


GRADUATE SYSTEMS (CSE-638)

Project Report on Google Cloud Microservices Demo for Web-Based E-Commerce App Analysis

Group Details:
MT24031 Deepankar Verma
MT24092 Siddhartha Vardhan
MT24057 Nakul Panwar
MT24122 Kanishk Saraswat

Project Report on Google Cloud Microservices Demo for E-Commerce Analysis

1. Abstract

The Google Cloud Microservices Demo for E-Commerce is a cloud-first microservices-based e-commerce application designed by Google Cloud. It demonstrates modern cloud-native development, microservices architecture, and scalable application deployment using Kubernetes.

This project involves setting up and analyzing the system's architecture, deployment process, service interactions, and performance profiling. The application was installed and run on a local system using Docker and Kubernetes to evaluate its functionality and scalability.

Official GitHub Repository: Google Cloud Microservices Demo

Forked Repository: GRS_GOOGLE_PROJECT

2. Problem Statement

E-commerce platforms require scalable, fault-tolerant, and high-performance architectures to handle dynamic workloads and transactions. Traditional monolithic applications struggle with scalability, maintenance, and resiliency. This project aims to analyze a microservices-based alternative that addresses these challenges using containerization and Kubernetes orchestration.

3. Project Goals

- Understand the microservices architecture and its implementation in cloud-native environments.
- Deploy the Online Boutique application on a Kubernetes cluster and evaluate its scalability.
- Analyze system performance, fault tolerance, and security measures.
- Profile the application to detect potential bottlenecks and optimize response times.

4. Expected Outcomes/Deliverables

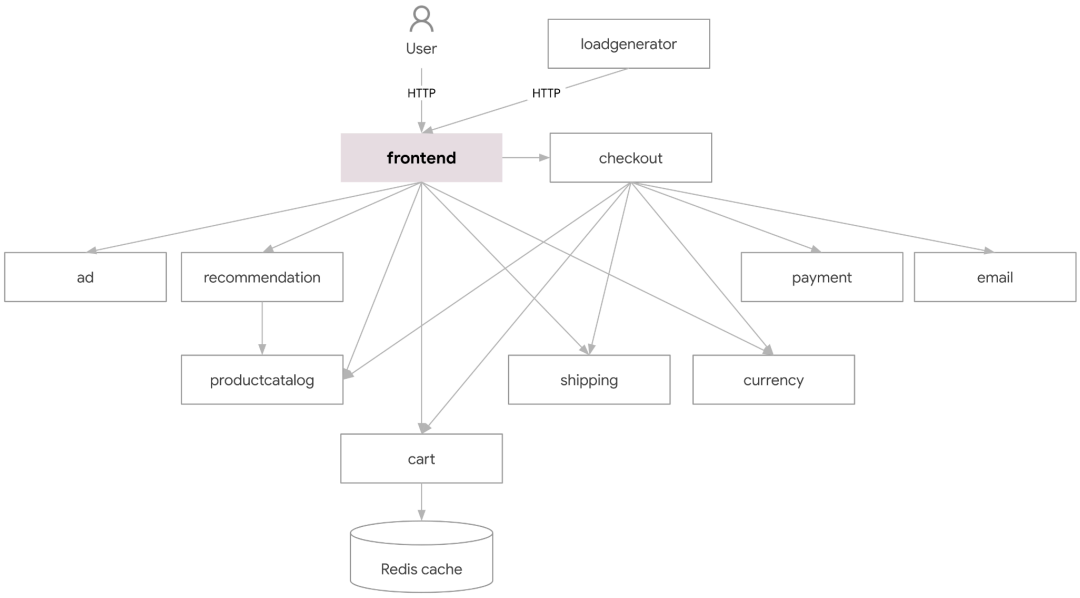
- Successfully deployed Google Cloud Microservices Demo in a Kubernetes environment.
- Performance profiling and identification of bottlenecks in microservices interactions.
- Real-time monitoring and observability setup for better service performance analysis.

5. Technologies Used

Technology	Purpose
Google Kubernetes Engine (GKE)	Cloud-based container orchestration
Docker	Containerization of microservices
Kubernetes	Deployment and management of microservices
gRPC	High-performance inter-service communication
Redis, Cloud Spanner, Memorystore, AlloyDB	Data storage and caching
Terraform	Infrastructure automation (optional)

6. System Design

The Online Boutique consists of 11 microservices, each responsible for a specific functionality. The services communicate over gRPC for efficient interaction.



Microservices Overview

Service	Language	Description
Frontend	Go	Web interface for users to browse and purchase items
Cart Service	C#	Stores and retrieves shopping cart items in Redis
Product Catalog	Go	Provides a list of products from a JSON file

Currency Service	Node.js	Converts currencies based on real-time exchange rates
Payment Service	Node.js	Simulates credit card transactions
Shipping Service	Go	Provides shipping cost estimates and mock order fulfillment
Email Service	Python	Sends order confirmation emails
Checkout Service	Go	Orchestrates order processing and payment
Recommendation Service	Python	Recommends products based on cart contents
Ad Service	Java	Displays targeted advertisements
Load Generator	Python	Simulates user traffic for testing

Key Architectural Features

•Microservices-Based: Independent services ensure modularity and scalability.

•gRPC Communication: Faster and more efficient than REST-based APIs.

•Kubernetes Deployment: Allows auto-scaling and high availability.

•Cloud-Native Design: Optimized for Google Cloud but can be run locally.

7. Profiling Strategies for Performance and Security

7.1 Observability usingAPI Traffic Profiling and Performance Monitoring

● Implements Prometheus and Grafana for API latency and request rate analysis.

7.2 Load Testing using Ngrok and Locust

● Ngrok was used to create an http tunnel to the external IP and then Locust is used to simulate multiple users sending http request's.

7.3 Kubernetes Observability and Traffic Analysis

● Uses Kiali, Kubeshark, Spekt8, and Speedscale to monitor API calls and network traffic.

● Provides real-time visualization of microservices interactions.

7.1 OBSERVABILITY

Overview of Prometheus and Grafana

Prometheus is an open-source monitoring and alerting toolkit. It collects metrics in a time-series format and uses PromQL for powerful querying. Grafana is a visualization tool that allows users to create dashboards using Prometheus data. Together, they form a robust observability stack. Observability is the **capability to measure the internal state of a system by examining its outputs**—like logs, metrics, and traces. It's more than just monitoring; it's about gaining deep insight into how your system behaves and why.

Prometheus:

- Pull-based scraping:

  Prometheus fetches metrics by regularly polling endpoints. This makes it easier to control and scale metric collection.

- Time-series database:

  Stores data with timestamps, allowing you to analyze trends and patterns over time.

- Service discovery:

  Automatically finds targets (like Kubernetes pods) to scrape, adapting dynamically as the infrastructure changes.

- Alerting capabilities:

  Supports rules to trigger alerts when metrics exceed thresholds, integrating with tools like Alertmanager or Slack

Grafana:

- Dashboard visualization:

  Grafana displays metrics as interactive charts, graphs, and gauges for real-time system insights.

- Alerting:

  Set up threshold-based alerts to notify teams via email, Slack, or other channels when issues arise.

- Multi-datasource support:

  Grafana can connect to various backends like Prometheus, InfluxDB, MySQL, and more in a single interface.

- Custom panels:

  Lets you build flexible and personalized dashboards using different visualization types

Installing Prometheus & Grafana using Helm in Kubernetes

The steps to install Grafana and Prometheus were as follows:

1. **Create a Monitoring Namespace**

First, isolate all monitoring-related components into a dedicated namespace:

Command: kubectl create namespace monitoring

2. **Add the Helm Repository**

Command:
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts helm repo update

3. **Install Prometheus Stack (which includes Grafana)**

The kube-prometheus-stack chart bundles Prometheus, Grafana, and other exporters like Alertmanager, Node Exporter, and Kube State Metrics.

Command:

```
helm install prometheus prometheus-community/kube-prometheus-stack \
  --namespace monitoring
```

Services Inside the monitoring Namespace

The monitoring namespace now hosts multiple interconnected components:

Service	Description
prometheus-operated	Core Prometheus instance responsible for metric collection. Scrapes data from Kubernetes nodes, pods, and services.
alertmanager-operated	Manages alert routing, silencing, and grouping. Works in conjunction with Prometheus to handle alert notifications.
grafana	Web-based UI for visualizing metrics and dashboards. Supports Prometheus as a data source and includes prebuilt dashboards.
kube-state-metrics	Exposes Kubernetes resource states (e.g., Deployments, Nodes, Pods) in a format suitable for Prometheus consumption.
node-exporter	Collects low-level system metrics like CPU, memory, disk, and network stats from each Kubernetes node.
prometheus-kube-controller-manager	Exposes metrics from the Kubernetes controller manager for monitoring.
prometheus-kube-scheduler	Provides metrics from the Kubernetes scheduler, useful for performance analysis and debugging.
prometheus-operator	Manages Prometheus CRDs (Custom Resource Definitions), streamlining setup, configuration, and lifecycle management.

DASHBOARDS USING GRAFANA:

We have created two separate dashboards:

1-)Node Metrics Dashboard

- Idle CPU & System Load:
- Shows CPU core usage and system load over time (1m, 5m, 15m averages).
- Useful for understanding how taxed the node is.
- Memory Usage:

  ->Displays used, cached, buffered, and free memory.

  ->Large “Memory Usage” indicator (20.5%) provides a quick glance of memory health.

  ->Disk I/O:

    ->Shows disk read/write latency (I/O time) over time.

    ->Highlights how much of the disk space is in use (8.64%).

- Network Usage:
- Received/Transmitted data across network interfaces.
- Useful for spotting network congestion or unusually high traffic.

Use Case:

Ideal for node-level debugging, performance bottleneck identification, and hardware resource monitoring.

2-) Cluster and Services Dashboard (Image 2)

This dashboard provides a macro view of the entire Kubernetes cluster and services running inside it. It's designed for cluster-level health monitoring and service profiling.

Key Metrics Displayed:

- Network I/O Pressure:

  ->Shows incoming/outgoing network traffic over time for the entire cluster.

- Cluster Resource Utilization:

  ->Memory Usage: 14.8% of total 38.81 GiB used.

->CPU Usage: 6.16% average utilization.

->Provides overall health status of the cluster.

- Containers CPU Usage:

  ->1-minute average CPU usage per container.

  ->Highlights workloads consuming the most CPU.

- All Processes CPU Usage:

  ->Visualizes CPU usage across all running processes.

  ->Good for checking if specific processes are hogging CPU.

Use Case:

Ideal for monitoring overall cluster health, tracking service-level resource consumption.


Home › Dashboards › Kubernetes cluster monitoring (via Prometheus)
Network I/O pressure
Network I/O pressure
Total usage
Cluster memory usage
14.2%
Cluster CPU usage (1m avg)
6.34%
Used
5.52 GiB
Total
38.81 GiB
Containers CPU usage
Containers CPU usage (1m avg)
All processes CPU usage
All processes CPU usage (1m avg)
Containers memory usage
Containers memory usage

7.2 LOAD TESTING

Load testing is a type of performance testing that evaluates how a system behaves under expected or peak user load. Its goal is to determine a system's responsiveness, stability, and scalability by simulating multiple users or requests to ensure it can handle real-world traffic without crashing or slowing down.

Key Points:

- Simulates multiple users accessing the system simultaneously.
- Measures performance metrics like response time, throughput, and error rate.
- Helps identify performance bottlenecks before deployment

Load Testing is done via the help of Ngrok and Locust.

Ngrok

Ngrok is a cross-platform tool that allows developers to expose a local development server to the public internet securely. It is particularly useful for testing webhooks, demos, or sharing work-in-progress web apps without deploying to a production server**.**

Key Features of Ngrok:

- Instant Public URLs: Easily share your localhost with a globally accessible URL.
- Secure Tunnels: Ngrok encrypts traffic with HTTPS and provides authentication mechanisms.
- Custom Subdomains & Reserved Domains: Assign memorable, consistent URLs for services.
- Request Inspection: View real-time traffic and HTTP request logs via a web UI.

Ngrok Establishes a Tunnel:

- Ngrok client connects to the ngrok cloud service.
- The service assigns a public URL (e.g., https://a1b2c3d4.ngrok.io).
- The ngrok server acts as a proxy and forwards incoming requests to your local machine through a persistent connection.

Load testing showed spikes in usage during simulated traffic via ngrok and external request tools.

Ngrok Overview - Simulating Multiple HTTP Requests

Ngrok exposes local servers to the public internet. It was used to test the

application with real HTTP requests.

Steps:

1. Start server locally
2. Run `ngrok http 8080`
3. Get public URL (e.g., https://xyz.ngrok.io)
4. Send test requests to this URL

LOCUST

Locust is an open-source performance testing tool used **for load testing web applications and APIs**. It allows you to **simulate concurrent users** interacting with your system to test its performance, scalability, and reliability under load.

Key Features:

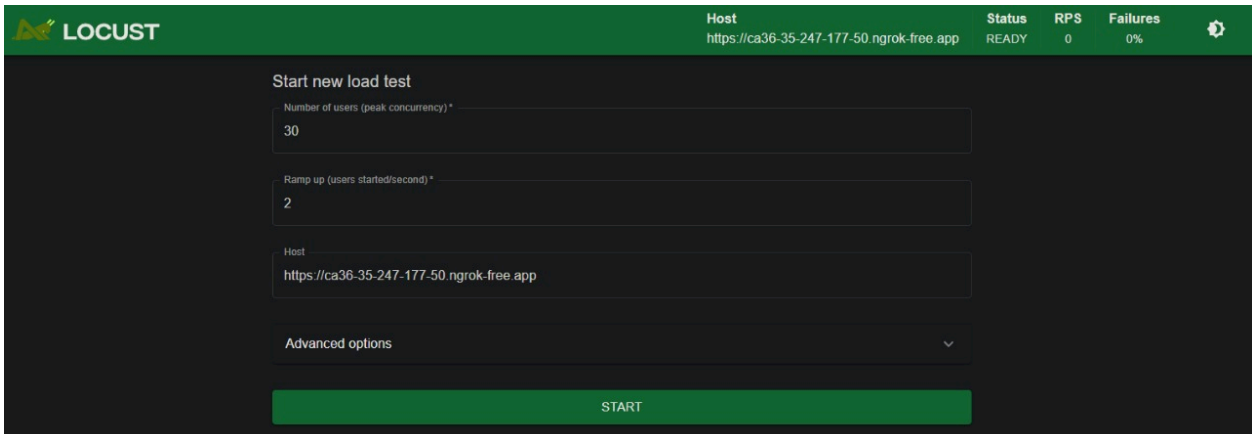
- **Python-based**: You define user behavior using Python code, making it flexible and easy to use.

- **Web UI**: Provides a real-time web-based UI to monitor test results.

- **Distributed & Scalable**: Supports running tests across multiple machines to simulate millions of users.

- **Customizable Workflows**: Easily simulate complex user interactions (like login, browsing, checkout, etc.).

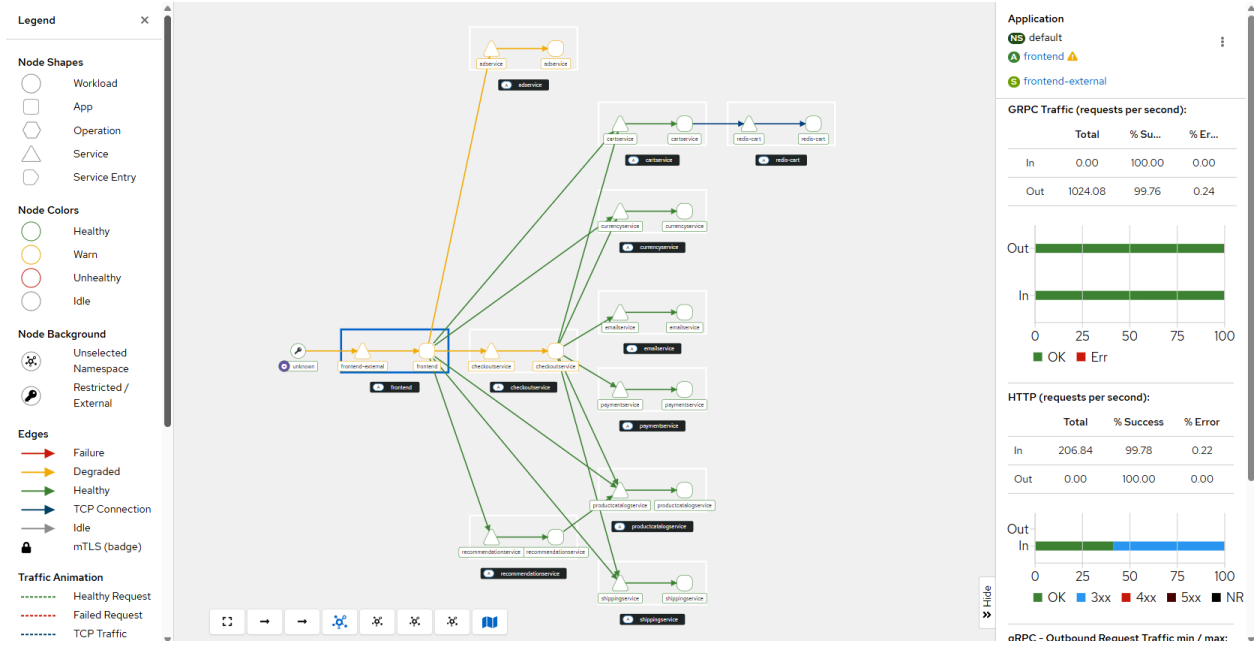
Below we showcase the various scenarios of Load Testing:
The testing with done with 2 replicas of each pod
Case-1: 20 users



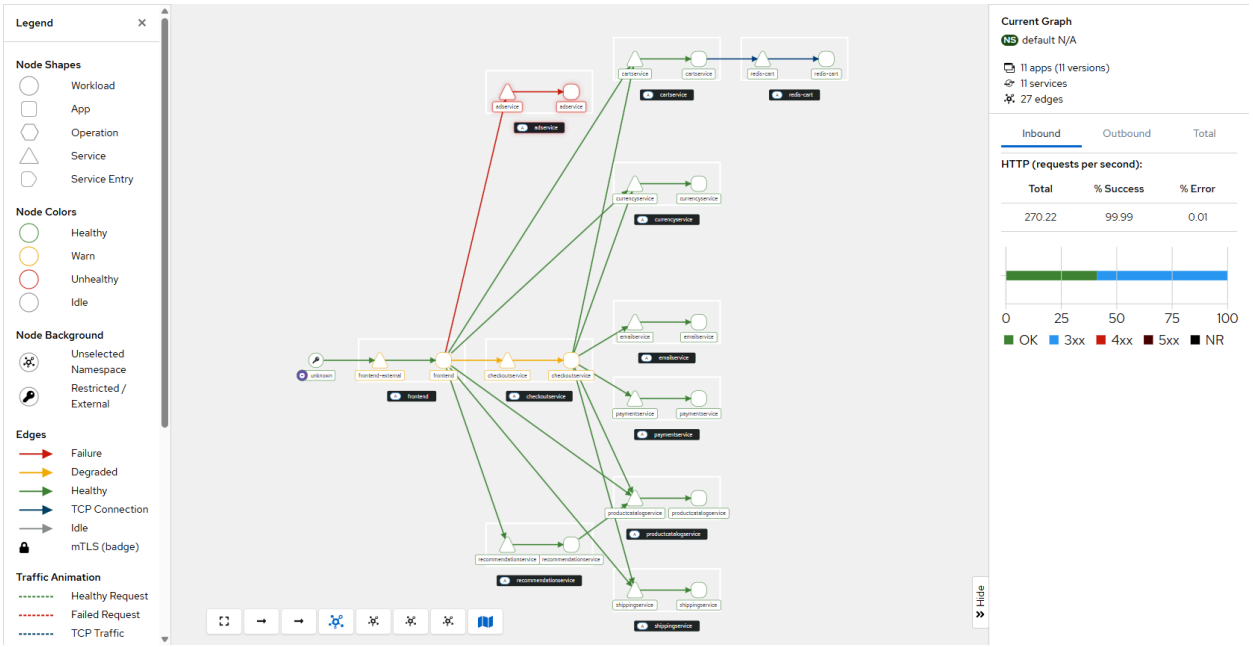
As we can observe the cpu usage increases from 6 percent to 21 percent.

Case-2: 500 users

But here the point to note is in the case of 500 users the Cpu usage just reached 23 percent but a new aspect was failure ramped up from 0 to 19 percent.



Some services (e.g., adservice) show warnings (orange color and yellow edges), indicating performance decline.
gRPC and HTTP success rates show small but notable error percentages (~0.24% and ~0.22% respectively).



One of the upstream services (adservice) is now marked unhealthy (red color), indicating real failures, not just warnings

7.3 Kubernetes Observability and Traffic Analysis

● Uses Kiali or Kubeshark or Spekt8, and Speedscale to monitor API calls and network
traffic.
● Provides real-time visualization of microservices interactions.

Istio , Kiali and weave-scope setup for Real Time Visualisation

Istio provides the intelligent service mesh layer for managing and securing microservice communication. Kiali offers a visual cockpit to understand Istio's topology, health, and configuration. Weave Scope complements this by providing a broader, real-time visualization of the entire Kubernetes cluster, showing the connections between pods, nodes, and processes, regardless of whether they are part of the Istio mesh or not.

Istio setup

```
kirusnovar@cloudshell:~$ istioctl version
client version: 1.24.3
control plane version: 1.24.3
data plane version: 1.24.3 (12 proxies)
kirusnovar@cloudshell:~$
```

Istio namespace

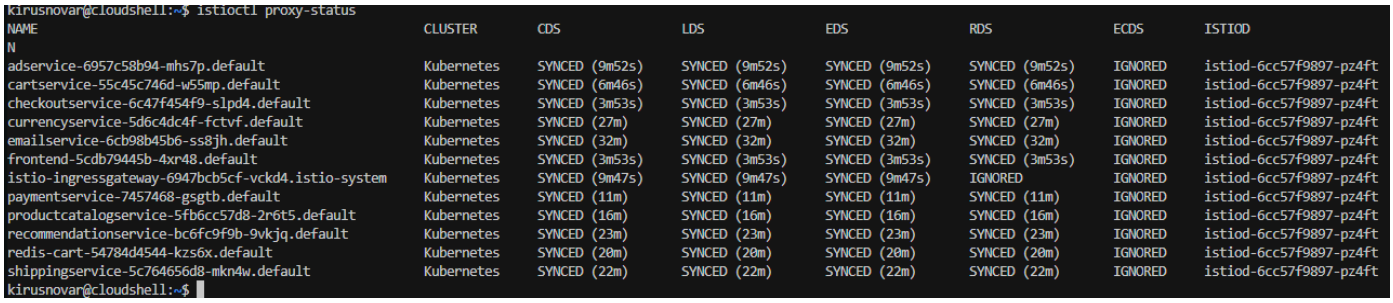
```
kirusnovar@cloudshell:~$ kubectl get pods -n istio-system | grep istio
istio-ingressgateway-6947bcb5cf-vckd4   1/1     Running   0          13h
istiod-6cc57f9897-pz4ft                 1/1     Running   0          13h
kirusnovar@cloudshell:~$
```

The output of the **kubectl get pods -n istio-system | grep istio** command shows two pods running in the istio-system namespace:

istio-ingressgateway-6947bcb5cf-vckd4: This is the Istio Ingress Gateway pod. It's responsible for managing traffic entering your service mesh from outside the cluster. It has been running for 13 hours.
istiod-6cc57f9897-pz4ft: This is the istiod pod, which is the core of the Istio control plane. It manages the configuration and provides services like traffic

routing rules, policy enforcement, and telemetry. It has also been running for 13 hours.
Both pods have a status of Running and have had 0 restarts, indicating they are operating normally. This confirms that the essential Istio components are up and running in your cluster.

Check Pods with Istio Sidecar (the "hooked in" pods):



```
kirusnovar@cloudshell:~$ istioctl proxy-status
NAME                                                   CLUSTER      CDS              LDS              EDS              RDS              ECDS         ISTIOD
N
adservice-6957c58b94-mhs7p.default                     Kubernetes   SYNCED (9m52s)   SYNCED (9m52s)   SYNCED (9m52s)   SYNCED (9m52s)   IGNORED      istiod-6cc57f9897-pz4ft
cartservice-55c45c746d-w55mp.default                   Kubernetes   SYNCED (6m46s)   SYNCED (6m46s)   SYNCED (6m46s)   SYNCED (6m46s)   IGNORED      istiod-6cc57f9897-pz4ft
checkoutservice-6c47f454f9-slpd4.default               Kubernetes   SYNCED (3m53s)   SYNCED (3m53s)   SYNCED (3m53s)   SYNCED (3m53s)   IGNORED      istiod-6cc57f9897-pz4ft
currencyservice-5d6c4dc4f-fctvf.default                Kubernetes   SYNCED (27m)     SYNCED (27m)     SYNCED (27m)     SYNCED (27m)     IGNORED      istiod-6cc57f9897-pz4ft
emailservice-6cb98b45b6-ss8jh.default                  Kubernetes   SYNCED (32m)     SYNCED (32m)     SYNCED (32m)     SYNCED (32m)     IGNORED      istiod-6cc57f9897-pz4ft
frontend-5cdb79445b-4xr48.default                      Kubernetes   SYNCED (3m53s)   SYNCED (3m53s)   SYNCED (3m53s)   SYNCED (3m53s)   IGNORED      istiod-6cc57f9897-pz4ft
istio-ingressgateway-6947bcb5cf-vckd4.istio-system     Kubernetes   SYNCED (9m47s)   SYNCED (9m47s)   SYNCED (9m47s)   IGNORED          IGNORED      istiod-6cc57f9897-pz4ft
paymentservice-7457468-gsgtb.default                   Kubernetes   SYNCED (11m)     SYNCED (11m)     SYNCED (11m)     SYNCED (11m)     IGNORED      istiod-6cc57f9897-pz4ft
productcatalogservice-5fb6cc57d8-2r6t5.default         Kubernetes   SYNCED (16m)     SYNCED (16m)     SYNCED (16m)     SYNCED (16m)     IGNORED      istiod-6cc57f9897-pz4ft
recommendationservice-bc6fc9f9b-9vkjq.default          Kubernetes   SYNCED (23m)     SYNCED (23m)     SYNCED (23m)     SYNCED (23m)     IGNORED      istiod-6cc57f9897-pz4ft
redis-cart-54784d4544-kzs6x.default                    Kubernetes   SYNCED (20m)     SYNCED (20m)     SYNCED (20m)     SYNCED (20m)     IGNORED      istiod-6cc57f9897-pz4ft
shippingservice-5c764656d8-mkn4w.default               Kubernetes   SYNCED (22m)     SYNCED (22m)     SYNCED (22m)     SYNCED (22m)     IGNORED      istiod-6cc57f9897-pz4ft
kirusnovar@cloudshell:~$
```

The output of the istioctl proxy-status command provides a snapshot of the synchronization status between the Istio control plane (istiod) and the Envoy proxies running alongside your services. Here's a breakdown of the columns:

1. **NAME**: The name of the Kubernetes pod. It usually includes the service name and a unique identifier.
2. **CLUSTER**: The Kubernetes cluster the pod belongs to. In this case, all are in the Kubernetes cluster.
3. **CDS** (Cluster Discovery Service): Indicates the synchronization status of cluster information (service endpoints) with the proxy. SYNCED means the proxy has the latest information. The time in parentheses shows how long ago it was synced.
4. **LDS** (Listener Discovery Service): Indicates the synchronization status of listeners (network ports and protocols) with the proxy. SYNCED means the proxy has the latest listener configuration.
5. **EDS** (Endpoint Discovery Service): Indicates the synchronization status of endpoints (IP addresses and ports of service instances) with the proxy. **SYNCED** means the proxy has the latest endpoint information.
6. **RDS** (Route Discovery Service): Indicates the synchronization status of HTTP routes with the proxy. SYNCED means the proxy has the latest routing rules. IGNORED for istio-ingressgateway is expected as it often uses its own configuration mechanisms for external routes.

7. **ECDS** (Envoy Configuration Discovery Service): This is a more generic configuration discovery service. Here it shows IGNORED for all, which might be the default state if not explicitly used.
8. **ISTIOD**: The name of the istiod instance that the proxy is connected to and receiving configuration from.

Weave-Scope Setup

```
kirusnovar@cloudshell:~$ kubectl get pods --all-namespaces | grep weave
weave             weave-scope-app-6c66bdfd58-97f58                  1/1     Running   0             15h
weave             weave-scope-cluster-agent-f544f46bc-cgdkq         1/1     Running   0             15h
kirusnovar@cloudshell:~$
```

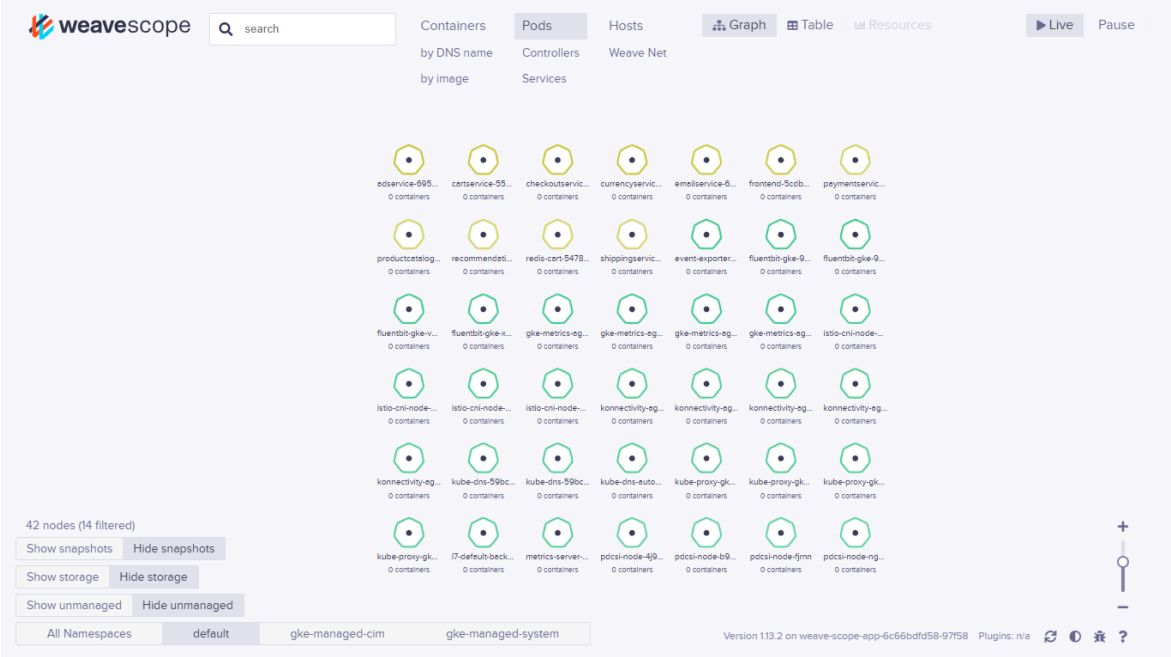
The output of the **kubectl get pods --all-namespaces | grep weave** command shows two pods running in the weave namespace:

weave/weave-scope-app-6c66bdfd58-97f58: This is the Weave Scope application pod. It provides the user interface that you access in your browser to visualize your cluster. It's running with 1 out of 1 container ready and has had 0 restarts, indicating it's stable and has been running for 15 hours.
weave/weave-scope-cluster-agent-f544f46bc-cgdkg: This is the Weave Scope cluster agent pod. It runs on each node in your cluster and collects data about the processes, containers, and network connections. It's also running with 1 out of 1 container ready and has had 0 restarts, running for 15 hours.
Both pods are in the Running state, which means Weave Scope is successfully deployed and its components are operational in your Kubernetes cluster

The weave-scope port number is **20001**
We will use google's internal exposing method

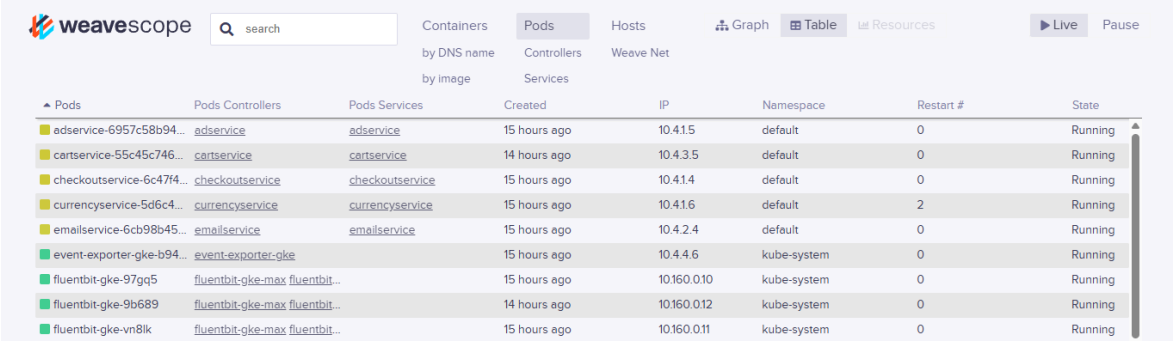
```
kirusnovar@cloudshell:~$ kubectl port-forward -n weave svc/weave-scope-app 4040:80 &
[1] 4617
kirusnovar@cloudshell:~$ Forwarding from 127.0.0.1:4040 -> 4040
Handling connection for 4040
Handling connection for 4040
Handling connection for 4040
```

Weavescope-Frontend Browser view



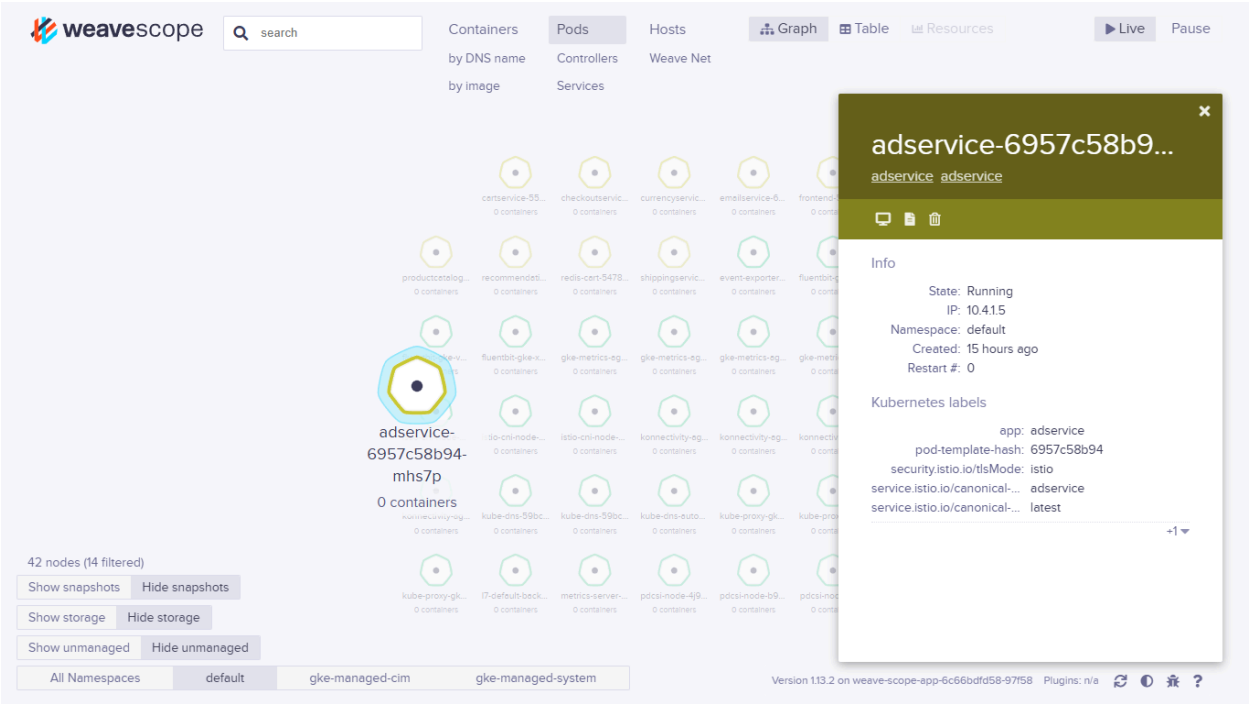
This Weave Scope screenshot visualizes the Kubernetes pods running within the **default namespace** as a grid of green hexagons, indicating a healthy status. Each hexagon is labeled with the specific pod name (e.g., adservice-6957c58b94-mh57p) and the number of containers running inside that pod (e.g., "2 containers"). This container count is a quick way to identify pods where the Istio sidecar proxy has been injected, as it typically adds an extra container alongside the application's primary container. The view allows for namespace filtering (with "default" currently selected), and it indicates a broader context of 42 nodes in the cluster, with the current display focusing on pods relevant to 14 of them. The "Live" indicator suggests this is a real-time view of the cluster's state, allowing for immediate observation of pod health and deployment status.

Table view



This Weave Scope table view lists Kubernetes pods, their controllers (like **Deployments**), **associated services**, **creation time**, **internal IP**, **namespace**, **restart count**, and **current state**. It provides a structured overview of pod details for easy inspection and sorting.

Details about selected pod (adservice)

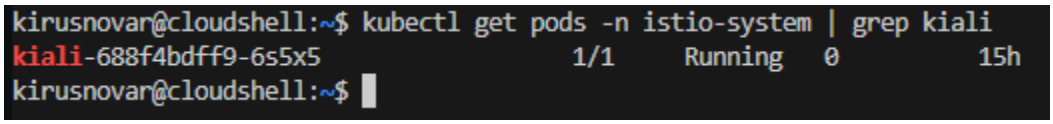


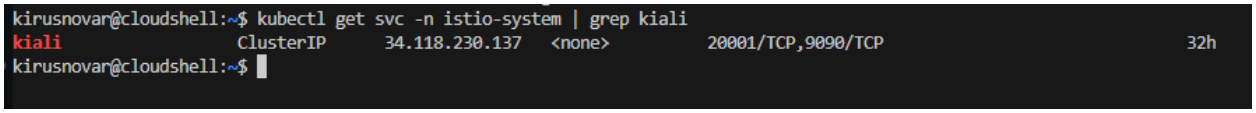
This Weave Scope view shows details about the selected pod: adservice-6957c58b9....

The highlighted pod in the main graph is adservice-6957c58b94-mh57p, and a pop-up window provides specific information about it:

1. **Name:** adservice-6957c58b9...
2. **Status:** Running, indicating the pod is currently active.
3. **IP:** 10.4.1.5, the internal network address of the pod.
4. **Namespace:** default, the Kubernetes namespace where the pod resides.
5. **Created:** 15 hours ago, the time since the pod was created.
6. **Restart count:** 0, meaning the containers within this pod have not restarted.
7. **Containers:** It incorrectly shows "0 containers" directly under the pod name in the graph. However, the presence of the Istio-related labels tell us that the Istio sidecar proxy is injected, meaning there should be at least two containers.

Kiali Setup

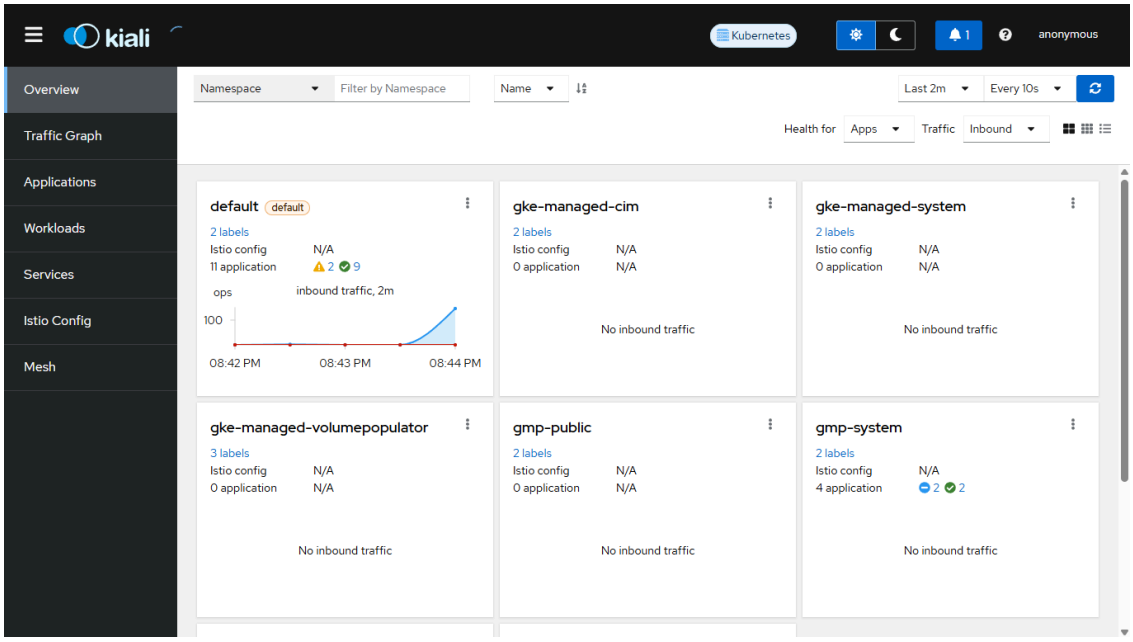




The Kiali setup information shows:
Kiali Pod: The command kubectl get pods -n istio-system | grep kiali indicates one **Kiali pod (kiali-688f4bdff9-6s5x5**) is running in the istio-system namespace and has been running for 15 hours.
Kiali Service: The command kubectl get svc -n istio-system | grep kiali shows the Kiali service named kiali with a ClusterIP of 34.118.238.137 and listening on ports 20001/TCP and 9090/TCP.
Port Number: It explicitly states that "The kiali-service port number is 20001".
Google Cloud Preview Port: An image shows a "Change Preview Port" dialog with the port number 20001 entered, suggesting you are using Google Cloud's internal method to expose the Kiali UI through this port.
Kiali is running as a single pod within the istio-system namespace and is accessible internally within your Kubernetes cluster via the kiali service on port 20001 and 9090. We are configuring Google Cloud's preview functionality to **expose the Kiali UI externally on port 20001**.

Kiali-Frontend Browser view

Here we can see status of different namespaces

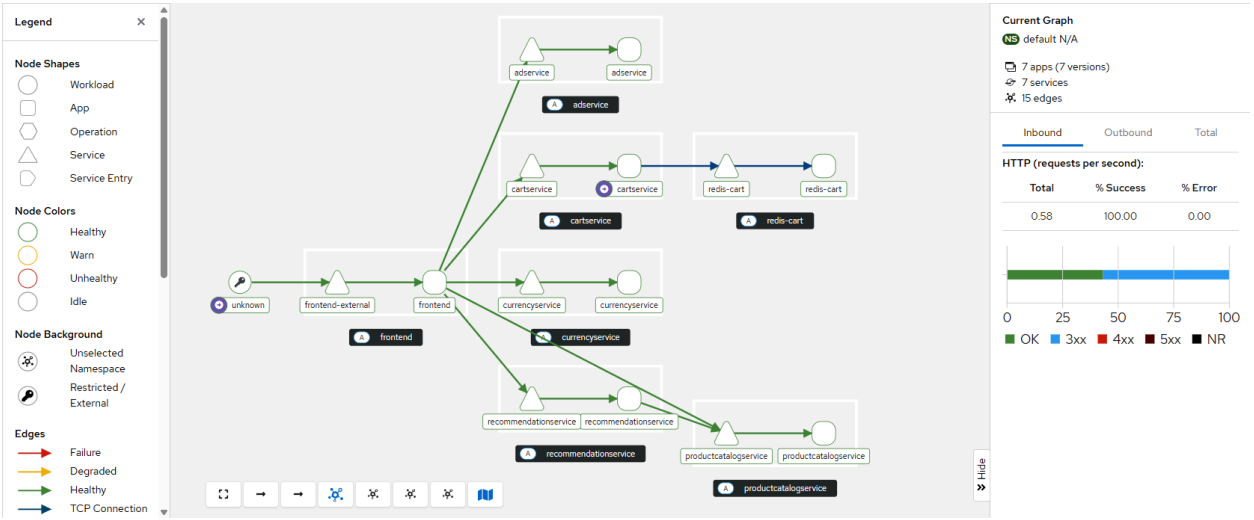


The Kiali Overview displays status cards for Kubernetes namespaces. The default namespace has 11 applications and shows inbound traffic. Other namespaces like **gke-managed-cim**, **gke-managed-system**, **gke-managed-volumepopulator**, and **gmp-public** have no applications or inbound traffic. The **gmp-system** namespace has 4 applications with some activity indicated by colored icons. Each card also shows the number of labels and whether Istio configurations are applied at the namespace level.

Each namespace card also indicates the count of applied labels and the presence of direct Istio configurations (which is "N/A" for all shown namespaces). This view provides a quick, at-a-glance understanding of application distribution and traffic patterns across your Kubernetes environment.

In the Kiali Overview screenshot, the small graph showing inbound traffic for the default namespace has a blue line because blue is the default color Kiali uses to represent the total operations per second (ops) for inbound traffic in its overview graphs

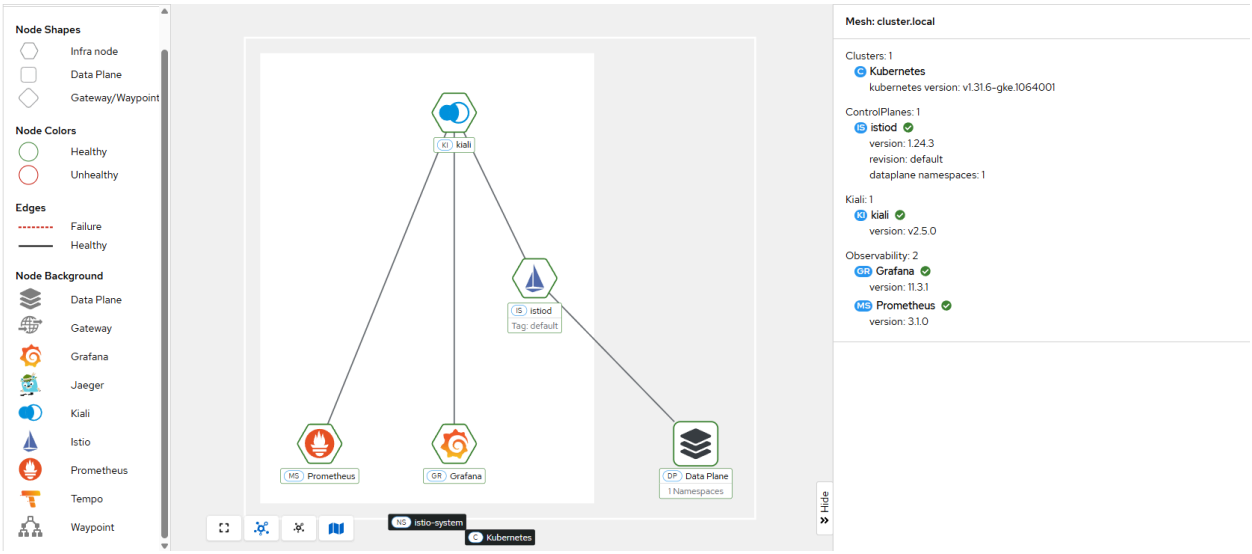
Kiali-Traffic Graph



The Kiali Traffic Graph displays HTTP traffic flow within the default namespace.

- This Graph is Generated in very low load which is 50 users
- (0.58 requests/sec total inbound) just o demonstrate every working network flow in a test environment
- Frontend is the central service receiving traffic from an external source (frontend-external) and routing it to other microservices.
- All services are healthy — shown by green arrows and nodes, with no HTTP errors (100% success rate).
- Redis-cart service communicates via TCP traffic (blue arrow) instead of HTTP,
  because it's a key-value store.
- Microservices architecture is well-structured: frontend delegates specific tasks like ads (adservice), shopping cart (cartservice), pricing (currencyservice), recommendations (recommendationservice), and products (productcatalogservice).

Kiali mesh view



- Kiali, Prometheus, and Grafana are deployed and healthy, providing observability for the cluster.
- Istiod (the Istio control plane component) is up, healthy, and managing the data plane.
- Mesh is named cluster.local, running on Kubernetes version v1.31.6-gke.1064001.
- Prometheus is running version 3.1.0 and Grafana is running version 11.3.1.
- All components are connected with healthy edges (solid black lines), meaning no critical issues in infrastructure communication.

8.Conclusive Outcomes / Deliverables

1. **Successful Deployment:**
    - The Google Cloud Microservices Demo (Online Boutique) was deployed in a Kubernetes environment (both local and GKE-based).
2. **Performance Profiling:**
    - Profiling tools like Prometheus, Grafana, Locust, and Ngrok were used to analyze system performance under various loads.
    - Identified CPU usage trends, bottlenecks, and failure rates, particularly under high user loads (e.g., 500 users).
3. **Real-Time Monitoring & Observability Setup:**
    - Implemented a robust observability stack:
        - Prometheus for metrics collection.
        - Grafana for dashboards (Node Metrics and Cluster & Services dashboards).
        - Kiali, Kubeshark, Weave Scope, and Speedscale for microservices visualization and traffic analysis.
    - Monitored gRPC/HTTP success rates, and visualized inter-service traffic flow and errors.
4. **Load Testing Scenarios:**
    - Created simulations for user traffic and analyzed system responses.
    - Observed performance degradation at scale (e.g., higher error rates and service instability when scaling to 500 users).

5. **Microservices Health and Visualization:**

    - Used Istio service mesh for traffic management and policy enforcement.

    - Visualized pod health and inter-service connections via Weave Scope and Kiali.

    - Ensured all essential microservices were running as expected with sidecar injection for telemetry.

6. **Infrastructure Automation & Deployment via Helm and Terraform:**

    - Helm was used for deploying observability tools.

    - Optional usage of Terraform for infrastructure provisioning was mentioned.

Microservices are ideal for these applications because they allow independent deployment and scaling of individual services, improving resilience and flexibility. This architecture also simplifies observability and fault isolation, making it easier to monitor and troubleshoot complex systems.