PCD| Homework 2

Bibire Constantin-Mihnea – ISS1

Dobrică-Spiriac Roxana-Iuliana – ISS1

Dominte Mihai-Alexandru – ISS1

Nastasiu Ștefan – ISS1

Project Overview

In this project, we developed a real-time chat application that allows users to communicate instantly. The system supports conversations between two or more friends and ensures low-latency messaging using WebSockets. Unlike traditional HTTP requests, WebSockets enable a persistent connection between clients and the server, making it ideal for instant messaging applications.

The application consists of three main components:

- **Native Cloud services**: We use **Firestore** for message storage, **Cloud Storage** for media files, and **Pub/Sub** for event-driven messaging.
- **On-premise application**: A metrics aggregation service that collects and stores chat performance data in BigQuery for analysis.
- **FaaS (Function as a Service)**: A **Sentiment Analysis function** that processes messages and classifies emotions in real-time.

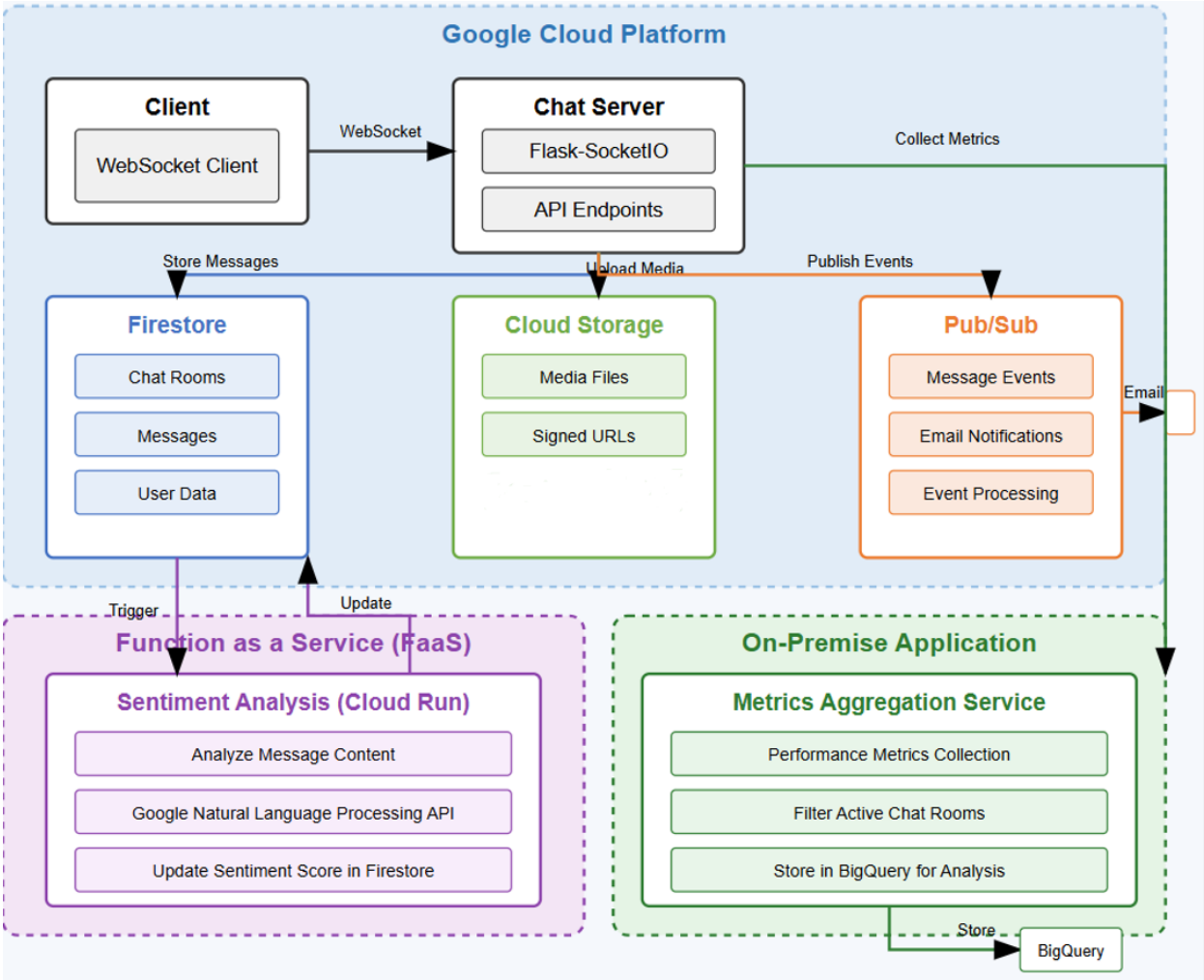


Figure 1. Architectural Diagram

Native Cloud Services

1. **Firestore**

   We use Firestore to store chat messages because it is a serverless, highly scalable NoSQL database that provides:

- Real-time synchronization: Changes made to the database are immediately reflected across all connected clients, ensuring instant message delivery.
- Automatic scaling: Firestore scales efficiently to handle an increasing number of chat users and messages without manual intervention.
- Structured data storage: Messages are stored in a hierarchical format (chats/{room_id}/messages), making retrieval efficient and logical.

Firestore is ideal for chat applications because it supports real-time updates and offers low-latency access across distributed systems.

2. Cloud Storage

Our application supports images sharing. Google Cloud Storage is used because:

- It securely stores large files while maintaining high availability.
- It provides signed URLs to allow users to access media files without exposing the storage bucket publicly.
- It supports automatic scaling, ensuring that increased storage demand does not impact performance.

3. Pub/Sub

Google Cloud Pub/Sub acts as a messaging middleware that helps us manage asynchronous tasks efficiently. In our application, Pub/Sub is used for:

- Email notifications: When a new message is sent, an event is published to a Pub/Sub topic, triggering an email notification to the recipient.
- Decoupling components: The chat application can continue sending messages without waiting for email notifications to be processed, improving performance.
- Scalability: Pub/Sub ensures that message processing remains fast and reliable, even as the number of users increases.

How Pub/Sub works:

- When a user sends a message, the application publishes an event to the Pub/Sub topic.
- A subscriber function listens for new messages and triggers an email notification to the recipient.
- This process ensures non-blocking communication, allowing the chat system to function smoothly even under heavy loads.

On-premise application

To monitor chat application performance, we implemented an on-premise application that periodically collects, processes, and stores chat performance metrics in **BigQuery**. The service gathers real-time analytics, such as active users, message latency, and error rates, ensuring that only relevant data (i.e., chat rooms with active users) is logged. This allows for efficient monitoring and optimization of the system.

Implementation Details

The on-premise application collects performance metrics from the chat system every minute and inserts them into a BigQuery table for analysis. The collected data includes:

- Active users: The number of users engaged in a chat room.
- Total messages: The count of messages sent within a chat room.
- Average latency: The delay in message delivery.
- Error count: The number of message transmission errors.
- Throughput: The rate of messages sent per second.

To ensure efficiency, the application fetches aggregated data from the chat system's /chat/metrics endpoint and filters out rooms with no active users before inserting records into BigQuery. This prevents unnecessary data storage and focuses on meaningful analytics. The service runs on-premise and uses Google Cloud's BigQuery API to store data securely.
How It Works:

- Every minute, the application queries the chat system for updated metrics.
- It processes the retrieved data and ensures only active chat rooms are recorded.
- The processed metrics are inserted into BigQuery for analysis.

By maintaining an up-to-date performance log, this component enables better observability, helping to diagnose issues and optimize chat performance in real-time.

FAAS Component: Sentiment Analysis in Chat

The application incorporates a Function-as-a-Service (FaaS) component deployed on Cloud Run to perform sentiment analysis on messages exchanged in chats. This serverless function is triggered whenever a new message is stored in the Firestore database, analysing its sentiment with **Google Natural Language Processing API**, and adding the magnitude and score values to the message fields in the database.

The sentiment analysis results can be utilized in various ways, such as:

- Content moderation, identifying potentially harmful or inappropriate messages.
- User feedback, providing insights into the overall tone of conversations.
- Enhancing user experience, allowing adaptive interactions based on sentiment trends.

WebSockets

Our application relies on WebSockets (via Flask-SocketIO) to enable real-time, bi-directional communication between users.

Unlike traditional HTTP requests, which require continuous polling to check for new messages, **WebSockets provide**:

- Persistent connections: Users stay connected to the server, receiving instant updates without repeatedly sending requests.

- Low latency: Messages are delivered in real-time, making conversations feel instantaneous.
- Efficient data transfer: Since WebSockets avoid the overhead of repeated HTTP requests, they reduce network load and improve scalability.

How WebSockets Are Used in Our Chat App:

1. Establishing a Connection
   When a user opens the chat, their client establishes a WebSocket connection with the server.
   This keeps the connection open, allowing the server to push new messages immediately.

2. Sending and Receiving Messages in Real-Time
   When a user sends a message, it is:
- Broadcasted to all users in the chat room.
- Stored in Firestore for persistence.
- Published to Google Cloud Pub/Sub for email notifications (if applicable).

3. Handling Media Messages
   WebSockets also support media file sharing. When a user uploads an image:
- The file is stored in Cloud Storage.
- A signed URL is generated for secure access.
- The WebSocket sends the media message to other users.

Challenges:

While WebSockets are great for real-time communication, they can introduce scaling challenges:

- Persistent connections require efficient resource management.
- If many users connect simultaneously, the server must handle concurrent events efficiently.

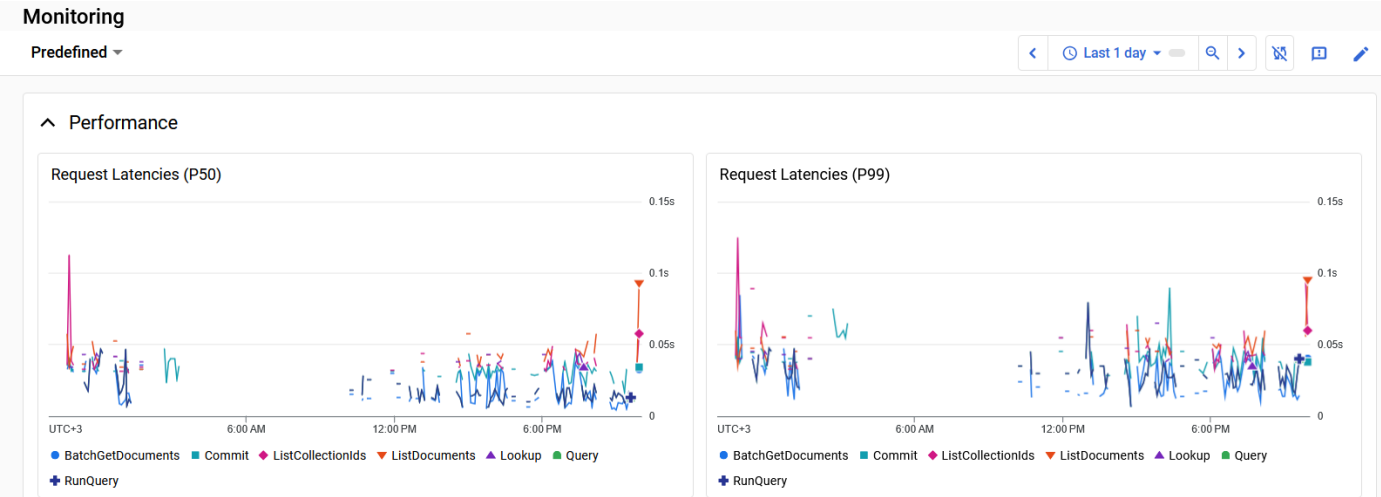
To scale WebSockets **effectively**, we can:

- Use multiple instances of the WebSocket server behind a load balancer.
- Integrate Pub/Sub or a message queue to distribute workload.
- Implement server-side optimizations (e.g., limiting idle connections).

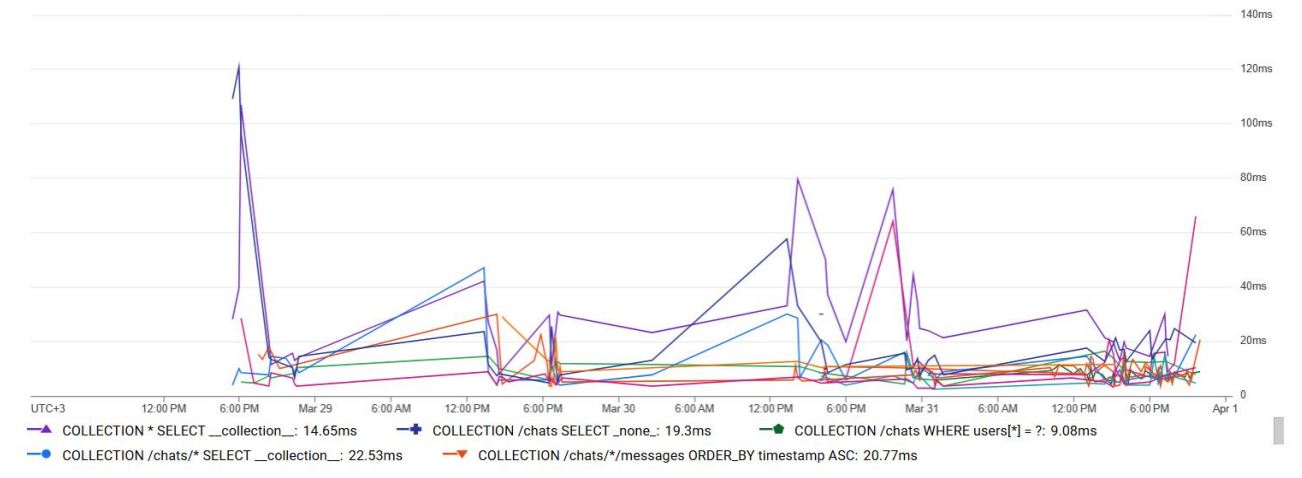
Characteristics of the System - Performance Metrics

1. Firestore

Latency



Firestore provides low-latency data access, with an average response time of ~0.04s, and a maximum observed latency of 0.12s. This ensures that messages are retrieved and displayed in real-time.



General Observations:

- Latency is mostly low (below 25ms) for most queries

- Two noticeable spikes in query latency (March 28 and March 31), where some queries exceed 140ms.
- The highest latency spikes seem to come from queries related to SELECT collection and ORDER BY timestamp ASC.

Key Query Performance:

- High Latency Spikes: COLLECTION * SELECT collection (Purple Line)
  Had the highest spike (~140ms) but later stabilized. These queries typically list subcollections, which Firestore does not index efficiently.

- COLLECTION /chats/*/messages ORDER_BY timestamp ASC (Red Line)
  Has consistent latency (~20ms) but spikes occasionally. Possible cause: High volume of messages or missing an index on timestamp.

  Most Queries Perform Well (Under 25ms):
- COLLECTION /chats WHERE users[*] = ? (Green Line) Fastest query (~9ms on average). Shows stable performance, which suggests that indexes are working well.

Fault Tolerance & Disaster Recovery

To ensure data durability and availability, we have enabled Firestore's disaster recovery features in Google Cloud. Firestore provides automatic multi-region replication, meaning our chat messages and user data are stored redundantly across multiple geographical locations. This ensures that in case of regional failures, outages, or unexpected disasters, data remains accessible with minimal downtime. (**Resilience to Failures**)

Additionally, Firestore offers point-in-time recovery (PITR) and **backup & restore capabilities**, which allow us to restore data in case of accidental deletions or corruption. By enabling these features, we enhance the resilience of our chat application and minimize the risk of data loss.

Scalability & Resource Allocation

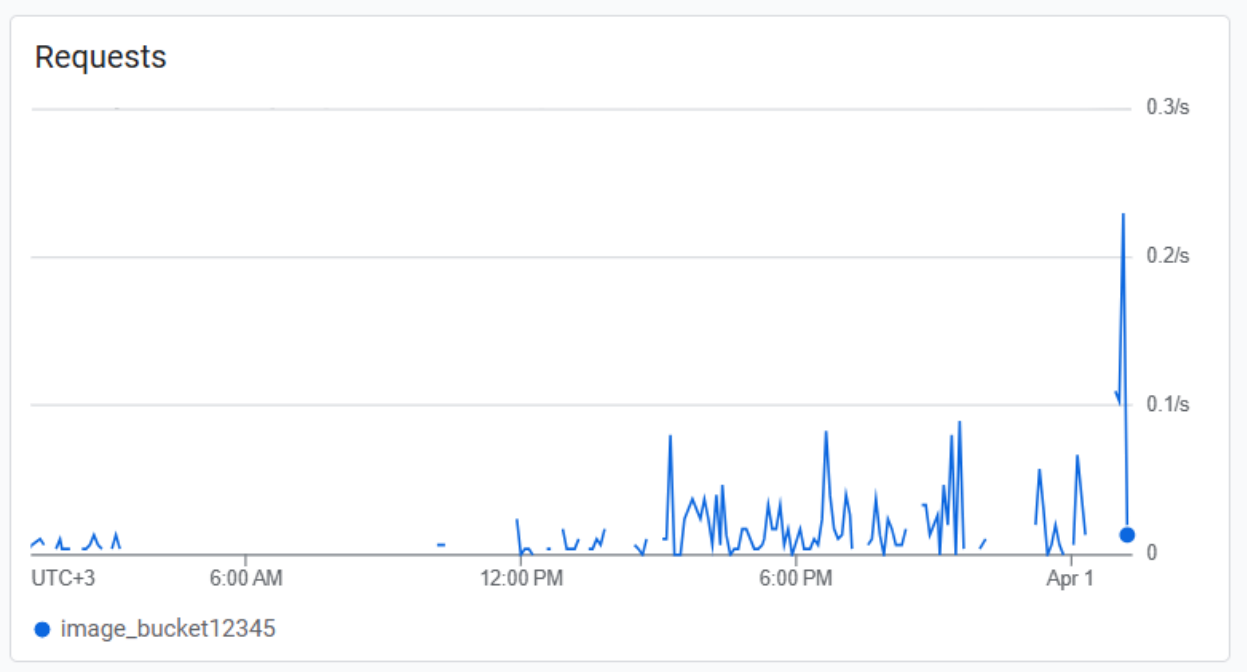
The application relies on Google Cloud's infrastructure, benefiting from automatic scalability without manual intervention. Firestore can handle millions of concurrent users without performance degradation.

Resource Allocation & Optimization

Serverless architecture, meaning resources are dynamically allocated as the application scales.

2. Cloud Storage

Latency



- Provides high-speed data retrieval, with low latency (<100ms in most regions) for media file access.
- Read/Write operations for images typically complete within 0.2s.
- Uses CDN caching to optimize delivery speed for frequently accessed files.
- Supports signed URLs, allowing temporary, secure access to media without unnecessary API calls.

Fault Tolerance & Disaster Recovery

- Automatically replicates data across multiple regions, ensuring data redundancy and protection against failures.
- Soft Delete Policy enabled, allowing data recovery within a defined retention period.
- Supports lifecycle policies for automated backups and storage optimization

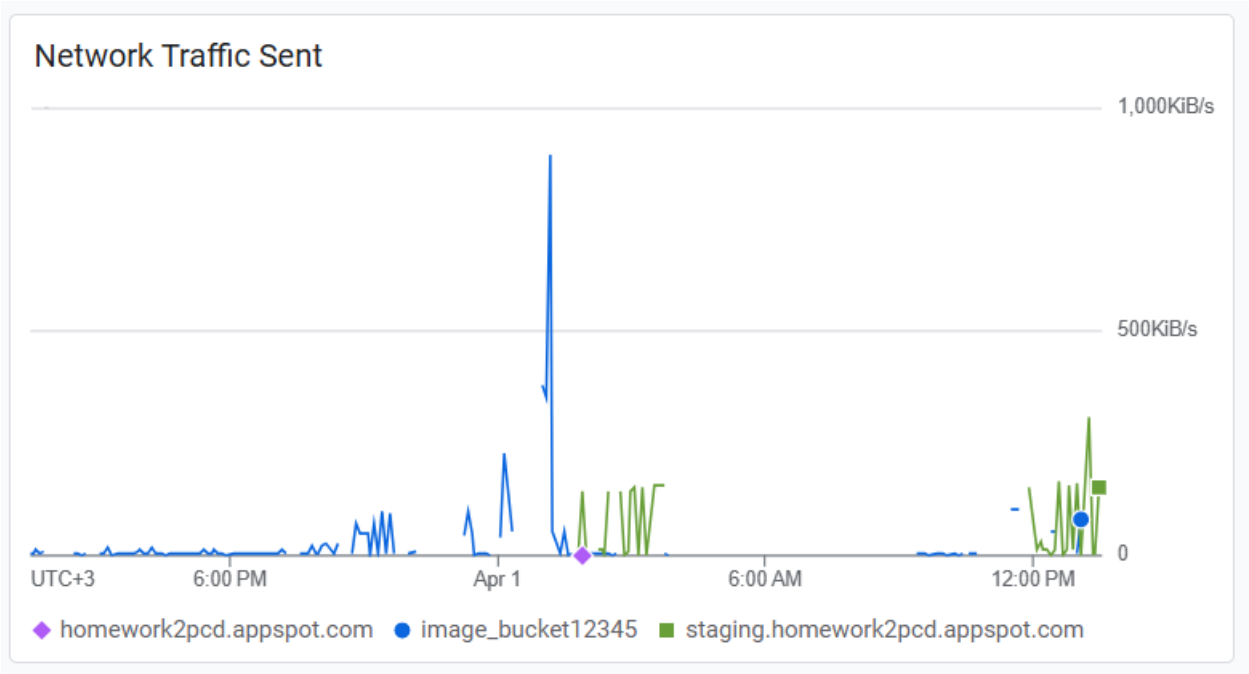
Scalability & Resource Allocation

- Can handle terabytes of data, automatically scaling with demand.
- No manual provisioning needed - storage expands dynamically.

Sustainability & Resource Optimization

- Supports auto-tiering, moving older files to cheaper storage options.
- Only uses resources when data is accessed or modified, reducing unnecessary compute usage.

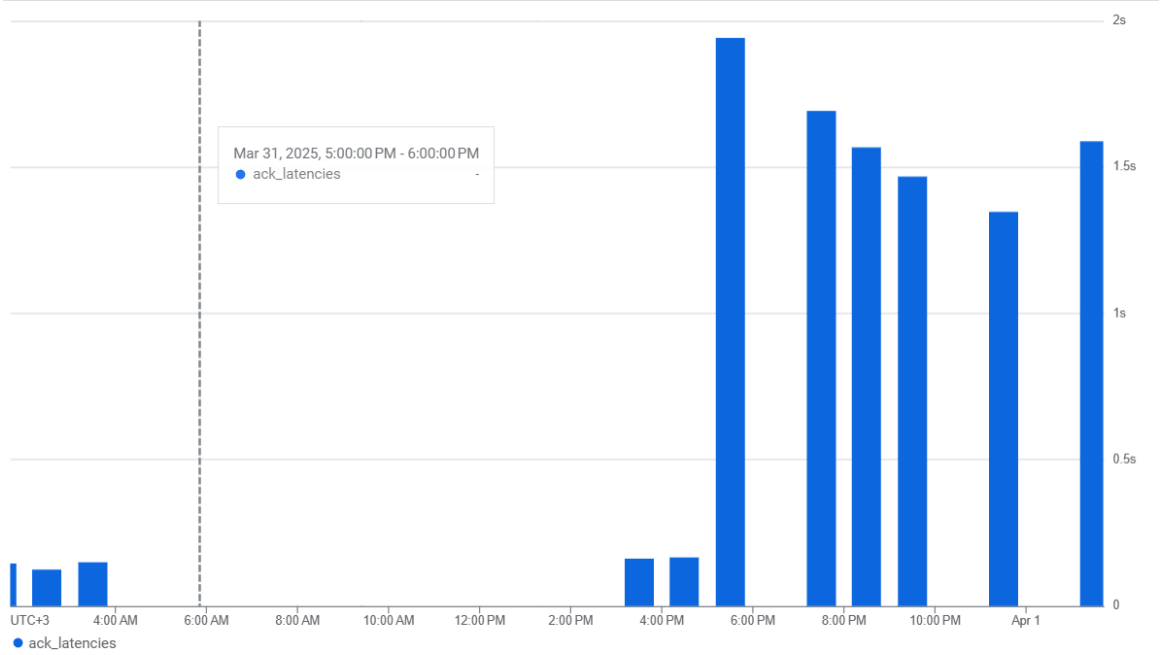
Network Traffic Sent



3. Pub/Sub

Latency

- Delivers messages with latency under 100ms, ensuring near real-time event processing.
- Supports asynchronous event-driven messaging, reducing bottlenecks in communication.



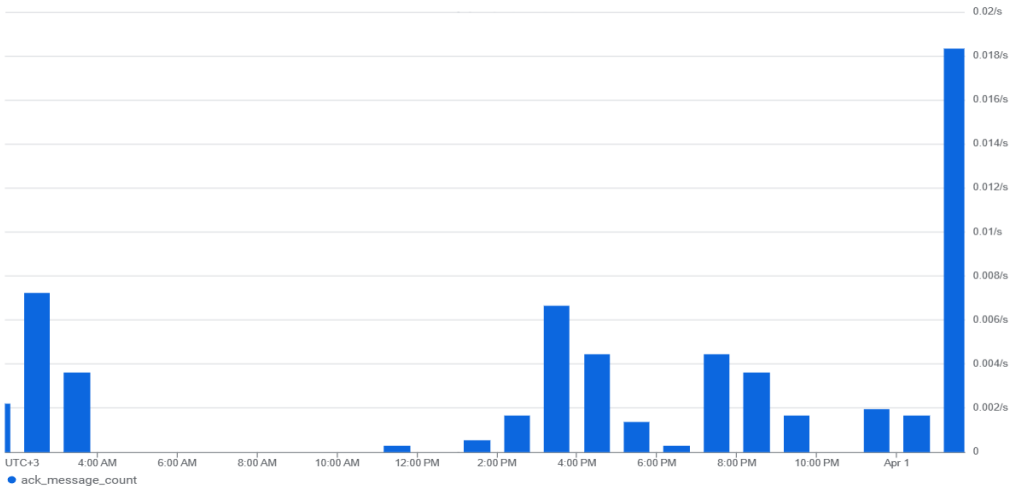
Fault Tolerance & Disaster Recovery

- Implements message retention & dead-letter queues, preventing message loss in case of failures.
- Ensures at-least-once delivery, meaning that even in case of temporary downtime, messages are not lost.

Scalability & Resource Allocation

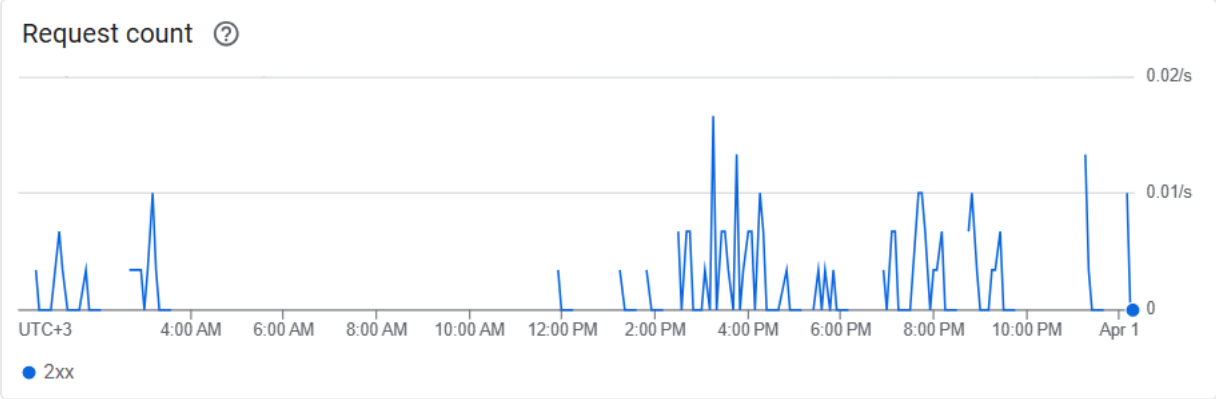
- Automatically scales to support millions of messages per second.
- Supports horizontal scaling, distributing load across multiple subscribers.

Ack message count - Cumulative count of messages acknowledged by Acknowledge requests, grouped by delivery type.

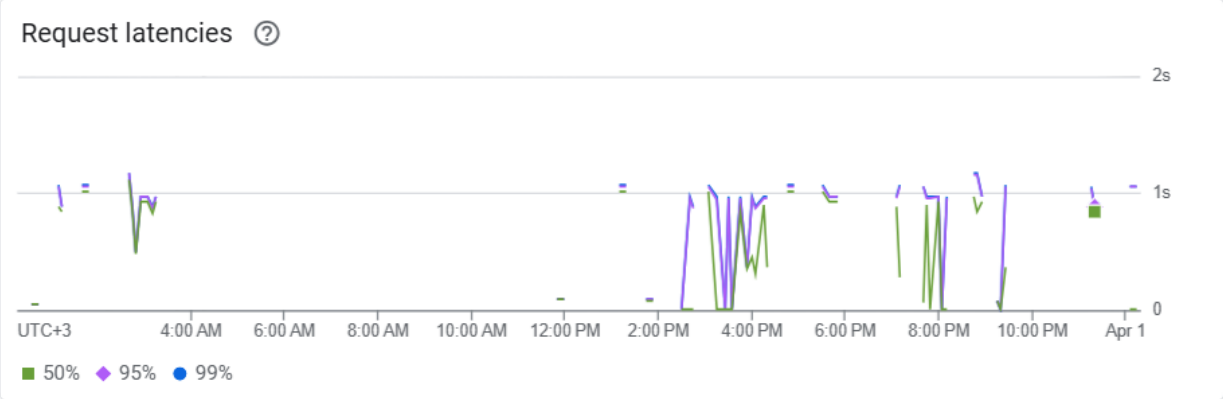


4. FaaS - process-sentiment

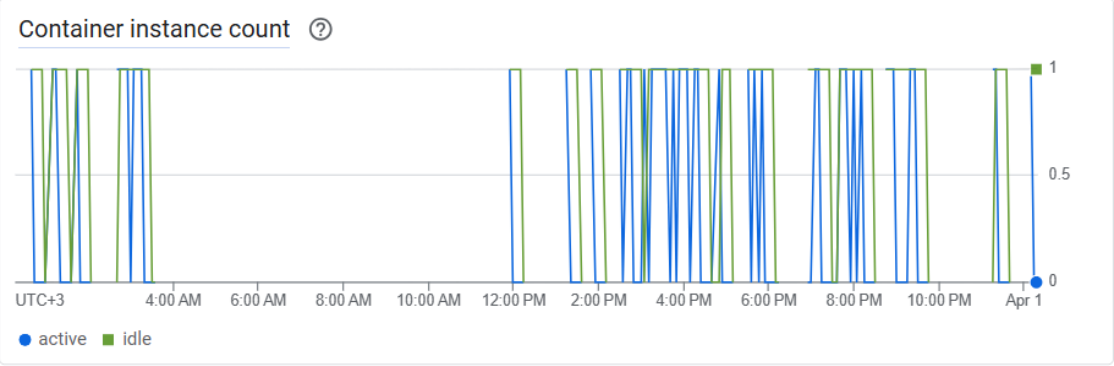
Request count - Number of requests reaching the service:



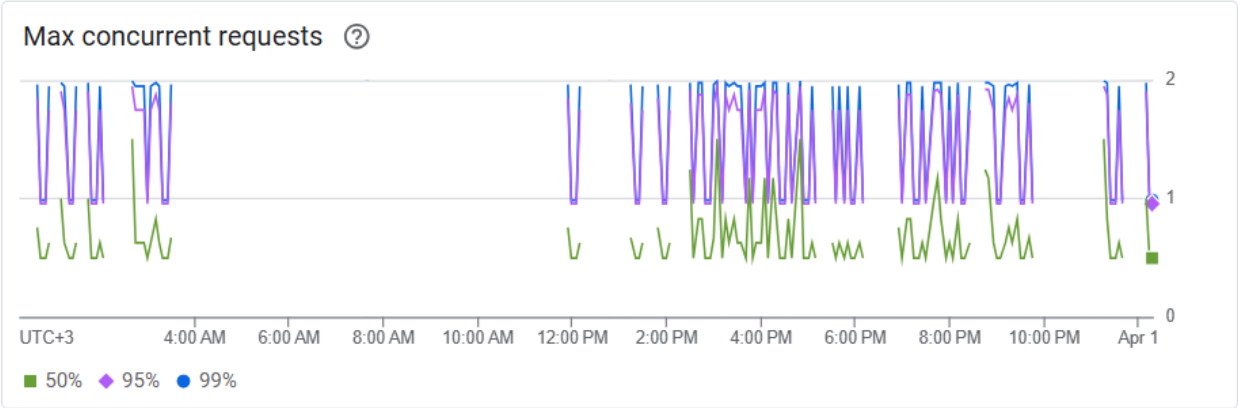
Request Latencies - Distribution of request times in milliseconds reaching the service:



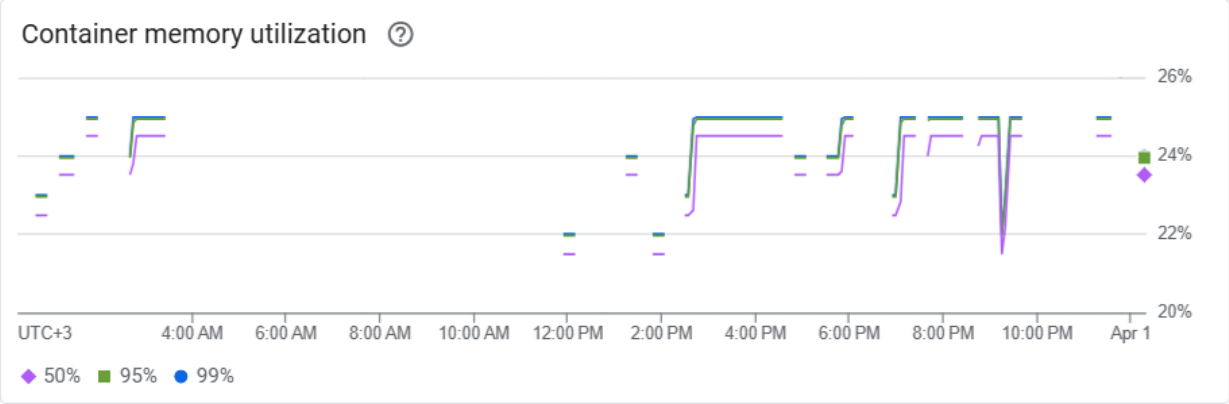
Container instance count - Number of container instances that exist for the service:



Max concurrent requests - Distribution of the maximum number of concurrent requests being served by each container instance over a minute:

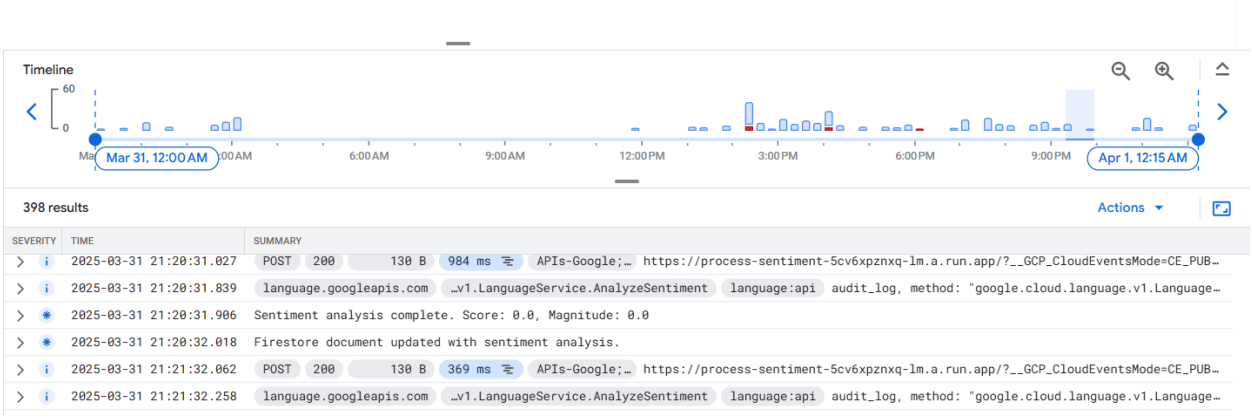


Container memory utilization - Container memory utilization distribution across all container instances:



Performance-wise the function displays impressive request latency times, one request taking a maximum of 1 sec to receive an answer. In case of high request volumes, the component is automatically scalable, creating new instances of the function to handle incoming requests.

API performance and execution



Modularity in Our Application

Our application follows a modular architecture, meaning its components are designed to function independently while communicating efficiently. This improves scalability, maintainability, and fault tolerance.

Benefits of Modularity

- **Scalability** – Each module can be scaled independently based on demand. For example, if message storage becomes a bottleneck, we can optimize Firestore without affecting real-time messaging.
- **Fault Tolerance** – A failure in one module (e.g., email notifications via Pub/Sub) does not impact real-time messaging.
- **Maintainability** – Code changes in one module do not require modifying the entire application. For instance, we can upgrade WebSockets without altering the storage logic.
- **Flexibility & Portability** – If we decide to migrate from Firestore to another database or change our cloud provider, we only need to modify the respective module without rewriting the entire application.

Amdahl's Law

Amdahl's Law is used to evaluate the maximum improvement in system performance when parts of a system are parallelized.

How It Applies to Our Application

1. WebSockets for Real-Time Messaging

Parallelized Component: WebSockets allow multiple users to communicate in real time. Each connection operates independently, supporting parallelism.

Sequential Bottleneck: Writing messages to Firestore is a limiting factor because it requires atomic transactions. Even with multiple Firestore nodes, write operations must be consistent and cannot be parallelized indefinitely.

2. Google Pub/Sub for Event-Driven Notifications

Parallelized Component: Multiple subscribers (such as notification services) can process these events in parallel.
Sequential Bottleneck: Ensuring message delivery order (e.g., notifications should not arrive out of order).

3. Cloud Storage for Media Handling

Parallelized Component: Google Cloud Storage handles uploads in parallel by distributing them across multiple nodes.
Sequential Bottleneck: Metadata processing and API calls still need to happen sequentially before file access is granted.

Amdahl's Law highlights that while parallelism significantly improves performance, it is ultimately limited by sequential bottlenecks such as database writes, metadata operations, and external service dependencies.

Comparing Our Chat Application Architecture with Twitter's Architecture

1. **Real-Time Messaging (WebSockets vs. Fanout Systems)**
- **Our App:** Uses WebSockets for real-time chat between users.
- **Twitter:** Uses Fanout Architecture—tweets are fanned out to followers in real-time. This means instead of sending each tweet as a broadcast to millions of users, they precompute which users need to receive the update.
- **Commonality:** Both architectures focus on low-latency messaging, but Twitter's fanout system is optimized for one-to-many communication
2. **Data Storage (Firestore vs. Twitter's Distributed Storage)**
- **Our App:** Uses Firestore (NoSQL, document-based, serverless).
- **Twitter:** Uses a mix of Cassandra (NoSQL) + Manhattan (custom DB) for tweets and a Redis caching layer for fast retrieval.
- **Commonality:** Both systems focus on high availability and low latency but handle data sharding differently:
    - **Firestore:** Automatically scales and replicates data across regions.

    - **Twitter:** Uses custom partitioning and replication strategies to handle billions of tweets.

3. **Scalability & High Availability**

- **Our App:** Scales using Google Cloud's serverless model (Firestore, Pub/Sub, Cloud Storage).
- **Twitter:** Uses custom-built distributed systems that scale across multiple data centers.
- **Commonality:** Both ensure fault tolerance via replication and distributed architecture, but Twitter's approach is more complex due to massive data volumes.

4. **Real-Time Notifications and Messaging**

- **Our App:** Uses Google Pub/Sub for asynchronous event handling, particularly for sending email notifications when a message is received.
- **Twitter:** Uses Kafka for event streaming (e.g., for analytics, notifications, and data pipelines).
- **Difference:** Twitter's event streaming system handles large-scale data distribution, while our Pub/Sub setup is focused on background processing (triggering emails).
- **Commonality:** Both use event-driven messaging to process tasks asynchronously, preventing delays in the main application flow.

What Can We Learn from Twitter?

- **Optimizing message distribution:** If we scale up, we might consider precomputing message distribution lists (similar to fanout).
- **Using a caching layer:** For frequent messages, we could introduce Redis to reduce Firestore read operations.
- **Enhancing message queueing:** If Pub/Sub becomes a bottleneck, we might explore Kafka-like event streaming for even lower latency.