NOVA SCHOOL OF SCIENCE & TECHNOLOGY

SCC Report

Sistemas de Computação Cloud

2024/2025

TuKano

Authors:

69369, Aurélio Miranda
70068, Rafael Pires

Lab class Nº P1

Professor:
Sérgio Duarte

Introduction

This project aims to explore how cloud computing platforms, specifically Microsoft Azure, can be utilized to create applications that are scalable, responsive, and highly available. By leveraging Azure's Platform as a Service (PaaS) offerings, this project seeks to modernize an existing web application, TuKano, to meet the demands of a broad and growing user base.

While we aimed to implement a fully functional cloud-native version of TuKano, challenges arose due to limited familiarity with the application's internal workings, which affected the ability to implement certain features fully. However, the project demonstrates key cloud computing principles using Azure's services.

Architecture Design

TuKano's original architecture: three-tier structure with Users, Shorts, and Blobs services. That was not scalable and had no replication or caching implemented which can cause a multitude of issues, especially to users located far from the server.

Azure PaaS Integration: Aims to solve this issues by providing scalable and fast services that leverage Azures portfolio.

- **Blob Storage**: Used for video storage (stores the actual videos).
- **Cosmos DB**: Data storage for user and shorts metadata. Explain the choice between SQL and NoSQL databases.
- **Redis Cache**: Reduces latency in data retrieval.

Implementation Details

- **Data Layer Transformation**:Cosmos DB was used in order to replace hibernate, as a NoSql and fast approach to deal with the Shorts metadata and User data. The transition was pretty straight forward once the setup was done, although the field Id was needed both in Users, Shorts, Followers, etc.. Classes.
- **Caching**: Redis Cache was utilized on selected operations like retrieving User/Short info and more. The use caching was implemented on top of the already existing Cosmos DB implementation with its own needed details in order to maintain consistency (deleting in the database also prompts a delete in the cache, for example).

Performance Analysis

Testing Setup

The testing setup consisted of a couple artillery tests (for performance and load testing) and also postman single tests for assessing functionality of a given operation.

SQL vs. NoSQL Performance

We performed standard inserts and gets to assess speed and reliability and we quickly found out that NoSQL outmatches PostgreSQL when it comes to both speed and reliability.

```
All VUs finished. Total time: 21 seconds

--------------------------------
Summary report @ 16:50:40(+0000)
--------------------------------

errors.ETIMEDOUT: ..................................................... 3
http.codes.200: ....................................................... 35
http.codes.403: ....................................................... 7
http.codes.500: ....................................................... 5
http.downloaded_bytes: ................................................ 42977
http.request_rate: .................................................... 7/sec
http.requests: ........................................................ 50
http.response_time:
  min: ................................................................ 443
  max: ................................................................ 9857
  mean: ............................................................... 3359.3
  median: ............................................................. 2465.6
  p95: ................................................................ 8520.7
  p99: ................................................................ 9416.8
http.response_time.2xx:
  min: ................................................................ 443
  max: ................................................................ 9857
  mean: ............................................................... 2507.3
  median: ............................................................. 1556.5
  p95: ................................................................ 5944.6
  p99: ................................................................ 8868.4
http.response_time.4xx:
  min: ................................................................ 1687
  max: ................................................................ 9412
  mean: ............................................................... 5735.4
  median: ............................................................. 5487.5
  p95: ................................................................ 8520.7
  p99: ................................................................ 8520.7
http.response_time.5xx:
  min: ................................................................ 2461
  max: ................................................................ 7466
  mean: ............................................................... 5997.2
  median: ............................................................. 6702.6
  p95: ................................................................ 7117
  p99: ................................................................ 7117
http.responses: ....................................................... 47
plugins.metrics-by-endpoint./tukano/rest/users/.codes.200: ............. 35
plugins.metrics-by-endpoint./tukano/rest/users/.codes.403: ............. 7
plugins.metrics-by-endpoint./tukano/rest/users/.codes.500: ............. 5
plugins.metrics-by-endpoint./tukano/rest/users/.errors.ETIMEDOUT: ..... 3
plugins.metrics-by-endpoint.response_time./tukano/rest/users/:
```

Figure 1 - Bulk user insert for PostgreSQL

From figure 1 we can see that a certain percentage of users (70% to be exact) are added successfully, this was obviously ran multiple times, but the result is mostly the same with 30% of users not being added, on top of that, it takes a long time to even try to add all the users (21 seconds).

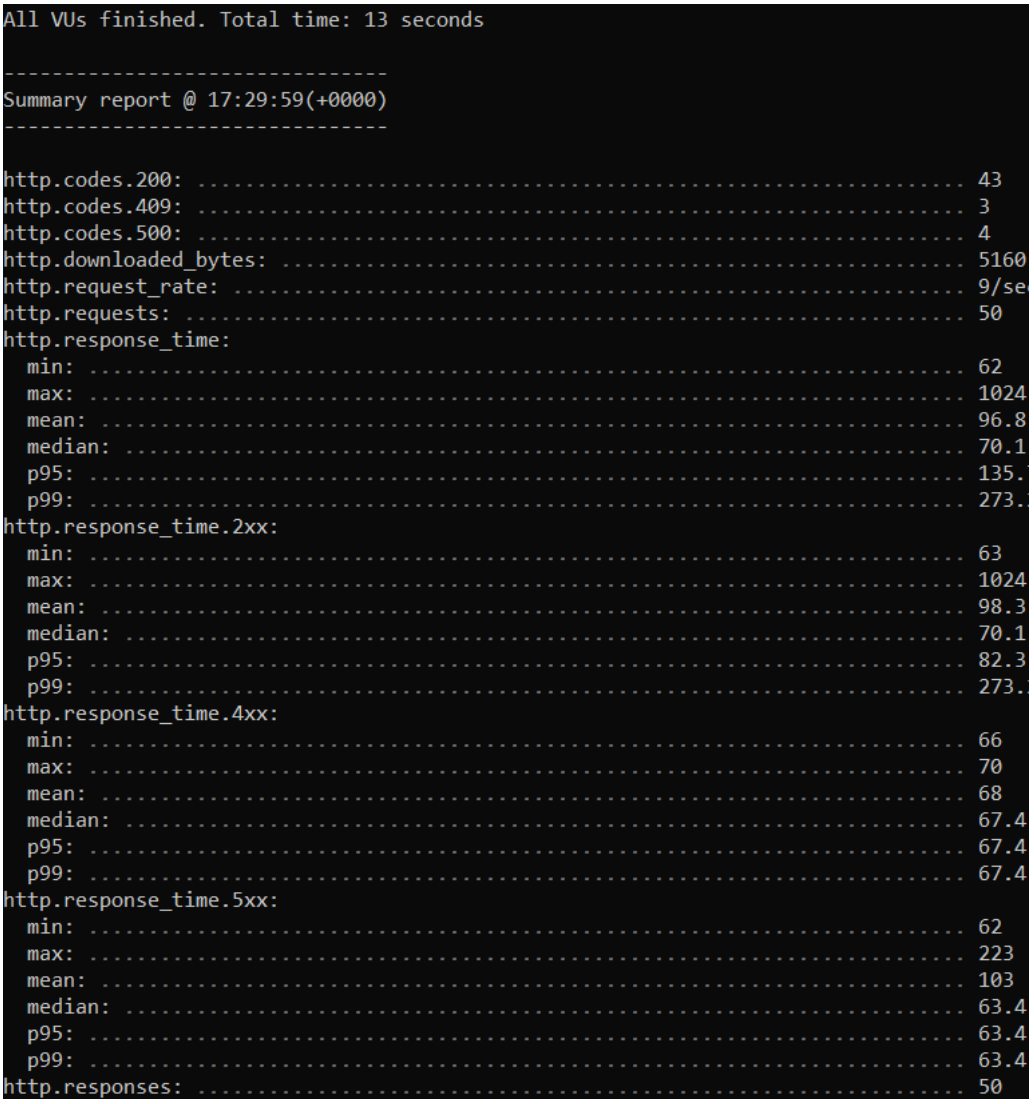

Figure 2 - Bulk user insert for NoSQL

When performing the same process of adding 50 users in a short amount of time, we can see clear differences in the performance and reliability of those additions, for instance, the time was reduced to 13 seconds (on average), the number of successful operations on average also increased (to around 83%).

Additionally, the higher throughput can be associated with NoSQL as it was able to handle 9 requests per second as opposed to the 7 by PostgreSQL.

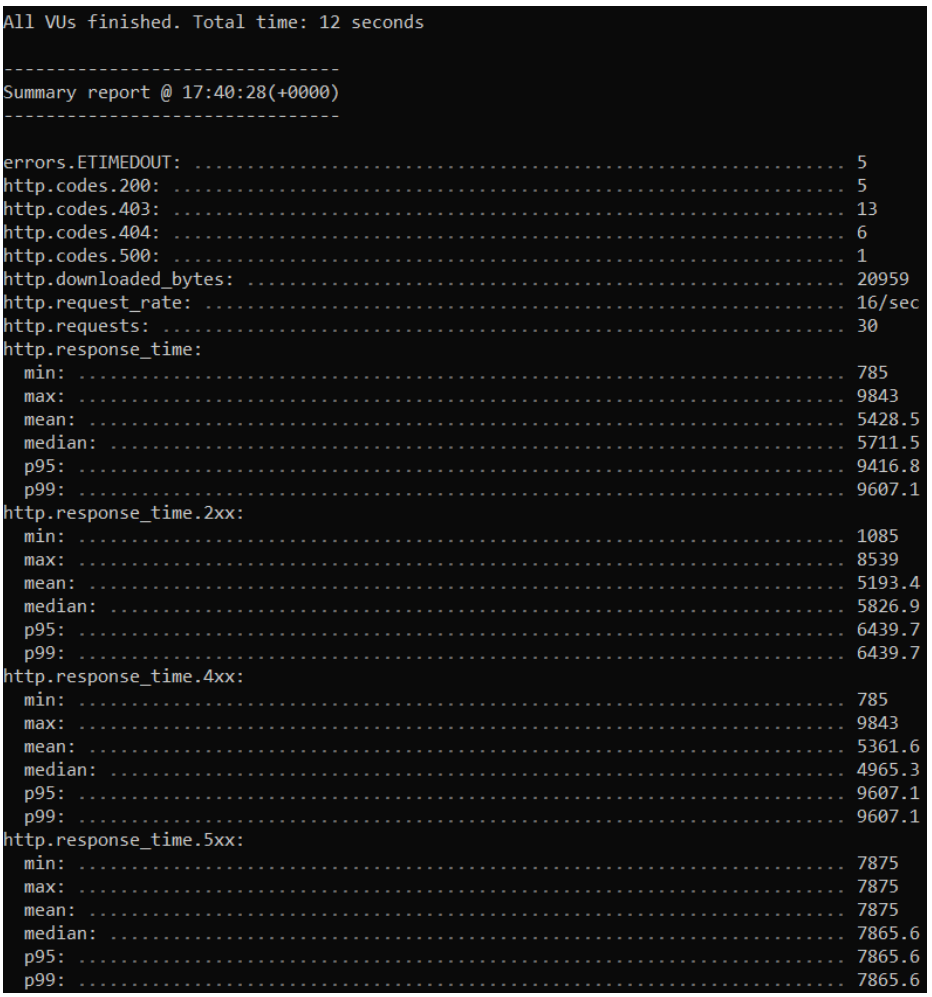

Figure 3 - Bulk user gets for PostgreSQL

We also tested the bulk gets for both PostgreSQL and NoSQL and, as mentioned before, NoSQL has an advantage when it comes to performance.

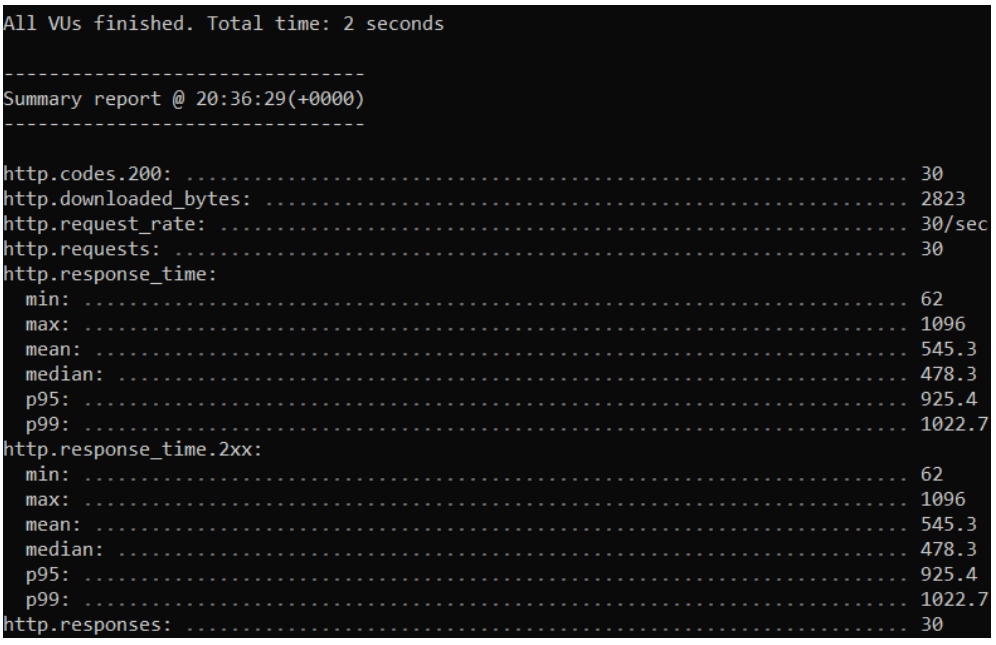

Figure 4 - Bulk user gets for NoSQL

As we can see from figures 3 and 4, NoSQL is able to handle much more efficiently the requests as the success rate is 100%, additionally, the throughput is also very high (30 requests per second) when compared to the PostgreSQL (16 requests per second), revealing that NoSQL handles better high latency situations.

Moreover, while we conducted tests we found out that our SQL implementation is very prone to overloading and, after we reached the connection limits (20) we were unable to perform any calls for some time.

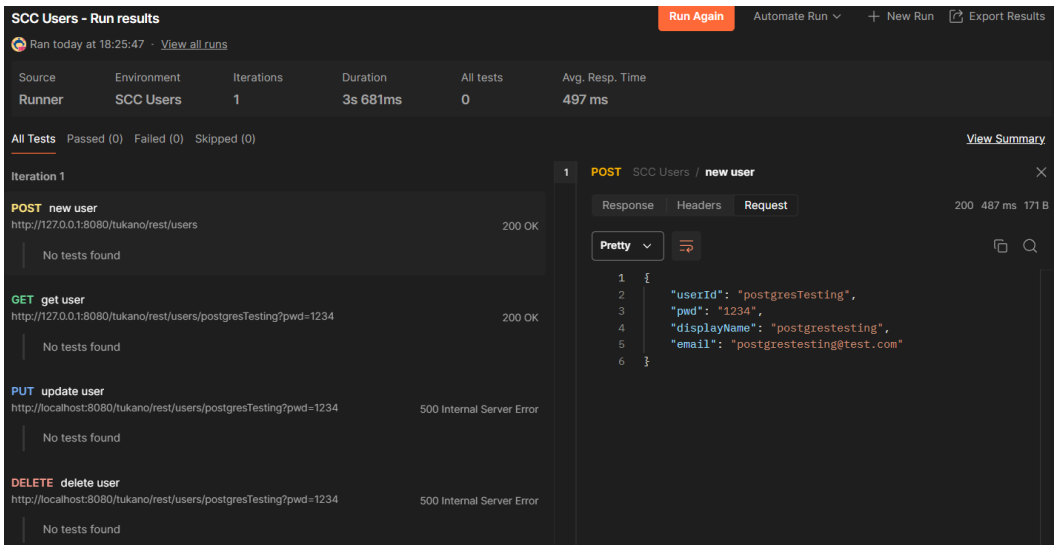

Figure 5 - Testing user functionalities for PostgreSQL

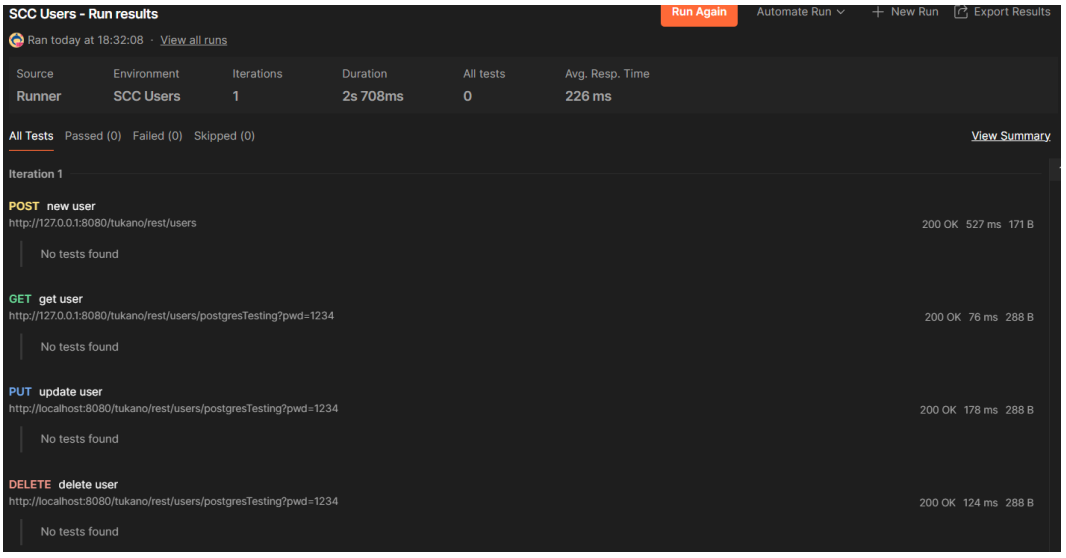

Figure 6 - Testing user functionalities for NoSQL

As we can see in figure 6, there doesn't seem to be an issue with connection limits in NoSQL which is another reason why NoSQL is a better choice for this specific project.

Overall, the choice between NoSQL and SQL comes down to the CAP theorem, so in this case, NoSQL seems to be the better option, offering availability and partition tolerance, in order to optimize performance for the TuKano app, given that it needs to keep users engaged at all times and high performance assures that the app loads data quickly.

SQL is a strong choice when high consistency is needed, however, in this case there might be struggles with scaling in a distributed setting because SQL can introduce complexity.

Impact of Caching

Along with Cosmos DB, caching proved to improve even more the performance of the app when fetching User and Shorts information. Although our implementation was simple, the results clearly favor the use of caching.

In this example we fetched a list of shorts recently added by a user.

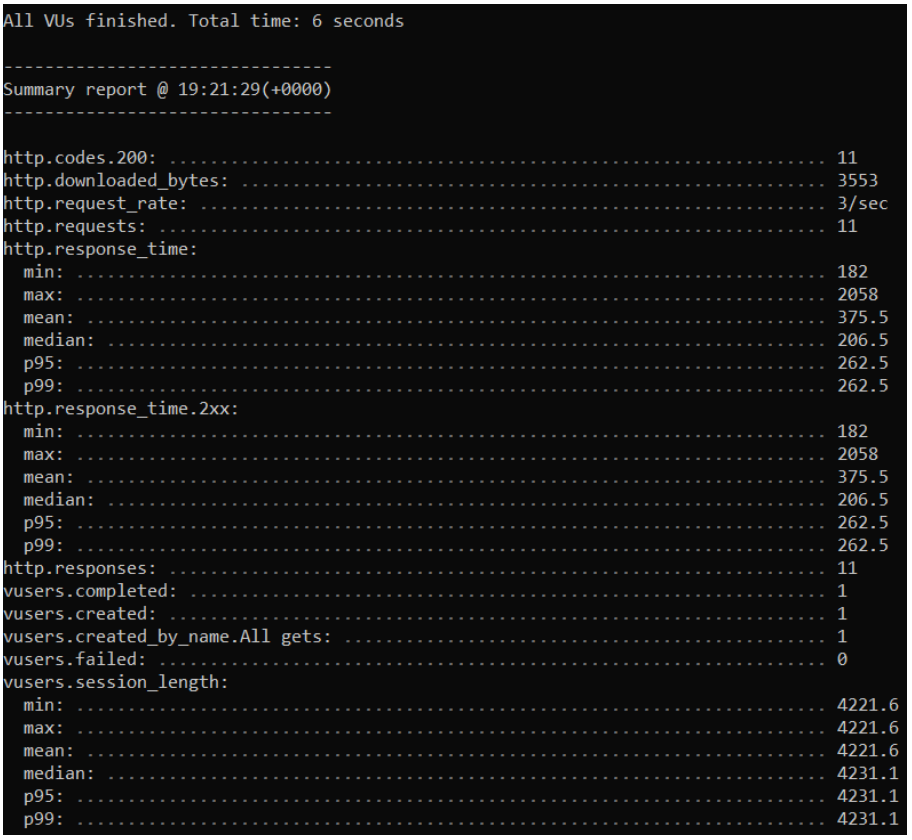

Figure 7 - Fetching shorts without cache

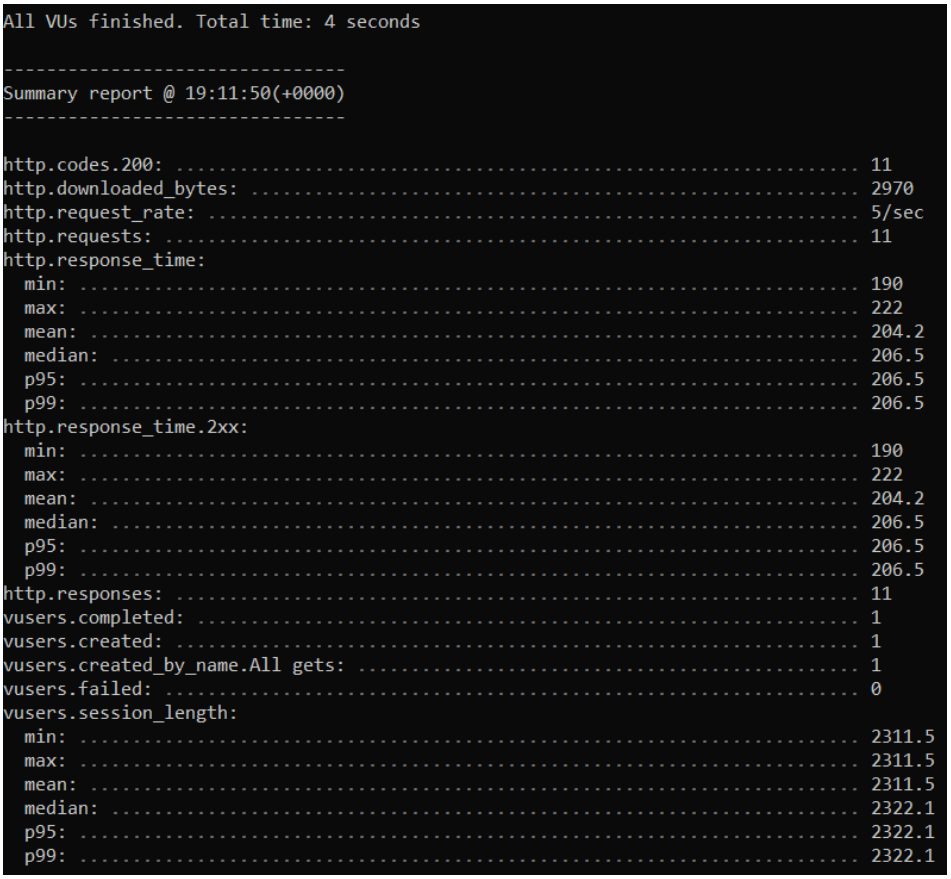

Figure 8 - Fetching shorts with cache

From figures 7 and 8 we can perceive that in general, Redis Cache reduces the response time by 50%, while this example is small (with just 2 seconds of difference) both the throughput and the mean response time is shorter when caching is used, for those reasons, caching is ideal to implement in functionalities like the user feed or the user session, for example.

Evaluation and Discussion

Summary of Results

Conversion to Azure (NoSQL + Redis Cache) includes:

- CRUD operations for Users and Shorts;
- CRD operations for Blobs (short videos);
- Pattern search for User;
- Get user shorts, get user followers.

Conversion to Azure (PostgreSQL) includes:

- CRD operations for Users and Shorts;

From the performance analysis we were able to conclude that NoSQL + Redis Cache is the best approach given the implementation we performed as it delivers a high availability which is a must for any social media application.

Challenges & Improvements

During the project development we encountered a lot of difficulties due to unfamiliarity with the source code and how the project worked since it was a part of a course we did not have (with us being MEI students). Because of this, we had a lot of delays due to compilation errors, figuring out our pom.xml configuration, how the REST requests worked, which ultimately resulted in only being able to do significant work on the project very close to the delivery deadline.

Additionally, we could not get the given base implementation of Tukano working in order to set up a baseline. Running it locally with Docker or in Azure resulted in 404 and 500 errors from REST requests using either Postman or Artillery.

Finally we had initially planned on building Azure Functions that performed the deletion of user resources after the same user was deleted, which would essentially remove majorly the time it would take to delete a user (when deleting from the User container we would simply add to another container, DeletedUsers for example, which would trigger a function that would then delete everything associated with that user).

Conclusion

We were able to utilize Azure to store our users and shorts using NoSQL (with and without Redis Cache) and PostgreSQL (without Redis Cache). Migrating the solution to the cloud resulted in better scalability. In terms of costs, we observed that our storage costs were not very high, less than 1 cent during the development, but the same could not be said for caching costs which had a significantly higher cost compared to our regular storage, around 350x more, thought this was to be expected since cache is low storage capacity, low retrieval times and high cost.

Although we couldn't fully convert all functionalities of the original product to work with Azure, we feel that our solution grasps the fundamentals of utilizing the Azure services.