

Cloud Computing Systems
Project 2
Performance Evaluation and Analysis

João Oliveira [61052]
Guilherme Franco [60226]

December 15, 2024

Contents

1	Introduction	2
1.1	Project Overview	2
1.2	Scope and Objectives	2
2	Ported Solution Design	3
2.1	Solution Architecture	3
2.2	Design Choices and Rationale	3
2.2.1	Blob Storage	4
2.2.2	Redis Cache in Azure Kubernetes Service	4
2.2.3	Hibernate + PostgreSQL in Azure Kubernetes Service	4
2.2.4	User Authentication using Cookies	5
3	Performance Evaluation	6
3.1	Evaluation Metrics and Methodology	6
3.2	Comparative Analysis with TuKano Baseline	6
3.2.1	Azure as PaaS vs Azure as Iaas	7
4	Discussion	9
4.1	Impact of Design Choices on Performance	9
4.2	Challenges and Limitations	9
5	Conclusion	11
5.1	Summary of Findings	11
5.2	Future Work	11
5.3	Use of AI tools	11
A	Appendix	12
A.1	Code Snippets and Configurations	12

Chapter 1

Introduction

1.1 Project Overview

Our first project consisted 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. This second project strives to retain the features offered in the first assignment, by replacing the Azure PaaS services with kubernetes based alternatives. This was done using Docker and Kubernetes, as IaaS facilities provided by Azure.

1.2 Scope and Objectives

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

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

- Persistent Volumes;
- Azure Kubernetes secrets;
- Postgres in a Docker Container;
- Redis Cache in a Docker Container;
- User authentication using cookies
- Azure Kubernetes clusters;

We mostly focused on adapting the logic and services to leverage the Azure Kubernetes services.

- We adapted the Cache to leverage Docker technologies (Docker Containers) for our users and shorts.

- Our Database was changed to leverage Docker technologies (Docker Containers), namely postgresSQL. This was implemented for the Users, Shorts and Likes database.

- We also changes the Blob storage method to leverage the persistent volumes in Docker containers

- We attempted to also implement the trigger for updating views in bobs, but because of a bug connecting to the SQL database it is not working, alhtought fully implemented and tested to be running.

2.2 Design Choices and Rationale

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

2.2.1 Blob Storage

Persistent storage in Docker containers ensures data durability, consistency, and availability across container life cycles. The previous implementation was using a Azure's Blob Storage to store the Blobs. Our current implementation uses a persistent volume inside a Docker Container to store them.

How this helps:

- Persistent storage allows data to outlive the container. this lets data like user uploads, databases, or logs to remain available even after container restarts or upgrades.
- With persistent storage, multiple containers can share the same data via network-attached storage or a distributed filesystem. This allows for scaling web apps horizontally by running multiple container instances and load balancing across containers without risking data inconsistency.
- Persistent storage enables easy backups of critical application data. If something goes wrong, such as a container crash or data corruption, the data stored persistently can be restored quickly without significant downtime.

Knowing all this we changed the Blob implementation to use persistent storage instead.

2.2.2 Redis Cache in Azure Kubernetes Service

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. We switched from Azure's managed Redis Cache to a Redis cache in a Docker Container.

How this helps:

- Flexibility: We can deploy Redis in a Docker container on any platform (on-premise, hybrid cloud, or any cloud provider), avoiding vendor lock-in.
- We can place the Redis container with our application containers, reducing network latency between the application and the cache.
- Resource Optimization: Running Redis in a container allows us to optimize resource usage by collocating multiple services (e.g., Redis, application, and database) on the same machine

2.2.3 Hibernate + PostgreSQL in Azure Kubernetes Service

We changed the database of the previous implementation to leverage **Hibernate** and **postgreSQL** in Azure Kubernetes Service. It's being deployed in a Docker Container.

How this helps:

- It can run anywhere—on-premises, in the cloud, or on hybrid setups—without being tied to a specific vendor.
- As an open-source database, we benefit from an active community and a wealth of extensions and plugins

PostgreSQL: Ideal for applications that need relational data modeling and require horizontal scaling.

This database was used to store Users and Shorts.

2.2.4 User Authentication using Cookies

To enhance the security and control of access to blob storage, we implemented a cookie-based user authentication mechanism. This feature ensures that only authenticated users can upload or download blobs, while administrative privileges are required to delete blobs.

Implementation Details:

- **Admin User Role:** A dedicated administrative user was created with exclusive permissions to delete blobs. Regular users are restricted from performing delete actions, ensuring better management and security of the application's resources.
- **Authentication Requirements:** Users must authenticate via a login process to obtain a valid cookie, which is then required for subsequent operations like uploading or downloading blobs. Each cookie remains valid for one hour, after which users must reauthenticate to continue accessing the application.
- **Cookie Validation:** The server validates cookies for each incoming request. If a cookie is invalid, expired, or missing, the server rejects the request and prompts the user to log in again. Only requests with a valid cookie are processed.

Rationale for Implementation:

- **Controlled Access:** This mechanism ensures that only authenticated users can interact with blob storage, reducing the risk of unauthorized access to sensitive resources.
- **Enhanced Security:** Restricting deletion actions to the admin user minimizes accidental or malicious data deletion, improving overall resource protection.
- **Simplicity and Scalability:** A cookie-based authentication system is easy to implement and manage. It integrates seamlessly with the containerized application architecture and supports horizontal scaling without introducing significant complexity.
- **Session Management:** Cookies provide a lightweight solution for managing user sessions, ensuring that users can maintain authenticated access without reauthenticating for every request within the session validity period.

This implementation enhances the security, scalability, and usability of the application while maintaining compatibility with the existing Docker and Kubernetes setup.

Chapter 3

Performance Evaluation

3.1 Evaluation Metrics and Methodology

To assess the performance of our solution, we conducted a series of tests using Artillery. These tests measured key metrics such as throughput and latency, allowing us to evaluate the ported solution against the TuKano baseline application. While we designed some custom tests, most of the tests were sourced from the SCC GitHub repository to ensure consistency and reliability.

However, it is important to highlight a significant limitation in the testing process: the environments for the two implementations differed substantially. The TuKano baseline application was tested using Azure cloud resources, while the performance evaluation for the Kubernetes-based implementation was conducted locally using Minikube clusters. This change was necessary because our Azure accounts had run out of free credits, making it impossible to continue testing in the cloud environment.

The use of a local testing setup introduces discrepancies in the results. Unlike cloud-based testing, local testing does not account for network latency, distributed infrastructure, or the scalability challenges typically encountered in cloud deployments. Consequently, while the metrics provide valuable insights, direct comparisons between the baseline and the current implementation must be interpreted with caution.

Despite these limitations, the testing process still offers a foundational understanding of the relative performance improvements introduced by the Kubernetes-based solution. However, further cloud-based testing would be essential to fully validate the results and accurately assess the impact of our design choices under real-world conditions.

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	16
	User Delete	4.00	10
Azure Baseline	Register user	50.25	30
	Upload shorts	97.3	20
	Realistic Flow	58.45	16
	User Delete	52.9	10
previous No cache Postgres	Register user	79.75	30
	Upload shorts	110.65	20
	Realistic Flow	210.6	16
	User Delete	503.1	10
previous Cache Postgres	Register user	663.7	30
	Upload shorts	241.0	20
	Realistic Flow	2165.0	16
	User Delete	89.6	10
Cache Local Kubernetes	Register user	5.8	30
	Upload shorts	29.1	20
	Realistic Flow	9.7	16
	User Delete	17.9	10

3.2.1 Azure as PaaS vs Azure as IaaS

Table 3.2: Comparison Between previous implementation and current implementation Targets for Each Test Type

Target Type	Test	Avg Res Time (ms)	Proj1 vs Proj2 (%)
Cache Proj2	Register user	5.8	99.13%
Cache Proj1		663.7	Reference
	Upload shorts	29.1	87.91%
		241.0	Reference
	Realistic Flow	9.7	99.55%
		2165.0	Reference
	User Delete	17.9	80.02%
		89.6	Reference

- **Cache - Register User:**

With caching enabled, Proj2 completes the operation in 5.8 ms, which is 99.13% faster than Proj1 (663.7 ms). This shows a more noticeable improvement in Azure Kubernetes' response time over Proj1 when caching is used, likely due to optimized retrieval for repeated operations.

- **Cache - Upload Shorts:**

Proj2 completes this operation in 29.1 ms, which is 87.91% faster than Proj1 at 241.0 ms. Caching improves Proj1's efficiency slightly, although the difference is not as marked as in other tests.

- **Cache - Realistic Flow:**

Proj2 takes 9.7 ms to complete this workflow, which is 99.55% faster than Proj1 at 2165.0 ms. This suggests that Azure Kubernetes performs exceptionally well in complex operations with caching, significantly reducing time compared to Proj1.

- **Cache - User Delete:**

With Proj2 completing the operation in 17.9 ms, it shows an 80.02% improvement over Proj1's 89.6 ms. This demonstrates solid optimization in simple deletion tasks with caching enabled.

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 would not go over the allocated number of credits.

When using Kubernetes secrets, we discovered it automatically hashed the variables without informing us, making us lose a lot of time until we figured this out.

Due to the high amount of logs Kubernetes generates, we had difficulties figuring out what was wrong at times.

We also ran out of credits to test Kubernetes in the Azure cloud, so we had to use Minikubes locally.

Lastly, during the testing phase, we discovered that Artillery, the tool we used for performance testing, couldn't properly handle cookies. Since our implementation uses cookies for user authentication, we couldn't test that version of the application. To work around this, we ended up testing a stripped-down version of the app without authentication, which isn't ideal. It got the job done for now, but it means the performance results don't fully reflect how the app would work with the authentication system in place.

One feature we did fully implement was an HTTP trigger, which is working perfectly in terms of functionality. Unfortunately, it's not behaving as expected due to an issue connecting to PostgreSQL. Every time the trigger tries to update an item in the database, we get the following error:

```
[http-nio-8080-exec-6] function.TriggerResource.update_views
POSTGRES_URL = jdbc:postgresql://postgres:5432/postgres-scc-2425,
POSTGRES_USER = citus, POSTGRES_PASSWORD = Admin1234
```

```
Error updating item in PostgreSQL: No suitable driver found for
jdbc:postgresql://postgres:5432/postgres-scc-2425
```

```
ERROR [http-nio-8080-exec-6]
org.jboss.resteasy.core.ExceptionHandler.handleWebApplicationException
RESTEASY002010: Failed to execute
```

This issue appears to be related to a missing or misconfigured PostgreSQL driver. We spent a good amount of time trying to debug it but weren't able to resolve it before the deadline. The trigger itself is implemented and functional, so once the database connection issue is resolved, it should work as intended.

Chapter 5

Conclusion

5.1 Summary of Findings

Azure Kubernetes Service gives full control over your containerized applications, and their configuration. You can define networking, scaling, storage, and deployment strategies in detail. This means we had an unprecedented flexibility in what we wanted to use for our application, with Docker Hub providing images for several databases, caches, etc.

Azure Kubernetes Service supports hybrid and multi-cloud strategies via Kubernetes' portability. You can run the same workload on Azure, on-premises, or other clouds. All our data was also kept due to the fact that Docker provides persistent Volumes, saving us a lot of time.

In conclusion, Azure Kubernetes Service provides unparalleled flexibility, control, and scalability for complex, containerized, and multi-cloud applications. However, it requires more effort and expertise compared to Azure PaaS. The choice between Azure Kubernetes Service and Azure PaaS depends on the application's complexity and the if we plan on expanding to multi-cloud applications.

5.2 Future Work

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

- Tukano Recommends - Every user automatically follows a system managed user named "Tukano Recommends". 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 adaptation of Tukano Views for this new infrastructure.

5.3 Use of AI tools

In order to faster do our project, we used ChatGPT to help creating some parts of the yaml files, namely adding the secrets to the postgres yaml file. We also used it to help write this report.

Appendix A

Appendix

A.1 Code Snippets and Configurations

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


```
kubernets_yaml >  commands.sh
1  kubectl delete -n default deployment scc2425-webapp postgres redis blob-http-trigger
2  kubectl delete pvc postgres-pvc redis-pvc media-pvc
3  kubectl delete -n default secret db-user-secret
4  kubectl delete -n default service scc2425-webapp-service postgres redis kubernetes blob-http-trigger-service
5  kubectl delete persistentvolume media-pv postgres-pv redis-pv
6
7  kubectl create secret generic db-user-secret \
8    --from-literal=DB_USER=admin \
9    --from-literal=DB_PASSWORD=AdminPassword \
10   --from-literal=POSTGRES_USER=citus \
11   --from-literal=POSTGRES_PASSWORD=Admin1234 \
12   --from-literal=POSTGRES_URL=jdbc:postgresql://postgres:5432/postgres-scc-2425 \
13   --from-literal=SECRET_TOKEN=secret
14
15  kubectl apply -f postgres-pvc.yaml
16  kubectl apply -f postgres.yaml
17
18  kubectl apply -f redis-pvc.yaml
19  kubectl apply -f redis.yaml
20
21  kubectl apply -f blob-pvc.yaml
22  kubectl apply -f blob.yaml
23
24  kubectl wait --for=condition=ready pod -l app=postgres
25
26  kubectl wait --for=condition=ready pod -l app=redis
27
28  kubectl apply -f blob-http-trigger.yaml
29
30  kubectl wait --for=condition=ready pod -l app=blob-http-trigger
31
32  kubectl apply -f webapp.yaml
33
34  kubectl wait --for=condition=ready pod -l app=scc2425-webapp
35
36
```

Figure A.1: Kubectl commands

```
kubernetes_yaml > ! postgres.yaml
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: postgres
5    labels:
6      app: postgres
7  spec:
8    replicas: 1
9    selector:
10     matchLabels:
11       app: postgres
12    template:
13     metadata:
14       labels:
15         app: postgres
16     spec:
17       containers:
18       - name: postgres
19         image: postgres
20         ports:
21         - containerPort: 5432
22         env:
23         - name: DB_USER
24           valueFrom:
25             secretKeyRef:
26               name: db-user-secret
27               key: DB_USER
28         - name: DB_PASSWORD
29           valueFrom:
30             secretKeyRef:
31               name: db-user-secret
32               key: DB_PASSWORD
33         - name: POSTGRES_USER
34           value: "citius"
35         - name: POSTGRES_PASSWORD
36           value: "Admin1234"
37         - name: POSTGRES_DB
38           value: "postgres-scc-2425"
39         - name: POSTGRES_ADMIN_USER
40           valueFrom:
41             secretKeyRef:
42               name: db-user-secret
43               key: POSTGRES_USER
```

Figure A.2: Excerpt of our Postgres yaml file