FACULDADE DE CIÊNCIAS E TECNOLOGIA

Project Assignment #2

SSC 2024/2025

Group

Eduardo João Constantino Ervideira — 72420

João Ricardo Bogalho Brilha — 70274

Table of Contents

Architecture Overview

For this second project assignment, we had to migrate from our PaaS solution to a Kubernetes-based infrastructure which required adapting and deploying the Tukano application, originally developed in Project 1.

We structured our application following a microservices approach, splitting it into two main containerized service components:

- **tukano-webapp**: service responsible for handling all user operations that do not involve manipulation of Blob data;
- **tukano-blobs**: service responsible for handling all Blob data operations.

The core services rely on a **PostgreSQL** database for persistent data storage, **Redis** for caching and user session management and persistent volumes for reliable storage of both blob data and database.

To manage external access to these services, we deployed an **nginx ingress controller**[1] that handles and ensures a consistent endpoint within the cluster, preventing issues with session cookies and making it easier to use the application as an end user.

Container Organization

Our approach to containerization emphasizes service independence, focusing on separation of concerns. This allows components to communicate while making it easier to manage resources and scale services on demand.

Blobs Service (tukano-blobs)

In contrast to our Project 1 implementation, where blob storage operations were integrated within the web application, our current architecture isolates these responsibilities into a dedicated blob service.

Built on Tomcat 10.0 with JDK 17, this service is containerized packaging our *blob-service-2.war* into its own Docker image. The service exposes port 8080 for HTTP communication and to ensure data persistence, it uses a dedicated persistent storage volume, which will be detailed in the Storage Solutions section.

Web Application (tukano-webapp)

The web application container represents our core service, also built on Tomcat 10.0 with JDK 17, this containerized service continues to support the key user interaction features from Project 1, including user management, follows, likes, feeds.

It’s also where users must perform their “login”, detailed below in the Authentication section, to be able to interact with the Blobs Service.

Postgres

The PostgreSQL container is deployed using the official PostgreSQL image (*postgres:latest*), with credentials handled via **Kubernetes secrets**.

The container is configured to use a persistent volume mount at */var/lib/postgresql/data/pgdata* to guarantee data durability – additional details can be found in the Storage Solutions section.

Redis

The Redis container is also deployed using the official Redis image (*redis:latest*).

It’s used to store user session information for authenticated operations, as well as information about recently accessed shorts and users to ease the strain on our database.

Kubernetes Infrastructure

Deployment Configuration

Blob Service

The Blob Service deployment is configured to run a single replica and uses our image (jbrilha/scc2425-tukano2-blobs:latest), hosted on Docker Hub, with **`imagePullPolicy: Always`** to ensure we always run the latest deployed version.

This deployment includes essential environment variables, such as:

- BLOB_STORAGE_PATH: Defines the storage path as */app/blobs*;
- REDIS_URL: Points to the *redis-service* for caching;
- WEBAPP_SERVICE_URL: Points to *tukano-webapp-service* to communicate with the web application;
- Sensitive environment variables like REDIS_KEY and SECRET are managed through Kubernetes secrets.

Network configuration uses Kubernetes' ClusterIP service type, exposing port 8080 which maps to the container's target port 8080, ensuring the service is only accessible within the cluster.

To maintain data persistence, a dedicated volume is mounted at */app/blobs/*, with further details provided in the Storage Solutions section.

The deployment includes a node selector constraint (**`kubernetes.io/os: linux`**) to ensure proper platform compatibility.

Web Application Service

Similar to our Blob Service, the Web Application deployment also runs a single replica and uses our image (jbrilha/scc2425-tukano2:latest), hosted on Docker Hub with the same image pull policy as before.

As before, this deployment includes the necessary environment variables, such as:

- REDIS_URL: Points to the *redis-service* for caching;
- POSTGRES_HOST: Points to the *postgres-service* for database access;
- POSTGRES_PORT: Defines database port as 5432;
- BLOBS_SERVICE_URL: Points to *tukano-blobs-service* for blob storage operations;
- Sensitive environment variables like REDIS_KEY, SECRET, POSTGRES_DB, POSTGRES_USER and POSTGRES_PASSWORD are managed through Kubernetes secrets.

Following the same network configuration as the Blob Service, it uses ClusterIP service type to expose port 8080 for internal communication, with external access controlled by our ingress controller.

Postgres

The PostgreSQL deployment runs a single replica using the official PostgreSQL image. It also includes essential environment variables such as:

- POSTGRES_DB;
- POSTGRES_USER;
- POSTGRES_PASSWORD;
- PGDATA: Defines the data directory as /var/lib/postgresql/data/pgdata.

All database credentials are managed, once again, through Kubernetes secrets.

Network configuration follows the same ClusterIP approach, exposing port 5432 which maps to the container's target port 5432, ensuring the database is only accessible within the cluster.

Redis

The Redis deployment runs a single replica using the official Redis image and includes specific resource constraints for performance management.

This deployment includes resource limits and requests:

- Memory: 256Mi (requests) to 512Mi (limits)
- CPU: 500m (requests) to 1000m (limits)

The Redis password is configured through the REDIS_KEY environment variable and passed as an argument to the Redis server. Network configuration follows the same ClusterIP approach as other services, exposing port 6379 which maps to the container's target port 6379, ensuring Redis is only accessible within the cluster.

Ingress

The Ingress configuration is split into two parts to handle routing for our different services. Both use the nginx ingress controller and implement regex-based path matching.

For the Blob Service, requests to **`/tukano/rest/blobs/*`** are routed to the tukano-blobs-service, with the path rewritten to **`/blob-service-2/rest/blobs`**.

Similarly, for the Web Application, all other requests to **`/tukano/rest/*`** (excluding blob paths) are routed to tukano-webapp-service and rewritten to **`/webapp-2/rest/`**.

Both Ingress configurations target port 8080 of their respective services and allow external access to our services through a single domain.

Secrets

Our secrets are managed through Kubernetes Secrets and are stored in a single secret resource named *tukano-secrets*. This includes credentials for Redis and Postgres, as well our secret for token generation.

These are referenced as environment variables in the services mentioned above, ensuring that credentials are not exposed in configuration files. With this approach it's easier to maintain and manage our secrets.

Storage Solutions

Our infrastructure uses Persistent Volumes and Persistent Volume Claims[2] to achieve data persistence.

The blob storage solution consists of a Persistent Volume mounted at /data/blob-storage on the host system. This volume is claimed through *blob-storage-pvc* and afterwards mounted in our blob service container at */app/blobs/*. The capacity allocated was ***1GB*** and the access mode chosen was ***ReadWriteOnce*** (which is sufficient since our blob service and database each run as single replicas).

For database persistence, we've configured a separate Persistent Volume mounted at */data/postgres-storage* on the host system too. This volume is claimed by *postgres-pvc* and mounted to PostgreSQL's data directory. This ensures that the state of the database is not lost even if the container is restarted or pods are rescheduled, thus preventing data loss. Regarding capacity and access mode, the same decision was made as for the blob service (***1GB*** and ***ReadWriteOnce***).

While both storage solutions currently utilize hostPath volumes, a more robust and flexible approach for real-world scenarios would involve transitioning to Azure Managed Disks, which offer benefits like automatic provisioning, scalability, and high availability.

Implementation Details

Functions

To replace the previous Azure Functions implementations of the ViewCounter and TukanoRecommendations, we implemented REST endpoints in our Web application to support the same functionalities, as follows:

- ### Counting Views

A simple REST endpoint to insert or update the Stats table in our database, each entity of this table contains only the `shortId` and an integer counter of views it received. To ensure the counter grows monotonically, we use a transaction to perform the increment operation.

The endpoint is triggered in the Blobs service for every download request to a given blob, and there is a Token validation step that is never shown to the end user, to ensure that views are not altered by anybody outside of the application.

- ### TukanoRecommends

Another REST endpoint, this time triggered by a Kubernetes CronJob that runs every day at midnight.

The criteria for recommending shorts is to select the top 5 most viewed and the top 5 most liked shorts from the database. In this case what we consider to be a “republish” is simply changing the `ownerId` to be the “tukRecs” account, as well as prepending it to the existing shortId to differentiate it from the original short, while the blobUrl remains the same so that users can download it from the source, thus avoiding republished shorts taking up unnecessary storage.

Users see these recommendations on their feed because, on account creation, they are made to follow the **tukRecs** account.

The recommendations are reset every time the endpoint is triggered, so if a user happens to delete their shorts that were previously selected, the republished versions will only be visible until the next update cycle of the recommendations.

Authentication

To authenticate users for operations on the Blobs Service, we used a Cookie-based approach similar to what was presented in Lab9, the key difference being that instead of a separate, dedicated REST endpoint, we opted to make the **`createUser`** and **`getUser`** operations act as a form of login themselves.

To achieve this, in addition to the request filter that would extract cookies from requests and place them in a ThreadLocal map, and without having to change the API to return Response types, we implemented a response filter that would do the inverse: set cookies on outgoing responses if they were present in the ThreadLocal storage for the aforementioned operations.

The cookies themselves contain the **`Session.uid`**, which is the key used to store active Sessions in the Redis Cache, with the pattern **`"session:" + <uid>`**, so that they can be retrieved easily and independently from the Blobs Service.

While on the topic of Authentication, for the purposes of blob deletion, a user is considered an administrator if their username is "**admin**". We recognize this isn't a particularly solid approach, but due to time constraints, and seeing as permission/role management isn't the core focus of this class/project, we opted to keep it simple so we could turn our attention to the more fundamental aspects of the project's development.

Test Results

In this section we start by analysing the results obtained from our test runs, with and without the use of Redis Cache as a layer between our application and the database, making use of graphics obtained from Artillery's Cloud Dashboard to aid in visualizing the behavior of our services.

The analysis will focus on the 95th and 99th percentile of request latency and the average and peak throughput measured in requests per second; so unless specified, the latency comparisons are referring to those percentiles.

Following the analysis, we contrast the behavior of the initial Azure Webapp deployment with the new Kubernetes based one, to draw conclusions on the performance of these alternatives for each of the test scenarios.

It's important to mention that for these tests we changed the way random users are generated by Artillery, this was done to ensure consistent uniqueness of userIds leading to less vUser failures due to incorrect testing rather than incorrect implementation of our application.

The first two tests share the same configuration:

```
target: http://4.208.85.132/tukano/rest
phases:
  - name: Warm up
    rampTo: 5
    duration: 10
    arrivalRate: 1
  - name: Main test
    duration: 20
    arrivalRate: 10
```

Visualization & Analysis

1. `creation_mixed.yaml`

This test scenario replicates common user behaviors in two distinct ways. The first consists of content creators who register on the platform and upload short videos. The second illustrates users who engage in social interaction by creating content, following other users, and liking shorts.

Attending to the results presented just below, we can observe a slight *decrease* in performance when using Redis, in the order of tens of milliseconds for the requests' latency on this test, with almost 3 full seconds of difference between the maximum latency observed.

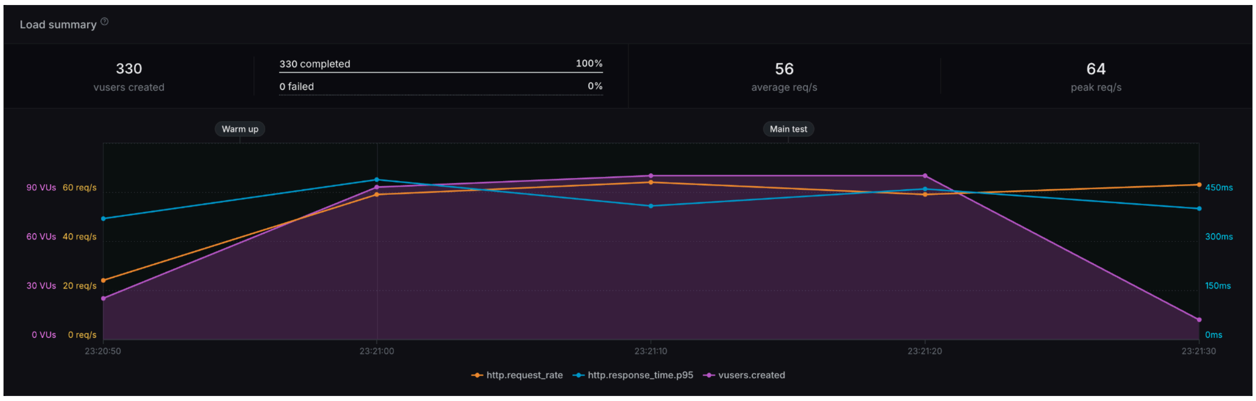
However, the difference in throughput is not very significant, with only one more request per second at peak load when not using Redis.

Compared to the first project, there is a considerable difference in performance across the board, detailed in each test's section.

1.1 Without Cache

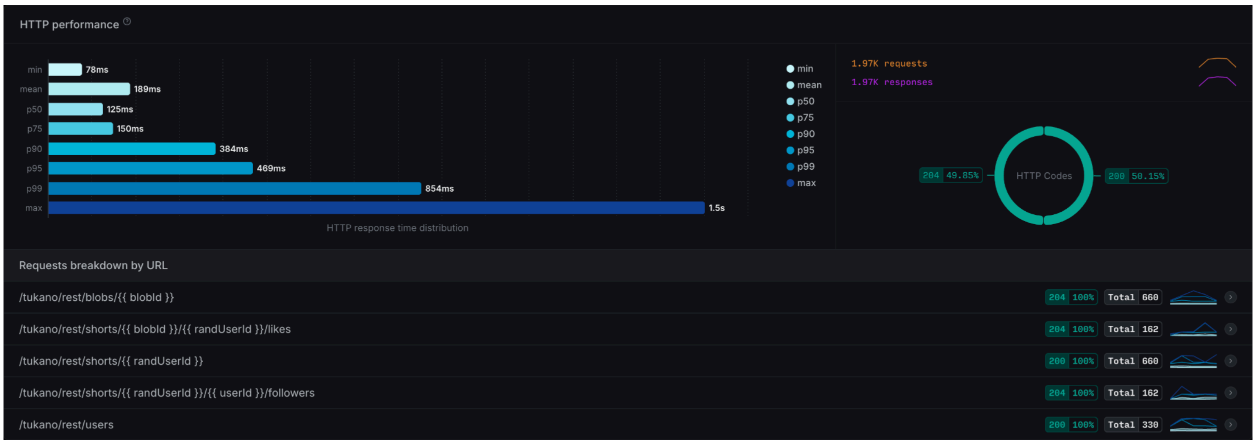
The latency experienced with this deployment is better than the CosmosDB NoSQL implementation by around 300 milliseconds, and between 2 and 3.5 seconds better than the CosmosDB for Postgres implementation.

While the throughput also shows an improvement of over 10 req/s on average and 5 req/s in peak compared to NoSQL, there is an astounding difference of 40 req/s compared to the previous Postgres implementation.


Load summary

330 vusers created	330 completed 100% / 0 failed 0%	56 average req/s	64 peak req/s

Fig. 1.1 a) — Load Summary


HTTP performance

Requests breakdown by URL

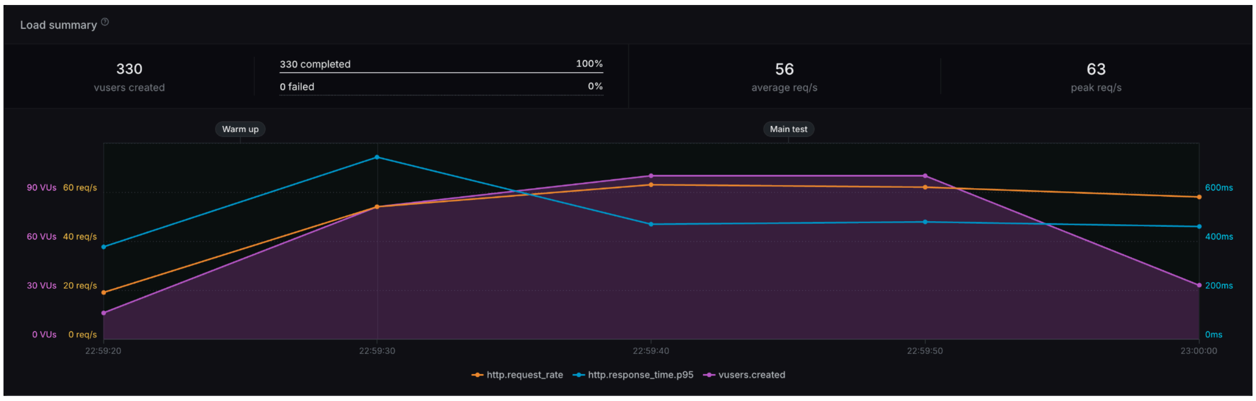
/tukano/rest/blobs/{{ blobId }}	204 100%	Total 660
/tukano/rest/shorts/{{ blobId }}/{{ randUserId }}/likes	204 100%	Total 162
/tukano/rest/shorts/{{ randUserId }}	200 100%	Total 660
/tukano/rest/shorts/{{ randUserId }}/{{ userId }}/followers	204 100%	Total 162
/tukano/rest/users	200 100%	Total 330

Fig. 1.1 b) — HTTP Performance

1.2 With Cache

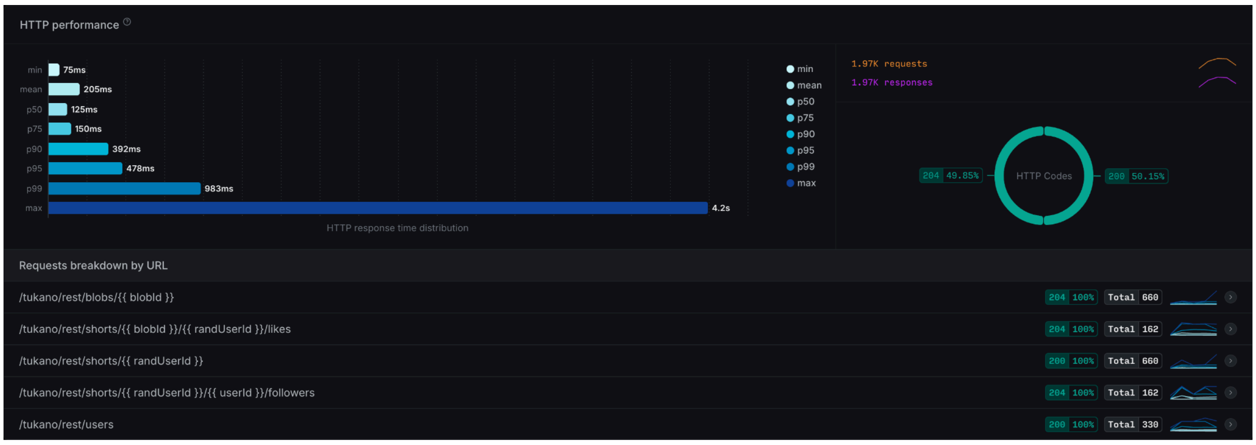
In this case, the latency observed with the previous project's deployment when using CosmosDB NoSQL is actually better than the the current implementation, with the latter showing an increase in latency of almost 300 milliseconds for the 95th and over 600 milliseconds for the 99th percentiles of requests, with CosmosDB Postgres still underperforming by around 350 and 1000 milliseconds for the respective percentiles.

Throughput still shows an improvement, with an increase of around 5 req/s on average and peak compared to NoSQL, and around 7 req/s when compared to CosmosDB Postgres.


Load summary

330 vusers created	330 completed 100% / 0 failed 0%	56 average req/s	63 peak req/s

Fig. 1.2 a) — Load Summary


HTTP performance

min	75ms
mean	205ms
p50	125ms
p75	150ms
p90	392ms
p95	478ms
p99	983ms
max	4.2s

HTTP response time distribution

1.97K requests
1.97K responses

HTTP Codes: 204 49.85% — 200 50.15%

Requests breakdown by URL

/tukano/rest/blobs/{{ blobId }}	204 100%	Total 660
/tukano/rest/shorts/{{ blobId }}/{{ randUserId }}/likes	204 100%	Total 162
/tukano/rest/shorts/{{ randUserId }}	200 100%	Total 660
/tukano/rest/shorts/{{ randUserId }}/{{ userId }}/followers	204 100%	Total 162
/tukano/rest/users	200 100%	Total 330

Fig. 1.2 b) — HTTP Performance

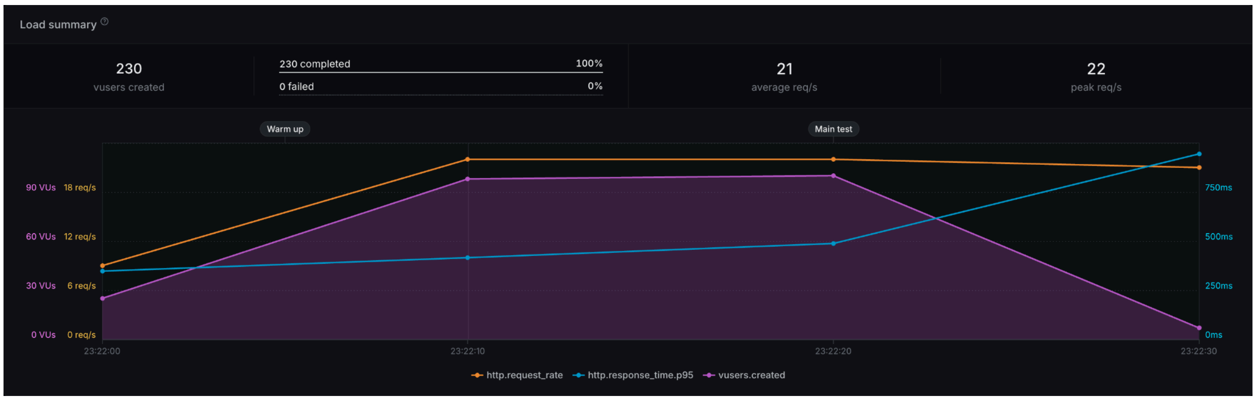
2. `access_mixed.yaml`

This test scenario replicates common user behaviors when interacting with the application as viewers rather than posters, it includes the retrieval of users' feeds, likes, shorts (with downloads), followers and even searching for users.

2.1 Without Cache

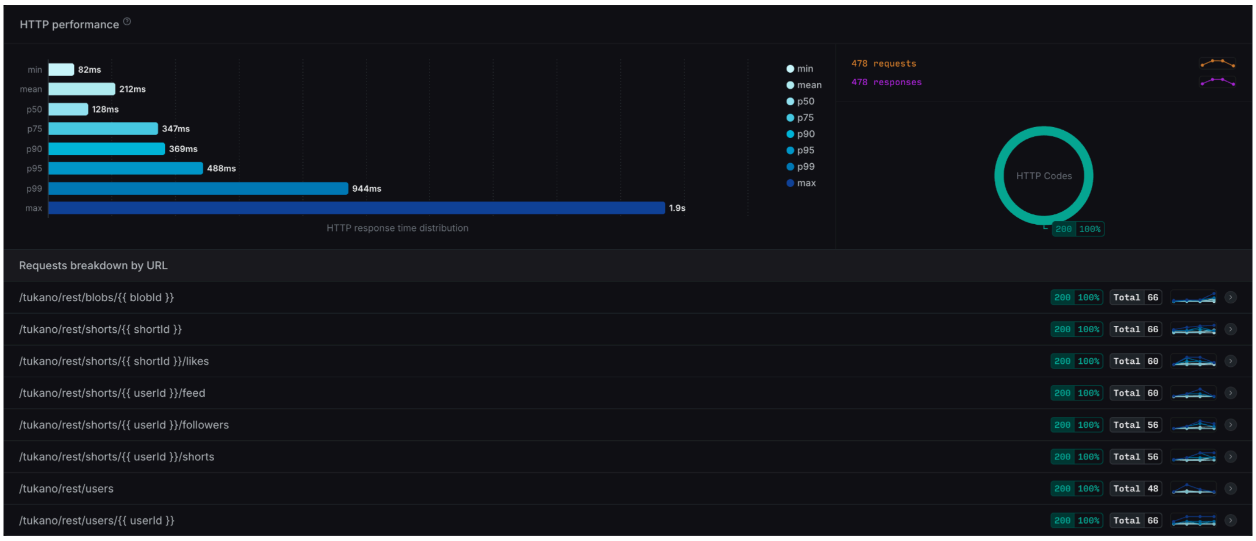
Regarding access tests, the previous implementation using CosmosDB NoSQL showed less latency than the current one, by about 200 and 300 milliseconds for the 95th and 99th percentile of latencies; however, and consistent with the results obtained so far, the CosmosDB Postgres version had worse performance than the containerized version, with a difference of around 400 and 500 milliseconds in the respective percentiles.

Throughput had a slight improvement in the new version: compared to CosmosDB NoSQL there were 4 and 2 additional req/s on average and in peak respectively; and compared to CosmosDB Postgres there was only an increase of 2 req/s in both metrics.


Load summary

230 vusers created	230 completed 100% / 0 failed 0%	21 average req/s	22 peak req/s

Fig. 2.1 a) — Load Summary


HTTP performance

min	82ms
mean	212ms
p50	128ms
p75	347ms
p90	369ms
p95	488ms
p99	944ms
max	1.9s

HTTP response time distribution

478 requests
478 responses

Requests breakdown by URL

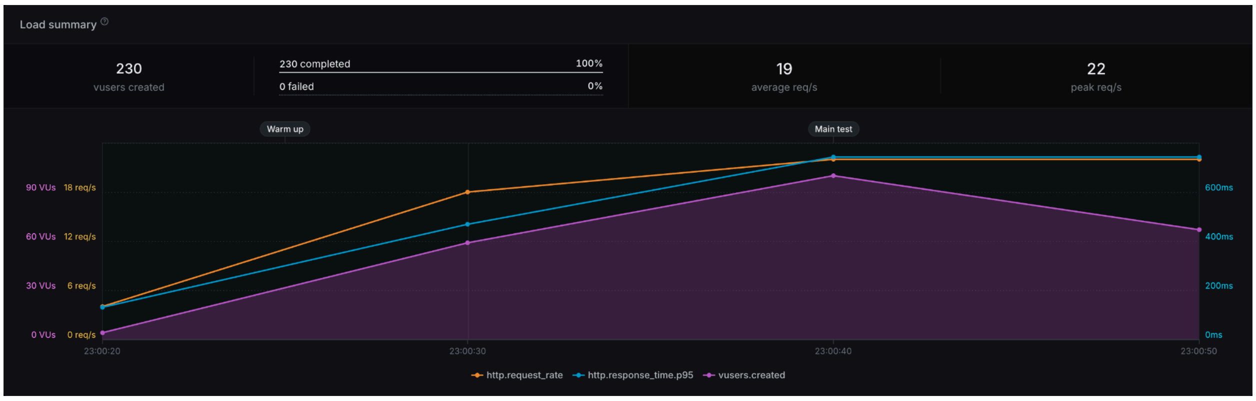
/tukano/rest/blobs/{{ blobId }}	200 100%	Total 66
/tukano/rest/shorts/{{ shortId }}	200 100%	Total 66
/tukano/rest/shorts/{{ shortId }}/likes	200 100%	Total 60
/tukano/rest/shorts/{{ userId }}/feed	200 100%	Total 60
/tukano/rest/shorts/{{ userId }}/followers	200 100%	Total 56
/tukano/rest/shorts/{{ userId }}/shorts	200 100%	Total 56
/tukano/rest/users	200 100%	Total 48
/tukano/rest/users/{{ userId }}	200 100%	Total 66

Fig. 2.1 b) — HTTP Performance

2.2 With Cache

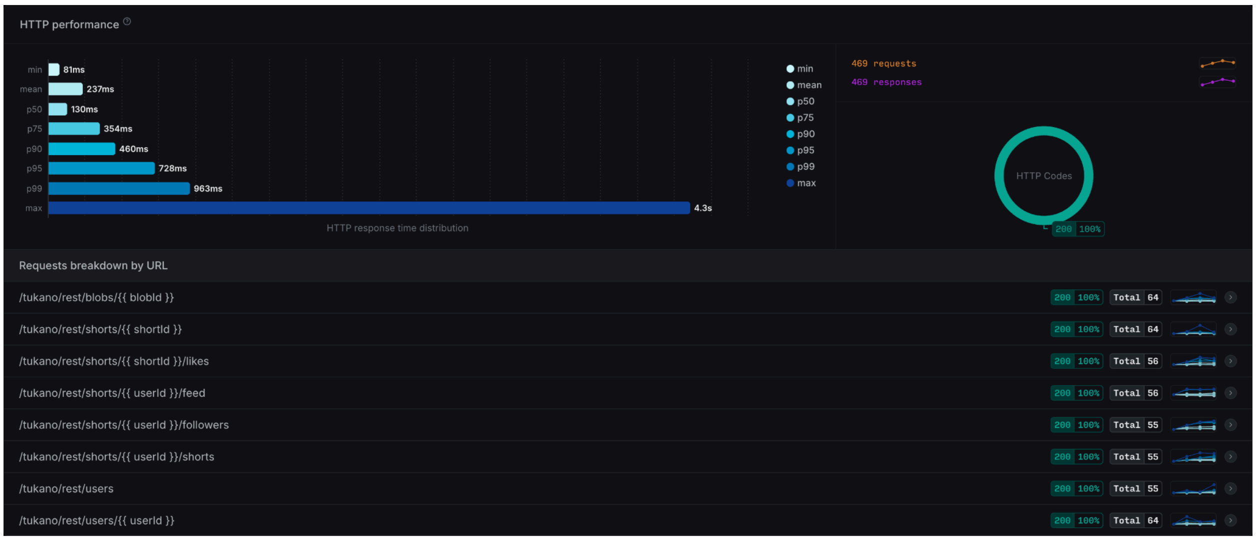
In this situation, the use of Cache made a tremendous difference on the previously deployed solution, with the CosmosDB NoSQL version outperforming the current implementation by over 600 milliseconds in both percentiles, while the CosmosDB Postgres had considerably worse performance with latency increases of close to 3 seconds.

Throughput, as we've shown so far, still showed an improvement in the Kubernetes based deployment, with the number of requests per second increasing by 2 and 4 on average and peak compared to NoSQL, and only increasing by 2 in peak compared to non-containerized Postgres.


Load summary

230 vusers created	230 completed 100% / 0 failed 0%	19 average req/s	22 peak req/s

Fig. 2.2 a) — Load Summary


HTTP performance

min	81ms
mean	237ms
p50	130ms
p75	354ms
p90	460ms
p95	728ms
p99	963ms
max	4.3s

469 requests
469 responses

Requests breakdown by URL

/tukano/rest/blobs/{{ blobId }}	200	100%	Total	64
/tukano/rest/shorts/{{ shortId }}	200	100%	Total	64
/tukano/rest/shorts/{{ shortId }}/likes	200	100%	Total	56
/tukano/rest/shorts/{{ userId }}/feed	200	100%	Total	56
/tukano/rest/shorts/{{ userId }}/followers	200	100%	Total	55
/tukano/rest/shorts/{{ userId }}/shorts	200	100%	Total	55
/tukano/rest/users	200	100%	Total	55
/tukano/rest/users/{{ userId }}	200	100%	Total	64

Fig. 2.2 b) — HTTP Performance

3. `creation_access_mixed.yaml`

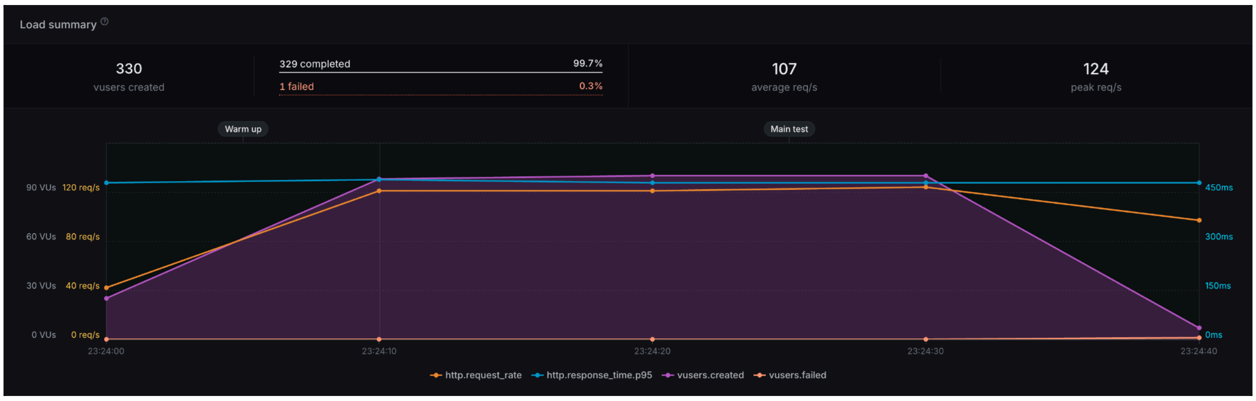
This test scenario essentially performs all the same operations that the previous tests perform, but does so simultaneously with more complex flows per vUser, this time we decided to run these tests without Redis as well because of what we had noticed regarding its impact in the previous tests.

The configuration for this test's phases was a little different than the previous two, yet entirely the same as the one used in the previous deployment, to allow for more requests to come in:

```
target: http://4.208.85.132/tukano/rest
phases:
  - name: Warm up
    rampTo: 5
    duration: 10
    arrivalRate: 1
  - name: Main test
    duration: 30
    arrivalRate: 10
```

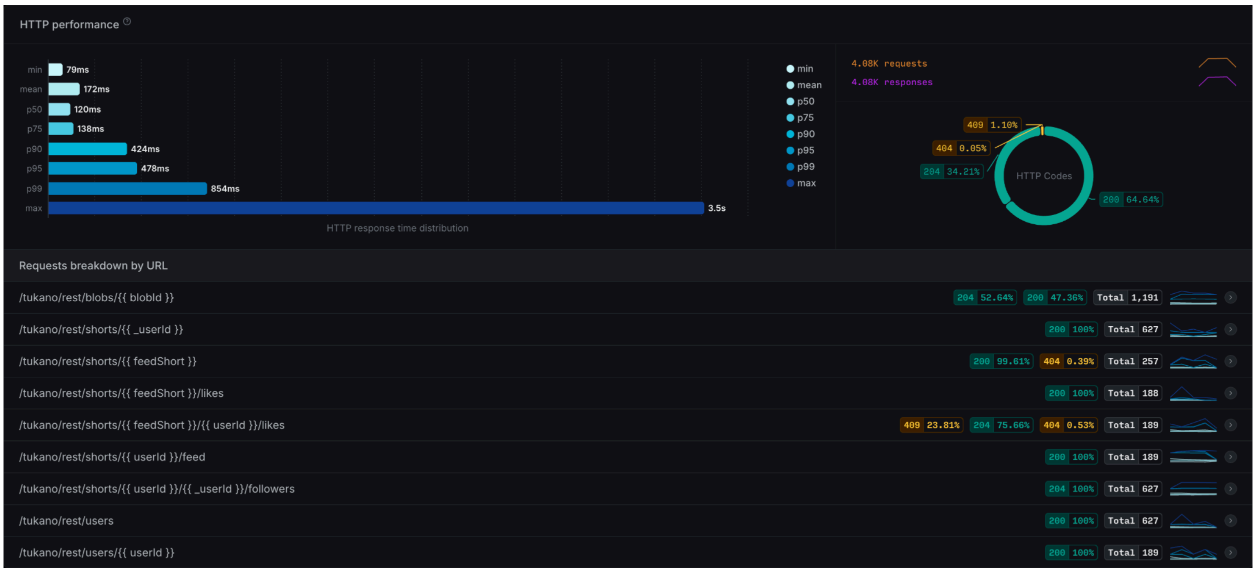
3.1 Without Cache

In the first project's implementation we ran these tests exclusively with Redis enabled, so we will not be comparing the values observed in 3.1a) and 3.1b) with the results obtained in the previous solution.


Load summary

330 vusers created	329 completed 99.7% / 1 failed 0.3%	107 average req/s	124 peak req/s

Fig. 3.1 a) — Load Summary


HTTP performance

min	79ms
mean	172ms
p50	120ms
p75	138ms
p90	424ms
p95	478ms
p99	854ms
max	3.5s

4.08K requests
4.08K responses

409 1.10%
404 0.05%
204 34.21%
200 64.64%

Requests breakdown by URL

/tukano/rest/blobs/{{ blobId }}		204 52.64%	200 47.36%	Total 1,191
/tukano/rest/shorts/{{ _userId }}			200 100%	Total 627
/tukano/rest/shorts/{{ feedShort }}		200 99.61%	404 0.39%	Total 257
/tukano/rest/shorts/{{ feedShort }}/likes			200 100%	Total 188
/tukano/rest/shorts/{{ feedShort }}/{{ userId }}/likes	409 23.81%	204 75.66%	404 0.53%	Total 189
/tukano/rest/shorts/{{ userId }}/feed			200 100%	Total 189
/tukano/rest/shorts/{{ userId }}/{{ _userId }}/followers			204 100%	Total 627
/tukano/rest/users			200 100%	Total 627
/tukano/rest/users/{{ userId }}			200 100%	Total 189

Fig. 3.1 b) — HTTP Performance

3.2 With Cache

This test displays a similar performance difference to what the “simpler” tests showed previously, with the NoSQL implementation showing less latency by around 300 and 600 milliseconds in the 95th and 99th percentile, while the previous Postgres implementation showed around 400 milliseconds of improvement in both.

As for throughput, the current implementation shows the most improvement in this test, with an increase of around 40 req/s compared to both previous implementations.

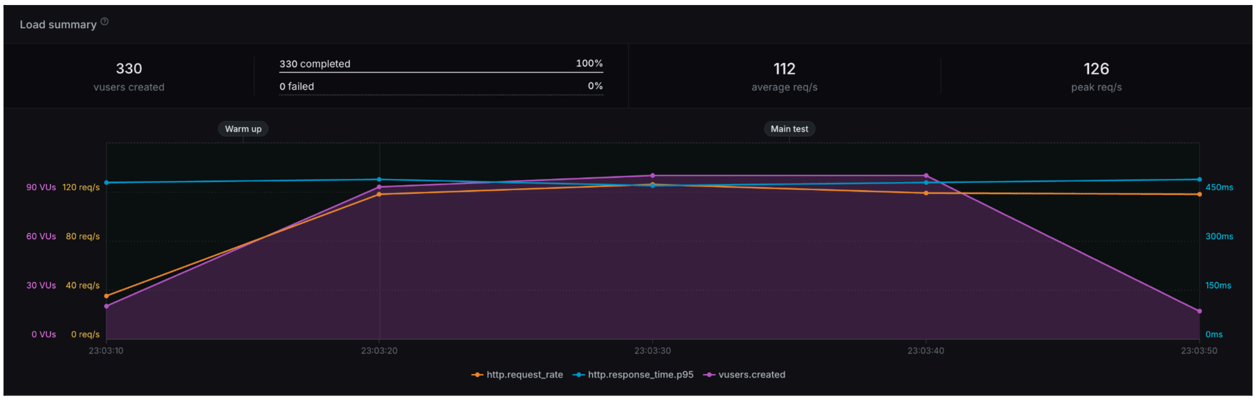
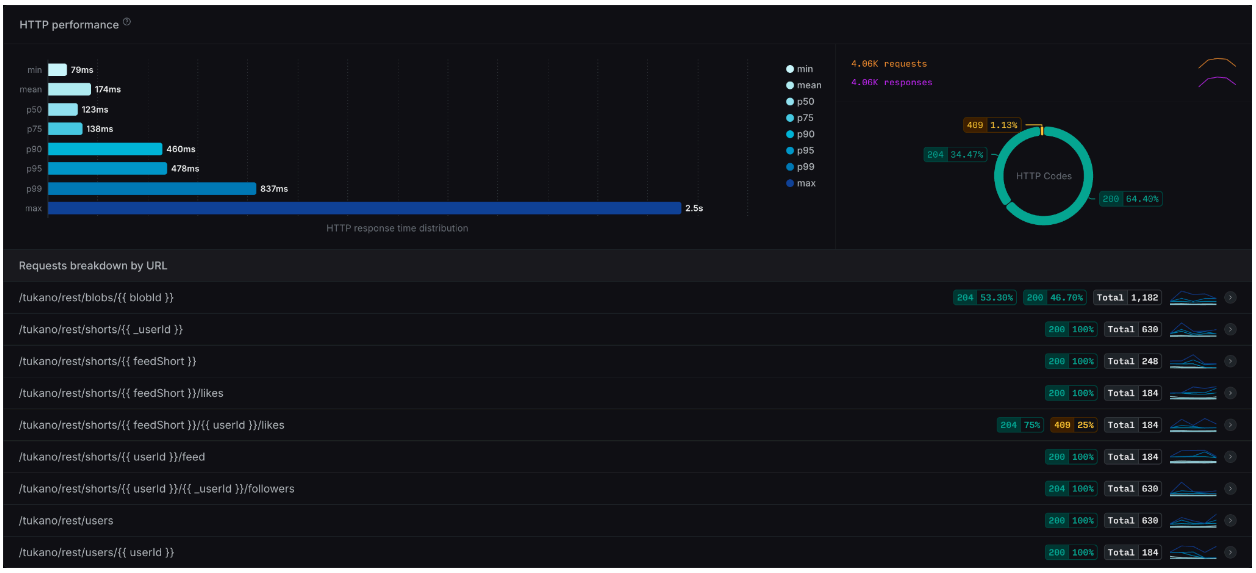


Fig. 3.2 a) — Load Summary



Requests breakdown by URL

/tukano/rest/blobs/{{ blobId }}	204 53.30%	200 46.70%	Total 1,182
/tukano/rest/shorts/{{ _userId }}		200 100%	Total 630
/tukano/rest/shorts/{{ feedShort }}		200 100%	Total 248
/tukano/rest/shorts/{{ feedShort }}/likes		200 100%	Total 184
/tukano/rest/shorts/{{ feedShort }}/{{ userId }}/likes	204 75%	409 25%	Total 184
/tukano/rest/shorts/{{ userId }}/feed		200 100%	Total 184
/tukano/rest/shorts/{{ userId }}/{{ _userId }}/followers		204 100%	Total 630
/tukano/rest/users		200 100%	Total 630
/tukano/rest/users/{{ userId }}		200 100%	Total 184

Fig. 3.2 b) — HTTP Performance

Comparison Summary

To wrap up these comparisons, we'd like to explain the circumstances in which we'd choose one deployment structure over the other.

One big factor is ease of use and management. Azure tooling is quite extensive, and because of this, navigating it sometimes becomes more complex than the goal we're trying to achieve, especially when compared to orchestrating containers and deploying them via Kubernetes.

With this in mind, we would opt to use a K8s based approach for applications that require a lot of separate components (services) working together, as opposed to single monolith applications.

Another benefit of using Kubernetes is its approach to containerization, which makes it almost trivial to develop and deploy new versions of our services, with the ability to build on top of existing images if need be. Additionally, the ability to define resource limits and replica count also makes it particularly easy to fine tune our deployment to meet the needs of the end user.

On the other hand, as the performance tests have shown, if the application (or its users) require the lowest possible latency, then an Azure managed solution would be a better fit. Despite the initial hurdles related to deployment and organization of different Azure services, once these are stabilized in the platform, there isn't much of an issue, if there is one at all, in keeping our application up to date and scalable.

One thing to note, however, is that the presence of an ingress controller in the Kubernetes deployment can also affect the latency of the services provided, so if the application were to remain a single monolith, perhaps the points in favor of an Azure managed solution would become void. We have no way of testing that at this time but the key point remains that there are certain trade-offs that must be made with each approach, depending on the overarching goals and requirements of our application.

Conclusion

In conclusion, having heard of Kuberntes but never having tried it before, we now understand the issues it solves and why it’s such a widely used and popular tool.

We also come away with a better understanding of how different approaches to deployment can affect our development time and concerns, noting that with Kubernetes we could spend most of our time simply focusing on implementation and architecture rather than deployment nuances.

Just like the previous report’s graphics, the ones shown here were generated by Artillery on their Cloud Dashboard, when running the following command: **`artillery run our_test.yaml --record --key our_api_key`**.[3]

Bibliography

[1] — About | NGINX Ingress Controller.
(October 07, 2024). [Consult. 04 dec. 2024].
Available at:
https://docs.nginx.com/nginx-ingress-controller/overview/about/

[2] — Persistent Volumes | Kubernetes.
(October 18, 2024). [Consult. 04 dec. 2024].
Available at:
https://kubernetes.io/docs/concepts/storage/persistent-volumes/

[3] —Artillery Cloud | Artillery Docs.
(October 23, 2024). [Consult. 09 nov. 2024].
Available at:
https://www.artillery.io/docs/get-started/artillery-cloud#configure-the-cli-to-send-data-to-artillery-cloud