Tukano meets IaaS: from PaaS to a Kubernetes-based Cloud Solution

Ricardo Bessa
r.bessa@campus.fct.unl.pt
MIEI, DI, FCT, UNL

Neel Badracim
n.badracim@campus.fct.unl.pt
MIEI, DI, FCT, UNL

ABSTRACT

TuKano is a social network inspired in existing video sharing services, such as TikTok or YouTube Shorts. In the initial phase of the project, a cloud-based version of Tukano was developed using Azure PaaS services, in order to increase the quality of service in a real-world deployment with a global user base.

Now, by using Azure IaaS facilities, we aim to transition to a Kubernetes-based architecture, building a containerized version of TuKano that takes advantage of customized Docker images built on top of public solutions offered by Docker Hub.

Index Terms - Microsoft Azure, Kubernetes, Docker, Tukano, Cloud, IaaS

1 INTRODUCTION

The original local version of TuKano was very limited in terms of scalability and availability, making it not suitable for a global use scenario. This way, for the first phase, we transformed the original centralized solution into a geo-replicated one that supports a scenario of a web application of high popularity at a regional (continental) scale. The proposed solution also supported the scenario of an application with a global scale that targets clients spanning across multiple continents (Europe and North America).

In this phase, we successfully replaced Azure PaaS services with Kubernetes-based alternatives by building custom Docker images, extending those available on Docker Hub. This transition allowed us to enhance flexibility and functionality in Tukano, using Docker and Kubernetes, as IaaS facilities provided by Azure.

In the following sections, we propose the general architecture of the Kubernetes-based version of Tukano, and then we describe our design choices about the different components of the solution. We present an experimental evaluation by showing the impact of the design choices and services on performance metrics such as throughput and operation latency and matching the new performance results against those obtained in the first assignment. Finally, we discuss those results to compare PaaS and IaaS solutions and present our final conclusions.

2 GENERAL ARCHITECTURE

TuKano was built with a Three-Tier Architecture, including the **Client Tier**, the **Application Tier** and the **Data Tier**. The last one also including a Caching Layer to improve the data access performance.

The initial version was implemented leveraging PaaS offerings in Azure. For the Application Tier, Azure App service was used in order to handle the infrastructure management and scalability, while for the Data Tier, Azure CosmosDB was the chosen service to store users' and shorts' data. This included two different types, such as NoSQL and PostgreSQL that allowed us to choose between a relational and a non-relational database. Additionally, we made use of Azure Redis Cache in the Caching Layer to refine data retrieval speed and reduce load on the database. Furthermore, the Azure Storage Account and its containers allowed us to store blobs posted by the users, while the Azure Functions were used to periodically update the views of the shorts in the DB and select the most popular ones to be republished by TuKano Recommends. We also used Azure Functions to implement a serverless version of the Blobs Service with HTTP triggers.

For the second assignment, the previously developed version of TuKano was adapted and consequently deployed using IaaS facilities provided by Azure.

Before deploying to Azure with the use of Kubernetes, we initially developed and tested the application locally using a multi-dockerized architecture, where each service operated as a Docker container. To allow the different containers to communicate with one another, we made use of a Docker Bridge Network, which enabled seamless communication from the different components of the application.

For this setup, we sourced base images from Docker Hub, selecting those that aligned closely with our requirements. These images were then extended to include custom configurations, ensuring they met the needs of our application.

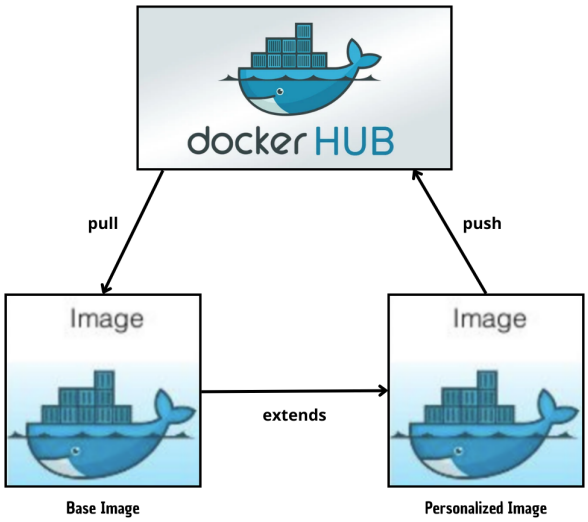

Figure 1: Creating Personalized Docker Images

Once the customization was complete, the images were pushed to our container registry for deployment. This process is represented in Figure 1.

To make this happen, we had different services such as:

- **WebApp Service** can be used by clients to access services defined on the TuKano REST API, sending HTTP requests to the corresponding REST endpoints.
- **Blobs Service** can be used for managing the media blobs that represent the actual videos. TuKano users can upload short videos to be viewed (and liked) by other users of the platform. In this version, it was implemented as a micro service with cookies for access control and a persistent volume to achieve data persistency.
- **DB Service** can be used for storing users and shorts' data. Provides a PostgreSQL relational database.
- **Cache Service** can be used to optimize the application performance by storing data in a high speed storage layer. Allows do reduce the load in the database and improves data access performance
- **Reverse Proxy Service** can be used to improve client access to the TuKano micro services
- **CronJob Service** can be used to periodically get the popular shorts posted by users.

After carefully evaluating and selecting the most appropriate Docker images for each container from Docker Hub, we extended these images with our custom configurations to meet the specific needs of our application.

In this phase, our solution retained the features implemented in the first assignment, but transitioned from Azure's PaaS facilities to a Kubernetes-based architecture leveraging Azure Kubernetes Service. Client requests, whether originating from the US or Europe, are routed through the Reverse Proxy, which is the only pod in the AKS cluster with an externally exposed IP. This proxy serves as the gateway, having its IP as LoadBalancer, forwarding requests to the appropriate service within the cluster.

Each microservice runs inside its own container, which is encapsulated within a Kubernetes Pod. As illustrated in Figure 2, the WebApp and Blobs services interact with the Cache to enable faster data retrieval. If the required data is not available in the cache, they fall back on the Database to fetch it. While the cache is used by the webapp to store popular items and the results of complex queries such as getting the feed of an user, for the blobs service cache is used to store user sessions used in the authentication process.

Additionally, to support TuKano Recommends, a CronJob was implemented. This job runs every minute to determine the most popular short and publishes it to users.

To enhance security, we stored our configuration variables as Kubernetes environment variables within the YAML deployment files, treating sensitive information such as database credentials, cache keys, and the shared secret as Kubernetes Secrets. This prevents data from being exposed when hard-coded and makes management easier, since in case of changes we only need to update the configuration and it's not necessary to modify and redeploy the application.

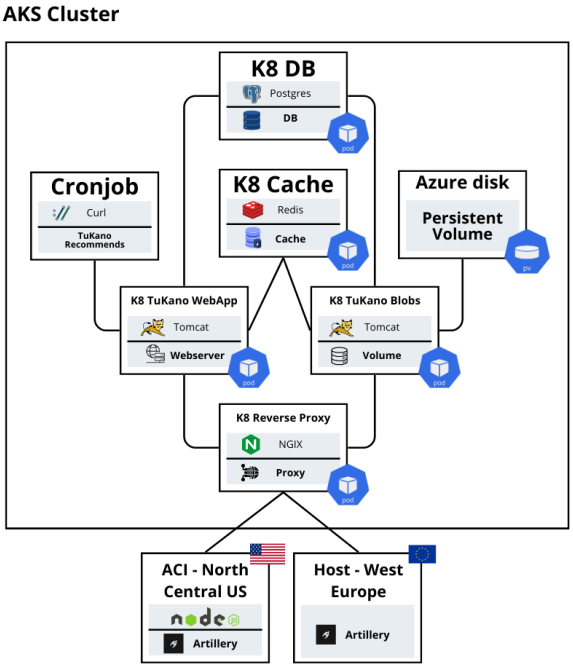

Figure 2: General Architecture

3 WEB APPLICATION

For the first assignment, we developed a modular solution that included different implementations of the several services offered by TuKano. It allowed to easily swap between different implementations by just activating the right flag.

This time, the TuKano Web Application was deployed in Kubernetes by extending tomcat:10.0-jdk17-openjdk image with our application war file. Once the Docker image was built, it was pushed to Docker Hub for use in the Kubernetes environment.

In Kubernetes, the container runs the web application image, exposing it on port 8080. Environment variables like POSTGRES_URL, REDIS_HOSTNAME, and others are injected to configure the application dynamically, ensuring secure access to external dependencies.

The Blob Service, was implemented as a micro service that allows an authenticated user to upload and download blobs, and the new admin user to delete them. To keep modularity, the previous version can still be used after selecting the right flags.

For the Data Tier, we transitioned to a Kubernetes solution that extended images from the Docker Hub. However, the Hibernate as well as CosmosDB NoSQL and PostgreSQL of Azure are still present and can still be used.

This time, the advanced features of TuKano, such as counting views and TuKano Recommends, were implemented with a CronJob present in the Kubernetes YAML. More details about these components shall be provided in the following sections.

4 BLOB STORAGE

In the previous version, our application made use of Azure Blob Storage to store videos as raw bytes in a storage account, replicated both in West Europe and North Central US. Additionally, it was still possible to use the original solution and save the blobs in the local file system by changing flags in the code. The Blob Service was also available as a serverless version by leveraging Azure Functions that were activated by HTTP triggers.

However, in this phase, this was changed to a microservice that extended a tomcat image - tomcat:10.0-jdk17-openjdk - and was deployed to the port 8080. This update was implemented due to the high user load, allowing the service to scale independently and improve performance under greater demand.

Contrary to what happened in the first version, this time interaction with the Blobs required user authentication before any request could be made. Upon signing in, a cookie was generated and the session data is stored in the cache, enabling the client to upload and download shorts. The cookie represents an authenticated user session that can be shared across all application server instances via a RedisCache. However, for this phase, deleting blobs required an admin, meaning only him could perform this action. This presented a challenge when deleting a user, as it was necessary for the admin to be logged in at the same time. To address this, a secret was shared between TuKano and the Blobs service. When a user is deleted, TuKano sends a request to the Blobs service to remove all their blobs, authenticating itself using the shared secret. This ensures that the Blobs service can verify the request as originating from TuKano rather than a malicious actor, maintaining the system's security and integrity.

Thread local storage can serve as global variables but with a scope private to a particular thread. In this case, we can use it to store the request cookies, which can then be accessed anywhere in the server code by the thread that is handling the request. Different requests will be handled by different threads, which will have their own private thread local storage. The advantage is that the REST resource API does not have to be changed with annotations to capture the cookies.

By reusing the original filesytem logic, we were able to write blobs as files inside the /mnt/vol directory of the blobs container. A Persistent Volume was used to ensure durability of these blobs. Unlike in-memory storage, the Persistent Volume keeps data intact even when the pod is restarted, redeployed, or crashes. This guarantees that the Blobs service can recover quickly in case of accidental deletion or data corruption. In TuKano's case, Azure Disk was used to protect against data loss in case of hardware failure.

For this service, some resource limits were established. The CPU limit caps usage at 1 core, allowing the container to scale and handle concurrent file operations during spikes in demand. This limit ensures adequate performance without monopolizing CPU resources in the cluster. The memory limit ensures that the container can handle increased load, such as multiple concurrent file uploads or downloads, while preventing it from consuming excessive memory that could impact other services. This value provides enough buffer to handle temporary memory spikes without crashing.

5 DATABASE STORAGE

In the previous version of TuKano, we extended the original database solution, which used Hibernate with two distinct database systems: CosmosDB NoSQL and PostgreSQL. These databases were used to store both user and short data. CosmosDB NoSQL provided a schema-less, highly scalable solution suitable for handling unstructured or semi-structured data, while PostgreSQL, as a relational database, offered a strong schema and support for structured data with complex relationships.

In the current phase, we transitioned to using PostgreSQL within the Kubernetes ecosystem. To achieve this, we extended the official postgres:17 Docker image in our custom Dockerfile. The extension involved injecting the necessary database credentials as environment variables and providing our database schema for initialization. Once the image was ready with our requirements, it was pushed to Docker Hub, making it available for deployment in Kubernetes.

A new flag was introduced, to allow the choice between this version or the previous one. The only modification in the application code involved checking the state of this flag. When the flag is activated, the system recognizes that the database is running as a Dockerized version, allowing it to adjust its connection logic accordingly.

The container runs our custom image, with resource limits and requests configured to optimize performance. A ClusterIP Service is also defined, exposing Postgres on port 5432 within the cluster, facilitating internal communication between pods. Additionally, in our Kubernetes deployment, we passed the PostgreSQL URL as an environment variable secret to the application containers. This URL serves as the connection string, allowing the services to communicate with the database. By injecting it as an environment variable, we ensure that the database connection details, such as the user and the password, are abstracted from the application code, improving security and flexibility.

The resource limits for the PostgreSQL deployment were carefully chosen to balance performance, and cost efficiency. The requests values specify the minimum resources that Kubernetes guarantees for the container, ensuring it has sufficient CPU (250m) and memory (512Mi) to handle baseline workloads without risk of being starved during periods of high cluster utilization. On the other hand, the limits values set an upper boundary for resource consumption, capping CPU usage at 1000m and memory at 1Gi.

6 REDIS CACHE

In the previous solution, to avoid too much access to the Database Layer, we deployed a Caching Layer. This happens because each access to the database costs money and resources. This way, to improve the performance of read operations we stored the popular items in the Azure Redis Cache.

In the IaaS version of TuKano, we followed a similar approach, without changing the application logic and applying

minor changes to the code. To make the solution modular, we introduce a new flag to point out that we are replacing the Azure Redis resource by a Dockerized version of the cache. The only change in the application code was made in the initialization method that defines how the app should connect to the caching layer. For this Dockerized version, we use port 6380 instead of the default Azure Redis port 6379. The hostname of the cache that should be used by TuKano to connect to Redis is now updated to the name of the deployed container with the caching service and it is provided to the webapp through an environment variable defined in the Kubernetes configuration script. We also pass a special flag to the webapp as an environment variable to inform TuKano if the cache should be used or not. This way, we can modify the behavior of TuKano on the fly by defining in the configurations file if this variable should have the value true for an active cache or false to ignore the caching layer. This is useful to make TuKano more flexible and change the behavior of the system in a fast and simple way, allowing us to quickly commute between different versions of the implementation and perform the load tests on those different versions. One of the advantages of using Kubernetes is that we can perform this update on the fly by just updating the configuration file, which will detect the updates and apply them to the running system. By passing this configuration information to TuKano using environment variables in the configuration file, we avoid to shutdown and redeploy the system just to configure this flag, avoiding to loose the levels of availability and reliability that the clients are expecting. We also use an environment variable to provide to TuKano the password to access the cache. Since this is a sensitive information, we define that password in the configuration file as a base64-encoded secret.

We searched for the most suitable Docker image in the Docker Hub to replace the Azure Redis cache by a containerized solution. We concluded that the official Redis image was not the most suitable one since it was more complex in terms of resources and more useful to create a database. For deploying the cache, we choose the image Redis-Alpine which is more simple and lightweight than the base Docker image of Redis, making it more suitable to provide a caching service where we want to maximize performance and to have more limited resources to offer a way to store just the popular data with an ephemeral approach.

To extend this image and make it more personalized to our application needs, we provide a configurations file redis.conf to the container. In this file, we define that the cache should allow connections from any IP address in the port 6379 and using the specified password for authentication. We disable all the persistence options since the cache will manipulate ephemeral data and we want to avoid the overhead of storing persistent data, which is not compatible with the performance levels that the cache should have. We also set a max memory limit of 512mb and define the eviction policy as LRU, following the specification of the original Azure Redis cache. Finally, to provide the maximum simplicity and performance to the caching layer we create only one database to act as a cache in Redis and we disable the cluster mode.

In Kubernetes configuration file, we create a Pod to wrap the cache container, with limited resources and a ClusterIP service to allow internal communication of the cache with the other pods in the cluster.

7 CRONJOB

CronJob is a new service that was introduced in the second phase. Its goal is to provide a Kubernetes-based implementation of the advanced features count views and TuKano recommends, replacing the Azure functions that were used to achieve the same functionalities in the previous version.

Every time a short is downloaded, its view count is updated in the cache rather than directly in the database. This reduces the number of database writes while still refreshing the views fast enough. During each one-minute interval, views are aggregated in the cache. When the CronJob triggers the **getPopular** function, it retrieves the accumulated view counts, clears the relevant cache entries, and writes the consolidated updates back to the database. Then TuKano recommends selects and republishes the most viewed short from the shorts created in the last 5 minutes.

TuKano provides a special endpoint to execute the get popular operation. To execute the CronJob, we extended a Docker Curl image that will make periodic requests to TuKano to execute the get popular operation. In the Kubernetes YAML, the CronJob is configured to run every minute, as defined by the **schedule: "* * * * *"** field. This schedule follows the standard cron syntax, where each asterisk represents a time unit (minute, hour, day of the month, month, day of the week). This CronJob container follows a **onFailure** restart policy that ensure that if the job fails, Kubernetes will attempt to restart it.

8 REVERSE PROXY

In the TuKano system, the reverse proxy plays a critical role in managing and optimizing network traffic. It acts as an intermediary between external clients, such as browsers or mobile apps, and the internal services running in the Kubernetes cluster.

The reverse proxy is responsible for directing incoming HTTP requests to the appropriate backend service. We introduced this service as an improvement to the first version where webapp and serverless blobs provided different URLs to the client. In that previous scenario clients would manage their access to the services by choosing to use the URL of the intended service, which complicates the client access to the system. Following a more realistic strategy, we deployed an NGIX container that was extended with the appropriate configuration to route the clients requests either to the webapp micro service or the blobs micro service. This way, we provide an unique entry point to the system, making the client access to the application more simple and straightforward.

9 EXPERIMENTAL EVALUATION

For the experimental evaluation, we followed a similar approach to the first version. We prepared six Artillery scripts with different operations. We have **users.yaml** to create users, reading from a CSV file with up to 1000 different users. We have **blobs.yaml** to, for each user, create a short, upload a blob, download a blob and delete all blobs. The blob contains just a couple of random bytes for testing purposes. We have **db.yaml** that executes a mix of write and read operations in the database for each user, including create shorts, follow other random user, like a short, get the user shorts, get the user feed, get the user likes and delete all shorts of the user. We have also **delete.yaml** that deletes all the users from the database. Finally we have **writes.yaml** and **reads.yaml** that execute only write or read operations from the database, including the ones mentioned above.

Total Time (s)	Rate (RPS)	Created Users
30	5	150
30	15	450
30	33	990

Table 1: General statistics of the experiments

We executed our own test scripts only. From the Artillery logs we extracted the mean response time as the general latency of the test. We execute the same tests for three different rates. We tested with arrival rate 5, 15 and 33. The increase of the rate means there are more concurrent requests per second. All tests have the duration of 30 seconds. The tests were executed for the running application deployed in Kubernetes. Due to the free student tier, we specified limited resources in the Kubernetes configuration file since the cluster resources are expensive. However, the provided resources configurations are enough to extract some interesting metrics about TuKano. The resource specification of the pods is presented in the following tables.

Service	Min RAM	Max RAM
Tukano Webapp	200 Mi	512 Mi
Redis Cache	512 Mi	1 Gi
Postgres	512 Mi	1 Gi
Blobs	200 Mi	512 Mi
Proxy	200 Mi	512 Mi

Table 2: Min and Max RAM for Tukano Services

Service	Min CPU (m)	Max CPU (m)
Tukano Webapp	1	1000
Redis Cache	250	
Postgres	250	
Blobs	1	
Proxy	1	

Table 3: Min and Max CPU for Tukano Services

For **Blob Storage** we previously tested the different implementations, including the original one with filesystem, the serverless blobs and the Azure blobs in Storage Accounts.

In this version we also provide the same load tests to the Kubernetes-based version, where persistent blobs were written as files in the /mnt/vol directory of the container. To test the durability of the data, we manually remove the blobs pod and then restart the pod. We used exec to inspect the /mnt/vol directory inside the container and verified that the blobs were still there, proving that the desired durability was achieved due to the Azure persistent volume. As for the load tests, we present the results in the following chart.

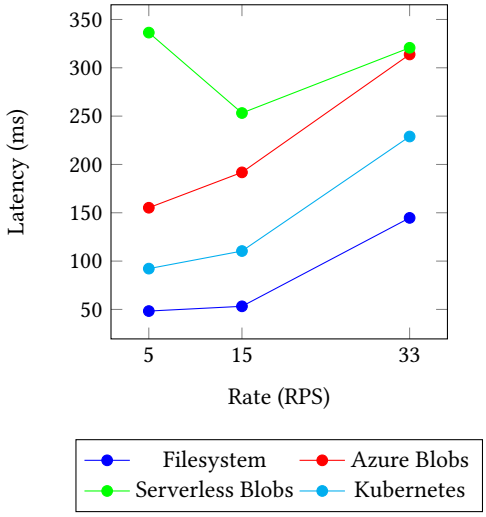

Figure 3: Latency vs Rate for Blobs implementations

As we observed before, the filesystem continues to be the fastest solution. However it is offers very limited scalability in a real deployment scenario. Serverless blobs presented a somehow slow and hard to predict performance, even though they are following a modern trend of serverless computing that allocates resources only when are needed and in the right amount. The Azure Blobs provided good levels of performance and more predictable results, being also an option that offers good scalability.

As for the new Kubernetes based solution, as we can perceive from the chart, it offers better performance than the Azure Blobs and the Serverless Blobs. This happens because this solution uses a very similar approach to the filesystem one, where blobs are written as files in a directory of the container. The read and write operations over files in the container are very fast and efficient, only being outperformed by the original filesystem implementation that created the blobs directly in the filesystem of the Azure machine running the application. The Azure Blobs version offered durability by providing the redundancy level GRS-RA and replicating blobs to North Europe to prevent the data to be lost if a region failure occurs. The Kubernetes version also offers durability by storing the blobs in an Azure disk, making sure they survive beyond the blobs pod lifetime.

The specification of the persistence volume is included in the following table.

Type	azure-managed-disk
Access Modes	ReadWriteOnce
Storage Class	azurefile
Requested Storage	1Gi

Table 4: Persistent Volume Details for Blobs

For the Blob Service we also decided to present load tests when clients are accessing the service through the provided NGIX proxy or through a direct load balancer that exposes externally the same service. The goal of this experiment was to evaluate the performance impact of the proxy component while clients are accessing a micro service within TuKano.

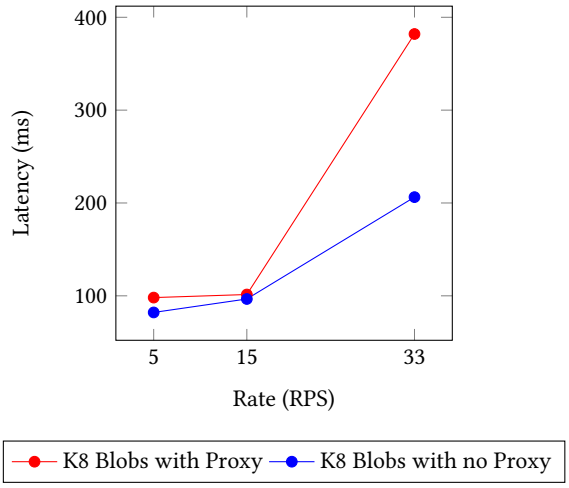

Figure 4: Latency vs Rate for K8 Blobs with and without Proxy)

From the chart we can perceive that for lower rates, the two alternatives have a similar performance. However, for a higher number of concurrent requests the proxy alternative does not scale well. This might happen because this alternative includes an extra hop to get to the micro service because clients need to contact first the proxy and only then their request is redirected to the right micro service. Without a proxy, clients can contact directly the micro service and there is no need for an extra hop. Besides that we also believe that the NGIX proxy would need extra configuration to handle heavier concurrent loads with more efficiency. Given that, both alternatives can be useful and valid options. While a proxy is a more standard solution, providing to client an unique and simple access point to the TuKano micro services, the direct contact alternative avoids the overhead of redirecting requests through the proxy. In the end, we believe that the proxy solution would be the most suitable one and with some concurrency improvements would also provide a good quality of service and an easier access to clients. This alternative could also be applied to the first version of TuKano, creating an improved access point to both webapp and the serverless blobs implemented as a separated micro service with Azure functions.

We deployed the Blobs Service as a separated micro service apart from the TuKano webapp because we predict that this service will be exposed to higher loads due to the popularity of operations like uploading and downloading blobs. This way, we benefit from the micro services architecture by handling both as independent components of the system that are able to scale differently. For the TuKano webapp, we provided a single replica in the Kubernetes configuration file. As for the blobs micro services, we conducted experiments on both versions with a single replica and a set of 3 replicas. The goal of this experiment was to evaluate the impact of scaling up a micro service that we believe that would be under heavy loads in a real scenario. We want to understand how the system behaves under these high demand periods and what it's the impact of providing extra replicas to deal with this demand. For this test we have users contacting directly the Blobs service using a Load Balancer that externally expose this service.

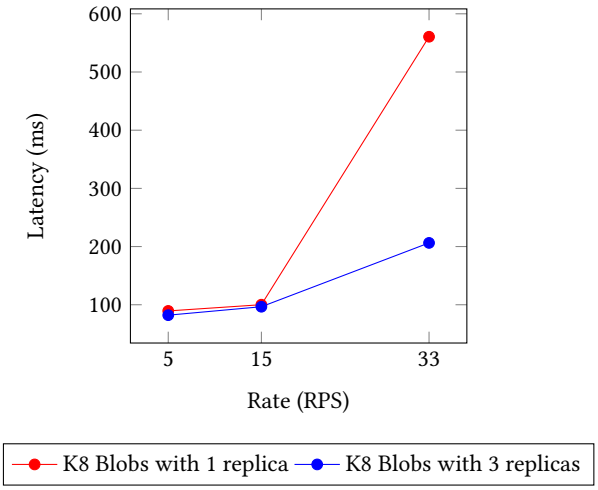

Figure 5: Latency vs Rate for K8 Blobs with 1 replica and 3 replicas

From the chart we can perceive that the Blobs micro service is more scalable when we provided a set of 3 replicas instead of a single replica. This reflects the behavior of the load balancer that handles well the concurrent requests, distributing them across the different replicas. If we have a single replica, the high demand periods will flood that replica with concurrent requests, causing the significant increase on the latency observed by users, as we can perceive from the experiment with higher rate. In this situation, the service will become really slow, compromising the quality of service and the availability of the system. At the certain point, the latency would continue to increase while the throughput of the system would decrease due to the overloaded resources. To deal with these challenges imposed during high demand periods, the choice of scaling up the service with more replicas appears to be the best solution.

For testing the different **database** solutions, we used the script db.yaml to test a mix of write and read operations. After the previous collected results, we now evaluate the performance of the Dockerized version of PostgreSQL as part of of the new Kubernetes-based version.

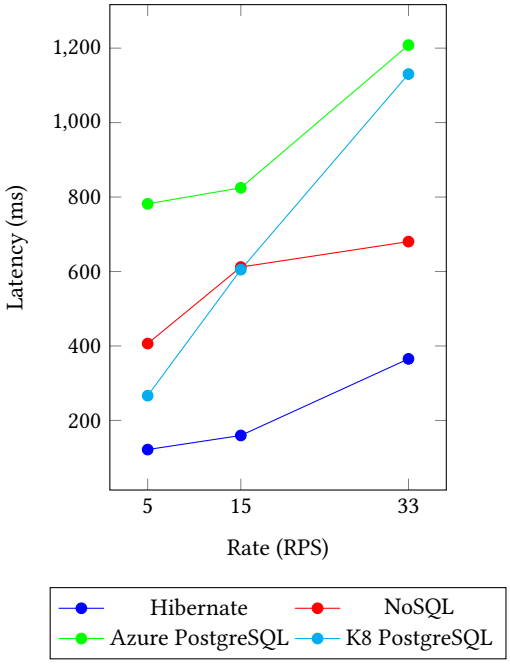

Figure 6: Latency vs Rate for different database implementations

As it can be observed, the Hibernate solution continues to be the fastest one because the database is in-memory. But this solution scalability is limited by the machine resources. Previously the best option was CosmosDB NoSQL because the data objects are treated as items stored in containers, following a very light logic and no strict schema. The Azure PostgreSQL solution is slower because it forces a strong relational schema, which introduces some additional overhead.

However, the Kubernetes version presents a PostgreSQL database with interesting results. From the chart we can perceive that for the lower rate of requests, the Dockerized PosgreSQL offers even a better latency than NoSQL. For the medium rate experiment, we perceive that the new version also matches the performance offered by NoSQL. In the higher rate experiment, the new version had a significant latency increase, showing some scalability limitations when it is under heavier concurrent loads. In this situation, the NoSQL is back to be the best option. Other interesting takeaway from this experiment is that the Kubernetes version of PostgreSQL outperforms the original Azure version of PostgreSQL with a significant margin except for the higher rate test where the performances are more similar. This shows that the new version is able to offer a competitive alternative for the database component.

However, in the first version we identified some tradeoffs that we should consider in the database component. For example, complex queries that need to combine different information, like fetching a user's feed that includes their own shorts and the shorts of users they follow, are challenging in NoSQL databases. Deleting a user also implies deleting their shorts, which requires multiple queries in NoSQL. In contrast, relational databases like PostgreSQL execute these operations with a more simple and efficient approach by leveraging primary and foreign key relationships. The following chart shows the latency for the delete user operation, including the new version with PosgreSQL in Kubernetes.

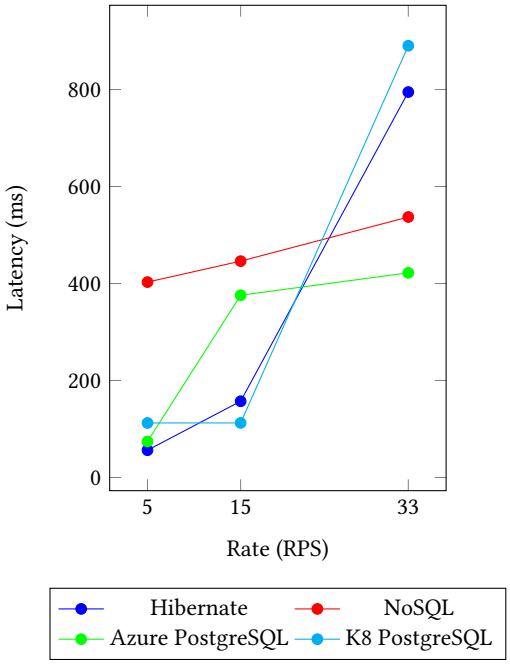

Figure 7: Latency vs Rate in Delete User Operation

As we can perceive from the chart, for the delete user operation the Azure PostgreSQL solution is faster than noSQL solution as expected. Same applies to other complex queries such as the get feed operation. The new version with Dockerized PosgreSQL has a performance hard to predict. For the first tests it presented better or comparable results to Azure PostgreSQL. However, for the higher rate the scalability appears to be compromised since we registered a very significant increase in the operation latency. During our experiments we also noticed that the Kubernetes version may take some additional time to reflect the deletion of the users, presenting some limitations when it is under heavier loads.

Previously we evaluated the performance of read and write operations in CosmosDB noSQL with and without a Redis Cache. The purpose of this experiment was to evaluate the impact on the performance while using a Redis Cache and what kind of operations are affected. The experiments were all conducted under rate of 5 RPS and tests of 30 seconds in total.

For the new version we executed the same experiments for the Dockerized Redis cache. AS expected, we could perceive the same behavior as in noSQL, where with cache read operations improved their performance and write operations increased their latency. To better compare the results of both versions, we provide the following chart.

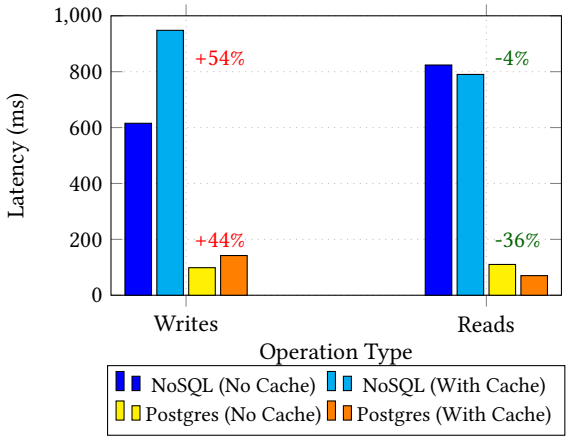

Figure 8: Latency Comparison for Reads and Writes with and without Cache

The previous conclusion was that the read operations performance was increased by the use of the Redis Cache. However the write operations were much slower with the cache because of the overhead of writing both in cache and in database. Our assumption is that in a social network the read operations will be more frequent than the write operations and, in this scenario, using a Redis Cache could be a good option to increase the quality of service of the application.

By comparing the previous version with the Kubernetes-based solution we also learn that the increase on the latency of write operations is smaller in the Dockerized Redis Cache (+44%) than the noSQL alternative (+54%), while the decrease of latency in read operations is higher in Dockerized Redis Cache (-36%) than the noSQL version (-4%). From these results we can conclude that in the Kubernetes version the positive impact of using a Redis Cache is more significant and the negative side effect in the latency of write operations is less expressive than the PaaS version of TuKano. This shows that the Redis Cache in Kubernetes is a very competitive alternative.

Up to this point all tests were executed from the perspective of a user in Western Europe where all the resources were deployed. Next we take into account the scenario of a global use with users also in North Central America. To evaluate the perspective of those users, we previously allocated an Azure Ubuntu VM in West US 2. This time, we followed a similar approach by creating an Azure Container Instance in North Central America instead of a VM.

We build our tester image by extending the node:18 public image from Docker Hub. In our personalized version, we installed artitllery and moved our artillery YAML scripts to a directory inside the container, as well as our CSV file with up to 1000 different users. Since we had some problems with the exec instruction to run the scripts inside the container, we also installed a simple SSH server. Using the public IP of the container, we connected to an interactive terminal by SSH and then we ran our test scripts and collected the results for the perspective of a far away user.

In the following chart we compare the latency from the new version with the previous PaaS version, which provided replication mechanisms to support a geo-replicated scenario with users spanning across multiple regions. We ran tests of 30 seconds with rate of 5 concurrent users for blobs service and database service.

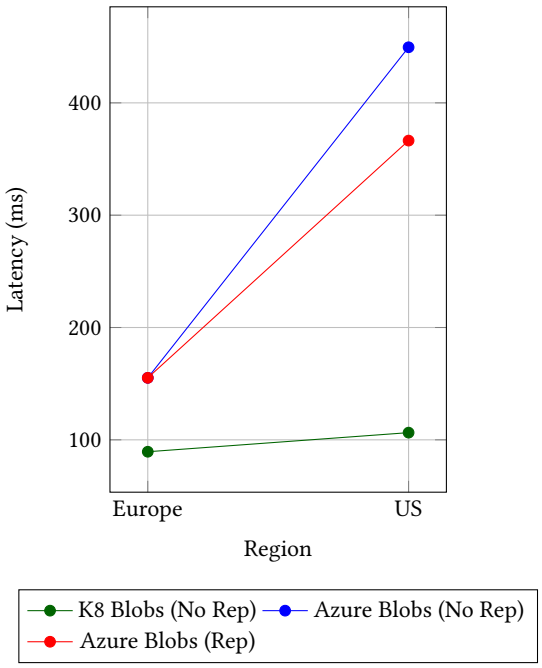

Figure 9: Latency Comparison: K8 Blobs vs Azure Blobs (Replication vs No Replication)

From the chart we can perceive that in the previous version we had a positive impact of geo-replication in the latency perceived by US users. The region to where we send the requests has a great importance in the quality of service of users. In this new version, the Kubernetes alternative does not provide the same geo-replication mechanisms. However, since the blobs in Kubernetes had a better performance than the Azure Blobs, in a geo-replicated scenario they still outperform the first version due to the efficiency of the operations. Still we need to consider that Kubernetes version may be not suitable for a real geo-replicated scenario because the

resources are only deployed in a Kubernetes cluster in West Europe. This way, far away users will experience higher latencies that may compromise the quality of the service of the application. Kubernetes are still a very efficient solution but more suitable to a local scenario for the internal management and use of resources by a particular set of clients such as a private company. For geo-replicated scenarios, Azure PaaS offers the best mechanisms to replicate the app components across different regions to provide good quality of service also to users far way from the primary region.

In our project repository there is a directory for the webapp micro service and a directory for the blobs micro service. In the first one all configuration scripts ca be found under the "kubernetes" directory, including all Dockerfile and configuration files for each component and also the Kubernetes configuration script. In the blobs micro service the scripts for this service can be found in the "docker" directory.

10 CONCLUSIONS

By our experimental evaluation we can conclude that the goal of transforming PaaS TuKano into a new efficient and flexible IaaS based version was achieved with success. To perform that transformation we used Azure IaaS Kubernetes facilities to find good alternatives to replace all the functionalities of the previous version by IaaS based solutions.

We were expecting more limited results from the Kubernetes version because of the complexity and overhead associated with the containers creation and management in the Dockerized version of the app. However, our results shows that in many cases the system could achieved a comparable or even better performance than the previous Azure PaaS components. We also believe that a more complex configuration of the low-level aspects of the infrastructure could improve even better the performance and scalability of the solution, making the Kubernetes version a very competitive and flexible one. The disadvantage is that to achieve that we would need more time and expertise to optimize the infrastructure low-level details. On the other hand, PaaS services of Azure offers a more simple and fast way to deploy scalable applications that support a geo-replicated scenario without the need for the developer to manage the low-level details about the infrastructure where the application is deployed.

At our personal side, we believe that we fulfilled our high expectations and we feel proud of the result that was achieved.