Grafana Alerting

With Multiple Instances



By, **M Junaid**

# Overview

This report explores the implementation of a monitoring and alerting system using **Grafana** and **Prometheus** in a **multi-instance environment**, containerized with **Docker Compose**. Prometheus collects metrics via **Node Exporter**, while Grafana serves as the **visualization and alerting layer**. Native Grafana alerting is configured per instance, with **Slack and email** used for notifications. **Host-mounted Docker volumes** ensure data persistence, and backups are transferred externally to support recovery. The setup enables **centralized observability** and **efficient alerting** across multiple monitored servers.

To achieve this, **Grafana and Prometheus are deployed on a centralized parent server**, while other monitored instances only run a **Node Exporter container via Docker Compose**, sending their metrics back to **Prometheus** located in the parent server for unified monitoring.

# 🧩 **Modules List – Grafana Alerting with Multiple Instances**

### Module 1: Monitoring Stack Setup Using Docker Compose

Setting up **Prometheus**, **Grafana**, **Node Exporter**, and **Alertmanager** as services.

### Module 2: Prometheus Configuration

Job configurations, service discovery, and health checks for multiple instances.

### Module 3: Open ports in security groups in both servers

Opening required ports in EC2 security groups on both parent and child servers.

### Module 4: Accessing Services via Public IP

Accessing Grafana, Prometheus, etc., through instance IP and exposed ports.

### Module 5: Grafana Data Source Integration

Adding **Prometheus** as a data source in **Grafana**.

### Module 6: Grafana Dashboard Setup

Importing **Node Exporter dashboard (ID: 1860)** and customizing it.

### Module 7: Creating Alert Rules in Grafana

Defining alert conditions for various metrics using **PromQL**.

### Module 8: Slack and Email Integration for Alerting

Setting up contact points and notification channels in **Grafana Alerting**.

### Module 9: Custom Alert & Resolve Templates

Creating and using templates for consistent and clear alert messages.

### Module 10: Persisting Data with Docker Volumes

Backing up container data and ensuring alert rules survive restarts.

### Module 11: Output and Verification

Testing dashboards, alert firing, notification delivery, and data recovery.

# 

# Module 1: Monitoring Stack Set-up Docker Compose

### 🔹 Objective:

To set up a containerized monitoring stack consisting of **Grafana, Prometheus**, **Node Exporter**, and optionally **Alertmanager** using **Docker Compose** for ease of deployment, networking, and management.

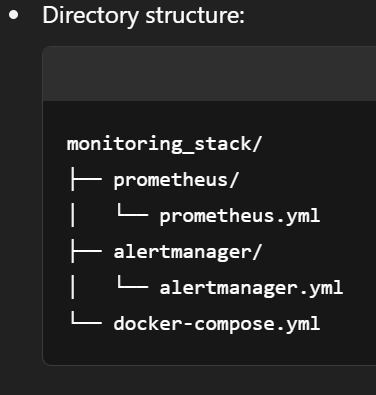
🔹 **What I Did:**

* Created a docker-compose.yml file defining services for:  
  + **Grafana** (port 3000)
  + **Prometheus** (port 9090)
  + **Node Exporter** (port 9100)
  + **Alertmanager** (port 9093 — later stopped as Grafana alerting was used)
* Configured a shared network (monitor-net) for inter-container communication.
* Mapped important ports to allow access from the public IP of the EC2 instance.
* Mounted host directories as volumes for each service to persist data across container restarts.
* Verified successful startup using docker-compose up -d.

🔹 **Key Files:**

* docker-compose.yml: Core configuration for deploying all services.

**Parent Server Monitoring stack folder Structure**



Source code reference:

version: '3.8'

networks:

monitor-net:

services:

prometheus-init:

image: alpine

container\_name: prometheus-init

command: ["sh", "-c", "chown -R 65534:65534 /prometheus"]

volumes:

- /home/ubuntu/data-backup/prometheus:/prometheus

entrypoint: ""

restart: "no"

networks:

- monitor-net

grafana-init:

image: alpine

container\_name: grafana-init

command: ["sh", "-c", "chown -R 472:472 /var/lib/grafana"]

volumes:

- /home/ubuntu/data-backup/grafana:/var/lib/grafana

entrypoint: ""

restart: "no"

networks:

- monitor-net

prometheus:

image: prom/prometheus:latest

container\_name: prometheus

depends\_on:

- prometheus-init

volumes:

- ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml

- ./prometheus/alert.rules.yml:/etc/prometheus/alert.rules.yml

- /home/ubuntu/data-backup/prometheus:/prometheus

command:

- '--config.file=/etc/prometheus/prometheus.yml'

- '--storage.tsdb.path=/prometheus'

ports:

- "9090:9090"

networks:

- monitor-net

grafana:

image: grafana/grafana:latest

container\_name: grafana

depends\_on:

- grafana-init

volumes:

- /home/ubuntu/data-backup/grafana:/var/lib/grafana

ports:

- "3000:3000"

networks:

- monitor-net

alertmanager:

image: prom/alertmanager:latest

container\_name: alertmanager

volumes:

- ./alertmanager/alertmanager.yml:/etc/alertmanager/alertmanager.yml

- ./alertmanager/template.tmpl:/etc/alertmanager/template.tmpl

- /home/ubuntu/data-backup/alertmanager:/alertmanager

command:

- '--config.file=/etc/alertmanager/alertmanager.yml'

ports:

- "9093:9093"

networks:

- monitor-net

node-exporter:

image: prom/node-exporter:latest

container\_name: node-exporter

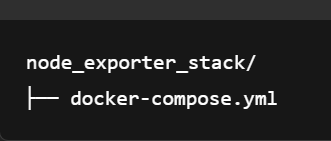
ports:

- "9100:9100"

networks:

- monitor-net

**Child-Server Directory Structure**



Source code reference: docker-compose.yml

version: '3.8'

services:

node-exporter:

image: prom/node-exporter:latest

container\_name: node-exporter

ports:

- "9100:9100"

restart: unless-stopped

Note: You **do not** need Prometheus, Grafana, or Alertmanager on the child servers.

### How It Works

* **Each child server** runs only node-exporter container to expose system metrics on port 9100.
* The **parent server** (which hosts Prometheus) must be configured with a scrape job pointing to each child’s **public/private IP** and :9100 port.

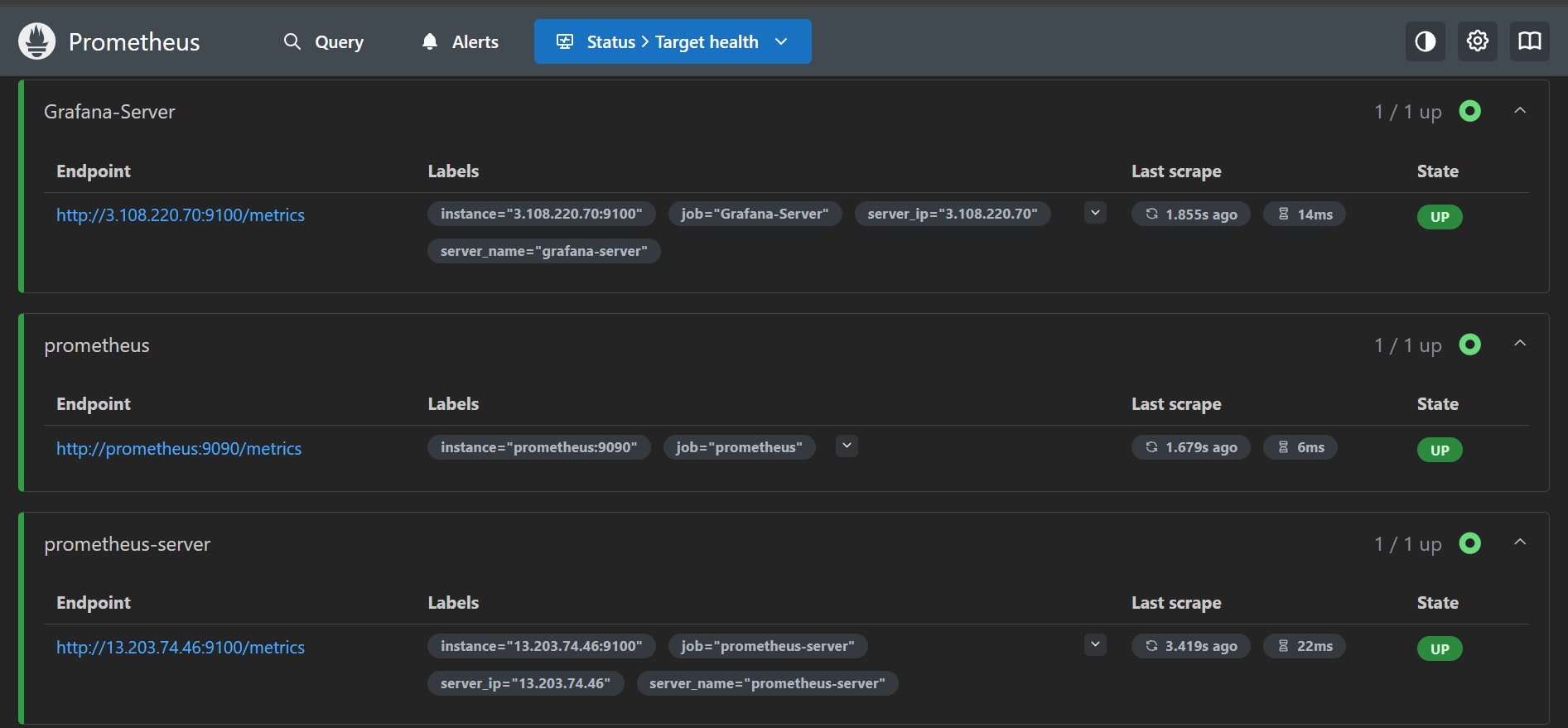
🔹 **Outcome:** A fully functional containerized monitoring environment that supports **observability**, **alerting**, and **persistent storage** — serving as the foundation for multi-instance metric collection and alert management.

# Module 2: Prometheus Configuration

🔹 **Objective:**

To configure **Prometheus** to monitor multiple instances by defining scrape jobs and validating service discovery.

🔹 **What I Did:**

* Edited prometheus.yml to add scrape jobs for:
  + Node Exporter (Grafana server)
  + Node Exporter (Prometheus server)
* Verified scrape targets using **Prometheus UI → Status > Targets**.
* Ensured job names were properly set for Grafana data source queries and alert scoping.

🔹 **Key Files:**

* prometheus.yml: Job definitions and scrape intervals.

Source code reference: prometheus.yml

global:

scrape\_interval: 5s

alerting:

alertmanagers:

- static\_configs:

- targets: ['alertmanager:9093']

scrape\_configs:

- job\_name: 'prometheus'

static\_configs:

- targets: ['prometheus:9090']

- job\_name: 'Grafana-Server'

static\_configs:

- targets: ['server-ip:9100']

labels:

server\_ip: 'ip-adress'

server\_name: 'grafana-server'

- job\_name: 'prometheus-server'

static\_configs:

- targets: ['server-ip:9100']

labels:

server\_ip: 'ip-adress'

server\_name: 'prometheus-server'

🔹 **Outcome:** Prometheus successfully collected metrics from both server instances with uniquely identifiable job labels.

# Module 3: Security Group Configuration

#### **🔹 Objective:**

Ensure that the monitoring stack components (Prometheus, Grafana, Node Exporter, Alertmanager) are accessible across multiple servers by configuring the necessary inbound rules in the **EC2 security groups**.

#### **🔹 What I Did:**

* **Parent Server (Main Monitoring Server):**
  + Opened port 3000 for **Grafana dashboard** access.
  + Opened port 9090 for **Prometheus UI** access.
  + Opened port 9093 (optional) for **Alertmanager UI**.
  + Allowed incoming traffic from all child server IPs for scraping metrics.
* **Child Servers (Node Exporter only):**
  + Opened port 9100 for Prometheus to **scrape system metrics**.
  + Ensured only the **parent server IP** can access this port for security.

#### **🔹 Security Group Rule Examples:**

| **Server** | **Port** | **Purpose** | **Source** |
| --- | --- | --- | --- |
| Parent Server | 9090 | Prometheus UI | 0.0.0.0/0 or My IP |
| Parent Server | 3000 | Grafana UI | 0.0.0.0/0 or My IP |
| Parent Server | 9093 | Alertmanager UI (optional) | 0.0.0.0/0 or My IP |
| Child Servers | 9100 | Node Exporter metrics | Parent Server IP only |

#### **🔹 Outcome:**

With these rules in place, the **parent Prometheus instance** is able to successfully scrape metrics from **Node Exporters** running on child servers, and the Grafana dashboard is accessible from a browser for visualization and alert management.

# Module 4: Accessing Services via Public IP

🔹 **Objective:** To allow access to Grafana, Prometheus, and other services via the EC2 instance’s **public IP** for remote monitoring and dashboard management.

🔹 **What I Did:**

* Verified security group rules in AWS to allow traffic on required ports.
* Accessed the following URLs:
  + Grafana: http://<EC2\_PUBLIC\_IP>:3000
  + Prometheus: http://<EC2\_PUBLIC\_IP>:9090
* Logged in to Grafana and Prometheus UI to verify everything was accessible externally.  
  

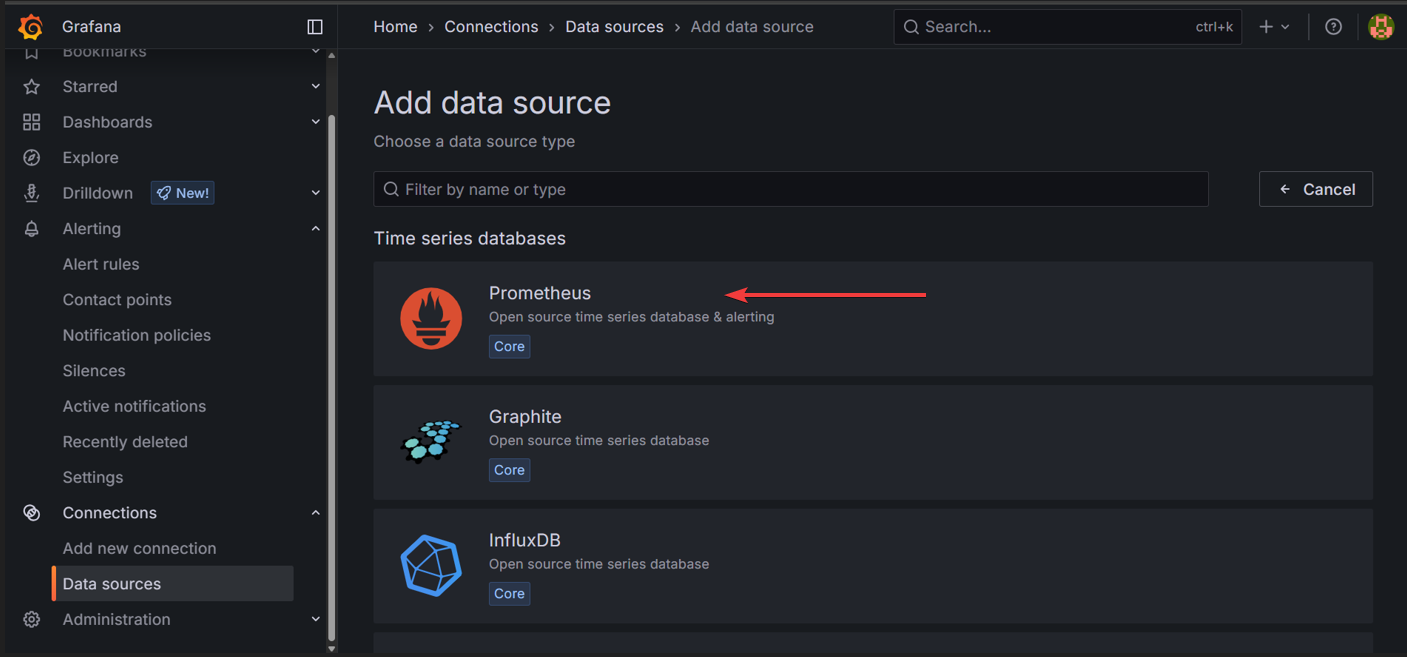
🔹 **Outcome:** All monitoring components were accessible through the public network, enabling dashboard access and system health checks from outside the server.

# Module 5: Grafana Data-Source Integration

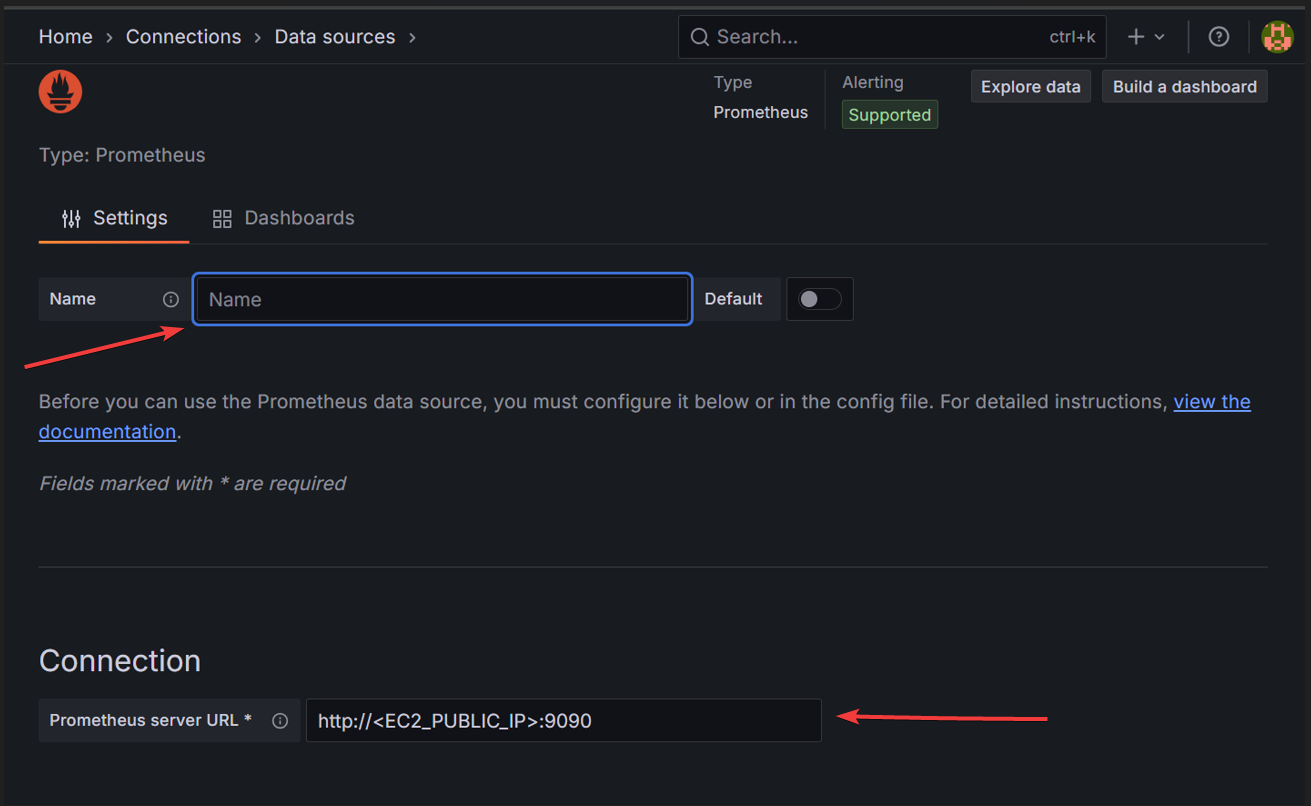
🔹 **Objective:** To connect **Grafana** with **Prometheus** as a data source for metric visualization.

🔹 **What I Did:**

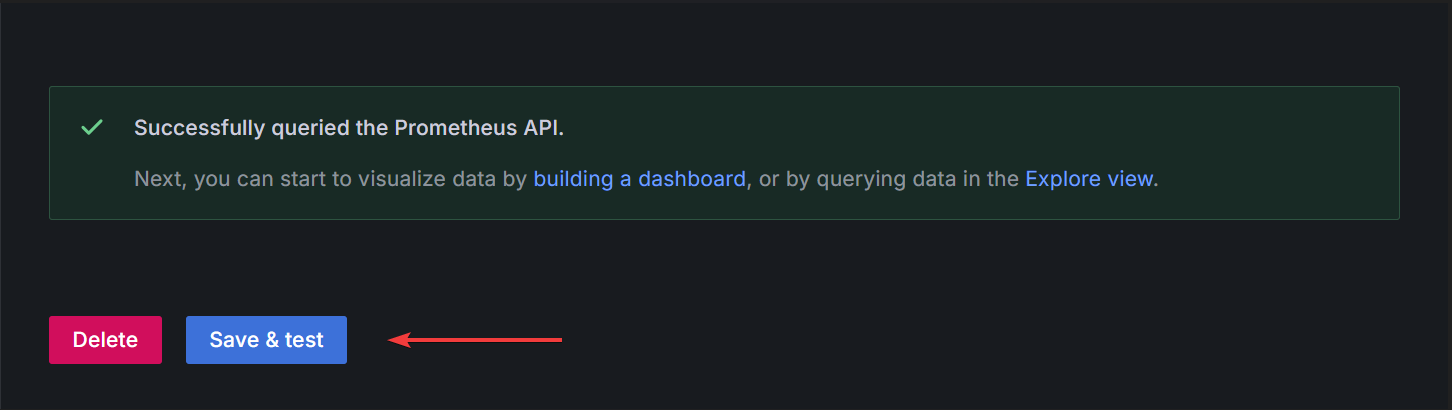
* Navigated to Grafana > Connections > Data Sources.
* Added **Prometheus** as a data source (http://<EC2\_PUBLIC\_IP>:9090).
* Click on **Name** → Provide your Data source name



* Click on Connection → Provide your monitoring server IP with port number of Prometheus.



* Tested the data source connection successfully.



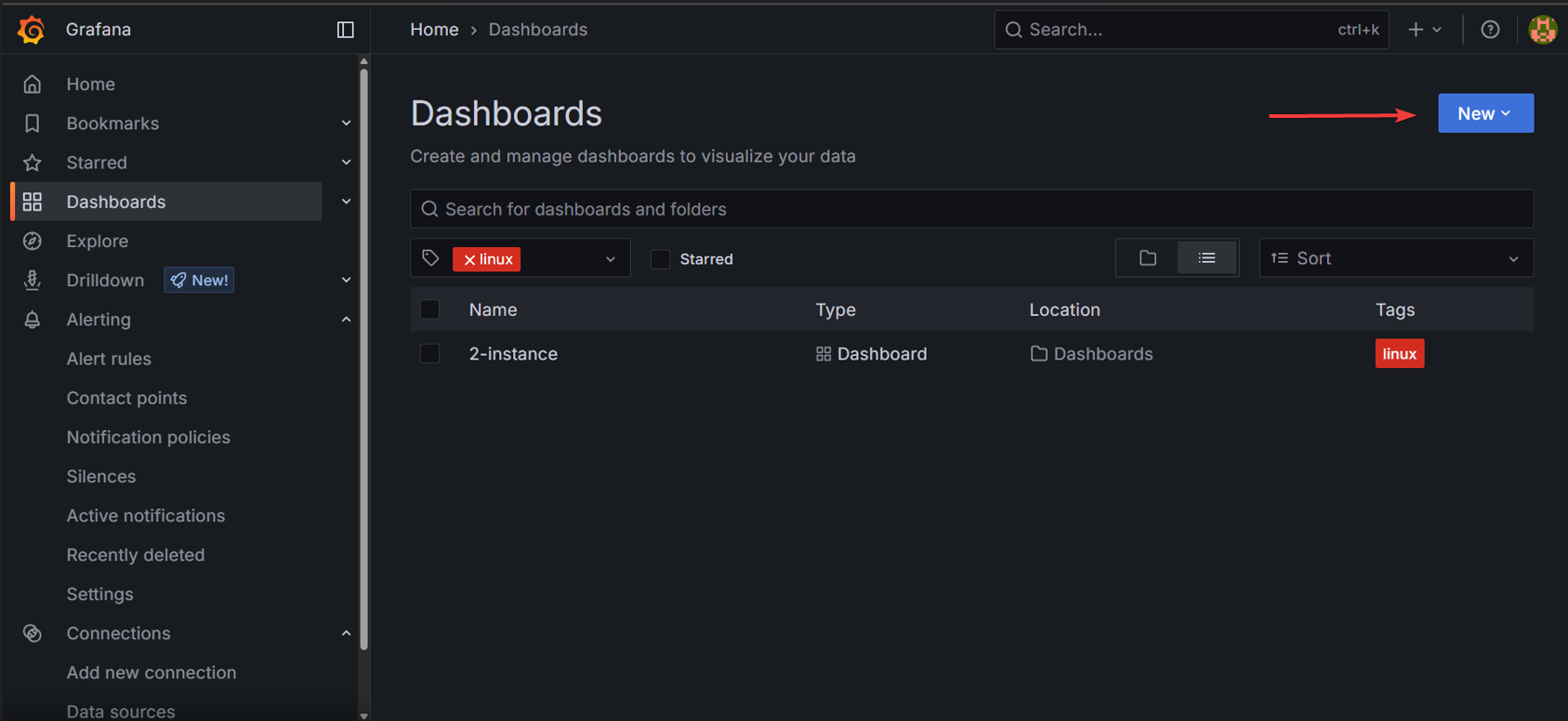
🔹 **Outcome:** Grafana was able to connect to Prometheus and fetch metrics from Prometheus, enabling real-time visualization for each job and instance.

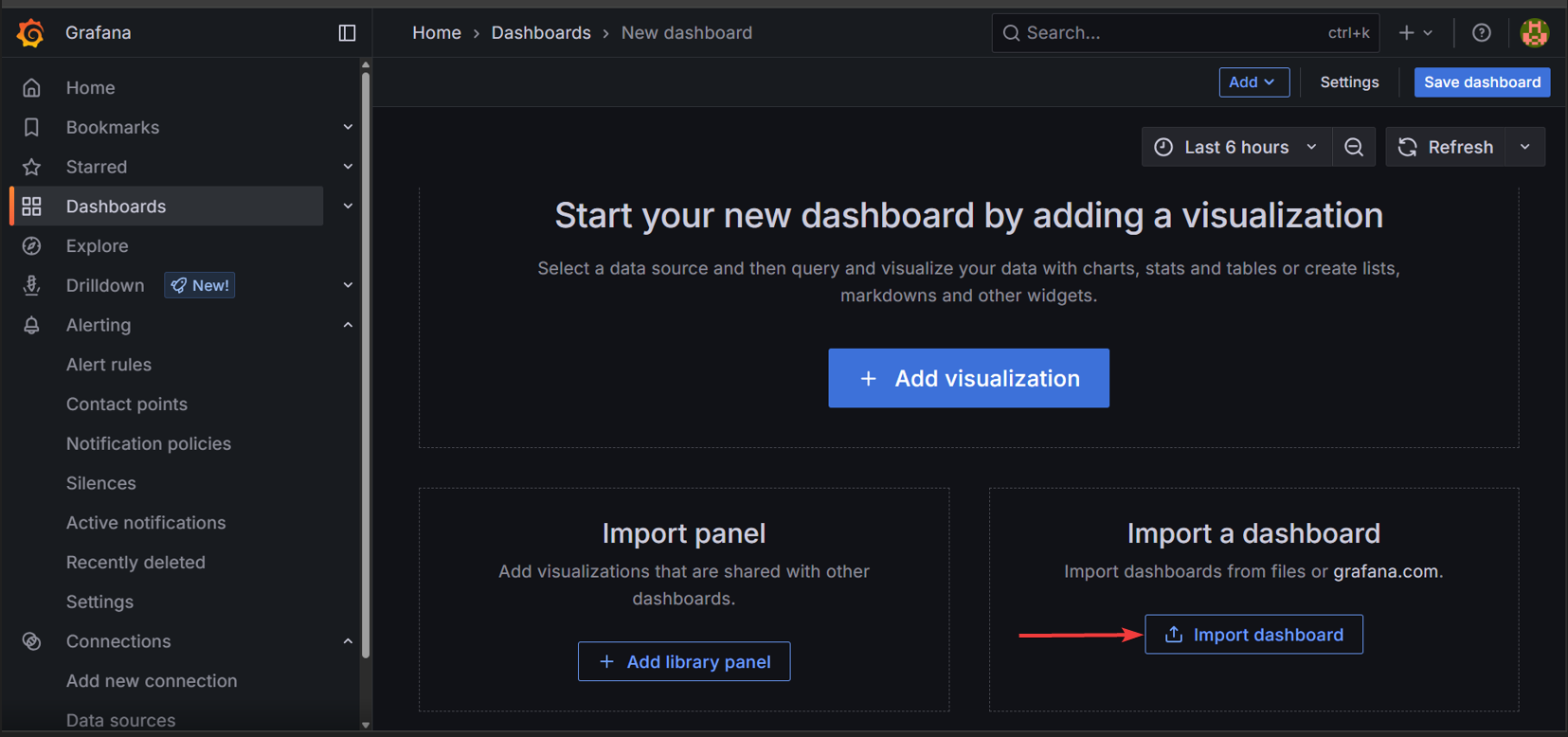
# Module 6: Grafana Dashboard Set-up

🔹 **Objective:** To visualize system performance metrics through pre-built dashboards in Grafana.

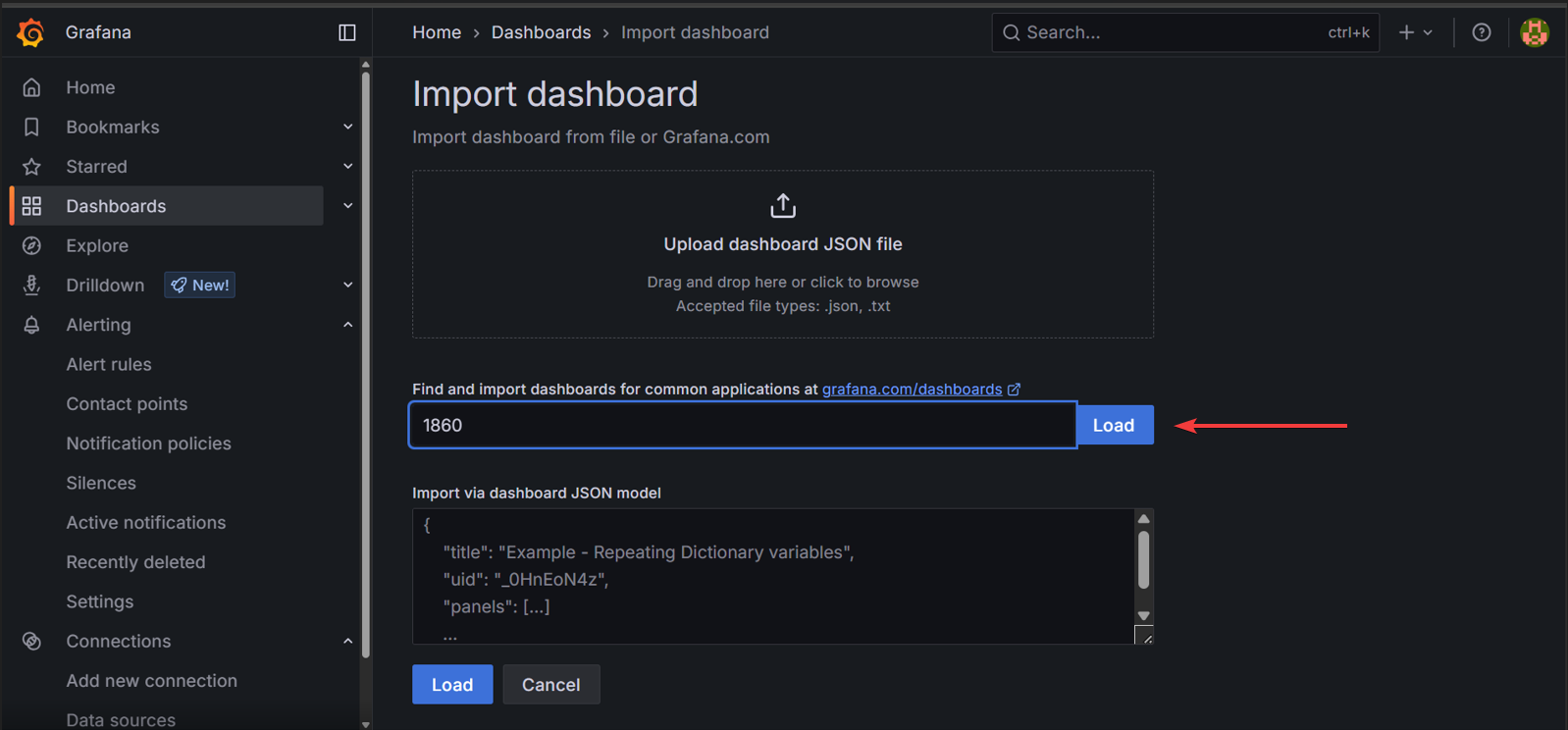
🔹 **What I Did:**

* Imported the **Node Exporter Full** dashboard (ID: 1860).
* **New → New Dashboard →** Click on **Import Dashboard**



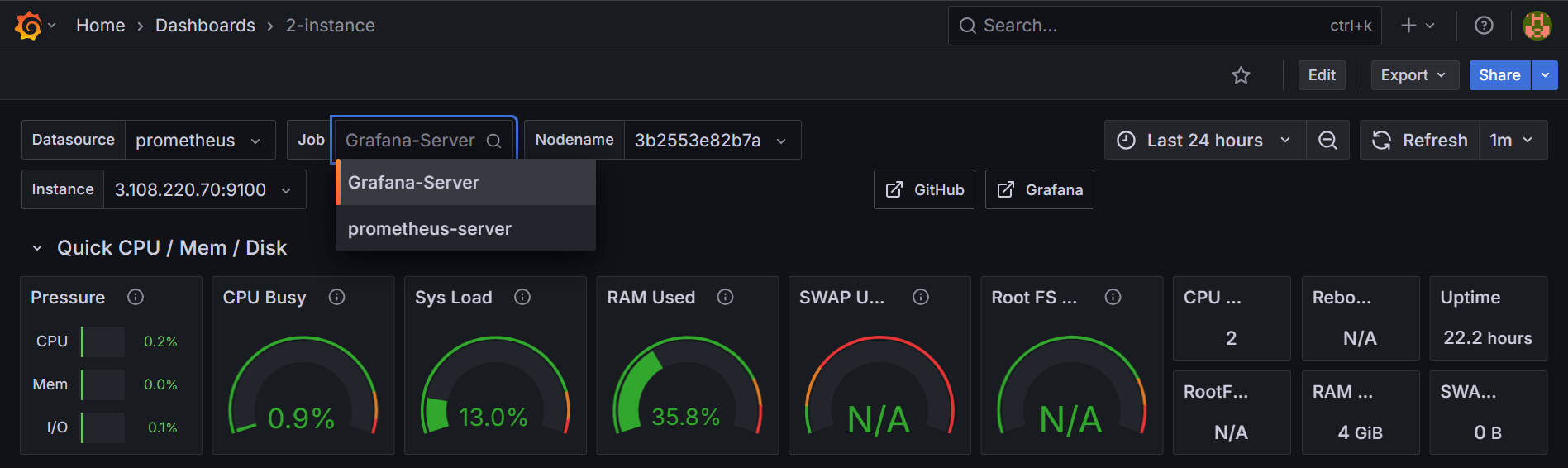


* Selected the Prometheus data source.



Note : Use code **(1860) → For Node Exporter Dashboard →** Click on **Load**

* Verified panels for CPU, Memory, and Disk metrics.



* Customized layout and titles for clarity.

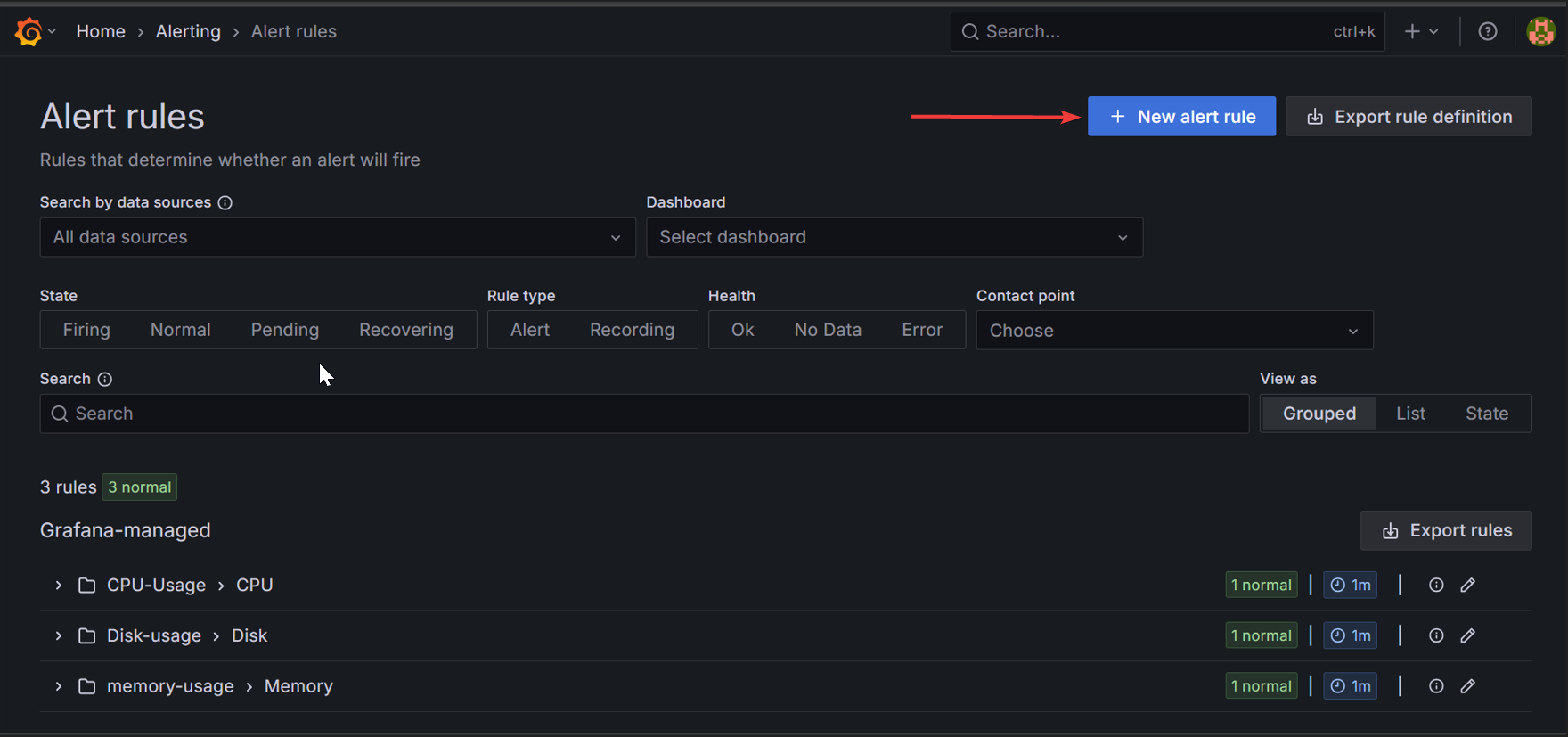
🔹 **Outcome:** Detailed dashboards were available for system monitoring, tailored for each monitored instance.

# Module 7: Creating Alert Rules in Grafana

🔹 **Objective:** To define **PromQL-based alert rules** in Grafana for tracking system resource usage.

🔹 **What I Did:**

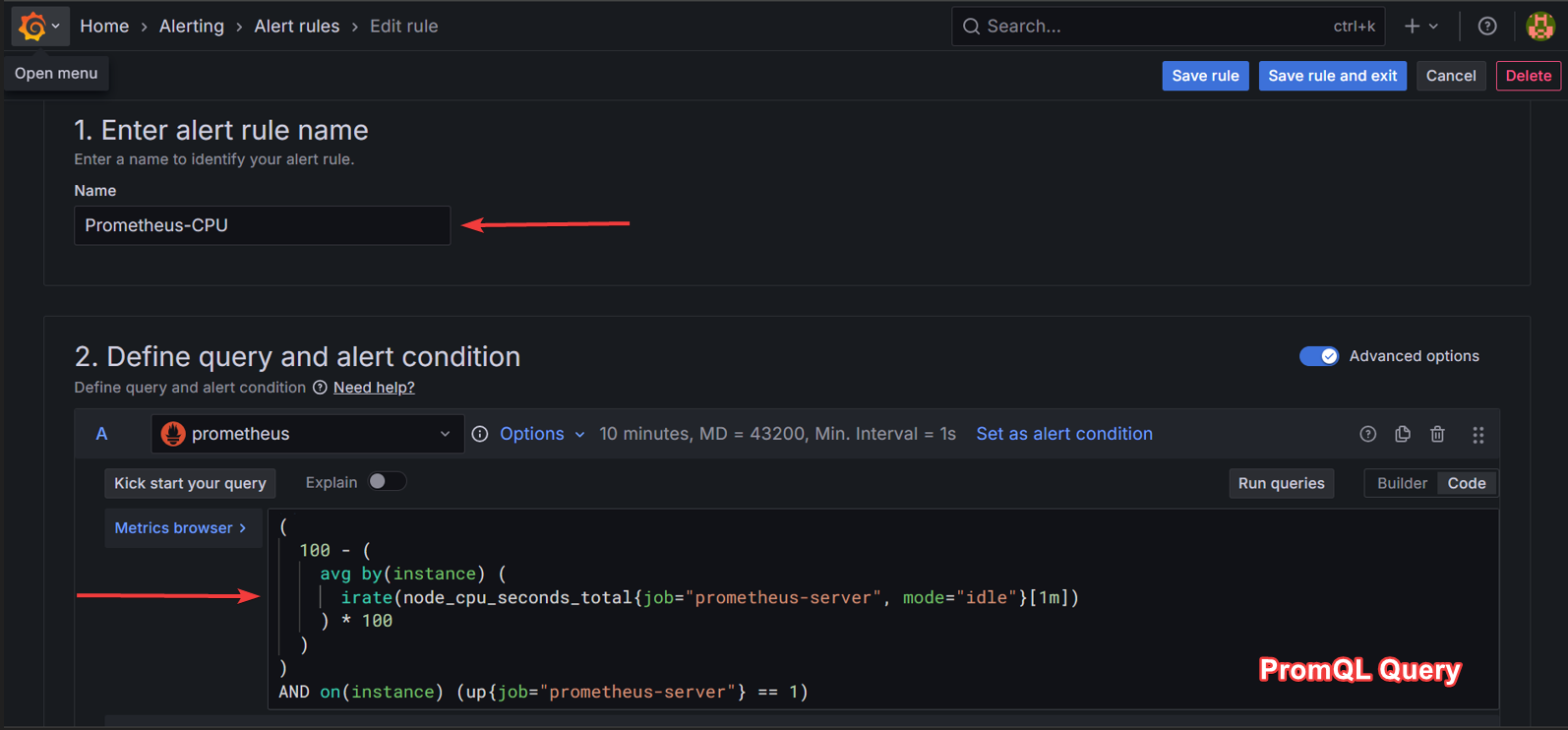
* Created separate alert rules for:
  + CPU usage
  + Memory usage
  + Disk usage
* Used expressions like irate(node\_cpu\_seconds\_total) and node\_memory\_MemAvailable\_bytes with thresholds.
* Scoped each alert rule using job, instance, or device labels.

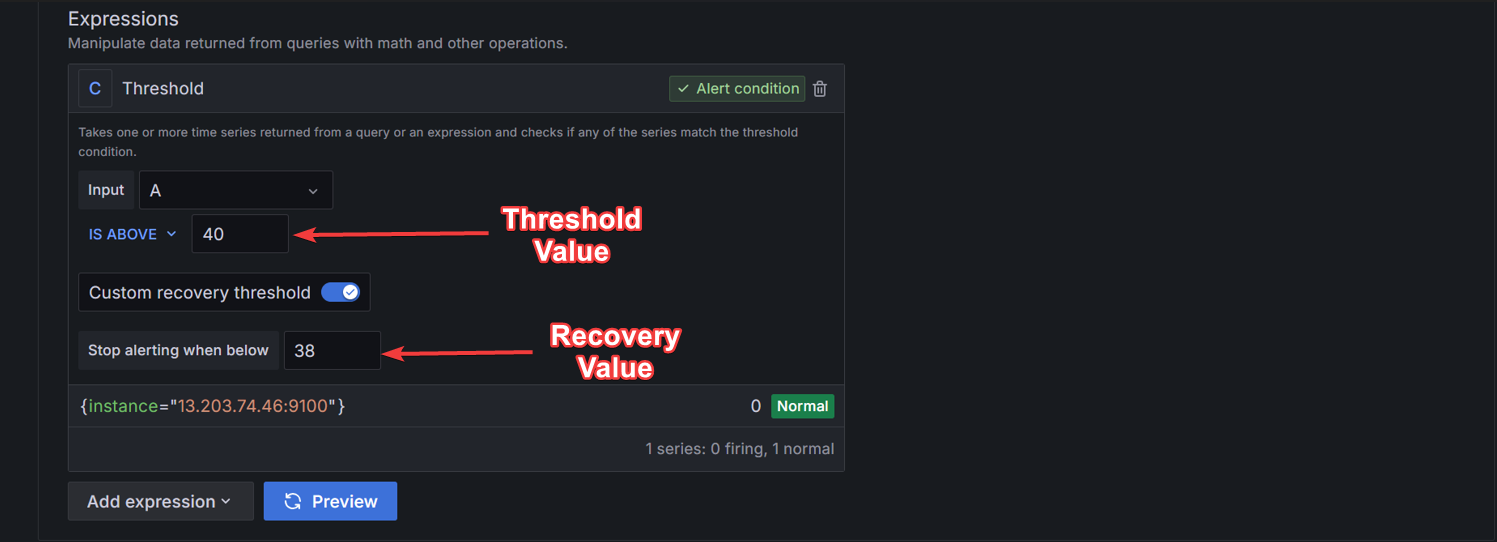
****

Click on **New alert rule →** to create new alert rule.

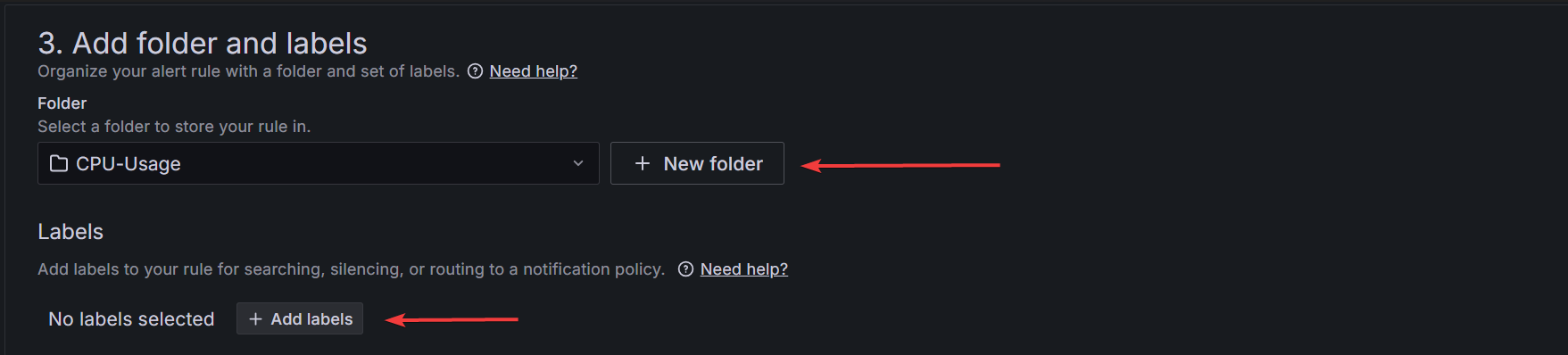
**Steps to Create Alert-rule:**

—> Provide **alert rule name** → Provide **PromQL Query** as a condition for your alert rule.



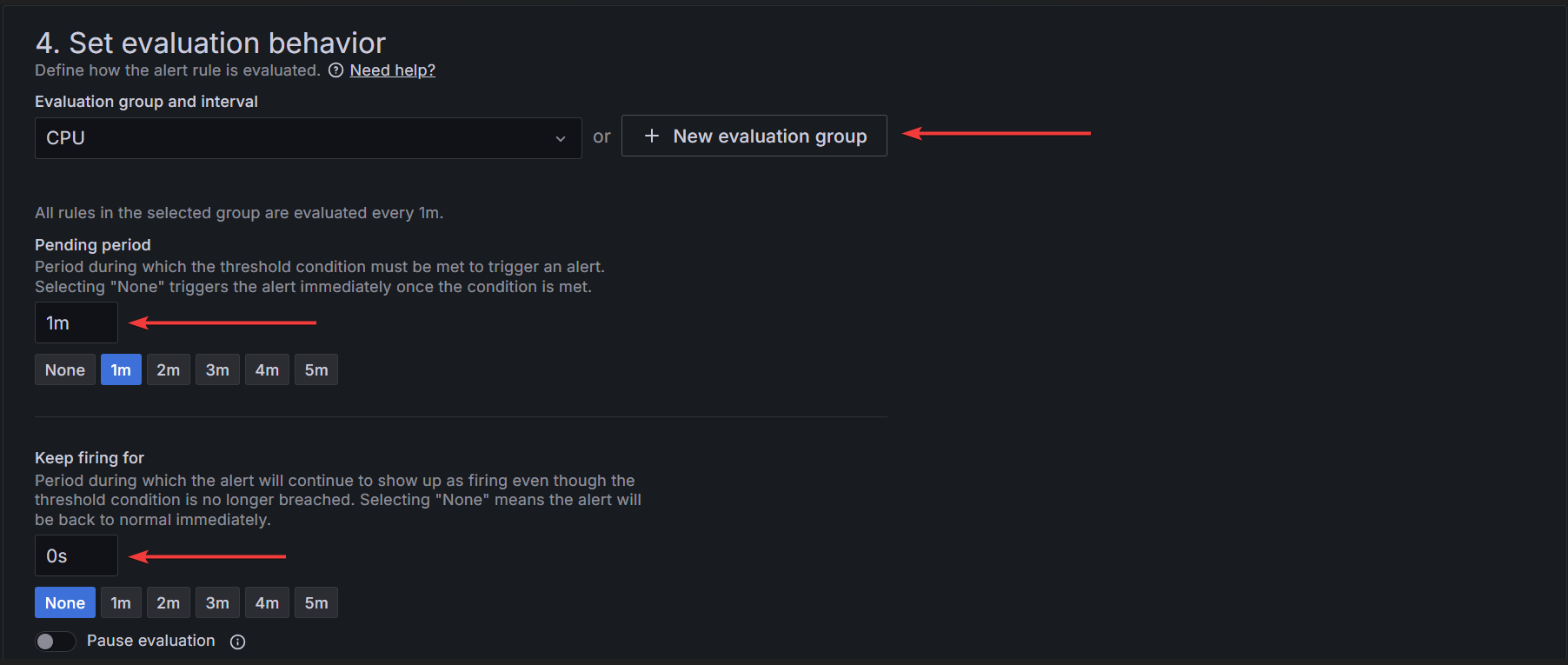
→ Set **Threshold value** for your condition → click on Preview

**Note** : You can provide **Custom Recovery Threshold value** for the condition

→ click on **New folder** → provide name for your folder

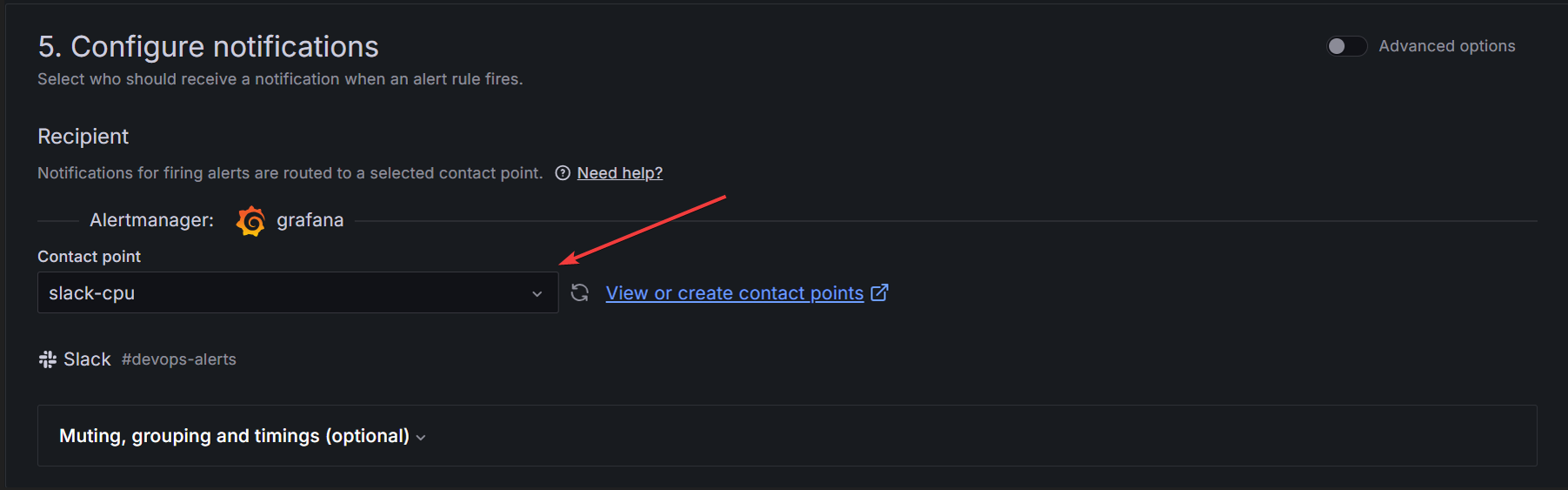
→ click on Add labels → provide your customized labels

**Note** : Folders is used to store the alert rules settings and the way you defined the PromQL query that you created.

→ click on **New evaluation group** → provide name for group

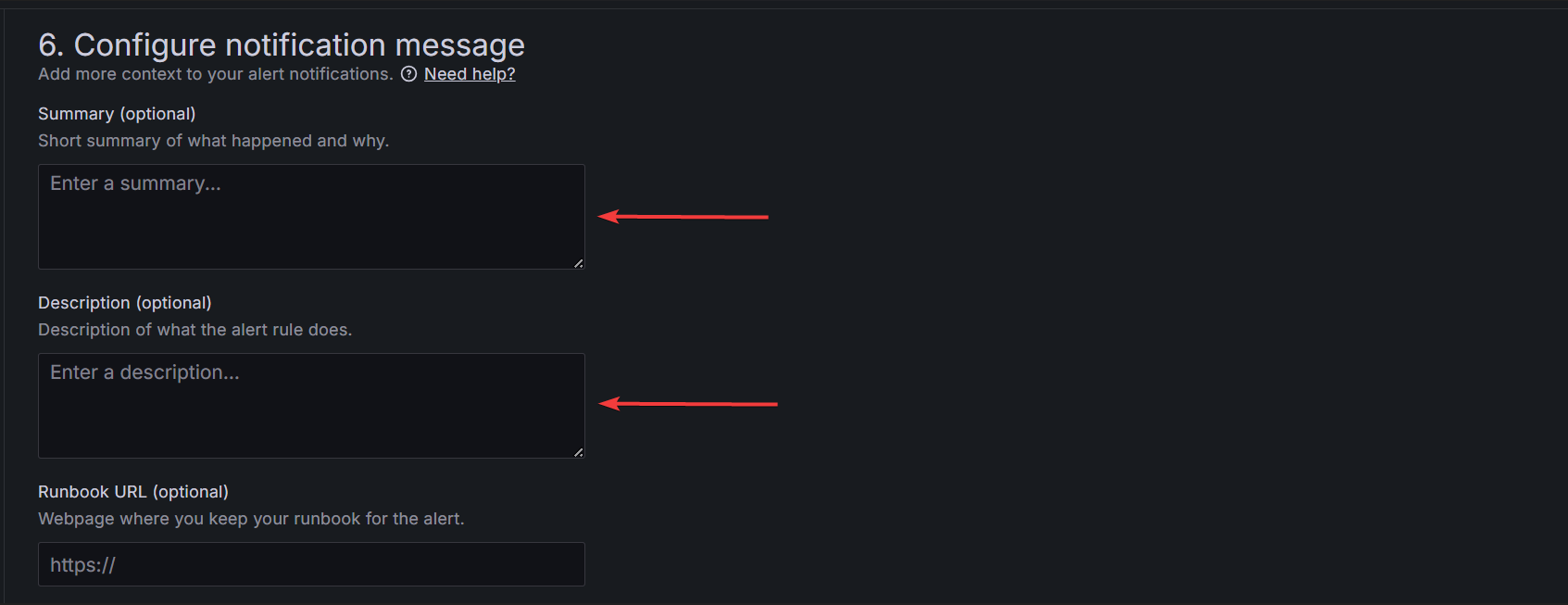
→ select → **Pending period** duration and **Keep firing for** durations

**Note** : All the evaluation period time, Pending period and Keep firing timings will be stored in the created group.



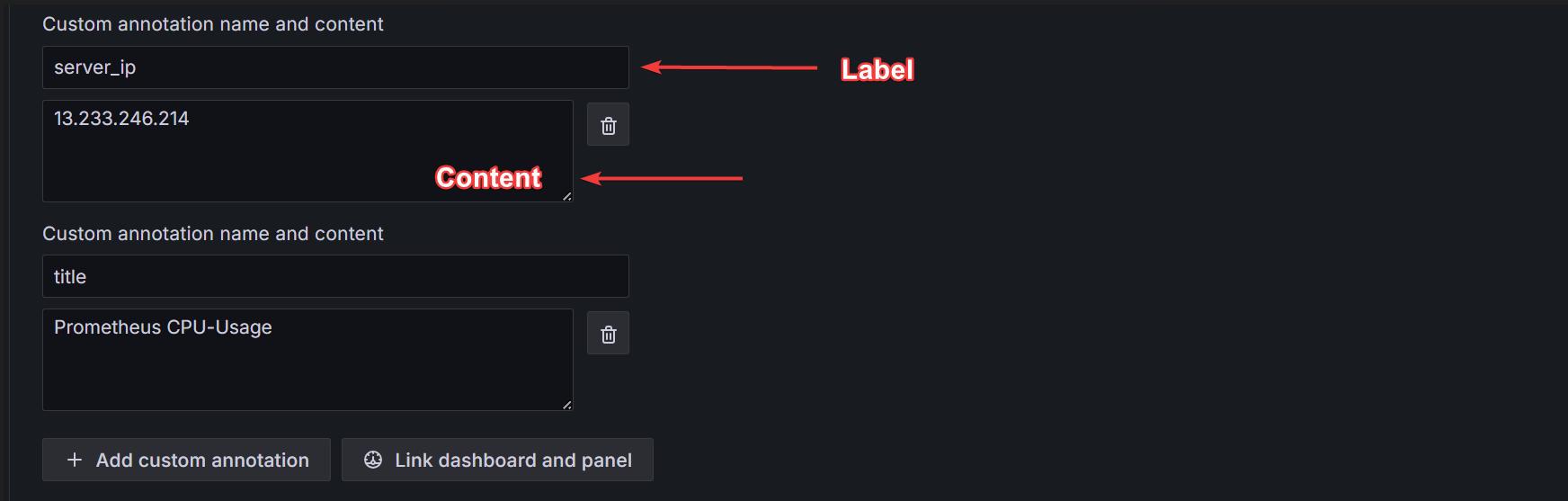
→ click on **Contact point** → select your **created contact point** that you wish to send alert

**Note** : Before using contact points, 1st you need to **create contact points**. After creating contact points, you will get the **dropdown for contact point** for your created contact points. (**Refer Module - 7 → to create contact points**)



→ Add **Summary** (for your alert message)

→ Add **Description** (for your alert message based on your condition)



→ Provide custom annotation → 1st block – provide label name

2nd block – content for that label to pass to the template

**Note** : Custom annotation name and content will be used, only if you are using custom templates for alerting message. (**Refer Module - 8 → for custom alerting templates**)

**PromQl Queries reference:** For CPU-usage →

(

100 - (

avg by(instance) (

irate(node\_cpu\_seconds\_total{job="Grafana-Server", mode="idle"}[1m])

) \* 100

)

)

AND on(instance) (up{job="Grafana-Server"} == 1)

For Memory-usage →

100 - (

avg by(instance) (

node\_memory\_MemAvailable\_bytes{job="Grafana-Server"}

/

node\_memory\_MemTotal\_bytes{job="Grafana-Server"}

) \* 100

)

and

on(instance) up{job="Grafana-Server"} == 1

For Disk-usage →

(

100 \* (1 - (

sum by(instance) (node\_filesystem\_free\_bytes{device="/dev/root", job="Grafana-Server"})

/

sum by(instance) (node\_filesystem\_size\_bytes{device="/dev/root", job="Grafana-Server"})

))

)

and

on(instance) up{job="Grafana-Server"} == 1

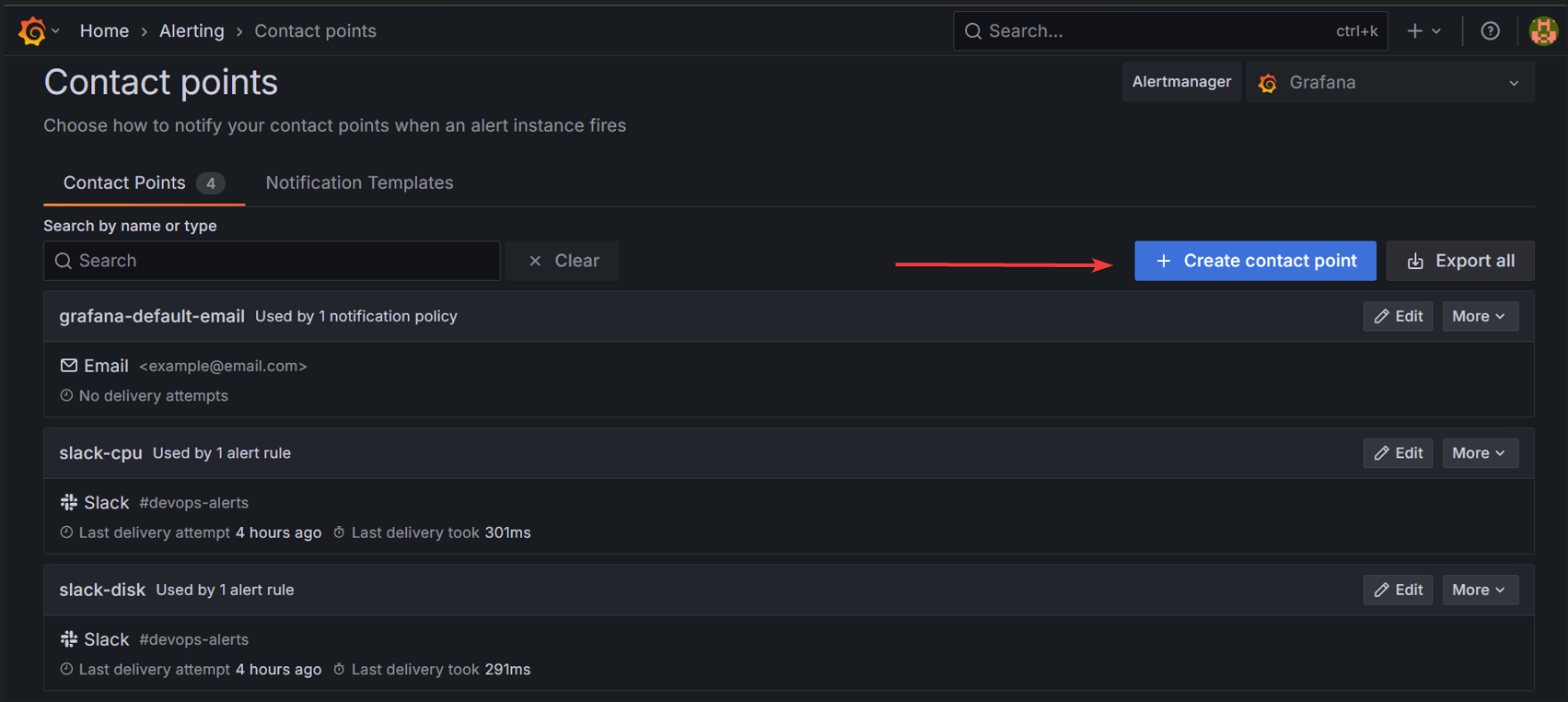
🔹 **Outcome:** Grafana could trigger alerts automatically when defined thresholds were exceeded for each job/instance.

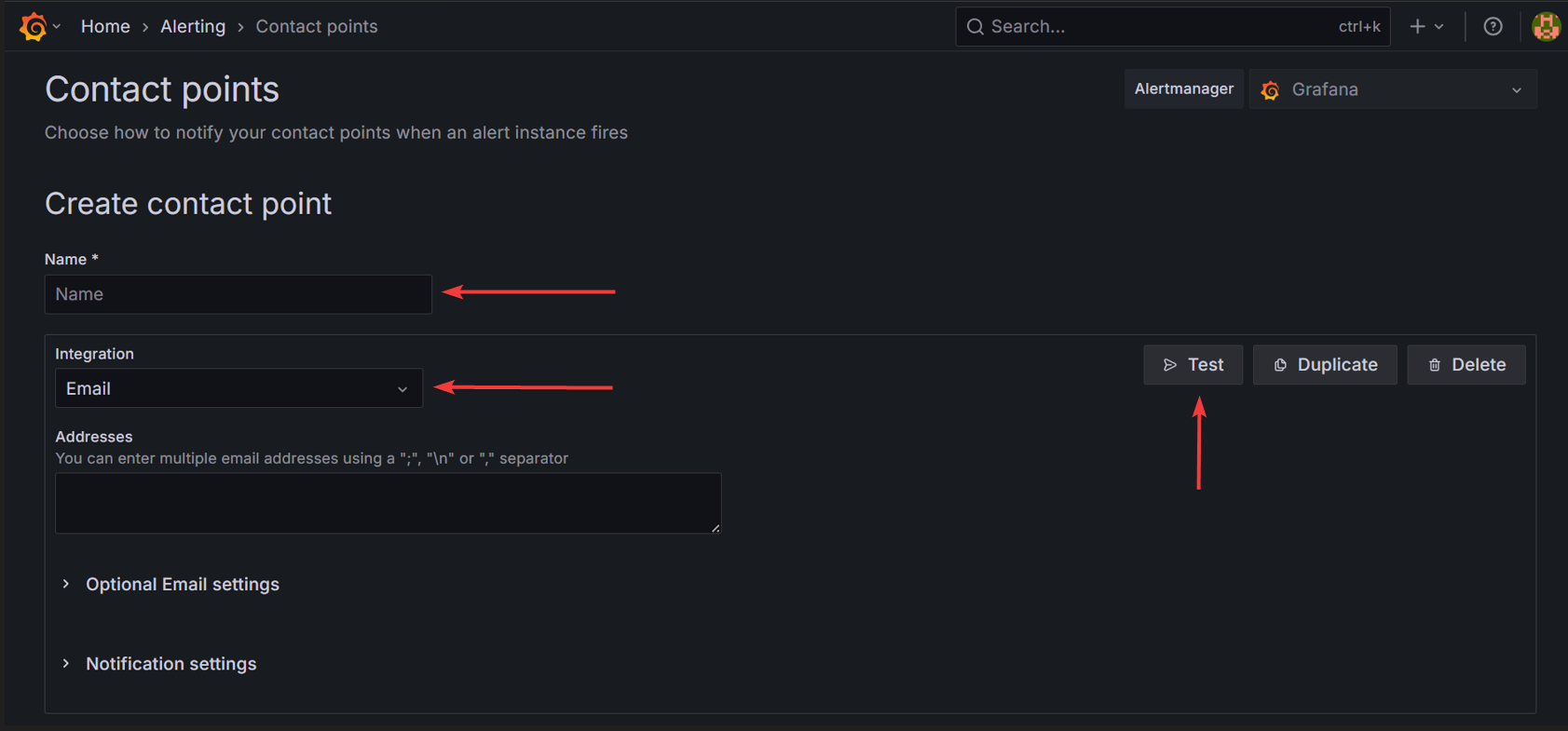
# Module 8: Contact points, Slack and Email Integration for Alerting

🔹 **Objective:** To notify users through **Slack** and **email** when alerts are fired or resolved.

🔹 **What I Did:**

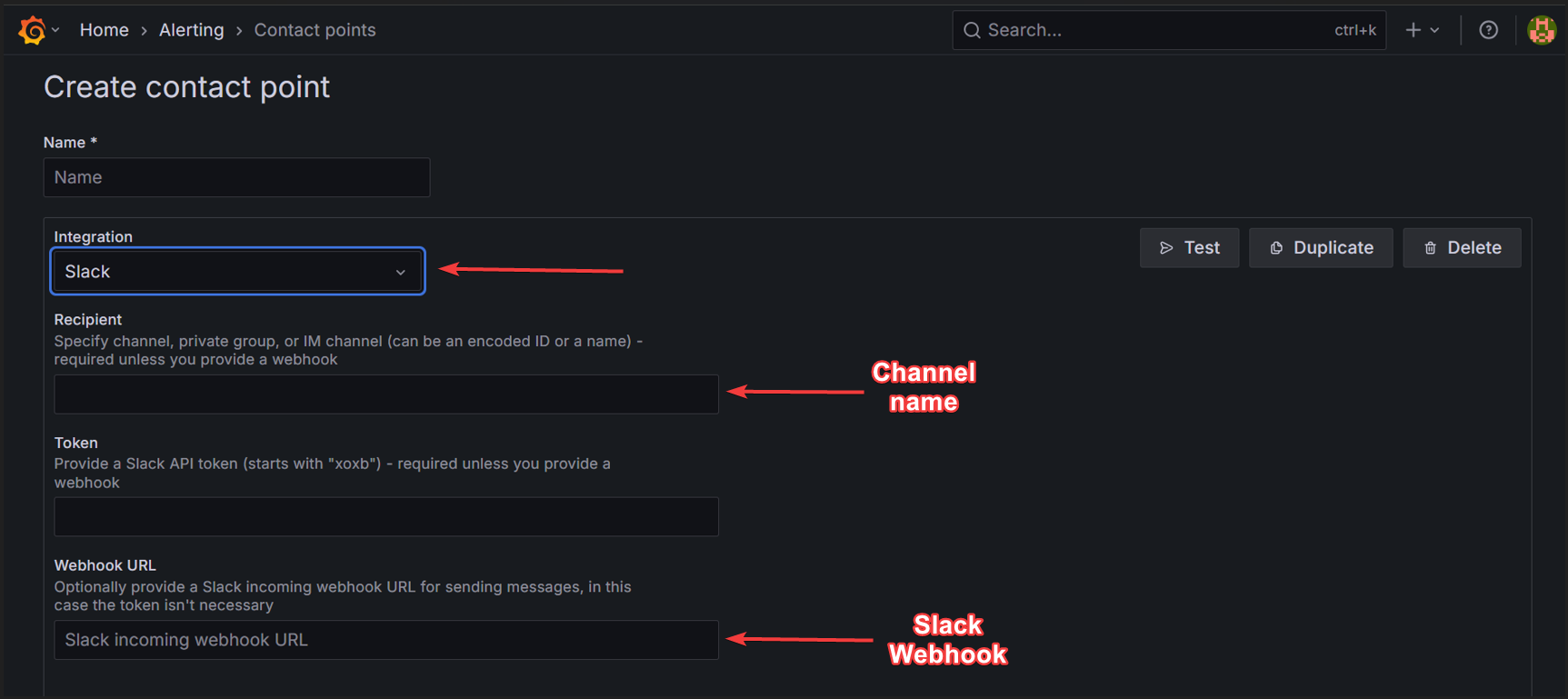
* Set up **Slack webhook** URL under Contact Points.
* Configured **SMTP** settings for email alerts.
* Created Contact Points with custom alert

→ Click on **Create contact point**

****

**→** Provide **Name** → select **Integration** (like Email, Slack etc..) → Test it

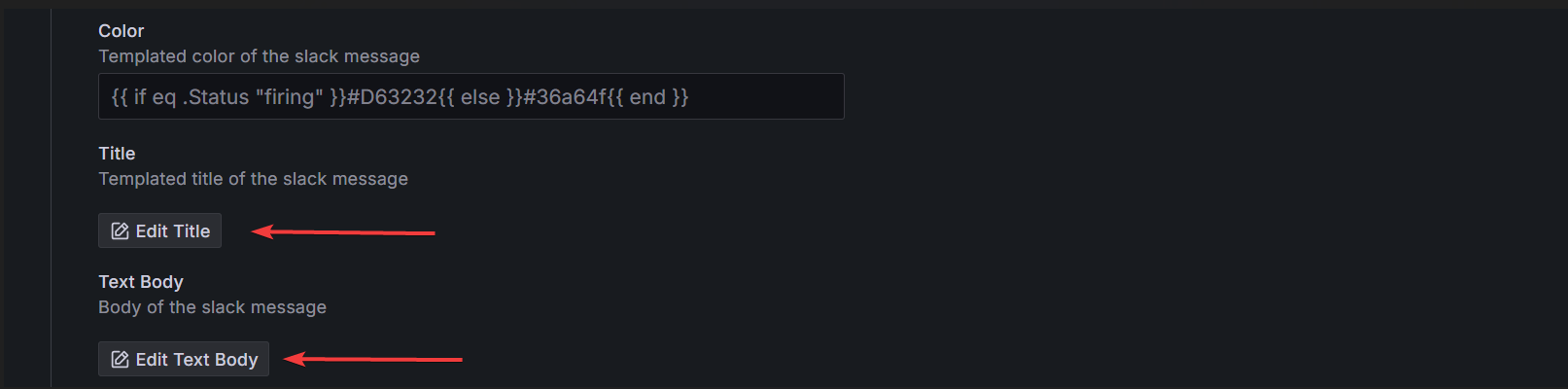
We integrated Slack, so **steps to do Slack integration** to receive alert notifications.



→ **Integration** → select **Slack** → provide **Recipient** (slack channel name)

→ Webhook URL → provide **Slack URL**

Note: You need to create **Slack Webhook URL** in **Slack API**.

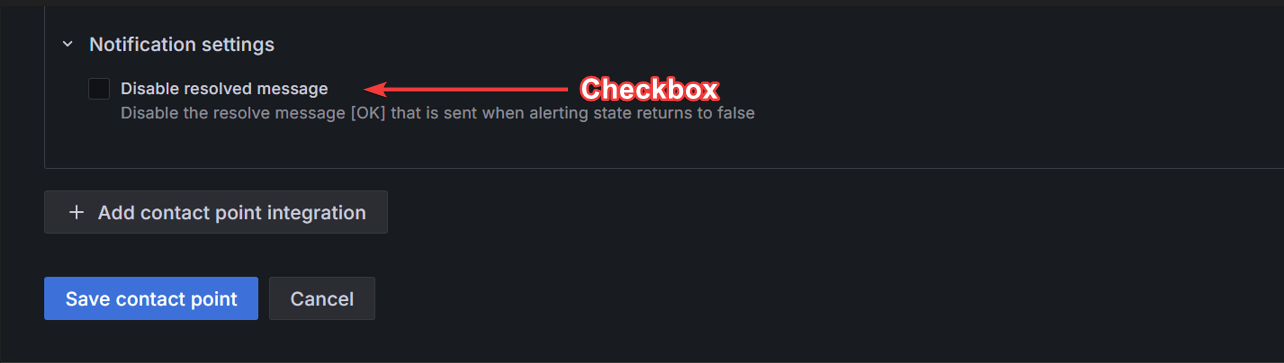


If you are using **Custom template** for Alert message, then you need to add customized **Title template** and **Alert Text Body**.

→ Title → Custom title template   
{{ if eq .Status "firing" }}🚨 {{ .CommonAnnotations.title }} Alert{{ else }}✅ {{ .CommonAnnotations.title }} Resolved{{ end }}

Note : If you are using custom title annotations means, you need to declare that labels and content in the Alert rules → Custom Annotations name and content.

Ex: Here CommonAnnotations.title is part of template variable and in that (**title** is the label and content that is declared for that label, passes the content to the slack alerting message). This is how it works.



You can Disable Resolve message if you don’t want resolve message.

→ Uncheck the Box → Disable resolved message

→ Click on **Save contact point** to create the integrated Slack contact point

* Created **Notification Policies** to route alerts accordingly.
* Tested alerts by lowering thresholds to intentionally trigger them.

🔹 **Outcome:** Alerts were successfully sent to Slack channels and email inboxes with proper formatting.

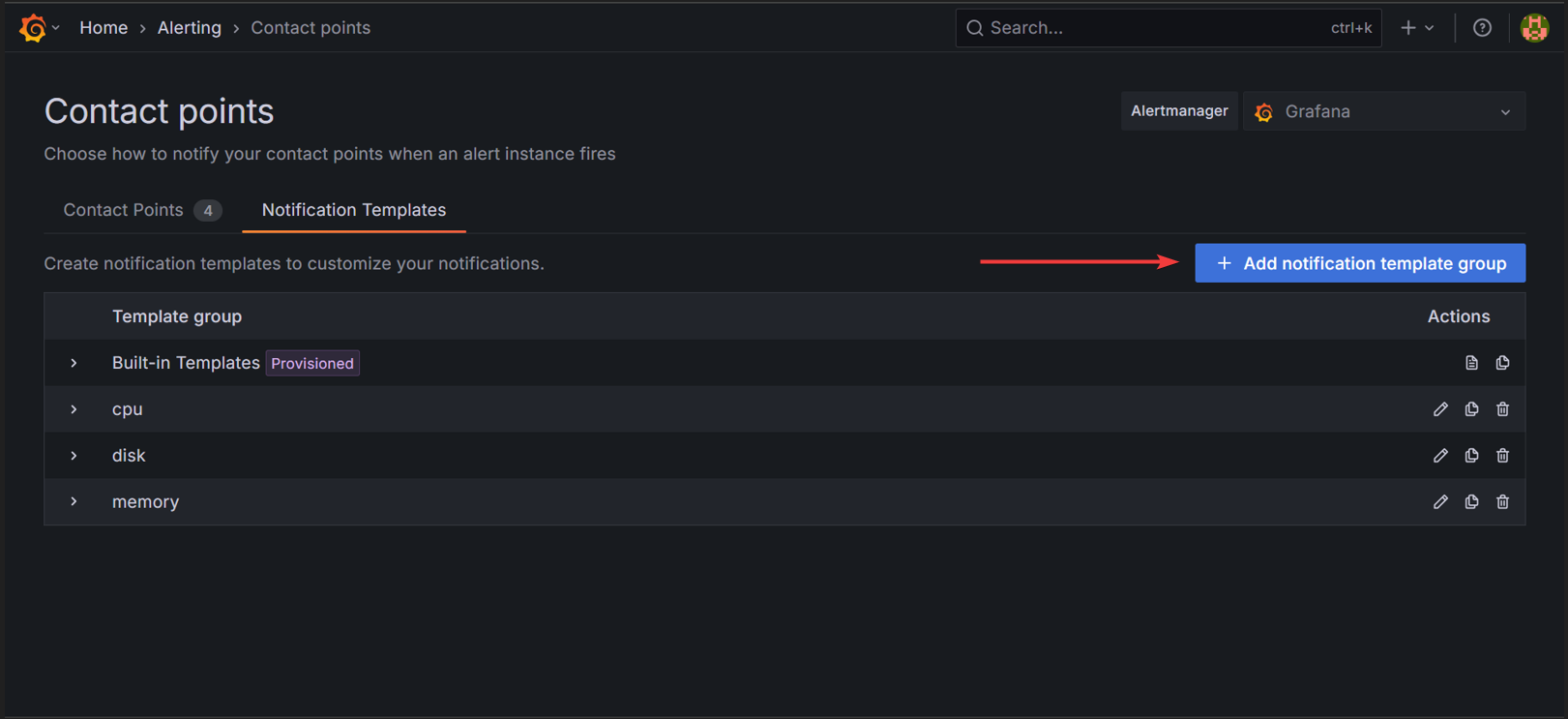
# Module 9: Custom Alert & Resolve Templates

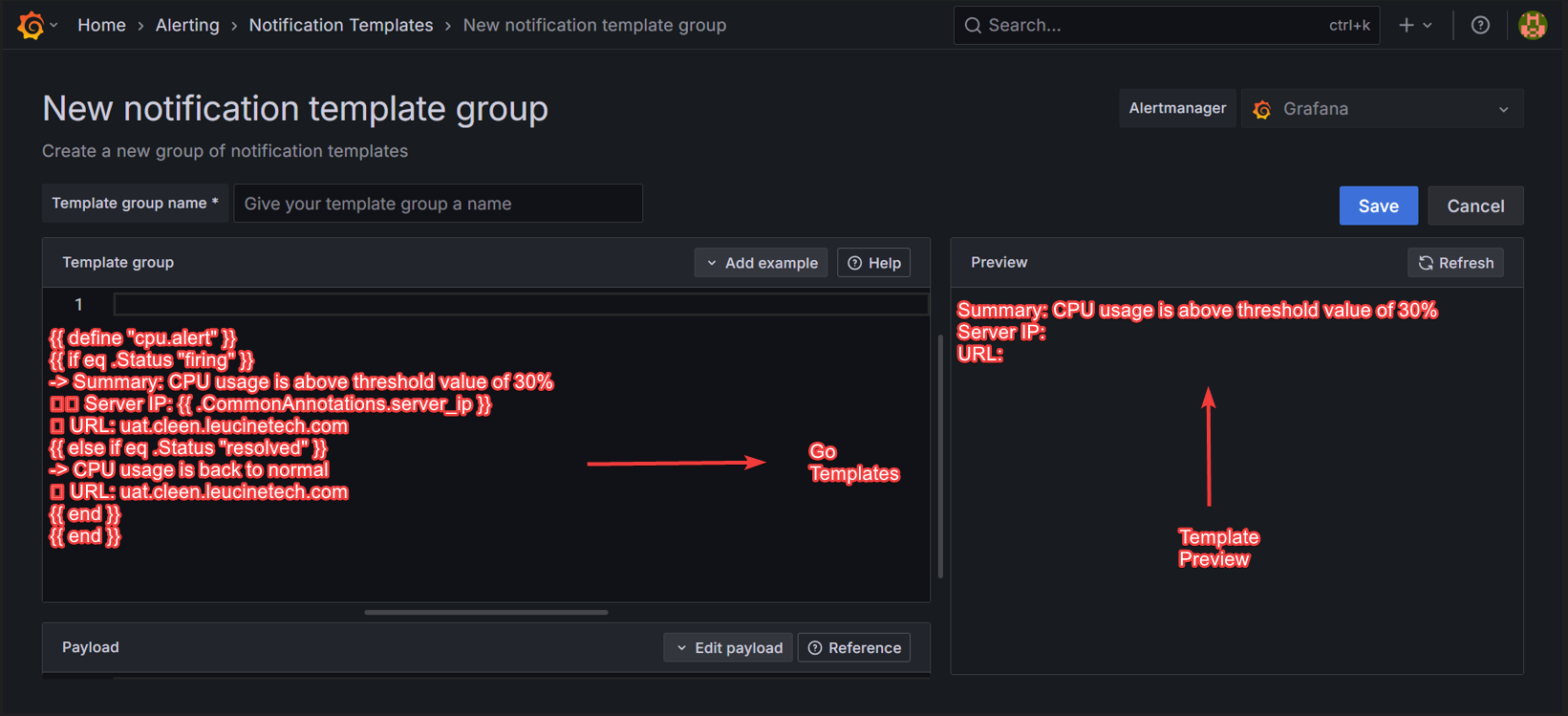
🔹 **Objective:** To format alerts consistently and clearly using Grafana’s templating system.

🔹 **What I Did:**

* Used Go templating to define **firing** and **resolved** states.
* Inserted dynamic fields like server IP (.CommonAnnotations.server\_ip) and alert name.
* Added emojis for visual distinction: 🚨 for alerts, ✅ for resolve.

Steps to create **Custom Go templating** for custom alert and resolve message:



→ click on **Add notification template group   
  
→** Provide **Template group name** → in editor Provide **custom Go templating alert message** body → Preview and check the template output → **Save** it  
Some custom Alert Body message Go templates :   
For **CPU alert** and **resolve** message template :

{{ define "cpu.alert" }}

{{ if eq .Status "firing" }}

-> Summary: CPU usage is above threshold value of 30%

🖥️ Server IP: {{ .CommonAnnotations.server\_ip }}

🌐 URL: uat.cleen.leucinetech.com

{{ else if eq .Status "resolved" }}

-> CPU usage is back to normal

🌐 URL: uat.cleen.leucinetech.com

{{ end }}

{{ end }}

For **Memory alert** and **resolve** message template :

{{ define "memory.alert" }}

{{ if eq .Status "firing" }}

-> Summary: Memory usage is above threshold value of 30%

🖥️ Server IP: {{ .CommonAnnotations.server\_ip }}

🌐 URL: uat.cleen.leucinetech.com

{{ else if eq .Status "resolved" }}

-> Memory usage is back to normal

🌐 URL: uat.cleen.leucinetech.com

{{ end }}

{{ end }}

For **Disk alert** and **resolve** message template :

{{ define "disk.alert" }}

{{ if eq .Status "firing" }}

-> Summary: Disk usage is above threshold value of 30%

🖥️ Server IP: {{ .CommonAnnotations.server\_ip }}

🌐 URL: uat.cleen.leucinetech.com

{{ else if eq .Status "resolved" }}

-> Disk usage is back to normal

🌐 URL: uat.cleen.leucinetech.com

{{ end }}

{{ end }}

🔹 **Outcome:** Alert messages were readable, informative, and aligned with expected format in both Slack and email.

# Module 10: Persisting Data with Docker Volumes

🔹 **Objective:** To ensure configuration and alert rules persist even if containers are restarted or destroyed.

🔹 **What I Did:**

* Used **host-mounted volumes** for:
  + /var/lib/grafana
  + /prometheus
  + /alertmanager

Volume Mounts : For **Grafana** :

volumes:

- /home/ubuntu/data-backup/grafana:/var/lib/grafana

For **Prometheus** :   
 volumes:

- ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml

- ./prometheus/alert.rules.yml:/etc/prometheus/alert.rules.yml

- /home/ubuntu/data-backup/prometheus:/prometheus

* Validated persistence by restarting containers and confirming dashboards remained intact.

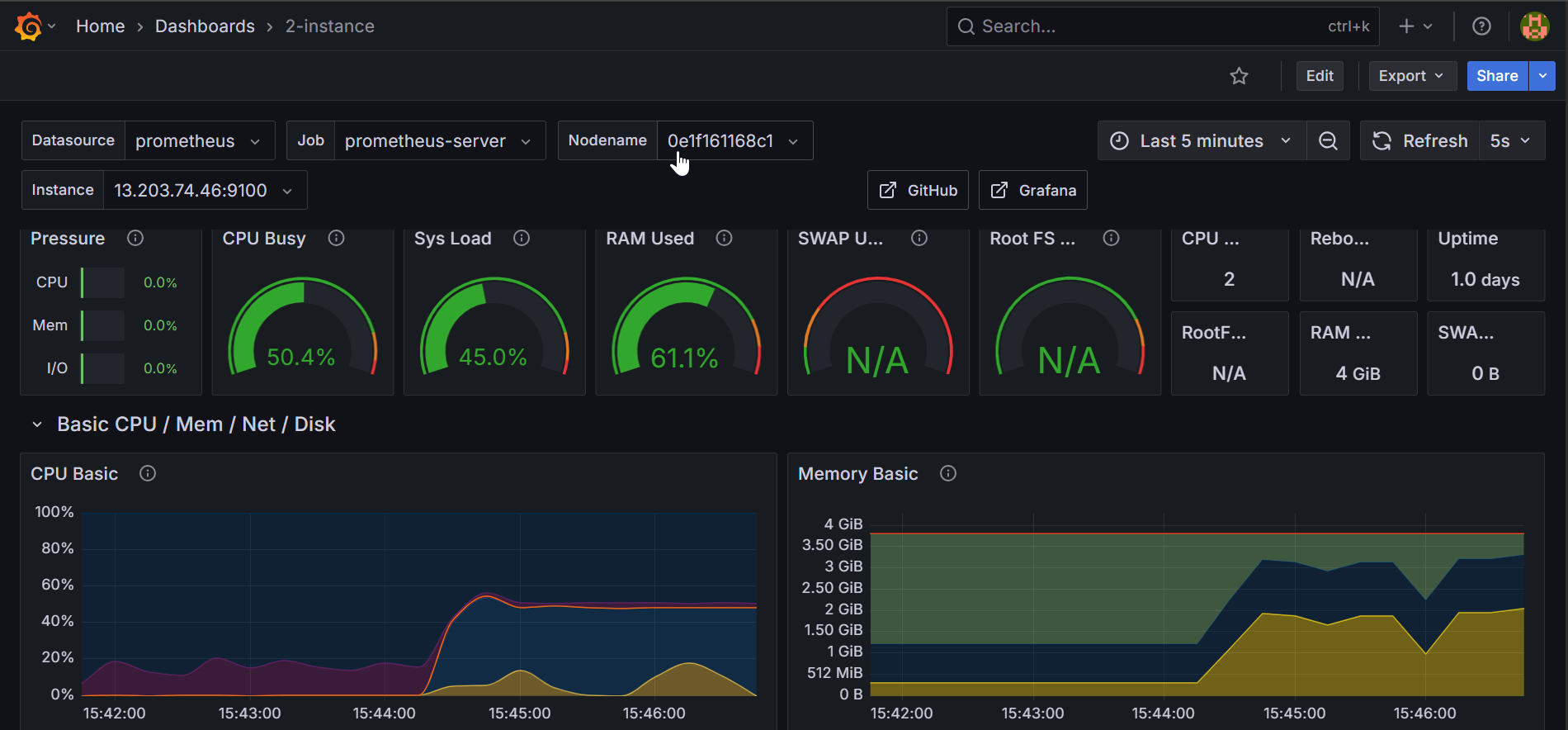
🔹 **Outcome:** Persistent storage was achieved; alert rules and dashboards were retained across restarts and transferable to other systems.

# Module 11: Output and Verification

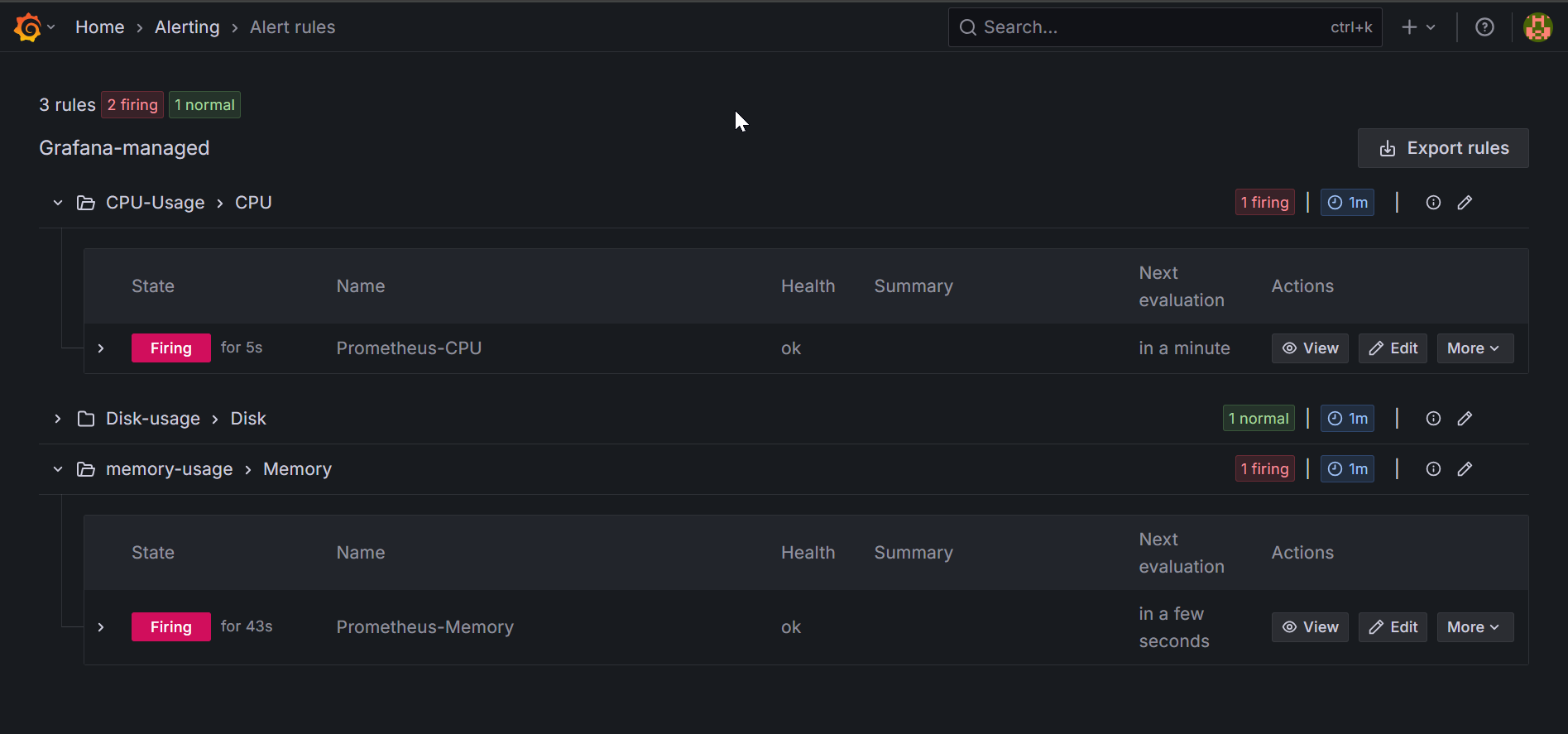
🔹 **Objective:** To validate the monitoring system’s behavior and confirm end-to-end functionality.

🔹 **What I Did:**

* Observed live metric collection in dashboards.

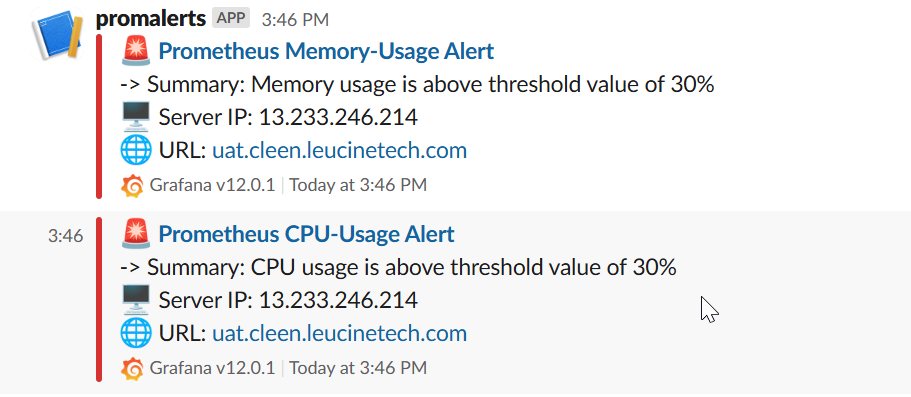


* Verified alert rule triggering on threshold breaches.

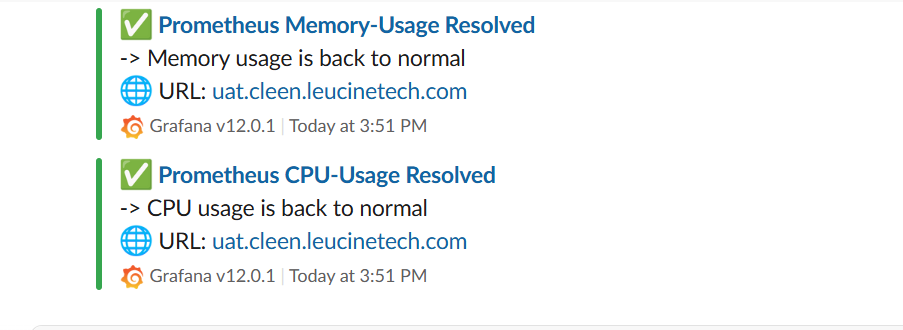


* Confirmed alert delivery through Slack/email.

For **Alert Message in Slack** :



For Resolve Message in Slack :



* Restarted Docker Compose and confirmed volume restore.

🔹 **Outcome:** Monitoring, alerting, notification, and data persistence were all verified to work across multiple Prometheus jobs and Grafana rules.

# 

# CONCLUSION

This report demonstrates a **streamlined, scalable monitoring setup** using **Grafana**, **Prometheus**, and **Node Exporter** across multiple instances. By **centralizing monitoring on a parent server** and deploying **Node Exporter on child nodes**, the system enables **efficient metric collection and alerting**. **Native Grafana alerts**, integrated with **Slack and email**, provide **real-time notifications**. **Persistent storage** and **external backups** ensure **data durability** and **easy recovery**. The architecture offers a **reliable and maintainable solution** for **multi-instance observability**.

# REFERENCES

Grafana Labs. (2025). *Grafana Documentation*.

Retrieved from [Technical documentation | Grafana Labs](https://grafana.com/docs/)