeline model cleanly separates data fetching from rendering. Custom functions can provide advanced capabilities without template authors needing to learn new syntax.
> - **Consequences**: Template authors need to learn Go template syntax; some advanced logic requires helper functions; pipeline adds complexity to notification rendering.

Template Stage	Purpose	Example Functions
Data Fetching	Retrieve external data	`getCMDBInfo`, `getPreviousIncidents`, `getOnCall`
Transformation	Process and filter data	`filterBySeverity`, `sortByTime`, `groupByTeam`
Conditional Logic	Control content flow	`ifSeverity`, `unlessMaintenance`, `switchCase`
Formatting	Prepare for output channel	`markdownToHTML`, `truncate 200`, `formatDuration`
Rendering	Generate final output	`renderSlackBlocks`, `renderEmailHTML`, `renderSMS`

Template Pipeline Implementation:

```
Alert + Receiver Context
        ↓
[Data Fetching Stage] → getRunbookURL(), getTeamContacts(), getSystemOwner()
        ↓
[Transformation Stage] → filterRelevantSections(), prioritizeActions()
        ↓
[Conditional Stage] → if severity>warning includeDetailedMetrics()
        ↓
[Formatting Stage] → formatForChannel(receiver.Type)
        ↓
[Rendering Stage] → applyTemplate(templateName)
        ↓
Final Notification Payload
```

PLAINTEXT

Extended Template Data Structure: The `TemplateCache` would be enhanced to support:

Field	Type	Description
`Stages`	`[]TemplateStage`	Ordered pipeline stages to execute
`Functions`	`map[string]interface{}`	Custom functions available in templates
`ChannelTemplates`	`map[string]string`	Different templates per notification channel

Example Advanced Template:

```go
{{- /* Fetch external data */ -}}

{{- $runbook := getRunbook .Alert.Labels.service -}}

{{- $oncall := getOnCall .Alert.Labels.team -}}

{{- $history := getSimilarAlerts .Alert 24h -}}


{{- /* Conditional content */ -}}

{{- if gt .Alert.Severity "warning" -}}

CRITICAL ALERT: {{ .Alert.Annotations.summary }}

{{- else -}}

Alert: {{ .Alert.Annotations.summary }}

{{- end -}}


{{- /* Show runbook if exists */ -}}

{{- if $runbook -}}

Runbook: {{ $runbook.URL }}

First steps: {{ $runbook.FirstSteps }}

{{- end -}}


{{- /* Show recurrence if pattern */ -}}

{{- if gt (len $history) 3 -}}

This alert has fired {{ len $history }} times in the last 24 hours.

{{- end -}}


{{- /* Channel-specific formatting */ -}}

{{- if eq .Receiver.Type "slack" -}}

{{- slackBlocks .Alert $runbook $oncall -}}

{{- else if eq .Receiver.Type "email" -}}

{{- emailHTML .Alert $runbook $history -}}

{{- end -}}
```

Integration Points:

1. **Rule Evaluation**: Enhanced `Rule.RenderAnnotations` to use pipeline templates.
2. **Notification Router**: `Router.Send` would apply receiver-specific templates before sending.
3. **Template Management**: API endpoints to create/update templates without restart.

Common Pitfalls in Advanced Templating:

⚠️ **Pitfall: Template Rendering Performance Degradation** Complex templates with external API calls can slow notification delivery from milliseconds to seconds.
Why it's wrong: Delays critical alerts, especially if templates call slow external services.
How to fix: Implement caching for external data (TTL-based), timeout templates (e.g., 500ms max), and provide fallback simple templates.

⚠️ **Pitfall: Template Errors Silently Dropping Notifications** A bug in a template causes rendering failure, resulting in no notification sent.
Why it's wrong: Critical alerts may be missed entirely.
How to fix: Always have a default fallback template, log template errors prominently, and implement dead-letter queue for failed notifications with manual retry.

10.5 Cross-System Integration and Ecosystem Connectivity

Mental Model: The Universal Translator Imagine a control room that not only speaks its native language but can translate messages to/from any other control system in the factory. It can understand alerts from legacy systems, forward relevant alerts to new systems, and maintain a unified view across disparate monitoring tools.

Modern organizations rarely have a single monitoring system. The alerting system should integrate seamlessly with existing tools like incident management systems, chat platforms, and infrastructure orchestration.

Key Integration Points:

1. **Incident Management**: Create/modify/resolve incidents in PagerDuty, Opsgenie, ServiceNow.
2. **ChatOps**: Bidirectional communication with Slack/Microsoft Teams for alert acknowledgment and action.
3. **Configuration Management**: Pull service ownership, runbooks, and escalation policies from CMDB.
4. **Workflow Automation**: Trigger remediation scripts, runbooks, or infrastructure changes.
5. **Observability Platform**: Integrate with tracing, logging, and profiling data for context.

Decision: Plugin Architecture for Integrations

> **Decision: Build Extensible Plugin System over Hard-coded Integrations**
>
> - **Context**: Need to support many external systems with different APIs and authentication methods. Hard-coding each integration creates maintenance burden and limits adaptability.
> - **Options Considered**:
>   1. **Hard-coded Integrations**: Implement each integration directly in the router. Simple initially but becomes unwieldy.
>   2. **Webhook-Only**: Require all integrations to implement webhook endpoints. Pushes complexity to users.
>   3. **Plugin System**: Define clean interfaces for integration points; implement each external system as a plugin.
> - **Decision**: Design plugin architecture with well-defined interfaces for notification receivers, data sources, and action handlers.
> - **Rationale**: Plugins allow community contributions, easier testing, and independent deployment. Organizations can build custom integrations without modifying core code. Plugin isolation prevents buggy integrations from crashing the entire system.
> - **Consequences**: Additional complexity in plugin loading, versioning, and lifecycle management. Need plugin sandboxing for security.

Plugin Type	Interface	Responsibility	Example Plugins
`ReceiverPlugin`	`SendNotification(Notification) error`	Deliver notifications to external system	Slack, PagerDuty, ServiceNow
`DataSourcePlugin`	`FetchData(Query) (Result, error)`	Retrieve external data for templates	CMDB, Git, Incident History
`ActionPlugin`	`ExecuteAction(Action) (Result, error)`	Perform automated actions	RestartService, ScaleDeployment
`WebhookPlugin`	`HandleWebhook(Request) (Response, error)`	Process incoming webhooks	ChatOps commands, status updates

Plugin Architecture Components:

1. **Plugin Registry**: Discovers and loads plugins at startup.
2. **Plugin Sandbox**: Runs plugins in isolated goroutines with resource limits.
3. **Plugin Configuration**: Each plugin has its own configuration section.
4. **Plugin Lifecycle**: `Init()`, `Start()`, `Stop()` methods for graceful management.

Example Plugin Interface:

```go
type ReceiverPlugin interface {
    // Metadata
    Name() string
    Version() string

    // Configuration
    Configure(config map[string]interface{}) error

    // Core functionality
    Send(ctx context.Context, notification Notification) error

    // Health checking
    HealthCheck() error
}

type ActionPlugin interface {
    Name() string
    AvailableActions() []ActionDefinition
    Execute(ctx context.Context, action Action, alert Alert) (ActionResult, error)
}
```

Integration Patterns:

1. **Bidirectional ChatOps**:
   - Alert sent to Slack with interactive buttons
   - User clicks "Acknowledge" → plugin updates alert state
   - User clicks "Run Playbook" → action plugin executes remediation
2. **Automated Incident Management**:
   - Critical alert automatically creates PagerDuty incident
   - Resolution clears alert → plugin resolves incident
   - Incident notes sync back to alert annotations
3. **Contextual Enrichment**:
   - Before notification, data source plugins fetch:
     - Service owner from CMDB
     - Recent deployments from CI/CD
     - Related logs from logging system
   - Enriched context included in notification

Common Pitfalls in Integration:

⚠️ **Pitfall: Integration Failures Blocking Alert Pipeline** If PagerDuty API is down, plugin might retry indefinitely, blocking all notifications.
Why it's wrong: Single point of failure in external system takes down entire alerting.
How to fix: Implement circuit breakers per plugin, timeout individual calls, and have fallback receivers (e.g., if PagerDuty fails, send to email instead).

⚠️ **Pitfall: Plugin Configuration Complexity** Each plugin has different configuration format, making system configuration fragile and hard to validate.
Why it's wrong: Operators make configuration errors that are only discovered at runtime.
How to fix: Define configuration schema per plugin, validate at startup, and provide configuration testing tools.

10.6 Advanced Visualization and Alert Exploration

Mental Model: The Interactive Investigation Dashboard Imagine a control room where clicking on an alert doesn't just show that gauge, but brings up related metrics, recent log entries, dependency maps, and similar past incidents—all in an interactive console that lets the investigator drill down, correlate, and hypothesize.

While alerting focuses on notification delivery, operators need tools to understand and investigate alerts. Advanced visualization turns alert data into insights.

Visualization Capabilities:

1. **Alert Timeline**: Visual representation of alert history across services.
2. **Topology Overlay**: Show alerts on service dependency graph.
3. **Metric Correlation Explorer**: Compare alerting metrics with related signals.
4. **Alert Similarity Search**: Find historically similar alerts and their resolutions.

Decision: Embedded Web UI with API-First Design

> **Decision: Build Single-Page Application Backed by Read-Optimized API**
>
> - **Context**: Operators need rich visualizations but the core system should remain API-first for integration flexibility.
> - **Options Considered**:
>   1. **External Dashboard Tool**: Export data to Grafana, Kibana, etc. Leverages existing tools but requires data duplication.
>   2. **Simple Status Pages**: Basic HTML pages showing current state. Lightweight but limited interactivity.
>   3. **Embedded SPA**: Full-featured React/Vue application served directly from alerting system.
> - **Decision**: Build embedded single-page application with read-optimized REST/GraphQL API.
> - **Rationale**: Embedded UI ensures visualization features evolve with alerting capabilities. API-first approach allows both the UI and external tools to access the same data. Modern SPA frameworks provide rich interactivity needed for investigation.
> - **Consequences**: Additional development and maintenance burden for UI code. Need to manage web assets alongside Go binaries.

UI Component	Purpose	Data Source	Interaction
Alert Inbox	Current active alerts	`/api/alerts`	Filter, sort, acknowledge
Timeline View	Historical alert patterns	`/api/alerts/history`	Zoom, pan, select time range
Topology Map	Service dependency with alerts	`/api/topology` + `/api/alerts`	Click to drill down, highlight paths
Alert Details	Investigation context	`/api/alerts/{id}` + related APIs	View annotations, runbooks, similar alerts
Silence Manager	Create/edit silences	`/api/silences`	Visual time picker, matcher builder

API Design for Visualization:

1. **Streaming Updates**: WebSocket or Server-Sent Events for real-time alert updates.
2. **Aggregate Endpoints**: Pre-computed statistics for dashboard widgets.
3. **Search and Filter**: Rich query language for finding specific alerts.
4. **Relationship Endpoints**: Fetch related alerts, metrics, logs.

Example Investigation Workflow:

1. Operator sees alert in inbox, clicks to view details.
2. Details page shows:
   - Alert annotations and labels
   - Metric graph around alert time (embedded from metrics collector)
   - Recent logs from affected service (integration with logging)
   - Similar historical alerts and their resolutions
   - Current on-call engineer and contact information
3. Operator can:
   - Create silence for maintenance
   - Acknowledge alert (via ChatOps integration)
   - Run diagnostic action (via action plugin)
   - Escalate to another team

Data Structures for Visualization API:

Type Name	Fields	Description
`AlertListResponse`	`Alerts []Alert`, `Total int`, `Page int`, `PageSize int`	Paginated alert list
`TimelineBucket`	`Time time.Time`, `AlertCounts map[string]int`, `SampleAlerts []string`	Time-bucket for timeline view
`TopologyNode`	`ID string`, `Type string`, `Alerts []Alert`, `Children []string`	Node in dependency graph
`SimilarAlert`	`Alert Alert`, `SimilarityScore float64`, `CommonLabels []string`	Similar historical alert with match score

Common Pitfalls in Visualization:

⚠️ **Pitfall: UI Complexity Overwhelming Core Functionality** Development focus shifts from reliable alert delivery to flashy visualizations.
Why it's wrong: Alerting system's primary job is reliable notification; visualizations are secondary.
How to fix: Keep UI as optional component, clearly separate UI and backend teams, and ensure UI failures don't affect alert pipeline.

⚠️ **Pitfall: Performance Degradation from UI Queries** Complex aggregation queries for dashboards slow down primary alert processing.
Why it's wrong: Visualization should not impact reliability of alert delivery.
How to fix: Use read replicas of alert data, implement query caching, and rate-limit UI API endpoints.

10.7 Compliance and Audit Trail

Mental Model: The Security Camera Archive Imagine every action in the control room—every button pressed, every decision made, every communication sent—is recorded with timestamp, operator identity, and before/after state. This archive allows reconstruction of events for compliance, training, and post-incident analysis.

Production alerting systems in regulated industries need detailed audit trails for compliance (SOC2, HIPAA, GDPR) and operational excellence.

Audit Requirements:

1. **Immutable Logging**: All modifications to system state recorded and tamper-evident.
2. **User Attribution**: Every action associated with authenticated user or service account.
3. **Before/After State**: Record state changes for critical operations.
4. **Retention Policy**: Retain audit logs for required duration (e.g., 7 years for financial).
5. **Search and Reporting**: Query audit logs for compliance reporting.

Decision: Append-Only Audit Log with Cryptographic Signing

> **Decision: Implement WAL-style Audit Log with Hash Chain Integrity**
>
> - **Context**: Need provable integrity of audit trail for compliance. Simple database logging can be modified retroactively.
> - **Options Considered**:
>   1. **Database Table with Permissions**: Store audit events in PostgreSQL with restricted access. Simple but mutable by DB admins.
>   2. **External Audit Service**: Send events to specialized audit system (AWS CloudTrail, etc). Offloads responsibility but adds dependency.
>   3. **Append-Only Log with Hash Chain**: Write events to append-only file with cryptographic linking.
> - **Decision**: Implement write-ahead audit log with hash chain for integrity verification.
> - **Rationale**: Hash chain (each block includes hash of previous block) creates tamper-evident log. Append-only file on durable storage provides simplicity and control. Can be periodically exported to external systems for long-term retention.
> - **Consequences**: Need log rotation and archival strategy; verification requires reading entire log; adds I/O overhead to state-changing operations.

Audit Event Type	Trigger	Recorded Information
`AlertStateChange`	Alert changes state	Alert ID, old state, new state, reason
`SilenceCreated`	Silence created	Silence ID, matchers, creator, time range
`SilenceModified`	Silence modified	Silence ID, before/after diff, modifier
`NotificationSent`	Notification delivered	Receiver, alert IDs, timestamp, success/failure
`ConfigurationChange`	Config updated	Before/after config, changer, diff
`UserAction`	Manual action via UI/API	User, action, parameters, result

Audit Log Structure:

1. **Block Format**: Each audit entry contains:
   - Previous block hash
   - Timestamp (nanosecond precision)

- Event type and data
- User/actor identity
- Digital signature (if using asymmetric crypto)
- Current block hash

2. **Integrity Verification**:
   - Periodic background job verifies hash chain
   - Any break indicates tampering
   - Verification results themselves logged
3. **Retention and Rotation**:
   - Log segments rotated by size/time
   - Old segments compressed and archived
   - Immutable storage (WORM) for compliance

Integration Points:

1. **All State-Modifying Methods**: Wrap with audit logging decorator:

```go
func (s *Silencer) AddSilenceWithAudit(silence *Silence, actor string) error {
    before := s.getSilenceSnapshot()
    err := s.AddSilence(silence)
    after := s.getSilenceSnapshot()

    audit.Log("SilenceCreated", map[string]interface{}{
        "actor": actor,
        "silence": silence,
        "before": before,
        "after": after,
    })

    return err
}
```

2. **Authentication Integration**: All API endpoints require authentication; user identity flows to audit log.
3. **Query Interface**: REST API for searching audit logs with filtering by time, actor, event type.

Compliance Reporting:

- **Who did what when**: Generate reports of all actions by user
- **Alert response times**: Measure time from alert to acknowledgment
- **Silence coverage**: Report percentage of time systems under maintenance
- **Notification reliability**: Track successful vs failed deliveries

Common Pitfalls in Audit Implementation:

⚠️ **Pitfall: Audit Logging Impacting Performance** Synchronous audit write for every state change adds latency to critical alert processing.
Why it's wrong: Slows down alert pipeline, especially during incident storms.
How to fix: Use buffered asynchronous writes with durable queue, batch events, and accept eventual consistency for audit trail.

⚠️ **Pitfall: Incomplete Audit Coverage** Some state changes happen through indirect paths and aren't audited.
Why it's wrong: Compliance gaps, inability to reconstruct events during post-mortem.
How to fix: Audit at the data layer (all writes to state maps), not just at API layer. Use middleware pattern to ensure all modifications go through audited code paths.

10.8 Edge Computing and Disconnected Operation

Mental Model: The Field Operations Kit Imagine a control room that can be packed into a ruggedized case and deployed to remote locations with intermittent connectivity. It continues monitoring local systems, making autonomous decisions, and queuing notifications for when connectivity is restored.

Edge deployments (IoT, retail stores, remote offices) need alerting that works during network partitions and synchronizes when connected.

Edge-Specific Requirements:

1. **Disconnected Operation**: Function without connection to central metrics or notification systems.
2. **Local Alerting**: Notify on-site personnel via local channels (buzzers, lights, local network).
3. **Queue and Forward**: Store alerts and metrics during disconnection, sync when online.
4. **Resource Constraints**: Run on limited CPU, memory, and storage devices.
5. **Autonomous Decisions**: Make alert decisions without central coordination.

Decision: Hierarchical Edge Architecture with Sync Bridge

> **Decision: Deploy Lightweight Edge Nodes with Periodic Central Sync**
>
> - **Context**: Need alerting at edge locations with poor connectivity, but also want centralized visibility and management.
> - **Options Considered**:
>   1. **Fully Independent Edge Nodes**: Each edge location runs complete alerting stack. Maximum independence but no central visibility.
>   2. **Thin Edge Proxies**: Edge only forwards metrics to central; all evaluation happens centrally. Simple but fails when disconnected.
>   3. **Hierarchical with Sync**: Edge nodes run full evaluation locally, periodically sync state with central parent.
> - **Decision**: Implement hierarchical architecture where edge nodes operate independently but sync to parent when possible.
> - **Rationale**: Provides continuous operation during disconnection while maintaining eventual central visibility. Edge nodes can be configured from central but have local fallback configuration. Sync protocol can handle intermittent connectivity.
> - **Consequences**: Need conflict resolution for state sync; edge nodes require full alerting binary; sync protocol adds complexity.

Component	Edge Node	Central Node	Sync Behavior
Rule Evaluation	Local, using local metrics	Central, using aggregated metrics	Rules can differ; edge overrides allowed
Alert State	Maintained locally	Aggregated from all edges	Edge pushes alerts to central
Silences	Local only or inherited	Can propagate to edges	Central can push silences to edges
Notifications	Local channels (LED, sound)	Central channels (Slack, PagerDuty)	Edge alerts can forward to central for notification

Edge Node Architecture:

1. **Lightweight Components**: Same components as central but with resource limits:
   - Rule Evaluator with reduced evaluation frequency
   - Grouper with smaller group retention
   - Local silences only (no inhibition for simplicity)
   - Basic router with local receivers (buzzer, LED, local HTTP)
2. **Sync Manager**:
   - Periodically attempts connection to parent
   - Pushes local alerts and state changes
   - Pulls configuration updates and central silences
   - Uses delta compression to minimize bandwidth
3. **Queue and Retry**:
   - Local disk queue for unsent alerts
   - Exponential backoff for connection attempts
   - Configurable retention (drop old alerts if queue full)

Data Structures for Edge Sync:

Type Name	Fields	Description
`SyncMessage`	`EdgeID string`, `Sequence uint64`, `Alerts []Alert`, `StateChanges []StateChange`, `ConfigHash string`	Message from edge to central
`SyncResponse`	`Accepted bool`, `CentralAlerts []Alert`, `ConfigUpdates []ConfigUpdate`, `Silences []Silence`	Response from central to edge
`EdgeConfig`	`EdgeID string`, `Rules []Rule`, `Receivers []Receiver`, `SyncInterval time.Duration`, `ResourceLimits ResourceLimits`	Configuration for edge node

Sync Protocol:

1. **Connection**: Edge initiates HTTPS connection to central with authentication.
2. **State Exchange**: Edge sends its current state fingerprint; central sends what it thinks edge should have.
3. **Delta Calculation**: Both sides compute differences.
4. **Application**: Apply updates in careful order (silences before alerts, etc.)
5. **Acknowledgment**: Both sides confirm sync completion.

Example Edge Deployment:

- **Hardware**: Raspberry Pi with 1GB RAM
- **Metrics**: Local Prometheus scraping 50 targets
- **Rules**: 20 simple threshold rules
- **Local Notification**: Piezo buzzer for critical, LCD display for warnings
- **Sync**: Every 5 minutes if connected, stores 24 hours of alerts offline

Common Pitfalls in Edge Alerting:

⚠️ **Pitfall: Sync Conflicts Creating Alert Storms** Edge creates alert while offline, central has different view, sync causes duplicate or conflicting alerts.
Why it's wrong: Operators see duplicate alerts or missed resolutions.
How to fix: Use vector clocks for conflict resolution, last-write-wins for silences, and alert de-duplication by fingerprint across edges.

⚠️ **Pitfall: Edge Resource Exhaustion** Edge device runs out of memory during incident storm, crashes, loses all state.
Why it's wrong: Edge becomes useless during actual emergency.
How to fix: Implement aggressive resource limits, circuit breakers on rule evaluation, and persist state to disk frequently.

10.9 Machine Learning for Alert Tuning

Mental Model: The Alert Gardener Imagine a gardener who not only waters plants but learns which plants need more water, which are getting too much, and adjusts automatically. The gardener observes which alerts get acknowledged quickly (valuable) versus which get ignored (noise), and prunes the noise while nurturing the signal.

Manual alert tuning is time-consuming and often neglected. ML can automatically adjust thresholds, detect noisy alerts, and suggest rule improvements.

ML Applications in Alert Tuning:

1. **Threshold Optimization**: Adjust thresholds based on historical normal ranges.
2. **Alert Fatigue Scoring**: Identify alerts that are frequently ignored or silenced.
3. **Similar Alert Clustering**: Group related alerts for consolidation.
4. **Predictive Tuning**: Adjust alert parameters based on predicted workload changes.

Decision: Supervised Learning with Human Feedback Loop

> **Decision: Use Classification Models Trained on Operator Actions**
>
> - **Context**: Need to distinguish valuable alerts from noise without predefined rules. Pure anomaly detection generates too many false positives.
> - **Options Considered**:
>   1. **Unsupervised Anomaly Detection**: Find alerts that are statistical outliers. Doesn't require training data but doesn't know what's actually important.
>   2. **Reinforcement Learning**: System learns by trying different thresholds and observing results. Too slow and risky for production.
>   3. **Supervised Classification**: Train models on historical alerts labeled by operator actions (acknowledged, silenced, ignored).
> - **Decision**: Implement supervised binary classification (valuable vs noise) using operator actions as implicit labels.
> - **Rationale**: Operator actions provide natural training signal—alerts that get quickly acknowledged are likely important; those repeatedly ignored are likely noise. Supervised learning provides explainable predictions (feature importance). Human can override model suggestions.
> - **Consequences**: Need to collect and label historical data; model requires retraining as systems evolve; false positives in classification could suppress important alerts.

ML Feature	Description	Weight in Model
`acknowledgement_time`	How quickly alert was acknowledged	Fast acknowledgment → important
`silence_duration`	How long alert was silenced	Long silence → likely noise
`recurrence_count`	How often alert fires	Frequent recurrence → may need tuning
`annotation_length`	Detail in alert annotations	Detailed annotations → more important
`escalation_path`	Whether alert was escalated	Escalation → important
`time_of_day`	When alert fired	Off-hours alerts may be more important

ML Pipeline:

1. **Feature Extraction**: From alert history, extract features for each alert instance.
2. **Label Generation**: Use operator actions as implicit labels:
   - **Positive** (valuable): Acknowledged < 5 minutes, escalated, commented
   - **Negative** (noise): Ignored > 1 hour, repeatedly silenced, same alert auto-resolved many times
3. **Model Training**: Train classifier (logistic regression, random forest) on historical data.
4. **Inference**: Score new alerts with model prediction.
5. **Action Suggestions**:
   - For high-noise-score alerts: suggest raising threshold or increasing `for_duration`
   - For high-value alerts: suggest lowering threshold or adding annotations

Integration with Alert Pipeline:

1. **Alert Scorer**: Component that runs in background, scoring recent alerts.
2. **Tuning Suggestions API**: Endpoint returning suggestions for rule improvements.
3. **Auto-Tuning (Optional)**: For low-risk rules, automatically apply suggested tuning with human approval workflow.

Example ML-Based Tuning:

- Alert "HighCPUUsage" fires 50 times/day, always auto-resolves, never acknowledged
- ML scores it as 95% noise
- System suggests: increase threshold from 80% to 90% or add `for_duration: 5m`
- Operator reviews suggestion, approves
- Rule automatically updated, noise reduced by 80%

Data Structures for ML Tuning:

Type Name	Fields	Description
`AlertFeatures`	`AlertID string`, `Features map[string]float64`, `Timestamp time.Time`	Extracted features for ML
`AlertLabel`	`AlertID string`, `IsValuable bool`, `Confidence float64`, `LabelSource string`	Human or implicit labeling
`TuningSuggestion`	`RuleName string`, `CurrentValue interface{}`, `SuggestedValue interface{}`, `Confidence float64`, `ExpectedImpact string`	Suggested rule modification
`ModelMetadata`	`Version string`, `TrainingDate time.Time`, `Accuracy float64`, `Features []string`	ML model information

Common Pitfalls in ML Tuning:

⚠️ **Pitfall: Model Drift Causing Missed Alerts** As systems evolve, old training data becomes irrelevant, model starts suppressing important new alert types.
Why it's wrong: Critical alerts get suppressed because they look like old noise patterns.
How to fix: Continuous retraining on recent data, concept drift detection, and human review of all suppression suggestions.

⚠️ **Pitfall: Feedback Loop Amplifying Bias** If model suppresses certain alerts, they never get operator attention, so model never learns they could be important.
Why it's wrong: Creates self-reinforcing bias where some alert types always get suppressed.
How to fix: Include exploration factor (sometimes show suppressed alerts), periodic manual review of suppressed alerts, and balanced sampling in training.

10.10 Cost Optimization and Resource Awareness

Mental Model: The Energy-Efficient Factory Imagine a control room that monitors not just operational metrics but also the cost of monitoring itself—electricity for sensors, network bandwidth for metrics, personnel time for alert response. It optimizes alerting to achieve reliability goals at minimum cost.

In cloud environments, monitoring and alerting have direct costs: metrics storage, query execution, notification fees. The system should be aware of these costs and optimize accordingly.

Cost Factors in Alerting:

1. **Metrics Collection**: Number of time series, scrape frequency, retention period.
2. **Rule Evaluation**: Query complexity, evaluation frequency, historical data usage.
3. **Notification Delivery**: Message volume, premium channel costs (SMS, phone calls).
4. **Storage**: Alert history, silence records, audit logs.
5. **Compute**: ML processing, correlation analysis.

Decision: Cost-Aware Alerting with Budget Constraints

> **Decision: Implement Cost Tracking and Optimization Scheduler**
>
> - **Context**: Need to balance alerting effectiveness with infrastructure costs, especially at scale.
> - **Options Considered**:
>   1. **Fixed Resource Limits**: Simple caps on metrics, rules, notifications. Prevents runaway costs but doesn't optimize.
>   2. **Cost Reporting Only**: Track and report costs but don't act on them. Creates awareness but no optimization.
>   3. **Dynamic Optimization**: Adjust alerting parameters based on cost budgets and effectiveness metrics.
> - **Decision**: Implement cost tracking with dynamic optimization for non-critical alerts.
> - **Rationale**: Critical alerts should always fire regardless of cost, but non-critical alerts can be tuned based on cost/benefit. Dynamic adjustment allows the system to stay within budget while maximizing coverage.
> - **Consequences**: Adds complexity to rule evaluation; need to define cost/benefit metrics; risk of over-optimizing and missing issues.

Cost Dimension	Measurement	Optimization Strategy
Metrics Query	Query execution time, data points scanned	Adjust evaluation frequency, simplify queries, use sampling
Notification Volume	Messages sent, premium channel usage	Increase grouping, raise thresholds, use cheaper channels first
Storage	Alert history size, silence count	Adjust retention periods, compress old data
Compute	CPU time for correlation/ML	Schedule intensive processing during off-peak

Cost-Aware Components:

1. **Rule Evaluator with Budget**:
   - Each rule has cost budget (query time, execution frequency)
   - Rules sorted by priority; lower priority rules get throttled if budget exceeded
   - Adaptive evaluation: reduce frequency when metrics are stable
2. **Notification Router with Cost Routing**:
   - Each receiver has cost per notification
   - Route to cheapest channel first, escalate to expensive channels if no acknowledgment
   - Batch notifications to reduce message count
3. **Metrics Query Optimizer**:
   - Cache query results for similar rules
   - Use approximate queries (sampling) for non-critical alerts
   - Downsample historical data for long `for_duration` checks

Cost Tracking Data Structures:

Type Name	Fields	Description
`CostMetric`	`Type string`, `Resource string`, `Units float64`, `CostUSD float64`, `Timestamp time.Time`	Individual cost measurement
`CostBudget`	`Period time.Duration`, `MaxUSD float64`, `CurrentUSD float64`, `PriorityRules []string`	Budget for alerting component
`OptimizationAction`	`Component string`, `Action string`, `Parameters map[string]interface{}`, `ExpectedSavings float64`	Suggested optimization
`CostEffectiveness`	`AlertID string`, `CostUSD float64`, `BusinessValue float64`, `Ratio float64`	Cost/benefit ratio for alert

Optimization Algorithms:

1. **Rule Priority Scheduling**:

```
total_budget = $100/month
rule_priority = [P0, P1, P2, P3]

foreach evaluation_cycle:
  available_budget = total_budget - spent_this_month
  if available_budget < threshold:
    skip_low_priority_rules(available_budget / cost_per_rule)
```

2. **Notification Channel Selection**:

```
cost_per_channel = {
  "email": $0.0001,
  "slack": $0.001,
  "sms": $0.05,
  "phone": $0.15
}

route_alert(alert):
  if alert.severity == "critical":
    send_to_all_channels()
  else:
    start_with_cheapest_channel()
    if no_ack_in_timeout():
      escalate_to_next_channel()
```

3. **Query Optimization**:
   - Cache query results for identical rules
   - Use rolling windows instead of full history scans
   - Sample metrics during low-priority evaluations

Integration with Cloud Cost APIs:

- Pull actual costs from cloud provider (AWS Cost Explorer, GCP Billing API)
- Attribute costs to specific teams/services via labels
- Showback/chargeback reporting for alerting costs

Common Pitfalls in Cost Optimization:

⚠️ **Pitfall: Over-Optimization Missing Critical Alerts** Cost savings from skipping rule evaluations result in missed detection of real issues.
Why it's wrong: Defeats the purpose of monitoring—reliability is more important than cost savings.
How to fix: Never skip P0/P1 rules, only optimize P3/P4. Implement canary checks that would have fired if rule ran.

⚠️ **Pitfall: Complex Optimization Creating Its Own Costs** Cost optimization logic itself consumes significant resources to track and decide.
Why it's wrong: Optimization overhead exceeds savings.
How to fix: Keep optimization simple and infrequent (hourly, not per-alert). Measure optimization overhead and ensure it's <10% of total cost.

10.11 Implementation Roadmap and Prioritization

While all these extensions provide value, implementing them requires careful prioritization based on organizational needs. The following table provides a suggested implementation roadmap:

Extension	Business Value	Implementation Effort	Dependencies	Suggested Phase
High Availability	Critical for production	High	None (builds on core)	Phase 1
Advanced Templating	Improved operator experience	Medium	Core system stable	Phase 1
Cross-System Integration	Ecosystem connectivity	Medium-High	Plugin system	Phase 2
Compliance & Audit	Regulatory requirements	Medium	Stable API	Phase 2
Predictive Alerting	Proactive operations	High	Historical metrics storage	Phase 3
Alert Correlation	Noise reduction	High	Topology mapping	Phase 3
Cost Optimization	Cloud cost control	Medium	Cost tracking infra	Phase 4
Edge Computing	Remote locations	High	Sync protocol	Phase 4
ML Alert Tuning	Automatic maintenance	High	ML infrastructure	Phase 5
Advanced Visualization	Investigation efficiency	Medium-High	Web UI framework	Phase 5

Phase 1: Reliability and Usability (Months 1-3)

- High Availability for production readiness
- Advanced Templating for better notifications
- Basic visualization (alert inbox, details)

Phase 2: Enterprise Integration (Months 4-6)

- Plugin system for integrations
- Compliance audit trail
- ChatOps bidirectional integration

Phase 3: Intelligence (Months 7-9)

- Predictive alerting foundation
- Basic correlation (temporal, topological)
- Cost tracking and reporting

Phase 4: Scale and Reach (Months 10-12)

- Cost optimization features
- Edge computing prototype
- Advanced correlation (ML-based)

Phase 5: Automation and Insight (Months 13-15)

- ML-based alert tuning
- Advanced visualization suite
- Full autonomous operations pilot

Each extension should be treated as a mini-project with its own design document, testing plan, and rollout strategy. The core system's clean interfaces and separation of concerns make these extensions feasible as additive components rather than fundamental rewrites.

Implementation Guidance

While this section focuses on future possibilities rather than current implementation, the architectural patterns described provide guidance for how to structure the codebase to accommodate future extensions.

Technology Recommendations for Extensions

Extension	Recommended Technologies	Integration Pattern
High Availability	etcd for coordination, memberlist for gossip, CRDT libraries (automerge, dgo)	Sidecar process or library within main binary
Predictive Alerting	Facebook Prophet, Kats, or custom ARIMA implementation	Separate service with gRPC API to core
Advanced Templating	Go text/template with custom functions, Lua for complex logic	Template engine library within router
Cross-System Integration	Go plugin system, HashiCorp go-plugin, or separate microservices	Plugin interface with dynamic loading
Compliance Audit	Append-only log with Merkle tree, AWS CloudTrail SDK	Decorator pattern around state changes
Edge Computing	SQLite for local storage, sync protocol with protocol buffers	Separate binary build with feature flags
ML Alert Tuning	TensorFlow Lite, ONNX Runtime, or scikit-learn via Python microservice	External service with REST API
Cost Optimization	Cloud provider SDKs, OpenCost for Kubernetes	Background job with reporting API
Visualization	React/Vue SPA, WebSocket for real-time, D3.js for charts	Separate web server embedding static assets

Recommended File Structure for Extensible Architecture

To accommodate future extensions, organize the codebase with clear boundaries:

```
alerting-system/
├── cmd/
│   ├── alertmanager/          # Main binary
│   ├── edge-node/             # Edge variant (future)
│   └── ml-tuner/              # ML service (future)
├── internal/
│   ├── core/                  # Core alerting logic (current project)
│   │   ├── alert/
│   │   ├── rule/
│   │   ├── silence/
│   │   └── router/
│   ├── extensions/            # Future extensions
│   │   ├── ha/                # High availability
│   │   │   ├── crdt/
│   │   │   ├── gossip/
│   │   │   └── sync/
│   │   ├── correlation/       # Alert correlation
│   │   │   ├── temporal/
│   │   │   ├── topological/
│   │   │   └── ml/
│   │   ├── prediction/        # Predictive alerting
│   │   │   ├── forecasting/
│   │   │   └── anomaly/
│   │   └── cost/              # Cost optimization
│   │       ├── tracking/
│   │       └── optimization/
│   ├── plugins/               # Plugin system
│   │   ├── receiver/
│   │   ├── datasource/
│   │   └── action/
│   └── api/                   # REST/GraphQL APIs
│       ├── v1/                # Core API
│       ├── v2/                # Extended API (future)
│       └── internal/          # Internal APIs
├── webui/                     # Embedded web UI (future)
│   ├── src/
│   ├── public/
│   └── package.json
└── pkg/                       # Public libraries
    ├── plugin-sdk/            # Plugin SDK for external developers
    └── client-go/             # Go client library
```

Extension Points in Current Design

The current system already has several extension points that future enhancements can leverage:

1. **Rule Evaluation Hooks**: The `Engine.evaluateAllRules` method can be extended with pre/post evaluation hooks for correlation or prediction.
2. **Pipeline Stages**: The `Coordinator.processAlertThroughPipeline` can be modified to include additional processing stages.
3. **Receiver Interface**: New notification channels can be added by implementing the `Receiver` interface.

4. **Template Functions**: Custom template functions can be registered in `TemplateCache`.
5. **Matcher Extensions**: The `Matcher` interface can be extended to support new matching operators.

Starter Code for Plugin System Foundation

Here's a minimal plugin interface that future extensions can build upon:

GO

```go
// internal/plugins/interface.go

package plugins


// Plugin is the base interface all plugins must implement

type Plugin interface {

    // Metadata

    Name() string

    Version() string

    Description() string



    // Lifecycle

    Init(config map[string]interface{}) error

    Start() error

    Stop() error



    // Health

    HealthCheck() HealthStatus

}


// HealthStatus represents plugin health

type HealthStatus struct {

    Status  string                 `json:"status"`  // "healthy", "unhealthy", "unknown"

    Message string                 `json:"message"`

    Details map[string]interface{} `json:"details,omitempty"`

}


// ReceiverPlugin sends notifications

type ReceiverPlugin interface {

    Plugin



    // Send delivers a notification

    Send(ctx context.Context, notification Notification) error



    // SupportedTypes returns the notification types this plugin handles

    SupportedTypes() []string

}


// Registry manages plugin loading and lifecycle

type Registry struct {

    plugins map[string]Plugin

    mu      sync.RWMutex

}
```

```go
func NewRegistry() *Registry {
    return &Registry{
        plugins: make(map[string]Plugin),
    }
}

// LoadPlugin dynamically loads a plugin from a .so file
func (r *Registry) LoadPlugin(path string) error {
    // TODO: Implement dynamic loading using plugin.Open
    // This allows adding new integrations without recompiling
}

// GetReceiver returns a receiver plugin by name
func (r *Registry) GetReceiver(name string) (ReceiverPlugin, bool) {
    r.mu.RLock()
    defer r.mu.RUnlock()

    p, exists := r.plugins[name]
    if !exists {
        return nil, false
    }

    rp, ok := p.(ReceiverPlugin)
    return rp, ok
}
```

Migration Strategy for Future Extensions

When implementing extensions, maintain backward compatibility:

1. **Feature Flags**: Use compile-time or runtime feature flags to enable extensions:

```go
// build tags in Go
// go build -tags=ha,correlation

// or runtime configuration
if config.EnableHA {
    coordinator = ha.NewHACoordinator(standardCoordinator)
}
```

2. **Data Migration**: Design extension data structures to be additive, not breaking:

```go
// Original Alert
type Alert struct {
    Labels      map[string]string
    Annotations map[string]string
    // ... existing fields
}

// Extended Alert (future)
type ExtendedAlert struct {
    Alert
    // New fields only
    CorrelationID string
    Prediction    *Prediction
    CostUSD       float64
}
```

3. **API Versioning**: Use API versioning for breaking changes:

```
/api/v1/alerts    # Current API
/api/v2/alerts    # Future with extensions
```

By planning for extensibility from the beginning, the alerting system can evolve to meet future needs without requiring complete rewrites.

11. Glossary

> **Milestone(s):** This glossary defines key terminology used throughout the entire Alerting System design document, providing a centralized reference for concepts spanning all four milestones (Rule Evaluation, Alert Grouping, Silencing & Inhibition, and Notification Routing).

The Alerting System introduces specific technical terminology to describe its components, behaviors, and design patterns. This glossary provides precise definitions for these terms to ensure consistent understanding across the design document and implementation.

Core Alerting Concepts

Term	Definition
Alert	A discrete signal indicating that a monitored condition has been met. It contains labels for identification, annotations for human-readable context, and metadata about its state and timing. The `Alert` struct is the fundamental data structure flowing through the system.
Alert Fatigue	The phenomenon where operators become desensitized to alerts due to excessive volume of low-value or repetitive notifications. The system combats this through grouping, silencing, and intelligent routing to reduce noise.
Annotations	Key-value pairs attached to alerts that provide human-readable context, such as descriptions, summaries, and links to runbooks. Unlike labels, annotations are not used for matching or grouping but appear in notifications.
Backpressure	Resistance or force opposing the desired flow of data through a system. In the alerting pipeline, if a component (like the router) cannot keep up with incoming alerts, it signals upstream components to slow down to prevent system overload.
Circuit Breaker	A design pattern that prevents repeated attempts to perform operations that are likely to fail. In notification routing, a `ReceiverCircuitBreaker` stops sending to a failing receiver after consecutive failures, allowing it to recover.
Common Labels	Labels that have identical values across all alerts within a notification group. These are extracted and displayed once in grouped notifications to reduce redundancy and highlight the shared context of the alert batch.
Conflict-Free Replicated Data Types (CRDTs)	Data structures that can be replicated across multiple nodes and merged without coordination, enabling eventual consistency in distributed systems. A potential future extension for achieving high availability in the alerting system.
Coordinator	The central orchestration component (`Coordinator` struct) that connects all other components (Rule Evaluator, Grouper, Silencer/Inhibitor, Router) and manages the unidirectional flow of alerts through the processing pipeline.
Dead-Letter Queue	Storage for failed notifications that cannot be delivered to their intended receivers. These notifications can be inspected and replayed manually once the underlying issue is resolved, ensuring no alert is permanently lost.
Debug Endpoint	An HTTP endpoint (served by `DebugServer`) that exposes internal system state for diagnostics, such as active rules, alert groups, and silences, without requiring log parsing or direct code inspection.
End-to-End Flow	The complete journey of an alert from detection (rule evaluation) through processing (grouping, silencing, inhibition) to final delivery (notification routing). Understanding this flow is critical for debugging and system comprehension.
EqualLabels	A list of label names specified in an `InhibitionRule` that must have identical values between the source alert and target alert for inhibition to apply. This ensures inhibition only occurs between related entities (e.g., same `cluster` and `service`).
Exponential Backoff	A retry strategy where wait time doubles after each failed attempt. Used by the router when sending notifications to receivers, preventing overwhelming a struggling service while still eventually attempting delivery.
Fail-Open	A safety principle where a component, when failing, defaults to allowing data to pass through rather than blocking it. For example, if the Silencer crashes, alerts should continue to flow to the Router rather than being lost.
Fingerprint	A unique identifier for an alert computed deterministically from its labels (via `Alert.Fingerprint()`). Used for deduplication, tracking alert instances across state transitions, and identifying alerts within groups.
For-Duration	The duration (specified in a `Rule`) that an alert condition must remain continuously true before transitioning from `pending` to `firing` state. This prevents transient spikes from triggering immediate notifications.
Graceful Degradation	The ability of a system to continue operating with reduced functionality during partial failures. For example, if email notifications fail, the system might continue sending Slack notifications while logging the email failure.
Group	A collection of related alerts aggregated together for notification purposes. The `Group` struct manages a set of alerts sharing common grouping label values and controls timing for notification dispatch.
Group Interval	The minimum time between sending notifications for the same alert group. Configurable per route via `Route.GroupInterval` , this prevents notification spam when alerts in a group are frequently updating.
Group Key	A deterministic string derived from the sorted values of grouping labels (e.g., `alertname` , `cluster`). Generated by `Grouper.calculateGroupKey()` , it uniquely identifies an alert group and is used as a map key.
Group Wait	The time to buffer newly arriving alerts in a group before sending the first notification. Configurable per route via `Route.GroupWait` , this allows multiple related alerts to be batched into a single notification.
Health Endpoint	An HTTP endpoint (served by `HealthStatus.ServeHTTP()`) that reports the operational status of system components. Used by monitoring systems to detect if the alerting system itself is healthy.
High Availability (HA)	A system design approach that ensures continuous operation during component failures, typically achieved through redundancy, failover mechanisms, and distributed state management. A future extension for the alerting system.

Term	Definition
Hot Reloading	The ability to apply configuration changes (rules, silences, routes) without restarting the process. This ensures continuous alert evaluation and avoids losing in-flight alert state during updates.
Inhibition	Automatic suppression of target alerts when a source alert with matching labels is actively firing. Implemented by the `Inhibitor` component, this prevents redundant notifications for symptoms when the root cause is already known.
Label	A key-value pair (e.g., `cluster="prod-us-east-1"`, `severity="critical"`) attached to alerts and time series for dimensionality. Labels enable matching, grouping, silencing, inhibition, and routing decisions.
Matcher	A label matching condition supporting exact equality (`=`) and regular expression (`=~`) matches. The `Matcher` struct contains the label name, expected value, and regex flag, with a `Match()` method for evaluation.
MetricQueryResult	The structured response from querying the metrics API, containing the status and data (including time series with metric labels and values). Used by the Rule Evaluator to determine if alert conditions are met.
Notification	A formatted message ready for delivery to an external system, containing a group of alerts and destination receiver information. The `Notification` struct is the output of the Grouper and input to the Router.
Orphaned Alert	An alert that has changed its grouping labels (e.g., due to label updates) and remains in its old group, becoming invisible to notifications for its new logical group. The Grouper must detect and move such alerts.
Pipeline	The sequence of processing stages (Rule Evaluator → Grouper → Silencer/Inhibitor → Router) through which alerts flow. The pipeline implements unidirectional flow with each component having a specific responsibility.
Prometheus Metrics	Instrumentation metrics exposed by the alerting system itself in Prometheus format (typically at `/metrics`). These allow monitoring the alerting system's health, performance, and internal state.
PromQL	Prometheus Query Language, used for selecting and aggregating time series data. The Rule Evaluator uses PromQL-like expressions (though simplified in this implementation) to define alert conditions.
Rate Limiting	Controlling the frequency of notifications sent to a receiver to prevent overload. Implemented via `RateLimiter` using a token bucket algorithm, ensuring maximum send frequency per channel is not exceeded.
Read-Only Feedback Loop	A design pattern where a component reads state from another component but does not modify it. For example, the Inhibitor reads the set of firing alerts from the engine but does not alter alert states.
Receiver	A destination for notifications, such as Slack, PagerDuty, email, or a generic webhook. The `Receiver` struct contains the receiver type and configuration specific to that notification channel.
REPL	Read-Eval-Print Loop, an interactive console for debugging and manual intervention. A potential future extension allowing operators to query system state and execute commands in real-time.
Route	A node in the routing tree that defines where alerts matching certain criteria should be sent. The `Route` struct contains matchers, a receiver reference, child routes, and timing parameters (group_wait, group_interval).
Routing Tree	A hierarchical configuration of routes where each node can have matchers and child routes. The router performs a depth-first traversal of this tree to determine which receivers should receive a notification.
Rule	A definition of an alert condition, including the PromQL-like expression, threshold, comparison operator, for-duration, and labels/annotations to attach to resulting alerts. The `Rule` struct is evaluated periodically.
Scope Creep	The uncontrolled expansion of project scope beyond original objectives. The Goals and Non-Goals section explicitly defines boundaries to prevent this during implementation.
Signal vs. Noise	The core challenge of distinguishing actionable alerts (signal) from irrelevant data (noise). The entire alerting system is designed to maximize signal and minimize noise through filtering, grouping, and intelligent routing.
Silence	A time-bounded suppression rule defined by label matchers. The `Silence` struct includes matchers, start/end times, and metadata; alerts matching an active silence are suppressed from notifications.
Split-Brain	A condition in distributed systems where network partitions cause components to operate independently, potentially making conflicting decisions. High-availability designs must address this risk.
Structured Logging	Logging with key-value pairs instead of plain text, making logs machine-parsable and enabling sophisticated filtering and analysis. Used throughout the alerting system for operational visibility.
Template Pipeline	Multi-stage template processing where alert annotations are rendered through a templating engine (`TemplateCache`) that can fetch additional data and transform values before inclusion in notifications.
Token Bucket	An algorithm for rate limiting that allows bursts up to a capacity (bucket size) while maintaining a long-term average rate (refill rate). Used by `RateLimiter` to control notification frequency.

Term	Definition
Unidirectional Flow	A design principle where data moves forward through components but never backward. Alerts flow from Rule Evaluator to Grouper to Silencer/Inhibitor to Router, simplifying reasoning about system state.
Vector Clock	An algorithm for generating partial ordering of events in distributed systems by maintaining version counters per node. Used in `VersionedAlert` for conflict resolution in a distributed alerting system.
Webhook	An HTTP callback for sending notifications to external systems. A generic receiver type that allows integration with any system capable of receiving HTTP POST requests with JSON payloads.

Advanced & Future Concepts

Term	Definition
Audit Trail	A secure, timestamped record of system activities (e.g., silence creation, configuration changes) for compliance, troubleshooting, and understanding historical actions.
Causal Inference	Determining cause-and-effect relationships from observational data. A potential enhancement where the system analyzes alert patterns to identify root causes rather than just correlating occurrences.
Correlation Engine	A component that identifies relationships between alerts based on timing, labels, or topology, potentially grouping related alerts or suggesting common root causes beyond simple label matching.
Cost Optimization	Reducing alerting system resource consumption and associated costs, potentially by tuning alert frequency, pruning low-value alerts, or implementing cost-aware routing decisions.
Dependency Graph	A representation of service dependencies that can be used to enhance alert routing or inhibition (e.g., suppress downstream alerts when an upstream dependency is failing).
Edge Computing	Computing at or near the source of data generation. A potential deployment model where rule evaluation happens close to metrics sources to reduce latency and central system load.
Machine Learning Tuning	Using ML algorithms to optimize alert parameters (thresholds, for-durations) based on historical data, reducing false positives and adapting to changing system behavior.
Plugin Architecture	A system design where functionality can be extended via plugins, allowing third-party receivers, custom grouping strategies, or specialized silence managers without modifying core code.
Predictive Alerting	Using historical data and trend analysis to forecast future alert conditions, potentially allowing proactive intervention before issues actually occur.

State & Status Terms

Term	Definition
Firing	An alert state indicating the condition has been met for the full for-duration and requires attention. Firing alerts trigger notifications unless suppressed by silencing or inhibition.
Inactive	The initial state of an alert condition before being evaluated. Not represented as a distinct state in the `Alert.State` field but conceptually precedes the pending state.
Pending	An alert state where the condition has been met but the for-duration has not yet elapsed. The alert is "warming up" and will transition to firing if the condition remains true.
Resolved	An alert state where a previously firing condition is no longer true. Resolved alerts may trigger resolution notifications to inform operators that the issue has cleared.
Suppressed	An alert that would normally trigger notifications but is being withheld due to matching an active silence or being inhibited by another firing alert. The alert retains its underlying state (firing/pending).

Component-Specific Terms

Term	Definition
Engine	The Rule Evaluator component (`Engine` struct) responsible for periodically querying metrics, evaluating rules, and generating alerts based on threshold comparisons.
Grouper	The component (`Grouper` struct) that aggregates related alerts into groups based on grouping labels, manages timing windows (group_wait, group_interval), and triggers batched notifications.
Inhibitor	The component (`Inhibitor` struct) that suppresses target alerts when source alerts with matching labels are firing, implementing the inhibition logic defined by `InhibitionRule` objects.
Router	The component that traverses the routing tree to send notifications to appropriate receivers based on label matching, applying rate limiting and retry logic for reliable delivery.
Silencer	The component (`Silencer` struct) that manages active silences, checks if alerts match any silence, and suppresses matched alerts from proceeding to notification routing.
TimerManager	A helper component (`timerManager` struct) that manages time-based events (group_wait, group_interval) with proper start, reset, and stop semantics, preventing timer leaks.

> **Key Insight:** Understanding these terms is essential for comprehending the system's architecture and implementation. The glossary serves as a reference when encountering unfamiliar terminology in other sections of the design document.
