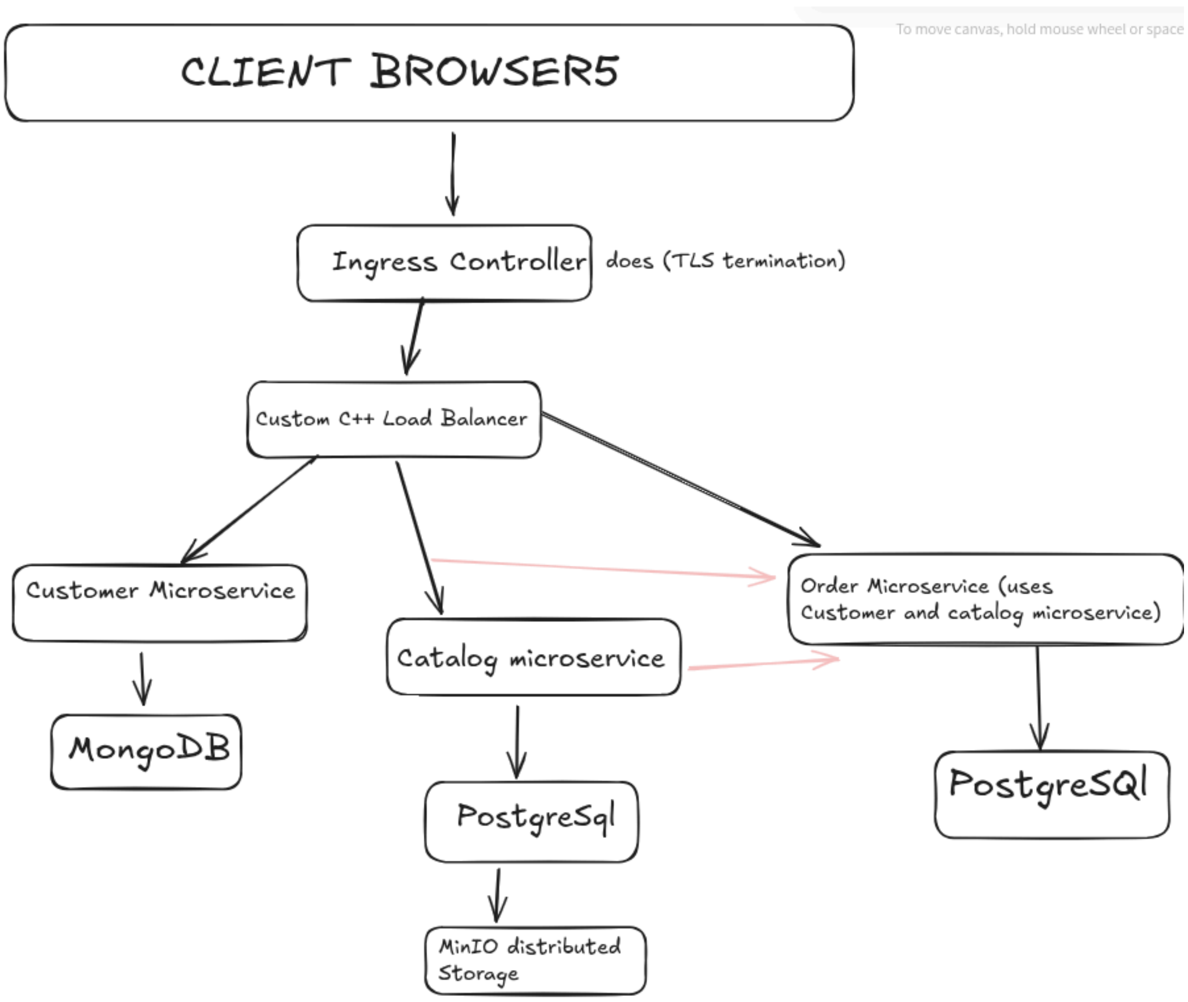
Complete Flow :

1. Cloned the microservices
2. Deployed on kubernetes
3. Replaced the original apache with Custom c++ load balancer.
4. Added ingress controller for TLS termination and front entry point.
5. Added horizontal pod autoscaler (HPA) for autoscaling based on traffic.
6. Connected Databases.
   Customer microservice - Mongo DB
   Catalog microservice - PostgreSQL for catalog items + MinIO for catalog images.
   Order microservice - PostgreSQL for storing order details
7. Added prometheus + Grafana for monitoring purpose
8. Run benchmark test with apache bench (AB).

Features :

- Deployed microservices on kubernetes.
- Started with nginx load balancer and tested all by extending nginx load balancer.
- Created Custom Load Balancer in C++.
- Implemented different routing strategies
  Round Robin
  Ip hash
  Least Connections
- Health checks
- TLS termination (basic SSL support).
- Traffic distribution across Kubernetes services.
- Content-aware routing (e.g., based on headers, URL paths).
- Sticky sessions (session affinity).
- Intelligent scaling (auto-adjusting replicas based on LB metrics).
- Observability hooks (export metrics to Prometheus/Grafana).
- Connected to PostgreSQL , MongoDB , MinIO
- Benchmark testing.

Final Architecture :



1. Project Setup and Initial Deployment

1.1 **Cloning the Source Repository**

The project began by cloning the reference microservice application from its public GitHub repository. This repository provides a well-defined set of microservices (`catalog`, `customer`, and `order`) that serve as the backend for our load balancing experiment.

The following command was used to clone the repository:git clone

```
https://github.com/ewolff/microservice-kubernetes.git
```

1.2 Kubernetes Cluster Initialization

To create a local, single-node Kubernetes environment for development and testing, **Minikube** was used.
The cluster was initialized and started with the following command:

```
minikube start
```

This command sets up the control plane and a worker node, making the Kubernetes API available for deploying applications.

1.3 Deployment of Microservices and Reverse Proxy

Once the Kubernetes cluster was running, the core application components were deployed. The application consists of three distinct microservices:

- **Customer Microservice:** Manages customer data.
- **Catalog Microservice:** Manages product catalog information.
- **Order Microservice:** Manages orders and communicates with both the **Customer** and **Catalog** services to fulfill its functions.

Initially, the project's default **Apache** server was deployed to act as a basic reverse proxy, directing traffic to the appropriate backend service.

The deployment was executed by applying the Kubernetes manifest files provided in the repository:

```
cd microservice-kubernetes/
kubectl apply -f
microservice-kubernetes-demo/microservices.yaml
```

After running this command, all three microservices and the Apache reverse proxy were running as pods within the Minikube cluster, ready for the custom load balancer integration.

---

2. Custom Load Balancer Implementation & Integration

2.1 Strategy Design / Load balancing

The custom c++ load balancer was used to configure traffic management.

Different load balancer strategies were used for different microservices.

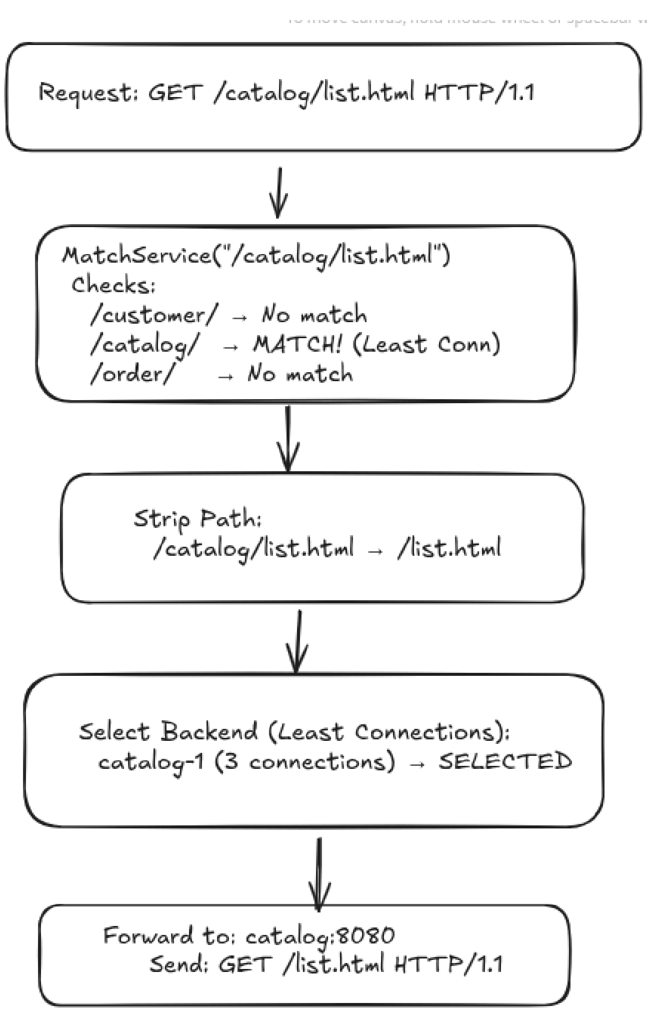
- **Upstream for Catalog (least_conn):** (Least Connections) strategy **Upstream for Customer (ip_hash)** The `ip_hash` strategy This creates "sticky sessions," providing a consistent user experience.
- **Upstream for Order (round_robin):** default `round_robin` strategy

```
// CREATE LOAD BALANCER

lb = make_unique<LoadBalancer>(80, 8081);



// CONFIGURE PATH-BASED ROUTING WITH DIFFERENT STRATEGIES



// Customer Service - IP Hash (Sticky Sessions)

lb->addService("/customer/", LoadBalancingAlgorithm::IP_HASH);

lb->addBackendToService("/customer/", "customer-1",
"customer", 8080, 3, 30);



// Catalog Service - Least Connections (Optimal Load
Distribution)

lb->addService("/catalog/",
LoadBalancingAlgorithm::LEAST_CONNECTIONS);

lb->addBackendToService("/catalog/", "catalog-1", "catalog",
8080, 3, 30);



// Order Service - Round Robin (Simple Rotation)

lb->addService("/order/",
LoadBalancingAlgorithm::ROUND_ROBIN);
```

```
lb->addBackendToService("/order/", "order-1", "order", 8080,
3, 30);
```

Path matching Logic

```
   // Find longest matching prefix

shared_ptr<ServiceConfig> LoadBalancer::matchService(const string& path) {

   shared_ptr<ServiceConfig> matched = nullptr;

   size_t maxLen = 0;

   // Loop through all configured services

   for (auto& [servicePath, config] : services) {

       // Check if request path starts with service path

       if (path.find(servicePath) == 0 && servicePath.length() > maxLen) {

           matched = config;  // Match found!

           maxLen = servicePath.length();  // Track longest match

       }
}
return matched;

}
```


Request: GET /catalog/list.html HTTP/1.1
MatchService("/catalog/list.html")
Checks:
/customer/ → No match
/catalog/ → MATCH! (Least Conn)
/order/ → No match
Strip Path:
/catalog/list.html → /list.html
Select Backend (Least Connections):
catalog-1 (3 connections) → SELECTED
Forward to: catalog:8080
Send: GET /list.html HTTP/1.1

When we send a request from the front end to the customer microservice
First it goes to the custom c++ load balancer and it finds the path using the match service function

Ex GET /catalog/list.html

It matches with /catalog/ , the c++ redirects the request to catalog microservice and strips the url to just GET /list.html.

The load balancer selects one of the pod to distribute load based on least connections strategy .

The microservice returns the list.html with status 200 OK.

2.2 Integrating with Kubernetes Ingress

To expose the application to the outside world and manage traffic flow into the cluster, an **Nginx Ingress Controller** was deployed. The Ingress acts as the primary entry point.

An Ingress resource was then configured to capture all incoming traffic (`/`) and route it exclusively to our custom C++ load balancer service.

Microservice ingress used is below:

```
spec:

 tls:

 - hosts:

   - microservices.local
```

```yaml
   secretName: microservice-tls
 ingressClassName: nginx
 rules:
 - host: microservices.local
   http:
     paths:
     - path: /
       pathType: Prefix
       backend:
         service:
           name: cpp-loadbalancer-service
           port:
             number: 80
```

2.2 Health Checks and Failover

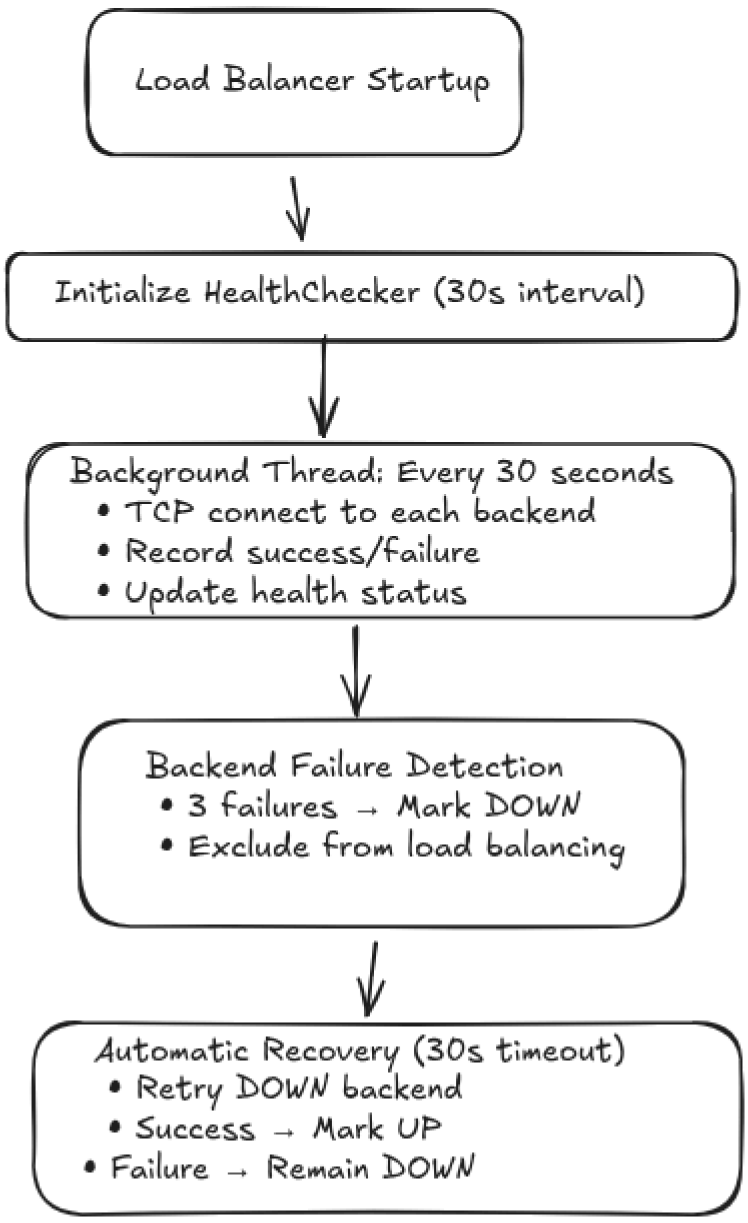
The custom C++ load balancer incorporates a robust health monitoring system that automatically detects and recovers from failures.

Overview

The health check system operates on two distinct levels:

1. **Backend Health Monitoring**: Continuously monitors the availability of microservices.
2. **Load Balancer Liveness**: Utilizes a Kubernetes health endpoint for monitoring the load balancer's pod.

Backend Health Check Mechanism TCP-Based Health Check


Load Balancer Startup
Initialize HealthChecker (30s interval)
Background Thread: Every 30 seconds
• TCP connect to each backend
• Record success/failure
• Update health status
Backend Failure Detection
• 3 failures → Mark DOWN
• Exclude from load balancing
Automatic Recovery (30s timeout)
• Retry DOWN backend
• Success → Mark UP
• Failure → Remain DOWN

3. TLS/HTTPS Security Setup

1. **Download and Install mkcert:**
   - The mkcert tool was installed to create a local certificate authority.
   - `brew install mkcert`
2. **Self-Signed Certificate Created:**
   - A TLS certificate was generated for the `microservices.local` domain.
   - `mkcert microservices.local`
3. **Local CA Installed:**
   - `mkcert`'s root certificate was added to the system's trusted certificate store.
   - `mkcert -install`
4. **Kubernetes Secret Created:**

    - The generated TLS certificate and key were stored as a Kubernetes secret.
    - `kubectl create secret tls microservices-tls --key microservices.local-key.pem --cert microservices.local.pem`
5. **Ingress Configured for HTTPS:**
    - The Ingress resource was updated to utilize the TLS certificate for secure connections.
6. **SSL/TLS Termination Enabled:**
    - The Ingress controller was ensured to be configured to terminate HTTPS connections.
7. **HTTPS Access Tested:**
    - Secure HTTPS requests to `[https://microservices.local:8443](https://microservices.local:8443)` were verified to be successful.

4. Adding Autoscaling

Overview

The Kubernetes Horizontal Pod Autoscaler (HPA) provides automatic scaling capabilities for microservices by adjusting the number of pod replicas based on real-time CPU and memory utilization. This ensures optimal resource allocation and application performance during varying traffic loads.

Configuration

1. Metrics Server Deployment:

The Metrics Server is a prerequisite for HPA and is deployed within the `kube-system` namespace to collect resource metrics from pods.

2. HPA Settings (per microservice):

Each microservice's HPA is configured with the following parameters:

- **`minReplicas`**: 1 (Ensures at least one pod is always running.)
- **`maxReplicas`**: 3 (Sets the upper limit for the number of pods.)
- **`CPU Target`**: 75% (Triggers a scale-up when average CPU utilization across pods exceeds 75%.)
- **`Memory Target`**: 180% (Triggers a scale-up when average memory utilization across pods exceeds 180%.)

3. Configured HPAs:

Three distinct HPAs have been configured for the following microservices, each integrated with a C++ Load Balancer using specific algorithms:

1. `catalog-hpa`: Manages the **Catalog service** (utilizes the Least Connections algorithm for load balancing).
2. `customer-hpa`: Manages the **Customer service** (utilizes the IP Hash algorithm for load balancing).
3. `order-hpa`: Manages the **Order service** (utilizes the Round Robin algorithm for load balancing).

Scaling Behavior

Scale-Up Process:

- **Trigger**: A scale-up event is initiated when the average CPU utilization across the pods exceeds 75% **OR** the average memory utilization exceeds 180%.
- **Pod Creation**: New pods are typically created and ready within 1-2 minutes of the trigger.
- **Load Balancer Integration**: The C++ Load Balancer automatically discovers and registers these newly created pods, immediately routing traffic to them.

Scale-Down Process:

- **Trigger**: A scale-down event is triggered after a 5-minute stabilization period, during which resource utilization remains below the specified targets.
- **Graceful Termination**: Excess pods are gracefully terminated to prevent disruption of ongoing connections.
- **Minimum Replicas**: A minimum of 1 replica is always maintained for each microservice, even during scale-down events.

Resource Limits

Each microservice pod is configured with the following resource requests and limits:

- `requests`:
  - `cpu`: 200m
  - `memory`: 512Mi
- `limits`:
  - `cpu`: 500m
  - `memory`: 1Gi

How It Works with the C++ Load Balancer

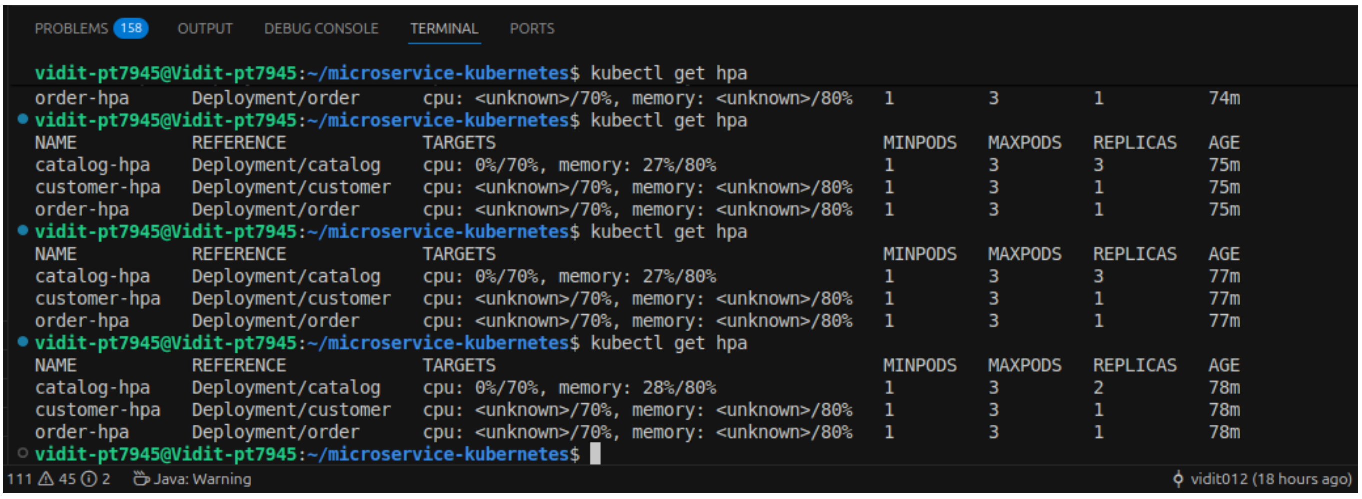
The HPA and C++ Load Balancer work in tandem to provide seamless autoscaling:

1. **Traffic Increase**: As traffic to a microservice increases, the CPU and/or memory utilization of its pods rises.
2. **HPA Detection**: The HPA continuously monitors these metrics and detects when

they exceed the configured thresholds.

3. **Pod Creation**: The HPA automatically creates new pods to handle the increased load.
4. **Service Discovery Updates**: Kubernetes' service discovery mechanism updates with the information about the new pods.
5. **Load Balancer Integration**: The C++ Load Balancer, through its discovery mechanism, identifies the newly available pods and adds them to its pool of healthy instances.
6. **Traffic Distribution**: The Load Balancer then intelligently distributes incoming traffic across all available pods, including the newly scaled-up ones, ensuring efficient resource utilization and responsiveness.

- **Run Autoscaling Demo**: To simulate and observe the autoscaling behavior, execute the provided demo script autoscaling demo.sh



```
vidit-pt7945@Vidit-pt7945:~/microservice-kubernetes$ kubectl get hpa
order-hpa      Deployment/order      cpu: <unknown>/70%, memory: <unknown>/80%   1         3         1          74m
vidit-pt7945@Vidit-pt7945:~/microservice-kubernetes$ kubectl get hpa
NAME           REFERENCE             TARGETS                                     MINPODS   MAXPODS   REPLICAS   AGE
catalog-hpa    Deployment/catalog    cpu: 0%/70%, memory: 27%/80%                1         3         3          75m
customer-hpa   Deployment/customer   cpu: <unknown>/70%, memory: <unknown>/80%   1         3         1          75m
order-hpa      Deployment/order      cpu: <unknown>/70%, memory: <unknown>/80%   1         3         1          75m
vidit-pt7945@Vidit-pt7945:~/microservice-kubernetes$ kubectl get hpa
NAME           REFERENCE             TARGETS                                     MINPODS   MAXPODS   REPLICAS   AGE
catalog-hpa    Deployment/catalog    cpu: 0%/70%, memory: 27%/80%                1         3         3          77m
customer-hpa   Deployment/customer   cpu: <unknown>/70%, memory: <unknown>/80%   1         3         1          77m
order-hpa      Deployment/order      cpu: <unknown>/70%, memory: <unknown>/80%   1         3         1          77m
vidit-pt7945@Vidit-pt7945:~/microservice-kubernetes$ kubectl get hpa
NAME           REFERENCE             TARGETS                                     MINPODS   MAXPODS   REPLICAS   AGE
catalog-hpa    Deployment/catalog    cpu: 0%/70%, memory: 28%/80%                1         3         2          78m
customer-hpa   Deployment/customer   cpu: <unknown>/70%, memory: <unknown>/80%   1         3         1          78m
order-hpa      Deployment/order      cpu: <unknown>/70%, memory: <unknown>/80%   1         3         1          78m
vidit-pt7945@Vidit-pt7945:~/microservice-kubernetes$
```

In the above image , **SCALE DOWN** behaviour is tested. The Catalog was already scaled to 3 replicas .
As the load was decreased and memory usage hit 27/80% , it automatically scaled down to 2 and then finally to 1.

5. Connecting Databases

- **PostgreSQL (postgres:13)**
  - **Used by:** Catalog and Order services.
  - **Deployment:** A single replica with 1Gi of persistent storage.
  - **Service:** Exposed via a ClusterIP on port 5432.

- **MongoDB (mongo:4.4)**
  - **Used by:** Customer service.
  - **Deployment:** A single replica with 1Gi of persistent storage.**Service:** Exposed via a ClusterIP on port 27017.

Customer microservice - MongoDB to store customer information.

Has a functionality of addd customer and remove customer
Add Customer pushes entry into the database
Delete customer removes the respective entry from the database.

Catalog Microservice : PostgreSQL is used to store the catalog items.
MinIo is used to store the images of catalog items.
Functionality
Add Item option pushes entry into the database.
Remove item removes the respective entry from the database,
Upload Images option renames and uploads it to MinIO..

Database Implementation

Configuration

Database credentials are securely stored within Kubernetes ConfigMaps:

- **postgres-config:**

```
spring.datasource.url=jdbc:postgresql://postgres:5432/microservicesdb
spring.datasource.username=admin
spring.datasource.password=admin123
spring.jpa.hibernate.ddl-auto=update
```

- **mongodb-config:**

```
spring.data.mongodb.host=mongodb
spring.data.mongodb.port=27017
spring.data.mongodb.database=microservicesdb
spring.data.mongodb.username=admin
spring.data.mongodb.password=admin123
```

Connection Mechanism for postgres to catalog microservice

- Kubernetes DNS resolves the hostname `postgres` to its ClusterIP (e.g.,

10.103.138.16).
- Spring Boot initializes a JDBC connection pool.
- Hibernate automatically creates the `item` table on application startup.
- Spring Data JPA manages all subsequent database operations.

Connection Mechanism for MongoDB with Customer microservice

- Kubernetes DNS resolves the hostname `mongodb` to its ClusterIP (e.g., 10.99.100.2).
- Spring Data MongoDB establishes a connection to the database.
- Collections are automatically created upon the insertion of the first document.
- Document-based storage is utilized for customer data.

CRUD Operations Flow

This section illustrates the flow of Create, Read, and Delete operations.

Adding Data (CREATE)

Example: Add a new item to the Catalog

```
1. User clicks "Add Item" on web form
   POST https://microservices.local:8443/catalog/

2. Ingress → C++ Load Balancer → Catalog Service

3. Catalog Service (Java/Spring Boot):
   @PostMapping("/")
   public String addItem(@ModelAttribute Item item) {
       itemRepository.save(item);   // JPA handles SQL
   }

4. JPA/Hibernate generates SQL:
   INSERT INTO item (name, price) VALUES ('Laptop',
55000);

5. PostgreSQL executes query:
   - Auto-generates ID (using SERIAL sequence)
   - Returns new item with ID

6. Response sent back: Item created with ID
```

Example: Delete an item

```
1. User clicks "Delete" button

  POST
https://microservices.local:8443/catalog/9?_method=DELETE

2. Catalog Service:

  @DeleteMapping("/{id}")

  public String deleteItem(@PathVariable Long id) {

      itemRepository.deleteById(id);

  }

3. Hibernate generates SQL:

  DELETE FROM item WHERE id=9;

4. PostgreSQL removes the row

5. ID sequence continues (next item will be ID 10)

  - Deleted IDs are NOT reused
```

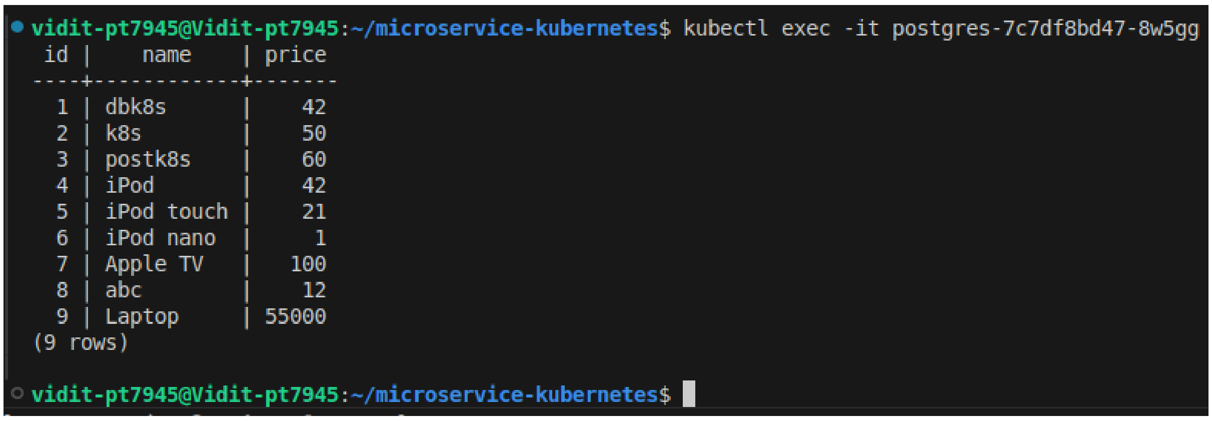
Data Persistence

- **PersistentVolume Mounting:** A PersistentVolume is mounted to each database pod.
- **Data Storage:**
  - PostgreSQL writes its data to `/var/lib/postgresql/data` on the PersistentVolume.
  - MongoDB writes its data to `/data/db` on the PersistentVolume.
- **Pod Restart/Crash:** If a database pod crashes or restarts:
  - A new pod is created and mounts the same PersistentVolume.
  - All previously stored data remains intact.
  - No data loss occurs.

Live Examples from Microservice:

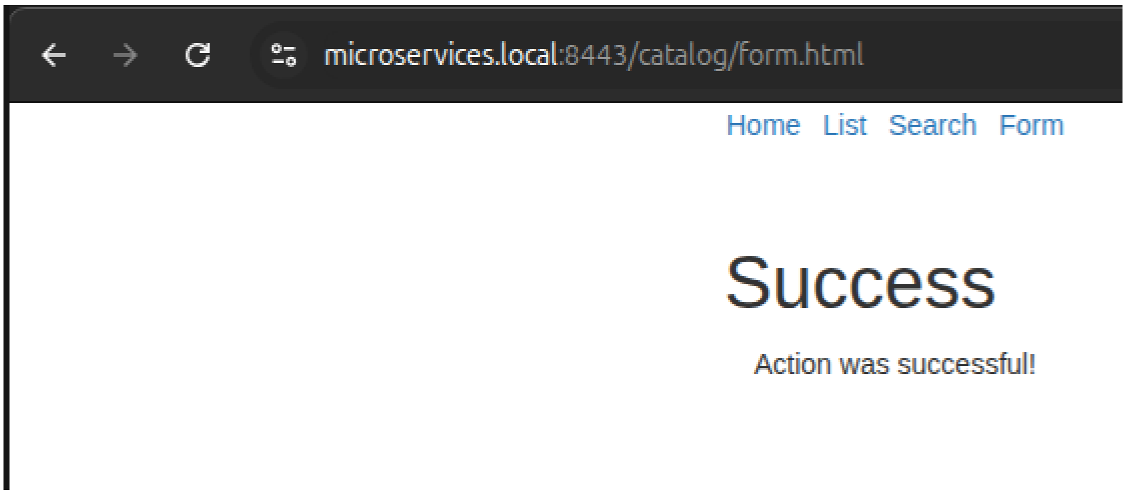
<u>Adding a data to the catalog table:</u>

Before adding the data , the table looks below:



```
vidit-pt7945@Vidit-pt7945:~/microservice-kubernetes$ kubectl exec -it postgres-7c7df8bd47-8w5gg
 id |    name     | price
----+-------------+-------
  1 | dbk8s       |    42
  2 | k8s         |    50
  3 | postk8s     |    60
  4 | iPod        |    42
  5 | iPod touch  |    21
  6 | iPod nano   |     1
  7 | Apple TV    |   100
  8 | abc         |    12
  9 | Laptop      | 55000
(9 rows)

vidit-pt7945@Vidit-pt7945:~/microservice-kubernetes$
```

After Adding Data through the microservice

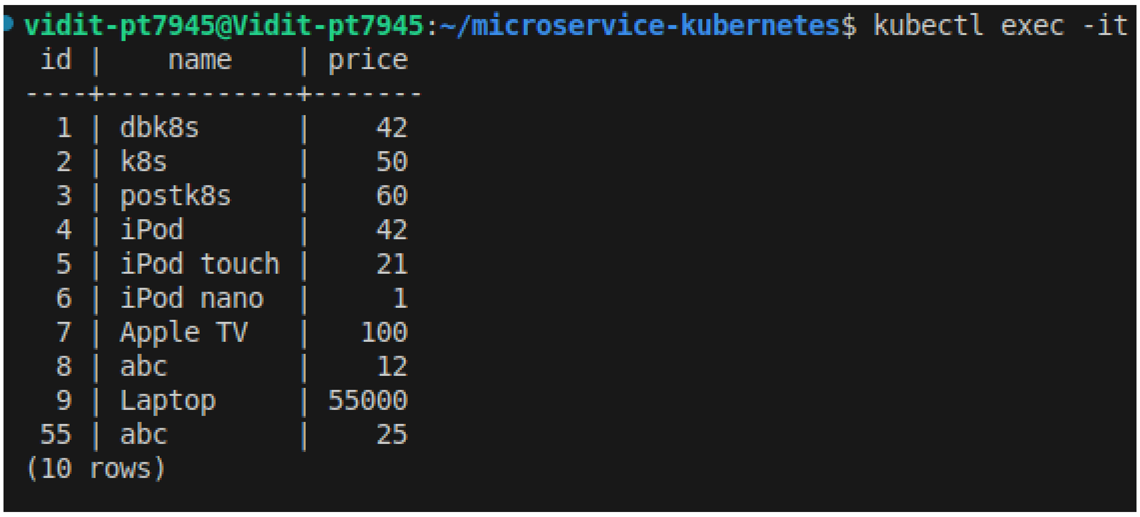


microservices.local:8443/catalog/form.html

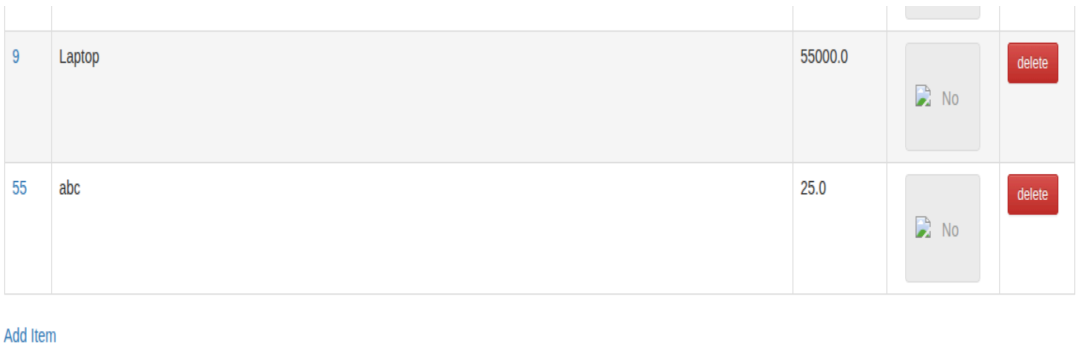
Home List Search Form

Success

Action was successful!



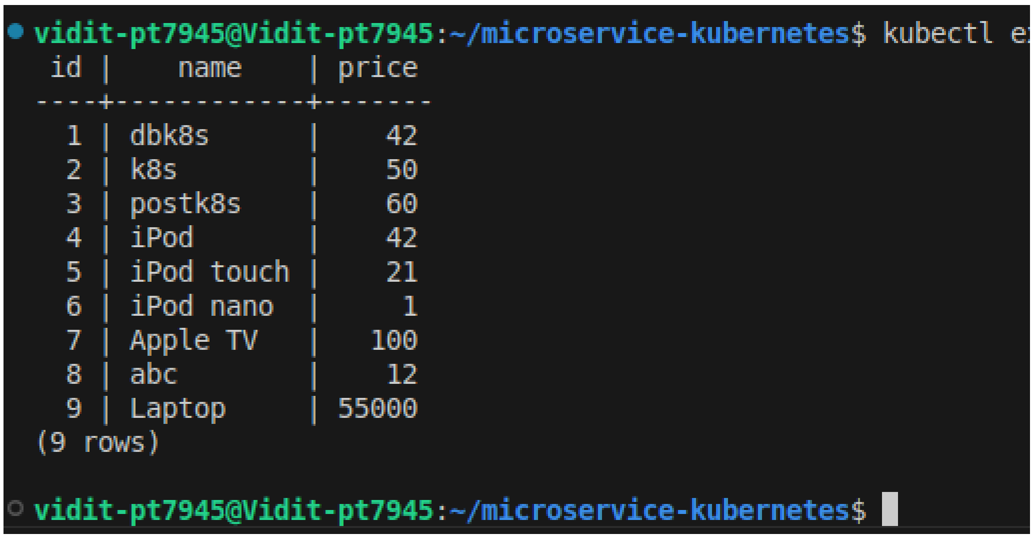
```
vidit-pt7945@Vidit-pt7945:~/microservice-kubernetes$ kubectl exec -it
 id |    name     | price
----+-------------+-------
  1 | dbk8s       |    42
  2 | k8s         |    50
  3 | postk8s     |    60
  4 | iPod        |    42
  5 | iPod touch  |    21
  6 | iPod nano   |     1
  7 | Apple TV    |   100
  8 | abc         |    12
  9 | Laptop      | 55000
 55 | abc         |    25
(10 rows)
```



9	Laptop	55000.0	No	delete
55	abc	25.0	No	delete

Add Item

After pressing the delete button



```
vidit-pt7945@Vidit-pt7945:~/microservice-kubernetes$ kubectl e
 id |    name     | price
----+-------------+-------
  1 | dbk8s       |    42
  2 | k8s         |    50
  3 | postk8s     |    60
  4 | iPod        |    42
  5 | iPod touch  |    21
  6 | iPod nano   |     1
  7 | Apple TV    |   100
  8 | abc         |    12
  9 | Laptop      | 55000
(9 rows)

vidit-pt7945@Vidit-pt7945:~/microservice-kubernetes$
```

6. MinIO Connection

1. Deployment Configuration

MinIO is deployed as a single pod with persistent storage, utilizing the following specifications:

- **Image:** `minio/minio:RELEASE.2023-09-30T07-02-29Z`
- **Credentials:** Stored in a ConfigMap, with `MINIO_ROOT_USER: minioadmin` and `MINIO_ROOT_PASSWORD: minioadmin`.
- **Persistent Storage:** A 10Gi PVC (`minio-pvc`) is mounted at `/data`.
- **Resources:**
  - **Memory:** 256Mi request, 512Mi limit
  - **CPU:** 250m request, 500m limit
- **Ports:**
  - `9000` (API)
  - `9001` (Console)

2. Service Connection

MinIO is exposed via a Kubernetes ClusterIP Service:

- **Service Name:** `minio`
- **Internal URL:** `http://minio:9000` (accessible within the cluster via Kubernetes DNS)
- **API Port:** `9000` (for S3-compatible operations)
- **Console Port:** `9001` (for web UI access)

3. Catalog Service Integration

The Catalog microservice connects to MinIO using the following properties:

- `minio.url=http://minio:9000`
- `minio.access-key=minioadmin`
- `minio.secret-key=minioadmin`
- `minio.bucket-name=catalog-images`

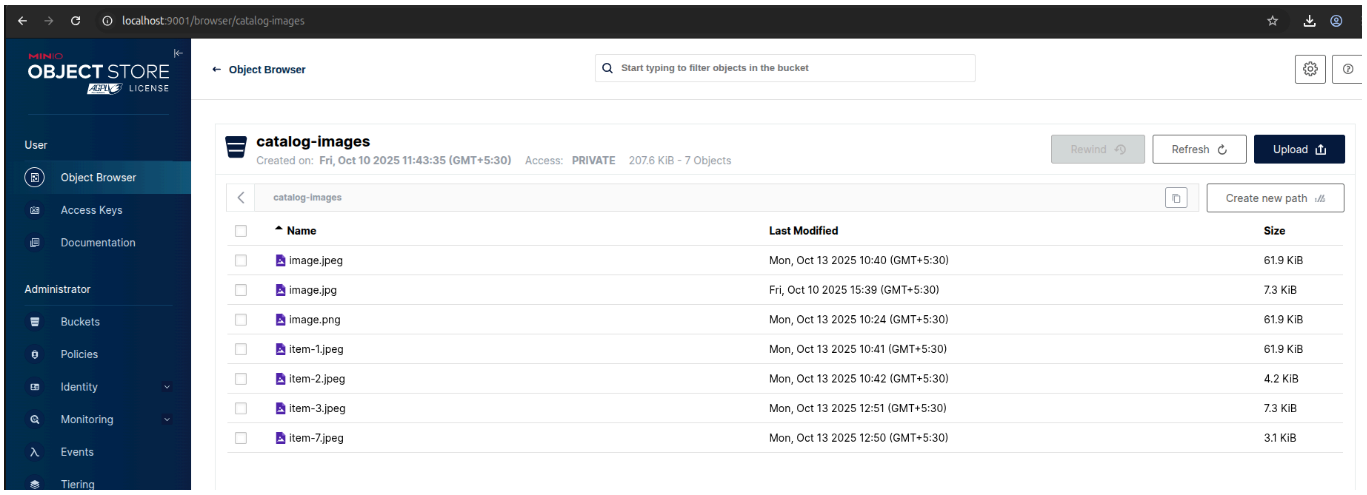
4. CRUD Operations Flow

The following outlines the Create, Read, Update, and Delete (CRUD) operations for images within MinIO:

- **Create:** The Catalog service uploads product images to the `catalog-images` bucket via the MinIO Java client.
- **Read:** Image URLs are generated using MinIO presigned URLs or public bucket access.
- **Update:** The old image is deleted, and a new image is uploaded with the same object key.
- **Delete:** The object is removed from the bucket when the corresponding product is deleted.

5. Access Methods

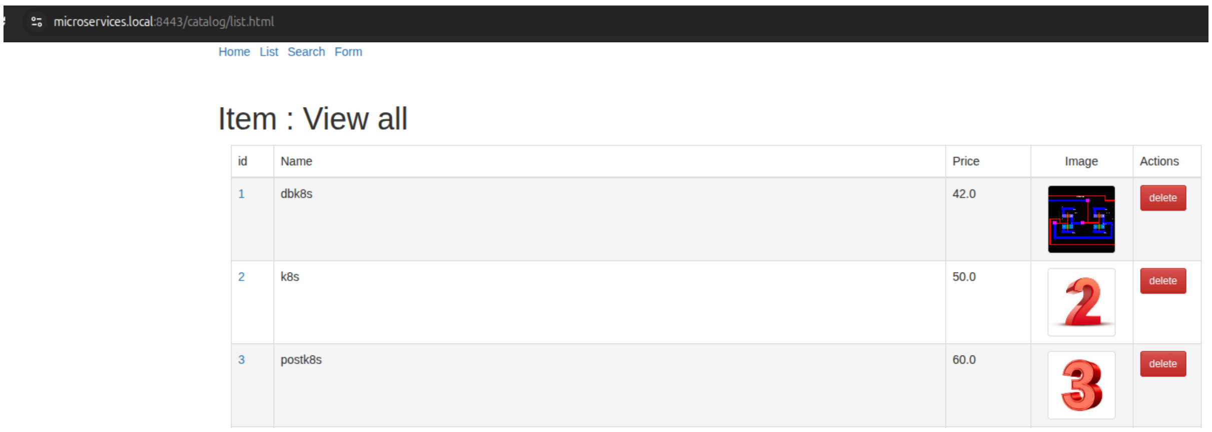
- **API Access:** Direct HTTP calls to `minio:9000` from within the cluster.
- **Console Access:** Port-forwarding for browser access using `kubectl port-forward svc/minio 9001:9001`.



catalog-images
Created on: Fri, Oct 10 2025 11:43:35 (GMT+5:30) Access: PRIVATE 207.6 KiB - 7 Objects

catalog-images

Name	Last Modified	Size
image.jpeg	Mon, Oct 13 2025 10:40 (GMT+5:30)	61.9 KiB
image.jpg	Fri, Oct 10 2025 15:39 (GMT+5:30)	7.3 KiB
image.png	Mon, Oct 13 2025 10:24 (GMT+5:30)	61.9 KiB
item-1.jpeg	Mon, Oct 13 2025 10:41 (GMT+5:30)	61.9 KiB
item-2.jpeg	Mon, Oct 13 2025 10:42 (GMT+5:30)	4.2 KiB
item-3.jpeg	Mon, Oct 13 2025 12:51 (GMT+5:30)	7.3 KiB
item-7.jpeg	Mon, Oct 13 2025 12:50 (GMT+5:30)	3.1 KiB



Item : View all

id	Name	Price	Image	Actions
1	dbk8s	42.0		delete
2	k8s	50.0		delete
3	postk8s	60.0		delete

In the above image , the images are being retrieved from the MinIO Catalog-images bucket.

7. Adding Prometheus and Grafana

Working:
1. Microservices → Expose /actuator/prometheus endpoints
↓
2. Prometheus-Server → Scrapes all targets every 15s
↓
3. Stores metrics in time-series database
↓
4. Grafana → Queries Prometheus via HTTP
↓
5. Displays dashboards in browser

Installation Commands

helm repo add prometheus-community https://promSetting Up Monitoring with Prometheus and Grafana

1. **Add Prometheus Community Helm repository:**
   helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
2. **Add Grafana Helm repository:**
   helm repo add grafana https://grafana.github.io/helm-charts
3. **Update Helm repositories:**
   helm repo update
4. **Install Prometheus:**
   helm install prometheus prometheus-community/prometheus

Grafana Installation

1. **Install Grafana using the official Helm chart:**
   helm install grafana grafana/grafana

⚙️ Prometheus Configuration1. Service Discovery

Prometheus automatically discovers Kubernetes services and pods using annotations. Each microservice (Catalog, Customer, Order, and Load Balancer) exposes metrics via Spring Boot Actuator.

Annotations for Metric Exposure:

Annotation	Value	Description
`prometheus.io/scrape`	`"true"`	Enables Prometheus to scrape metrics from this pod.

`prometheus.io/port`	`"8080"`	Specifies the port on which metrics are exposed.
`prometheus.io/path`	`"/actuator/prometheus"`	Specifies the path to the metrics endpoint.

Scraping Configuration:

- **Scrape Interval:** 15 seconds
- **Scrape Job:** `kubernetes-pods` (auto-discovers annotated pods)

2. Scrape Targets

The following are the specific target URLs from which Prometheus collects metrics:

Target URL	Description
`http://catalog:8080/actuator/prometheus`	Catalog microservice metrics
`http://customer:8080/actuator/prometheus`	Customer microservice metrics
`http://order:8080/actuator/prometheus`	Order microservice metrics

3. Metrics Exposed

Each microservice exposes Spring Boot Actuator metrics under `/actuator/prometheus`. Key metrics collected include:

Metric Name	Description
`jvm_memory_used_bytes`	JVM heap and non-heap memory usage
`process_cpu_usage`	CPU usage of the service
`http_server_requests_seconds_count`	Count of HTTP requests served

4. Data Source Configuration

Grafana is configured to use Prometheus as its data source:

Parameter	Value
Data Source Type	Prometheus

URL	`http://prometheus-server:80`
Access Mode	Server (Grafana backend connects directly)
Scrape Interval	`15s` (matches Prometheus configuration)

5. Dashboard Import

Preconfigured JSON dashboards were imported for enhanced visualization:

Dashboard Name	Purpose
`grafana-microservices-metrics-dashboard.json`	Tracks application-level performance metrics
`grafana-pod-metrics-dashboard.json`	Displays pod-level resource utilization

🌐 Access MethodsAccessing Prometheus UI

To access the Prometheus web interface:
kubectl port-forward service/prometheus-server 9090:80
Access URL: http://localhost:9090Accessing Grafana UI

To access the Grafana web interface:
kubectl port-forward service/grafana 3000:80
Access URL: http://localhost:3000
Key Components Overview

The monitoring infrastructure relies on several key components:

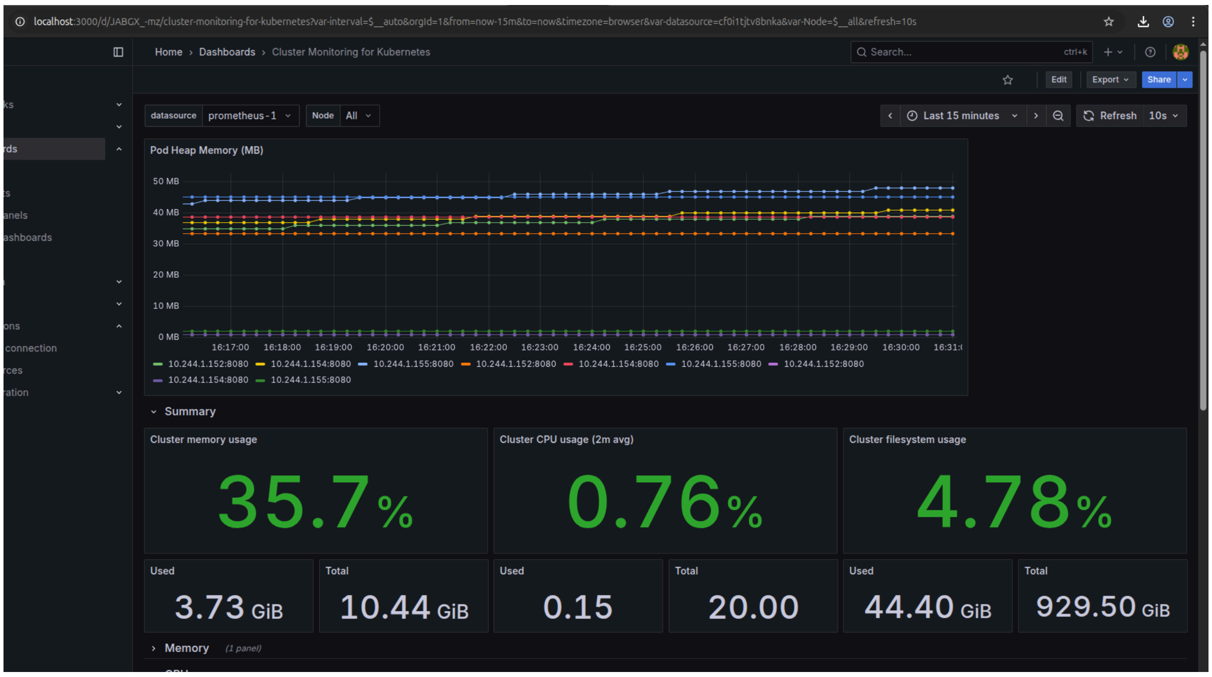
Component	Kubernetes Service	Port	Function
Prometheus Server	`prometheus-server`	3000	Collects and stores metrics
Grafana	`grafana`	9090	Visualizes dashboards

🔗 Grafana-Prometheus Integration

```
These are examples of PromQL queries that can be used in
Grafana dashboards:CPU Usage per Pod
```

```
process_cpu_usage{job="kubernetes-pods"} * 100
Memory Usage per Service (in MB)
sum(jvm_memory_used_bytes{area="heap"}) by
(kubernetes_pod_name) / 1024 / 1024
HTTP Request Rate (per minute)
sum(rate(http_server_requests_seconds_count[1m])) by (uri)
Outcome
```

Dashboard Screenshots:

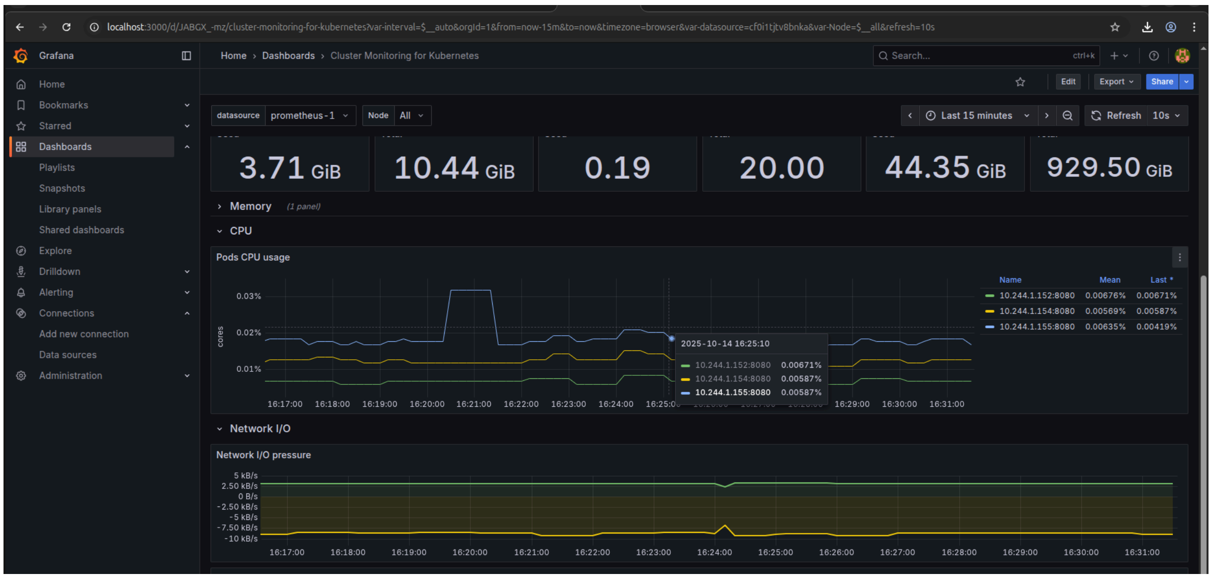


localhost:3000/d/JABGX_-mz/cluster-monitoring-for-kubernetes?var-interval=$__auto&orgId=1&from=now-15m&to=now&timezone=browser&var-datasource=cf0i1tjtv8bnka&var-Node=$__all&refresh=10s
Home › Dashboards › Cluster Monitoring for Kubernetes
Edit
Export
Share
Last 15 minutes
Refresh 10s
Pod Heap Memory (MB)
50 MB
40 MB
30 MB
20 MB
10 MB
0 MB
10.244.1.152:8080
10.244.1.154:8080
10.244.1.155:8080
Summary
Memory (1 panel)

datasource: prometheus-1 · Node: All

Cluster memory usage	Cluster CPU usage (2m avg)	Cluster filesystem usage
35.7%	0.76%	4.78%

Used	Total	Used	Total	Used	Total
3.73 GiB	10.44 GiB	0.15	20.00	44.40 GiB	929.50 GiB



datasource: prometheus-1 · Node: All

3.71 GiB	10.44 GiB	0.19	20.00	44.35 GiB	929.50 GiB

Name	Mean	Last *
10.244.1.152:8080	0.00676%	0.00671%
10.244.1.154:8080	0.00569%	0.00587%
10.244.1.155:8080	0.00635%	0.00419%

2025-10-14 16:25:10
10.244.1.152:8080 0.00671%
10.244.1.154:8080 0.00587%
10.244.1.155:8080 0.00587%

8. Benchmarking test with Apache testbench

benchmarking.Tool Used: Apache Bench (ab)

Apache Bench is the primary load testing tool used for performance benchmarking.

Why Apache Bench?

- Industry-standard HTTP load testing tool
- Simple to use and interpret
- Built-in statistics (latency percentiles, throughput)
- Supports HTTPS, keep-alive connections
- Perfect for REST API testing

Benchmark Script: `benchmark.sh`

What it does:

- Runs 5 comprehensive tests:
    - **Test 1:** Catalog Service baseline (5,000 requests, 50 concurrency)
    - **Test 2:** Customer Service baseline (5,000 requests, 50 concurrency)
    - **Test 3:** Order Service baseline (5,000 requests, 50 concurrency)
    - **Test 4:** High concurrency test (10,000 requests, 100 concurrency)
    - **Test 5:** Sustained load test (30,000 requests × 3 services, 20 concurrency each)
- Collects metrics:
    - Requests per second (throughput)
    - Mean latency
    - Failed requests
    - Latency percentiles (p50, p95, p99)
- Generates reports:
    - Individual test results saved to `benchmark-results/`
    - Summary report with all metrics
    - Pod status during tests
    - HPA status (autoscaling triggers)

Test Types

1. **Baseline Performance Test**
   ab -n 5000 -c 50 -k -s 30 https://microservices.local:8443/catalog/
    - **Purpose:** Measure normal load performance
    - **Metrics:** Throughput, latency, reliability
    - **Services tested:** Catalog, Customer, Order
2. **High Concurrency Test**
   ab -n 10000 -c 100 -k -s 30 https://microservices.local:8443/catalog/
    - **Purpose:** Test under heavy concurrent connections

- **Result:** Catalog handled 290 req/s (3.35x baseline)

Metrics Collected

Performance Metrics:

- Requests per second: 86.80 [#/sec] (mean)
- Time per request: 576.028 [ms] (mean)
- Failed requests: 0
- 50% 453 ms ← Median latency
- 95% 1499 ms ← 95th percentile
- 99% 2298 ms ← 99th percentile (tail latency)

System Metrics:

- Pod CPU/Memory usage
- HPA status (current vs target replicas)
- Pod restarts
- Connection pool stats

Service	Throughput	Mean Latency	p99 Latency	Load Balancer Strategy
Catalog	86.80 req/s	576 ms	2,298 ms	Least …
Customer	70.82 req/s	706 ms	3,750 ms	IP Ha…
Order	15.52 req/s	3,221 ms	10,602 ms	Roun…
Catalog (High Concurrency)	290.99 req/s	344 ms	1,594 ms	Least …

9. Results and Observations

- Successfully replaced Apache with C++ Load Balancer

- Achieved secure HTTPS ingress with TLS termination
- Enabled autoscaling for all microservices
- Verified database persistence across restarts
- Real-time monitoring achieved via Prometheus + Grafana
- Benchmark shows improved throughput using Least Connections strategy