# Dynamic Alert Configuration System

## Feasibility & Approach Document

**Version:** 1.0
**Date:** January 2026
**Status:** Feasibility Analysis for POC

---

## Executive Summary

This document outlines a feasible approach to enable dynamic, customer-specific alert configuration without manual YAML editing or system downtime. The solution allows platform administrators to configure, modify, and manage alerts for multiple customers through a REST API and web interface while maintaining system stability and isolation.

**Key Outcome:** Customers can have fully customized alerts, notification preferences, and routing rules that can be updated in real-time without affecting other customers or requiring system restarts.

---

## Problem Statement

### Current Limitations

1. **Manual Configuration Required**
   - Alert rules exist in static YAML files
   - Any change requires manual file editing
   - Risk of syntax errors and misconfigurations
2. **No Multi-Tenancy Support**
   - Cannot isolate alerts per customer
   - Single configuration affects all customers
   - No customer-specific customization
3. **Operational Risk**
   - Changes require Prometheus restart (downtime)
   - Editing one customer's alerts can break others
   - No validation before applying changes
4. **Scalability Concerns**

- Manual process doesn't scale beyond 5-10 customers
- Configuration drift across environments
- No audit trail of changes

**Business Requirements**

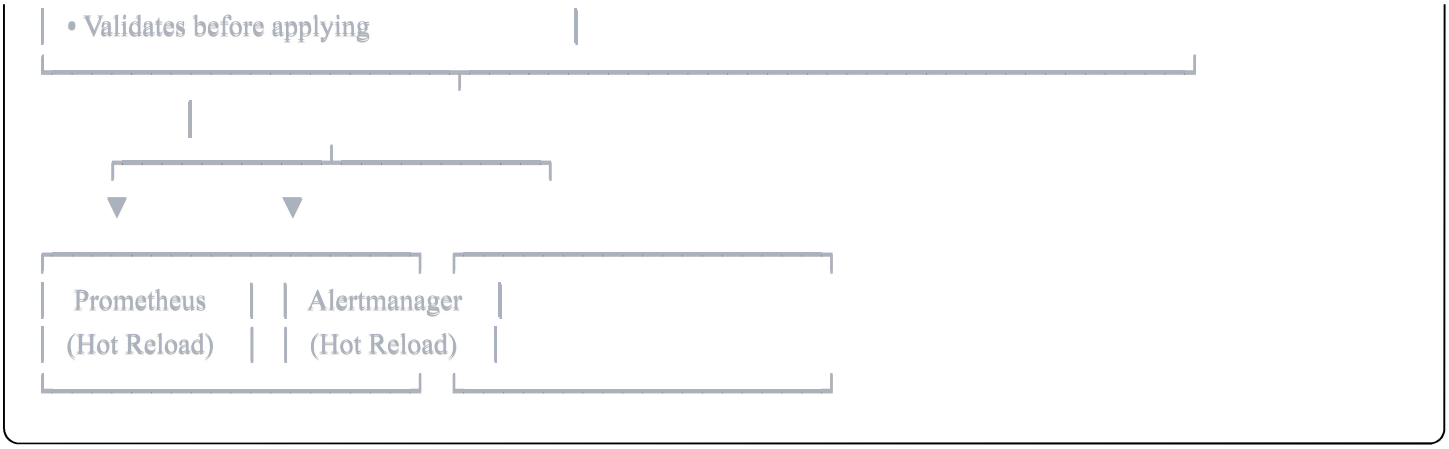
From User Story 1, the system must support:

- ✅ Configurable alert definitions (application & infrastructure)
- ✅ Per-customer thresholds and conditions
- ✅ Dynamic enable/disable without downtime
- ✅ Custom notification templates per customer
- ✅ Isolation between customers
- ✅ Audit trail of all changes

---

## Proposed Solution

### High-Level Architecture




**Core Components**

**1. PostgreSQL Database**

**Purpose:** Central source of truth for all alert configurations

**Stores:**

- Alert definitions with full PromQL expressions
- Customer-specific thresholds and parameters
- Notification receiver configurations
- Email/Slack/PagerDuty templates per customer
- Routing rules per customer
- Complete audit log of all changes

**Benefits:**

- Transactional consistency
- ACID guarantees
- Queryable audit trail
- Backup/restore capabilities

**2. Alert Management API**

**Purpose:** Provides programmatic interface for alert configuration

**Capabilities:**

- Create/Read/Update/Delete alert definitions
- Enable/disable alerts per customer
- Manage notification receivers

- Configure routing rules
- Validate PromQL expressions before saving
- Bulk import/export for migrations

**Benefits:**

- Version-controlled configuration
- API-first design for automation
- Input validation before persistence
- Role-based access control ready

**3. Configuration Sync Engine**

**Purpose:** Bridges database and Prometheus/Alertmanager

**Workflow:**

1. Triggered when alert configuration changes
2. Fetches all enabled alerts from database
3. Groups alerts by category (application/infrastructure)
4. Generates complete alerts.yml file in memory
5. Validates using Prometheus promtool
6. Writes atomically to filesystem
7. Triggers Prometheus hot reload via HTTP API
8. Repeats for Alertmanager configuration

**Benefits:**

- Automatic synchronization
- Validation before apply
- Atomic updates (all-or-nothing)
- Non-blocking operation

**4. Hot Reload Mechanism**

**Purpose:** Apply configuration changes without downtime

**How it works:**

- Prometheus provides `/-/reload` HTTP endpoint

- Accepts POST request to reload configuration
- Re-reads YAML files without restart
- Typically completes in 1-5 seconds
- Existing queries continue during reload

**Benefits:**

- Zero downtime
- No service interruption
- Fast propagation (seconds, not minutes)
- Built-in Prometheus capability

---

## Technical Approach

### Step 1: Database Schema Design

#### Alert Definitions Table

- Stores complete PromQL expressions (not decomposed)
- Includes customer_id for isolation
- Contains all metadata (severity, urgency, category)
- Tracks enabled/disabled status
- Records creation/modification timestamps

#### Notification Receivers Table

- Stores email addresses per customer
- Contains customizable email subject templates
- Contains customizable email body templates (HTML)
- Supports multiple channel types (email, Slack, PagerDuty)
- Links to specific customers

#### Routing Rules Table

- Defines which alerts go to which receivers
- Supports severity-based routing (critical vs warning)
- Supports category-based routing (application vs infrastructure)

- Priority-ordered for complex routing logic
- Customer-specific configuration

**Audit Log Table**

- Records every create/update/delete operation
- Captures before/after values
- Tracks who made the change and when
- Enables compliance and troubleshooting

**Step 2: API Implementation**

**REST Endpoints:**

- `POST /customers/{id}/alerts` - Create new alert
- `PUT /customers/{id}/alerts/{alertId}` - Update existing alert
- `PATCH /customers/{id}/alerts/{alertId}/enabled` - Toggle on/off
- `DELETE /customers/{id}/alerts/{alertId}` - Remove alert
- `GET /customers/{id}/alerts` - List all alerts
- `POST /customers/{id}/receivers` - Configure notifications
- `POST /customers/{id}/routing` - Configure routing

**Request Flow:**

1. API receives request with alert configuration
2. Validates PromQL syntax using Prometheus promtool
3. Validates customer_id is present in expression
4. Checks for duplicate alert names
5. Persists to database
6. Triggers async sync job
7. Returns immediately (non-blocking)
8. Sync completes in background (1-5 seconds)

**Validation:**

- PromQL expression syntax validation
- Required field validation
- Duplicate name detection

- Customer ID isolation enforcement
- Threshold value range checking

**Step 3: YAML Generation Process**

**Single File Approach:** All customers' alerts stored in one `alerts.yml` file, isolated by `customer_id` label.

**Generation Logic:**

1. Query database for all enabled alerts
2. Group by category (application_alerts, infrastructure_alerts)
3. For each alert:
    - Use PromQL expression exactly as stored
    - Add customer_id label
    - Add severity, urgency, category labels
    - Add annotations (summary, description, impact, action)
4. Build complete YAML structure in memory
5. Validate using promtool before writing
6. Write to temporary file
7. Atomically rename to alerts.yml

**Alertmanager Configuration:**

1. Query database for all customer receivers and routing rules
2. Build routing tree:
    - Root route groups by customer_id
    - Each customer gets sub-routes for severity/category
    - Each route points to customer-specific receiver
3. Generate receiver configurations:
    - Email receivers with custom templates
    - Slack receivers with webhooks
    - PagerDuty receivers with integration keys
4. Write to temporary file
5. Atomically rename to alertmanager.yml

**Atomic Updates:**

- Always write to temporary file first
- Validate before committing
- Use atomic rename operation
- Prevents partial/corrupted configurations

**Step 4: Hot Reload Execution**

**Prometheus Reload:**

1. Send HTTP POST to `http://prometheus:9090/-/reload`
2. Prometheus re-reads configuration files
3. Compiles new alert rules
4. Activates new rules within 1-5 seconds
5. Existing metrics collection continues uninterrupted
6. Query API remains available during reload

**Alertmanager Reload:**

1. Alertmanager watches config file for changes
2. Automatically reloads on file modification
3. Or send SIGHUP signal for immediate reload
4. Routing updates take effect within seconds
5. In-flight alerts continue processing

**Error Handling:**

- If validation fails, keep old configuration
- Log errors for troubleshooting
- Notify administrators of sync failures
- Implement retry logic with exponential backoff

**Step 5: Customer Isolation**

**Label-Based Isolation:** Every metric must include `customer_id` label:

```
http_request_duration_seconds{customer_id="customer-abc", service="nmt"}
```

**PromQL Expression Requirements:**

- All expressions must filter by customer_id
- API validates customer_id is present
- Prevents cross-customer data leakage
- Enables per-customer alerting

**Routing Isolation:**

- Alertmanager routes first by customer_id
- Each customer's alerts go to their receivers only
- Email templates are customer-specific
- Complete notification isolation

---

## Feasibility Analysis

**Technical Feasibility: ✅ HIGH**

**Evidence:**

1. **Prometheus Hot Reload is Standard**
   - Built-in capability since Prometheus 2.0
   - Used in production by thousands of companies
   - Well-documented and stable
2. **YAML File Size is Manageable**
   - Average alert rule: ~500 bytes
   - 100 customers × 20 alerts = 1 MB file
   - Prometheus handles 10,000+ rules easily
   - Reload time: 2-5 seconds for 10,000 rules
3. **Database-Driven Configuration is Proven**
   - Standard pattern in SaaS platforms
   - PostgreSQL handles this workload easily
   - Transactional guarantees ensure consistency
4. **API-First Approach is Industry Standard**
   - Similar to how Datadog, New Relic work
   - RESTful design is well-understood

- Easy to integrate with UI or automation

**Potential Challenges:**

- Learning curve for PromQL expressions
- Need to ensure customer_id in all metrics
- Monitoring the sync process itself

**Mitigations:**

- Provide PromQL templates for common patterns
- API validation enforces customer_id requirement
- Health checks and logging for sync process

**Operational Feasibility: ✅ HIGH**

**Benefits:**

1. **Zero Downtime Updates**
   - Hot reload means no service interruption
   - Changes apply in seconds
   - No maintenance windows required
2. **Customer Isolation**
   - One customer's changes don't affect others
   - Database transactions ensure consistency
   - Label-based filtering prevents data leakage
3. **Audit Trail**
   - Every change is logged with timestamp and user
   - Can roll back to previous configurations
   - Meets compliance requirements
4. **Validation Before Apply**
   - Invalid configurations are rejected
   - Production system protected from bad configs
   - Clear error messages for troubleshooting

**Operational Requirements:**

- Monitoring of sync process (alerts on sync failures)
- Backup strategy for database
- Documentation for support team

**Scalability: ✅ HIGH**

**Performance Characteristics:**

| Customers | Alerts/Customer | Total Alerts | File Size | Reload Time | Sync Time |
|---|---|---|---|---|---|
| 10 | 20 | 200 | ~100 KB | <1 sec | <1 sec |
| 50 | 20 | 1,000 | ~500 KB | 1-2 sec | 1-2 sec |
| 100 | 20 | 2,000 | ~1 MB | 2-3 sec | 2-3 sec |
| 500 | 20 | 10,000 | ~5 MB | 3-5 sec | 3-5 sec |

**When to Consider Alternatives:**

- 1,000+ customers: Evaluate Grafana Mimir (multi-tenant Prometheus with Rules API)
- 50,000+ rules: Consider sharding or recording rules
- 100+ changes per minute: Implement batching

**For POC Scale (10-100 customers):**

- Single file approach is optimal
- Hot reload is fast enough
- Database is not bottleneck
- No architectural changes needed

**Implementation Complexity: ✅ MEDIUM**

**Components to Build:**

1. Database schema (1-2 days)
2. REST API (3-5 days)
3. Sync engine (3-5 days)
4. Validation logic (2-3 days)
5. Web UI (5-7 days, optional for POC)