Cloud Computing Systems Project 2 - Report

AUTHORS

Bruno Melo - 60019
Frederico Silva - 60247

2024-12-06

Contents

1 Introduction

The objective of this project assignment is to adapt and deploy the Tukano application developed in the first project, using Docker and Kubernetes as Infrastructure-as-a-Service (IaaS) tools provided by Azure. This solution is designed to maintain the core features of the original application while replacing the Azure Platform-as-a-Service (PaaS) offerings with Kubernetes-based alternatives.

The key challenge in this project lies in leveraging the scalability, speed, and availability offered by Docker [1] and Kubernetes within Azure [2, 3, 4], instead of relying on PaaS services. The solution takes advantage of Docker images, which were pulled from Docker Hub and customized as necessary to meet the project's requirements.

This report will explore the technical steps taken to port the original application to a containerized architecture, with a focus on integrating Docker containers and orchestrating them using Kubernetes. We will also discuss the challenges faced during the deployment process, particularly with respect to scaling, managing workloads, and ensuring high availability. Additionally, a performance evaluation will be conducted, comparing the Kubernetes-based solution with the original deployment, to identify the factors influencing the overall user experience and system efficiency.

2 Modifications Required

To adapt the Tukano project from Azure PaaS to Kubernetes, we restructured the application into microservices, including the main Tukano app, a caching service using a Redis Docker Hub image, and a PostgreSQL database service based on a Docker Hub PostgreSQL image. Azure Blob Storage was replaced with Kubernetes Persistent Volumes to handle media data. The code base was updated to integrate with these Kubernetes-managed components, and YAML configuration files were created to define deployments, services, and storage.

During the migration, we encountered compatibility issues as some SQL queries and application logic developed in phase 1 did not function correctly within the Kubernetes environment. This required modifications to both the SQL queries and parts of the application code to ensure seamless integration.

We also enhanced security by implementing authentication for blob operations via cookies. Users must authenticate by logging in through the endpoint http://serverip/tukano-1/rest/login with valid credentials. Upon successful login, they receive an authentication cookie, which must be provided to upload or download blobs. The delete operation is restricted to admin accounts, adding an additional layer of security by ensuring that only administrators can remove stored blobs. An admin account has username admin, and has to be created to test this functionality.

3 Experimental Evaluation

We used the same YAML tests from phase 1 for the performance evaluation of our solution, evaluating most operations across each distinct module. We only changed the Blob storage tests slightly as they now required the users to perform the login operation before trying to do any operation over blobs. We measured the performance of our Kubernetes implementation with and without cache and over various regions and compared these values with the ones obtained in phase 1. All the results of our experimental evaluation can be consulted on our GitHub Repository, under directory **ResultsP2** which can be accessed here.

Below, we present the results for selected methods:

Users				
Base Solution	Azure Solution	Kubernetes	Kubernetes + Redis (NORTH EU)	Kubernetes + Redis (US NORTH)
914.8	154.9	49.4	70.2	93.9
101.6	211.4	59.4	64.4	87.5
124.1	151.3	57.7	48.3	92.6

Figure 1: Mean HTTP Response Time for User requests

	Shorts					
	Base Solution	Azure Solution	Azure + Cache Solution	Kubernetes	Kubernetes + Redis (NORTH EU)	Kubernetes + Redis (US NORTH)
Post Short	377.7	565.4	1076.0	51.6	65	101.3
Get Short ID	80.7	402.9	99.3	49.8	56.4	95.9
Get Shorts	130.1	151.3	57.7	50.6	68.4	96.7
Post Follow	123.3	312.6	268.6	57.5	58.8	97.2
Get Followers	78.2	263.0	1100.4	51.9	56.7	96.0
Post Like	113.3	572.6	171.0	52.8	55.7	101.0
Get Feed	108.6	274.5	310.5	60.1	65.6	100.4

Figure 2: Mean HTTP Response Time for Short requests

	Blobs				
	Base Solution	CosmosDB North Europe	CosmosDB US North	Kubernetes North Europe	Kubernetes US North
Upload Blob	377.7	565.4	1076.0	60.9	94.2
Download Blob	80.7	402.9	99.3	61.2	95.1

Figure 3: Mean HTTP Response Time for Blob requests

4 Analysis of Performance

The data indicates that the Kubernetes-based deployment was the fastest among the other solutions tested. However, there are several factors that must be taken into account when interpreting these results.

A major consideration is that the tests in phase two were conducted on a different private network, which was considerably faster than the network used in phase one. While the network in phase one had a download speed of 50 Mbps, the phase two network ranged between 500 and 1000 Mbps. This substantial difference in network speeds likely had a significant impact on the observed latency, making it difficult to draw definitive conclusions regarding the performance of Kubernetes-based versus Azure PaaS solutions. Unfortunately, it wasnât possible to run the tests on the same network for both phases in order to get more accurate results.

Another interesting observation was the lack of clear performance benefits from using Redis. In fact, in all cases where Redis was implemented, performance was worse than when it was not used. This outcome was unexpected, and while it could be attributed to the specific testing environment, it's possible that the cacheless solution was already sufficiently fast, making the added complexity of cache management slow down processing. There are simply too many variables, especially the change in network conditions, to draw any conclusions regarding this issue.

If we assume that the Kubernetes solution is indeed faster than the Azure PaaS approach, one potential explanation could lie in the differences in architecture between the two. Kubernetes utilizes Azure container services, which operate at a lower level compared to the more abstracted Azure PaaS offerings. While Azure PaaS services are designed to simplify deployment and management, this abstraction can introduce overhead, potentially slowing down performance. In contrast, Kubernetes allows for more direct and efficient resource

We successfully deployed the test system in Azure Container Instances [5, 6] and collected results from clients located in different data centers, specifically North Europe and North US, with the Tukano application hosted in the North Europe server. The results aligned with our expectations, as the average response times from clients in North US were approximately 33% higher than those from clients in North Europe. Although this difference is modest, considering the already low mean response times across all tests, it remains consistent throughout all requests. This consistent pattern makes it a reliable indicator of the impact that geolocation has on service and deployment performance.

To provide more insight into how we achieved this final step, we created a Docker image based on an existing Node.js Docker Hub image. We installed Artillery on the container and copied our YAML scripts into it. Afterward, we created a new resource group in North US and deployed a container within it. By accessing the containerâs shell, we were able to execute the YAML tests, effectively simulating a client located in North US. The Dockerfile and commands used for this setup are available in the artillery-docker file in our GitHub repository here.

5 Conclusion

In this project, we investigated the impact and performance of Kubernetes-based microservices for deploying and managing web applications. Our findings suggest that transitioning existing applications to this model is relatively straightforward, especially if the application is modular in design. Additionally, this approach appears to deliver significant performance improvements compared to using cloud PaaS solutions, as Kubernetes allows the application to operate closer to the hardware, leveraging lower-level system efficiencies.

Kubernetes also offers remarkable flexibility, particularly in fault tolerance and replication. It can seamlessly recover from failures by replacing crashed pods automatically, ensuring continuous availabilityâan essential feature for mission-critical services that cannot afford downtime. When combined with Azure services, Kubernetes unlocks a wide range of possibilities, offering a powerful blend of flexibility and scalability for modern cloud-native applications.

We would also like to acknowledge the use of AI tools, specifically ChatGPT, which assisted with code refactoring and troubleshooting during the project.

In conclusion, Kubernetes-based microservices not only enhance application performance and resilience but also open up new opportunities for scalable and reliable deployments in diverse cloud environments.

References

[1] I. Docker, "Dockerfile reference." https://docs.docker.com/reference/dockerfile/, 2024. Accessed: 2024-12-07.

[2] Microsoft, "Azure kubernetes service (aks) documentation." https://learn.microsoft.com/en-us/azure/aks/, 2024. Accessed: 2024-12-07.

[3] Microsoft, "Quick kubernetes deployment using azure cli." https://learn.microsoft.com/en-us/azure/aks/learn/quick-kubernetes-deploy-cli, 2024. Accessed: 2024-12-07.

[4] T. K. Authors, "Kubernetes documentation home." https://kubernetes.io/docs/home/, 2024. Accessed: 2024-12-07.

[5] Microsoft, "Azure container instances documentation." https://learn.microsoft.com/en-us/azure/container-instances/, 2024. Accessed: 2024-12-07.

[6] Microsoft, "Azure container instances quickstart." https://learn.microsoft.com/en-us/azure/container-instances/container-instances-quickstart, 2024. Accessed: 2024-12-07.