**🔬Lab Guide: Basic Monitoring and Alerting with Prometheus & Grafana**

**🎯 Learning Objectives**

By the end of this lab, trainees will be able to:

1. Deploy **Prometheus** to collect metrics from a service.
2. Deploy **Grafana** to visualize metrics.
3. Set up **basic alerts** for critical metrics.
4. Interpret monitoring data and simulate alert scenarios.

**🛠️ Prerequisites**

* Trainees should have:
  + **Linux/MacOS** system or VM
  + Docker installed (Docker Desktop is fine)
  + Basic familiarity with Linux CLI
* Optional: Sample service to monitor (e.g., a simple web server or microservice)

**Deploy the sample application : app.py**

from flask import Flask, Response

from prometheus\_client import Counter, Histogram, generate\_latest, CONTENT\_TYPE\_LATEST, CollectorRegistry

import time

app = Flask(\_\_name\_\_)

# Create a custom registry to export only our metrics

registry = CollectorRegistry()

# Metrics

REQUEST\_COUNT = Counter(

"request\_count\_total",

"Total number of HTTP requests",

["endpoint", "status"], # 👈 added "status" label

registry=registry

)

REQUEST\_LATENCY = Histogram(

"request\_latency\_seconds",

"Request latency in seconds",

["endpoint"],

registry=registry

)

# Home endpoint

@app.route("/")

def home():

with REQUEST\_LATENCY.labels(endpoint="/").time():

time.sleep(0.1) # Simulate proc. delay (~100ms) to make it realistic

REQUEST\_COUNT.labels(endpoint="/", status="200").inc() # 👈 updated

return "Hello from Raman App!"

# Dynamic endpoint /<name>

@app.route("/<name>")

def greet(name):

endpoint\_label = f"/{name}"

with REQUEST\_LATENCY.labels(endpoint=endpoint\_label).time():

time.sleep(0.1) # Simulate processing delay (~100ms)

REQUEST\_COUNT.labels(endpoint=endpoint\_label, status="200").inc() # 👈 updated

return f"Hello, {name}!"

# Failing endpoint to simulate errors

@app.route("/fail")

def fail():

REQUEST\_COUNT.labels(endpoint="/fail", status="500").inc() # 👈 new for failures

return "Internal Error", 500

# Metrics endpoint

@app.route("/metrics")

def metrics():

return Response(generate\_latest(registry), mimetype=CONTENT\_TYPE\_LATEST)

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", port=5000)

**requirements.txt**

Flask==2.3.3

prometheus-client==0.21.0

sudo apt update

sudo apt install python3.12-venv

python3 -m venv venv

**# 2. Activate it**

source venv/bin/activate

**# 3. Install dependencies inside the venv**

pip install -r requirements.txt

**Step 1: Launch Prometheus using Docker**

1. Create a **raman-prom.yml** file with basic configuration:

global:

scrape\_interval: 10s

scrape\_configs:

- job\_name: 'raman-service'

static\_configs:

- targets: ['host.docker.internal:5000']

**Create a shared network:**

Sudo docker network create raman-observability-net

**Run Prometheus**:

sudo docker run -d \

--name raman-prometheus \

--network raman-observability-net \

--add-host=host.docker.internal:host-gateway \

-v "$(pwd)/raman-prom.yml":/etc/prometheus/prometheus.yml \

-p 9090:9090 \

prom/prometheus

**Run Grafana**:

sudo docker run -d \

--name raman-grafana \

--network raman-observability-net \

-p 3000:3000 \

grafana/grafana

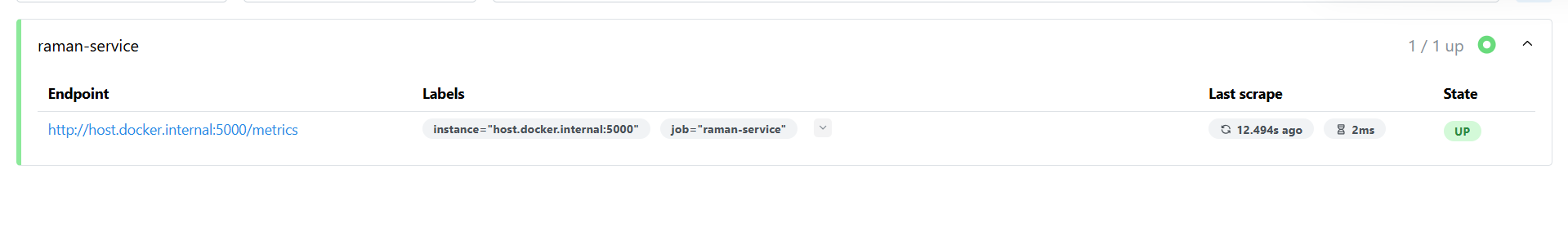
**Step 5 – Check Prometheus Target**

Go to:

http://<your-server-ip>:9090/targets

You should see:

raman-service – UP

****

**Configure Grafana**

1. Open Grafana:

http://<your-server-ip>:3000

(Default login: admin/admin)

1. Go to:

⚙️ → Data Sources → Add Data Source → Prometheus

Set URL:

http://raman-prometheus:9090

Click **Save & Test** → Should say “Data source is working”.

**Step 7 – Create Grafana Dashboard**

1. Go to:

Dashboards → New → New Dashboard

**Panel 1 – Total Requests**:

* Query:

request\_count\_total

* Title: **Total Requests**
* Visualization: Time series

**Panel 2 – Average Latency**:

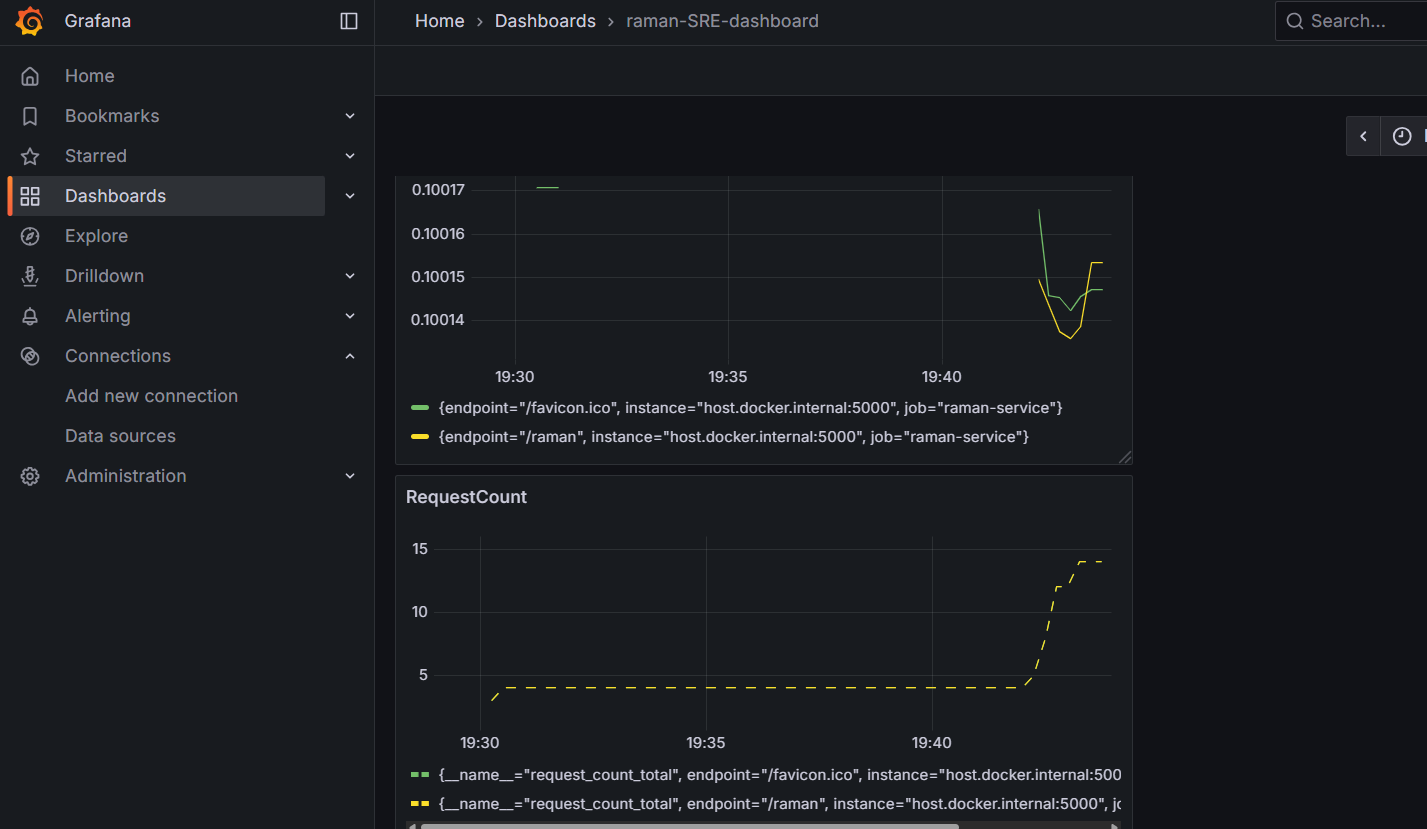
* Query:

rate(request\_latency\_seconds\_sum[1m]) / rate(request\_latency\_seconds\_count[1m])

* Title: **Average Latency**
* Visualization: Time series

Save as:

raman-SRE-dashboard



**🧪 Lab Guide: Incident Simulation + Mock Postmortem Report**

**🎯 Scenario Setup**

**System Under Test:**

* A simple **web service** (orders-api) deployed in Docker.
* It exposes metrics via Prometheus.
* Grafana dashboard shows:
  + Request count
  + Latency
  + Error rate

**Normal Baseline:**

* Latency: ~100ms
* Error Rate: 0%
* Traffic: ~50 requests/min

**🔥 Incident Simulation**

**Step 1 – Create the Incident**

* At **19:40**, simulate a **memory leak / crash** in the orders-api container:
* docker kill orders-api
* The service goes **down hard** (no responses).

**Step 2 – Monitoring Detects Problem**

* Prometheus alert fires:
  + **HighErrorRate:** http\_requests\_total{status=500} > 20% for 2m
  + **ServiceDown:** up{job="orders-api"} == 0

**Step 3 – On-Call Team Responds**

Roles:

* **Incident Commander (IC)** – coordinates response.
* **Ops Engineer** – checks logs, metrics.
* **Comms Lead** – updates Slack/Status page.

Timeline:

* 19:41 → PagerDuty alerts IC.
* 19:42 → Team joins Zoom bridge.
* 19:43 → Ops Engineer confirms orders-api container not running.
* 19:45 → IC declares **SEV-2 Incident** (partial outage).
* 19:46 → Ops restarts container:
* docker start orders-api
* 19:47 → Service back online, latency stabilizes.
* 19:48 → Error rate drops to baseline.
* 19:50 → IC declares incident resolved.

**Total duration:** ~10 minutes

**📄 Mock Postmortem Report**

**1. Summary**

On **2025-08-16 from 19:40–19:50 IST**, the orders-api service was unavailable due to an unexpected container crash. Approximately 40% of user requests failed with HTTP 500 errors during this period. The issue was mitigated by restarting the container.

**2. Impact**

* **Users Affected:** Customers using the checkout flow.
* **Requests Impacted:** ~400 requests failed out of ~1,000 total in that window.
* **Business Impact:**
  + Failed checkouts → potential loss of orders/revenue.
  + Customer support tickets increased during the outage.
* **Severity:** SEV-2 (customer-facing outage, mitigated quickly).

**3. Root Cause**

The root cause was a **container crash caused by unhandled memory exhaustion** in the orders-api service.

* Application logs showed:
* OOMKilled: process out of memory
* No automatic restart policy was configured for this container in Docker Compose.

**4. Detection**

* **Primary Detection:** Prometheus alert (orders\_api\_down) fired at 19:41.
* **Secondary Signals:** Grafana dashboards showed request count drop to zero and error rate spiking to 100%.
* **Time to Detect (TTD):** ~1 min after failure.

**5. Response Timeline**

| **Time** | **Event** |
| --- | --- |
| 19:40 | Container crash occurred |
| 19:41 | Prometheus alert fired (ServiceDown) |
| 19:42 | PagerDuty paged on-call engineer |
| 19:43 | Bridge created, team assembled |
| 19:45 | IC declared SEV-2 incident |
| 19:46 | Ops restarted container |
| 19:47 | Service restored |
| 19:50 | IC declared incident resolved |

**6. Resolution**

* Ops restarted the container manually.
* Monitoring confirmed recovery (latency back to 100ms, error rate 0%).

**7. Lessons Learned**

**What Went Well** ✅

* Monitoring detected the issue quickly (alert fired within 1 min).
* Incident roles were assigned promptly (IC, Ops, Comms).
* Service recovered quickly once container restarted.

**What Didn’t Go Well** ❌

* Root cause (memory leak) not detected before crash.
* Container lacked auto-restart policy.
* No runbook existed for “orders-api container down” scenario.

**8. Action Items**

**Prevent Recurrence:**

* Add restart: always policy in Docker Compose for orders-api (Owner: Ops, Due: 1 week).
* Add memory limits & monitoring alerts for container memory usage (Owner: SRE, Due: 2 weeks).
* Investigate memory leak in /checkout code path (Owner: Dev, Due: 2 weeks).

**Improve Detection:**

* Create synthetic probe (health-check endpoint) for checkout flow (Owner: QA, Due: 1 week).
* Configure burn-rate alert for error budget consumption (Owner: SRE, Due: 2 weeks).

**Improve Process:**

* Write runbook for container crash scenario (Owner: IC, Due: 1 week).
* Run chaos experiment: simulate container OOM kills monthly (Owner: SRE, Due: ongoing).

**9. Severity and Error Budget Impact**

* Incident lasted **10 minutes**.
* Availability SLO: **99.9% per month**.
* Error budget consumed:
  + Monthly allowed downtime: ~43 minutes.
  + Outage consumed ~10 minutes (~23% of budget).

**10. Follow-up**

A follow-up meeting will be held on **2025-08-20** to review progress on action items.

**🔬 Lab Guide: Implement SLI Tracking for a Sample Web Service**

**🎯 Learning Objectives**

By the end of this lab, participants will be able to:

* Implement **Service Level Indicators (SLIs)** in Prometheus using PromQL.
* Track **availability** and **latency percentiles** for a web service.
* Compare SLIs against **SLO targets** with thresholds in Grafana.
* Simulate failures to observe **error budget consumption** in action.

**📘 Background for Trainees**

In previous labs:

* You defined **SLIs, SLOs, and SLAs** .
* You collected **raw metrics** with Prometheus and Grafana .

Now, we’re going further:

* **SLIs = metrics interpreted as reliability signals.**
* Example:
  + *Raw metric*: number of HTTP requests.
  + *SLI*: percentage of successful requests (availability).

**🛠️ Pre-requisites**

* Prometheus and Grafana already running .
* Sample Flask app (app.py) running with metrics at /metrics.
* Grafana dashboard created with at least one panel.

**Step 1 – Define the SLOs**

For this lab, assume:

* **Availability SLO**: 99.9% successful requests per month.
* **Latency SLO**: 95% of requests must complete in < 300 ms.

**Step 2 – Identify Metrics Exposed**

Your Flask app provides:

* request\_count\_total – Counter of requests (with endpoint label).
* request\_latency\_seconds\_bucket – Histogram of request latencies.

**Step 3 – Write PromQL Queries for SLIs**

**A. Availability SLI**  
Formula:

Availability = SuccessfulRequests / TotalRequests

PromQL:

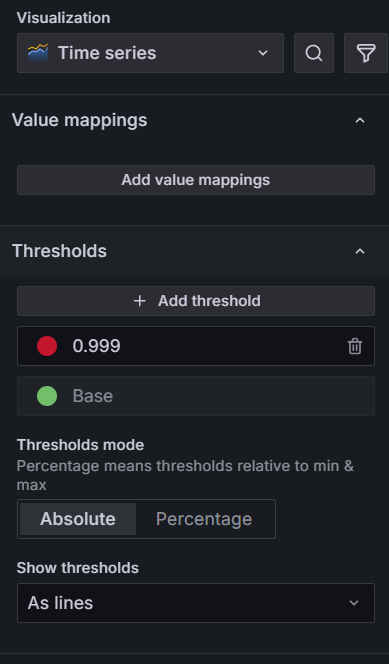
sum(rate(request\_count\_total{status="200"}[1m]))

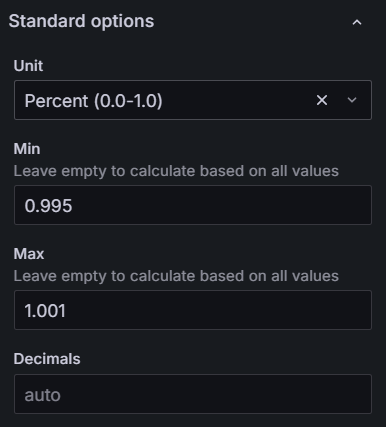
/

sum(rate(request\_count\_total[1m]))

**Create grafana panel : Availability SLI Panel**

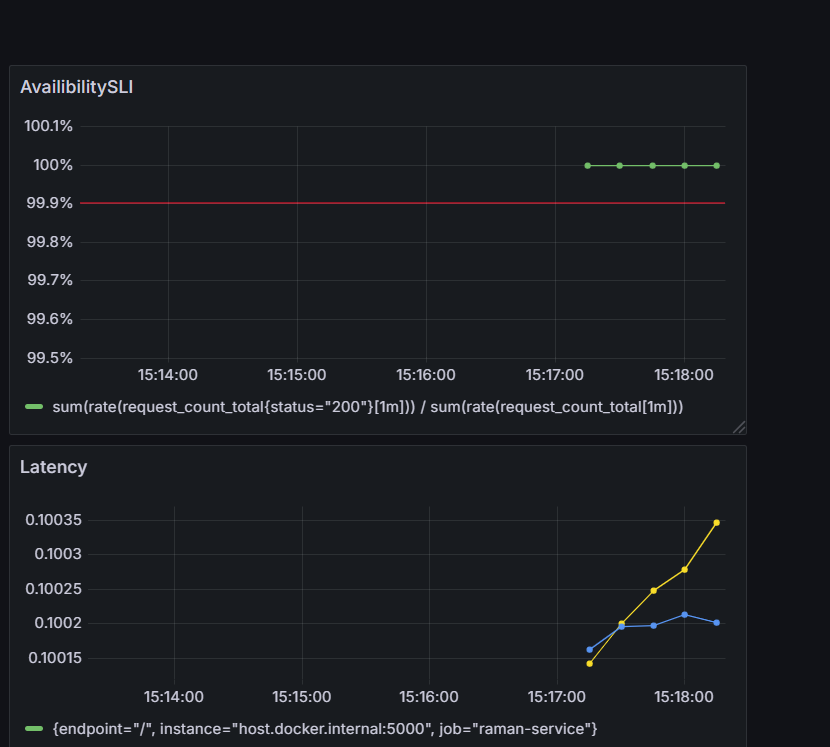
* + Query = Availability PromQL.
  + Visualization = Time series.
  + Add **threshold line** at 0.999 (absolute value).
  + Change the standard unit to percentage (0-10) and set the y axis to zoom in the graph.





YOU WILL SEE SOMETHING LIKE BELOW AS THE FINAL RESULT

Below means 100% of request hitting our application are successful.



**B. Latency SLI (95th percentile)**  
Formula:

p95 latency = 95% of requests are faster than this value

PromQL:

histogram\_quantile(

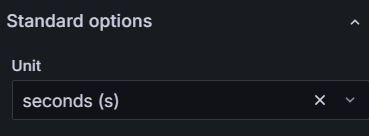
0.95,

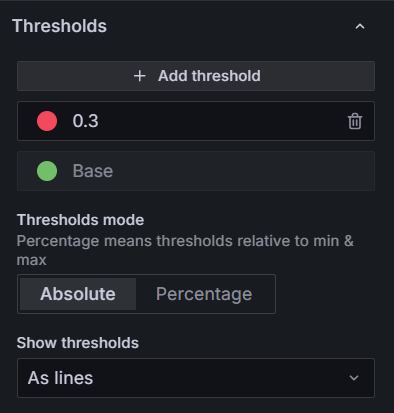
sum(rate(request\_latency\_seconds\_bucket[1m])) by (le)

)

**Create Grafana Panels**

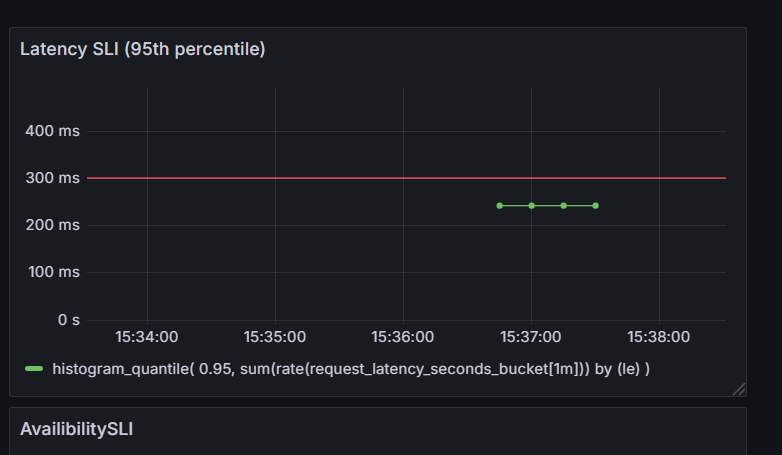
1. **95th Percentile Latency Panel**
   * Query = Latency PromQL.
   * Visualization = Time series.
   * Add **threshold** at 0.3 (300 ms = 0.3s).
   * Change the standard unit to seconds.





Result → You now see reliability signals, not just raw counts.

Below means 95% of requests completed faster than 243 ms.



**Step 5 – Simulate Failures (to See Breaches)**

If you don’t simulate, the dashboard will always look healthy. Try these:

**A. Latency Spike**  
In app.py, change delay in / endpoint:

time.sleep(1.0) # instead of 0.1

NOTE : MAKESURE TO UPDATE IN /raman approute

Restart the service → Latency SLI panel will cross 0.3s threshold.

**B. Error Burst**  
Add a failing endpoint:

@app.route("/fail")

def fail():

return "Internal Error", 500

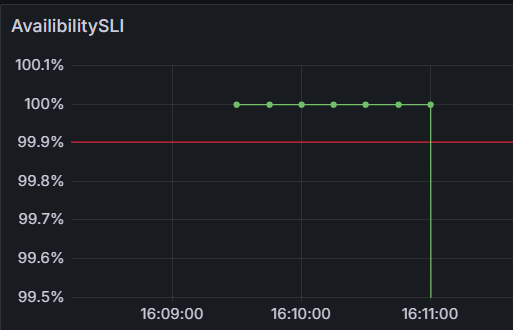
Lets give a continuous load to our app at /raman:

while true; do curl -s -o /dev/null -w "%{http\_code}\n" http://localhost:5000/raman; sleep 1; done

Generate load for failures to mimic failures or failed requests:

ab -n 30 -c 5 http://localhost:5000/fail

→ Availability SLI drops below 99.9%.



**C. Crash the Service**

Stop the application

→ Availability SLI plummets to 0 until restarted.

**📄 Deliverables**

By the end of this lab, participants will have:

* Grafana panel showing **availability % vs 99.9% SLO line**.
* Grafana panel showing **95th percentile latency vs 300 ms SLO line**.
* Observed SLO **breaches** via failure injection.

**💡 Reflective Questions**

* How do SLIs differ from raw metrics?
* Did you see the latency or availability cross thresholds during simulation?

👉 This lab makes the leap from *seeing numbers* to *enforcing reliability objectives* — the real work of an SRE