Cloud Computing Systems Project 1 Performance Evaluation and Analysis

João Oliveira [61052]
Guilherme Franco [60226]

November 10, 2024

Contents

Chapter 1

Introduction

1.1 Project Overview

Our project consists in porting an existing web application - Tukano - to the Microsoft Azure Cloud platform. To that end, the centralized solution that is provided was modified to leverage the Azure PaaS portfolio, in ways that agree with current cloud computing engineering best practices. All this was done in order to make the application scalable, faster and highly available.

1.2 Scope and Objectives

This report will explore what components of Azure PaaS we chose and how these affected the performance of the web application.

We will use the base web application as a benchmark and then compare the values with our ported solution. By using the base TuKano project as a benchmark we will be able to objectively see exactly how our solution improves our not the base project.

Chapter 2

Ported Solution Design

2.1 Solution Architecture

TuKano is organized as a three-tier architecture, where the application-tier, comprises three REST services:

Users - for managing users individual information;

Shorts - for managing the shorts metadata and the social networking aspects, such as users feeds, user follows and likes.

Blobs - for managing the media blobs that represent the actual videos.

We leveraged these azure components:

- Azure Blobs.
- Azure CosmosDB;
- Azure Redis Cache;
- Azure Functions;

We mostly focused on optimising the database and storage.

- We added a Cache (Azure Cache for Redis) for our users and Shorts with a TTL associated with each key.
- Our Database was changed to leverage Azure services, namely Azure CosmosDB NoSQL and Azure CosmosDB for postgreSQL. This was implemented for the Users and Shorts database.
- We also changes the Blob storage method to leverage the Azure Blob Storage component

Finally, we implemented the Views functionality - Shorts metadata is extended with total views statistics, refreshed as fast as feasible.

- In order to achieve this we leveraged Azure functions, namely HTTP Triggers.

2.2 Design Choices and Rationale

We will now go over each of our Design choices and why we implemented them.

2.2.1 Azure Blob Storage

Our rationale was essentially to make the storage scalable and to improve performance in accesses to the storage. The previous implementation was using a local File storage to store the Blobs. Our current implementation uses Azure's Blob Storage.

How this helps:
- Blob Storage is optimized for large, unstructured data.
- Azure's global data centers make it possible to replicate data closer to users worldwide, reducing latency compared to local storage.
- Azure Blob Storage is designed to handle massive amounts of unstructured data (e.g., images, videos, backups) and scales automatically. Unlike local storage, there's no need to worry about disk capacity limits.
- Storage scales with usage, meaning as your application grows, Blob Storage can expand to meet demand without manual intervention or infrastructure upgrades.

Knowing all this we changed the Blob implementation to use Azure's Blob Storage service instead.

2.2.2 Azure Cache for Redis

We decided to implement a cache in the application layer in order to improve latency significantly by avoiding direct accesses to the database, since these usually take a lot more time.

How this helps:
- Redis Cache stores data in memory rather than on disk, providing ultra-fast access to data. This makes it an ideal solution for frequently accessed data, such as session data or caching API responses.
- Redis offers versatile data structures (such as strings, hashes, lists, sets, and sorted sets) that are optimized for quick access and efficient processing, making it well-suited for caching scenarios.

We decided to implement the Cache for storing Users and Shorts, as well as the Likes associated with each short. We avoided using the cache when it came to getting more than one Short at the same time due to how it was stored in the database (needing to go through every Short in the database).

2.2.3 CosmosDB

We changed the database of the base solution to leverage both Azure CosmosDB **NoSQL** and Azure CosmosDB for **postgreSQL**. The one being used is controlled using an environmental variable.

How this helps:
- Cosmos DB replicates data across multiple Azure regions with built-in features for data consistency, fault tolerance, and automatic failover.
- With its multi-region replication and low-latency design, Cosmos DB provides a 99th percentile latency of under 10 milliseconds for both reads and writes.

CosmosDB NoSQL vs CosmosDB for postgreSQL:

These are some key differences between the two database's.

Cosmos DB NoSQL (Core API): Supports a document-based NoSQL model and is compatible with the SQL API. Ideal for unstructured or semi-structured data, like JSON documents.

Cosmos DB for PostgreSQL: Provides a distributed, relational data model and is fully compatible with the PostgreSQL API. Ideal for applications that need relational data modeling and require horizontal scaling.

Cosmos DB NoSQL: Uses a SQL-like query language that works with JSON documents. While flexible, it may not support complex relational joins as PostgreSQL does.

Cosmos DB for PostgreSQL: Supports the full range of PostgreSQL's relational query capabilities, including complex joins, aggregations, and stored procedures, which are beneficial for relational applications.

We decided to connect Hibernate and CosmosDB for postgre SQL in order to preserve most of the original logic. Unfortunately CosmosDB NoSQL doesn't support the complex relational joins that PostgreSQL does and so we had to adapt some of the logic according to which database is being used.

These databases were used to store Users and Shorts.

2.2.4 Total Views statistics

We implemented this feature with the help of HTTP Triggers.

How this helps:

- HTTP-triggered functions are stateless, meaning each invocation is isolated and does not retain data between calls. They're ideal for event-driven workloads that only need to process incoming HTTP requests without maintaining state.

- When a web application experiences traffic spikes, HTTP-triggered functions automatically scale out to handle the increased load, removing the need for manual intervention. This makes them highly resilient for applications that might experience sudden demand surges.

When a user downloads a Blob this activates our trigger that calls a function that updates the number of views.

Chapter 3

Performance Evaluation

3.1 Evaluation Metrics and Methodology

We used Artillery tests to test our solution in comparison to the Tukano Baseline application. We created some of our own tests but opted to use the ones in the SCC github repository to ensure stability and consistency.

3.2 Comparative Analysis with TuKano Baseline

Present a comparison between the ported solution and the TuKano baseline application. Use tables and graphs to illustrate differences in performance metrics.

Table 3.1: Benchmark Results for Each Target and Test Type

Target	**Test**	**Avg Res Time (ms)**	**Request Rate (req/sec)**
Local Deployment	Register user	2.35	30
	Upload shorts	3.15	20
	Realistic Flow	3.35	7
	User Delete	4.00	10
Azure Baseline	Register user	50.25	30
	Upload shorts	97.3	20
	Realistic Flow	58.45	7
	User Delete	52.9	10
No cache Cosmos	Register user	79.75	30
	Upload shorts	110.65	20
	Realistic Flow	210.6	7
	User Delete	503.1	10
No cache Postgres	Register user	86.3	30
	Upload shorts	149.55	20
	Realistic Flow	74.25	7
	User Delete	188.25	10
Cache Postgres	Register user	663.7	30

Continued on next page

Target	**Test**	**Avg Res Time (ms)**	**Request Rate (req/sec)**
	Upload shorts	241.0	20
	Realistic Flow	2165.0	7
	User Delete	89.6	10
Cache Cosmos	Register user	467.5	30
	Upload shorts	227.2	20
	Realistic Flow	278.6	7
	User Delete	-	-

3.2.1 CosmosDB vs Postgres

Table 3.2: Comparison Between Cosmos and Postgres Targets for Each Test Type

Target Type	**Test**	**Avg Res Time (ms)**	**Cosmos vs Postgres (%)**
No cache Cosmos	Register user	79.75	-7.60%
No cache Postgres		86.3	Reference
	Upload shorts	110.65	-26.04%
		149.55	Reference
	Realistic Flow	210.6	+183.52%
		74.25	Reference
	User Delete	503.1	+167.39%
		188.25	Reference
Cache Cosmos	Register user	467.5	-29.57%
Cache Postgres		663.7	Reference
	Upload shorts	227.2	-5.73%
		241.0	Reference
	Realistic Flow	278.6	-87.13%
		2165.0	Reference
	User Delete	-	-
		89.6	Reference

- **No Cache - Register User:**
  Cosmos has an average response time of 79.75 ms, which is 7.6% faster than Postgres at 86.3 ms. This indicates that for registering a user without caching, Cosmos is slightly more efficient than Postgres.

- **No Cache - Upload Shorts:**
  Cosmos performs this operation in 110.65 ms, which is 26.04% faster than Postgres (149.55 ms). The substantial difference suggests that Cosmos handles this data upload more efficiently, likely due to differences in write optimizations between the two databases.

- **No Cache - Realistic Flow:**
  Cosmos has an average response time of 210.6 ms, which is 183.52% slower than Postgres (74.25 ms). This result shows that Cosmos significantly underperforms in a complex workflow scenario when compared to Postgres, indicating that Cosmos may struggle with multi-step or compound operations without caching.

- **No Cache - User Delete:**
  Cosmos takes 503.1 ms to delete a user, making it 167.39% slower than Postgres at 188.25 ms. This indicates that Cosmos is slower in handling deletion operations, which could suggest a limitation in its transactional handling for such tasks compared to Postgres.

- **Cache - Register User:**
  With caching enabled, Cosmos completes the operation in 467.5 ms, which is 29.57% faster than Postgres (663.7 ms). This shows a more noticeable improvement in Cosmos's response time over Postgres when caching is used, likely due to optimized retrieval for repeated operations.

- **Cache - Upload Shorts:**
  Cosmos completes this operation in 227.2 ms, which is 5.73% faster than Postgres at 241.0 ms. Caching improves Cosmos's efficiency slightly, although the difference is not as marked as in other tests.

- **Cache - Realistic Flow:**
  Cosmos takes 278.6 ms to complete this workflow, which is 87.13% faster than Postgres at 2165.0 ms. This suggests that Cosmos performs exceptionally well in complex operations with caching, significantly reducing time compared to Postgres.

- **Cache - User Delete:**
  Only the Postgres time is provided (89.6 ms), while Cosmos is not measured or does not support caching for deletions in this test.

Summary of Findings

- **Cosmos DB generally benefits from caching,** especially for complex operations like the Realistic Flow test.

- **Without caching, Cosmos outperforms Postgres** in some simpler operations, like Register User and Upload Shorts.

- **Cosmos performs slower than Postgres** in complex operations without caching, indicating potential limitations in handling certain multi-step or transactional tasks.

- **Postgres is more consistent across different operations,** while Cosmos shows significant variance in performance depending on caching and the type of operation.

Chapter 4

Discussion

4.1 Impact of Design Choices on Performance

Due to exceeding the credit limit on Azure our group was unable to extensively test our solution although we did manage to get some tests done. These were the most impactful design choices for improving our response times and latency:

Azure Blob Storage for Media Content

By leveraging Azure Blob Storage, we observed a significant reduction in latency for media retrieval compared to the local file system. Blob Storage's scalability also ensured that response times remained consistent, even with increased load.

Azure Redis Cache for Frequently Accessed Data

Redis Cache's in-memory storage allows data to be retrieved almost instantaneously, avoiding time-consuming database queries for each user request. For example, caching user profiles and frequently accessed shorts metadata provided a drastic reduction in access times by keeping data in memory and bypassing the database layer.

Choice Between Cosmos DB NoSQL and Cosmos DB for PostgreSQL

Supporting both Cosmos DB NoSQL and Cosmos DB for PostgreSQL allowed us to compare response rates for different types of data models. Cosmos DB NoSQL, being optimized for key-value and document-based storage, provided faster access for simpler, non-relational data. However, Cosmos DB for PostgreSQL proved better for handling complex, relational data operations, where structured queries and joins were necessary.

4.2 Challenges and Limitations

Due to restrictions on the amount of credits we had allocated we were unable to complete all the tests we would have liked. Another unfortunate challenge was that when approaching the end of our credits Azure's Servers started to timeout. We believe this was due to Caps put in place so we wouldn't go over the allocated number of credits.

Due to the fact that Cosmos Postgre and Cosmos DB NoSQL are very different we had to essential code 2 different databases at the same time in parallel.

Chapter 5

Conclusion

5.1 Summary of Findings

Our performance evaluation showed significant improvements in latency and request throughput in several areas. Azure Blob Storage, for instance, proved to be an effective solution for managing the application's large, unstructured media data. It provided scalable and low-latency access to blobs, allowing faster uploads and retrievals. Additionally, Azure Redis Cache reduced database load and improved data retrieval speed, especially for frequently accessed data like user sessions and metadata for "shorts." Azure Cosmos DB also demonstrated resilience and high availability, though our comparative analysis highlighted the limitations of CosmosDB NoSQL for complex relational queries, which were better handled by Cosmos DB for PostgreSQL.

In conclusion, this project demonstrated the advantages and trade-offs of porting an existing web application to Azure PaaS. While the migration improved scalability and availability, it required careful consideration of the trade-offs between service features and the application's performance goals.

5.2 Future Work

There were a couple of features we wanted to finish but weren't able to, namely:

- Tukano Recomends - Every user automatically follows a system managed user named "Tukano Recomends". This user will republish selected content from the collection of shorts publish by general TuKano userbase.
- Geo-Replication support - The solution has support for geo-replicated deployment.
- The use of Spark Computation to implement our solution.

5.3 Use of AI tools

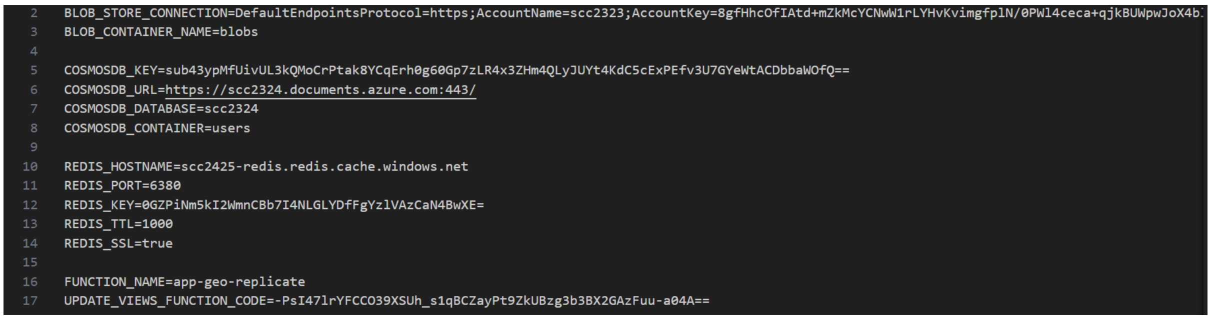
In order to faster do our project, we used ChatGPT to help creating some methods, namely on the method `getFeed` to create the logic to use either Postgres or CosmosDB. We also used it to helps us with simulation the transaction logic in CosmosDB. And to generate some REGEX.

Appendix A

Appendix

A.1 Code Snippets and Configurations

Include any important code snippets, configuration details, or setup instructions relevant to the project.



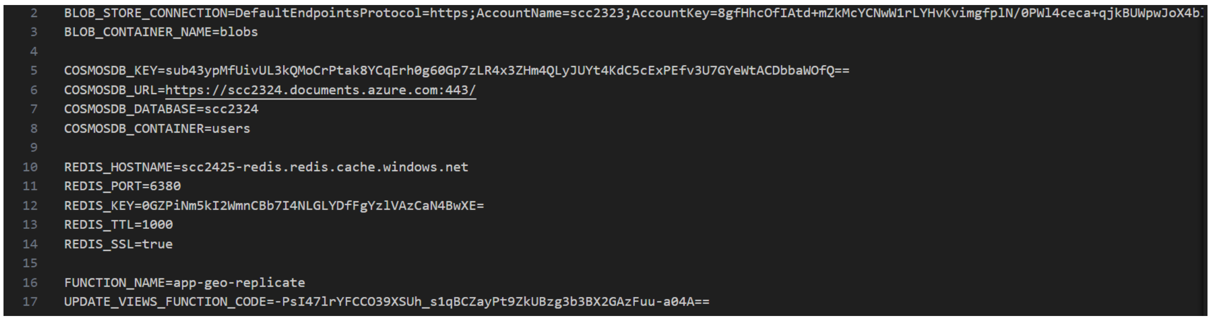
```
BLOB_STORE_CONNECTION=DefaultEndpointsProtocol=https;AccountName=scc2323;AccountKey=8gfHhcOfIAtd+mZkMcYCNwW1rLYHvKvimgfplN/0PWl4ceca+qjkBUWpwJoX4b
BLOB_CONTAINER_NAME=blobs

COSMOSDB_KEY=sub43ypMfUivUL3kQMoCrPtak8YCqErh0g60Gp7zLR4x3ZHm4QLyJUYt4KdC5cExPEfv3U7GYeWtACDbbaWOfQ==
COSMOSDB_URL=https://scc2324.documents.azure.com:443/
COSMOSDB_DATABASE=scc2324
COSMOSDB_CONTAINER=users

REDIS_HOSTNAME=scc2425-redis.redis.cache.windows.net
REDIS_PORT=6380
REDIS_KEY=0GZPiNm5kI2WmnCBb7I4NLGLYDfFgYzlVAzCaN4BwXE=
REDIS_TTL=1000
REDIS_SSL=true

FUNCTION_NAME=app-geo-replicate
UPDATE_VIEWS_FUNCTION_CODE=-PsI47lrYFCCO39XSUh_s1qBCZayPt9ZkUBzg3b3BX2GAzFuu-a04A==
```

Figure A.1: Hibernate configuration file



```
BLOB_STORE_CONNECTION=DefaultEndpointsProtocol=https;AccountName=scc2323;AccountKey=8gfHhcOfIAtd+mZkMcYCNwW1rLYHvKvimgfplN/0PWl4ceca+qjkBUWpwJoX4b
BLOB_CONTAINER_NAME=blobs

COSMOSDB_KEY=sub43ypMfUivUL3kQMoCrPtak8YCqErh0g60Gp7zLR4x3ZHm4QLyJUYt4KdC5cExPEfv3U7GYeWtACDbbaWOfQ==
COSMOSDB_URL=https://scc2324.documents.azure.com:443/
COSMOSDB_DATABASE=scc2324
COSMOSDB_CONTAINER=users

REDIS_HOSTNAME=scc2425-redis.redis.cache.windows.net
REDIS_PORT=6380
REDIS_KEY=0GZPiNm5kI2WmnCBb7I4NLGLYDfFgYzlVAzCaN4BwXE=
REDIS_TTL=1000
REDIS_SSL=true

FUNCTION_NAME=app-geo-replicate
UPDATE_VIEWS_FUNCTION_CODE=-PsI47lrYFCCO39XSUh_s1qBCZayPt9ZkUBzg3b3BX2GAzFuu-a04A==
```

Figure A.2: Props file with keys