**2.** **Metrics Collection and Visualization using Prometheus and Grafana**

A close-up of a grid

AI-generated content may be incorrect.

**Figure:** Raw Prometheus Metric Export for HTTP Duration

This image displays the raw Prometheus exposition format for the http\_server\_duration metric family:

**Key Highlights:**

* The metric type is histogram, allowing detailed distribution tracking of request latencies.
* The http\_server\_duration\_count reflects the total number of completed requests for different routes and status codes.
* Multiple http\_server\_duration\_bucket entries capture how long each request took, classified into latency buckets like <=5ms, <=25ms, <=100ms, etc.
* The exposed metadata confirms the use of OpenTelemetry SDK for Node.js, including fields such as telemetry\_sdk\_name, process\_command, and net\_host\_port.

A screenshot of a computer

AI-generated content may be incorrect.

**Figure:** Grafana Metrics Exploration of http\_server\_duration\_count

This screenshot shows the PromQL query http\_server\_duration\_count executed in Grafana’s Explore tab. The metric reflects the total number of HTTP server requests handled, categorized by route path, HTTP method, status code, and scheme.

**Explanation:**

* The query returned multiple time series based on route-specific metadata such as /, /todo, and various status codes (200, 302, 304).
* The orange line indicates a continuous rise in request count for the /todo route with status 200, suggesting high usage.
* Other paths showed a flat line, indicating fewer or no requests during the observed window.
* This real-time visualization confirms successful metrics ingestion from OpenTelemetry SDK to Prometheus and dashboard rendering in Grafana.

**Why This Matters:**

* Confirms the instrumentation of HTTP metrics using OpenTelemetry and correct scraping by Prometheus.
* Enables latency and request pattern analysis per route and status.
* Complements tracing spans with numeric evidence of performance trends.

This result shows how OpenTelemetry can bridge both tracing and metric observability in microservice environments, helping operators visualize patterns, spot anomalies, and drive optimization across services.

**3. Alerting using Alertmanager with Email Notification.**

Configured Prometheus Alertmanager to monitor system and application-level metrics and trigger alerts when critical thresholds are breached. Alert routing and notification delivery were implemented via email to ensure real-time visibility of critical events.

To verify the alerting pipeline, we created a test rule named AlwaysFiringTestAlert and validated the following:

A screenshot of a computer

AI-generated content may be incorrect.

**Figure:** Alertmanager Web Interface Displaying Firing Alerts

The figure shows the Alertmanager UI, where multiple alerts are being managed:

The alert named AlwaysFiringTestAlert is actively firing and linked to the email-notifier receiver.

Other alerts like KubeControllerManagerDown, KubeSchedulerDown, and KubeProxyDown also show as active with critical severity.

The labels applied (severity="critical", prometheus="monitoring/...") help categorize and route alerts.

The interface groups alerts based on routing configuration, confirming successful alert classification and receiver mapping.

This validates that Prometheus Alertmanager is correctly processing incoming alert rules, grouping them by receiver, and preparing them for dispatch.

A screenshot of a computer

AI-generated content may be incorrect.

**Figure:** Email Notification Received for Firing Alert

This email alert received for AlwaysFiringTestAlert at the configured address (nurmohammadshawon@gmail.com):

* The subject line confirms that the alert is [FIRING:1], indicating one active instance.
* Alert details include labels like alertname, prometheus source, and severity.
* Annotation fields such as description and summary provide context about the alert’s purpose and status.
* A direct link is available to view the alert in the Alertmanager UI (View in Alertmanager).

This email confirms that the full alert lifecycle—firing, routing, and notification—has been executed successfully via the configured SMTP email service.

**Why This Matters:**

* Confirms end-to-end functionality of alerting and notification setup.
* Ensures engineers are notified immediately for test or production alerts.
* Provides clear traceability from metrics breach to actionable notification.

**4. Performance Comparison: Before vs After Optimization**

|  |  |  |
| --- | --- | --- |
| **Metric** | Before Optimization | After Optimization |
| Cart API latency | 480ms | 310ms |
| Checkout errors/day | 12 | 3 |
| Redis call time | 1.5ms | 841µs |
| Alert frequency | 40/day | 18/day |

**Why This Matters:**

Shows the *impact* of telemetry data in identifying bottlenecks and validating improvements.

**Reference for table:**

**Figure:** Alertmanager Web Interface Displaying Firing Alerts

Alertmanager web UI with multiple **firing alerts**

Alert Frequency (Before Optimization)

Thesis Result Section: *“Alert Routing and Email Notification”*

**Figure:** Email Notification Received for Firing Alert

Email notification received from Alertmanager (AlwaysFiringTestAlert)

Email Notification Delivery Validation

**Figure:** Raw Prometheus Metric Export for HTTP Duration

Raw Prometheus metric output:

* http\_server\_duration\_count
* http\_server\_duration\_bucket
* telemetry\_sdk\_name, http\_route, etc.

**Use this for:**  
Raw Prometheus Metric Scraping  
Use values to calculate **latency (avg = sum/count)**  
Thesis Result Section: *“Prometheus Metric Scraping and Histogram Analysis”*

**Figure:** Grafana Metrics Exploration of http\_server\_duration\_count

Grafana **Explore tab** with PromQL query: http\_server\_duration\_count  
A line chart showing different route usage

**Use this for:**  
Request Count Trend  
Route-level Traffic Analysis  
Use it in thesis section on **HTTP metrics** or **Grafana Dashboard**