**1.1 Monitoring**

Using Prometheus to gather the metrics for Kubernetes service and visualizing the data by leveraging visualization capabilities of Grafana.

* + 1. **Helm**

Deploying Prometheus and Grafana to the EKS with the help oof Helm.

A black screen with white text

Description automatically generated

Components of the kube-prometheus stack:

* **Prometheus**: A time series database that collects and stores metrics from Kubernetes clusters
* **Alertmanager**: Manages alerts for metrics that exceed preconfigured thresholds
* **Grafana**: Visualizes metrics and plot data using dashboards
* **Exporters**: Metric collection agents that expose their metrics via an HTTP endpoint
* **Prometheus Operator**: Spins up and manages Prometheus instances

Below are services running :

A screen shot of a computer screen

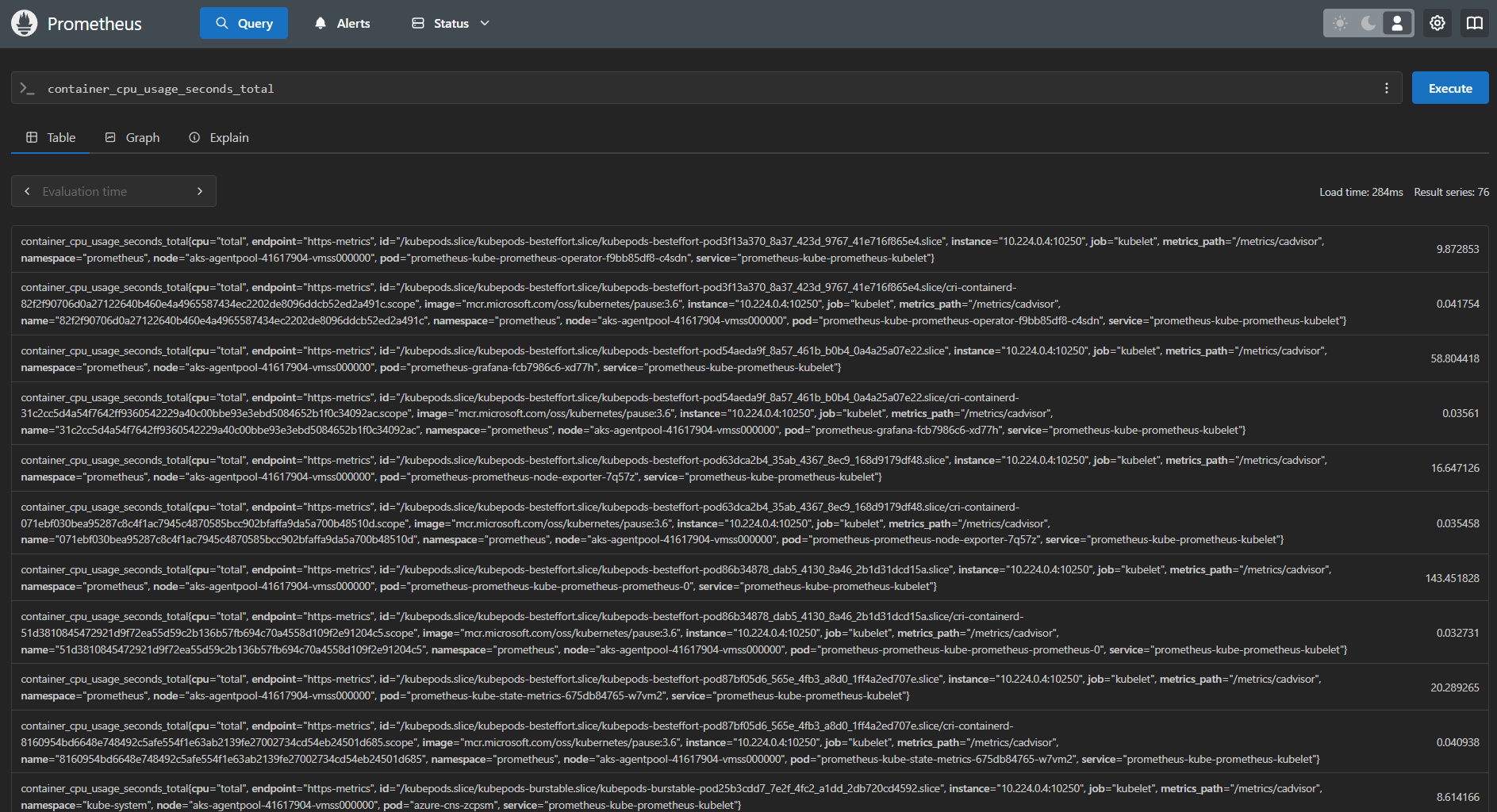
Description automatically generated

* + 1. **Prometheus**

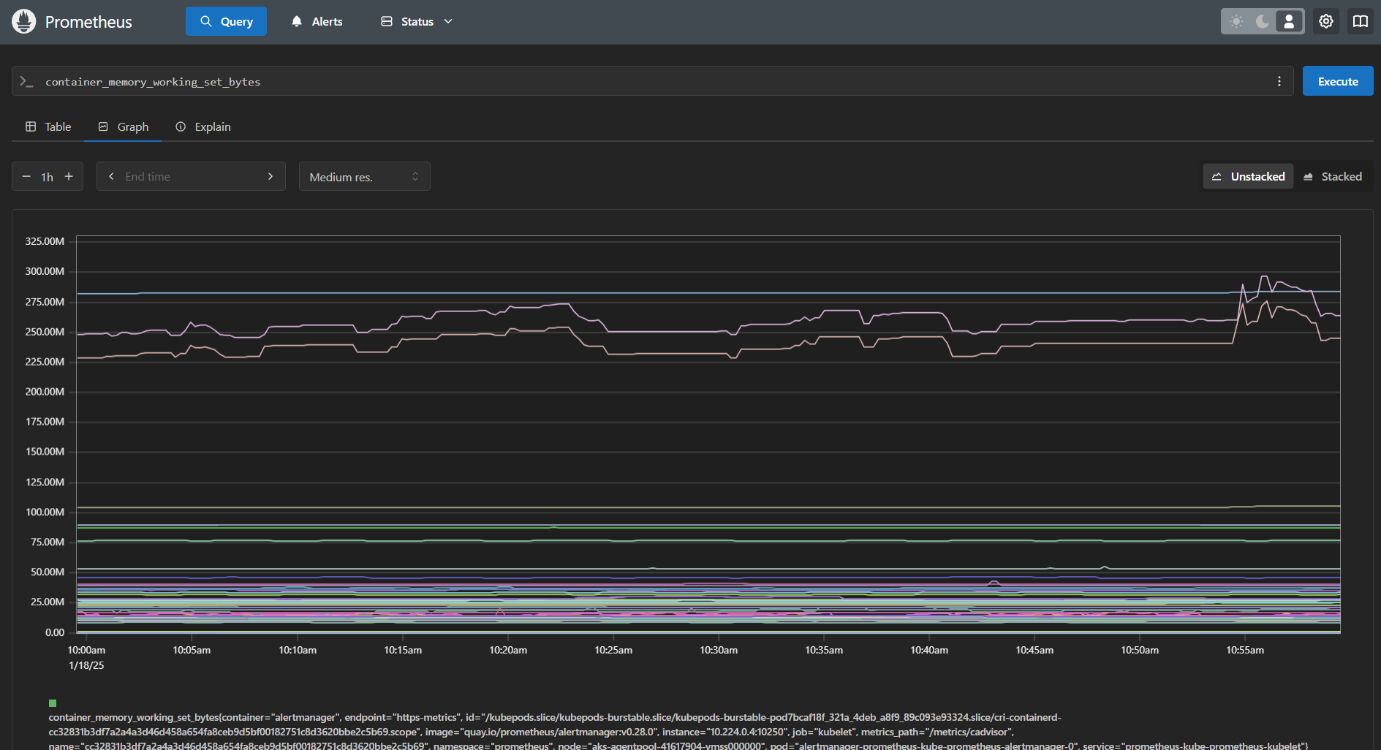
Prometheus is configured to scrape and collect metrics from various services using exporters and Kubernetes’ built-in APIs. Below are the key components of the Prometheus setup:

* **Node Exporter**: Collects CPU, memory, disk I/O, and network statistics from Kubernetes nodes.
* **cAdvisor**: Gathers container-level metrics such as CPU, memory, and disk usage for all pods.
* **Application-specific Exporters**: Custom exporters for backend, frontend, and Redis services to expose application-specific metrics.

CPU usage:



Memory Usage:



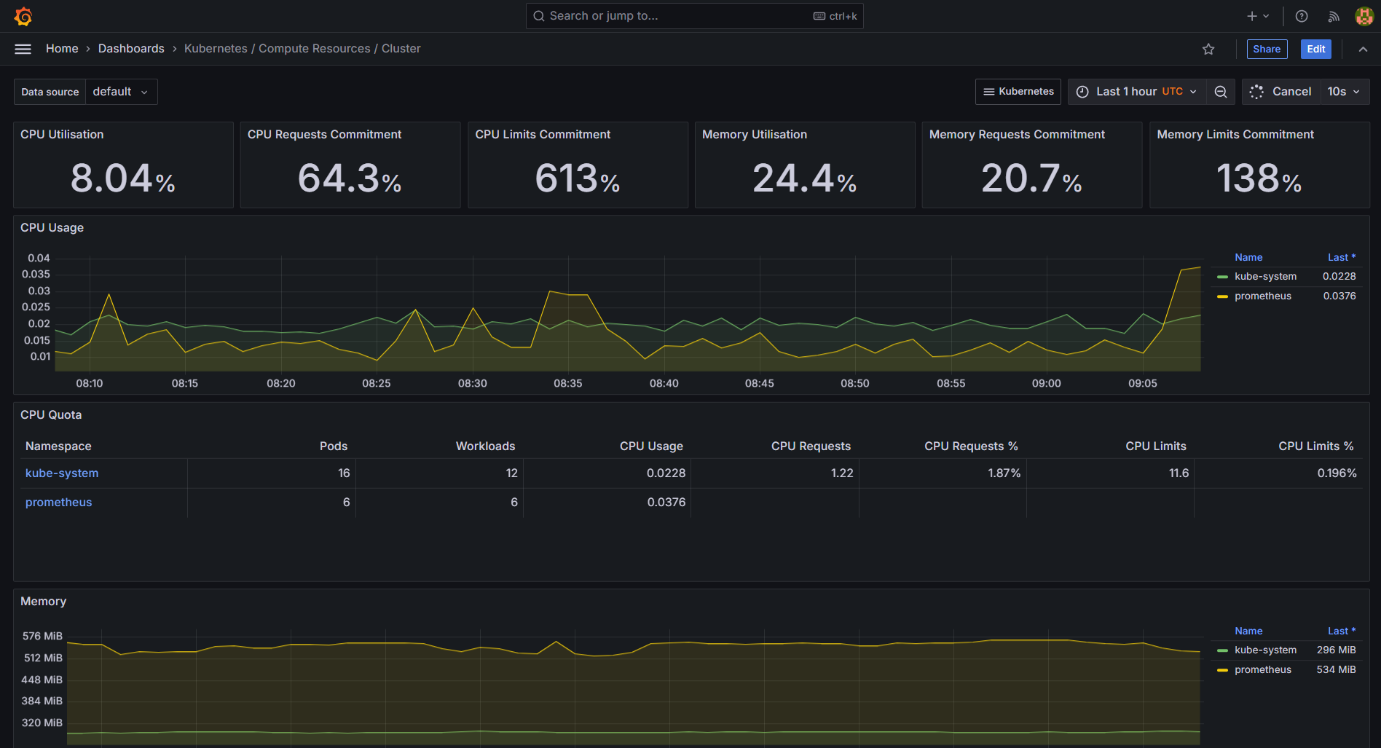
* + 1. **Grafana**

Connecting to Grafana running in EKS by port forwarding on to local machine:

A black screen with white text

Description automatically generated

Grafana Cluster Dashboard:



**1.1.4 Metrics Monitored**

* **CPU and Memory Usage**
  + Per-container resource utilization to track performance and detect resource constraints.
* **HTTP Request Metrics**
  + Request rates, latencies, and error rates for both the backend and frontend services.
* **Redis Performance Metrics**
  + Memory usage.
  + Hit/miss rates to evaluate caching efficiency.

**1.2 Performance**

Load-Testing: Applied load on to the EKS which increases the utilization of CPU to around 80 percent.

A screenshot of a computer

Description automatically generated

Load testing was conducted on the backend service using **JUnit** to simulate a workload of 100 users over 5 minutes with an increasing request rate.

**1.2.1 Observations and Bottlenecks**

* **Backend Latency**: Increased request latency under high load.
* **Redis Performance**: High miss rates during concurrent requests.
* **CPU Utilization**: CPU usage exceeded 80% during peak load.

**1.2.2 Recommendations**

1. **Horizontal Scaling**:
   * Increase the number of replicas for the backend service to distribute the load.
   * Deploy Redis with a master-slave configuration for scaling read operations.
2. **Database Optimization**:
   * Tune Redis eviction policies to improve memory utilization.
   * Pre-load commonly accessed data to minimize misses.
3. **Application Tuning**:
   * Implement caching for frequently accessed backend responses.
   * Use request batching to reduce the number of independent Redis queries.

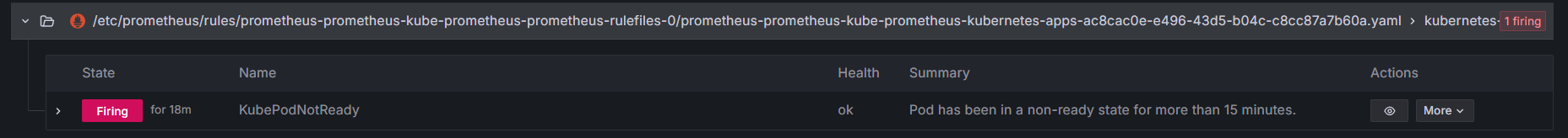
**1.3 Alerting**

Alerts: Rules are written for alerts for CPU, that got initiated once a certain condition happens they automatically got fired.

A screenshot of a computer

Description automatically generated

As shown below, an alert gets fired based on the alert rule defined in it.



**1.4 Summary of Findings**

* **Observations:**
* Backend latency increased under load.
* Redis memory usage reached critical thresholds.
* CPU usage peaked, indicating a need for scaling.
* **Proposed Improvements:**
* Horizontal scaling for backend and Redis services.
* Database optimizations for Redis.
* Application-level caching and request batching.