Transition of Tukano to a Kubernetes deployment

Guilherme Fernandes
60045
gc.fernandes@campus.fct.unl.pt

Francisco Moura
60174
fte.moura@campus.fct.unl.pt

Abstract

This project transitions the TuKano social media application into a containerized framework utilizing Azure Kubernetes Service (AKS). The primary objectives include simplifying deployment procedures, automating scalability, and enhancing application resilience. The report examines the containerization of TuKano's services and employs Kubernetes native features such as load balancing and auto-scaling. Authentication protocols are strengthened to secure and limit access to blob storage, guaranteeing that only admin users are permitted to execute delete actions. Performance assessments compare containerized and non-containerized deployments regarding resource efficiency, fault tolerance, and scalability under significant load.

1 Introduction

Azure Kubernetes Service (AKS) provides a fully managed platform that simplifies the deployment and management of containerized applications using Kubernetes, by transitioning to AKS, TuKano can create a more adaptable and modular system and be independent from the Cloud-managed services previously used, such as Azure Cache for Redis and Azure App Service. One key advantage of adopting Kubernetes is its cost-saving potential, it enables the running of multiple services on shared infrastructure and allows for dynamic resource scaling according to demand. This approach helps us optimize resource utilization and reduce expenses compared to using several standalone Azure services.

2 System Architecture

2.1 Migration of Azure Functions Functionalities

Through this process, we have transitioned the following services:

- **Azure App Service:** Instead of using the managed Azure app service, a Tomcat container is used to host the Web Server.
- **Azure Functions:** We removed the use of Azure Functions to handle the blobs functionality, instead providing its functionality with a specific blobs service in the Kubernetes cluster.
- **Azure Cache for Redis:** Previously managed through Azure Cache, caching is now handled using a Redis containerized solution. A Redis Docker image has been introduced as the new caching layer.
- **Database:** The PostgreSQL database, previously managed by Azure PostgreSQL, has been migrated to a containerized PostgreSQL instance using a Docker image.

These adjustments enhance the flexibility and control of the system while maintaining the functionality and performance standards provided by the earlier Azure services.

2.2 Token Structure and Modifications

Authentication mechanisms from the previous version were extended to the blobs service, and role-based access control (RBAC) is now integrated into the system, describing a "user" and "admin". The JSON Web Token (JWT) created when a user logs in, contains data desciribing user's role in the token payload, which dictates their access level to the Tukano services. In our model, admin users have full permissions, including the ability

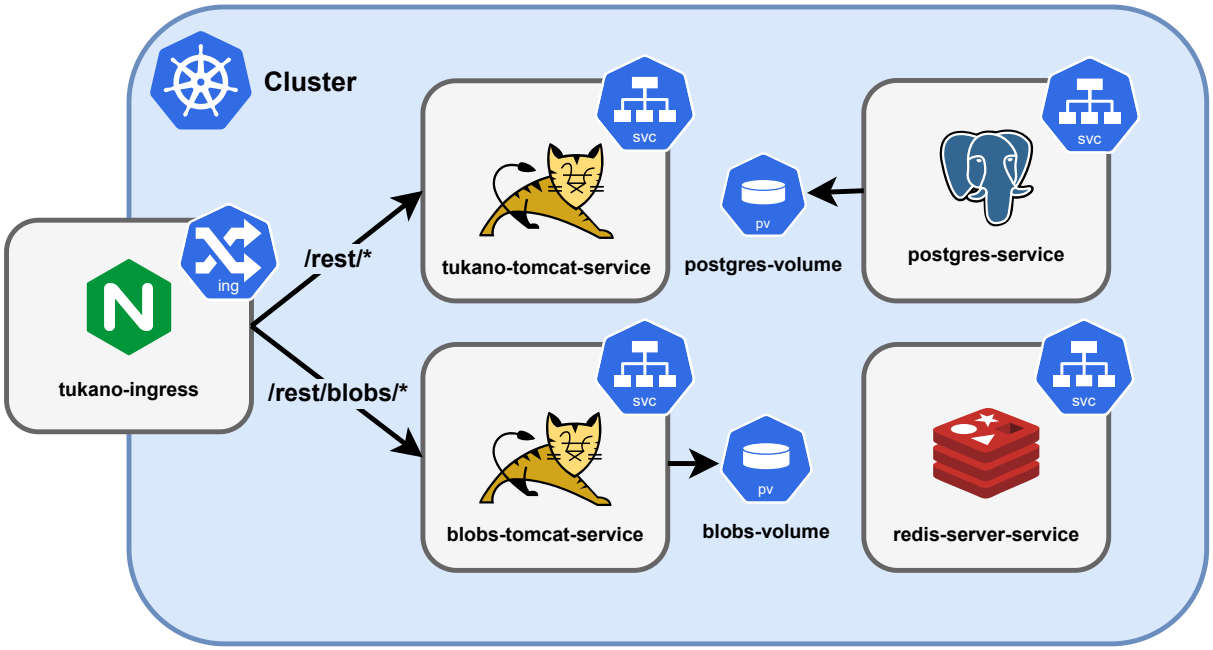

Figure 1: Tukano Kubernetes cluster architecture

to delete blobs, whereas regular users are confined to reading and writing blobs without the privilege to delete them.

2.3 Architectural Overview

2.3.1

The Kubernetes cluster of Tukano, as seen in Figure 1, includes services for the Redis Cache, Postgres DB, Tomcat Web Server for Tukano and Tomcat Web Server for Blobs, and an ingress controller for distributing the traffic to the Tukano and blobs service.

We will now go over how these services are built in the Kubernetes cluster:

PostgreSQL. This service is of `ClusterIP` type as the database only needs to be accessed from inside the cluster. The deployment associated with the service deploys a single pod, containing a `postgres:14` image. For saving the database state, a Persistent Volume is used with a given Persistent Volume Claim, mounted in `/var/lib/postgresql/data`. For setting up the database a Secret object is used, exposing the `POSTGRES_DB`, `POSTGRES_USER`, `POSTGRES_PASSWORD` environment variables for setting up the database.

Redis Cache. The Redis Cache service is deployed as a `ClusterIP` service to allow only internal communication within the cluster. It is implemented using a single pod running the official `redis-stack:latest` Docker image, unlike the database, Redis does not use a Persistent Volume, as it is primarily intended to act as an in-memory cache rather than a persistent data store. For debugging reasons an internal `ClusterIP` service also exposes a Redis Insight web instance.

Tukano and Blobs services. The Tukano and Blobs services are designed as separate components within the Kubernetes cluster to handle different responsibilities and optimize scalability based on different expected access patterns.

- **Tukano Ingress:** A NGINX Ingress object using the AKS application routing add-on is used to route external traffic to the internal endpoints of the Blobs and Tukano services. As represented in Figure 1 the ”blobs” endpoint (`/rest/blobs/*`) of the REST API will be handled by the Blobs service and other endpoints (`/rest/*`) by the Tukano service.

- **Tukano Service:** This service is responsible for handling user accounts and the ”shorts” feature of the social media application. It operates within a Tomcat container, and its network traffic is directed through the `tukano-tomcat-service`, which is configured as a `ClusterIP`. The Tukano service interacts with a PostgreSQL database

to store user information and ”shorts” metadata.

- **Blobs Service:** The Blobs service handles the management of blob storage, including operations such as uploading, downloading, and deleting media files (e.g., videos or images). Since it is anticipated that blob-related operations will receive higher traffic compared to other functionalities, the Blobs service is deployed as an independent component, ensuring that it can scale independently of the Tukano service to meet the demand for media access. Its network traffic is directed through the `tukano-tomcat-service`, which is configured as a `ClusterIP`. For saving the blobs, a Persistent Volume is used with a given Persistent Volume Claim, mounted in `/data/blobs`.

This separation enhances the system's scalability and resilience, as the two services can be independently scaled and updated, optimizing resource allocation and minimizing the risk of performance bottlenecks under high load conditions.

3 Testing

4 Specification of Resources for the tests

For the testing, the following resources specifications of Azure Kubernetes Service were used:

- VM Size: Standard_DS2_v2
- Nodes: 1
- OS: Ubuntu

4.0.1 Availability and Reliability

5 Caching vs. No Caching with Redis

Response Time median (ms)	
With Cache	**Without Cache**
67	4965

Table 1: Comparison of median requests response times with and without cache.

In this section, we compare the impact of using caching with Redis versus not using caching on the server's response times within a Kubernetes environment, as shown in Table 1, when caching is enabled, the median response time is significantly lower at 67 ms compared to 4965 ms when caching is not used.

The performance improvement comes from Redis caching frequently accessed data, which reduces the need to query the database or perform resource-intensive operations for each request. By caching data in memory, the server can quickly retrieve the information, resulting in a substantial reduction in response time, particularly under high traffic conditions.

The test scenario was divided into three distinct phases, an initial phase, where the traffic gradually rose from 1 to 5 requests per second, a peak phase, seeing a surge from 256 to 1048 requests per second, and a deceleration phase, where the load tapered off. Without caching, the server struggled with request handling during peak times, leading to increased latency and timeouts. Conversely, using a cache allowed for more efficient traffic management, maintaining much lower and steadier response times throughout the test. The operation tested was a GET request for a user's feed that follows a vast number of others, which needs to fetch and process substantial data.

In conclusion, implementing Redis caching within a Kubernetes environment provides a significant performance boost, reducing latency and improving the scalability of the application.

6 Server Load Test and Response Time Analysis

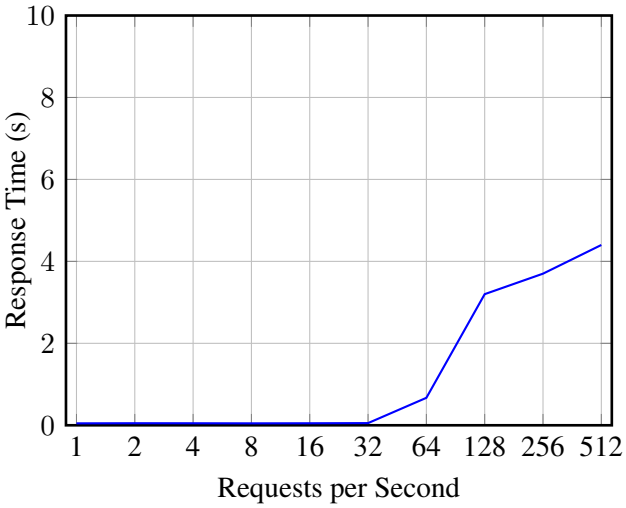

Figure 2: Requests response time (in seconds) as a function of requests per second. The chart shows how the response time increases as the request rate increases.

To evaluate the server's performance and scalability under varying traffic conditions, we con-

ducted a series of tests simulating realistic usage scenarios that reflect typical user interactions with the app. The goal was to measure the server's response time as the request rate increased, with results summarized in Figure 2, where the response times represent the median over a 20-second testing window for each traffic level. These tests were designed to mimic normal user behavior, such as fetching feeds or interacting with the app, providing insights into how the server performs under actual usage patterns.

The graph shows that at lower traffic volumes (e.g., 1 to 32 requests per second), the server maintains a steady response time below 0.1 seconds, indicating that the system can handle these loads effectively, however, as the number of requests per second exceeds 64, response times begin to rise significantly, at 128 and 256 requests per second, response times escalate to 3.2 and 3.7 seconds, respectively, with the sharpest increase occurring at 512 requests per second, where the response time reaches 4.4 seconds.

7 Global Response Time Analysis Across Regions

Region	Response Time (ms)
Australia Central	290
Brazil South	194
East Asia	210
France Central	**71**
Japan East	267
North Europe	107
South Africa North	171
West US	198

Table 2: Median response times (in milliseconds) for accessing the app from different regions.

The data in Table 2 shows that the app performs best when accessed from France Central, with a response time of 71 ms, this is expected, as the app is hosted in France Central. Regions closer to France, such as North Europe (107 ms), experience relatively low latency, while more distant regions, like Brazil South (194 ms), East Asia (210 ms), and West US (198 ms), show increased response times. The highest latencies are observed in Australia Central (290 ms) and Japan East (267 ms), reflecting the impact of geographical distance.

These higher response times can be mitigated by implementing geo-replication by possibly scaling the application across multiple geographic locations, routing user requests to the nearest replica, reducing latency. This approach would improve the performance and responsiveness for users globally,

8 Conclusion

The results of the performance tests highlight the need for a more scalable solution to handle increasing traffic volumes effectively, while Kubernetes offers a robust mechanism for horizontal scaling, enabling the dynamic addition of pods to accommodate demand, we have not implemented this feature in our current setup. Automated horizontal scaling would bring flexibility to the infrastructure, allowing for efficient scaling up during peak traffic and scaling down during low usage, optimizing resource utilization and costs. However, it is important to note that relying solely on Kubernetes for scaling sacrifices certain managed features provided by cloud platforms like Azure, such as built-in geo-replication and redundancy, these features would need to be manually configured in Kubernetes, potentially increasing operational complexity. On the other hand, Kubernetes provides significant advantages in terms of portability, as a widely adopted standard, it enables easy migration of infrastructure between cloud providers and also on-premise clusters, reducing vendor lock-in and offering greater flexibility for future growth.