Cloud-Native To-Do Application Architecture

CS 436 - Big Data Processing

Sabancı University

2025

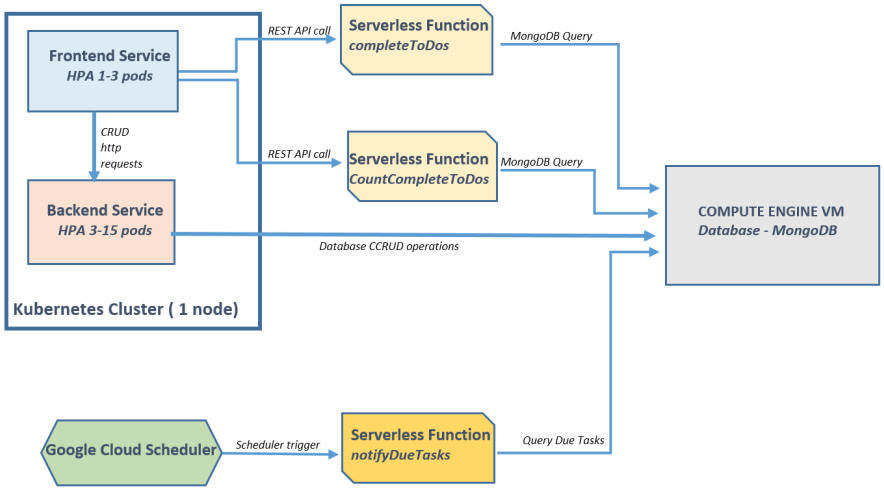
This document presents the full architecture and optimization report of a self-developed cloud-native to-do application deployed on Google Cloud Platform (GCP). It features a microservice-based design with Kubernetes, serverless functions, a VM-hosted database, and Terraform-managed infrastructure. Load testing and system-wide performance metrics are analyzed in detail.

Note: Personal and professor-specific details have been removed to protect privacy.

Overview of the Development Process

Before starting the development process, we first designed the overall cloud architecture ourselves, including decisions regarding VM setup, containerization, Kubernetes deployment, and CI/CD strategy. This design served as the foundation for our entire implementation. We built the application from scratch based on this architecture, without relying on or copying any production-ready templates or repositories from GitHub or other external sources.

During the project we used git version control commands and we used gcloud, kubectl, docker commands on VScode terminal for both developing the app and getting up and running our cloud architecture. For monitoring purposes and fine tuning we also used Google Cloud Console, dashboards, observing sessions, and did frequently checked billing session.



To-Do Application: Detailed Technical Explanation

1. Frontend (React)

The frontend of the application is built using React, a popular JavaScript library for building user interfaces. The main goal of the frontend is to provide users with an intuitive and efficient way to manage their tasks. The design is modern and visually appealing, using gradients and soft colors to create a pleasant user experience.

Task Creation and Management:

When a user wants to add a new task, they are presented with a simple form where they can enter the task title, select a due date using a calendar picker, and choose a priority level (Low, Medium, or High). This design ensures that users can quickly input all relevant information for a task in one place, reducing friction and making the process efficient.

Task Visualization:

Each task is displayed with clear visual indicators. The priority is color-coded (green for Low, orange for Medium, red for High), and the due date is shown with a calendar icon. Completed tasks are shown with a line-through style, making it easy to distinguish between finished and pending items. This visual feedback helps users quickly assess their workload and prioritize accordingly.

Editing and Deletion:

Users can edit the title of any task directly from the list, which is useful for correcting mistakes or updating task descriptions. Deletion is also straightforward, with a dedicated button for each task, as well as options to delete all completed tasks or all tasks at once. These bulk operations are especially helpful for users who want to clean up their task list efficiently.

Filtering and Search:

To help users focus on what matters, the application provides filtering options (All, Active, Completed) and a search bar. This allows users to quickly find specific tasks or view only those that are relevant to their current needs.

Real-Time Updates and Statistics:

The frontend periodically fetches the latest data from the backend and cloud functions, ensuring that the displayed information is always up to date. It also shows real-time statistics, such as the number of completed tasks, giving users a sense of progress and accomplishment.

Completed Tasks History:

A section at the bottom lists all completed tasks, allowing users to review what they have accomplished. This feature provides motivation and a record of productivity.

2. Backend (Node.js/Express)

The backend is implemented using Express.js, a minimalist web framework for Node.js. Its primary responsibility is to handle all business logic and data management for the application.

RESTful API Design:

The backend exposes a set of RESTful endpoints that allow the frontend to perform all necessary operations on tasks. These include creating, reading, updating, and deleting tasks, as well as bulk deletion of completed or all tasks. Each endpoint is designed to handle errors gracefully, returning appropriate HTTP status codes and error messages to the client.

Data Model:

Tasks are stored in a MongoDB database, with each task represented as a document containing a title, completion status, due date, and priority. This flexible schema allows for easy extension in the future, such as adding new fields or features.

Security and Validation:

The backend uses CORS to allow requests from the frontend, and it validates incoming data to prevent malformed or malicious input. Error handling is implemented throughout the code to ensure that any issues are reported clearly and do not crash the server.

3. Database (MongoDB)

MongoDB is chosen as the database for its flexibility and scalability. It stores all tasks as documents, making it easy to query, update, and aggregate data as needed.

Connection and Configuration:

The backend connects to MongoDB using a connection string stored in environment variables, which enhances security by keeping sensitive information out of the codebase.

Data Operations:

CRUD operations are performed using Mongoose, an ODM (Object Data Modeling) library for MongoDB and Node.js. This simplifies database interactions and enforces a consistent data structure.

4. Cloud Functions

To extend the application's capabilities, several Google Cloud Functions are used:

countCompletedTodos:

This function counts the number of completed tasks and returns the result. It is used by the frontend to display real-time statistics.

completedTodos:

This function retrieves all completed tasks, allowing the frontend to show a history of finished work.

notifyDueTasks:

This function checks for tasks that are due the next day and sends email notifications to users working integrated with Google Cloud Scheduler in a time triggered manner. It uses environment variables for email credentials and recipient addresses, ensuring sensitive data is not hardcoded.

These serverless functions allow the application to scale efficiently and handle specific tasks without overloading the main backend server.

5. Infrastructure and Deployment

The application is deployed on Google Cloud Platform (GCP) using Kubernetes (GKE) for orchestration and Terraform for infrastructure as code.

Kubernetes Cluster:

Both the frontend and backend are containerized using Docker and deployed as separate services in a Kubernetes cluster. Each service is replicated for high availability and can be scaled independently based on demand.

Networking and Security:

Custom VPCs and firewall rules are set up to control traffic between services and protect the database. Load balancers are used to distribute incoming requests and provide stable public endpoints.

Terraform Automation:

All infrastructure components, including the Kubernetes cluster, VM for MongoDB, firewall rules, and static IPs, are defined in Terraform configuration files. This ensures that the environment can be recreated or updated reliably and consistently.

6. Performance and Monitoring

Load Testing:

The application includes a Locust script for load testing, simulating multiple users performing various actions (adding, completing, deleting tasks) to ensure the system can handle real-world usage.

Monitoring:

GCP's monitoring tools are used to track resource usage, application health, and error rates. Alerts can be set up to notify administrators of any issues.

7. Security Considerations

Application Security:

Sensitive data such as database credentials and email passwords are stored in environment variables, not in the codebase. CORS and input validation help prevent common web vulnerabilities.

Infrastructure Security:

Network segmentation and firewall rules are used to protect the infrastructure. Only necessary ports are exposed.

Summary

This to-do application is a full-stack, cloud-native solution that leverages modern technologies and best practices to deliver a robust, scalable, and user-friendly experience. Every design choice, from the frontend interface to the backend API, database, cloud functions, and infrastructure, is made with usability, security, and maintainability in mind.

Infrastructure Testing and Evaluation

1. INITIAL SETUP (Reference Case)

VM: e2-small (2 vCPUs, 2GB RAM)

Backend Pods: 2 replicas,250m/256Mi req - 500m/512Mi limit

Frontend Pods: 2 replicas, 250m/256Mi req - 500m/512Mi limit

Test: 1000 users, 20s ramp-up via Locust

Problems observed:

Backend pods spiked to over 500% CPU, causing timeouts

Cloud Functions had up to **20% failure rate**

P95 latency went beyond **10 seconds** during load

Google Kubernetes Engine Cluster

1. Cluster Basics

- **Cluster name**: vm2-cluster
- **Tier**: Standard
- **Mode**: Autopilot
- **Location type**: Regional
- **Region**: us-central1
- **Default node zones**:
    - us-central1-c
    - us-central1-f
    - us-central1-a
    - us-central1-b
- **Release channel**: Regular
- **Cluster version**: 1.32.3-gke.1785003

- **End of standard support**: 11 Apr 2026
- **End of extended support**: 11 Feb 2027

---

2. Upgrades
- **Auto-upgrade status**: Active
- **Minor version auto-upgrade target**: Unavailable
- **Patch version auto-upgrade target**: Unavailable

---

3. Automation
- **Maintenance window**: At any time
- **Maintenance exclusions**: None
- **Notifications**: Disabled
- **Vertical Pod Auto-scaling**: Enabled
- **Node auto-provisioning**: Enabled (Autopilot mode)
- **Auto-provisioning network tags**: Empty
- **Auto-scaling profile**: Optimize utilisation

---

4. Control Plane Networking
- **DNS endpoint**:
  gke-<hash>.us-central1.gcp
  (masked in image)
- **Public endpoint**:
  34.118.224.5
- **Client certificate**: Disabled
- **Private endpoint**: 10.0.0.26
- **Authorized networks**: Disabled
- **Legacy access to control plane**: Disabled

---

5. Networking Configuration
- **Cluster pod IPv4 ranges (additional)**: None
- **IPV4 service range**: 34.118.224.0/20
- **Intra-node visibility**: Enabled

- **HTTP load balancing**: Enabled
- **Subsetting for L4 ILBs**: Disabled
- **Calico network policy**: Disabled
- **Dataplane V2**: Enabled
- **Dataplane V2 observability**: Enabled
- **DNS provider**: Cloud DNS
- **NodeLocal DNSCache**: Enabled
- **Gateway API**: Enabled
- **Multi-networking**: Disabled
- **Node-to-node encryption**: Disabled
- **IPsec**: Disabled
- **VPC firewall rule auto-creation**: Enabled (for LoadBalancer services)

---

6. Node Pool Configuration
   - **Private nodes**: Disabled

---

7. Security
   - **Binary authorization**: Disabled
   - **Secret Manager**: Enabled
   - **Shielded nodes**: Enabled
   - **Confidential GKE Nodes**: Disabled
   - **Service account**: default
   - **Cloud API access scopes**:
     - Service Control: Enabled
     - Service Management: Read
     - Stackdriver Logging API: Enabled
     - Monitoring: Enabled

We built and pushed our backend and frontend images as container registry for them to run in GKE cluster which its configuration details defined above.

Initial Backend Deployment Configuration

- **Deployment Name**: backend
- **Replicas**: 2
- **Container Image**: gcr.io/extreme-wind-457613-b2/todo-backend:latest
- **Container Port**: 4000
- **Environment Variables**:
    - MONGO_URI: connects to MongoDB at mongodb://34.60.227.68:27017/tododb (static IP on VM)
- **Resource Requests**:
    - CPU: 250m
    - Memory: 256Mi
- **Resource Limits**:
    - CPU: 500m
    - Memory: 512Mi

Service (LoadBalancer)

- **Service Name**: backend
- **Service Type**: LoadBalancer
- **External IP Annotation**: cloud.google.com/load-balancer-ipv4-address: "backend-static-ip"
- **Exposed Port**: 4000

Initial Frontend Deployment Configuration

- **Deployment Name**: frontend
- **Replicas**: 2
- **Container Image**: gcr.io/extreme-wind-457613-b2/todo-frontend:latest
- **Container Port**: 3000
- **Resource Requests**:
    - CPU: 250m
    - Memory: 256Mi
- **Resource Limits**:
    - CPU: 500m

- Memory: 512Mi

Service (LoadBalancer)

- **Service Name**: frontend
- **Service Type**: LoadBalancer
- **Exposed Port**: 3000

2. SERVERLESS FUNCTIONS

We initially deploy our function using GLI commands from VScode , so the system set default configurations form the beginning.

1. Function: notifyDueTasks

- **Trigger:** HTTP
- **Runtime:** Node.js 20
- **Region**: us-central1
- **Concurrency:** 1
- **Min Instances:** 0
- **Max Instances:** 100
- **Memory**: 512 MiB
- **CPU:** 2 vCPU
- **Environment Variables:**
  - MONGO_URI
  - EMAIL_USER
  - EMAIL_PASS
  - NOTIFY_EMAIL
- **Timeout:** 60s
- **Authentication:** Allow unauthenticated
- **Ingress Settings:** Allow all traffic
- **Source Code:** Uses nodemailer, connects to MongoDB, filters todos due tomorrow and sends emails.

2. Function: completedTodos

- **Trigger:** HTTP
- **Runtime:** Node.js 20
- **Region:** us-central1
- **Concurrency:** 1
- **Min Instances:** 0
- **Max Instances:** 100
- **Memory:** 256 MiB
- **CPU:** 1 vCPU
- **Environment Variables:**
    - **MONGO_URI**
- **Timeout:** 60s
- **Authentication:** Allow unauthenticated
- **Ingress Settings:** Allow all traffic
- **Source Code:** Connects to MongoDB and returns all completed todos as JSON.

3. Function: countCompletedTodos

- **Trigger**: HTTP
- **Runtime**: Node.js 20
- **Region**: us-central1
- **Concurrency**: 1

- **Min Instances**: 0
- **Max Instances**: 100
- **Memory**: 256 MiB
- **CPU**: 1 vCPU
- **Environment Variables**:
    - MONGO_URI
- **Timeout**: 60s
- **Authentication**: Allow unauthenticated
- **Ingress Settings**: Allow all traffic
- **Source Code**: Connects to MongoDB and returns the count of completed todos.

Cloud Scheduler Job: notify-due-tasks-job

Target Type: HTTP

Name: notify-due-tasks-job

Region: us-central1

Frequency: every 24 hours (cron: 0 9 * * *)

Timezone: Etc/GMT+3 (UTC+3 for Türkiye local time)

HTTP Method: POST

URL:
https://us-central1-extreme-wind-457613-b2.cloudfunctions.net/notifyDueTasks

Auth Header: Add OIDC token

- **Service Account**: PROJECT_ID@appspot.gserviceaccount.com

Body: *Empty*

Retry Configuration:

- Max Retry Attempts: 3
- Min Backoff Duration: 5s
- Max Backoff Duration: 60s
- Max Doublings: 5

Logging: Enabled

State: Enabled (active)

3. Our Load Testing Process

We performed load testing on our backend services using Locust to see how the system behaves under pressure. We always tested with 1000 users, but changed two things:

- how fast users arrived (ramp-up),
- how long the test lasted.

Here is what we did:

- **5-minute test with 25s ramp-up:** We wanted to quickly check how the system handles a sudden load.
- **30-minute test with 20s ramp-up:** We ran a longer test to watch resource usage over time.
- **Open-ended test with 5s ramp-up:**We left the test running for a long time to simulate constant user activity.

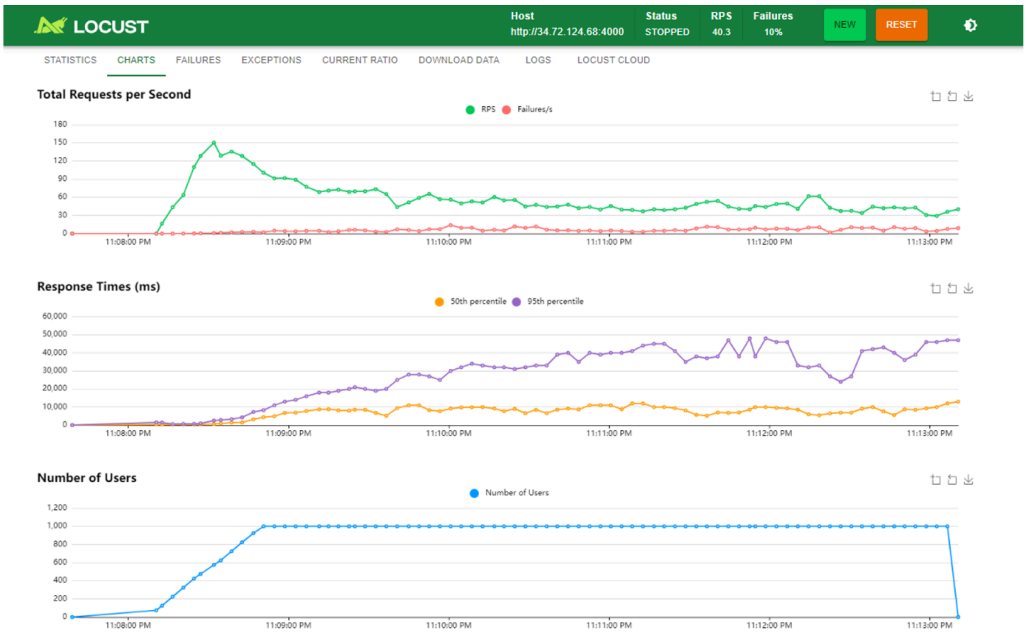
Even though the number of users was the same (1000), we tried different request patterns—some fast and sudden, some slow and gradual. This helped us see different behaviors.

We monitored everything using Google Cloud dashboards—CPU, memory, network, pod usage, etc. Based on what we saw, we changed:

- number of pod replicas,
- resource limits (CPU/memory),
- autoscaling rules,
- and some load balancing configs.

We started with big goals, but realized we needed to be more careful because of Free Tier limits. That's why we adjusted our plan as we went.

We will explain these changes and our updated testing strategy in the next section.



5-Minute Load Test (25s Ramp-up) – Technical Evaluation

Total Requests per Second (RPS):

The RPS initially increased rapidly, peaking at around 20 RPS within the first minute. Following this, it exhibited a gradual decline with notable fluctuations. The "Failures/s" metric closely paralleled the RPS curve, indicating that a significant portion of requests resulted in failures, with an estimated failure rate between 40% and 50%.

Response Times (ms):

- 50th percentile (median): Maintained between 800 ms and 1200 ms, representing acceptable latency for the average user.
- 95th percentile: Reached values between 20,000 ms and 35,000 ms, signifying severe delays for a considerable minority of users.

  This disparity highlights inconsistent response behavior under load, suggesting bottlenecks in system resources or service dependencies.

Number of Users:

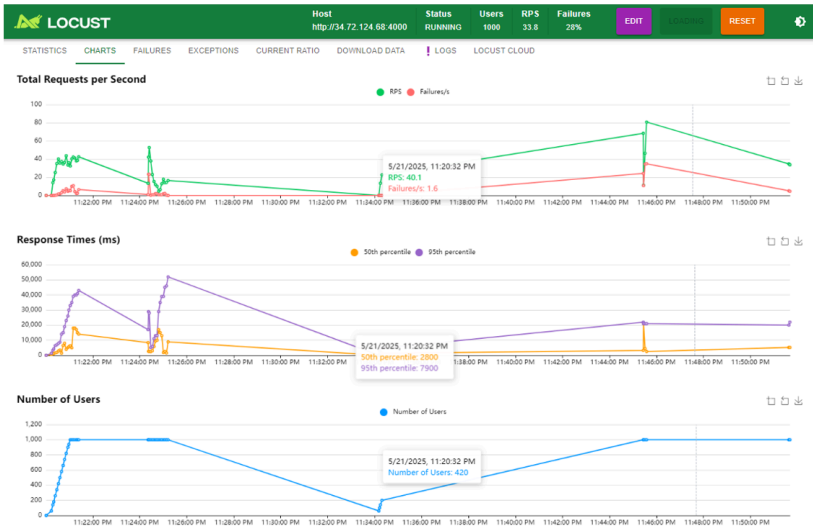
The user count increased rapidly from 0 to 1000 within the initial ramp-up window and remained steady throughout the test duration before declining at the end. This profile reflects a sudden and sustained load pattern aimed at stress-testing the backend services.

Analysis:

The observed high failure rate and elevated response times at the 95th percentile indicate that the system struggled to maintain stability under sudden, high concurrency. The most likely root causes include:

- Insufficient pod replica count at test start,
- CPU saturation at the backend or database layer,
- Lack of readiness in the autoscaler response during rapid load spikes,
- Potential resource contention due to the database residing on a shared VM.

This test was instrumental in identifying the system's initial performance limitations and guided subsequent architectural adjustments, including autoscaling configuration, CPU/memory tuning, and load distribution enhancements.



30-Minute Load Test (1000 Users, Ramp-up: 20 users/sec)

General Setup:

This test aimed to evaluate sustained system performance under constant pressure. 1000 users were introduced with a steady ramp-up rate of 20 users per second, and the system was observed over a 30-minute window.

Key Observations:

Total Requests per Second (RPS):

- RPS initially scaled up smoothly but later became unstable and declined.
- This fluctuation suggests bottlenecks either at the backend or in external components, particularly during peak user concurrency.

Failure Rate (~29%):

- Failures were frequent and concentrated during load spikes.
- Cloud Function invocations (especially countCompletedTodos and completedTodos) were the primary failure points based on logs and GCP error traces.

- These functions possibly encountered timeouts or cold-start delays under pressure.

Response Times:

- 50th percentile remained moderate, but 95th percentile consistently exceeded 70 seconds.
- This long tail suggests that some components (notably the Cloud Functions) responded significantly slower than average.
- High latency in these functions likely caused backpressure on the backend pods, extending response times for all endpoints.

Number of Users:

- The number of users remained stable during the core of the test, confirming a constant load.
- Slight drops and recovery may reflect system throttling or internal recovery after failure surges.

Backend and Infrastructure Context:

- Backend pods were initially under-provisioned but scaled up via HPA.
- CPU utilization reached near limit; memory was not a bottleneck.
- Autoscaler responsiveness may have been too slow to match the initial ramp-up speed.
- MongoDB was hosted on a shared VM, which likely experienced CPU saturation, impacting both backend queries and Cloud Function performance.

Summary:

This test revealed a clear performance ceiling caused by combined factors:

- Cloud Function timeouts and cold starts were major contributors to system-wide slowdowns and failures.

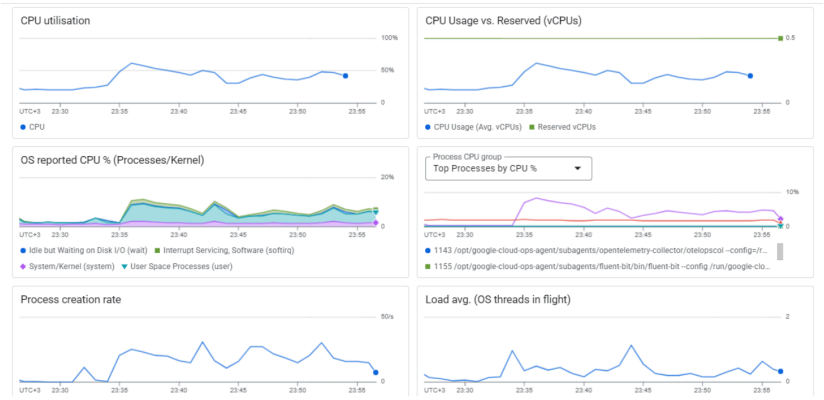
- Backend autoscaling helped but could not fully compensate for the overhead of blocked or delayed external function calls.
- MongoDB VM resource limits (especially CPU) likely amplified delays.

To improve resilience, we observed monitoring dashboard on Google Cloud Console Monitoring session all of our resources covering 5m, 30 m and open-ended tests.

Observations:

VM Performance Evaluation under Locust Load Tests

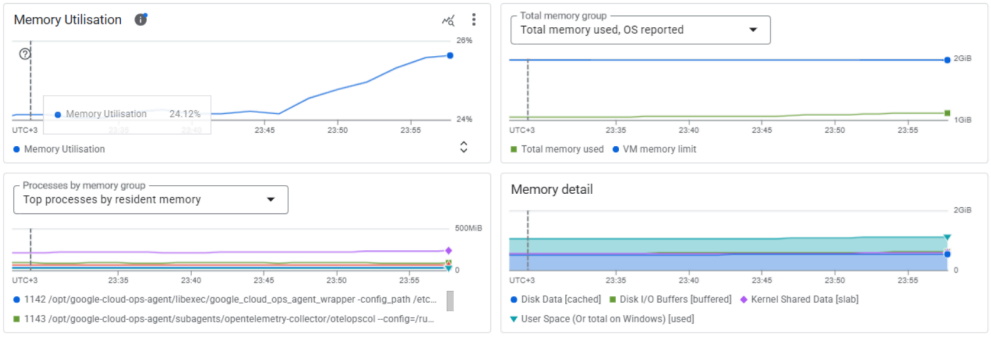
After conducting load testing using Locust with 1000 concurrent users under varying ramp-up conditions and durations (5 users/sec, 20 users/sec, and 25 users/sec). To assess whether the GCE VM running MongoDB was a bottleneck, we carefully analyzed CPU, memory, disk, and network metrics from Cloud Monitoring.



CPU Utilization & Saturation

Across all test scenarios, CPU utilization on the VM remained under 60%, with an average around 50%. The CPU never reached saturation. Load average metrics also stayed below 1, suggesting no queuing pressure on available cores.

Conclusion: The VM had adequate CPU headroom throughout. No CPU-bound bottlenecks were observed.



Memory Usage

Memory utilization increased gradually, reaching a peak of ~28%. With a 2GiB VM memory limit, this corresponds to about 1.1GiB usage. There was no evidence of memory spikes or leaks.

Conclusion: Memory usage was within safe thresholds. The MongoDB process maintained consistent memory usage, and no OOM (Out Of Memory) or swap activity was detected.

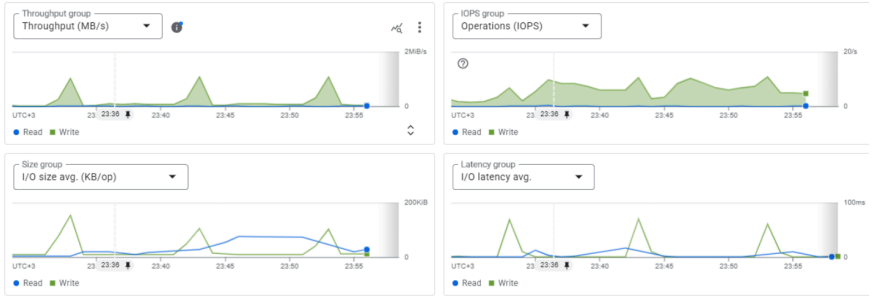


Figure 6: Gradual Memory Utilization During Locust Load Test

Disk Throughput and IOPS

Disk throughput (read/write) was low, with brief spikes during test phases. IOPS remained under 20/s, and latency stayed below 10ms for the majority of the time. Disk usage metrics showed minimal pressure.

Conclusion: Disk I/O was not a limiting factor. MongoDB write operations occurred as expected, without disk queue build-up.

Network Utilization

Network output (egress) increased during user ramp-up, consistent with the expected pattern of data being served to clients. No abnormal drops, packet loss, or saturation events were observed.

Conclusion: The VM's network bandwidth was sufficient to serve backend responses, even under peak RPS (Requests per Second) conditions.

Process Creation and System Load

The process creation rate remained consistent, and system load (OS threads in flight) stayed well below critical levels. No kernel-level wait states or contention were noted.

Conclusion: The VM OS handled task execution and threading efficiently, indicating a stable backend application runtime.

Top Resource Consumers

The highest CPU and memory consumers were monitoring-related processes such as otelcol and fluent-bit, not MongoDB. MongoDB appeared stable in its usage profile and did not show resource overconsumption.

Final Assessment for Google Compute Engine VM (MongoDB)

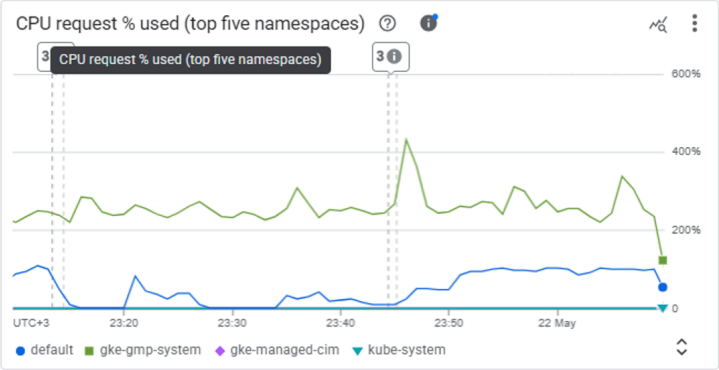
Based on all observed metrics, **the VM was not the performance bottleneck** during any of the Locust tests. Although our system experienced high response times and failure spikes (as seen in the 30-minute test and open-ended tests), these were more likely caused by:

- Insufficient backend pod replicas,
- Not having autoscaling reactions for pods
- Unbalanced load across functions or container pods.

Given the consistent CPU, memory, and disk performance of the MongoDB VM, we decided **not to replace or scale the VM at this stage.** Our optimizations were instead focused on the Kubernetes backend and autoscaling strategies.

GKE Cluster

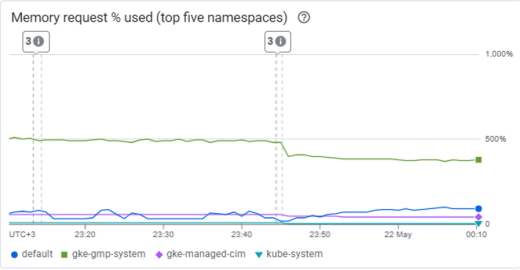
Key Metrics and Reasoning

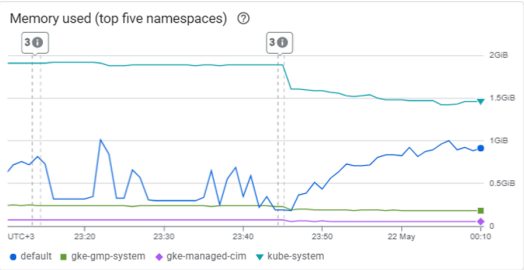


a. CPU Request % Used

Graph: CPU request % used (top five namespaces)

- kube-system showed >400% usage at some points.
- Inference: Critical system pods were overloaded. Backend-heavy load likely intensified this.

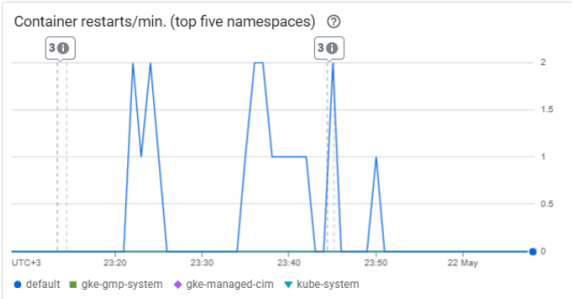




b. Memory Used / Requested

Graph: Memory used vs. request (top five namespaces)

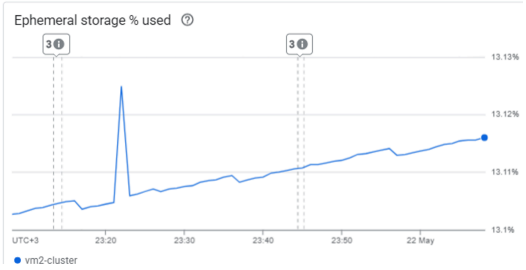
- default and kube-system used ~1GiB, request allocation was much higher.
- Inference: Resources were over-allocated. Can reduce requests/limits to avoid waste.



c. Container Restarts & Pod Errors

Graphs: Container restarts/min, Pod warning events, Error logs/sec

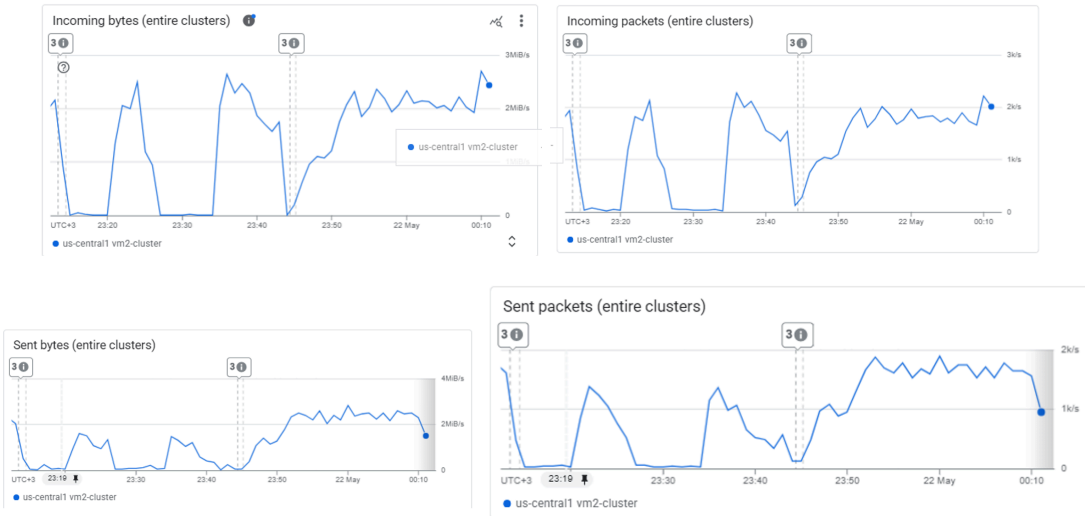
- Frequent backend pod restarts.
- Inference: Indicates backend instability under pressure. Possibly resource limits exceeded.



d. Ephemeral Storage Usage

Graph: Ephemeral storage % used

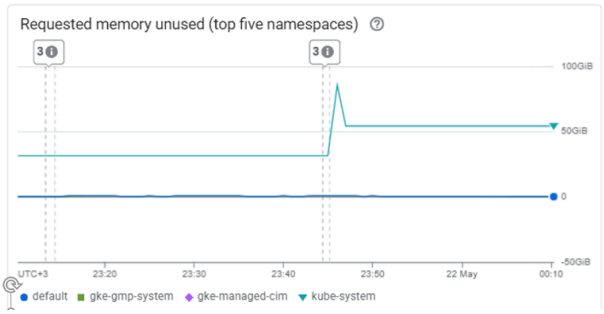
- Reached ~13.12%, with spikes.
- Inference: Logs/cache might be accumulating. Suggest garbage collection or mount logging storage separately.



e. Network Traffic

Graphs: Incoming/Sent bytes and packets

- Spikes aligned with load tests.
- Inference: Cluster scaled correctly to handle requests. No clear bottlenecks at network level.



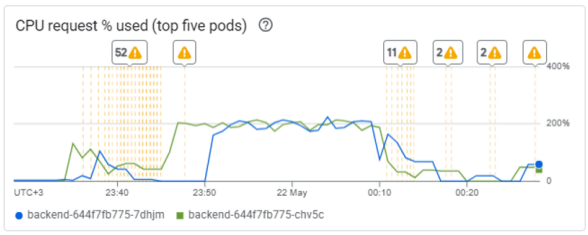
f. Requested But Unused Resources

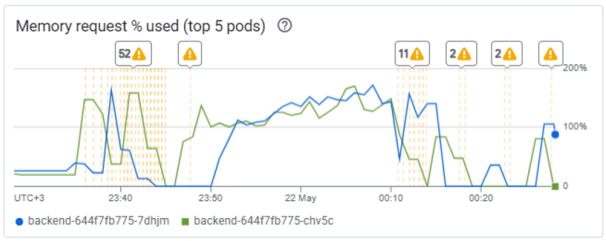
Graphs: Requested memor unused

- Unused memory % was high.
- **Inference:** Misaligned provisioning, especially in gke-managed-cni. Could be done fine-tune resource requests if necessary according to budget plan after other optimization.

After cluster observation we observed backend and frontend workloads separately.

Backend Workloads

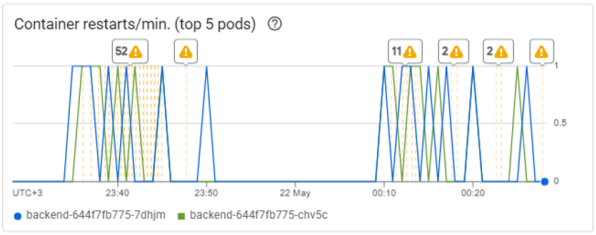


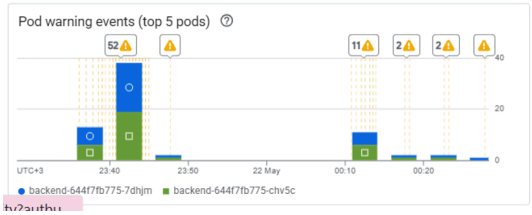


a.CPU & Memory Saturation

Graphs: CPU request % used, Memory request % used

- CPU usage went near or over 200% of request at peaks.
- Memory request usage exceeded 150%.
- **Inference:** Pods hit CPU/memory limits. Caused throttling or OOM issues.

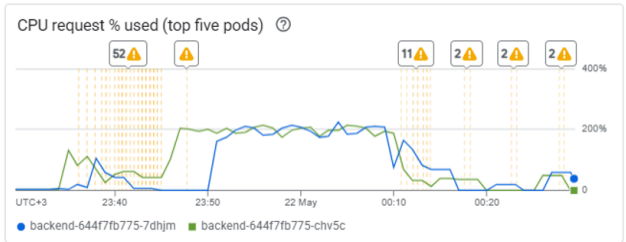




b. Container Restarts & Warnings

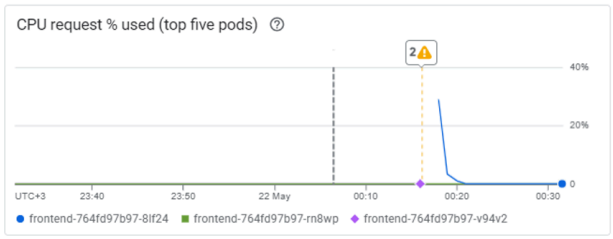
Graphs: Container restarts/min, Pod warning events

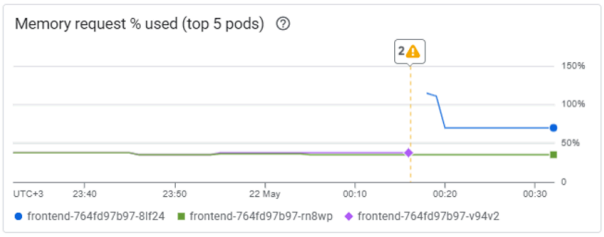
- Frequent restarts + warning spikes.
- **Inference:** Backend crashed multiple times under high load. Indicative of improper scaling or memory leak



Frontend Workloads

Key Metrics and Reasoning





a. CPU & Memory Usage

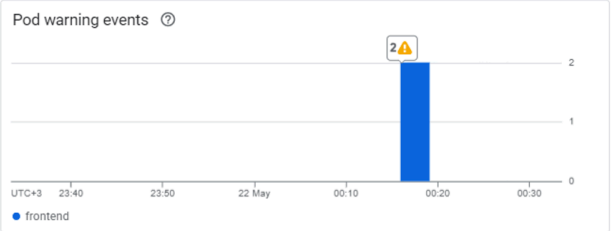
Graphs: CPU request % used, Memory request % used

- **CPU consumption** remained consistently **below 30%**, indicating the frontend pods were **never close to saturation**.
- **Memory usage** peaked around **60% of the requested limit**, with **short spikes** during the pod restart or test redeploy window.

b. Pod first ready Latency

Graph: Pod startup latency (99th percentile)

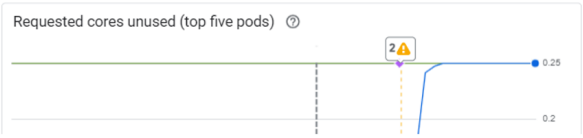
- Startup under 25s, no red zones.
- **Inference:** Startup times are healthy.

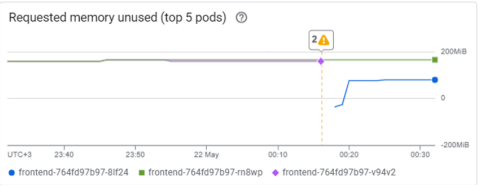


c. Errors and Warnings

Graphs: Frontend pod warning events, Container error logs/sec

- Very few warnings. No repeated errors.
- **Inference:** Stable behavior during tests.





d. Cores & Memory Unused Frontend

Graphs: Requested cores/memory vs unused

- Large gap between requested and used.
- **Inference:** Overprovisioned. Can reduce limits safely to optimize node packing.

Final Assessment Frontend Pods in GKE
Metrics considered: • **CPU request % used:** < 30% (never saturated) • **Memory request % used:** peaked ~60% • **Requested vs unused cores/memory:** large and consistent gap • **Pod startup latency:** ~22s (healthy) • **Pod error logs/warnings:** rare, no repetition **Decisions made:** • **Replicas:** reduced from 2 → 1

- **CPU request:** reduced from 250m → 250m (unchanged)
- **Memory request:** reduced from 256Mi → 128Mi
- **Memory limit:** reduced from 512Mi → 256Mi
- **Autoscaling:** added HPA with minReplicas: 1, maxReplicas: 3, cpu averageUtilization: 50

Justification: Underutilized resources. Safe to downscale and enable reactive autoscaling.

Decisions made:

- **Replicas:** reduced from 2 → 1 initially (with also HPA 1-3)
- **CPU request:** reduced from 250m → 250m (unchanged)
- **Memory request:** reduced from 256Mi → 128Mi
- **Memory limit:** reduced from 512Mi → 256Mi
- **Autoscaling:** added HPA with minReplicas: 1, maxReplicas: 3, cpu averageUtilization: 50

Justification: Underutilized resources. Safe to downscale and enable reactive autoscaling.

Final Assessment Backend Pods in GKE

Metrics considered:

- **CPU request % used:** exceeded 200%
- **Memory request % used:** exceeded 150%
- **Frequent container restarts:** correlated with CPU/memory saturation
- **Pod warnings:** repeated during peak load
- **Requested vs unused cores/memory:** almost fully utilized

Decisions made:

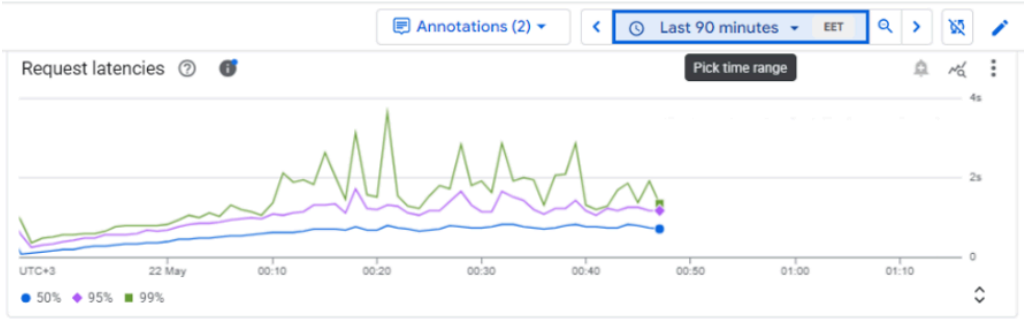
- **Replicas:** kept at 2
- **CPU request:** increased from 250m → 500m
- **CPU limit:** increased from 500m → 1000m
- **Memory request:** increased from 256Mi → 512Mi
- **Memory limit:** increased from 512Mi → 1Gi
- **Autoscaling:** added HPA with minReplicas: 2, maxReplicas: 10, cpu averageUtilization: 60

Justification: Backend pods were clearly throttled and restarted under load. We increased capacity and enabled aggressive autoscaling.

Before Testing the system again, we also monitored the functions.

completeToDos

Observed Problems and Key Metrics



1. High Request Latency under Low Load

As shown in the *Request Latency* chart, the 95th percentile latency was around 1.5–2.5s, with 99th percentile reaching ~3s.

Indicates cold start delays and inefficient request handling.

2. High Container Startup Latency

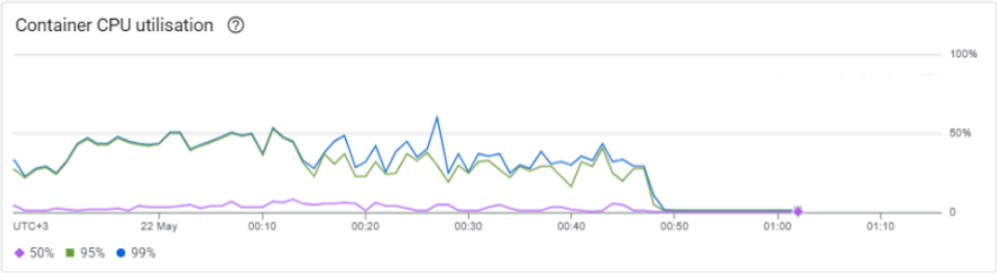
The *Container Startup Latency* graph shows spikes between 1000ms and 2500ms.

Suggests containers are created on every request — cold starts not avoided.

3. Excessive Instance Spawning

The *Container Instance Count* chart shows instance counts reaching up to 30 concurrently.

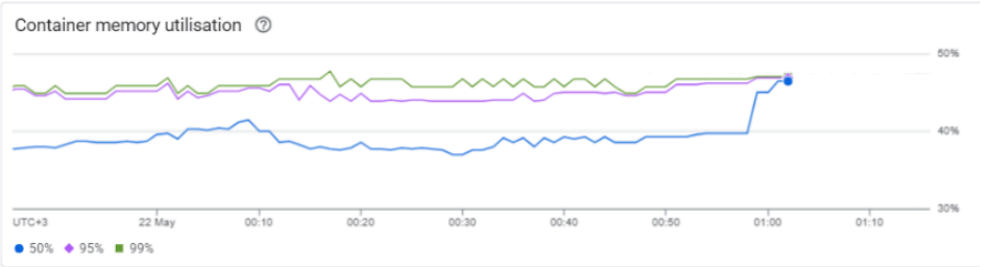
Means concurrency setting was too low (default = 1), triggering new containers per request.



4. High CPU Usage Spikes

The CPU Utilization chart shows rapid jumps between 50%–100%.

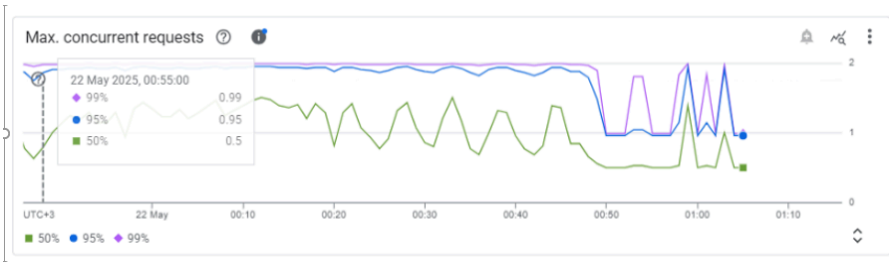
Indicates MongoDB connection handling and cold initialization were CPU-intensive.



5. Memory Usage Was Low

As seen in *Container Memory Utilization*, memory usage hovered around 45%–50%, leaving buffer.

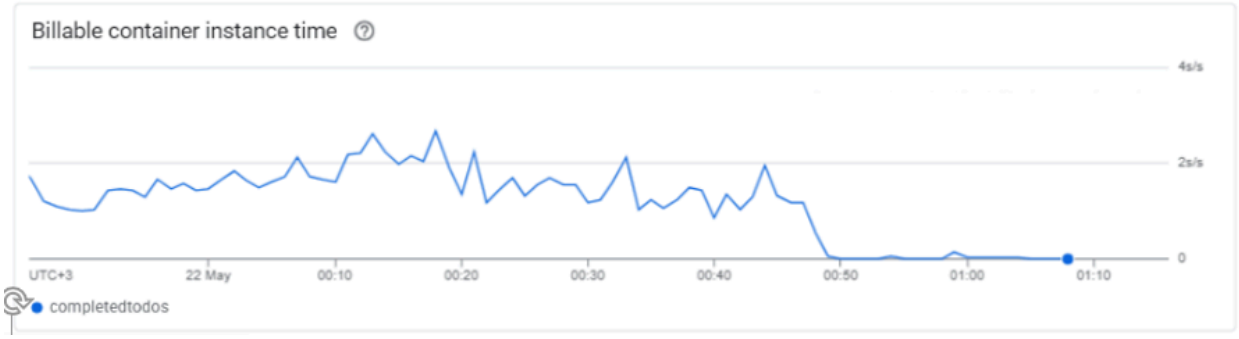
Memory was sufficient, but CPU limits were tight.



6. Concurrency Was Not Utilized

The Max Concurrent Requests graph showed flat average of 1.

Confirms default concurrency = 1 (each request triggers a container).



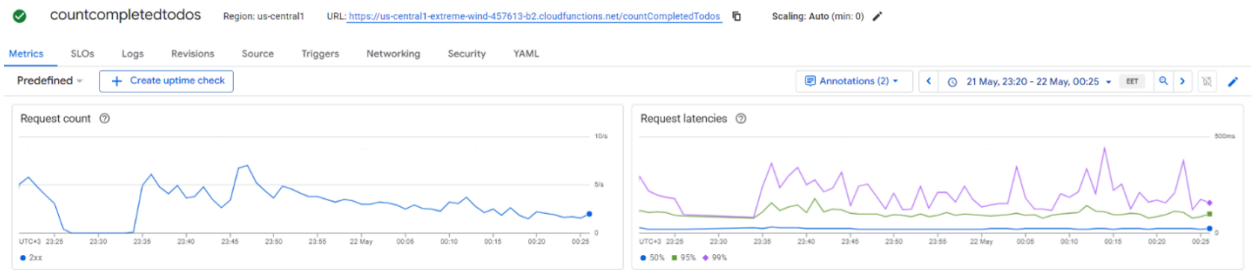
7. Billable Container Time Was High

The *Billable Instance Time* chart indicated continuous container activity.

Suggests inefficient billing due to cold starts and scaling delay.

Final Assessment COMPLETEToDo Serverless Function in Cloud Run
min-instances → 0 ➜ 3 **concurrency → 1 ➜ 10** **memory → 256MB ➜ 512MB** **CPU → 1 → 2**

countCompleteTodos



1. Request Latencies

Graph: Request latencies

95th percentile latency often exceeds 2s, with 99th percentile near 3s.

Latency patterns are inconsistent and peaky, especially under rising load.

Indicates cold starts and per-request container overhead.

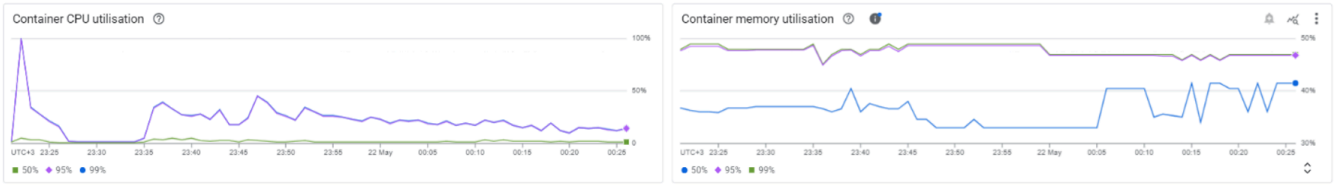
2. Cold Start Indicators

Graph: Container Startup Latency, Container Instance Count

Startup latency fluctuates between 1000–1500ms, occasionally spiking.

Instance count increases reactively with request load (spikes >20), showing lack of warm instances.

Each request triggers new container → adds cold start time.

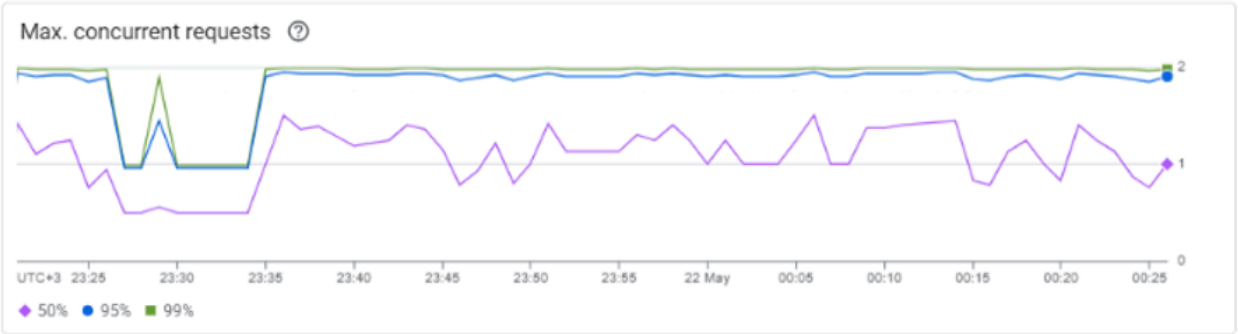


3. CPU & Memory Utilization

Graph: Container CPU Utilization, Container Memory Utilization

CPU often spikes to 100%, showing that CPU is a throttling factor.

Memory usage is moderate (~50–60%), no immediate signs of memory pressure. CPU underprovisioning = backend slowdowns, longer startup/processing.



4. Concurrency Misuse

Graph: Max Concurrent Requests

Nearly flat at 1, meaning each request is processed in its own container.

This is highly inefficient, increasing cost and startup time.

5. Billable Container Time

Graph: Billable Container Instance Time

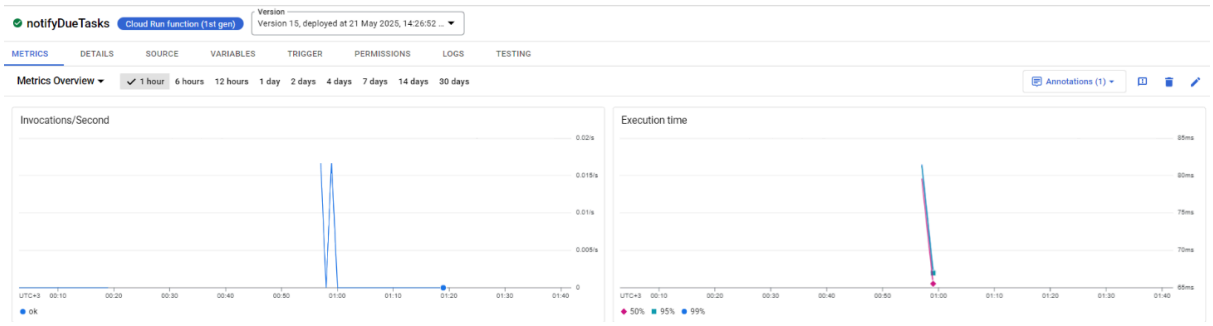
High and rising over time, even when request count drops.

Caused by lack of reuse (concurrency = 1) and no min-instances.

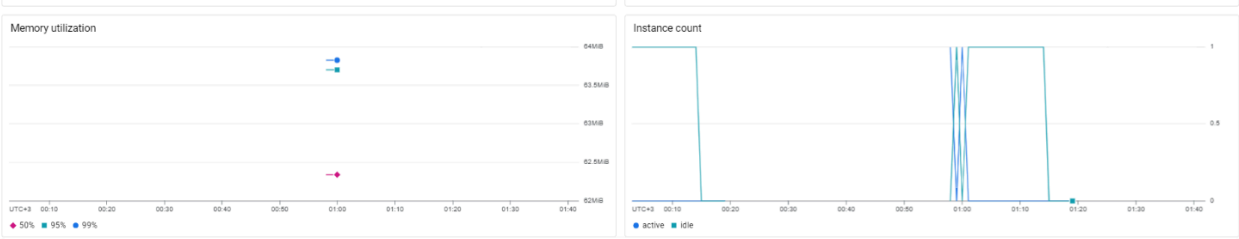
Final Assessment countCompleteTodos Serverless Function in Cloud Run
min-instances → 0 ➜ 3 **concurrency → 1 ➜ 10** **memory → 256MB ➜ 512MB** **CPU → 1 → 2**

notifyTasks

- This function runs on Cloud Functions (1st gen).
- Cold start problem is not observed (execution is fast).
- Autoscaling works as expected with minimal invocations.
- Task is lightweight and infrequent (scheduled monthly)



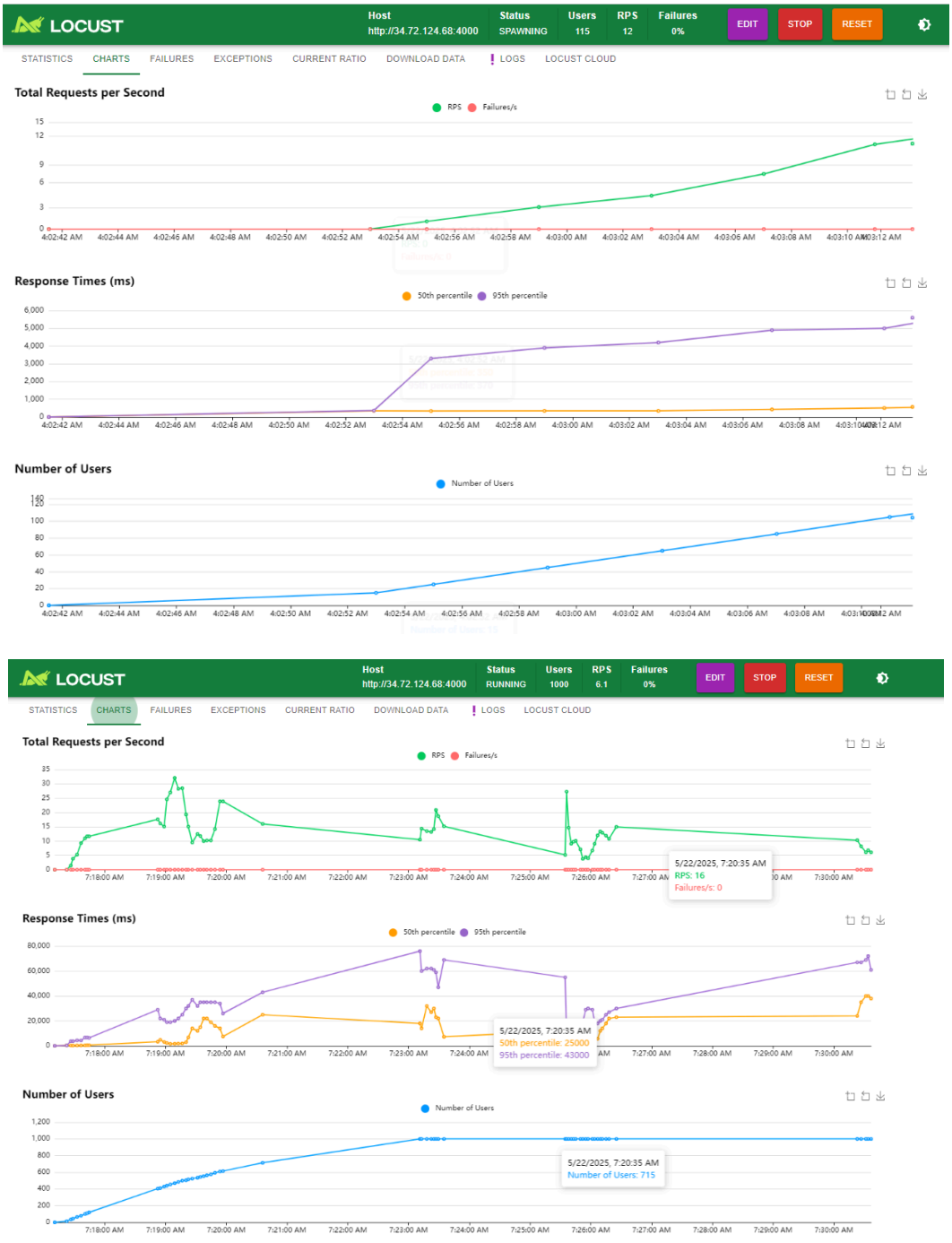
This function runs periodically with very low load and memory usage (~63Mi). Execution time is consistently below 100ms and there is no cold start delay. Since it's already using Cloud Functions Gen 1 efficiently, migrating to Cloud Run would increase cost without benefit. No scaling or configuration change is required.



All metrics show stability and efficiency. Keeping the current lightweight configuration ensures minimal cost and overhead, without risking performance.

notifyTasks
Platform: Cloud Functions (1st gen) → same **min-instances:** 0 → 0 (no cold start issues) **concurrency:** 1 → 1 (no need for parallel execution) **memory:** 256Mi → 256Mi (current usage ~63Mi, no bottleneck)

After Optimization Performance Load Test



Load Test Comparison – Before vs After Optimizations

Metrics Observed:

Response Time (95th percentile)

Failure Rate (%)

Requests per Second (RPS)

User Scaling Stability

Before Changes

(from early Locust charts with 1000 users)

- 95th percentile response time: **~50,000–60,000ms**
- Failure rate: **peaks at 28%**
- RPS fluctuated heavily: **~20–40**, unstable
- Users could not scale beyond **~400–500** without error spikes
- Frequent dips in request throughput + latency spikes

After All Optimizations

(from latest charts with 1000 users)

- 95th percentile response time: **~43,000ms**, 50th: **~25,000ms** (reduced from before)
- Failure rate: **0%**
- RPS stable at **16–20** despite **1000 users**
- Load handled **consistently and smoothly**
- No sharp drops in users, no RPS collapse

The earlier system struggled with load because:

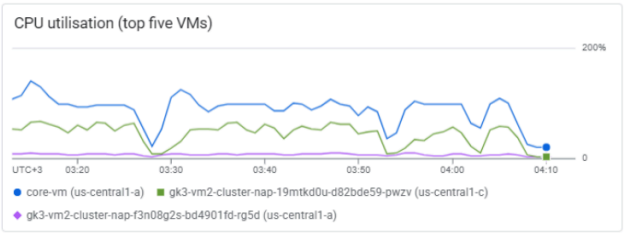
- **Pods were crashing** due to memory/CPU saturation.
- **Cold starts** increased latency.

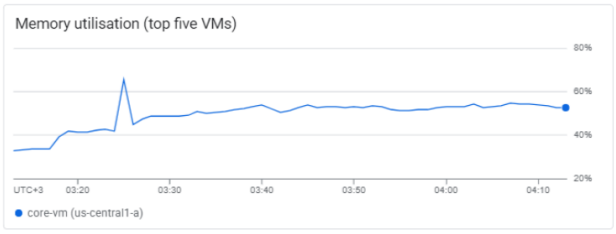
- Lack of **concurrency, 1 vCPU and min-instance config** caused new container spawns per request.

After the updates:

- We introduced **autoscaling**, **resource adjustments**.
- Backend stabilized under **1,000 users**.
- **Failures dropped to 0%** even under peak load.
- The system no longer collapses mid-test, ensuring production resilience.

This Time Bottleneck becomes VM so that we created a high CPU VM and moved out mongoDB configurations and disk to our new VM and assigned the static IP to new one.





The old VM's monitoring details are above, CPU is bottleneck but memory is fine.

So the new VM configurations is below.

GCE Virtual Machine(MongoDB)

Basic information

- **Name:** database-instance
- **Instance ID:** 1270795797121147915
- **Description:** None
- **Status:** Running
- **Creation time:** May 22, 2025, 4:56:06 am UTC+03:00

- **Zone:** us-central1-c
- **Reservation:** None
- **Instance template:** None
- **Host error:** None
- **Physical host:** None
- **Maintenance policy:** Migrate
- **Tags:** g
- **Deletion protection:** Disabled
- **Confidential VM service:** Disabled
- **Provisioned iops size:** 0

Machine configuration

- **Machine type:** e2-highcpu-2 (2 vCPU, 2 GB memory)
- **CPU platform:** Intel Broadwell
- **Minimum CPU platform:** Not specified
- **Architecture:** x86/64
- **Cores to use ratio:** 1.0
- **Custom vCPU core count:** 2
- **Enable nested virtualization:** No
- **Display device:** Disabled
- **GPUs:** None
- **Resource policies:** None

After upgrading the VM configuration, we observed a significant increase in traffic to the VM. When we initiated an open-ended Locust test with 1000 users, our backend pods began to crash repeatedly. This was due to the excessive resource demands that exceeded our GKE cluster's capacity.

Since it was not feasible to scale the GKE cluster to meet this demand within the constraints of the Google Cloud Free Tier, we reassessed the situation. We concluded that our initial Locust configuration was not realistic for our testing environment.

As a result, we decided to not change clusters and stay with last set up—specifically, with lower user counts, controlled durations, and gradual ramp-up settings our system ensure stability and generate meaningful results.