DevOps Monitoring

Prometheus, Grafana & Observability for Modern Applications

Prepared By: Rashi Rana
Corporate Trainer

Monitoring Fundamentals

> **Monitoring** is the practice of collecting, analyzing, and acting on data about your systems and applications to ensure reliability, performance, and availability.

Why Monitoring in DevOps?

- **Proactive Issue Detection:** Identify problems before they impact users
- **Performance Optimization:** Understand system behavior and bottlenecks
- **Capacity Planning:** Make informed decisions about resource allocation
- **Compliance & SLA:** Meet service level agreements and regulatory requirements
- **Root Cause Analysis:** Quickly diagnose and resolve incidents
- **Business Intelligence:** Gain insights into user behavior and system usage

Three Pillars of Observability

Metrics

Numerical data representing system state over time (CPU, memory, request rate)

Logs

Event details and contextual information about what happened in the system

Traces

Distributed request tracking showing the path of requests through microservices

Monitoring vs. Observability

Aspect	Monitoring	Observability
Focus	Known problems and predefined metrics	Understanding system behavior and unknown issues
Approach	Reactive - alerts when thresholds are breached	Proactive - enables exploration and investigation
Data	Aggregated metrics and dashboards	Rich context with metrics, logs, and traces
Questions	"Is the system working?"	"Why is the system behaving this way?"

Importance of Observability

- **Complex Systems:** Modern microservices architectures are too complex for traditional monitoring
- **Unknown Unknowns:** Ability to investigate issues you didn't anticipate
- **Faster MTTR:** Reduce Mean Time To Resolution with better context
- **Developer Experience:** Enable developers to understand their applications in production
- **Business Impact:** Connect technical metrics to business outcomes

> **Key Insight:** Observability is not just about collecting data - it's about making systems understandable and debuggable in production environments.

Metrics, Logs, and Alerts

Metrics: Numerical Data

> **Metrics** are numerical measurements that represent the state of your system at a specific point in time, collected at regular intervals.

Types of Metrics

Counter

Always increasing values (requests served, errors occurred)

Gauge

Values that can go up or down (CPU usage, memory consumption)

Histogram

Distribution of values (request duration, response sizes)

Summary

Quantiles and totals (95th percentile response time)

Logs: Event Details

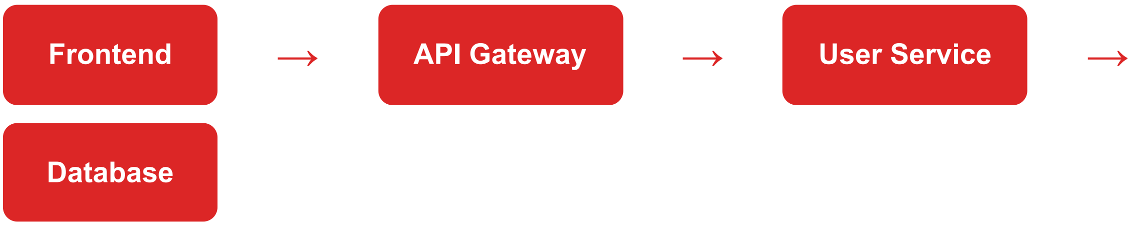
> **Logs** are timestamped records of events that occurred in your system, providing context and details about what happened.

Log Levels and Structure

```
# Structured Log Example
{
"timestamp": "2023-10-23T10:30:00Z",
"level": "ERROR",
"service": "user-service",
"message": "Failed to authenticate user",
"user_id": "12345",
"error_code": "AUTH_FAILED",
"request_id": "req-abc123"
}
```

Traces: Distributed Request Tracking

> **Traces** show the journey of a request through multiple services, helping understand performance bottlenecks and dependencies.



Frontend → API Gateway → User Service → Database

Alerts: Actionable Notifications

> **Alerts** notify you when something requires attention, enabling quick response to issues before they impact users.

Alert Types

Type	Description	Example
Threshold-based	Triggered when metric crosses predefined threshold	CPU usage > 80%
Anomaly-based	Triggered when behavior deviates from normal patterns	Request rate 3x higher than usual
Composite	Multiple conditions must be met	High CPU AND high memory

Type	Description	Example
Absence	Triggered when expected data is missing	No heartbeat for 5 minutes

Importance of Alerts

- **Early Warning:** Detect issues before they become critical
- **Automation:** Enable automated responses to common issues
- **Escalation:** Ensure the right people are notified at the right time
- **Documentation:** Create audit trail of system events

> **Alert Fatigue:** Too many alerts can desensitize teams. Focus on actionable alerts that require human intervention.

Prepared By: Rashi Rana
Corporate Trainer

SLA, SLO, SLI – Concepts

> Service Level concepts provide a framework for measuring and communicating service reliability and performance expectations.

Service Level Indicators (SLI)

What are SLIs?

Quantitative measures of service behavior - the actual metrics you measure

- **Availability:** Percentage of successful requests
- **Latency:** Time to process requests (95th percentile)
- **Throughput:** Requests processed per second
- **Error Rate:** Percentage of failed requests

Service Level Objectives (SLO)

What are SLOs?

Target values for SLIs - internal goals for service performance

- **Availability SLO:** 99.9% uptime (8.76 hours downtime/year)
- **Latency SLO:** 95% of requests < 200ms
- **Error Rate SLO:** < 0.1% error rate

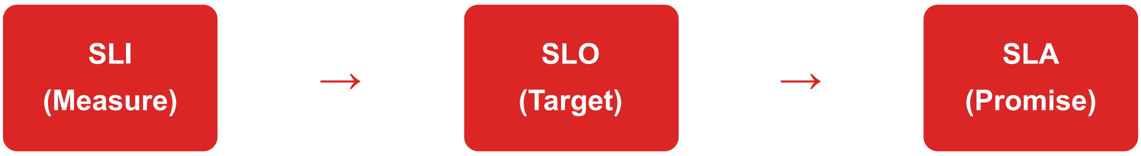
Service Level Agreements (SLA)

What are SLAs?

Contractual commitments to customers with consequences for not meeting them

- **Business Contract:** Legal agreement with customers
- **Penalties:** Credits or refunds for SLA breaches
- **Conservative:** Usually less strict than internal SLOs

Relationship Between SLI, SLO, and SLA



SLI (Measure) → SLO (Target) → SLA (Promise)

Example: E-commerce Website

Metric	SLI	SLO	SLA
Availability	Successful requests / Total requests	99.95% availability	99.9% availability
Latency	95th percentile response time	< 150ms for 95% of requests	< 200ms for 95% of requests
Error Rate	5xx errors / Total requests	< 0.05% error rate	< 0.1% error rate

Error Budget

> **Error Budget** = 100% - SLO. It represents how much unreliability you can tolerate while still meeting your objectives.

- **99.9% SLO = 0.1% Error Budget** (43.8 minutes/month)

- **Balance Innovation vs Reliability:** Spend error budget on new features
- **Freeze Deployments:** When error budget is exhausted

> **Best Practice:** Start with SLIs that matter to users, set realistic SLOs based on current performance, and make SLAs more conservative than SLOs.

Prepared By: Rashi Rana
Corporate Trainer

Prometheus Architecture and Components

What is Prometheus?

> **Prometheus** is an open-source monitoring system that collects metrics from applications and infrastructure, stores them as time-series data, and provides powerful querying capabilities.

Core Components

Prometheus Server

Core component that scrapes, stores metrics and serves queries

Exporters

Agents that expose metrics from systems (Node Exporter, MySQL Exporter)

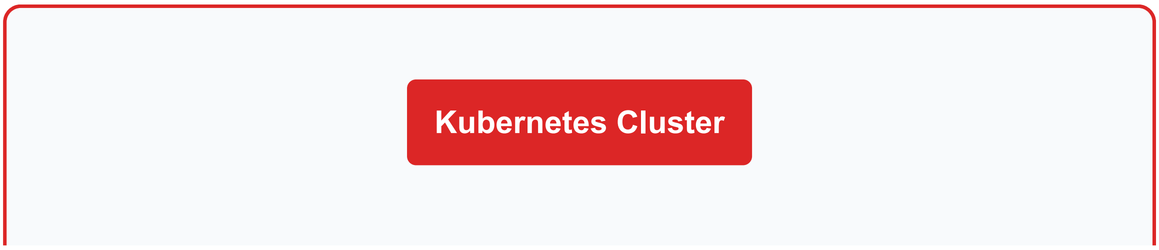
Pushgateway

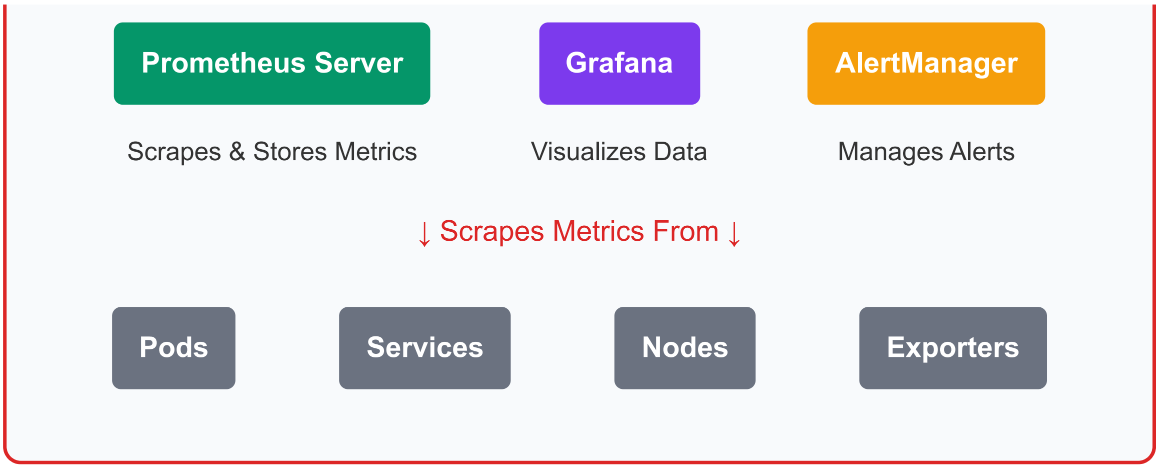
For short-lived jobs that can't be scraped directly

AlertManager

Handles alerts sent by Prometheus and routes them to receivers

Prometheus Architecture on Kubernetes





Prometheus Server

Scrapes & Stores Metrics

Grafana

Visualizes Data

AlertManager

Manages Alerts

↓ Scrapes Metrics From ↓

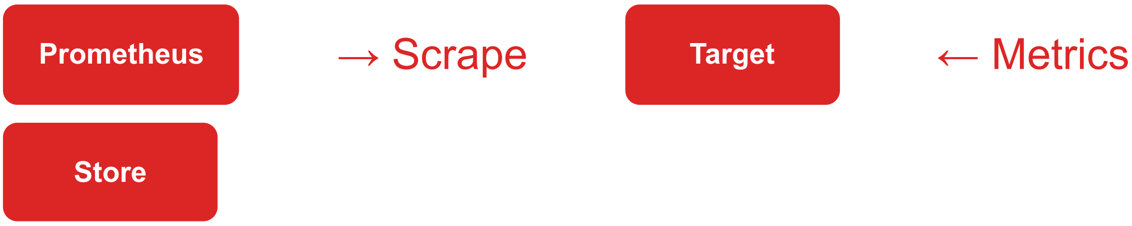
Pods

Services

Nodes

Exporters

How Prometheus Works (Pull Model)



Prometheus → Scrape Target ← Metrics

Store

Pull Model Benefits

- **Centralized Control:** Prometheus decides what and when to scrape
- **Service Discovery:** Automatically finds new targets in Kubernetes
- **Health Monitoring:** Knows if targets are unreachable
- **Security:** Targets don't need to know about Prometheus location

Grafana - Visualization Layer

> **Grafana** connects to Prometheus as a data source and creates beautiful, interactive dashboards for monitoring data visualization.

Key Grafana Features

- **Dashboard Creation:** Drag-and-drop interface for building dashboards
- **Multiple Data Sources:** Prometheus, InfluxDB, MySQL, etc.
- **Alerting:** Visual alerts with notification channels
- **Templating:** Dynamic dashboards with variables
- **Sharing:** Export/import dashboards as JSON

Perfect Team: Prometheus collects and stores metrics, Grafana visualizes them, and AlertManager handles notifications - creating a complete monitoring solution.

Prepared By: Rashi Rana
Corporate Trainer

Deploying Monitoring Stack with Helm

Why Helm for Monitoring?

> **Helm** simplifies Kubernetes deployments by packaging applications into charts. The kube-prometheus-stack chart includes Prometheus, Grafana, and AlertManager pre-configured.

kube-prometheus-stack Chart

Complete Stack

Prometheus, Grafana, AlertManager in one chart

Pre-configured

Ready-to-use dashboards and alerts

Kubernetes Native

ServiceMonitors, PrometheusRules CRDs included

Customizable

Override values for specific requirements

Step-by-Step Deployment

1 Add Helm Repository

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update
```

2 Create Namespace

```
kubectl create namespace monitoring
```

3 Install the Stack

```
helm install prometheus prometheus-community/kube-
prometheus-stack \
--namespace monitoring \
--set grafana.adminPassword=admin123
```

4 Verify Installation

```
kubectl get pods -n monitoring
kubectl get svc -n monitoring
```

What Gets Deployed

Component	Service Type	Default Port	Purpose
Prometheus	ClusterIP	9090	Metrics collection and storage
Grafana	ClusterIP	80	Dashboard and visualization
AlertManager	ClusterIP	9093	Alert routing and notifications
Node Exporter	DaemonSet	9100	Node-level metrics

Accessing the Services

```
# Access Prometheus UI
kubectl port-forward -n monitoring svc/prometheus-kube-prometheus-
prometheus 9090:9090
# Open: http://localhost:9090

# Access Grafana UI
kubectl port-forward -n monitoring svc/prometheus-grafana 3000:80
# Open: http://localhost:3000 (admin/admin123)

# Access AlertManager UI
```

```
kubectl port-forward -n monitoring svc/prometheus-kube-prometheus-alertmanager 9093:9093
# Open: http://localhost:9093
```

Custom Values Configuration

```yaml
# values.yaml
prometheus:
prometheusSpec:
retention: 30d
storageSpec:
volumeClaimTemplate:
spec:
storageClassName: standard
accessModes: ["ReadWriteOnce"]
resources:
requests:
storage: 50Gi

grafana:
adminPassword: mySecurePassword
service:
type: LoadBalancer
persistence:
enabled: true
size: 10Gi

alertmanager:
alertmanagerSpec:
storage:
volumeClaimTemplate:
spec:
storageClassName: standard
accessModes: ["ReadWriteOnce"]
resources:
requests:
storage: 10Gi
```

Deploy with Custom Values

```
# Install with custom configuration
helm install prometheus prometheus-community/kube-prometheus-stack \
--namespace monitoring \
--values values.yaml
```

```
# Upgrade existing installation
helm upgrade prometheus prometheus-community/kube-prometheus-stack
\
--namespace monitoring \
--values values.yaml
```

Production Ready: The Helm chart provides a production-ready monitoring stack with persistent storage, proper RBAC, and pre-configured dashboards!

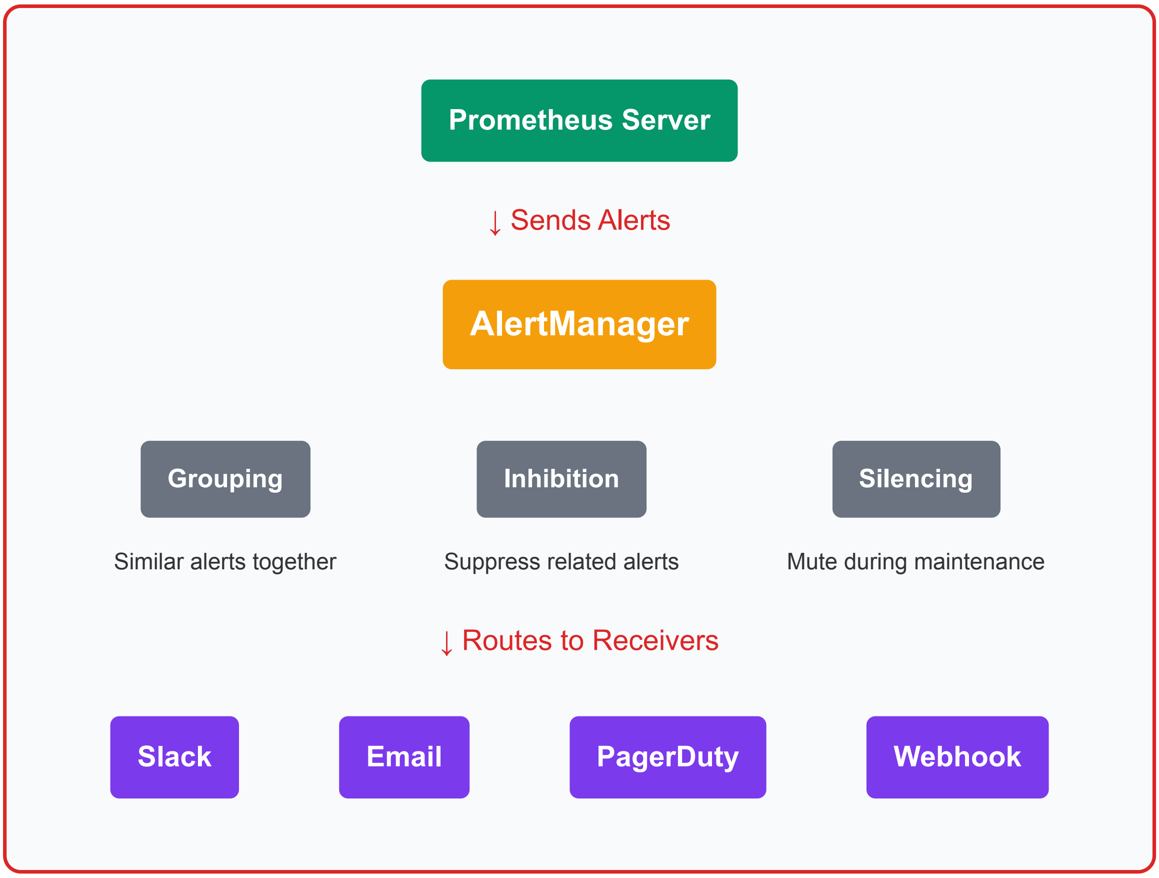
Prepared By: Rashi Rana
Corporate Trainer

AlertManager Architecture and Configuration

What is AlertManager?

> **AlertManager** receives alerts from Prometheus, groups them, and routes them to the correct notification channels like email, Slack, or PagerDuty.

AlertManager Architecture



Prometheus Server

↓ Sends Alerts

AlertManager

Grouping

Similar alerts together

Inhibition

Suppress related alerts

Silencing

Mute during maintenance

↓ Routes to Receivers

Slack

Email

PagerDuty

Webhook

Key AlertManager Features

Grouping

Routing

Combines similar alerts to reduce notification spam

Sends alerts to different teams based on labels

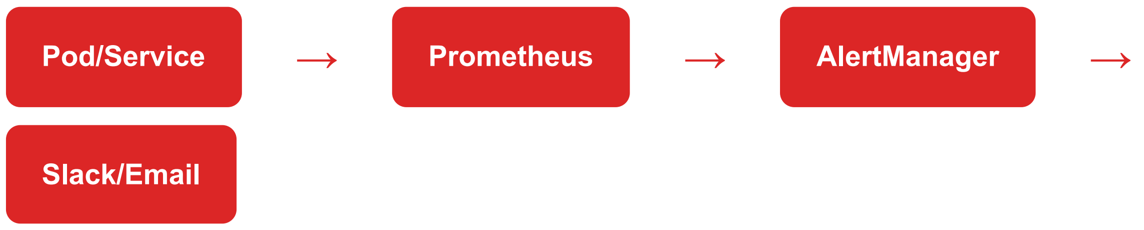
Inhibition

Suppresses alerts when higher priority ones are active

Silencing

Temporarily mute alerts during planned maintenance

Alert Flow in Kubernetes



Pod/Service → Prometheus → AlertManager → Slack/Email

Basic AlertManager Configuration

```
# alertmanager.yml
global:
smtp_smarthost: 'localhost:587'
smtp_from: 'alerts@company.com'

route:
group_by: ['alertname', 'cluster']
group_wait: 10s
group_interval: 10s
repeat_interval: 1h
receiver: 'web.hook'
routes:
- match:
severity: critical
receiver: 'critical-alerts'

receivers:
- name: 'web.hook'
webhook_configs:
- url: 'http://127.0.0.1:5001/'
- name: 'critical-alerts'
email_configs:
- to: 'admin@company.com'
```

```
subject: 'Critical Alert: {{ .GroupLabels.alertname }}'
body: 'Alert details: {{ range .Alerts }}{{ .Annotations.summary }}{{
end }}'
```

Alert States

State	Description	Action
Inactive	Alert condition is not met	No action taken
Pending	Condition met but waiting for 'for' duration	Monitoring, not yet firing
Firing	Condition met for specified duration	Alert sent to AlertManager

Creating Alert Rules

```
# prometheus-rules.yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
name: basic-alerts
namespace: monitoring
spec:
groups:
- name: basic.rules
rules:
- alert: HighCPUUsage
expr: 100 - (avg(irate(node_cpu_seconds_total{mode="idle"}[5m])) *
100) > 80
for: 2m
labels:
severity: warning
annotations:
summary: "High CPU usage on {{ $labels.instance }}"
description: "CPU usage is above 80% for more than 2 minutes"

- alert: PodCrashLooping
expr: rate(kube_pod_container_status_restarts_total[15m]) > 0
for: 5m
labels:
severity: critical
annotations:
summary: "Pod {{ $labels.pod }} is crash looping"
```

```
description: "Pod has restarted {{ $value }} times in the last 15
minutes"
```

Smart Alerting: AlertManager ensures you get the right alerts to the right people at the right time, without overwhelming your team with noise.

Prepared By: Rashi Rana
Corporate Trainer

Alertmanager and SLA Thresholds

Introduction to Alertmanager

> **Alertmanager** handles alerts sent by Prometheus server, taking care of deduplicating, grouping, and routing them to correct receivers like email, Slack, or PagerDuty.

Alertmanager Features

Grouping

Group similar alerts to reduce noise

Inhibition

Suppress alerts when other alerts are firing

Silencing

Temporarily mute alerts during maintenance

Routing

Send alerts to different teams based on labels

Creating Alerts in Prometheus

```
# prometheus-rules.yaml
groups:
- name: example-alerts
rules:
- alert: HighCPUUsage
expr: 100 - (avg by(instance)
(irate(node_cpu_seconds_total{mode="idle"}[5m])) * 100) > 80
for: 2m
labels:
severity: warning
```

```
team: infrastructure
annotations:
summary: "High CPU usage detected"
description: "CPU usage is above 80% for more than 2 minutes on {{
$labels.instance }}"

- alert: HighMemoryUsage
expr: (node_memory_MemTotal_bytes - node_memory_MemAvailable_bytes) /
node_memory_MemTotal_bytes * 100 > 85
for: 5m
labels:
severity: critical
team: infrastructure
annotations:
summary: "High memory usage detected"
description: "Memory usage is above 85% for more than 5 minutes on {{
$labels.instance }}"
```

Example Alert Rules

Alert	Expression	Threshold
CPU Alert	100 - (avg(irate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)	> 80%
Memory Alert	(1 - node_memory_MemAvailable_bytes/node_memory_MemTotal_bytes) * 100	> 85%
Disk Alert	(1 - node_filesystem_avail_bytes/node_filesystem_size_bytes) * 100	> 90%
Pod Restart	increase(kube_pod_container_status_restarts_total[1h])	> 5

SLA Thresholds and Use

> Set alert thresholds based on your SLOs to ensure you're notified before SLA breaches occur.

SLA-based Alert Strategy

- **Error Budget Alerts:** Alert when 50% of error budget is consumed
- **Burn Rate Alerts:** Alert on fast error budget consumption
- **Availability Alerts:** Alert when availability drops below SLO
- **Latency Alerts:** Alert when response times exceed SLO

```
# SLA-based alert example
- alert: ErrorBudgetBurn
expr: (
1 - (
sum(rate(http_requests_total{code!~"5.."}[5m])) /
sum(rate(http_requests_total[5m]))
)
) > 0.001 # 0.1% error rate (SLO breach)
for: 2m
labels:
severity: critical
annotations:
summary: "SLA breach: Error rate above 0.1%"
```

Creating Alerts from Grafana GUI

> Grafana provides a user-friendly interface to create alerts directly from dashboards without writing YAML files.

Steps to Create GUI Alerts

1. **Open Dashboard Panel:** Click on panel title → Edit
2. **Go to Alert Tab:** Click "Alert" tab in panel editor
3. **Create Alert Rule:** Click "Create Alert" button
4. **Set Conditions:** Define query, condition, and threshold
5. **Configure Notifications:** Select notification channels
6. **Save Alert:** Test and save the alert rule

GUI Alert Configuration

Query

Condition

Use existing panel query or create new one

IS ABOVE, IS BELOW, IS OUTSIDE RANGE

Evaluation

Set frequency and duration for alert evaluation

Notifications

Choose Slack, email, webhook channels

Example GUI Alert Setup

```
# GUI Alert Configuration Example:
Query: avg(cpu_usage_percent)
Condition: IS ABOVE 80
Evaluation: every 1m for 2m
Notification: #alerts-channel
Message: "CPU usage is {{ $value }}% on {{ $labels.instance }}"
```

> **Alert Tuning:** Start with conservative thresholds and adjust based on false positive rates and actual incidents.

Prepared By: Rashi Rana
Corporate Trainer

Hands-on Lab: Complete Monitoring Setup

Lab Objective

Deploy a complete monitoring stack with Prometheus, Grafana, and AlertManager on Kubernetes using Helm, then configure alerts and dashboards.

Lab Steps Overview

Step 1

Deploy monitoring stack with Helm

Step 2

Configure Slack integration

Step 3

Create custom alerts

Step 4

Build Grafana dashboard

Step 1: Deploy Monitoring Stack

```
# Add Helm repository
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
helm repo update

# Create namespace
```

```
kubectl create namespace monitoring

# Install complete stack
helm install prometheus prometheus-community/kube-prometheus-stack \
--namespace monitoring \
--set grafana.adminPassword=admin123 \
--set prometheus.prometheusSpec.retention=30d

# Verify installation
kubectl get pods -n monitoring
kubectl get svc -n monitoring
```

Step 2: Access Services

```
# Access Prometheus (Terminal 1)
kubectl port-forward -n monitoring svc/prometheus-kube-prometheus-prometheus 9090:9090
# Open: http://localhost:9090

# Access Grafana (Terminal 2)
kubectl port-forward -n monitoring svc/prometheus-grafana 3000:80
# Open: http://localhost:3000 (admin/admin123)

# Access AlertManager (Terminal 3)
kubectl port-forward -n monitoring svc/prometheus-kube-prometheus-alertmanager 9093:9093
# Open: http://localhost:9093
```

Step 3: Configure Slack Alerts

```
# Create alertmanager-config.yaml
apiVersion: v1
kind: Secret
metadata:
name: alertmanager-main
namespace: monitoring
stringData:
alertmanager.yml: |
global:
slack_api_url: 'https://hooks.slack.com/services/YOUR/SLACK/WEBHOOK'
route:
group_by: ['alertname']
group_wait: 10s
```

```
group_interval: 10s
repeat_interval: 1h
receiver: 'slack-notifications'
receivers:
- name: 'slack-notifications'
slack_configs:
- channel: '#monitoring'
title: 'Alert: {{ .GroupLabels.alertname }}'
text: '{{ range .Alerts }}{{ .Annotations.summary }}{{ end }}'
```

```
# Apply configuration
kubectl apply -f alertmanager-config.yaml

# Restart AlertManager
kubectl rollout restart statefulset/alertmanager-prometheus-kube-prometheus-alertmanager -n monitoring
```

Step 4: Create Custom Alert

```
# Create custom-alert.yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
name: custom-alerts
namespace: monitoring
labels:
prometheus: kube-prometheus
role: alert-rules
spec:
groups:
- name: custom.rules
rules:
- alert: HighCPUUsage
expr: 100 - (avg(irate(node_cpu_seconds_total{mode="idle"}[5m])) * 100) > 80
for: 2m
labels:
severity: warning
annotations:
summary: "High CPU usage detected"
description: "CPU usage is {{ $value }}% for more than 2 minutes"
```

```
# Apply custom alert
kubectl apply -f custom-alert.yaml
```

```
# Check if alert is loaded
kubectl get prometheusrules -n monitoring
```

Step 5: Create Grafana Dashboard

1. **Login to Grafana:** http://localhost:3000 (admin/admin123)
2. **Create Dashboard:** Click "+" → "Dashboard"
3. **Add Panel:** Click "Add new panel"
4. **Configure Query:**

   ```
   # CPU Usage Query
   100 - (avg(irate(node_cpu_seconds_total{mode="idle"}[5m]))
   * 100)

   # Memory Usage Query
   (1 - (node_memory_MemAvailable_bytes /
   node_memory_MemTotal_bytes)) * 100
   ```

5. **Save Dashboard:** Give it a name and save

> **Lab Complete!** You now have a fully functional monitoring stack with Prometheus collecting metrics, Grafana visualizing them, and AlertManager sending notifications to Slack.

Prepared By: Rashi Rana
Corporate Trainer

Summary and Key Takeaways

What We Learned

> You now understand how to implement a complete monitoring solution using Prometheus, Grafana, and AlertManager on Kubernetes with Helm.

Core Concepts Mastered

Monitoring Fundamentals

Three pillars: Metrics, Logs, Traces

SLA/SLO/SLI

Service level concepts for reliability

Prometheus Architecture

Pull-based metrics collection system

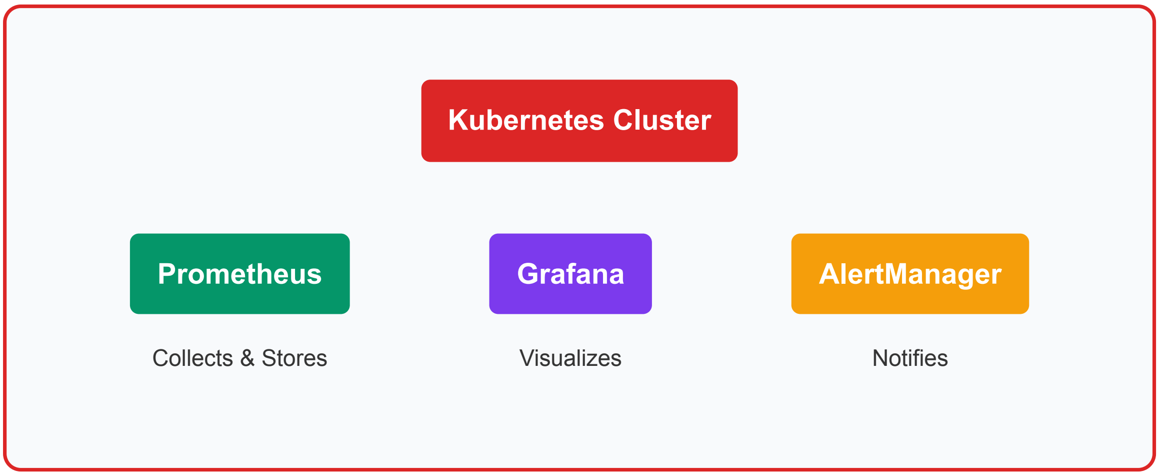
Grafana Dashboards

Visualization and alerting platform

Technical Skills Gained

- **Helm Deployment:** Deploy monitoring stack with kube-prometheus-stack chart
- **Alert Configuration:** Create PrometheusRules and configure AlertManager
- **Slack Integration:** Set up notification channels for team collaboration
- **Dashboard Creation:** Build custom Grafana dashboards with PromQL queries
- **Kubernetes Monitoring:** Monitor pods, services, and cluster resources

Architecture Understanding



Production Best Practices

- **Persistent Storage:** Configure storage for metrics retention
- **Resource Limits:** Set appropriate CPU/memory limits
- **High Availability:** Deploy multiple replicas for critical components
- **Security:** Enable authentication and RBAC
- **Backup:** Regular backup of Grafana dashboards and configurations

Next Steps for Learning

1. **Advanced PromQL:** Learn complex queries and functions
2. **Custom Exporters:** Create application-specific metrics exporters
3. **Distributed Tracing:** Implement Jaeger or Zipkin for microservices
4. **Log Management:** Add ELK/EFK stack for centralized logging
5. **Multi-cluster Monitoring:** Federated Prometheus setup

Common Commands Reference

```
# Deploy monitoring stack
helm install prometheus prometheus-community/kube-prometheus-stack
-n monitoring

# Access services
kubectl port-forward -n monitoring svc/prometheus-kube-prometheus-
prometheus 9090:9090
kubectl port-forward -n monitoring svc/prometheus-grafana 3000:80
kubectl port-forward -n monitoring svc/prometheus-kube-prometheus-
```

```
alertmanager 9093:9093

# Create custom alerts
kubectl apply -f prometheus-rules.yaml

# Check alert rules
kubectl get prometheusrules -n monitoring
```

Key Resources

- **Prometheus Documentation:** https://prometheus.io/docs/
- **Grafana Documentation:** https://grafana.com/docs/
- **Helm Charts:** https://prometheus-community.github.io/helm-charts/
- **PromQL Guide:** https://prometheus.io/docs/prometheus/latest/querying/basics/

> **Congratulations!** You're now equipped to implement comprehensive monitoring solutions for modern applications and infrastructure. Start monitoring your applications today!

Prepared By: Rashi Rana
Corporate Trainer