DOCKER AND KUBERNETES

Ticket-Issue Tracking App Deployment

Anusha Adarakatti
[01FE22BCS186]

This application is a multi-service issue tracking system designed to separate user and admin responsibilities using microservices architecture. It uses PostgreSQL as the database and Docker Compose to coordinate containerized services. The application is designed to be scalable and Kubernetes-ready.

1. Create the Application

a] Technology stack:
Flask (Python), PostgreSQL, HTML templates

b] Set up the backend:

- user_service/ and admin_service/ contain independent Flask applications.
- app.py in each folder defines routes for reporting and managing issues.
- HTML templates are rendered dynamically.
- Use requirements.txt for Python dependencies:

bash

```
pip install -r requirements.txt
```

---

2. Build and Push Docker Images to Docker Hub

a. Dockerize the Services:
Each service has its own Dockerfile.

b. Build the Docker images locally:

bash

```
docker build -t user-service ./user_service

docker build -t admin-service ./admin_service
```

c. Tag the Docker images:

bash

docker tag user-service your_dockerhub/user-service

docker tag admin-service your_dockerhub/admin-service

d. Push to Docker Hub:

bash

docker login

docker push your_dockerhub/user-service

docker push your_dockerhub/admin-service

3. Run the Application using Docker Compose

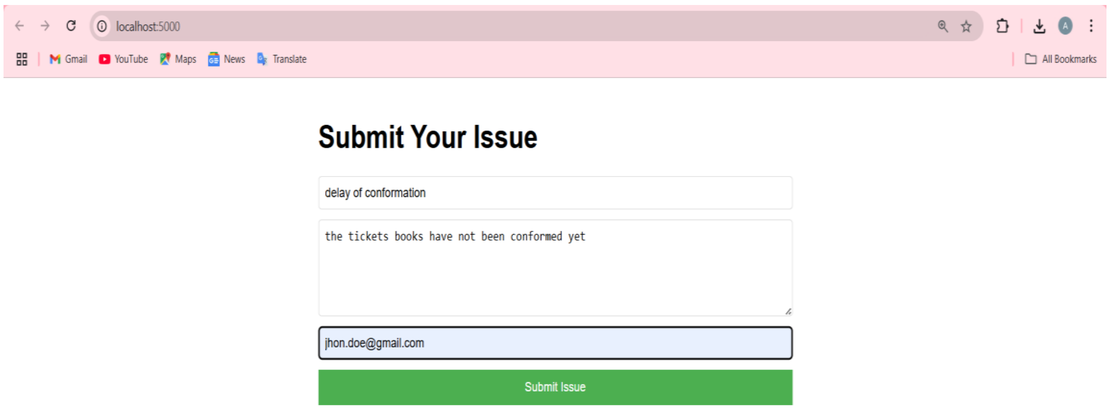
a. Start Containers

bash

docker-compose up --build

b. Services Available at:

- user_service: http://localhost:5000
- admin_service: http://localhost:5001
- PostgreSQL DB: localhost:543

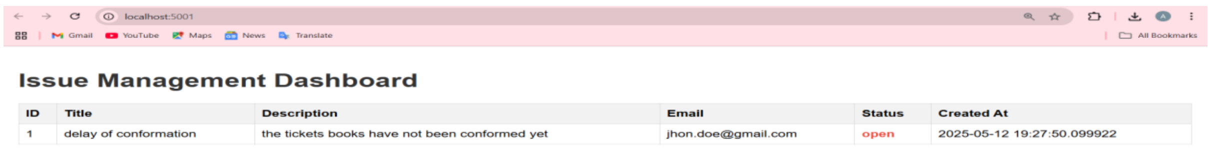


Submit Your Issue

delay of conformation

the tickets books have not been conformed yet

jhon.doe@gmail.com



Issue Management Dashboard

ID	Title	Description	Email	Status	Created At
1	delay of conformation	the tickets books have not been conformed yet	jhon.doe@gmail.com	open	2025-05-12 19:27:50.099922

KUBERNETES DEPLOYMENT

1. Setup Minikube Multinode Cluster

a. Install Tools

- Install Minikube and kubectl

b. Start Cluster with 3 Nodes

bash

```
minikube start --nodes=3
```

---

2. Deploy the Application on Minikube

a. Kubernetes Manifests Used:

- postgres.yaml
- user-app.yaml
- admin-app.yaml

b. Apply Manifests

bash

```
kubectl apply -f k8s/postgres.yaml
kubectl apply -f k8s/user-app.yaml
kubectl apply -f k8s/admin-app.yaml
```

c. Verify Services and Pods

bash

```
kubectl get pods
```

kubectl get services

---

3. Access the Application

a. Port Forward Services

bash

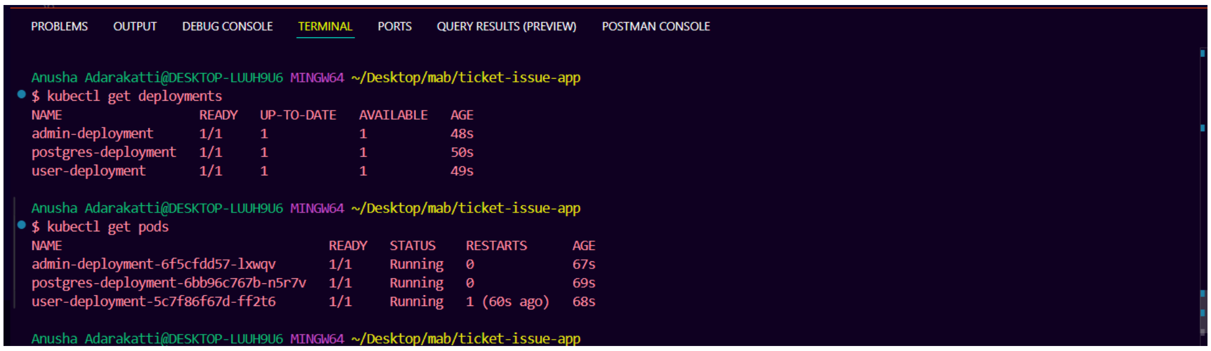
kubectl port-forward svc/user-service 5000:5000

kubectl port-forward svc/admin-service 5001:5001

b. Access via Browser

- http://127.0.0.1:5000 (User)
- http://127.0.0.1:5001 (Admin)

Deployments and Pods Details:



```
Anusha Adarakatti@DESKTOP-LUUH9U6 MINGW64 ~/Desktop/mab/ticket-issue-app
$ kubectl get deployments
NAME                  READY   UP-TO-DATE   AVAILABLE   AGE
admin-deployment      1/1     1            1           48s
postgres-deployment   1/1     1            1           50s
user-deployment       1/1     1            1           49s

Anusha Adarakatti@DESKTOP-LUUH9U6 MINGW64 ~/Desktop/mab/ticket-issue-app
$ kubectl get pods
NAME                                   READY   STATUS    RESTARTS      AGE
admin-deployment-6f5cfdd57-lxwqv       1/1     Running   0             67s
postgres-deployment-6bb96c767b-n5r7v   1/1     Running   0             69s
user-deployment-5c7f86f67d-ff2t6       1/1     Running   1 (60s ago)   68s

Anusha Adarakatti@DESKTOP-LUUH9U6 MINGW64 ~/Desktop/mab/ticket-issue-app
```

LOAD TESTING AND SCALABILITY (K8s)

Objective:

To assess system performance by simulating 300, 600, 900, 1200 concurrent requests.

Key Metrics:

- CPU usage
- Memory usage
- Response time (latency)
- Throughput (requests/sec)

Tools Used:

- curl in loop via request.sh

- Apache Benchmark (ab)

System Resource Setup:

Metric	Value
Initial Replicas	1 pod/service
Resources/Pod	2 CPU, 2 GB RAM
Cluster Size	4 nodes (Minikube)

Scaling Pods:

bash

```
kubectl scale deployment user-app --replicas=3

kubectl scale deployment admin-app --replicas=3
```

RESULTS

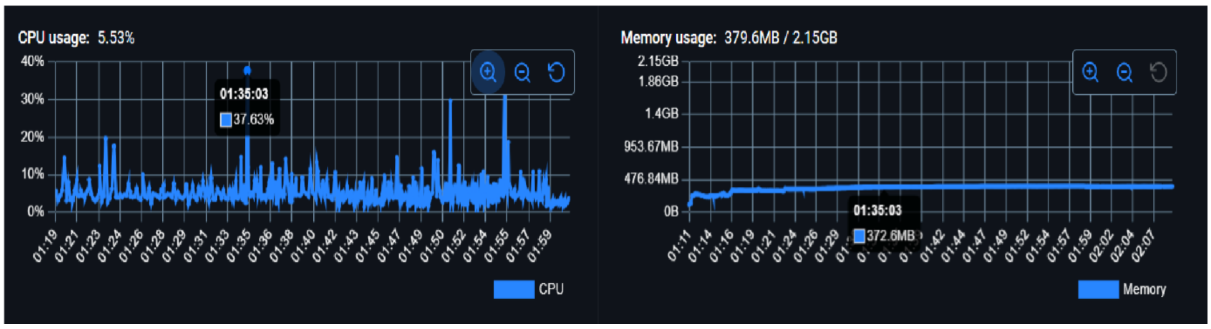
Resource Usage:

Requests	CPU Usage (%)	Memory (MB)
300	28.17	701.6
600	36.17	704.2
900	46.82	706.5
1200	65.70	707.0

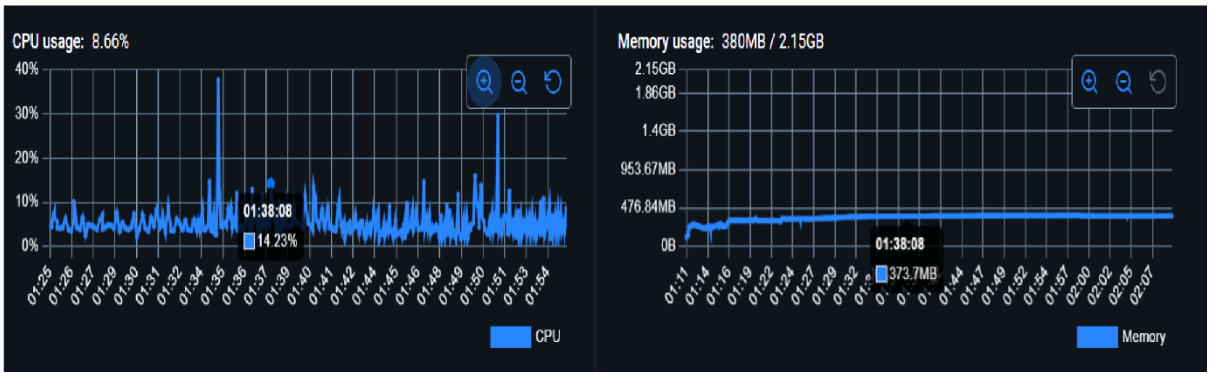
<u>Throughput and Latency analysis for minikube :</u>

minikube-m01

Replica=1



Replica=3: kubectl scale deployment user-app--replicas=3

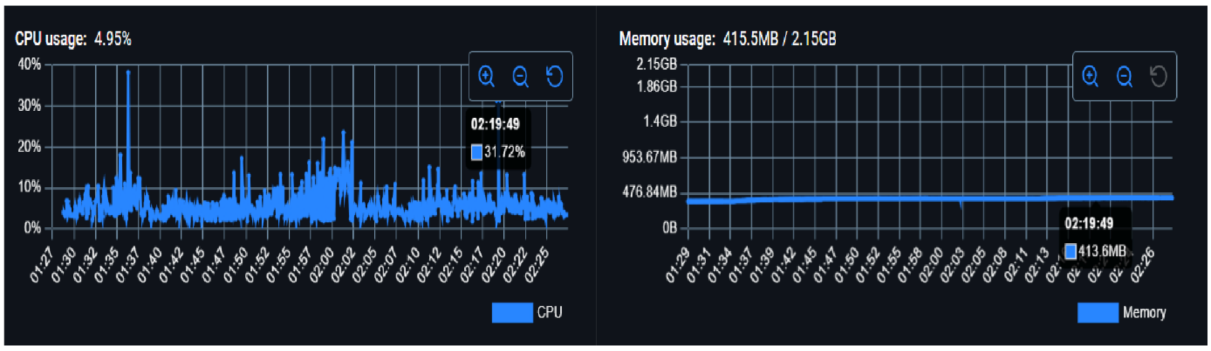


for ($i=1; $i -le [300,600,900,1200]; $i++) { curl http://127.0.0.1:64417 }

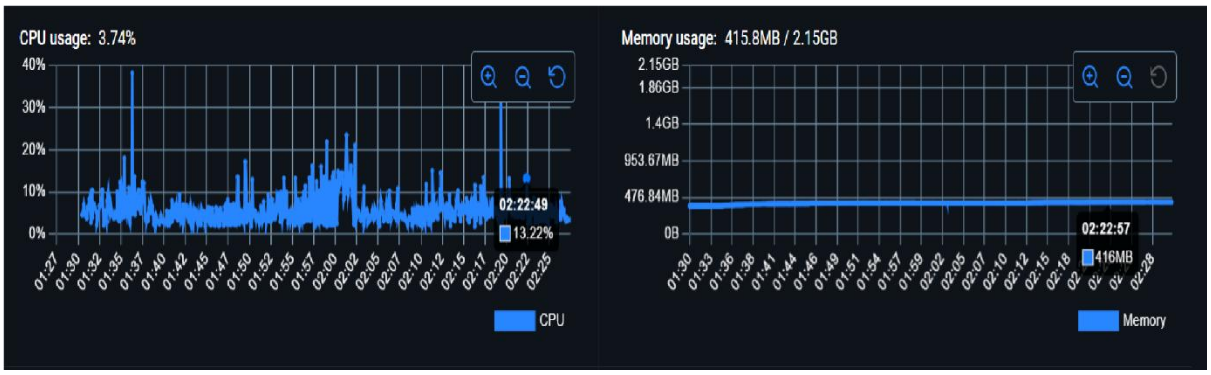
Number of Requests	CPU Usage (%)	Memory Usage (in MB)
300	10.30	379.1
600	12.07	380.0
900	15.23	384.7
1200	17.64	385.9

minikube-m02

Replica=1



Replica=3: kubectl scale deployment admin-app--replicas=3



for ($i=1; $i -le [300,600,900,1200]; $i++) { curl http://127.0.0.1:59428 }

Number of Requests	CPU Usage (%)	Memory Usage (in MB)
300	14.07	415.7
600	15.79	416
900	17.11	417.2
1200	19.05	418.5

Throughput and Latency Analysis (Apache Benchmark)

Replica=1:

PS C:\xampp\apache\bin> .\ab -n 300 -c 10 http://127.0.0.1:59428/

Requests per second: 141.32 [#/sec] (mean) [THROUGHPUT]

Time per request: 70.762 [ms] (mean) [LATENCY ANALYSIS]

Time per request: 7.076 [ms] (mean, across all concurrent requests) Transfer rate: 238.34 [Kbytes/sec] received

PS C:\xampp\apache\bin> .\ab -n 600 -c 10 http://127.0.0.1: 59428/

Requests per second: 391.20 [#/sec] (mean) [THROUGHPUT]

Time per request: 25.562 [ms] (mean) [LATENCY ANALYSIS]

Time per request: 2.556 [ms] (mean, across all concurrent requests) Transfer rate: 659.77 [Kbytes/sec] received

PS C:\xampp\apache\bin> .\ab -n 900 -c 10 http://127.0.0.1: 59428/

Requests per second: 649.82 [#/sec] (mean)   [THROUGHPUT]

Time per request: 15.389 [ms] (mean) [LATENCY ANALYSIS]

Time per request: 1.539 [ms] (mean, across all concurrent requests) Transfer rate: 1095.94 [Kbytes/sec] received

Throughput Analysis:

1] Throughput increases as the number of requests increases: 141 → 391 → 649 req/sec.

2] Server handles more load efficiently — no signs of performance saturation.

3] Indicates good scalability and minimal resource contention.

Latency Analysis:

1] Latency per request decreases as load increases:

70.76 ms → 25.56 ms → 15.39 ms.

2]Suggests server warms up or benefits from internal optimizations (e.g., caching or CPU ramp-up).

3] Lower latency at higher loads is a sign of efficient backend handling.

Increased replicas to 3:

kubectl scale deployment admin-service --replicas=3

PS C:\xampp\apache\bin> .\ab -n 300 -c 10 http://127.0.0.1: 59428/

Requests per second: 371.72 [#/sec] (mean)

Time per request: 26.902 [ms] (mean)

Time per request: 2.690 [ms] (mean, across all concurrent requests)

PS C:\xampp\apache\bin> .\ab -n 600 -c 10 http://127.0.0.1: 59428/

```
Requests per second:    510.96 [#/sec] (mean)
Time per request:       19.571 [ms] (mean)
Time per request:       1.957 [ms] (mean, across all concurrent requests)
```

```
PS C:\xampp\apache\bin> .\ab -n 900 -c 10 http://127.0.0.1: 59428/
Requests per second:    680.06 [#/sec] (mean)
Time per request:       14.705 [ms] (mean)
Time per request:       1.470 [ms] (mean, across all concurrent requests)
```

Throughput Analysis (Requests per Second)

- Increasing the number of replicas from 1 to 3 led to improved throughput across all levels of incoming requests.
- At 300 requests, throughput increased significantly—from 141 requests/second to 371 requests/second.
- At 600 requests, the throughput rose from 391 requests/second to 510 requests/second, indicating a strong performance boost.
- At 900 requests, throughput improved marginally—from 649 requests/second to 680 requests/second—suggesting that the system may be approaching its performance threshold.
- Overall, the server handled higher loads more efficiently with 3 replicas, particularly under low to medium request volumes.

Latency Analysis (Time per Request in ms)

- Latency per request decreased as the number of replicas increased, indicating improved responsiveness under load.
- At 300 requests, latency dropped significantly from 70.76 ms to 26.90 ms.
- At 600 requests, latency reduced from 25.56 ms to 19.57 ms, showing continued performance gains.
- At 900 requests, latency decreased slightly from 15.39 ms to 14.70 ms, suggesting diminishing returns at higher loads.
- The most notable reductions in latency occurred under lower traffic conditions, indicating that the server responds more efficiently when not heavily burdened.

Conclusion

- Increasing the number of replicas from 1 to 3 led to significant improvements in both throughput and latency.
- The server demonstrated effective scalability, efficiently distributing incoming load across multiple replicas.
- This architecture is well-suited for handling higher traffic volumes while maintaining optimal performance and responsiveness.