Hands-on Lab: Deploy Prometheus and Grafana on Kubernetes

Lab Overview

This comprehensive lab guide will walk you through deploying Prometheus and Grafana on an on-premises Kubernetes cluster for monitoring pod CPU and memory usage. You'll learn to set up a complete monitoring stack and create custom Grafana dashboards.

Estimated Time: 60-90 minutes

Difficulty Level: Intermediate

Table of Contents

1. Prerequisites and Environment Verification

1.1 Hardware Requirements

Component	Minimum	Recommended
CPU	2 cores	4 cores
Memory	4GB RAM	8GB RAM
Disk Space	20GB	50GB
Network	100 Mbps	1 Gbps

1.2 Software Requirements

- **Kubernetes Cluster**: Version 1.16 or higher (on-premises installation)
- **kubectl**: Latest version, configured and authenticated
- **Access Level**: Cluster-admin privileges required
- **Internet Access**: Required for pulling container images from Docker Hub

1.3 Pre-Deployment Verification Checklist

Execute the following commands to verify your environment:

Check 1: Verify kubectl Installation and Configuration

```
kubectl version --short
```

Expected Output:

```
Client Version: v1.28.x
Server Version: v1.28.x
```

Check 2: Verify Cluster Connectivity

```
kubectl cluster-info
```

Expected Output:

```
Kubernetes control plane is running at https://x.x.x.x:6443
CoreDNS is running at https://x.x.x.x:6443/api/v1/namespaces/kube-system/services/kube-dr
```

Check 3: Verify Node Status

```
kubectl get nodes
```

Expected Output:

```
NAME       STATUS   ROLES           AGE   VERSION
master-1   Ready    control-plane   10d   v1.28.x
worker-1   Ready    &lt;none&gt;          10d   v1.28.x
worker-2   Ready    &lt;none&gt;          10d   v1.28.x
```

All nodes should show **STATUS: Ready**

Check 4: Verify Cluster-Admin Permissions

```
kubectl auth can-i create clusterrole
kubectl auth can-i create clusterrolebinding
kubectl auth can-i create namespace
```

Expected Output for all commands:

```
yes
```

Check 5: Verify Available Resources

```
kubectl top nodes
```

Note: If metrics-server is not installed, this command will fail. This is acceptable for this lab.

Check 6: Test Internet Connectivity

```
curl -I https://hub.docker.com
```

Expected Output:

```
HTTP/2 200
```

1.4 Network Port Requirements

Service	Port	Protocol	Access Type
Prometheus	9090	TCP	Internal/NodePort 30000
Grafana	3000	TCP	Internal/NodePort 32000
NodePort Range	30000-32767	TCP	External (if using NodePort)

2. Part 1: Deploy Prometheus on Kubernetes

Prometheus is a powerful monitoring and alerting toolkit that collects and stores metrics as time-series data. In this section, we'll deploy Prometheus to monitor our Kubernetes cluster.

2.1 Architecture Overview

The Prometheus deployment consists of:

- **Prometheus Server**: Core component that scrapes and stores metrics
- **ConfigMap**: Stores Prometheus configuration and scrape jobs
- **RBAC Resources**: Service Account, ClusterRole, and ClusterRoleBinding
- **Service**: Exposes Prometheus UI

2.2 Step 1: Create Monitoring Namespace

Creating a dedicated namespace helps organize monitoring resources and isolate them from application workloads.

```
kubectl create namespace monitoring
```

Verify namespace creation:

```
kubectl get namespace monitoring
```

Expected Output:

```
NAME         STATUS   AGE
monitoring   Active   5s
```

2.3 Step 2: Create RBAC Resources for Prometheus

Prometheus needs permissions to access Kubernetes API to discover and scrape metrics from pods, services, and nodes.

Create file: `prometheus-rbac.yaml`

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
  - nodes
  - nodes/proxy
  - services
  - endpoints
  - pods
  verbs: ["get", "list", "watch"]
- apiGroups:
  - extensions
  resources:
  - ingresses
  verbs: ["get", "list", "watch"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
```

```
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: default
  namespace: monitoring
```

Understanding the RBAC Configuration:

- **ClusterRole**: Defines permissions across the entire cluster
- **get, list, watch**: Read-only permissions for discovering Kubernetes resources
- **nonResourceURLs**: Allows accessing the /metrics endpoint
- **ClusterRoleBinding**: Binds the role to the default service account in monitoring namespace

Apply the RBAC configuration:

```
kubectl apply -f prometheus-rbac.yaml
```

Verify RBAC creation:

```
kubectl get clusterrole prometheus
kubectl get clusterrolebinding prometheus
```

2.4 Step 3: Create Prometheus ConfigMap

The ConfigMap contains Prometheus configuration including scrape jobs for various Kubernetes components.

Create file: `prometheus-config-map.yaml`

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-server-conf
  labels:
    name: prometheus-server-conf
  namespace: monitoring
data:
  prometheus.rules: |-
    groups:
    - name: devopscube demo alert
      rules:
      - alert: High Pod Memory
        expr: sum(container_memory_usage_bytes) &gt; 1
        for: 1m
        labels:
          severity: slack
        annotations:
          summary: High Memory Usage
  prometheus.yml: |-
```

```
    global:
      scrape_interval: 15s
      evaluation_interval: 15s
    rule_files:
      - /etc/prometheus/prometheus.rules
    alerting:
      alertmanagers:
      - scheme: http
        static_configs:
        - targets:
          - "alertmanager.monitoring.svc:9093"
    scrape_configs:
      - job_name: 'node-exporter'
        kubernetes_sd_configs:
          - role: endpoints
        relabel_configs:
        - source_labels: [__meta_kubernetes_endpoints_name]
          regex: 'node-exporter'
          action: keep

      - job_name: 'kubernetes-apiservers'
        kubernetes_sd_configs:
        - role: endpoints
        scheme: https
        tls_config:
          ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
        bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
        relabel_configs:
        - source_labels: [__meta_kubernetes_namespace, __meta_kubernetes_service_name, __
          action: keep
          regex: default;kubernetes;https

      - job_name: 'kubernetes-nodes'
        scheme: https
        tls_config:
          ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
        bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
        kubernetes_sd_configs:
        - role: node
        relabel_configs:
        - action: labelmap
          regex: __meta_kubernetes_node_label_(.+)
        - target_label: __address__
          replacement: kubernetes.default.svc:443
        - source_labels: [__meta_kubernetes_node_name]
          regex: (.+)
          target_label: __metrics_path__
          replacement: /api/v1/nodes/${1}/proxy/metrics

      - job_name: 'kubernetes-pods'
        kubernetes_sd_configs:
        - role: pod
        relabel_configs:
        - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_scrape]
          action: keep
          regex: true
```

```
      - source_labels: [__meta_kubernetes_pod_annotation_prometheus_io_path]
        action: replace
        target_label: __metrics_path__
        regex: (.+)
      - source_labels: [__address__, __meta_kubernetes_pod_annotation_prometheus_io_por
        action: replace
        regex: ([^:]+)(?::\d+)?;(\d+)
        replacement: $1:$2
        target_label: __address__
      - action: labelmap
        regex: __meta_kubernetes_pod_label_(.+)
      - source_labels: [__meta_kubernetes_namespace]
        action: replace
        target_label: kubernetes_namespace
      - source_labels: [__meta_kubernetes_pod_name]
        action: replace
        target_label: kubernetes_pod_name

    - job_name: 'kube-state-metrics'
      static_configs:
        - targets: ['kube-state-metrics.kube-system.svc.cluster.local:8080']

    - job_name: 'kubernetes-cadvisor'
      scheme: https
      tls_config:
        ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
      bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
      kubernetes_sd_configs:
      - role: node
      relabel_configs:
      - action: labelmap
        regex: __meta_kubernetes_node_label_(.+)
      - target_label: __address__
        replacement: kubernetes.default.svc:443
      - source_labels: [__meta_kubernetes_node_name]
        regex: (.+)
        target_label: __metrics_path__
        replacement: /api/v1/nodes/${1}/proxy/metrics/cadvisor

    - job_name: 'kubernetes-service-endpoints'
      kubernetes_sd_configs:
      - role: endpoints
      relabel_configs:
      - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scrape]
        action: keep
        regex: true
      - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scheme]
        action: replace
        target_label: __scheme__
        regex: (https?)
      - source_labels: [__meta_kubernetes_service_annotation_prometheus_io_path]
        action: replace
        target_label: __metrics_path__
        regex: (.+)
      - source_labels: [__address__, __meta_kubernetes_service_annotation_prometheus_io
        action: replace
```

```
          target_label: __address__
          regex: ([^:]+)(?::\d+)?;(\d+)
          replacement: $1:$2
        - action: labelmap
          regex: __meta_kubernetes_service_label_(.+)
        - source_labels: [__meta_kubernetes_namespace]
          action: replace
          target_label: kubernetes_namespace
        - source_labels: [__meta_kubernetes_service_name]
          action: replace
          target_label: kubernetes_name
```

Understanding the Configuration:

- **global.scrape_interval**: How often Prometheus scrapes targets (15s)
- **global.evaluation_interval**: How often Prometheus evaluates alerting rules (15s)
- **scrape_configs**: Defines what endpoints to scrape
  - **kubernetes-apiservers**: Monitors Kubernetes API server
  - **kubernetes-nodes**: Monitors node-level metrics
  - **kubernetes-pods**: Auto-discovers pods with prometheus.io/scrape annotation
  - **kubernetes-cadvisor**: Collects container metrics
  - **kubernetes-service-endpoints**: Discovers services with prometheus.io/scrape annotation
  - **kube-state-metrics**: Monitors Kubernetes object states

Apply the ConfigMap:

```
kubectl apply -f prometheus-config-map.yaml
```

Verify ConfigMap creation:

```
kubectl get configmap prometheus-server-conf -n monitoring
kubectl describe configmap prometheus-server-conf -n monitoring
```

2.5 Step 4: Create Prometheus Deployment

The Deployment manages the Prometheus server pod with appropriate resource limits and volume mounts.

Create file: `prometheus-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prometheus-deployment
  namespace: monitoring
  labels:
    app: prometheus-server
```

```
spec:
  replicas: 1
  selector:
    matchLabels:
      app: prometheus-server
  template:
    metadata:
      labels:
        app: prometheus-server
    spec:
      containers:
        - name: prometheus
          image: prom/prometheus:latest
          args:
            - "--storage.tsdb.retention.time=12h"
            - "--config.file=/etc/prometheus/prometheus.yml"
            - "--storage.tsdb.path=/prometheus/"
          ports:
            - containerPort: 9090
          resources:
            requests:
              cpu: 500m
              memory: 500M
            limits:
              cpu: 1
              memory: 1Gi
          volumeMounts:
            - name: prometheus-config-volume
              mountPath: /etc/prometheus/
            - name: prometheus-storage-volume
              mountPath: /prometheus/
      volumes:
        - name: prometheus-config-volume
          configMap:
            defaultMode: 420
            name: prometheus-server-conf
        - name: prometheus-storage-volume
          emptyDir: {}
```

Understanding the Deployment:

- **replicas: 1**: Single Prometheus instance (increase for HA in production)
- **storage.tsdb.retention.time=12h**: Keeps metrics for 12 hours
- **resources**: CPU and memory limits prevent resource exhaustion
- **volumeMounts**:
  - Configuration from ConfigMap mounted at /etc/prometheus/
  - Storage volume for time-series data at /prometheus/
- **emptyDir**: Temporary storage (use PersistentVolume in production)

Apply the Deployment:

```
kubectl apply -f prometheus-deployment.yaml
```

Verify Deployment:

```
kubectl get deployment -n monitoring
kubectl get pods -n monitoring
```

Expected Output:

```
NAME                     READY   STATUS    RESTARTS   AGE
prometheus-deployment-xxx-yyy   1/1     Running   0          30s
```

Check Pod Logs:

```
kubectl logs -f -n monitoring $(kubectl get pods -n monitoring -l app=prometheus-server -
```

Expected Log Output:

```
level=info ts=... caller=main.go:... msg="Server is ready to receive web requests."
```

2.6 Step 5: Create Prometheus Service

Expose Prometheus using a NodePort service for external access.

Create file: `prometheus-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: prometheus-service
  namespace: monitoring
  annotations:
    prometheus.io/scrape: 'true'
    prometheus.io/port: '9090'
spec:
  selector:
    app: prometheus-server
  type: NodePort
  ports:
    - port: 8080
      targetPort: 9090
      nodePort: 30000
```

Understanding the Service:

- **type: NodePort**: Exposes service on all node IPs
- **port: 8080**: Service port inside the cluster

- **targetPort: 9090**: Container port where Prometheus listens
- **nodePort: 30000**: External port to access Prometheus
- **annotations**: Makes Prometheus scrape itself

Apply the Service:

```
kubectl apply -f prometheus-service.yaml
```

Verify Service:

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

Expected Output:

```
NAME                 TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
prometheus-service   NodePort   10.96.xxx.xxx   &lt;none&gt;        8080:30000/TCP   10s
```

2.7 Step 6: Access Prometheus Dashboard

Method 1: Using NodePort (Recommended for on-prem)

Get any node IP:

```
kubectl get nodes -o wide
```

Access Prometheus in your browser:

```
http://&lt;NODE-IP&gt;:30000
```

Method 2: Using Port Forwarding (For testing)

```
kubectl port-forward -n monitoring svc/prometheus-service 9090:8080
```

Access at: `http://localhost:9090`

2.8 Step 7: Verify Prometheus Targets

1. Open Prometheus UI
2. Navigate to **Status** → **Targets**
3. Verify the following jobs are present:
    - kubernetes-apiservers
    - kubernetes-nodes
    - kubernetes-pods

- kubernetes-cadvisor
- kubernetes-service-endpoints

Expected Status: Most targets should show **UP** status (some may be **DOWN** if components aren't deployed yet)

3. Part 2: Deploy Grafana on Kubernetes

Grafana is a visualization and analytics platform that allows you to query, visualize, and understand your metrics.

3.1 Architecture Overview

The Grafana deployment consists of:

- **Grafana Server**: Web application for dashboards
- **ConfigMap**: Pre-configures Prometheus as a data source
- **Service**: Exposes Grafana UI

3.2 Step 1: Create Grafana DataSource ConfigMap

This ConfigMap automatically configures Prometheus as a data source when Grafana starts.

Create file: `grafana-datasource-config.yaml`

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: grafana-datasources
  namespace: monitoring
data:
  prometheus.yaml: |-
    {
        "apiVersion": 1,
        "datasources": [
            {
               "access":"proxy",
                "editable": true,
                "name": "prometheus",
                "orgId": 1,
                "type": "prometheus",
                "url": "http://prometheus-service.monitoring.svc:8080",
                "version": 1
            }
        ]
    }
```

Understanding the DataSource Configuration:

- **access: proxy**: Grafana server makes requests to Prometheus (not browser)

- **editable: true**: Users can modify data source settings
- **type: prometheus**: Specifies Prometheus as the data source type
- **url**: Internal Kubernetes service URL for Prometheus

Apply the ConfigMap:

```
kubectl apply -f grafana-datasource-config.yaml
```

Verify ConfigMap:

```
kubectl get configmap grafana-datasources -n monitoring
kubectl describe configmap grafana-datasources -n monitoring
```

3.3 Step 2: Create Grafana Deployment

Create file: `grafana-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: grafana
  namespace: monitoring
spec:
  replicas: 1
  selector:
    matchLabels:
      app: grafana
  template:
    metadata:
      name: grafana
      labels:
        app: grafana
    spec:
      containers:
      - name: grafana
        image: grafana/grafana:latest
        ports:
        - name: grafana
          containerPort: 3000
        resources:
          limits:
            memory: "1Gi"
            cpu: "1000m"
          requests:
            memory: 500M
            cpu: "500m"
        volumeMounts:
          - mountPath: /var/lib/grafana
            name: grafana-storage
          - mountPath: /etc/grafana/provisioning/datasources
            name: grafana-datasources
```

```
            readOnly: false
      volumes:
        - name: grafana-storage
          emptyDir: {}
        - name: grafana-datasources
          configMap:
            defaultMode: 420
            name: grafana-datasources
```

Understanding the Deployment:

- **replicas: 1**: Single Grafana instance
- **image: grafana/grafana:latest**: Official Grafana image
- **containerPort: 3000**: Grafana's default port
- **volumeMounts**:
  - `/var/lib/grafana`: Grafana's data directory
  - `/etc/grafana/provisioning/datasources`: Auto-provisioned data sources
- **emptyDir**: Temporary storage (data lost on pod restart)

Apply the Deployment:

```
kubectl apply -f grafana-deployment.yaml
```

Verify Deployment:

```
kubectl get deployment grafana -n monitoring
kubectl get pods -n monitoring -l app=grafana
```

Expected Output:

```
NAME                       READY   STATUS    RESTARTS   AGE
grafana-xxxxxxxxxx-yyyyy   1/1     Running   0          20s
```

Check Pod Logs:

```
kubectl logs -f -n monitoring $(kubectl get pods -n monitoring -l app=grafana -o jsonpath
```

Expected Log Output:

```
logger=settings t=... lvl=info msg="Starting Grafana"
logger=http.server t=... lvl=info msg="HTTP Server Listen" address=[::]:3000
```

3.4 Step 3: Create Grafana Service

Create file: `grafana-service.yaml`

```
apiVersion: v1
kind: Service
metadata:
  name: grafana
  namespace: monitoring
  annotations:
    prometheus.io/scrape: 'true'
    prometheus.io/port: '3000'
spec:
  selector:
    app: grafana
  type: NodePort
  ports:
    - port: 3000
      targetPort: 3000
      nodePort: 32000
```

Understanding the Service:

- **type: NodePort**: Exposes Grafana on all node IPs
- **port: 3000**: Service port
- **targetPort: 3000**: Grafana container port
- **nodePort: 32000**: External port to access Grafana

Apply the Service:

```
kubectl apply -f grafana-service.yaml
```

Verify Service:

```
kubectl get svc grafana -n monitoring
```

Expected Output:

```
NAME      TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
grafana   NodePort   10.96.xxx.xxx   &lt;none&gt;        3000:32000/TCP   10s
```

3.5 Step 4: Access Grafana Dashboard

Method 1: Using NodePort

Get any node IP:

```
kubectl get nodes -o wide
```

Access Grafana in your browser:

```
http://&lt;NODE-IP&gt;:32000
```

Method 2: Using Port Forwarding

```
kubectl port-forward -n monitoring svc/grafana 3000:3000
```

Access at: `http://localhost:3000`

3.6 Step 5: Login to Grafana

Default Credentials:

- Username: `admin`
- Password: `admin`

First Login Steps:

1. Enter username: `admin`
2. Enter password: `admin`
3. You will be prompted to change the password
4. Set a new secure password
5. Click **Save**

3.7 Step 6: Verify Prometheus Data Source

1. In Grafana, click the ☰ **menu** (top left)
2. Go to **Connections** → **Data sources**
3. You should see **prometheus** in the list
4. Click on **prometheus**
5. Scroll down and click **Save & Test**
6. You should see: ✓ **Data source is working**

If data source test fails:

```
# Check Prometheus service is running
kubectl get svc prometheus-service -n monitoring

# Check Prometheus pod is running
kubectl get pods -n monitoring -l app=prometheus-server
```

```
# Test connectivity from Grafana pod to Prometheus
kubectl exec -n monitoring $(kubectl get pods -n monitoring -l app=grafana -o jsonpath='{
```

4. Part 3: Create Grafana Dashboard for Pod CPU and Memory Metrics

Now we'll create a dashboard to visualize pod CPU and memory usage using Prometheus metrics.

4.1 Understanding the Metrics

CPU Metrics:

- `container_cpu_usage_seconds_total`: Total CPU time consumed
- `rate(container_cpu_usage_seconds_total[5m])`: CPU usage rate over 5 minutes

Memory Metrics:

- `container_memory_usage_bytes`: Current memory usage in bytes
- `container_memory_working_set_bytes`: Current working set (more accurate for limits)

4.2 Method 1: Import Pre-built Dashboard

This is the fastest way to get a comprehensive Kubernetes monitoring dashboard.

Step 1: Navigate to Dashboard Import

1. In Grafana, click the ☰ **menu** (top left)
2. Go to **Dashboards**
3. Click **New** → **Import**

Step 2: Import Dashboard ID 315

1. In the **Import via** grafana.com field, enter: `315`
2. Click **Load**
3. Select **prometheus** as the data source
4. Click **Import**

Dashboard ID 315 provides:

- Cluster-wide CPU, memory, and filesystem usage
- Per-node resource utilization
- Individual pod metrics
- Container statistics

Alternative Dashboard IDs to Try:

Dashboard ID	Name	Description
315	Kubernetes Cluster Monitoring	Classic comprehensive dashboard

Dashboard ID	Name	Description
6417	Kubernetes Cluster	Modern design with detailed metrics
15760	Kubernetes / Views / Pods	Focused pod-level view
13838	Kubernetes Cluster	Clean interface with namespace filtering

Step 3: Explore the Dashboard

- Use the **Node** dropdown to filter by specific nodes
- Observe real-time CPU and memory usage
- Scroll through different panels showing various metrics

4.3 Method 2: Create Custom Dashboard from Scratch

For learning purposes, let's create a simple custom dashboard.

Step 1: Create New Dashboard

1. Click the ☰ **menu** (top left)
2. Go to **Dashboards**
3. Click **New** → **New Dashboard**
4. Click **Add visualization**
5. Select **prometheus** as data source

Step 2: Create CPU Usage Panel

1. Panel title: `Pod CPU Usage`
2. In the **Query** field, enter:

```
sum(rate(container_cpu_usage_seconds_total{pod!="", namespace!=""}[5m])) by (pod, namespa
```

3. **Explanation:**
   - `rate()`: Calculates per-second rate over 5 minutes
   - `pod!=""`: Excludes empty pod names
   - `sum() by (pod, namespace)`: Aggregates by pod and namespace
4. In the right panel:
   - **Legend**: `{{namespace}}/{{pod}}`
   - **Unit**: `short` or `cores`
   - **Graph type**: Time series
5. Click **Apply**

Step 3: Create Memory Usage Panel

1. Click **Add** → **Visualization**

2. Select **prometheus** as data source
3. Panel title: `Pod Memory Usage`
4. In the **Query** field, enter:

```
sum(container_memory_working_set_bytes{pod!="", namespace!=""}) by (pod, namespace) / 102
```

5. **Explanation:**
   - `container_memory_working_set_bytes`: Current memory usage
   - `/ 1024 / 1024`: Converts bytes to MB
   - `sum() by (pod, namespace)`: Aggregates by pod and namespace
6. In the right panel:
   - **Legend**: `{{namespace}}/{{pod}}`
   - **Unit**: `decmbytes` (MB)
   - **Graph type**: Time series
7. Click **Apply**

Step 4: Add Table View for Top Consumers

1. Click **Add** → **Visualization**
2. Select **prometheus** as data source
3. Panel title: `Top CPU Consumers`
4. **Visualization type**: Table
5. In the **Query** field, enter:

```
topk(10, sum(rate(container_cpu_usage_seconds_total{pod!="", namespace!=""}[5m])) by (poc
```

6. **Format**: Table
7. Click **Apply**

Step 5: Add Memory Table

1. Click **Add** → **Visualization**
2. Select **prometheus** as data source
3. Panel title: `Top Memory Consumers`
4. **Visualization type**: Table
5. In the **Query** field, enter:

```
topk(10, sum(container_memory_working_set_bytes{pod!="", namespace!=""}) by (pod, namespa
```

6. **Format**: Table
7. Click **Apply**

Step 6: Arrange Panels

1. Drag panels to arrange them in a logical layout
2. Resize panels by dragging corners
3. Suggested layout:
   - Top row: CPU Usage (left), Memory Usage (right)
   - Bottom row: Top CPU Consumers (left), Top Memory Consumers (right)

Step 7: Save Dashboard

1. Click the **Save dashboard** icon (top right)
2. Name: `Kubernetes Pod Metrics`
3. Add description (optional)
4. Click **Save**

4.4 Additional Useful Queries

Network I/O by Pod:

```
sum(rate(container_network_receive_bytes_total{pod!=""}[5m])) by (pod, namespace)
```

Pod Restart Count:

```
sum(kube_pod_container_status_restarts_total) by (pod, namespace)
```

CPU Throttling:

```
sum(rate(container_cpu_cfs_throttled_seconds_total{pod!=""}[5m])) by (pod, namespace)
```

Memory Limits vs Usage:

```
sum(container_memory_working_set_bytes{pod!=""}) by (pod, namespace) / sum(container_spec
```

5. Verification and Testing

5.1 Verify Complete Stack

Check all resources in monitoring namespace:

```
kubectl get all -n monitoring
```

Expected Output:

```
NAME                                        READY   STATUS    RESTARTS   AGE
pod/prometheus-deployment-xxxxxxxxxx-yyyyy   1/1     Running   0          10m
pod/grafana-xxxxxxxxxx-yyyyy                 1/1     Running   0          5m

NAME                          TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AG
service/prometheus-service   NodePort   10.96.xxx.xxx   &lt;none&gt;        8080:30000/TC
service/grafana             NodePort   10.96.xxx.xxx   &lt;none&gt;        3000:32000/TCP

NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/prometheus-deployment   1/1     1            1           10m
deployment.apps/grafana                 1/1     1            1           5m
```

5.2 Test Prometheus Metrics Collection

Access Prometheus UI:

```
http://&lt;NODE-IP&gt;:30000
```

Test Query 1: Check if metrics are being collected

Go to **Graph** tab and enter:

```
up
```

Click **Execute**. You should see multiple targets with value `1` (UP status).

Test Query 2: Check pod metrics

```
sum(rate(container_cpu_usage_seconds_total[5m])) by (pod)
```

Click **Execute**. You should see CPU usage for various pods.

Test Query 3: Check node metrics

```
node_memory_MemAvailable_bytes
```

If this returns no data, node-exporter needs to be deployed (covered in troubleshooting).

5.3 Test Grafana Dashboards

1. Access Grafana at `http://<NODE-IP>:32000`
2. Navigate to your dashboard
3. Set time range to **Last 15 minutes**
4. Verify:
    - CPU usage graphs show data

- Memory usage graphs show data
- Tables populate with pod information
- Legend shows pod names correctly

5.4 Generate Load for Testing

To see meaningful metrics, let's create test pods with resource usage.

Create a test deployment:

```
kubectl create deployment nginx-test --image=nginx --replicas=3
```

Add CPU-intensive pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: cpu-stress
  namespace: default
spec:
  containers:
  - name: stress
    image: polinux/stress
    command: ["stress"]
    args: ["--cpu", "2", "--timeout", "300s"]
    resources:
      requests:
        cpu: 100m
        memory: 100Mi
      limits:
        cpu: 500m
        memory: 200Mi
```

Save as `cpu-stress-pod.yaml` and apply:

```
kubectl apply -f cpu-stress-pod.yaml
```

Wait 2-3 minutes and check Grafana dashboard. You should see:

- Increased CPU usage for the `cpu-stress` pod
- Pod appearing in "Top CPU Consumers" table

5.5 Verification Checklist

Use this checklist to ensure everything is working:

- [ ] Prometheus pod is running and healthy
- [ ] Grafana pod is running and healthy

- [ ] Prometheus UI is accessible
- [ ] Grafana UI is accessible
- [ ] Can login to Grafana with credentials
- [ ] Prometheus data source is configured in Grafana
- [ ] Dashboard shows CPU metrics for pods
- [ ] Dashboard shows Memory metrics for pods
- [ ] Metrics update in real-time (auto-refresh)
- [ ] Can filter by namespace/pod in dashboards
- [ ] Test pods appear in metrics

6. Troubleshooting Guide

6.1 Common Issues and Solutions

Issue 1: Prometheus Pod Not Starting

Symptoms:

- Pod shows `CrashLoopBackOff` or `Error` status

Diagnosis:

```
kubectl describe pod -n monitoring $(kubectl get pods -n monitoring -l app=prometheus-ser
kubectl logs -n monitoring $(kubectl get pods -n monitoring -l app=prometheus-server -o j
```

Common Causes and Solutions:

a) Configuration Error in ConfigMap

```
# Validate YAML syntax
kubectl get configmap prometheus-server-conf -n monitoring -o yaml

# Check for indentation issues in prometheus.yml
```

b) Insufficient Resources

```
# Check node resources
kubectl top nodes

# Reduce resource requests in deployment if needed
```

c) Permission Issues

```
# Verify RBAC is created
kubectl get clusterrole prometheus
kubectl get clusterrolebinding prometheus
```

Issue 2: Grafana Shows "No Data" in Dashboards

Diagnosis Steps:

Step 1: Verify Prometheus is collecting metrics

1. Access Prometheus UI: `http://<NODE-IP>:30000`
2. Go to **Status** → **Targets**
3. Check if targets are UP

Step 2: Test Prometheus data source in Grafana

1. Go to **Connections** → **Data sources**
2. Click **prometheus**
3. Click **Save & Test**
4. Should show: "Data source is working"

Step 3: Test query directly in Prometheus

In Prometheus UI, run:

```
up
```

If this works but Grafana shows no data, the issue is with Grafana connection.

Solutions:

a) Incorrect Data Source URL

Edit Grafana datasource ConfigMap:

```
kubectl edit configmap grafana-datasources -n monitoring
```

Ensure URL is: `http://prometheus-service.monitoring.svc:8080`

Restart Grafana pod:

```
kubectl rollout restart deployment grafana -n monitoring
```

b) Network Policy Blocking Traffic

```
# Check if network policies exist
kubectl get networkpolicy -n monitoring
```

```
# If restrictive policies exist, add rule to allow Grafana → Prometheus
```

Issue 3: Missing Metrics (node_exporter, kube-state-metrics)

Symptoms:

- Dashboards show "N/A" or missing panels
- Queries like `node_memory_MemAvailable_bytes` return no data

Solution: Deploy kube-state-metrics

kube-state-metrics provides Kubernetes object metrics (pods, deployments, etc.)

```
# Deploy kube-state-metrics
kubectl apply -f https://raw.githubusercontent.com/kubernetes/kube-state-metrics/master/e
kubectl apply -f https://raw.githubusercontent.com/kubernetes/kube-state-metrics/master/e
```

Verify deployment:

```
kubectl get pods -n kube-system -l app.kubernetes.io/name=kube-state-metrics
```

Solution: Deploy Node Exporter

Node exporter provides node hardware metrics.

Create file: `node-exporter-daemonset.yaml`

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-exporter
  namespace: monitoring
  labels:
    app: node-exporter
spec:
  selector:
    matchLabels:
      app: node-exporter
  template:
    metadata:
      labels:
        app: node-exporter
    spec:
      hostNetwork: true
      hostPID: true
      containers:
      - name: node-exporter
        image: prom/node-exporter:latest
        args:
          - --path.procfs=/host/proc
```

```
          - --path.sysfs=/host/sys
          - --collector.filesystem.mount-points-exclude=^/(sys|proc|dev|host|etc)($$|/)
        ports:
        - containerPort: 9100
          hostPort: 9100
          name: metrics
        volumeMounts:
        - name: proc
          mountPath: /host/proc
          readOnly: true
        - name: sys
          mountPath: /host/sys
          readOnly: true
      volumes:
      - name: proc
        hostPath:
          path: /proc
      - name: sys
        hostPath:
          path: /sys
---
apiVersion: v1
kind: Service
metadata:
  name: node-exporter
  namespace: monitoring
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/port: "9100"
spec:
  type: ClusterIP
  ports:
  - name: metrics
    port: 9100
    targetPort: 9100
  selector:
    app: node-exporter
```

Apply:

```
kubectl apply -f node-exporter-daemonset.yaml
```

Verify:

```
kubectl get daemonset node-exporter -n monitoring
kubectl get pods -n monitoring -l app=node-exporter
```

Issue 4: Cannot Access Prometheus/Grafana UI

Symptoms:

- Browser shows "Connection refused" or timeout
- `http://<NODE-IP>:30000` not accessible

Diagnosis:

```
# Check services
kubectl get svc -n monitoring

# Check firewall rules on nodes
sudo iptables -L -n | grep 30000
sudo iptables -L -n | grep 32000
```

Solutions:

a) Firewall blocking NodePort

On each Kubernetes node:

```
# For iptables
sudo iptables -I INPUT -p tcp --dport 30000 -j ACCEPT
sudo iptables -I INPUT -p tcp --dport 32000 -j ACCEPT

# For firewalld
sudo firewall-cmd --permanent --add-port=30000/tcp
sudo firewall-cmd --permanent --add-port=32000/tcp
sudo firewall-cmd --reload
```

b) Wrong Node IP

Ensure you're using the correct node IP:

```
kubectl get nodes -o wide
```

Use the IP from the `EXTERNAL-IP` or `INTERNAL-IP` column.

c) Service Not Created Properly

```
# Delete and recreate service
kubectl delete svc prometheus-service -n monitoring
kubectl apply -f prometheus-service.yaml

kubectl delete svc grafana -n monitoring
kubectl apply -f grafana-service.yaml
```

Issue 5: Grafana Dashboard Import Fails

Symptoms:

- "Failed to import dashboard" error
- Dashboard shows blank panels

Solutions:

a) Data Source Not Selected

During import, ensure you select **prometheus** from the dropdown.

b) Manual JSON Import

1. Download dashboard JSON from grafana.com
2. In Grafana, go to **Dashboards → Import**
3. Paste JSON content
4. Select prometheus data source
5. Click Import

c) Dashboard Incompatibility

Try alternative dashboard IDs:

- 315 (Classic)
- 6417 (Modern)
- 13838 (Clean interface)

6.2 Debugging Commands

Check all resources:

```
kubectl get all -n monitoring
kubectl describe deployment prometheus-deployment -n monitoring
kubectl describe deployment grafana -n monitoring
```

Check logs:

```
# Prometheus logs
kubectl logs -f -n monitoring $(kubectl get pods -n monitoring -l app=prometheus-server -

# Grafana logs
kubectl logs -f -n monitoring $(kubectl get pods -n monitoring -l app=grafana -o jsonpath
```

Test internal connectivity:

```
# From Grafana pod, test Prometheus connectivity
kubectl exec -n monitoring $(kubectl get pods -n monitoring -l app=grafana -o jsonpath='{

# Should return: HTTP/1.1 200 OK
```

Check ConfigMaps:

```
kubectl get configmap -n monitoring
kubectl describe configmap prometheus-server-conf -n monitoring
kubectl describe configmap grafana-datasources -n monitoring
```

Restart pods:

```
kubectl rollout restart deployment prometheus-deployment -n monitoring
kubectl rollout restart deployment grafana -n monitoring
```

6.3 Performance Optimization

If Prometheus is using too much memory:

1. Reduce retention time:

Edit `prometheus-deployment.yaml`:

```
args:
  - "--storage.tsdb.retention.time=6h"  # Reduced from 12h
```

2. Reduce scrape interval:

Edit `prometheus-config-map.yaml`:

```
global:
  scrape_interval: 30s  # Increased from 15s
```

If Grafana is slow:

1. Increase memory limits:

Edit `grafana-deployment.yaml`:

```
resources:
  limits:
    memory: "2Gi"  # Increased from 1Gi
    cpu: "2000m"
```

2. Add caching:

```
kubectl set env deployment/grafana -n monitoring GF_CACHE_MEMORY_TTL=24h
```

7. Cleanup Instructions

When you're done with the lab, clean up resources.

7.1 Delete Test Pods

```
kubectl delete deployment nginx-test
kubectl delete pod cpu-stress
```

7.2 Delete Monitoring Stack

Option 1: Delete entire namespace (recommended)

```
kubectl delete namespace monitoring
```

This removes:

- Prometheus deployment, service, and config
- Grafana deployment, service, and config
- All ConfigMaps and secrets

Option 2: Delete individual components

```
# Delete Grafana
kubectl delete -f grafana-service.yaml
kubectl delete -f grafana-deployment.yaml
kubectl delete -f grafana-datasource-config.yaml

# Delete Prometheus
kubectl delete -f prometheus-service.yaml
kubectl delete -f prometheus-deployment.yaml
kubectl delete -f prometheus-config-map.yaml
kubectl delete -f prometheus-rbac.yaml

# Delete optional components
kubectl delete -f node-exporter-daemonset.yaml
```

7.3 Delete RBAC Resources

```
kubectl delete clusterrole prometheus
kubectl delete clusterrolebinding prometheus
```

7.4 Verify Cleanup

```
kubectl get all -n monitoring
kubectl get namespace monitoring
kubectl get clusterrole prometheus
kubectl get clusterrolebinding prometheus
```

All commands should return "No resources found" or "NotFound" errors.

8. Additional Resources and Next Steps

8.1 Enhance Your Setup

1. Add Persistent Storage

Replace `emptyDir` with PersistentVolumeClaim:

```
volumes:
  - name: prometheus-storage-volume
    persistentVolumeClaim:
      claimName: prometheus-pvc
```

2. Enable HTTPS/TLS

Use Ingress with TLS certificates for secure access.

3. Add Alertmanager

Configure alerts for critical conditions:

- High CPU usage
- High memory usage
- Pod restarts
- Node down

4. Implement Authentication

Add OAuth or LDAP authentication to Grafana.

5. High Availability

- Run multiple Prometheus replicas with Thanos
- Use external storage (S3, GCS)
- Deploy Grafana with database backend (PostgreSQL)

8.2 Useful Grafana Dashboard IDs

ID	Name	Best For
315	Kubernetes Cluster Monitoring	Complete cluster overview
6417	Kubernetes Cluster	Detailed pod metrics
13838	Kubernetes Cluster	Modern UI with filtering
1860	Node Exporter Full	Detailed node metrics
15760	Kubernetes / Views / Pods	Pod-focused view
893	Docker and System	Container metrics

8.3 Learning Resources

Official Documentation:

- Prometheus: https://prometheus.io/docs/
- Grafana: https://grafana.com/docs/
- Kubernetes Monitoring: https://kubernetes.io/docs/tasks/debug/

Prometheus Query Language (PromQL):

- PromQL Basics: https://prometheus.io/docs/prometheus/latest/querying/basics/
- PromQL Examples: https://prometheus.io/docs/prometheus/latest/querying/examples/

Community Dashboards:

- Grafana Dashboard Library: https://grafana.com/grafana/dashboards/

8.4 Certification Preparation

This lab covers concepts tested in:

- **Prometheus Certified Associate (PCA)**
- **Certified Kubernetes Administrator (CKA)**
- **Certified Kubernetes Application Developer (CKAD)**

9. Summary

What You Accomplished

In this lab, you successfully:

1. ✓ Deployed Prometheus on Kubernetes with proper RBAC configuration
2. ✓ Configured Prometheus to scrape Kubernetes metrics
3. ✓ Deployed Grafana on Kubernetes

4. ✓ Connected Grafana to Prometheus as a data source
5. ✓ Created or imported Grafana dashboards for pod CPU/memory monitoring
6. ✓ Verified metrics collection and visualization
7. ✓ Learned troubleshooting techniques

Key Concepts Learned

- **Prometheus Architecture**: Server, exporters, service discovery
- **Kubernetes Monitoring**: RBAC, ConfigMaps, services
- **PromQL**: Query language for metrics
- **Grafana**: Visualization and dashboards
- **cAdvisor**: Container metrics source
- **Service Discovery**: Auto-discovery of Kubernetes resources

Production Considerations

When moving to production:

1. Use PersistentVolumes for data persistence
2. Implement high availability with multiple replicas
3. Configure long-term storage (Thanos, Cortex)
4. Set up alerting with Alertmanager
5. Enable authentication and authorization
6. Use Ingress with TLS for secure access
7. Implement resource quotas and limits
8. Regular backup of Grafana dashboards
9. Monitor the monitoring stack itself
10. Document your alerting runbooks

Appendix A: Quick Reference Commands

Common kubectl Commands

```
# Get all resources in monitoring namespace
kubectl get all -n monitoring

# Describe a resource
kubectl describe &lt;resource-type&gt; &lt;resource-name&gt; -n monitoring

# View logs
kubectl logs -f &lt;pod-name&gt; -n monitoring
```

```
# Get pod shell
kubectl exec -it &lt;pod-name&gt; -n monitoring -- /bin/sh

# Port forward
kubectl port-forward -n monitoring svc/&lt;service-name&gt; &lt;local-port&gt;:&lt;servic

# Delete resource
kubectl delete &lt;resource-type&gt; &lt;resource-name&gt; -n monitoring

# Restart deployment
kubectl rollout restart deployment &lt;deployment-name&gt; -n monitoring
```

Useful PromQL Queries

```
# All targets status
up

# CPU usage by pod
sum(rate(container_cpu_usage_seconds_total{pod!=""}[5m])) by (pod, namespace)

# Memory usage by pod
sum(container_memory_working_set_bytes{pod!=""}) by (pod, namespace)

# Network received by pod
sum(rate(container_network_receive_bytes_total{pod!=""}[5m])) by (pod)

# Pod restart count
sum(kube_pod_container_status_restarts_total) by (pod, namespace)

# Node CPU usage
100 - (avg(rate(node_cpu_seconds_total{mode="idle"}[5m])) * 100)

# Node memory usage percentage
(1 - (node_memory_MemAvailable_bytes / node_memory_MemTotal_bytes)) * 100
```

Access URLs

```
# Prometheus (NodePort)
http://&lt;NODE-IP&gt;:30000

# Grafana (NodePort)
http://&lt;NODE-IP&gt;:32000

# Prometheus (Port Forward)
kubectl port-forward -n monitoring svc/prometheus-service 9090:8080
http://localhost:9090

# Grafana (Port Forward)
kubectl port-forward -n monitoring svc/grafana 3000:3000
http://localhost:3000
```

Appendix B: YAML File Repository

All YAML files used in this lab:

1. `prometheus-rbac.yaml` - RBAC for Prometheus
2. `prometheus-config-map.yaml` - Prometheus configuration
3. `prometheus-deployment.yaml` - Prometheus deployment
4. `prometheus-service.yaml` - Prometheus service
5. `grafana-datasource-config.yaml` - Grafana datasource
6. `grafana-deployment.yaml` - Grafana deployment
7. `grafana-service.yaml` - Grafana service
8. `node-exporter-daemonset.yaml` - Node Exporter (optional)
9. `cpu-stress-pod.yaml` - Test pod for generating load

GitHub Repository:

- Prometheus: https://github.com/techiescamp/kubernetes-prometheus
- Grafana: https://github.com/bibinwilson/kubernetes-grafana

Conclusion

Congratulations on completing this comprehensive lab on deploying Prometheus and Grafana on Kubernetes! You now have a functional monitoring stack that provides visibility into your cluster's health and performance.

This setup serves as a foundation for:

- Production-grade monitoring implementations
- Custom alerting configurations
- Advanced visualization techniques
- Performance optimization efforts

Continue exploring Prometheus and Grafana features to build robust observability into your Kubernetes infrastructure.

Happy Monitoring!