Project Report Phase 1 - SCC

João Bernardo[62612] and João Silva[70373]

Universidade Nova de Lisboa, Portugal

Abstract. This report details the migration of a locally hosted web application similar to Tiktok, named Tukano, to the cloud using Microsoft Azure. Specifically, it explains the design choices for the application, the resources used in implementation, and evaluations of both the chosen resources and the app's performance in various real-world scenarios.

Keywords: Microsoft Azure Cloud · Web App · Platform-as-a-Service

1 Introduction

1.1 What is Tukano?

Tukano is a web app that implements a social network inspired in existing video sharing services, such as TikTok or Youtube Shorts. TuKano users can upload short videos to be viewed (and liked) by other users of the plataform. The social network aspect of TuKano resides on having users follow other users, as the main way for the platform to populate the feed of shorts each user can visualize.

1.2 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.

1.3 Migration to the Cloud

The migration of Tukano to the cloud using Microsoft Azure is driven by several factors aimed at enhancing performance, scalability, and overall user experience. Initially, Tukano was a locally hosted web application that relied on on-premises infrastructure to handle user requests, manage video uploads and downloads, and store shorts metadata.

Migrating Tukano to Microsoft Azure offers several advantages that address the challenges faced by the previous version, such as:

- **Scalability** - Azure's cloud services allow Tukano to automatically scale resources up or down based on real-time demand, supporting periods of peak usage.
- **Availability** - One of the main advantages of cloud services is the possibility of replicate the app and it's resources in several strategic regions, giving a better user experience for everyone.

2 Implementation

2.1 Azure Blobs

Migration to Azure Blobs was relatively straight forward. The original implementation consisted of uploading blobs into the OS file system. Azure Blob Storage solution provides a similar experience, once the connection with the service is established we are able to work with a container and leverage a simple API (upload(), downloadContent() and delete()), in order to replace the original implementation.

2.2 Azure CosmosDB

As described above, tukano has two services that rely on a database: users and shorts. The original implementation used Hibernate with local storage, an object–relational mapping tool for Java that provides a framework for mapping an object-oriented domain model to a relational database. The downside of using Hibernate with local storage is that it can be hard for scalability for when the app grows, both in terms of performance and availability.

Azure CosmosDB is a cloud-based solution that offers scalability, flexibility and performance benefits, allowing the app to better grow in the real world scenario.

This service offers several API's to be implemented. Our project offers **CosmosDB for NoSQL**, a non-SQL solution and **PostgresSQL**, a SQL based solution.

Data Storage Organization - For the organization of the data within both database solutions, we chose to maintain the original solution with 4 different tables/containers for the **users**, **shorts**, **likes** and **followers**. This revealed to be better for versatility, since this way the same code in *JavaUsers* and *JavaShorts* is used for both solutions.

2.3 Azure Redis Cache

To improve the app performance, we implemented caching of users and shorts to optimize most requests. For that, Azure offers the Redis Cache, which stores key-value pairs, but also lists and other data structures.

For the Users service, we chose to store users individually, in the form **user:userId := user**, for a faster access when getting the user or checking if the user is in the system. We also stored the users in a cached list to improve performance in the searchUsers operation, giving the key name **LIST_USERS**

For the Shorts service, we opted to only store shorts and not likes and followers. To store the shorts, we use three ways to cache them: individually, in the form **short:shortId := short**; shorts by user, in the form **user:userId:shorts := list(shorts)**; and users feed, in the form **user:userId:feed := list(shorts)**.

2.4 Azure Functions

Increment Views - To increase the views of a short when a user downloads the corresponding blob, the approach was to create a http trigger that, when called, retrieves the short from the database and updates the view count. Also, to ensure consistency between the database and cache, the function also updates the cache with the new short.

Tukano Recommends - For the tukanoRecommends function, the approach was to create a Http trigger and then deleting in both cache and database all the previously recommended shorts so as not to display always the same shorts. The function is called every time we call getFeed(), so as to keep the calls to a minimum and have a more targeted use.

Then we take the cached shorts, and reorder them based on most viewed, then most liked and finally latest. This was done to keep the database access to a minimum so the extra latency of running the process on the cloud is less noticeable. In the end, we republish the shorts as (tukano+shortId) with it's owner being Tukano, our managed user.

The republished shorts are the top 5 selected shorts taken from the ordered list mentioned previously. As all users when they are created already follow Tukano, after calling tukanoRecommends, getFeed() takes care of displaying these shorts by displaying all shorts of users followed by the user calling the function.

2.5 Geo replication

For the geo replication, we only enabled this feature for the CosmosDB NoSQL database, with the built-in solution in Azure Portal.

2.6 Artillery

For the results shown bellow, we chose to use the teacher's artillery tests because they were more complete than ours. All the metrics and times used are from the Realistic Flow test.

3 Results

After testing each combination of solutions (with and without cache, sql and nosql database), we arrived at the following table:

Metric/Solution	SQL		NoSQL	
	Cache	No Cache	Cache	No Cache
Min	60	62	63	63
Mean	299	656	592	267
p50	97	93	109	89
p75	596	347	714	166
p90	872	2600	1200	620
p95	963	3600	3200	1200
p99	1100	5400	4500	2100
max	3700	6100	10000	4500

Table 1: Table with time performance (in ms) for each DB solution with and without cache

3.1 Evaluation of the results

After looking into the results, we can see that the NoSQL solution performs better for no cache solutions, which leads as to the conclusion that the non-relational database performs better, likely due the

fact that usually there's more overhead in processing SQL.

This can also be attributed to the low performing 1vCore and 2GB of RAM node in the PostgreSQL cluster, versus the 400 R/Us output on the NoSQL, with the latter being much more optimized for read-heavy operations. One thing supporting this is the difference in p90 and above operations. This operations are called more often (like getFeed()), which is a read-heavy operation and NoSQL handles it much better than the PostgreSQL option. In our case our cache results are a bit misleading since one of our features (Tukano Recomends) only works when there's cache involved as explained here.

Since it runs in a cloud function when cold starts occur, it can impact overall response time making it less consistent that the no cache approach. For example, when calling directly the function in postman for the first time it leads to a 500ms overhead, later stabilizing to around 200/250 ms.

Also, our solution of caching all users and shorts may not be the most efficient according to our tests. This is because operations like searchUsers() and getShorts() are not the most common and instead getFeed() is. As our cache is not the best in size and performance (250 Mb), maybe it would have been beneficial if more of it's space was allocated only to more read intensive operations.

We tested our solution with No SQL and cache, but without the Tukano Recommendations and, as predicted, the results are much better:

Metric	NoSQL w/Cache
Min	59
Mean	118
p50	86
p75	103
p90	133
p95	194
p99	369
max	5200

Table 2: Table with time performance (in ms) for the No SQL and Cache solution without Tukano Recommendations

In conclusion, the solution using the PostgreSQL Database and caching shorts and users proved to be the best one, when using the our implementation of Tukano Recommendations.

4 Lessons Learned and Future Enhancements

4.1 Alternative implementations to Tukano Recommendations

After analyzing the results, we concluded that our current implementation of Tukano Recommends may not be optimal. The getFeed operation is highly requested, and calling the Azure function for each request results in high latency and increased costs.

An alternative approach could be to update the list of recommended shorts asynchronously whenever a user likes or downloads a short, as these actions directly impact the like and view counts.

Another possible approach would be to implement Tukano Recommends with a time-triggered function that updates the user's feed every x minutes.

4.2 Caching blobs

After delivering the project, we noticed that caching the blobs associated with the most popular shorts would be a good solution for a faster download.

5 Observations

- To test our solution with and without cache, we created a new JavaUsers and JavaShorts class but without the code relative to the cache. This modifications were not included in the submitted commit point, but are present in the latest commit of the repository.
- When testing our solution with the artillery tests, we noticed that some test were throwing a TIMEDOUT error. We suspected the problem was with the low TIMEOUT policy in the artillery tests for http requests, so we increased this timeout to 10 seconds, solving this errors.
- When running the artillery tests with the PostgreSQL solution, the operations involving the Azure functions (Increase Views and Tukano Recommends) often failed, throwing an error about the session factory being null. When testing with Postman, these errors didn't arise.
- We had problems running the basic solution at docker, so that we could compare the results with the cloud solution.