Project Report Phase 2 - SCC

João Bernardo[62612] and João Silva[70373]

Universidade Nova de Lisboa, Portugal

Abstract. This report continues the work done on the previous stage of the project by introducing containerization of the different services of the application (databases, cache, azure functions app services, etc), turning them into micro-services and managing them with a Kubernetes cluster.

Keywords: Microsoft Azure Cloud · Kubernetes · Microservices

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 platform. 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 Containerization of services

Contrasting with the Azure approach, this allows for finer control over the technologies used to build the application. We liked this approach better, since it takes advantage of the application's architecture. Also, Kubernetes possibilities enhance the development experience, for example, in our case the Ingress resource was very useful to route the requests. We would say that the least intuitive part to do with micro-services were the Azure Functions, on that the public cloud provided services had the upper hand.

2 Implementation

2.1 Database

Previously, the database (CosmosDB or PostgreSQL) was hosted as an Azure-managed resource. In our updated architecture, we containerized the PostgreSQL solution and deployed it as a Kubernetes Deployment using an official Docker image from Docker Hub (postgres:<version>: `https://hub.docker.com/_/postgres`).

The Deployment is backed by a Persistent Volume Claim (PVC) to ensure data persistence, even if the pod restarts. The PostgreSQL container stores its data in the directory /var/lib/postgresql/data, which is mapped to the PVC.

We secured the password of the database using a Kubernetes Secret, ensuring secure storage.

2.2 Redis Cache

For our caching layer, we containerized Redis using the official Redis image (redis:latest) from Docker Hub (`https://hub.docker.com/_/redis`). The Redis instance is deployed as a Kubernetes Deployment and exposed as a ClusterIP Service on its default port 6379, allowing internal communication within the Kubernetes cluster.

The JedisPool configuration in the application was updated to point to the Redis Service's within the cluster.

2.3 Blobs service

The blobs service is implemented as a separate microservice responsible for managing blob storage. This service was containerized using a custom Docker image created with a Dockerfile and deployed to Kubernetes as a Deployment. The Deployment is exposed via a ClusterIP Service on port 8080, enabling internal communication within the cluster

To ensure persistence of the blobs, the service utilizes a Persistent Volume bound through a PVC, ensuring that blob data remains intact even if the pod restarts. This design ensures high availability and reliability of the blobs service, critical for application functionality.

2.4 Tukano App

The Tukano application, which includes the users and shorts services, is packaged as a single Docker image. This image was built using a custom Dockerfile.

The application is deployed to Kubernetes as a Deployment and exposed via a ClusterIP Service on port 8080.

2.5 Ingress

To enable external access to the application, we configured a Kubernetes Ingress resource. The Ingress routes requests to both the users/shorts service and the blobs service, allowing them to share the same base URL.

For local development, we expose the Ingress using the minikube tunnel command, which maps the Ingress Controller's external IP to 127.0.0.1.

2.6 Azure Functions

To replace the Azure functions we create two Rest resources for each function to be called by the other services when needed. These resources run on the same pod/deployment as the main tukano application (with users and shorts service)

2.7 Authentication

For the authentication, we followed the example of the practical lessons by adding the @CookieParam to the endpoints that need the cookie, being the Upload, Download, Delete and DeleteAll endpoints from the blobs service, and the DeleteUser and DeleteShort endpoint from the users and shorts services, respectively.

Some assumptions were made regarding the authentication policies, like:

- Any user authenticated can download any blob
- Only logged in short owners can upload their shorts
- When a user deletes itself or a short, the blobs associated to the delete shorts will remain in the system
- If a admin deletes a user or short, the associated blobs will also be deleted
- Admins can also upload shorts from another users

Another small feature we implemented is, when a user makes login successfully, the system redirects him to the getUser endpoint.

Additionally, to login the user, there is a new endpoint in the app at /rest/login

2.8 Project Structure

In this phase, we ended up changing considerably the project structure to accommodate the additional blobs service. So, we now have the following main files/packages:

- **blobs** - This package has the blobs logic and blobs rest endpoint definitions. It also has its own pom.xml and Dockerfile
- **common** - This is a package that contains all the common java classes that used both in the blobs and users/shorts services. This way, we don't need to have repeated code. This module is used as a dependency in the users/shorts and blobs services. This package is compiled as a .jar.
- **k8s** - This package contains all the kubernetes .yaml files to deploy the application to a cluster in the namespace **tukano**
- **users-shorts** - This package has the users and shorts logic, including it's logic and rest endpoint definitions. Similar to the blobs package, it has it's own pom.xml and Dockerfile
- **pom.xml** - A general pom.xml is used to compile all three packages (blobs, common and users-shorts) at the same time and create their respective .war and .jar files.

2.9 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.

2.10 Test bed and deployment environments

The development of the project was entirely made using minikube. After running artillery tests on the local deployment, the next step was deploy the Kubernetes resources to a Azure AKS Cluster, were we performed the same tests both locally and simulating users in different regions.

3 Results

3.1 Local deployment

The following are the results from the local deployment using minikube and local tests

Metric/Solution	Cache		No Cache
	Recommendations	No Recommendations	No recommendations
Min	3	3	3
Mean	132	15	15
p50	28	9	9
p75	116	16	19
p90	334	34	29
p95	672	46	38
p99	1500	66	87
max	1800	398	392

Table 1: Table with time performance (in ms) for the cache and no cache solution, with and without Tukano Recommends - Minikube

Disclaimer: Because our Tukano Recommendations implementation is not prepared to run solely on database, we were only able to test the no cache solution without recommendations.

The table above shows that the solution using recommendations experiences significantly higher request times. This is primarily attributed to the heavy load imposed by the Tukano Recommendations on the cache. Additionally, the feed endpoint, which frequently invokes the Tukano Recommendations function, emerges as the most requested endpoint in the test flow. This high volume of requests further contributes to the elevated request times for this endpoint, as illustrated in the images below.

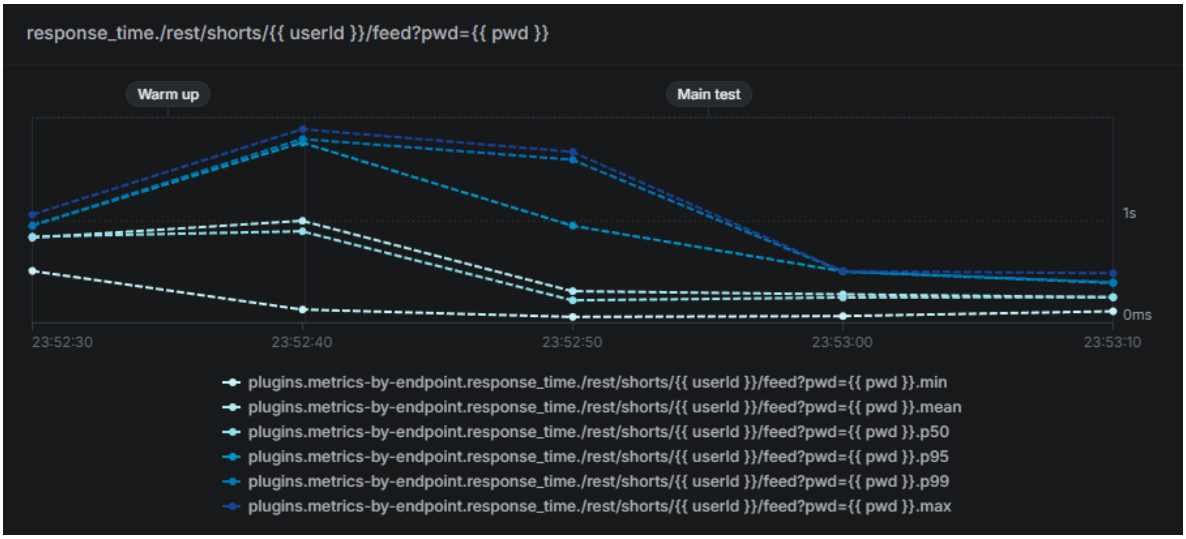

Fig. 1: Response time for the getFeed with cache

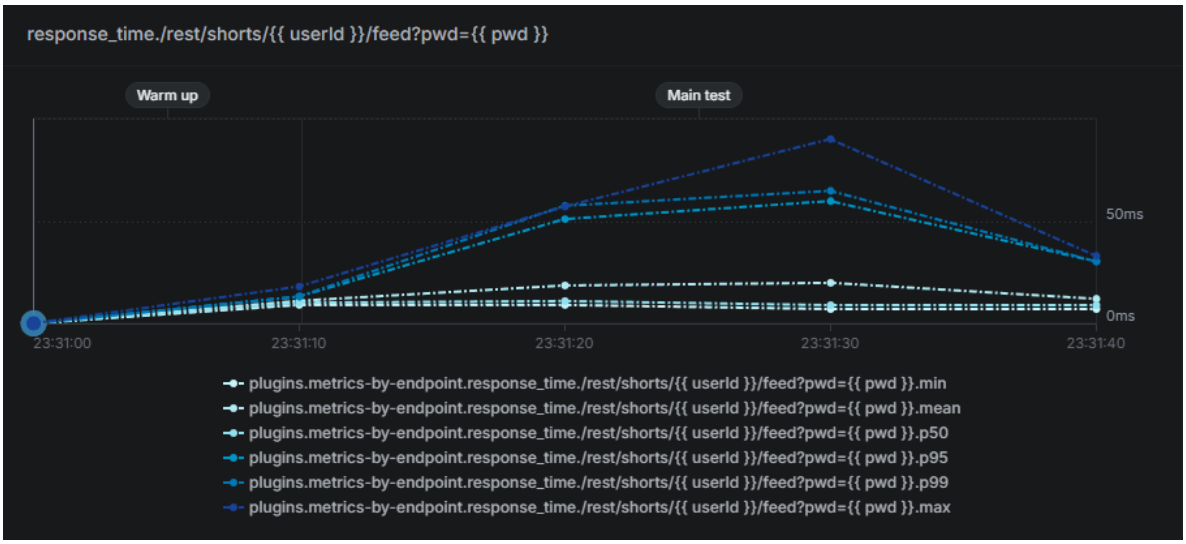

Fig. 2: Response time for the getFeed without cache

To better compare the solutions with and without caching, we ran the application with caching enabled but without Tukano Recommendations. The results were so similar that we concluded the presence or absence of caching does not significantly impact the app's performance. However, its important to note that these results are based on a local Kubernetes deployment using Minikube, which may not fully represent performance in a production environment.

3.2 Remote deployment (AKS Cluster)

Local tests - To test the application in a real environment, where it is exposed to the web, we deployed the Kubernetes services to a Azure AKS Cluster. Here are the Artillery test results.

At first, we deployed the application in North Europe and ran the artillery tests locally, and the results are as follows.

Metric/Solution	Cache		No Cache
	Recommendations	No Recommendations	No recommendations
Min	60	61	59
Mean	375	90	84
p50	84	86	76
p75	133	97	84
p90	1000	107	97
p95	2100	113	111
p99	3700	180	252
max	6100	349	384

Table 2: Table with time performance (in ms) for the cache and no cache solution, with and without Tukano Recommends - AKS Cluster

Again, the performance of the solution with the tukano recommendations feature shows that the response times are a lot higher than the solutions without it. This again is explained by the feed endpoint and it's inefficiency and high request rate.

Artillery tests from different locations - To truly test the application as in a real world scenario, where users from all over the world would access the app, we build a container for the artillery tests and deployed it to various locations in Azure, namely: northeurope, westus and eastus

The following results show the request times when running the realistic flow, with and without cache, with no recommendations:

Metric/Solution	Cache			No Cache		
	North Europe	East US	West US	North Europe	East US	West US
Min	2	66	127	2	68	127
Mean	13	75	140	10	77	148
p50	9	72	136	8	74	144
p75	14	76	141	11	78	153
p90	25	84	150	17	86	159
p95	31	89	153	22	91	166
p99	53	107	183	35	103	207
max	374	355	487	279	323	448

Table 3: Table with time performance (in ms) for the cache and no cache solution without Tukano Recommends simulating users from different locations - AKS Cluster

3.3 Evaluation of the results

At first, we saw that performance metrics with cache were much worse than with no cache. However, that is a behavior that was not expected, at least not of the magnitude. After trying to investigate the issue, we linked it to our Tukano Recommendations function. This function involves a great deal of processing power since it accesses multiple times both database and cache in both read and write cases. Since this function is called every time with getFeed(), it is hindering the performance, as seen by the graphs in the results section. Even so, with the function not working we still only see comparable performance between the cache and no cache deployments, leading to believe there's room for improvement in our cache solution. As we are working with micro-services a cache miss can prove to be detrimental to the performance since this causes a bigger network overhead having to both connecting to Redis pod, than to App Server and than to the Postgres pod.

Comparing to the first iteration where we used Azure's services where we will be using both the cache and no cache versions and SQL only (just like our micro-services approach) performance is looking much better. Both locally, as well as with AKS we are getting much lower latency. This could be related to the nature of the deployment meaning that access to network resource within the cluster (especially if located in the same node) has lower latency than the Azure's service TCP/TLS connections.

Comparing the data between regions it's very much what we expected. The North Europe region where our cluster is deployed performs the best, even better than locally. Most likely due to the optimizations the data center has compared to local deployment. Both East US and West US have slightly worse

results and that's due to the latency with East US giving the least latency of the two, since it's geographically closer to North Europe.

4 Observations

- The previously Azure Functions are now REST resources, similar to the other existing resources.
- There is a commands.txt file in the repository with all the commands we used to compile, build, deploy and test the resources

5 Run the app

5.1 Locally on Minikube

- To run the Tukano app on Minikube, we first need to start the docker engine and run `minikube start` ;
- Then, we need to compile the project, build the `users-shorts` and `blobs` docker images and push them to docker hub
- Now we deploy the Kubernetes resources, starting by the namespaces, volumes, secrets, services and deployments for Redis and Postgres
- Then, after the Postgres pod is ready, deploy the `users-shorts` and `blobs` services
- Finally, deploy the ingress service and run `minikube tunnel` to expose this service for external access
- The IP address should be 127.0.0.1 and the endpoints should be accessible in the /rest/* path (http://127.0.0.1/rest/*)

5.2 Deploy to Azure AKS

- After creating a resource group and service principal, create a AKS cluster on that resource group, using the credentials from the service principal and change the kubernetes context by running the `az aks get-credentials` command
- Deploy to the AKS cluster the resources to enable the ingress resource to be used by using the ingress-nginx resource optimized for Azure Cloud
- Execute the steps 2 to 5 from the local deployment
- Get the ingress control external ip by running `kubectl get services -namespace ingress-nginx`
- Finally, access the application with the url http://<EXTERNAL-IP>/rest/*