Analysis of Microservices Architecture for a Social Network Platform: Deployment, Performance, and Scalability on Kubernetes

Subject: Graduate Systems - Winter Semester, Group-29

Dhruvkumar Patel	**Harsh Pala**	**Gour Krishna Dey**	**Pulkit Kumar**
MT24032,IIIT-Delhi	*MT24119, IIIT-Delhi*	*MT24035,IIIT-Delhi*	*MT24069,IIIT-Delhi*
dhruvkumar24032@iiitd.ac.in	harsh24119@iiitd.ac.in	gour24035@iiitd.ac.in	pulkit24069@iiitd.ac.in

Abstract

This project analyzes the architecture, performance, and scalability of a social network platform implemented using microservices architecture, specifically leveraging the DeathStarBench [1] social network implementation. The system comprises multiple microservices communicating via Thrift RPCs, deployed on Kubernetes for orchestration. The project involves setting up the application, analyzing its microservices architecture, evaluating performance under different load conditions, and assessing various deployment configurations. By varying container locations across different nodes, we aim to provide insights into the benefits and challenges of microservices architecture for social network applications, particularly in terms of scalability, resilience, and performance optimization.

I. Problem Statement

Modern social media platforms demand high availability, low latency, and seamless scalability—challenges that traditional monolithic architectures often fail to meet. Microservices offer a modular solution but introduce complexities in deployment, communication, and monitoring.

This project investigates the deployment and performance of a microservices-based social network (DeathStarBench) on Kubernetes. We analyze how service placement and configuration strategies affect scalability, latency, and resource utilization.

II. Project Goals

- **Deploy the DeathStarBench Social Network:** Containerize and orchestrate all microservices using Kubernetes.
- **Understand the Architecture:** Analyze service interconnects, data flow, and RPC-based communication.
- **Evaluate Performance:** Measure system metrics across different configurations:
  - We evaluate the system under varying configurations by altering node placements, the number of replicas, and the number of nodes. As detailed in Section VI, our experimental setup includes deployments of microservices in the cloud across three configurations: (i) single node with a single replica, (ii) single node with three replicas, and (iii) multiple nodes with a single replica. While auto-scaling mechanisms could be employed, we chose to fix the number of nodes in order to ensure a fair and consistent comparison of load balancing performance across the different setups.

- **Run Load Tests:** Simulate user activity and study system behavior under stress.
- **Optimize Deployment:** Identify best deployment strategies based on experimental outcomes.

III. Expected Outcomes/Deliverables

- A fully functional microservices social network on Kubernetes.
- A comprehensive performance report including latency, throughput, and scalability insights.
- Documentation and a final presentation covering methodology, findings, and recommendations.

IV. System Design

The DeathStarBench Social Network is a microservices-based cloud-native application that replicates the behavior of a real-world social media platform. It is designed to benchmark and evaluate distributed system components, particularly in environments using Kubernetes and service meshes. The application is structured using more than 20 microservices, each handling a specific responsibility, and communicating over the RPC protocol, named Thrift RPC. These services are written in multiple languages (e.g., C++, Java, Go, Node.js, Python) to simulate polyglot environments typical in modern production systems.

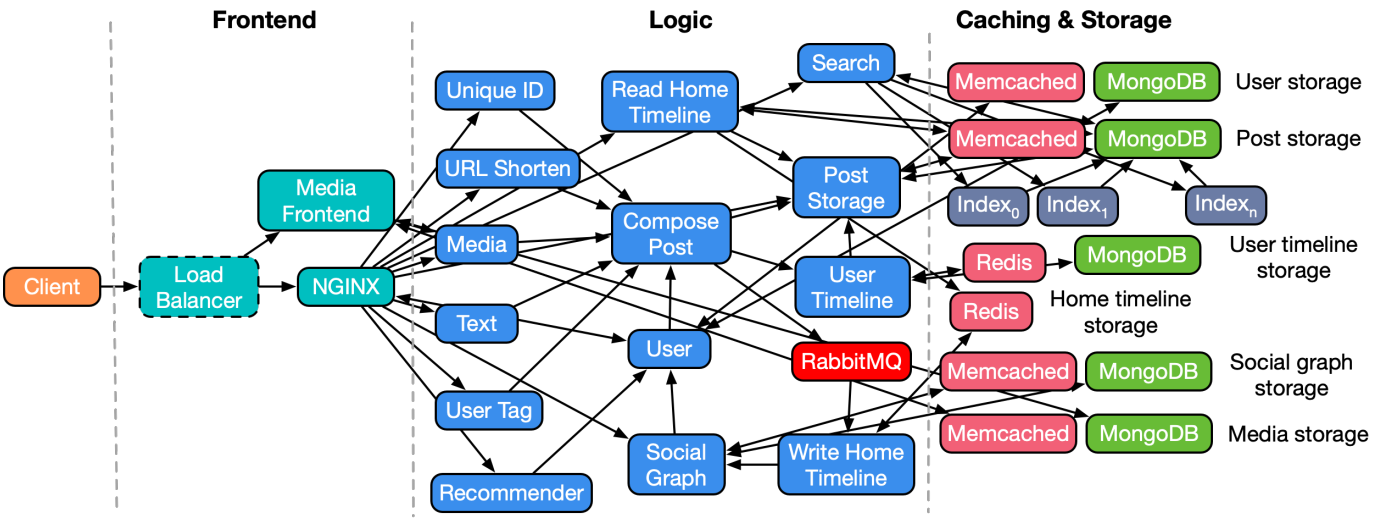


Fig. 1: Deathstar Microservice Architecture: Social Networking

A. Microservices and Supporting Services

The system consists of core microservices, each responsible for a specific functionality:

- **nginx-web-server:** Entry point for client HTTP requests.
- **compose-post-service:** Coordinates post creation across services.
- **text-service:** Handles text content of posts.
- **media-service:** Manages image/video uploads.
- **post-storage-service:** Stores metadata about posts.
- **unique-id-service:** Generates unique post/user IDs.

- **user-service:** Manages user data and login info.
- **social-graph-service:** Tracks follow relationships.
- **home-timeline-service:** Displays posts from followed users.
- **user-timeline-service:** Displays personal post history.
- **user-mention-service:** Processes tagged mentions in posts.
- **url-shorten-service:** Provides URL shortening.

B. Technologies Used

The deployment was done locally via Docker Compose. The following steps summarize the setup process:

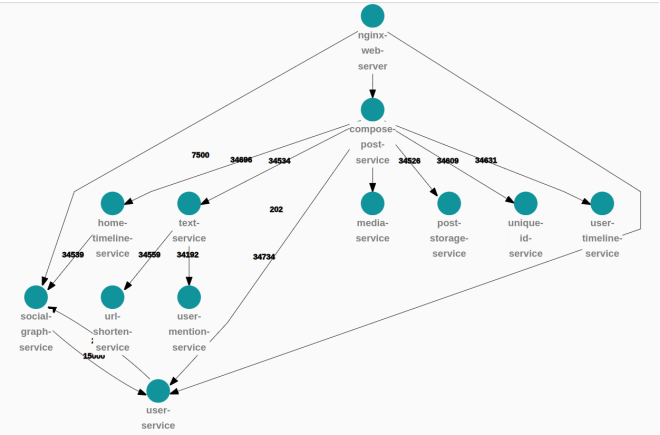
- Cloned the `DeathStarBench` repository using `git clone --recursive`.
- Navigated to the `socialNetwork` directory.
- Launched all services with `docker-compose up -d`.
- Verified container status using `docker ps`.
- Populated MongoDB with synthetic data for core services.
- Simulated load using the `wrk2` workload generator and Lua scripts.

C. Service Interactions

Each microservice communicates with others over Thrift. For example:

- `compose-post-service` → `text-service` on port **7500**
- `compose-post-service` → `user-mention-service` on port **34734**
- `compose-post-service` → `post-storage-service` on port **34609**

These services form a call graph during execution. Below is an example of the system interaction flow visualized via Jaeger:



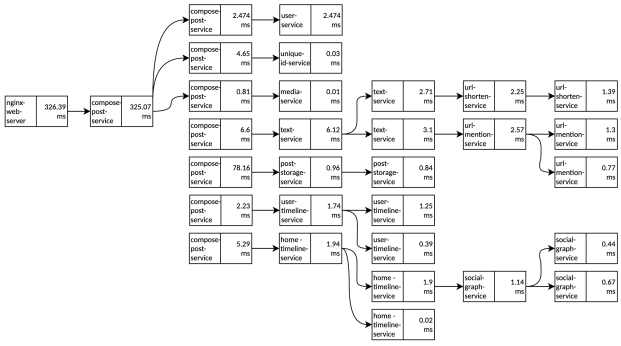


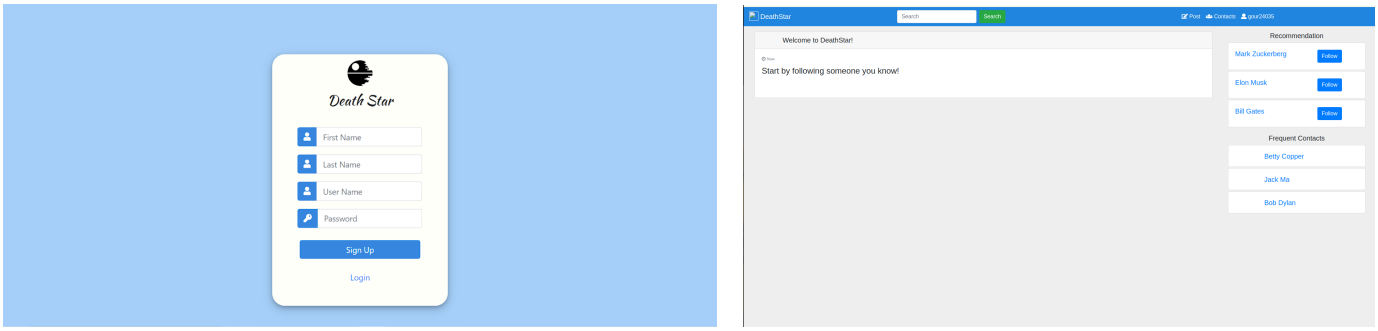
Fig. 2: Service Interaction Trace

V. System Implementation

A. Local Deployment using Docker

The DeathStarBench social network was initially deployed locally using Docker Compose, offering an easy way to spin up all microservices and their dependencies.

- The project repository was cloned recursively using: `git clone --recursive`.
- Docker and Docker Compose were installed and verified.
- From within the `socialNetwork` directory, all services were brought up using: `docker-compose up -d`.
- The setup initializes core microservices including `user`, `post`, `social-graph`, `home-timeline`, `user-timeline`, and `nginx`, among others.
- MongoDB was seeded with a synthetic dataset to simulate user and post activity.
- The load generator `wrk2` was compiled and configured using Lua scripts to simulate realistic user behaviors.
- Jaeger was enabled as a distributed tracing solution and accessed at: `http://localhost:16686`.
- The frontend was made available at: `http://localhost:8080`.



(a) Login Screen (b) Home Feed

Fig. 3: Frontend UI (Exposed at Port no 8080)

B. Local Kubernetes Migration

To replicate a more production-like deployment environment and enable advanced observability tools, the DeathStarBench social network was migrated from Docker Compose to **Kubernetes** using **Helm charts**.

- The existing Docker-based deployment was containerized further into Kubernetes-compatible resources using Helm.
- A local Kubernetes cluster via `k3s` was set up and verified for resource provisioning.
- Helm v3 was installed and initialized for templated service deployment.
- From the `k8s` directory of the DeathStarBench repository, the following command was used to deploy all microservices:

```
$ helm install social-network. -n deathstarbench --create-namespace
```

C. Deployment on Google Cloud using Kubernetes

For cloud-based deployment, the microservices were containerized and orchestrated using Kubernetes on Google Cloud Platform (GCP).

- A GKE (Google Kubernetes Engine) cluster was created using -

```
$ gcloud container clusters create deathstar-cluster \
   --zone asia-south1-a \
   --num-nodes=<number of nodes>

$ gcloud container clusters get-credentials deathstar-cluster --zone asia-south1-a

$ kubectl create namespace deathstarbench

$ helm install social-network . -n deathstarbench --create-namespace
```

- Kubernetes manifest files were written for each microservice, defining deployments, services, and configuration maps.
- Docker images were built locally and pushed to Google Container Registry (GCR) for cloud access.
- Jaeger and MongoDB services were deployed as part of the cluster.
- Services were exposed either via NodePorts or LoadBalancers for external access.
- Kubernetes commands like `kubectl apply -f *.yaml` were used to deploy services.
- Helm charts were explored to simplify dependency management.
- The deployed services were monitored using Kubernetes Dashboard and logs.

D. Creativity and Problem Solving

In large-scale microservice architectures like **DeathStarBench/SocialNetwork**, achieving robust observability and effective debugging presents unique challenges, especially when standard tools and metrics are unavailable out-of-the-box. During our implementation, we encountered several such obstacles and addressed them with creative engineering solutions:

- **Observability Without Prometheus Metrics:** DeathStarBench services do not expose Prometheus-compatible `/metrics` endpoints, making it difficult to use standard monitoring tools. To overcome this, we deployed **Pixie**, an open-source observability platform that leverages eBPF (Extended Berkeley Packet Filter) probes to provide auto-telemetry. Pixie enabled us to:
  - Visualize the entire microservice call graph in real-time.
  - Monitor live HTTP metrics such as RPS, latency, and error rates per endpoint.
  - Generate CPU flamegraphs to identify performance bottlenecks.
  - Achieve all of the above *without any code changes or service-level instrumentation*.

  This approach provided Google Cloud-like observability directly on our local Kubernetes cluster, significantly improving our ability to monitor, debug, and profile the system.
- **Reviving Jaeger Tracing for DeathStarBench:** The default Jaeger tracing setup failed due to incomplete header propagation and lack of service instrumentation. We addressed this by:
  - Reconfiguring the `docker-compose.yaml` and Jaeger configuration files.

  - Ensured all services and dependencies were deployed in the same user defined bridged docker network named `dsbnet`.
  - Verified successful inter-service communication and trace collection within the shared network.

  These manual interventions restored trace visibility and allowed us to verify service-to-service traces in the Jaeger UI, enabling root cause analysis and performance profiling across microservices.
- **Version Compatibility:** Initial Jaeger setup failed due to version mismatches. We resolved this by downgrading to a stable Jaeger release compatible with DeathStarBench, ensuring reliable tracing functionality.
- **Port Conflicts:** Services such as Jaeger and NGINX require unique ports across local and cloud environments to avoid conflicts. We standardized port assignments and updated deployment manifests to ensure consistent service accessibility.
- **Load Testing in Kubernetes:** Running `wrk2` for load testing inside a containerized environment presented configuration challenges. We created a dedicated Kubernetes job container specifically for `wrk2`, allowing us to orchestrate and automate load testing as part of our CI/CD pipeline.
- **kubectl CLI Limitations:** Errors arising from version mismatches between the `kubectl` client and the Kubernetes cluster were resolved by manually syncing client and cluster versions, ensuring smooth management and deployment operations.
- **Service Failures and Debugging:** We utilized extensive logs and the `kubectl describe` command to diagnose and resolve service failures. This iterative debugging process allowed us to quickly identify root causes and restart failed pods as needed.

These creative solutions transformed our local environment into a production-grade observability and testing stack, enabling faster debugging, realistic load testing, and robust performance optimization prior to cloud deployment. This reflects engineering creativity in addressing the practical constraints of large-scale microservice architectures.

	Pixie	**Prometheus**
Pros	1. No manual instrumentation due to eBPF auto-telemetry. 2. Real-time insights with minimal setup.	1. Efficient time-series storage and querying. 2. Large ecosystem and community support.
Cons	1. Higher resource overhead. 2. Cannot extract granular custom metrics like Prometheus.	1. Requires manual code instrumentation. 2. Limited out-of-the-box tracing and flamegraph capabilities.

TABLE I: Pros and Cons of using Pixie over Prometheus

VI. System Evaluation

This section presents the experimental evaluation of our microservice-based system under varying deployment configurations. The primary goal is to analyze the performance and behaviour of the system under different load conditions by altering the number of replicas and nodes. The following configurations were tested: (i) single node with a single replica, (ii) single node with three replicas, and (iii) three nodes with a single replica each. The synthetic load generator was used to simulate realistic web traffic during the experiments.

Synthetic load generator code snippet:

```
$  ../wrk2/wrk -D exp -t <num-threads> -c <num-conns> -d <duration> -L -s ./wrk2/
   scripts/social-network/compose-post.lua http://<nginx-ip>:8080/wrk2-api/post/
   compose -R <reqs-per-sec>
```

A. Single Node, Single Replica

In this configuration, the microservices were deployed on a single node with only one replica running. The aim of this setup was to observe the baseline performance of the system without any load distribution mechanisms.

The load testing was conducted using a synthetic traffic generator configured to send 10 requests per second for a duration of 60 seconds. The test was executed using 12 threads and 400 connections to mimic concurrent access patterns. Figure 4 illustrates the deployment setup of this configuration.

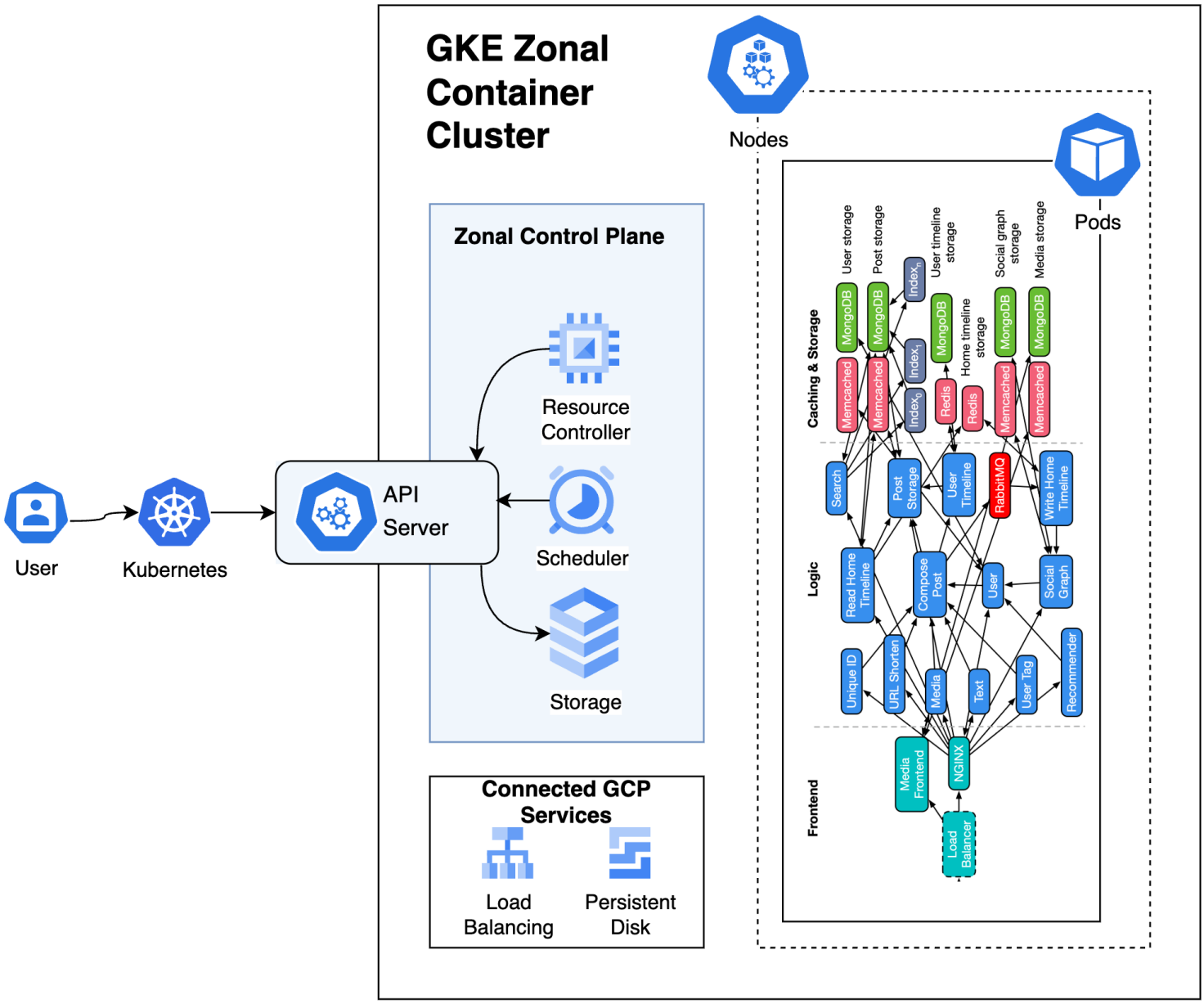


Fig. 4: Deployment architecture for single node, single replica configuration

CLI Outputs

```
palaharsh@debian:~/DeathStarBench/socialNetwork$ ../wrk2/wrk -D exp -t 12 -c 4
Running 1m test @ http://35.200.241.115:8080/wrk2-api/post/compose
  12 threads and 400 connections

  Thread calibration: mean lat.: 3509.533ms, rate sampling interval: 12328ms
  Thread calibration: mean lat.: 3102.208ms, rate sampling interval: 11370ms
  Thread calibration: mean lat.: 4472.040ms, rate sampling interval: 17203ms
  Thread calibration: mean lat.: 3886.493ms, rate sampling interval: 12886ms
  Thread calibration: mean lat.: 3036.056ms, rate sampling interval: 10412ms
  Thread calibration: mean lat.: 6462.549ms, rate sampling interval: 17235ms
  Thread calibration: mean lat.: 4368.310ms, rate sampling interval: 15097ms
  Thread calibration: mean lat.: 3688.789ms, rate sampling interval: 12075ms
  Thread calibration: mean lat.: 3214.336ms, rate sampling interval: 11788ms
  Thread calibration: mean lat.: 7615.692ms, rate sampling interval: 18726ms
  Thread calibration: mean lat.: 3336.804ms, rate sampling interval: 11911ms
  Thread calibration: mean lat.: 4711.695ms, rate sampling interval: 15605ms
```

Fig. 5: CLI output of the synthetic load generator during single-node load testing using WRK2. This includes request generation details and configuration parameters such as threads, connections, and request rate.

```
----------------------------------------------------------------------
Test Results @ http://35.200.241.115:8080/wrk2-api/post/compose
  Thread Stats   Avg      Stdev     99%   +/- Stdev
    Latency    30.27s    13.53s    0.95m    60.87%
    Req/Sec     6.03      2.65    12.00     76.32%
  Latency Distribution (HdrHistogram - Recorded Latency)
 50.000%   30.65s
 75.000%   41.29s
 90.000%   49.41s
 99.000%    0.95m
 99.900%    0.97m
 99.990%    0.99m
 99.999%    0.99m
100.000%    0.99m
```

Fig. 6: WRK2 latency statistics including average latency, percentile distribution, and request statistics. These results are critical in evaluating service responsiveness under sustained load.

Plots and Observations

The experiment observations are as listed below:

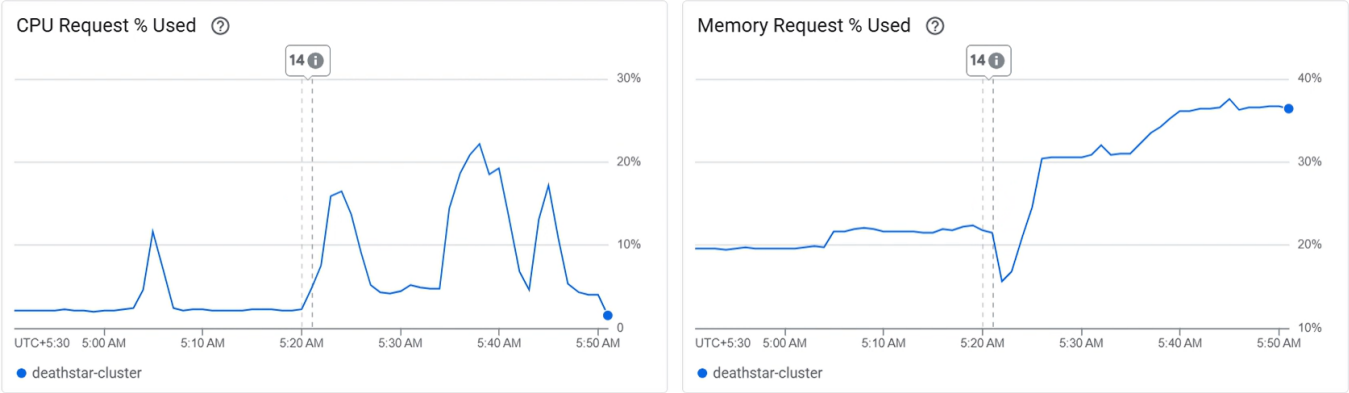


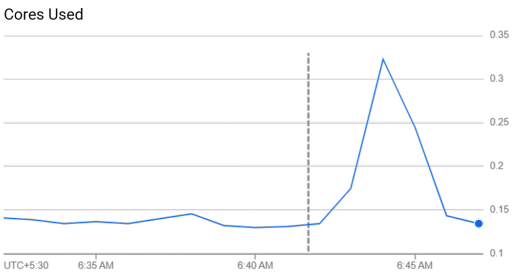
Fig. 7: Resource monitoring plot showing spikes in CPU and memory request percentages during synthetic load execution. The plot reflects the stress imposed on the system during high-throughput testing.

Kubernetes services | No active alerts | 0 kubernetes services with active alerts | View all

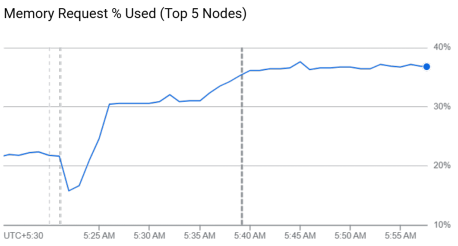
Name	Alerts	Labels	Container restarts	Error logs	CPU utilization	Memory utilization	Disk utilization
compose-post-service	0	Cluster: deathstar-cl... +3	0	5,419	0.01% of 0.2 CPU	1.56% of 256 MiB	0% of 188.76 GiB
home-timeline-redis	0	Cluster: deathstar-cl... +3	0	0	2.04% of 0.1 CPU	1.29% of 128 MiB	0% of 94.38 GiB
home-timeline-service	0	Cluster: deathstar-cl... +3	0	9	0.01% of 0.1 CPU	2.21% of 128 MiB	0% of 94.38 GiB
jaeger	0	Cluster: deathstar-cl... +3	2	5	0.69% of 0.1 CPU	24.04% of 128 MiB	0% of 94.38 GiB
jaeger-lb	0	Cluster: deathstar-cl... +3	2	5	0.69% of 0.1 CPU	24.04% of 128 MiB	0% of 94.38 GiB

1 – 5 of 43

Fig. 8: Snapshot of error logs from the Kubernetes engine on GCP, showing the types of errors encountered during heavy traffic simulation. This helps identify possible failure points or misconfigurations.



(a) Redis service performance under load showing a significant spike in cores used. This indicates the demand on the caching layer during high-frequency post composition requests.



(b) Overall memory usage plot highlighting an approximate 20% increase during stress testing. This provides evidence of system-level impact due to load.

Fig. 9: CPU and Memory utilization

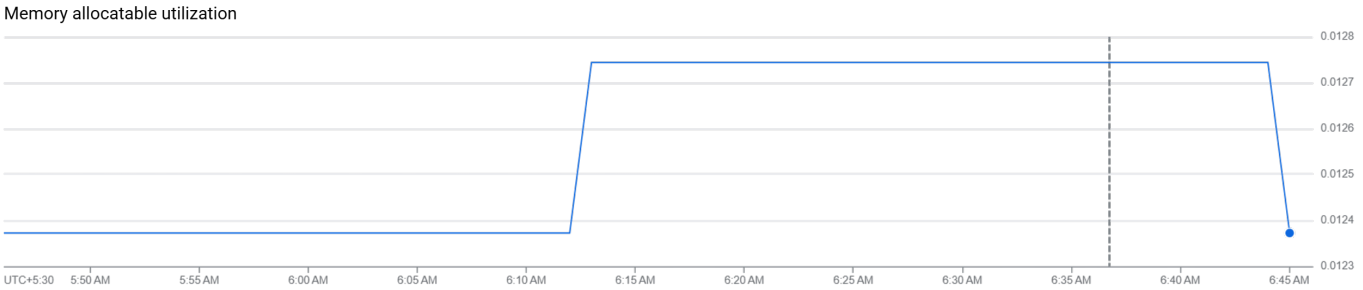


Fig. 10: Redis memory allocatable utilization plot showing trends of memory demand increase due to large-scale request operations.

Creative Plots Visualization from Pixie

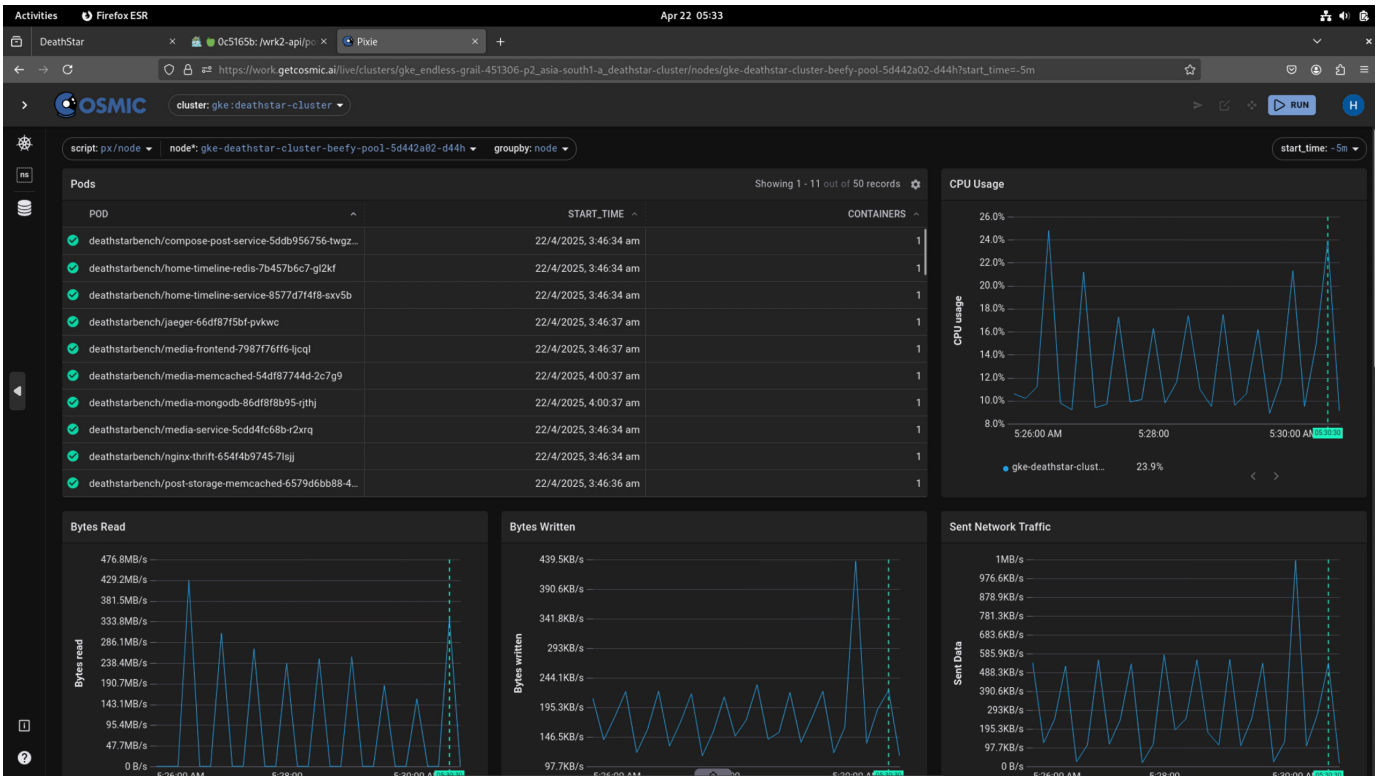


Fig. 11: Pixie visualization depicting read/write throughput and network I/O spikes during synthetic load simulation. This view helps correlate system I/O load with request patterns.

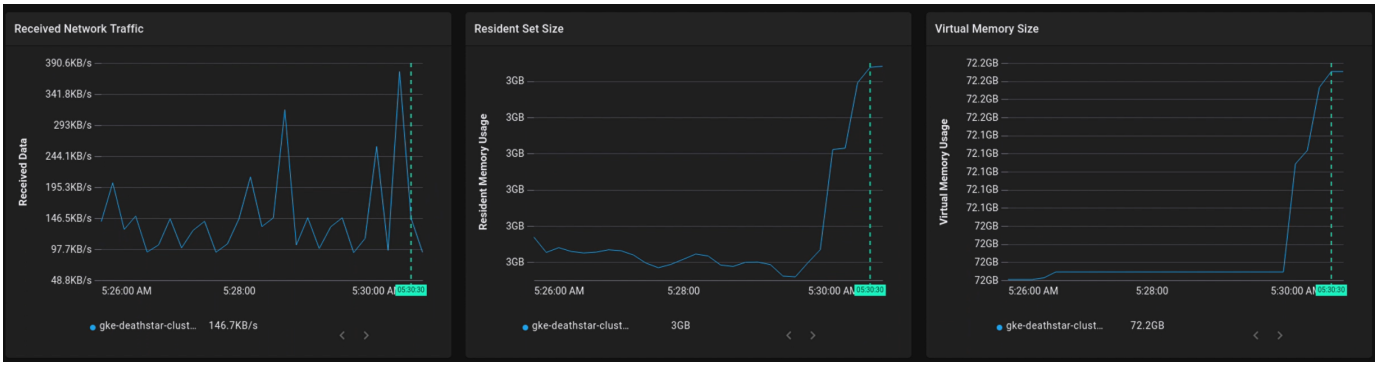


Fig. 12: Pixie plot showing spikes in network traffic, virtual memory size (VMS), and resident set size (RSS). These metrics reveal memory pressure and network stress levels.

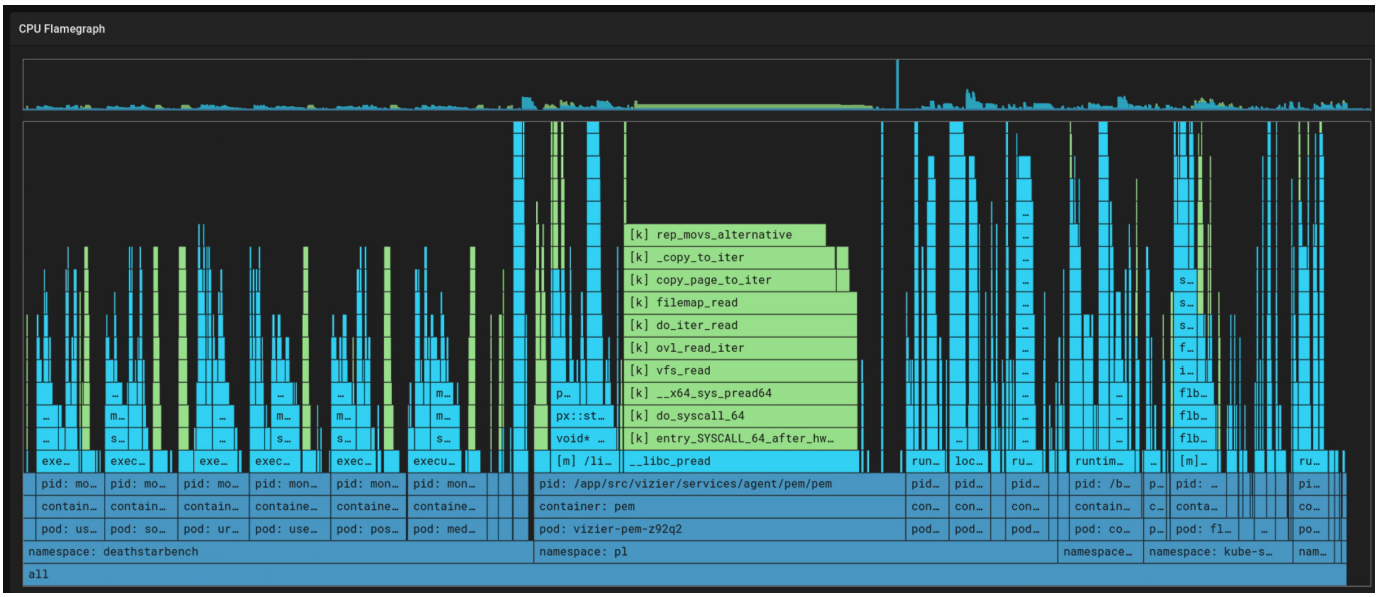


Fig. 13: CPU Flame Graph from Pixie illustrating the call stack and CPU function usage distribution. It helps identify bottlenecks and hot paths during load.

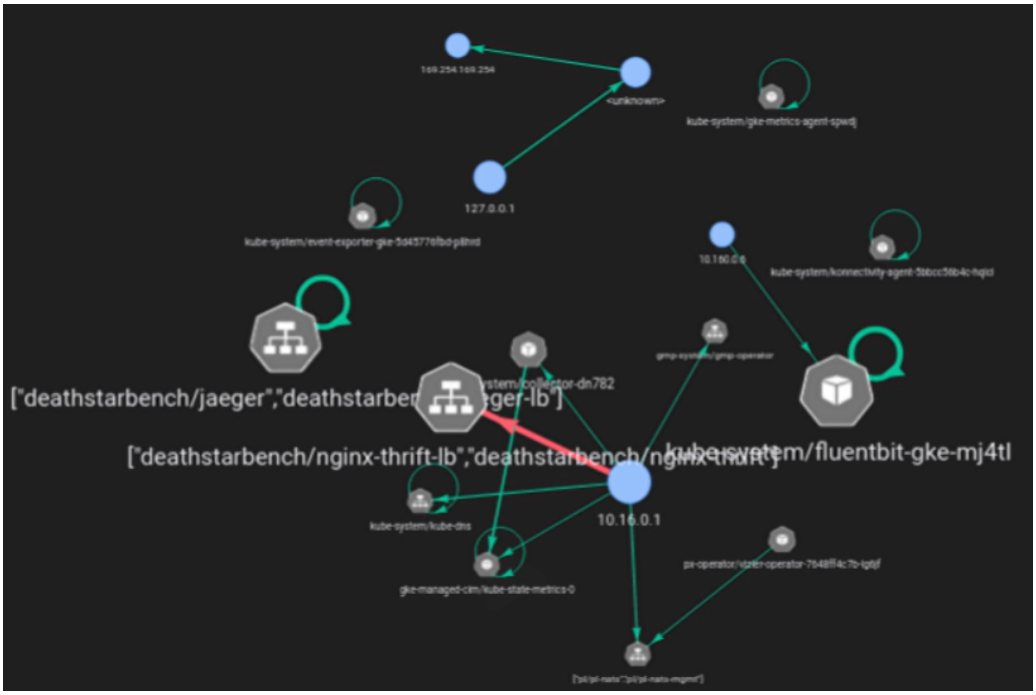


Fig. 14: Pixie-generated cluster graph showing inter-service communications and dependencies. This is useful for understanding workload propagation during testing.

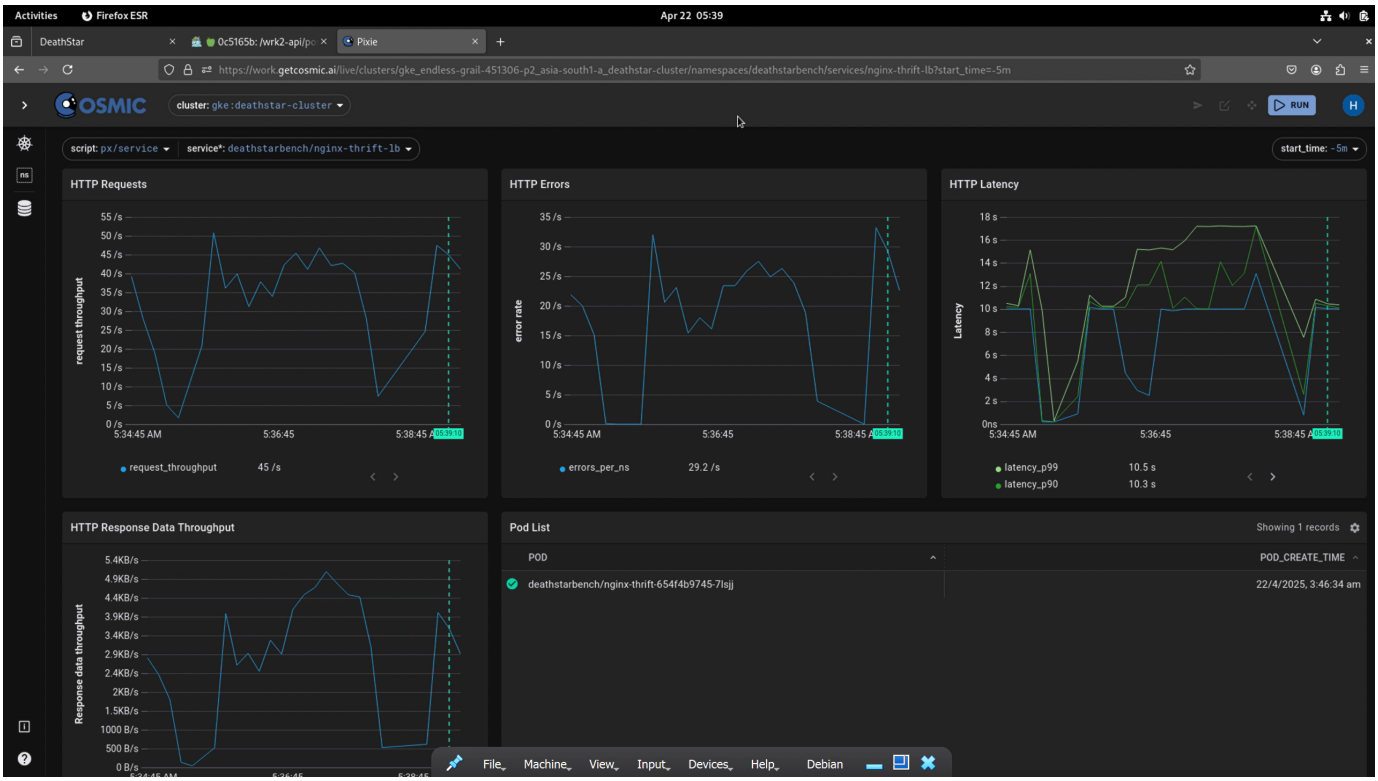


Fig. 15: Pixie HTTP analysis dashboard displaying spikes in request volume, latency, errors, and response throughput. These metrics provide a complete picture of endpoint stress.

B. Single Node, Three Replicas

This configuration maintains a single-node setup but scales the application horizontally by increasing the number of replicas to three. This setup evaluates intra-node load balancing behavior and examines improvements in throughput and latency under increased concurrent load.

The synthetic load generator was configured to simulate high concurrency by sending 50 requests per second for 60 seconds using 12 threads and 400 connections.

Figures 16 and 17 depict the deployment architecture and CLI view for the single node with three replicas.

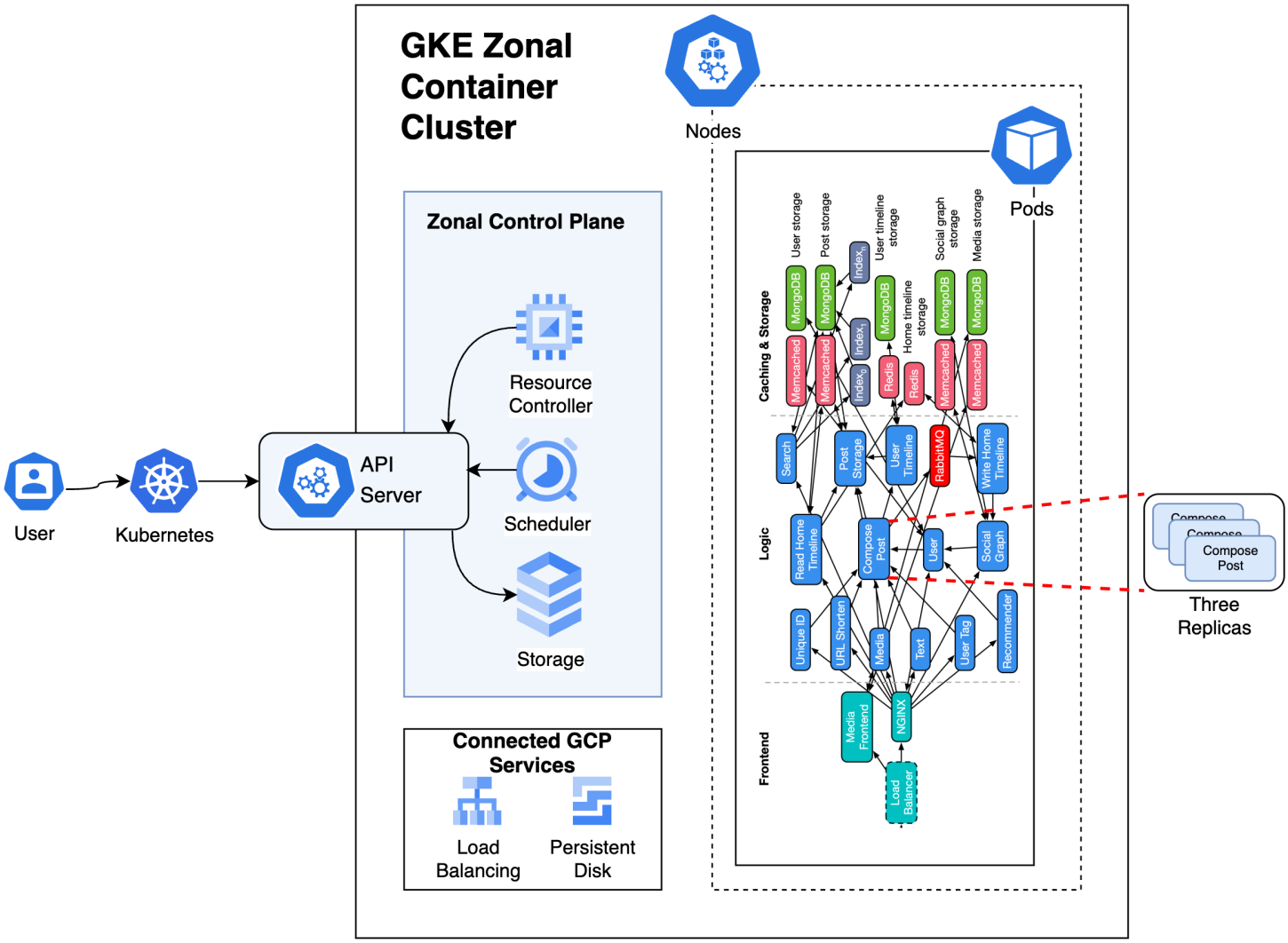


Fig. 16: GKE deployment architecture view showing a single node hosting three application replicas.

CLI Outputs

```
palaharsh@debian: ~/DeathStarBench/socialNetwork
deployment.apps/media-mongodb            1/1     1            1           3h15m
deployment.apps/media-service            1/1     1            1           3h15m
deployment.apps/nginx-thrift             1/1     1            1           3h15m
deployment.apps/post-storage-memcached   1/1     1            1           3h15m
deployment.apps/post-storage-mongodb     1/1     1            1           3h15m
deployment.apps/post-storage-service     1/1     1            1           3h15m
deployment.apps/social-graph-mongodb     1/1     1            1           3h15m
deployment.apps/social-graph-redis       1/1     1            1           3h15m
deployment.apps/social-graph-service     1/1     1            1           3h15m
deployment.apps/text-service             1/1     1            1           3h15m
deployment.apps/unique-id-service        1/1     1            1           3h15m
deployment.apps/url-shorten-memcached    1/1     1            1           3h15m
deployment.apps/url-shorten-mongodb      1/1     1            1           3h15m
deployment.apps/url-shorten-service      1/1     1            1           3h15m
deployment.apps/user-memcached           1/1     1            1           3h15m
deployment.apps/user-mention-service     1/1     1            1           3h15m
deployment.apps/user-mongodb             1/1     1            1           3h15m
deployment.apps/user-service             1/1     1            1           3h15m
deployment.apps/user-timeline-mongodb    1/1     1            1           3h15m
deployment.apps/user-timeline-redis      1/1     1            1           3h15m
deployment.apps/user-timeline-service    1/1     1            1           3h15m

NAME                                              DESIRED   CURRENT   READY   AGE
replicaset.apps/compose-post-service-5ddb956756   3         3         3       3h15m
replicaset.apps/home-timeline-redis-7b457b6c7     1         1         1       3h15m
replicaset.apps/home-timeline-service-8577d7f4f8  1         1         1       3h15m
replicaset.apps/jaeger-66df87f5bf                 1         1         1       3h15m
replicaset.apps/media-frontend-7987f76ff6         1         1         1       3h15m
replicaset.apps/media-memcached-54df87744d        1         1         1       3h15m
replicaset.apps/media-mongodb-86df8f8b95          1         1         1       3h15m
replicaset.apps/media-service-5cdd4fc68b          1         1         1       3h15m
replicaset.apps/nginx-thrift-654f4b9745           1         1         1       3h15m
replicaset.apps/post-storage-memcached-6579d6bb88 1         1         1       3h15m
replicaset.apps/post-storage-mongodb-5c9fd8dbbc   1         1         1       3h15m
replicaset.apps/post-storage-service-7745cd8b95   1         1         1       3h15m
replicaset.apps/social-graph-mongodb-84f75d69d7   1         1         1       3h15m
replicaset.apps/social-graph-redis-b9b8bc878      1         1         1       3h15m
replicaset.apps/social-graph-service-d95bbf5f7    1         1         1       3h15m
replicaset.apps/text-service-845f497f4c           1         1         1       3h15m
replicaset.apps/unique-id-service-798bc654d8      1         1         1       3h15m
replicaset.apps/url-shorten-memcached-78c7d97fc4  1         1         1       3h15m
replicaset.apps/url-shorten-mongodb-77bbd77b84    1         1         1       3h15m
```

Fig. 17: CLI output listing the active pods distributed across three replicas.

```
palaharsh@debian:~/DeathStarBench/socialNetwork/helm-chart/socialnetwork$ helm upgrade social-network .  --reuse-values  --set compose-post-service.replicas=3 -n deathstarbench
Release "social-network" has been upgraded. Happy Helming!
NAME: social-network
LAST DEPLOYED: Tue Apr 22 06:59:25 2025
NAMESPACE: deathstarbench
STATUS: deployed
REVISION: 2
TEST SUITE: None
```

Fig. 18: Command output confirming the scaling of pods to three replicas in the cluster.

GCP Web Dashboard Views and Observations

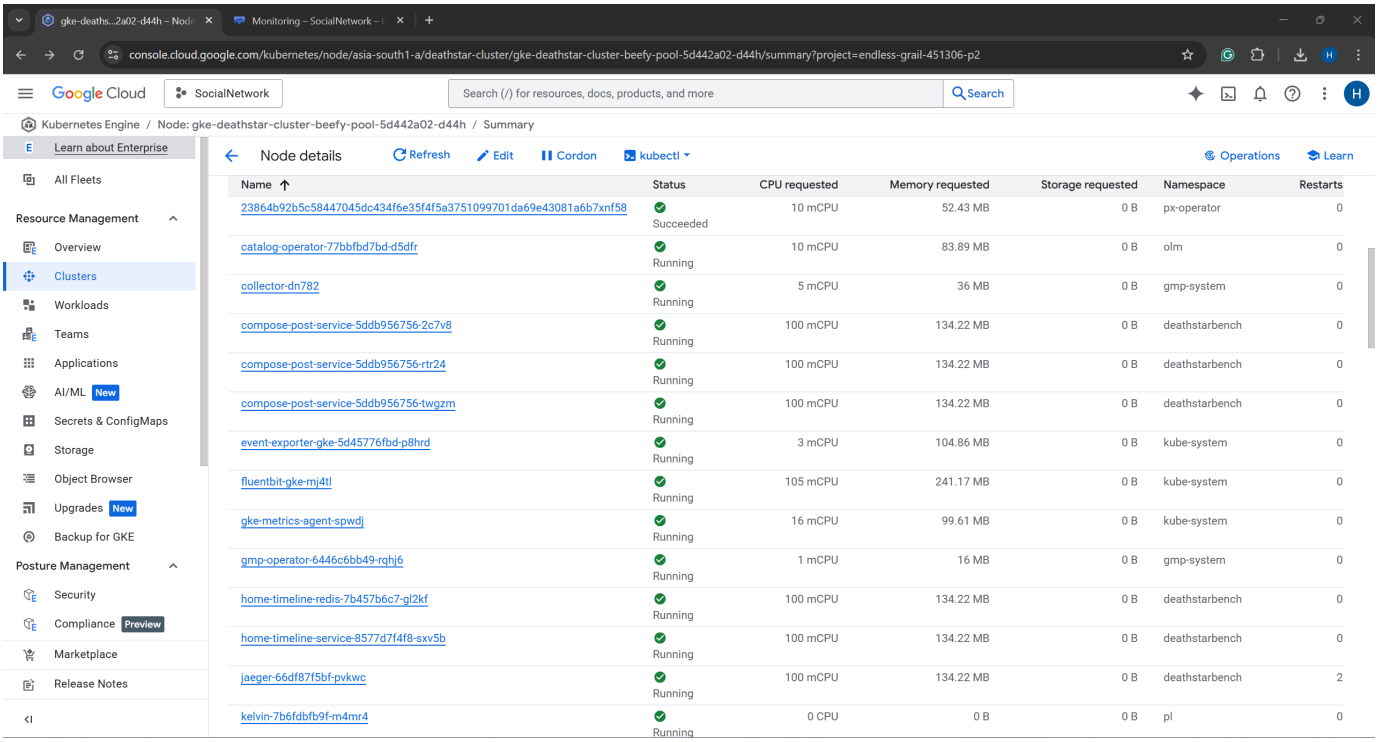


Fig. 19: GCP dashboard view showing the running pods across three replicas.

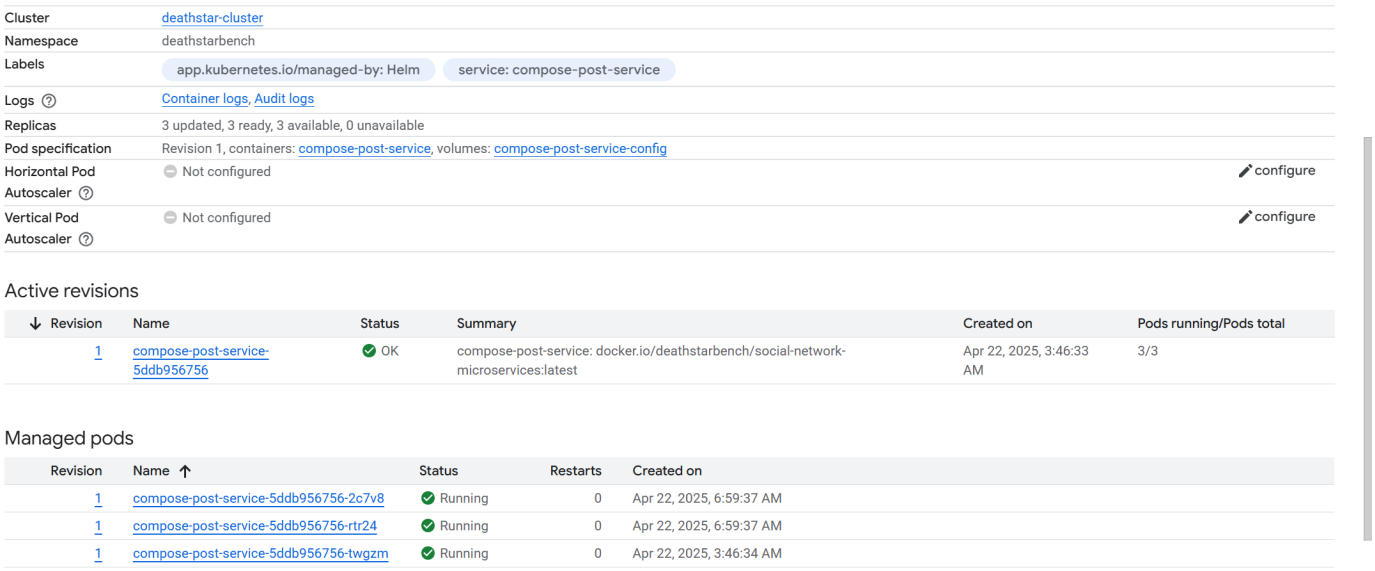


Fig. 20: Managed view of pod replica deployments on GCP indicating successful horizontal scaling.

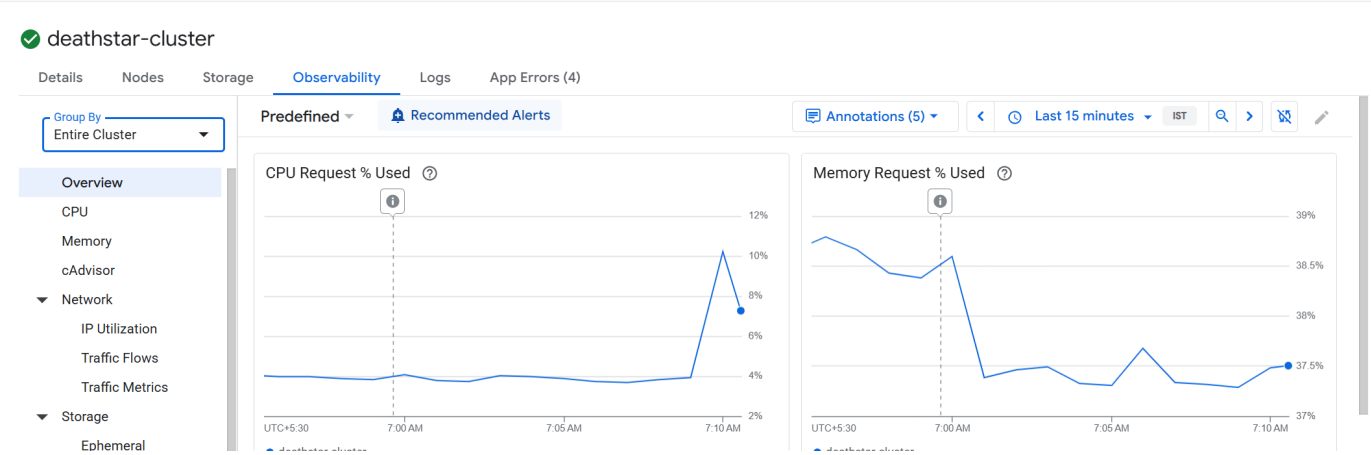


Fig. 21: CPU and memory usage spike of approximately 10% observed during load testing on the cluster.

Replica-wise Load Distribution

The following figures demonstrate the fairly distributed load across each of the three replicas, validating the effectiveness of intra-node load balancing:

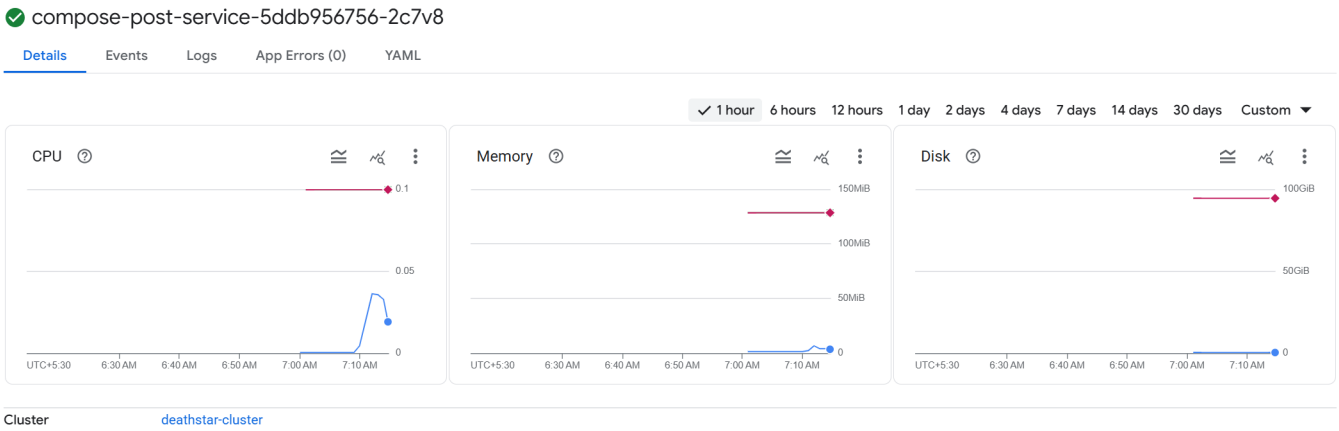


Fig. 22: Monitoring data from Replica 1 indicating consistent CPU and memory utilization.

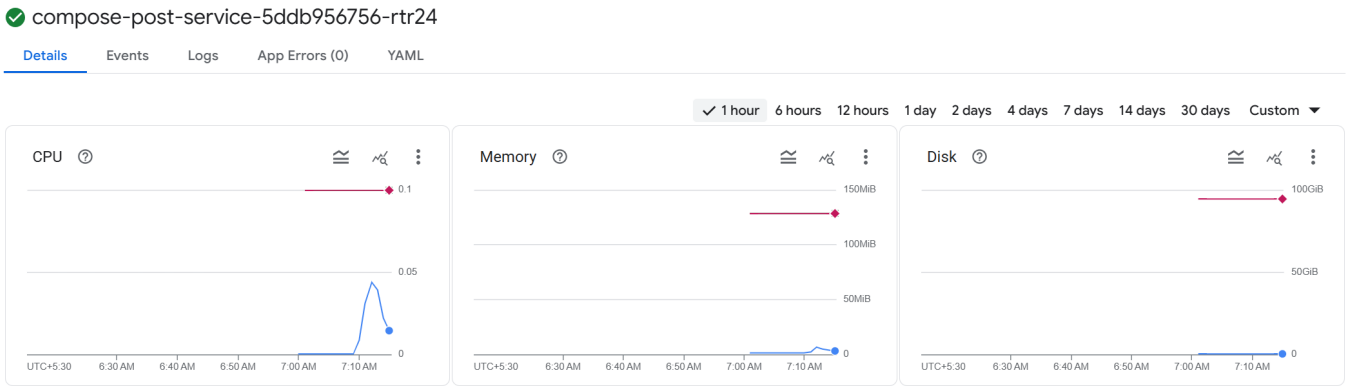


Fig. 23: Monitoring data from Replica 2 reflecting even load distribution.

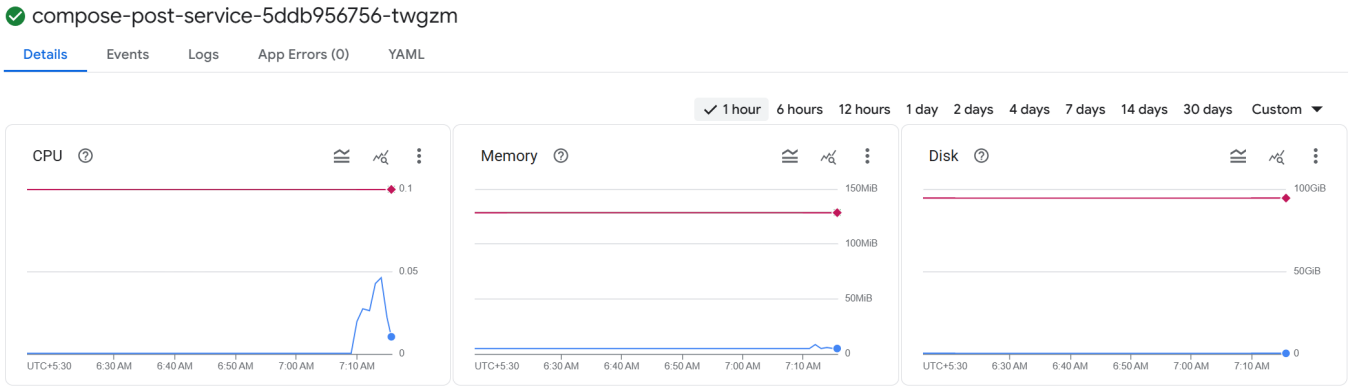


Fig. 24: Monitoring data from Replica 3 demonstrating balanced request handling similar to other replicas.

C. *Three Nodes, Single Replica Each*

In this final configuration, the microservices are deployed across three different nodes, each hosting a single replica. This setup is designed to evaluate inter-node load balancing and the performance benefits of spreading the application across multiple machines.

To test this configuration, we employed a longer-duration load test with a single thread and 8000 connections for a total of 300 seconds, simulating a distributed and high-volume traffic scenario.

Figure 25 presents the deployment view for the multi-node setup with one replica per node.

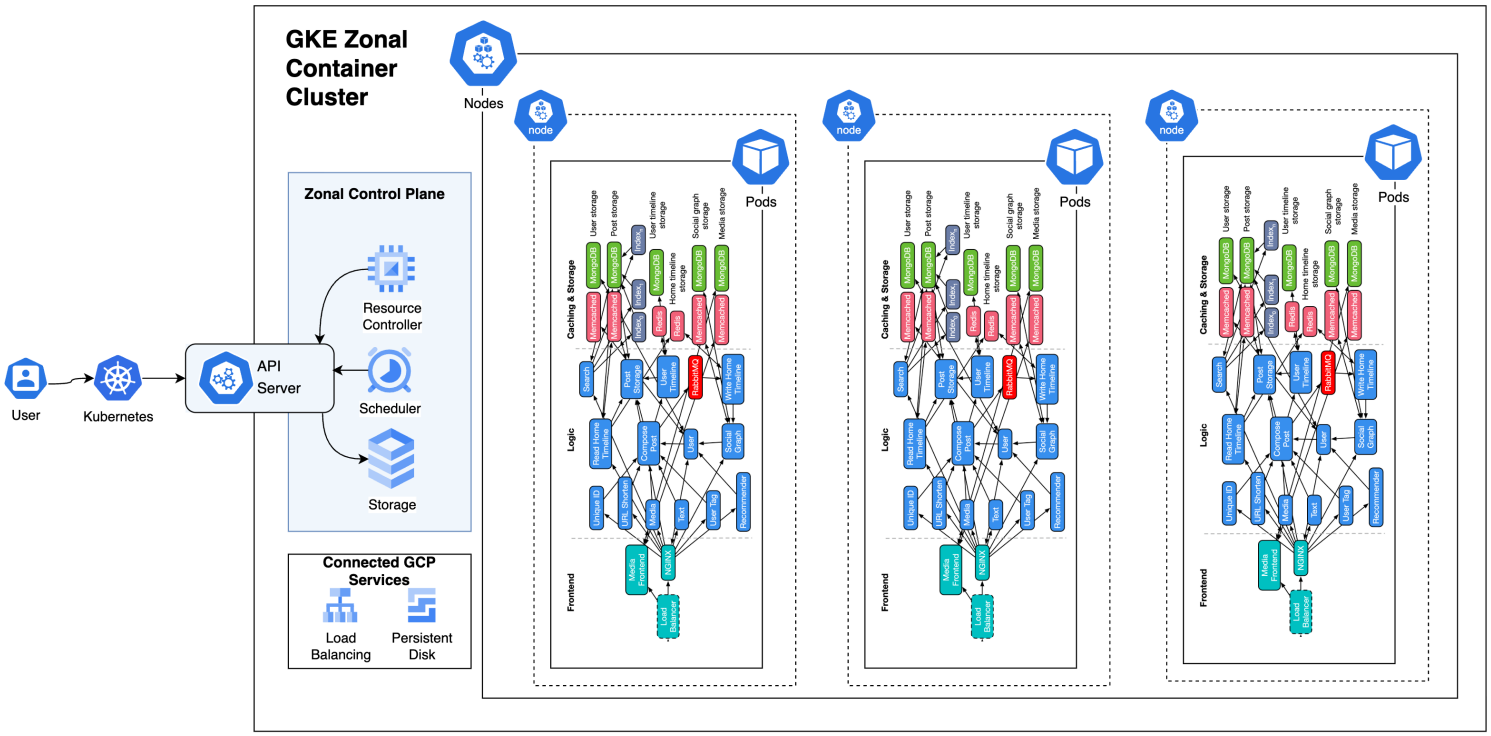


Fig. 25: Deployment view of three-node setup with single replica per node

Plots and graphs

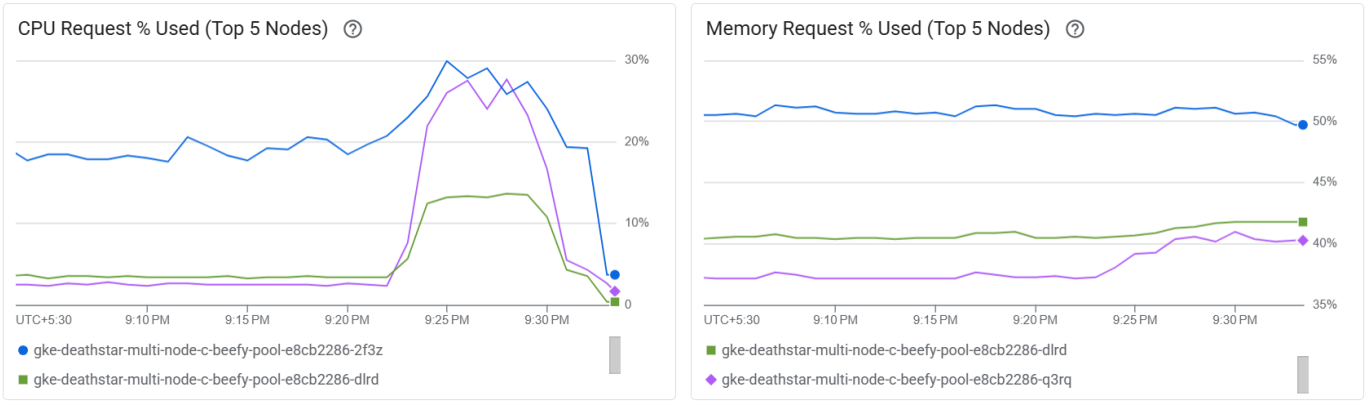


Fig. 26: CPU and memory spikes across three nodes during load testing

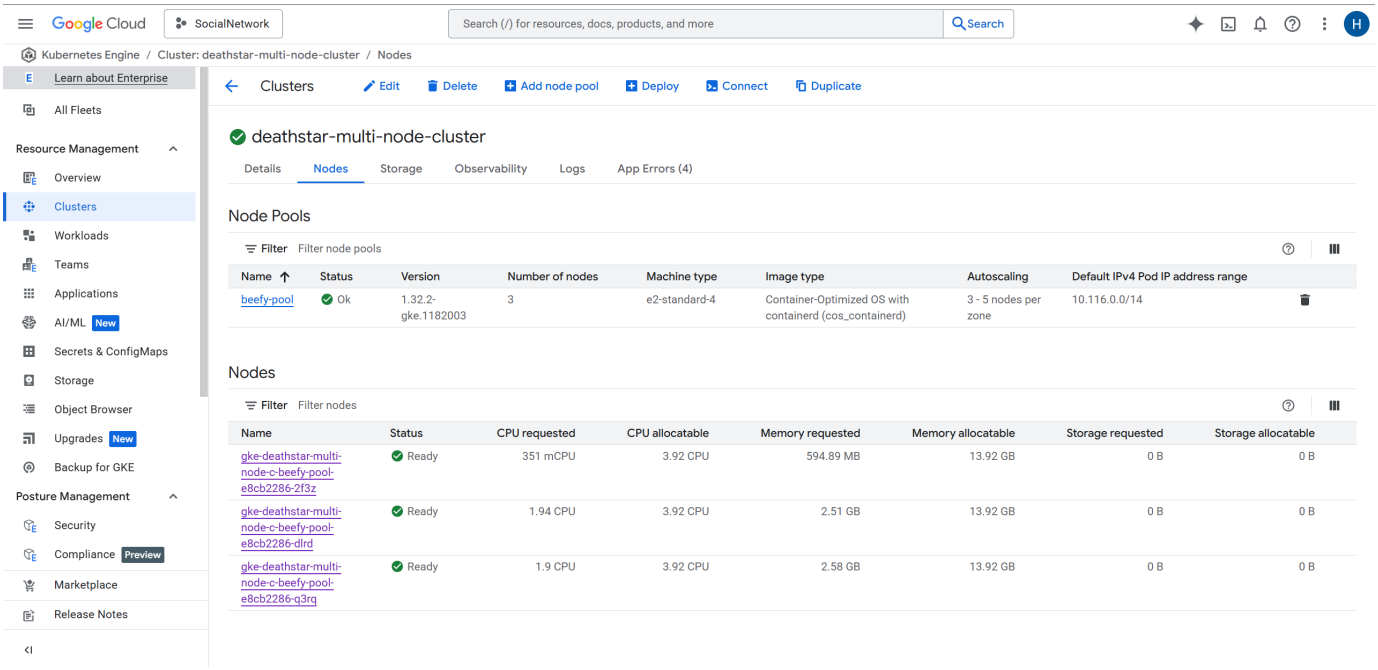


Fig. 27: Multiple node deployment status on GCP

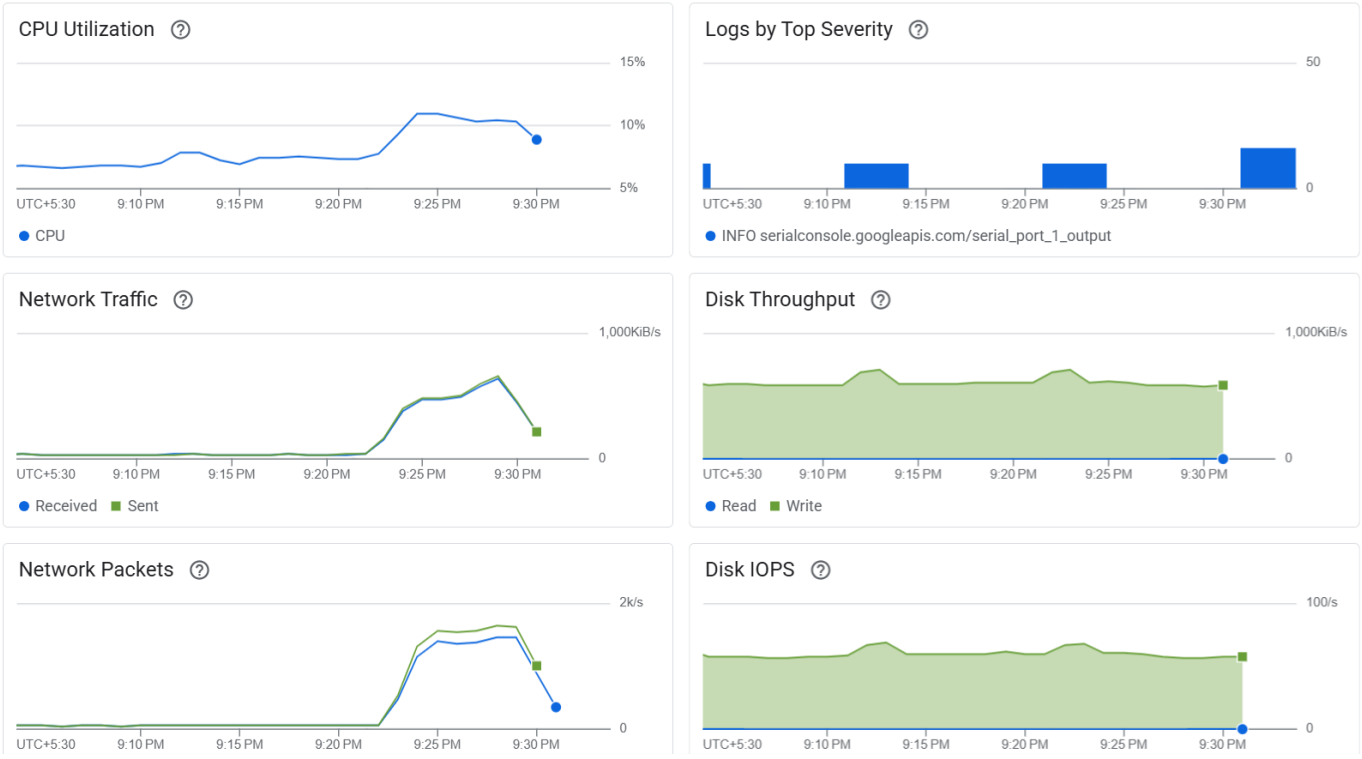


Fig. 28: Resource utilization graph for Node 1 during load testing

The following graphs showcase the individual node spikes in resources during load testing.

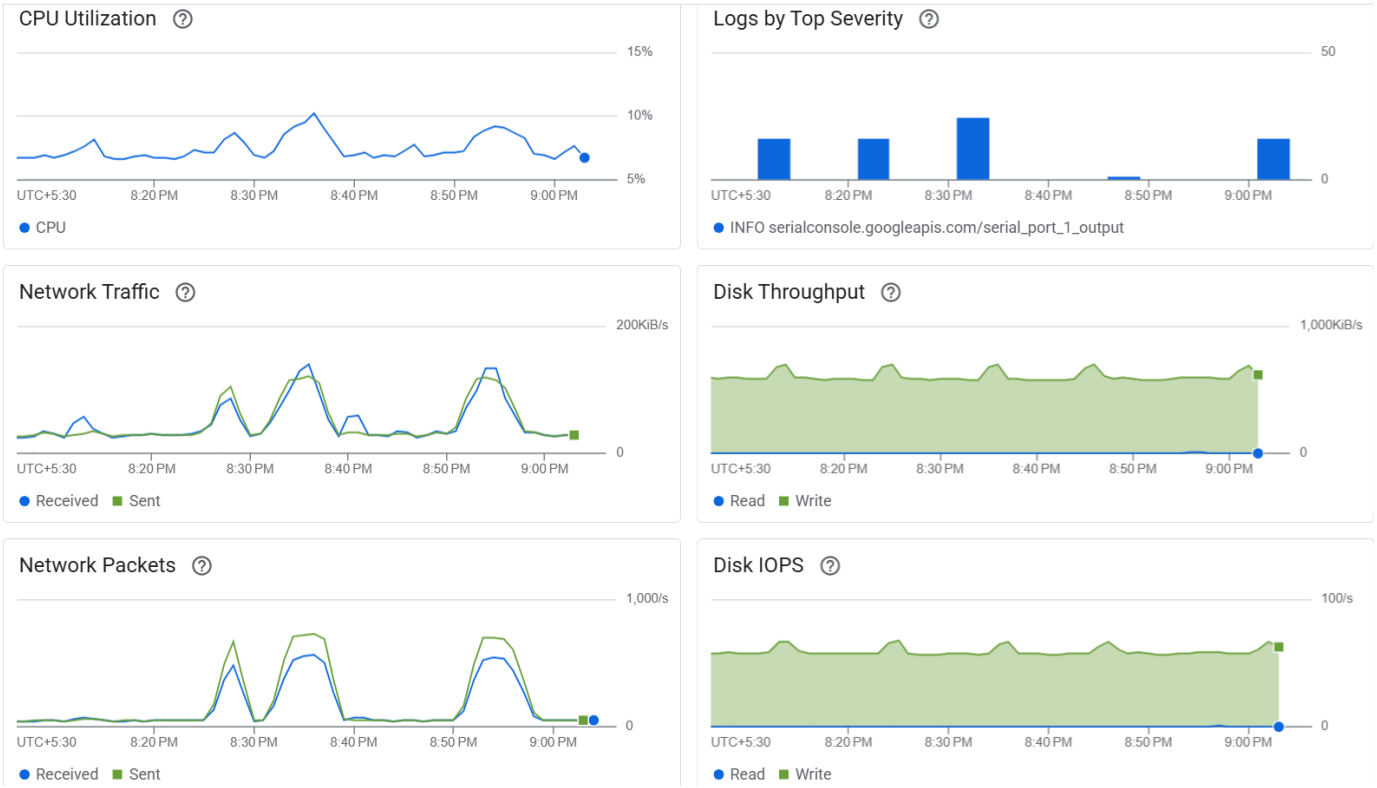


Fig. 29: Detailed metrics for Node 1 performance under load

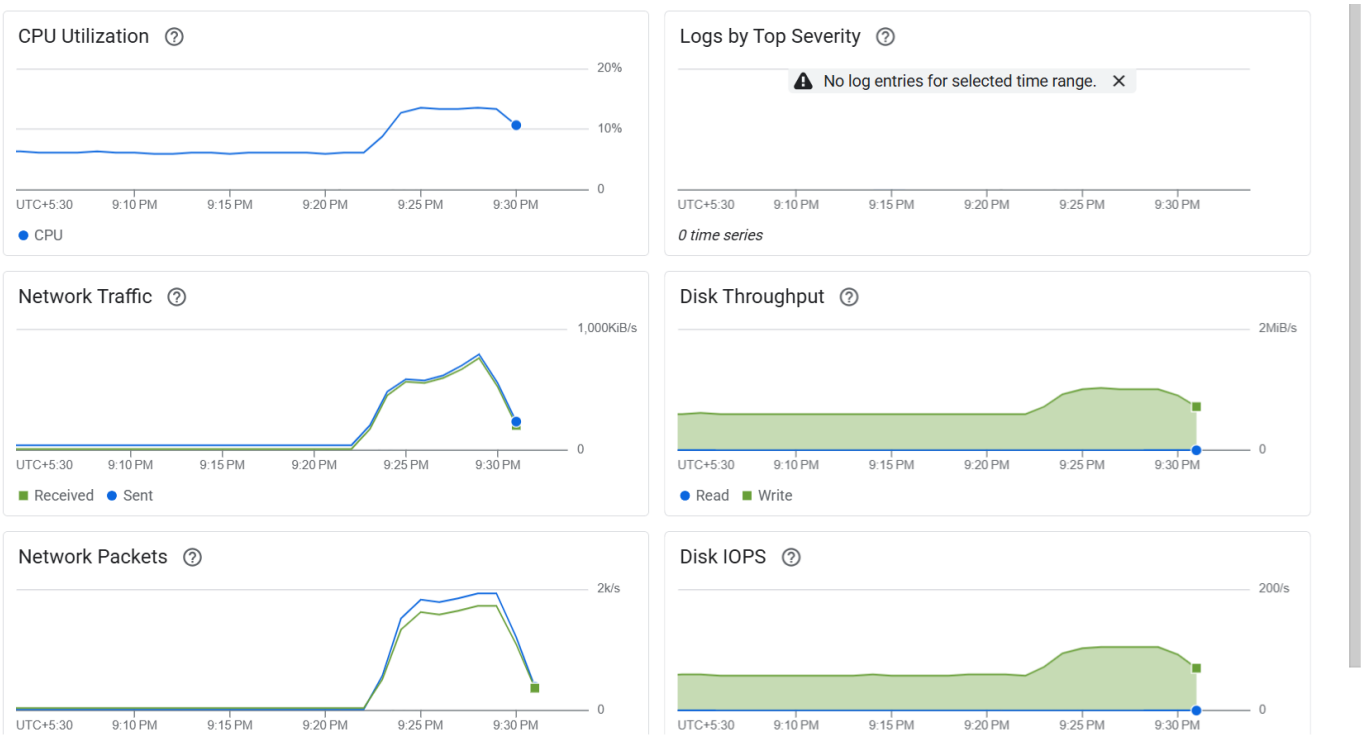


Fig. 30: Resource utilization graph for Node 2 during load testing

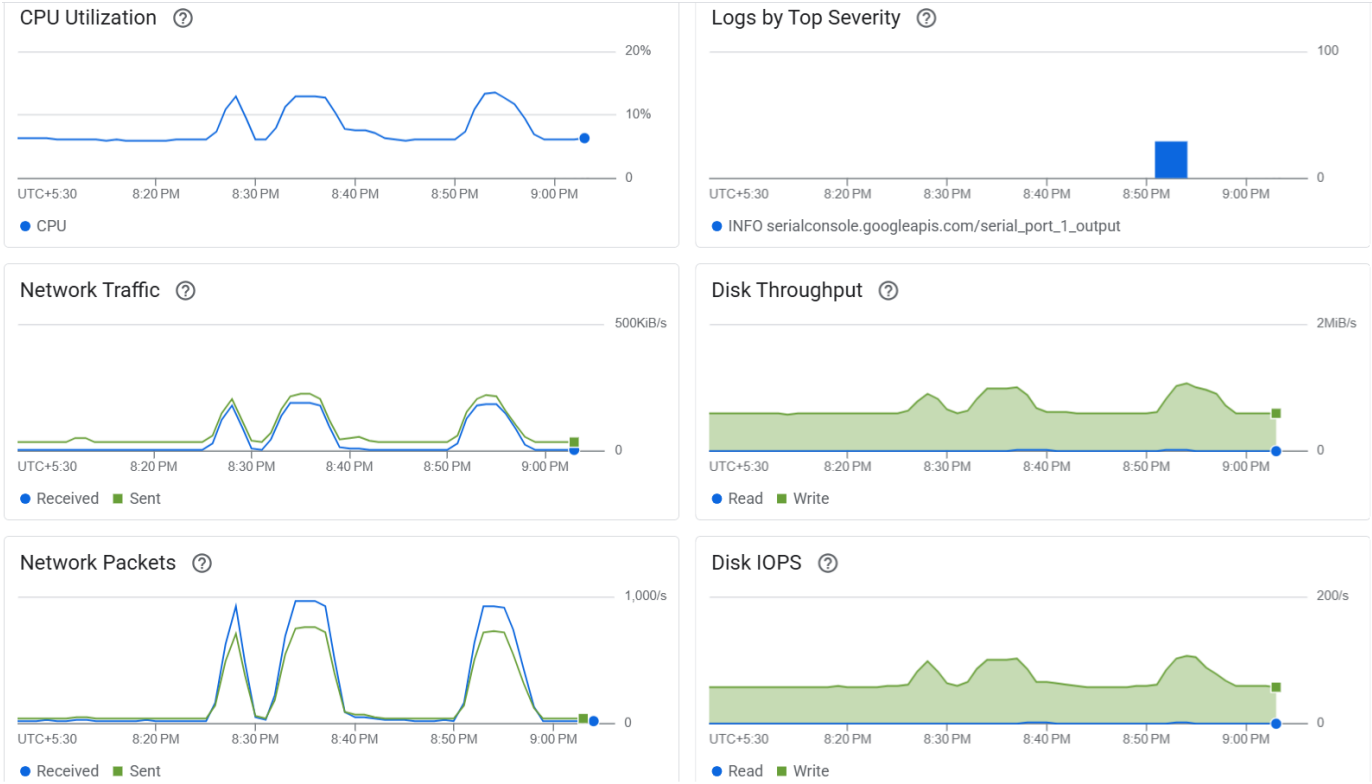


Fig. 31: Detailed metrics for Node 2 performance under load

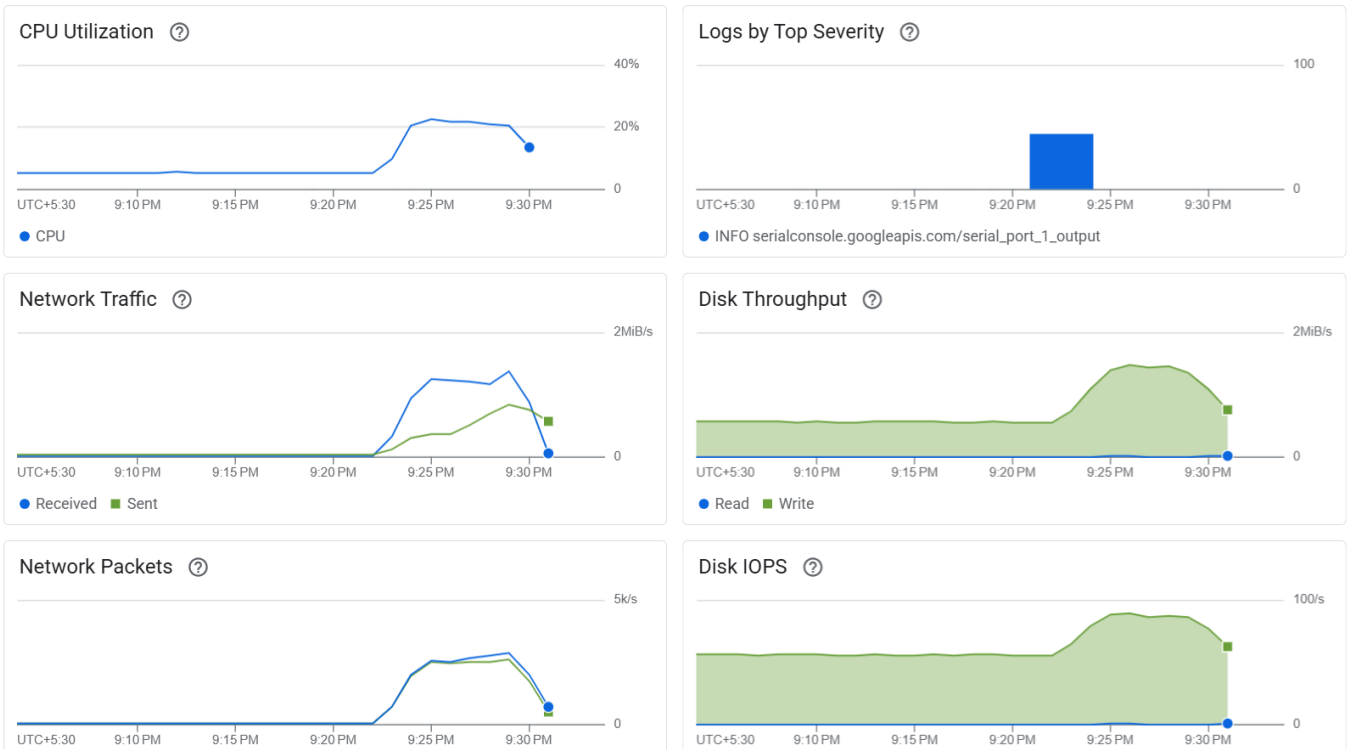


Fig. 32: Resource utilization graph for Node 3 during load testing

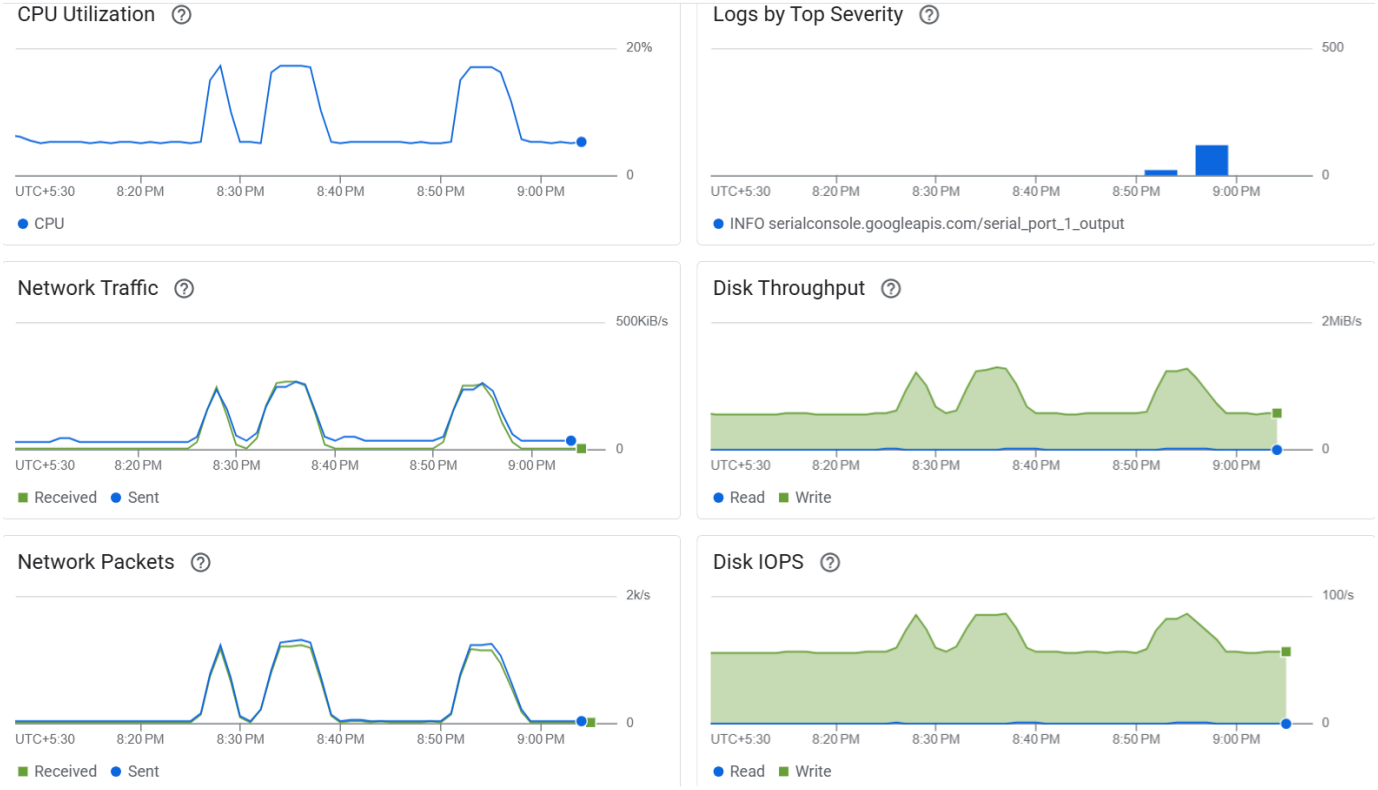


Fig. 33: Detailed metrics for Node 3 performance under load

VII. Key Takeaways

Single Node, Single Replica

- All traffic is directed to a single instance.
- CPU utilization peaks at ∼21% with 10 RPS.
- Suitable for low-traffic applications or prototyping.
- No load balancing is required here, so performance is linear and expected.

Single Node, Three Replicas

- Although three replicas were deployed, unfair CPU distribution was observed: Replica-1: ∼11%, Replica-2: ∼9%, Replica-3: ∼8%
- This imbalance suggests that GCP Load Balancer may not be effectively distributing requests across replicas on the same node.
- Possible cause: Session affinity or round-robin inefficiency when all replicas are in the same pod pool on a single node.
- The total load is light (400 connections over 60s), so imbalance may not cause bottlenecks yet.

Multi Node, Three Replicas

- Requests were distributed across three different nodes.
- RPS reached 100,000+ under a high-volume test (8000 connections for 300s).
- Distribution: Node-1: ∼12%, Node-2: ∼14%, Node-3: ∼21%
- Much better balance than the single-node case, showcasing the advantage of horizontal scaling across nodes.
- The slight skew towards Node-3 might be due to replica readiness, health check lags, or uneven pod-to-node mapping.

TABLE II: Load Testing Configuration and Resource Utilization Summary

Configuration	Threads	Conns.	Dur. (s)	RPS	CPU Utilization	Avg. Util.
Single Node Single Replica	12	400	60	10	∼21%	∼21%
Single Node Three Replicas	12	400	60	50	Replica-1: ∼11% Replica-2: ∼9% Replica-3: ∼8%	∼10%
Multi Node Three Replicas	1	8000	300	100000+ (Distributed)	Node-1: ∼12% Node-2: ∼14% Node-3: ∼21%	∼16%

Why Single Node Three Replicas May Be Imbalanced? Despite using GCP Load Balancer:

- Kubernetes does not guarantee perfectly balanced traffic across pods if they're on the same node.
- If session affinity or sticky sessions are enabled (even by default), certain pods receive more requests.
- GCP's health checks and pod readiness probes may take time to evenly distribute traffic.
- Load balancing at the GCP ingress level may also behave differently for intra-node traffic versus cross-node traffic

VIII. Conclusive Outcomes

Through this project, the DeathStarBench Social Network microservice architecture was successfully deployed, tested, and observed under both **Docker** and **Kubernetes** environments. The migration to Kubernetes enabled a more realistic infrastructure setup, mimicking production-like scalability and observability patterns.

To address the lack of native monitoring and tracing support in the benchmark:

- **Pixie** was integrated into the local Kubernetes deployment to enable **real-time observability** without code instrumentation.
- **Jaeger** was revived with manual header propagation and custom tracing patches, enabling **end-to-end distributed tracing**.
- Observability was extended to **Google Cloud**, where the same services were deployed and monitored using:
  - **Cloud Monitoring** (formerly Stackdriver) for system metrics.
  - **Cloud Logging** for logs aggregation.
  - **Cloud Trace** and **Profiler** for latency analysis and CPU profiling.

Performance Testing & CPU Utilization Patterns

A wide range of **load-testing configurations** were executed using `wrk2`, varying:

- Threads, connections, and request-per-second (RPS) targets.
- Service placements, replicas, and node sizes.
- Kubernetes horizontal scaling and resource limits.

Under these workloads, the following metrics were monitored across both **local (Pixie)** and **cloud (GCP)** deployments:

- **CPU utilization:** Pixie plots, GCP.
- **Memory allocation:** Pixie dashboards, GCP.
- **Network traffic:** Pixie NetFlow view, GCP.
- **Pod-level performance:** Kubernetes Metrics + GCP.
- **Inter-service latency:** Pixie + Jaeger Traces.

These insights revealed clear **utilization patterns**:

- Services like `compose-post`, `user`, and `media` saw **high CPU spikes** under heavy concurrency.
- Backend services maintained more **consistent resource usage** but exposed bottlenecks in I/O.
- Network throughput and response times correlated directly with service chaining and load bursts.
- Cloud observability helped validate these behaviors at scale, revealing differences in auto-scaling efficiency and network overheads.

Overall, the combination of **local and cloud-based observability**, custom engineering fixes, and varied performance tests enabled a deep understanding of system behavior, supporting improved scaling strategies, fault diagnosis, and deployment optimizations.

REFERENCES

[1] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud edge systems." in *ASPLOS*, I. Bahar, M. Herlihy, E. Witchel, and A. R. Lebeck, Eds. ACM, 2019, pp. 3–18. [Online]. Available: http://dblp.uni-trier.de/db/conf/asplos/asplos2019.htmlGanZCSRKBHRJHPH19