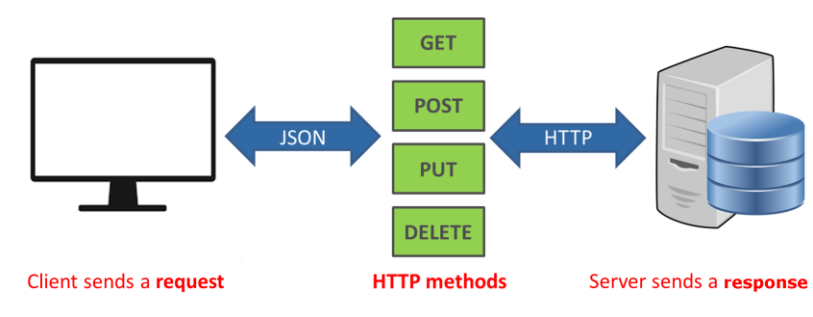
IT4031 – VAUED Assignment 02 Final Report – Group 2025_A1_G28

1. Introduction

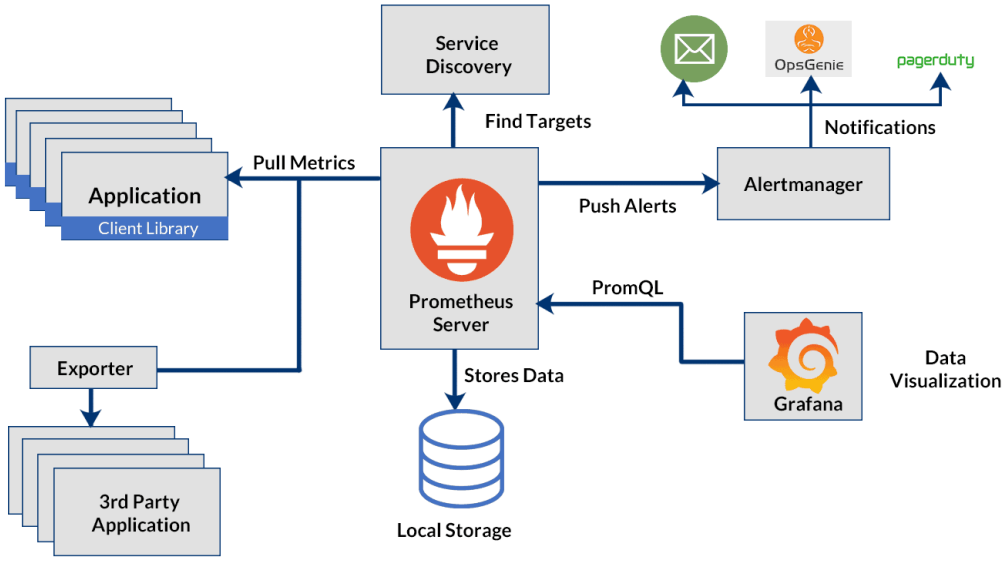
This report presents the development and deployment of a full-stack observability solution for an Employee Management REST API using Prometheus and Grafana. The system was designed to demonstrate real-world principles of modern application monitoring, API design, metric exposure, and alert-driven observability using open-source tools.

The project incorporates several foundational concepts:

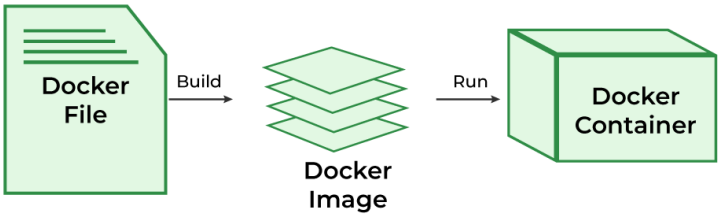
- **RESTful API**: A standardized web service interface that supports Create, Read, Update, and Delete (CRUD) operations. This enables structured communication between client and server.



- **Prometheus**: A time-series monitoring tool used to scrape and store metrics exposed by the API. It allows real-time insights into system behavior and supports queries using PromQL.



- **Metrics**: Quantifiable data points (like request count, error rate, and request duration) used to assess system health and performance. These are collected both automatically and through custom instrumentation.
- **Grafana**: A powerful visualization platform that integrates with Prometheus to display metrics on dashboards. It supports diverse chart types and enables users to track trends visually.
- **Alerting**: The ability to define rules that automatically trigger warnings or alerts based on metric thresholds. Grafana's unified alerting system handles rule evaluation and provides visibility into critical events.
- **Docker Compose**: A container orchestration tool that allows all services (API, Prometheus, Grafana, Node Exporter) to run together in isolated but coordinated containers, simplifying setup and deployment.



By integrating these concepts, the project provides a complete observability solution suitable for small-scale enterprise applications and educational demonstrations.

What is Observability?

Observability is the ability to understand what is happening inside a system based on the data it exposes, even if you cannot interact with it directly.

In simple terms, it means:

"Can I tell what's going wrong (or right) in my application just by looking at the data it outputs?"

Why Do We Implement Observability?

Modern applications run in complex, distributed environments. Without observability:

- You don't know if your app is working correctly
- You can't identify slowdowns, errors, or abnormal usage
- You can't detect issues before users complain

Benefits of Adding Observability

Aspect	Benefit
Metrics	Track how often employees are added, updated, deleted
Request tracking	Monitor how many users call the API, and which endpoints
Latency tracking	Identify slow requests or overloaded endpoints
Alerts	Get notified when error counts or traffic spikes occur
Visualization	Use Grafana to see everything in real-time
Debugging	Use Prometheus to troubleshoot after an issue has happened

2. Objective

The primary objective of this project was to develop a RESTful Employee Management API and integrate it with a full observability stack using Prometheus and Grafana. The goal was not only to implement CRUD operations but also to expose and monitor key application metrics, visualize performance data, and automate alerting critical events. The system was deployed using Docker Compose to ensure ease of setup, reproducibility, and service orchestration across different environments.

2. Technology Stack

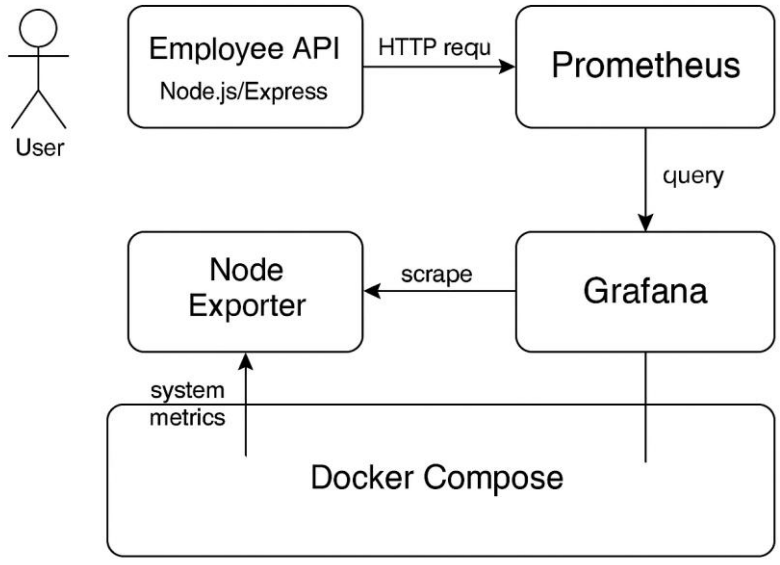
- **Node.js + Express**: Used to build the core RESTful API that handles CRUD operations for employee records.
- **Prometheus**: Acts as the monitoring backend by scraping exposed metrics from the API and collecting time-series data.
- **Grafana**: Visualizes metrics from Prometheus using customizable dashboards and provides alerting capabilities based on thresholds.
- **Docker & Docker Compose**: Enables containerization and orchestrated deployment of all services (API, Prometheus, Grafana, Node Exporter).
- **Node Exporter**: An additional service that collects system-level metrics (CPU, memory, disk) for Grafana to visualize.

4. System Architecture

The observability system is composed of the following components, orchestrated using Docker Compose:

- **Employee API**: A Node.js/Express service that handles HTTP requests related to employee data. It also exposes Prometheus-compatible metrics via the /metrics endpoint.
- **Prometheus**: Scrapes metrics from the API and Node Exporter at regular intervals and store them as time-series data.
- **Grafana**: Connects to Prometheus as a data source and displays the metrics in rich dashboards. It also evaluates alert rules based on defined thresholds.
- **Node Exporter**: Provides operating system metrics like CPU usage, memory, and disk I/O.
- **Docker Compose**: Manages the lifecycle and network configuration of all services, enabling consistent and isolated environments.

The diagram above illustrates how the services interact. Prometheus pulls data from both the API and Node Exporter, and Grafana queries Prometheus to power the dashboards and alerts. All components are deployed and managed together using Docker Compose.



5. API Design and Implementation

5.1. Building the API – server.js

The Employee Management API supports the following endpoints. Created a Node.js + Express API that supports:
- GET /employees
- GET /employees/:id
- POST /employees
- PUT /employees/:id
- DELETE /employees/:id

All changes to employee data and error scenarios are counted and tracked using Prometheus custom counters.

prom-client to define metrics:

- http_requests_total – counts each API call
- employees_added_total, updated, deleted
- errors_total – counts all client/server-side errors
- http_request_duration_seconds – tracks request duration for histograms

Added Persistence:

- All employee data and counter values are saved to data.json
- Loaded at startup so nothing is lost when the container restarts

Exposed Metrics Endpoint**:**

- /metrics returns all Prometheus-compatible metrics

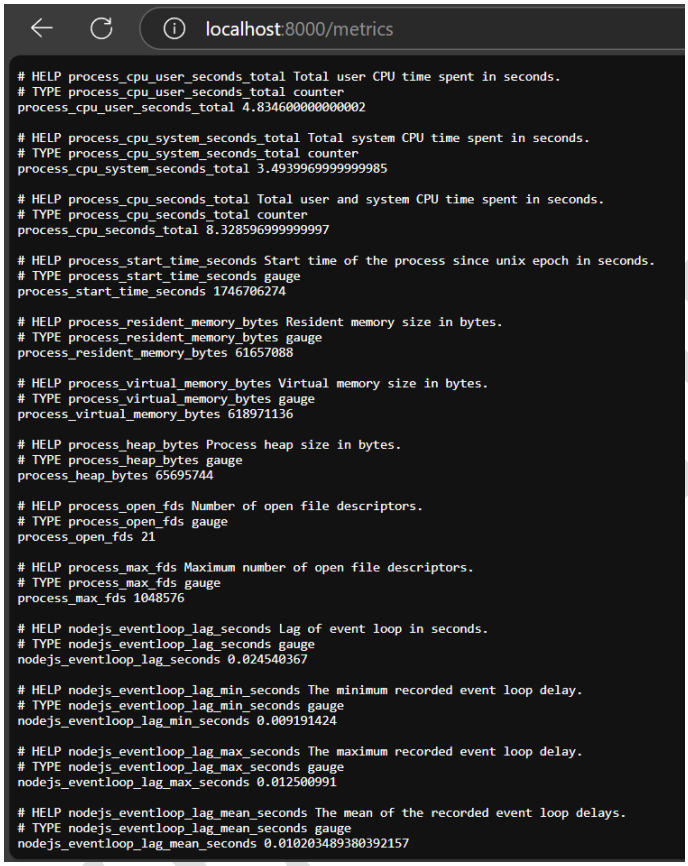
app > server.js > requestCounter

```
// Import required modules
const express = require('express');          // Web framework for building REST APIs
const client = require('prom-client');       // Prometheus metrics client
const fs = require('fs');                    // File system for data persistence
const path = require('path');                // To resolve file paths
const app = express();                       // Create an Express app instance
const port = 8000;                           // Define the port the app will listen on

app.use(express.json()); // Middleware to parse JSON request bodies

const DATA_FILE = path.join(__dirname, 'data.json');

// Load Persistent Data (if exists)

let employees = {
    1: { name: "John Doe", position: "Manager", salary: 80000 },
    2: { name: "Jane Smith", position: "Developer", salary: 60000 }
};
let nextId = 3;
let counterState = {
    employees_added_total: 0,
    employees_updated_total: 0,
    employees_deleted_total: 0,
    errors_total: 0
};

if (fs.existsSync(DATA_FILE)) {
    const data = JSON.parse(fs.readFileSync(DATA_FILE, 'utf8'));
    employees = data.employees || employees;
    nextId = data.nextId || nextId;
    counterState = data.counterState || counterState;
}
```

```
// Prometheus Metric Configuration

client.collectDefaultMetrics();

const requestCounter = new client.Counter({
    name: 'http_requests_total',
    help: 'Total number of HTTP requests',
    labelNames: ['method', 'endpoint']
});

const employeeAddedCounter = new client.Counter({
    name: 'employees_added_total',
    help: 'Total number of employees added'
});

const employeeUpdatedCounter = new client.Counter({
    name: 'employees_updated_total',
    help: 'Total number of employee updates'
});

const employeeDeletedCounter = new client.Counter({
    name: 'employees_deleted_total',
    help: 'Total number of employee deletions'
});

const errorCounter = new client.Counter({
    name: 'errors_total',
    help: 'Total number of errors (e.g., not found, bad request)'
});

const httpRequestDurationSeconds = new client.Histogram({
    name: 'http_request_duration_seconds',
    help: 'Duration of HTTP requests in seconds',
    labelNames: ['method', 'route', 'code'],
    buckets: [0.1, 0.3, 0.5, 1, 1.5, 2, 5] // seconds
});

// Restore counter values
employeeAddedCounter.inc(counterState.employees_added_total);
employeeUpdatedCounter.inc(counterState.employees_updated_total);
employeeDeletedCounter.inc(counterState.employees_deleted_total);
errorCounter.inc(counterState.errors_total);
```

```
// Middleware to Track Requests Automatically

app.use((req, res, next) => {
    const end = httpRequestDurationSeconds.startTimer();
    res.on('finish', () => {
        const routePath = req.route?.path || req.path;
        requestCounter.labels(req.method, routePath).inc();
        end({ method: req.method, route: routePath, code: res.statusCode });
    });
    next();
});

// API ROUTES

app.get('/employees', (req, res) => {
    res.json(employees);
});

app.get('/employees/:id', (req, res) => {
    const employee = employees[req.params.id];
    if (!employee) {
        errorCounter.inc();
        counterState.errors_total++;
        saveData();
        return res.status(404).json({ error: 'Employee not found' });
    }
    res.json(employee);
});

app.post('/employees', (req, res) => {
    const { name, position, salary } = req.body;
    if (!name || !position || !salary) {
        errorCounter.inc();
        counterState.errors_total++;
        saveData();
        return res.status(400).json({ error: 'Name, position, and salary are required.' });
    }
    const id = nextId++;
    employees[id] = { name, position, salary };
    employeeAddedCounter.inc();
    counterState.employees_added_total++;
    saveData();
    res.status(201).json({ id, ...employees[id] });
});
```

```
app.put('/employees/:id', (req, res) => {
    const employee = employees[req.params.id];
    if (!employee) {
        errorCounter.inc();
        counterState.errors_total++;
        saveData();
        return res.status(404).json({ error: 'Employee not found' });
    }
    const { name, position, salary } = req.body;
    if (name) employee.name = name;
    if (position) employee.position = position;
    if (salary) employee.salary = salary;
    employeeUpdatedCounter.inc();
    counterState.employees_updated_total++;
    saveData();
    res.json(employee);
});

app.delete('/employees/:id', (req, res) => {
    if (!employees[req.params.id]) {
        errorCounter.inc();
        counterState.errors_total++;
        saveData();
        return res.status(404).json({ error: 'Employee not found' });
    }
    delete employees[req.params.id];
    employeeDeletedCounter.inc();
    counterState.employees_deleted_total++;
    saveData();
    res.status(204).send();
});

app.get('/metrics', async (req, res) => {
    res.set('Content-Type', client.register.contentType);
    res.end(await client.register.metrics());
});

function saveData() {
    fs.writeFileSync(DATA_FILE, JSON.stringify({ employees, nextId, counterState }, null, 2));
}

app.listen(port, () => {
    console.log(`Employee Management API running at http://localhost:${port}`);
});
```



5.2. Setting Up Prometheus

We configured Prometheus to scrape metrics from:

- employee-api:8000/metrics
- node-exporter:9100/metrics

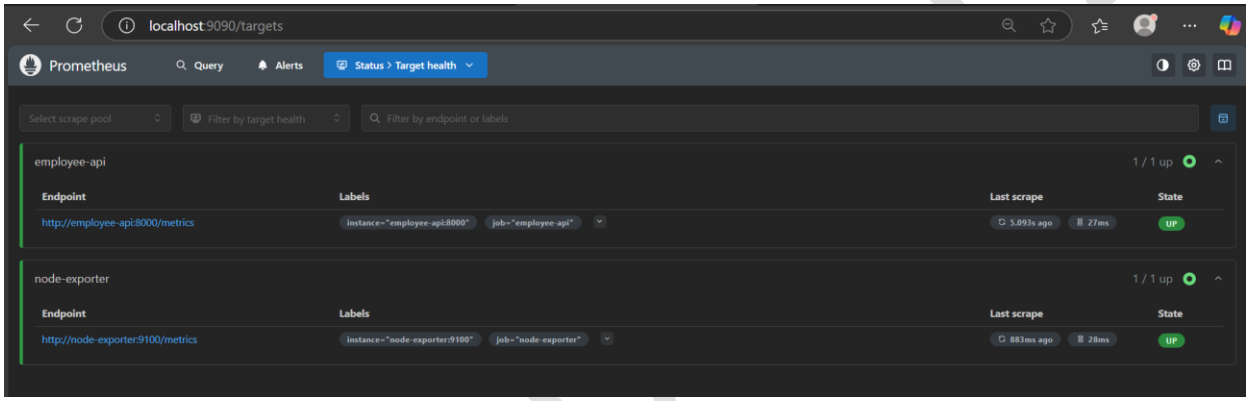
Steps taken:

- Defined scrape jobs in prometheus/prometheus.yml
- Set scrape_interval: 5s for near real-time tracking
- Launched Prometheus UI at http://localhost:9090 to:
    - Test queries using PromQL (e.g. rate(employees_added_total[1m]))
    - Check if targets were UP

Why Prometheus?
It acts as a time-series database and central hub for all metrics from the API and host system.

prometheus > prometheus.yml

```yaml
# File: prometheus/prometheus.yml

global:
  scrape_interval: 5s

scrape_configs:
  - job_name: 'employee-api'
    static_configs:
      - targets: ['employee-api:8000']

  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter:9100']
```



5.3. Creating Grafana Dashboards

Grafana was used to **visualize all metrics and define alert rules**.

Steps taken:

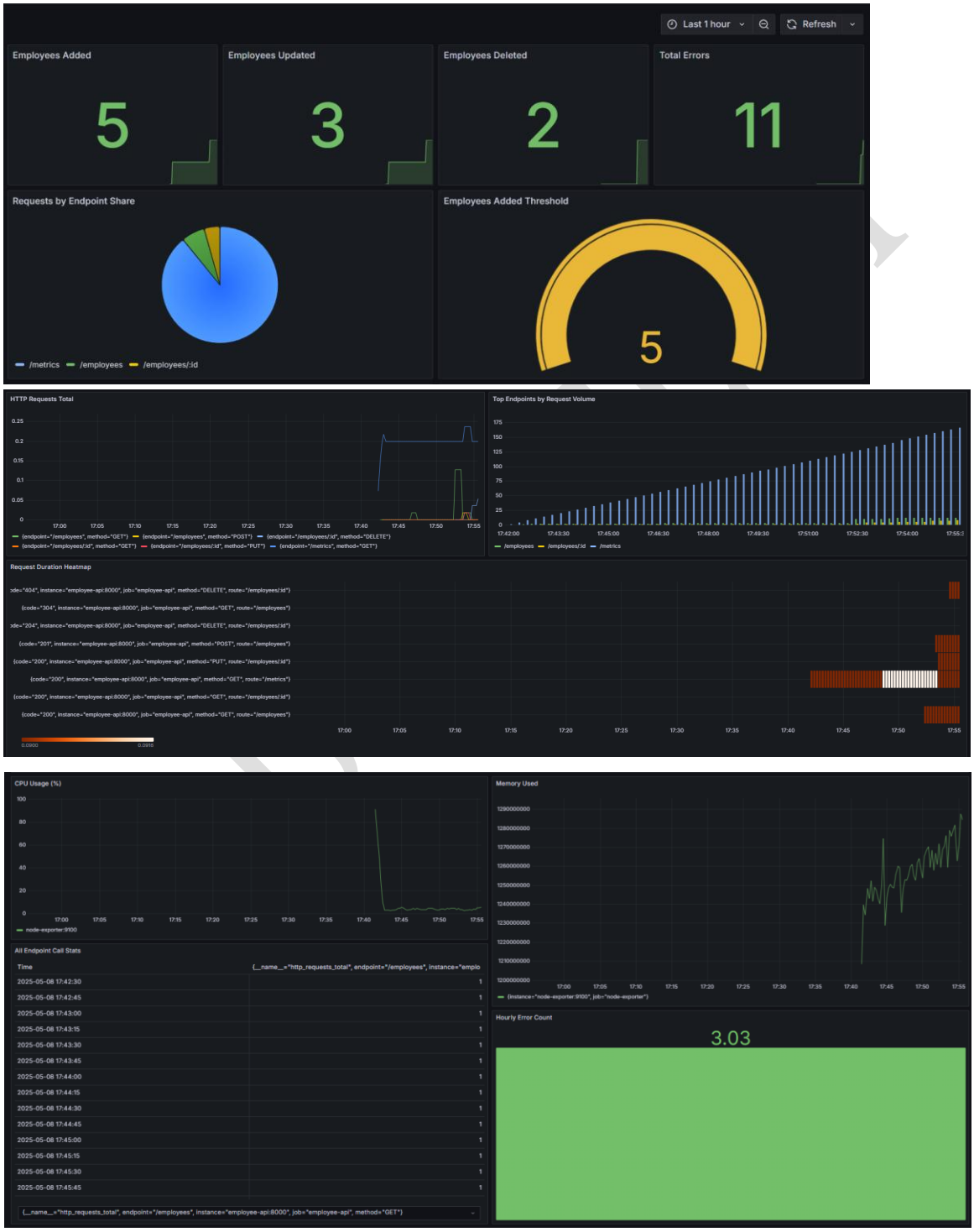
- Opened Grafana at http://localhost:3000
- Logged in as admin / vaued
- Added Prometheus as a data source
- Imported a custom dashboard (grafana/dashboard.json)

Visualizations created:

The Grafana dashboard includes the following visualization types:
- Stat Panels: Total employees added, updated, deleted, errors
- Time Series: HTTP requests over time, CPU usage, memory usage
- Bar Chart: Top request endpoints
- Gauge: Threshold tracking for employee additions
- Heatmap: Request duration distribution using histograms
- Table: Raw metric listings by endpoint or method

These visualizations allow for immediate and insightful understanding of application and system performance.

Why Grafana?
It provides powerful visualizations, supports alerts, and integrates natively with Prometheus.



5.4. Docker and Containerization

All services were containerized using Docker Compose.

Services defined:

- employee-api: The Node.js backend
- prometheus: For metrics collection
- grafana: For visualization and alerts
- node-exporter: For system-level metrics

Benefits of Docker:

- Easy to run across machines
- Consistent environment
- No dependency installation required
- Persistent volumes for Grafana and data.json

All services were defined in a `docker-compose.yml` file. The API was built from a custom Dockerfile. Persistent volumes were used for Grafana dashboards and Prometheus data. The system can be rebuilt and started using:
- `docker-compose up --build`
- `docker-compose down`
The `data.json` file is used to persist employee data and counter states across container restarts.

app > Dockerfile > ...

```
# Use the official Node.js image from Docker Hub
FROM node:18

# Set working directory inside the container
WORKDIR /app

# Copy package.json and package-lock.json if available
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the entire app directory contents into the container
COPY . .

# Expose the API port
EXPOSE 8000

# Command to run your application
CMD ["node", "server.js"]
```

docker-compose.yaml > ...

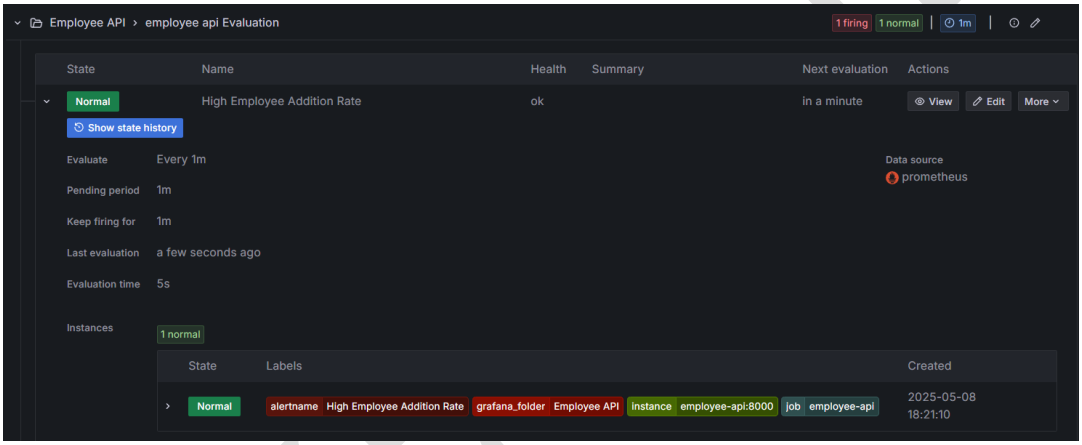
```
Run All Services
services:
  Run Service
  employee-api:
    build: ./app
    container_name: employee-api
    ports:
      - "8000:8000"
    volumes:
      - ./app:/app

  Run Service
  prometheus:
    image: prom/prometheus
    container_name: prometheus
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  Run Service
  grafana:
    image: grafana/grafana
    container_name: grafana
    ports:
      - "3000:3000"
    volumes:
      - ./grafana:/etc/grafana/provisioning
      - grafana-storage:/var/lib/grafana
    depends_on:
      - prometheus

  Run Service
  node-exporter:
    image: prom/node-exporter
    container_name: node-exporter
    ports:
      - "9100:9100"

volumes:
  grafana-storage:
```

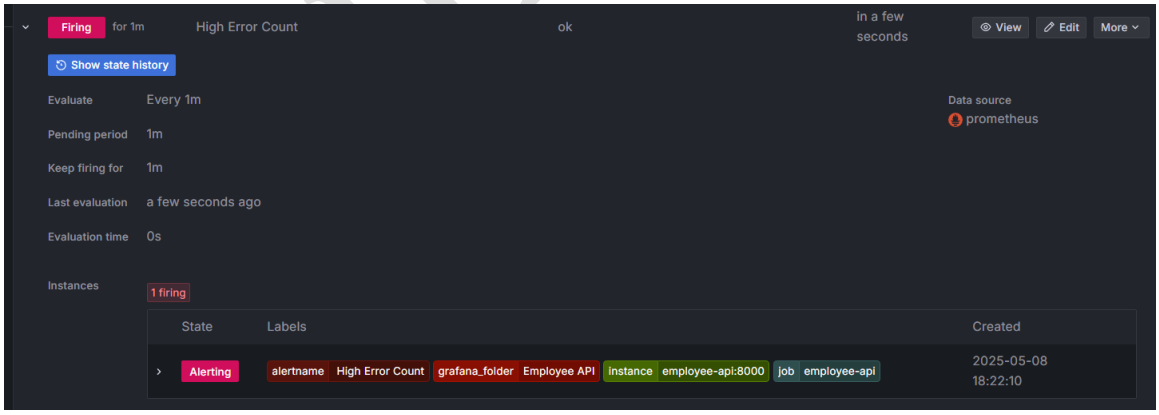
Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
vaued_assignment2	-	-	-	1.06%	21 minutes ago	
node-exporter	69e947992078	prom/node-exporter	9100:9100	0%	21 minutes ago	
employee-api	72de5aebce82	vaued_assignment2-employee-api	8000:8000	0.68%	21 minutes ago	
prometheus	44cfb7240b28	prom/prometheus	9090:9090	0%	21 minutes ago	
grafana	3c3dbb250ca9	grafana/grafana	3000:3000	0.38%	21 minutes ago	

5.5. Alerting Configuration

Grafana alerts were created using the unified alerting system. Two major alerts were configured:
- High Employee Addition Rate: Triggered if `rate(employees_added_total[1m]) > 5`
- High Error Count: Triggered if `errors_total > 5`

Alerts were evaluated every minute, and their state (OK, Firing) was visible in Grafana's Alerting tab.





6. Conclusion

The assignment was successfully completed with a fully functional, observable REST API for employee management. All three tasks were implemented: a working web service, real-time monitoring using Prometheus, rich Grafana dashboards, alert rules for automated observability, and containerized deployment for repeatability and portability.