Scientific Report

Distributed Chat Application Analysis

Students:

Cojocaru George
Dinu Darius Dragoș
Iosub Miruna Elena
Popa Ștefan-Eduard

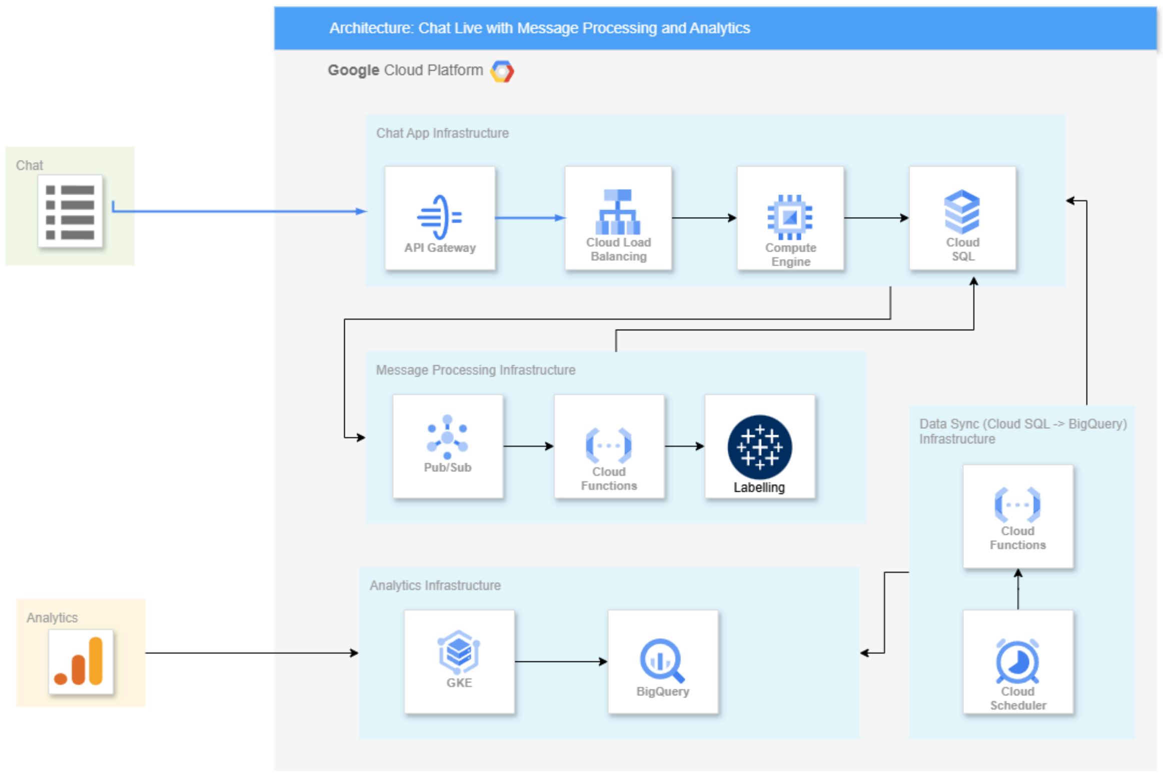
Faculty of Computer Science, UAIC
April 2025

Introduction

This report analyzes a distributed chat application deployed on Google Cloud Platform (GCP) using Python, Node.js, Express, Socket.io, Kubernetes, PostgreSQL, and Docker. The system is designed to handle real-time communication while ensuring scalability, reliability, and efficiency. The report evaluates key architectural characteristics, including performance, latency, reliability, and transparency.

System Architecture

The system consists of the following components:



Chat App

The Chat App Infrastructure is the core of the system, powering real-time communication:

- Frontend (Client App): A lightweight HTML + JavaScript web application that connects to the backend via WebSockets. It allows users to send and receive messages instantly.
- Backend (Real-time Engine): Built with Node.js, Express, and Socket.IO, this server handles:
  - User session handling and WebSocket connections
  - Broadcasting messages to users
  - Writing messages to the database
  - Publishing messages to the Pub/Sub system for further processing
- API Gateway: Controls access to backend services, adding a layer of authentication and security.
- Cloud Load Balancer: Distributes incoming traffic across multiple backend instances for scalability and fault tolerance.
- Compute Engine: Hosts the containerized backend (packaged with Docker). Autoscaling ensures smooth handling of concurrent users.
- Cloud SQL (PostgreSQL): Stores chat messages, user metadata, timestamps, and other persistent data. It acts as the transactional backbone of the app.

Message Processing

The Message Processing Infrastructure enriches and transforms messages asynchronously:

- Pub/Sub: Chat messages are pushed to a Google Cloud Pub/Sub topic from the backend, enabling decoupling between real-time messaging and processing.
- Cloud Functions: Written in Node.js or Python, these serverless functions:
  - Consume messages from Pub/Sub
  - Clean or validate content
  - Trigger downstream processing like tagging or classification

- Labelling Service: This is a custom Python-based module that performs NLP-based enrichment tasks, such as:
    - Topic Classification (e.g., categorizing messages based on their content)
    - Toxicity Detection using pre-trained models (optional)
- This adds valuable metadata to the messages before they are stored or passed on for analytics.

Analytics

The Analytics Infrastructure provides insights and dashboards based on enriched chat data:

- Kubernetes: Runs analytics workloads and dashboards.
- BigQuery: Acts as the main data warehouse. Enriched chat messages (including tags and sentiment scores) are stored here for querying using SQL or integrated tools like Looker Studio.
- Cloud SQL → BigQuery Sync:
    - Cloud Scheduler triggers a Cloud Function (written in Python) at regular intervals.
    - This function extracts new data from Cloud SQL, processes it if needed, and loads it into BigQuery.
    - Data is deduplicated and pre-aggregated before insertion to optimize query performance.

Performance Analysis

Performance is a critical factor in real-time applications. This chat system ensures performance efficiency through:
- WebSockets: Uses an event-driven model to minimize overhead compared to HTTP polling.
- Load Balancing: Distributes incoming traffic across multiple instances to prevent overload.
- Database Optimization: Messages are stored in a PostgreSQL database with indexed queries to optimize retrieval.
- Auto-scaling: Automatically adjusts the number of running instances based on demand.

Latency Considerations

Latency directly impacts user experience in a chat application. The system minimizes latency through:
- WebSockets for Real-Time Communication: Ensures instant message delivery.
- Regional Deployment: The system is deployed in the europe-west1 region to serve users in proximity efficiently.
- Cloud NAT & VPC Peering: Provides optimized networking for backend communication without unnecessary routing overhead.

Reliability & Fault Tolerance

The application maintains high availability and fault tolerance through:
- Instance Group Management: Ensures that at least two instances are always running.
- Health Checks: Automatically detects and replaces unhealthy instances.
- Database Backups: Automated backups at 03:00 daily to prevent data loss.
- Stateless Backend: Each backend instance does not maintain user session state, ensuring seamless failover.

Transparency & Observability

Transparency in distributed systems is essential for debugging and monitoring. The system incorporates:
- Cloud Logging & Monitoring: Enables tracking of application and network performance.
- Request Tracing: Logs incoming requests and responses for debugging.
- Load Balancer Logging: Monitors traffic distribution and identifies bottlenecks.

Security Measures

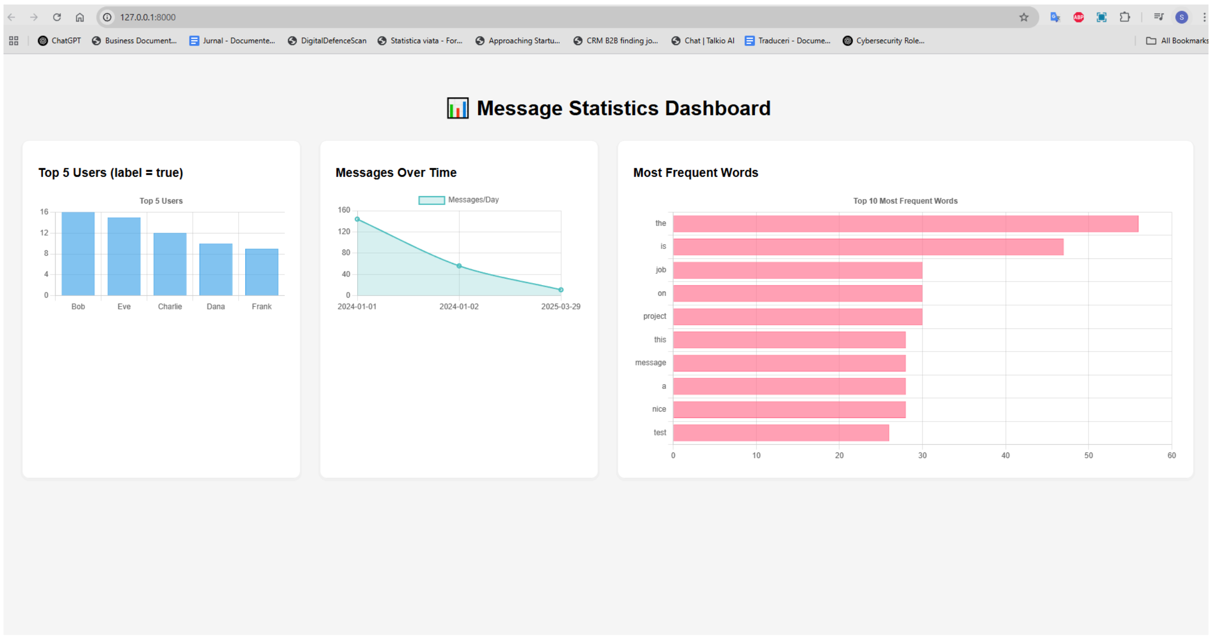
Security is implemented at multiple levels:
- Private Database Access: The database is not publicly accessible, reducing attack surfaces.
- Service Account Permissions: Follows the principle of least privilege to minimize risks.

Scalability & Future Enhancements

The system is designed for scalability:
- Horizontal Scaling: Can increase instance count dynamically based on traffic.
- Global Load Balancing: Future improvements could involve multi-region deployment for broader coverage.
- Cache Layer: Implementing Redis could further enhance performance by caching frequently accessed data.

Dashboard Screenshot

Below is a screenshot of the Message Statistics Dashboard, visualizing user activity, message frequency, and most common words.



Cloud Function

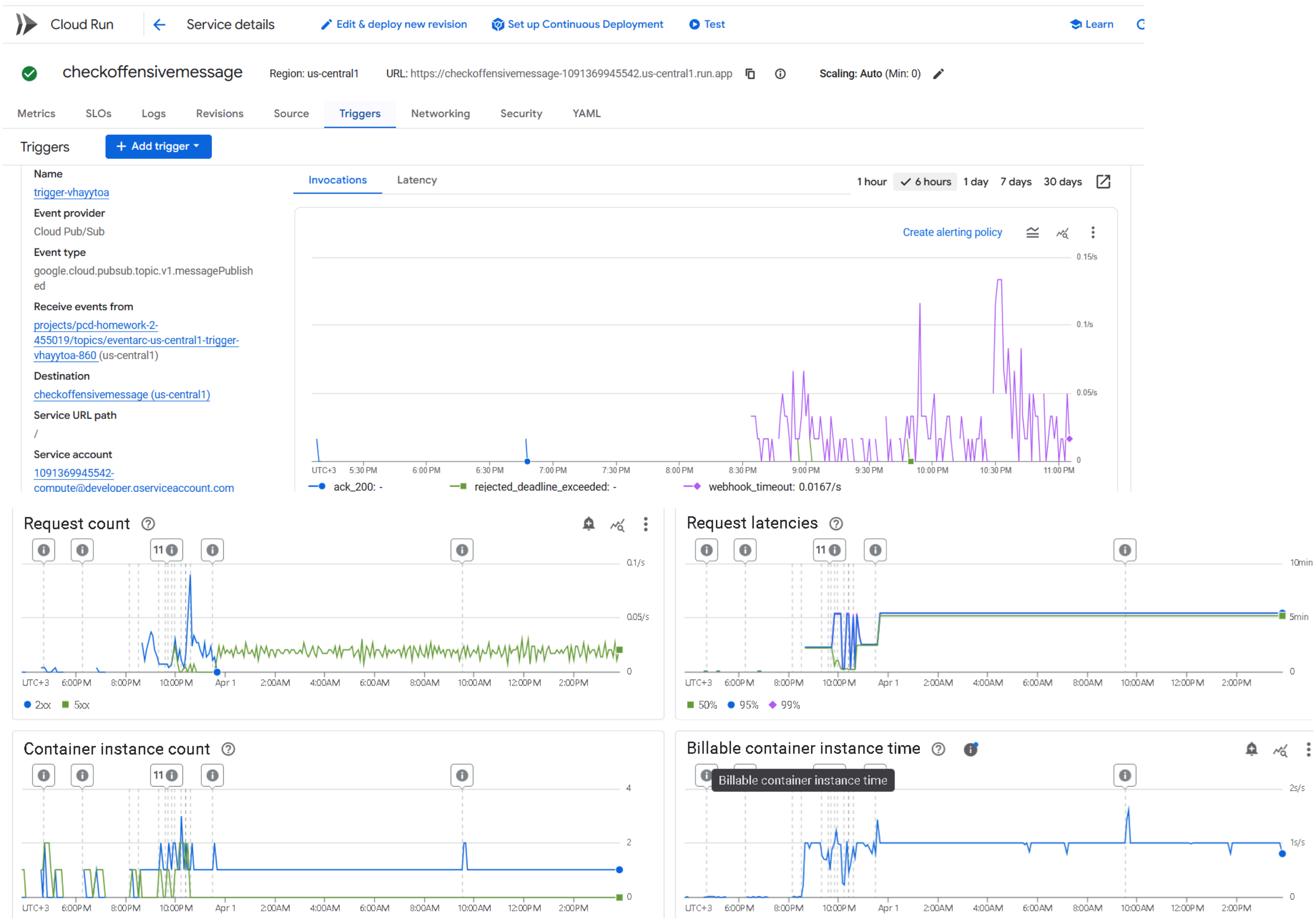
One of the core components of the application is represented by the Cloud Function enabled through the "Cloud Run" service of Google Cloud Platform (GCP). We have developed such a cloud function for quickly and efficiently labeling messages sent by users as offensive or not offensive, based on a threshold we can manipulate.

The source of the cloud function is written in python and is deployed using the GCP Console. The function connects to the cloud hosted DB and checks whether a message is offensive or not by iterating over it and verifying if it has any terms contained in a predefined list of "offensive terms".

In order to maintain an efficient connection with the rest of the application's components, the Cloud Function has a trigger of type "Pub/Sub" which allows it to intercept events in an efficient manner, only when actual messages are sent within the app, therefore saving valuable resources within the cloud space.

As for security, the function only allows calls from within the same VPC (authenticated calls) and we used the built-in testing tool provided by GCP in order not to risk attacks on the cloud function such as DDoS.

Below is a screenshot recording the triggers of type "Pub/Sub" that were used to invoke the cloud function.



Cloud Run   Service details   Edit & deploy new revision   Set up Continuous Deployment   Test   Learn

checkoffensivemessage   Region: us-central1   URL: https://checkoffensivemessage-1091369945542.us-central1.run.app   Scaling: Auto (Min: 0)

Metrics   SLOs   Logs   Revisions   Source   Triggers   Networking   Security   YAML

Triggers   + Add trigger

Name
trigger-vhayytoa

Event provider
Cloud Pub/Sub

Event type
google.cloud.pubsub.topic.v1.messagePublished

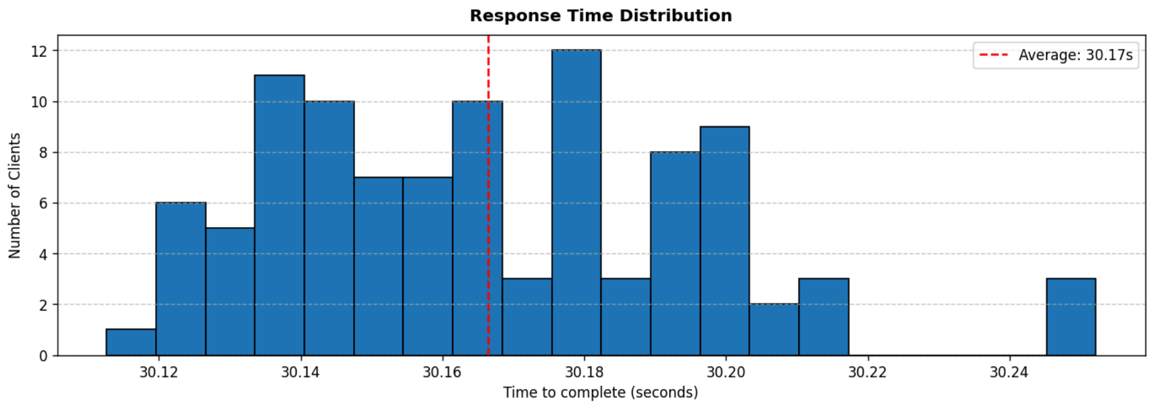
Receive events from
projects/pcd-homework-2-455019/topics/eventarc-us-central1-trigger-vhayytoa-860 (us-central1)

Destination
checkoffensivemessage (us-central1)

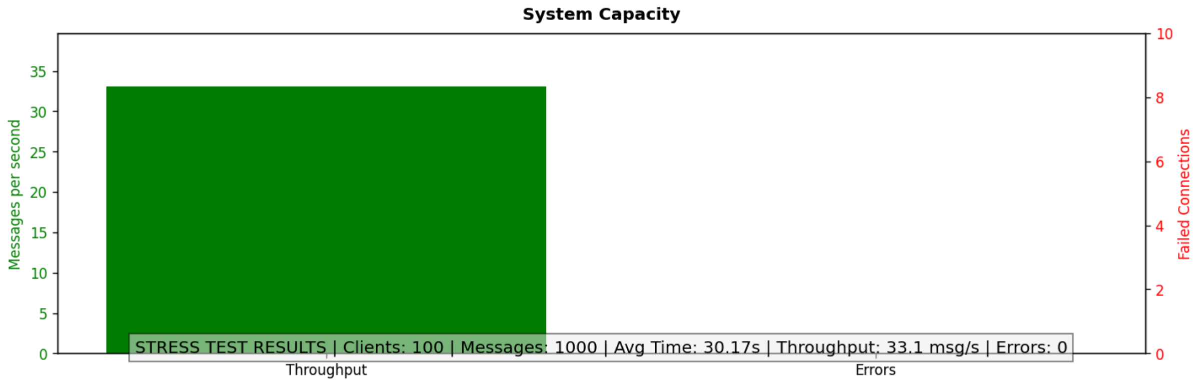
Service URL path
/

Service account
1091369945542-compute@developer.gserviceaccount.com

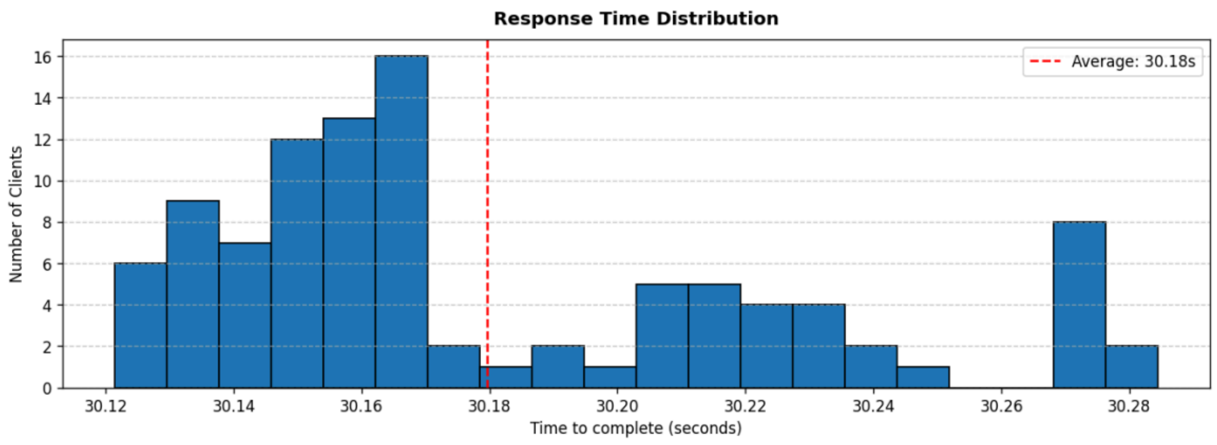
Metrics

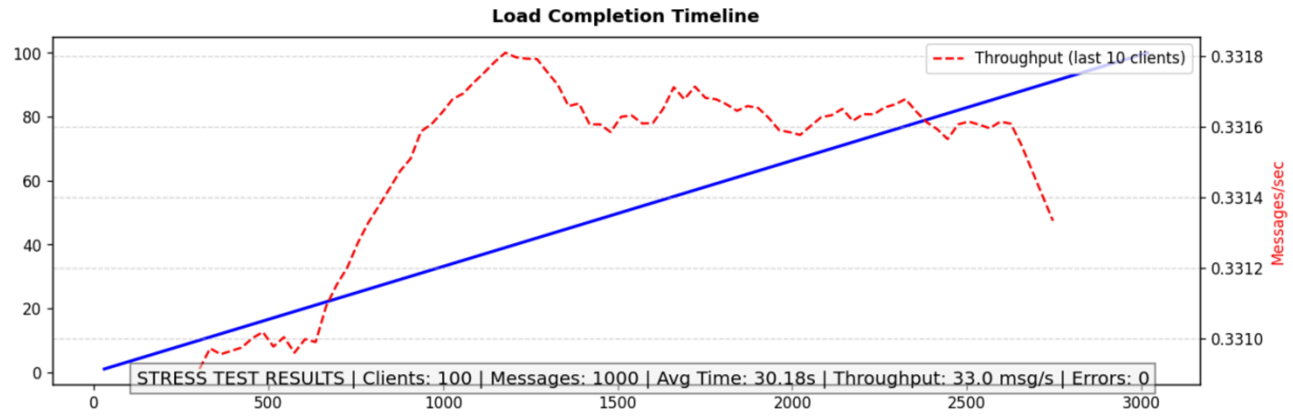


Response Time Distribution
Average: 30.17s
Number of Clients
Time to complete (seconds)



System Capacity
Messages per second
Failed Connections
Throughput
Errors
STRESS TEST RESULTS | Clients: 100 | Messages: 1000 | Avg Time: 30.17s | Throughput: 33.1 msg/s | Errors: 0





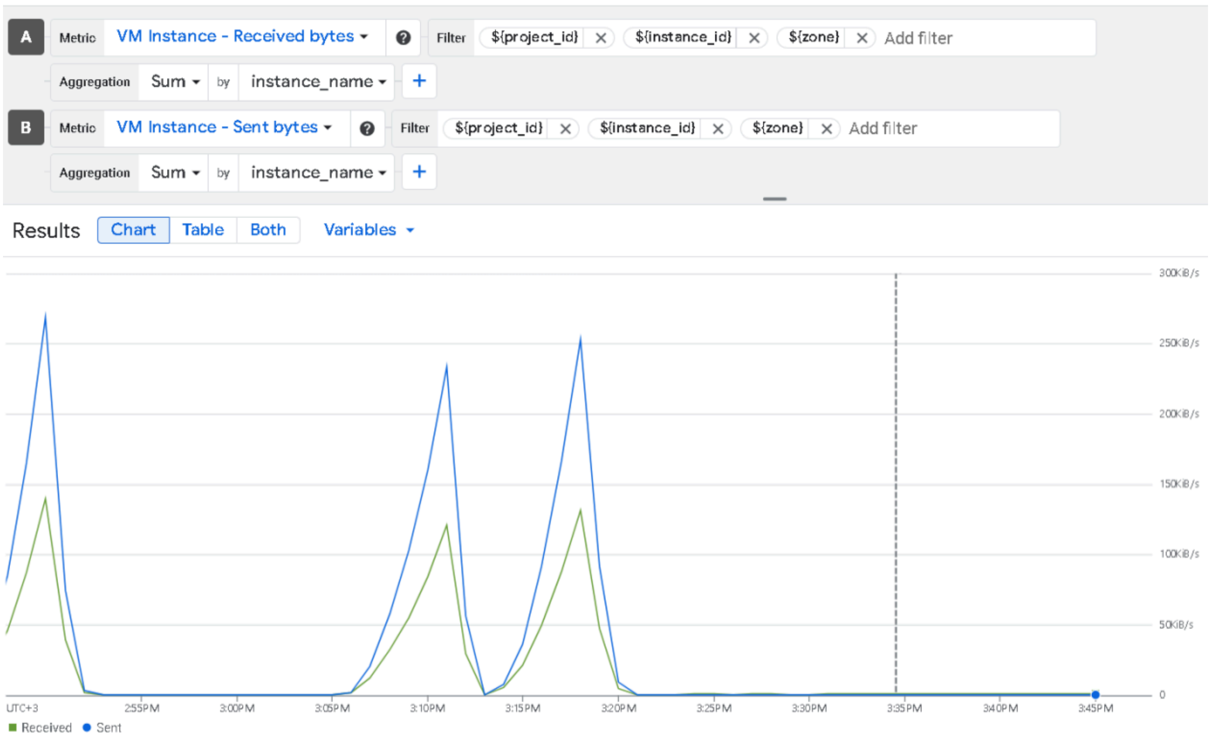
STRESS TEST RESULTS | Clients: 100 | Messages: 1000 | Avg Time: 30.18s | Throughput: 33.0 msg/s | Errors: 0

1. Response Time Distribution

Shows if most requests cluster around a certain time (good) or are spread out (problems)
Red bars highlight statistical outliers
Clearly shows if you have a long tail of slow requests

2. Load Completion Timeline

Blue line: Cumulative clients completing over time
Straight line = consistent performance
Flattening curve = system slowing down
Red dashed line: Instantaneous throughput
Dropping throughput = resource exhaustion
Spikes = inconsistent performance



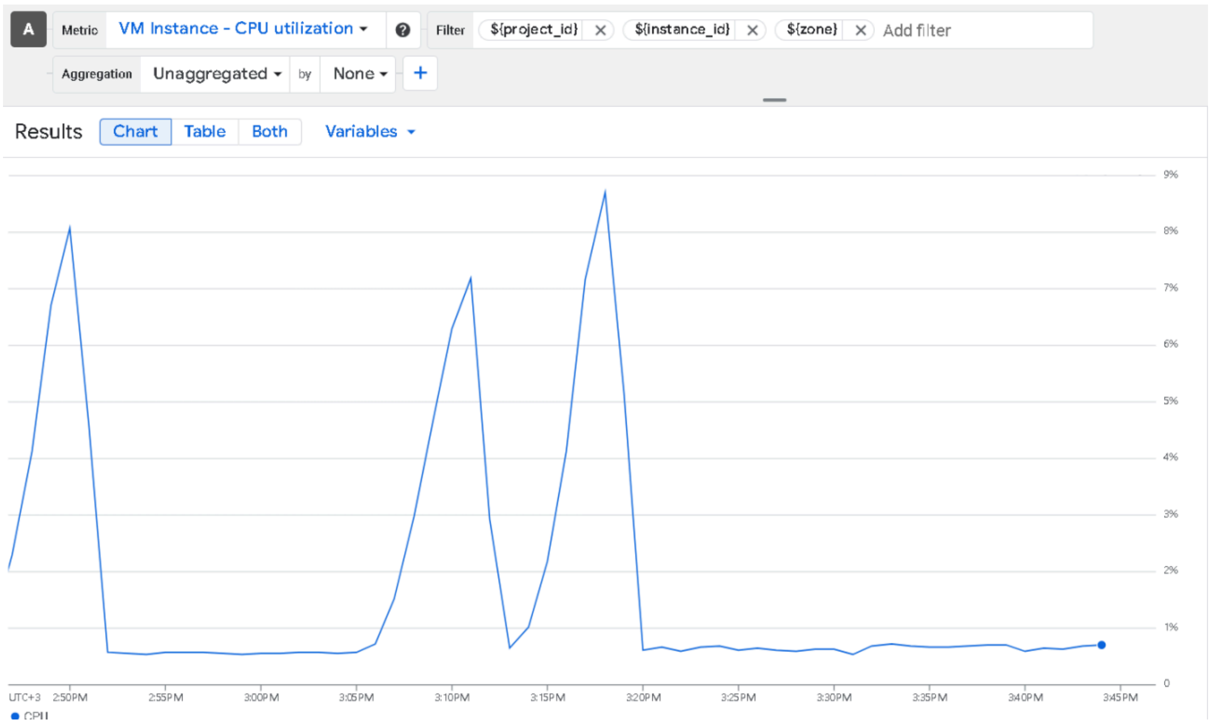
A Metric: VM Instance - Received bytes; Filter: ${project_id} ${instance_id} ${zone}; Add filter

Aggregation: Sum by instance_name

B Metric: VM Instance - Sent bytes; Filter: ${project_id} ${instance_id} ${zone}; Add filter

Aggregation: Sum by instance_name

This graphic shows network traffic metrics for VM instances, split into received and sent bytes. The data is aggregated by instance name. The entries like "300B/s" suggest bandwidth usage at given times or thresholds.



A Metric: VM Instance - CPU utilization; Filter: ${project_id} ${instance_id} ${zone}; Add filter

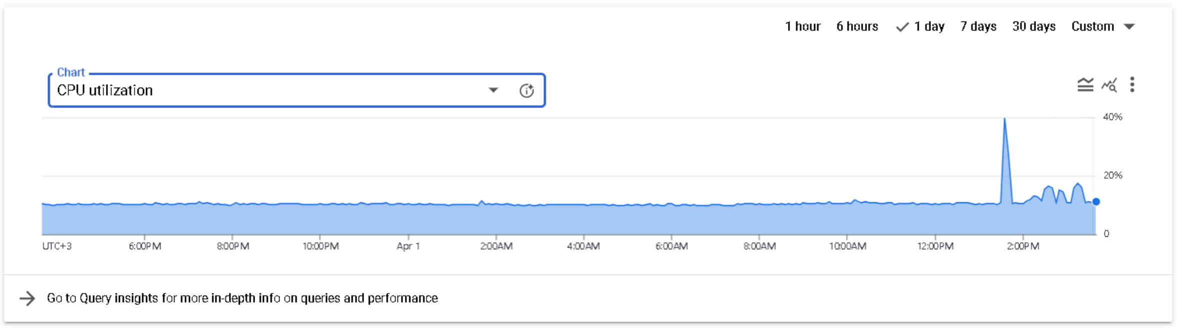
Aggregation: Unaggregated by None

This graphic shows CPU utilization for a VM instance.

All instances > chatapp-db-instance

chatapp-db-instance

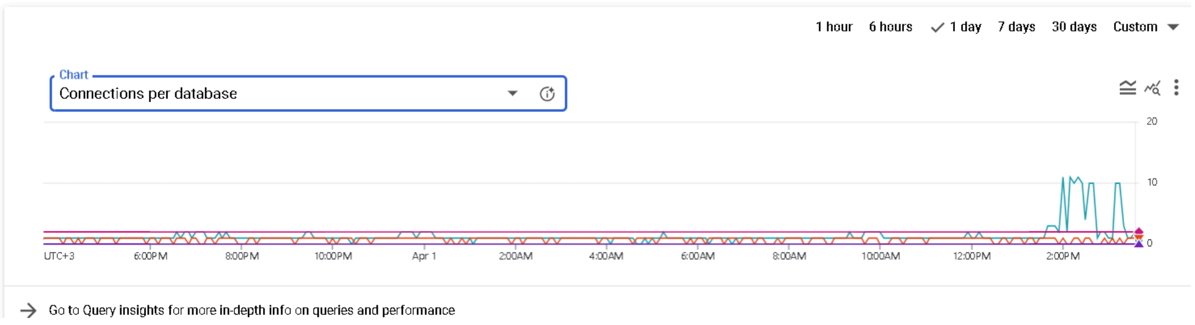
PostgreSQL 13



The graph shows spikes/dips in transaction volume, indicating peak usage periods or potential bottlenecks.

chatapp-db-instance

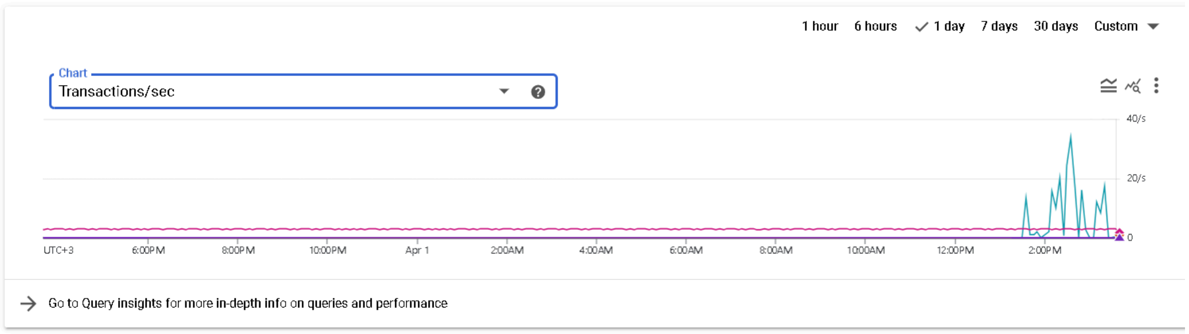
PostgreSQL 13



High connection counts may indicate heavy user load or connection leaks. The repeated timestamps suggest data logging issues.

chatapp-db-instance

PostgreSQL 13



The graph shows spikes/dips in transaction volume, indicating peak usage periods or potential bottlenecks.

Conclusion

This chat application demonstrates an efficient, reliable, and scalable distributed system leveraging GCP's cloud infrastructure. The use of Terraform for infrastructure management, Docker for containerization, and WebSockets for real-time messaging ensures an optimized experience. Future improvements can further enhance system resilience and performance.