FACULDADE DE
CIÊNCIAS E TECNOLOGIA

Project Assignment #1

SSC 2024/2025

Group

Eduardo João Constantino Ervideira — 72420

João Ricardo Bogalho Brilha — 70274

Table of Contents

Data Structures Modifications

User:

Renamed attribute "`userId`" to "`id`": To comply with Azure Cosmos DB's requirement for a unique id field in each document.

Short:

Renamed attribute "`shortId`" to "`id`": Similar to the User structure, to match Cosmos DB's unique identifier requirements;

Added attribute "`totalViews`": To implement the Counting Views functionality. This field is similar to "`totalLikes`" in that it doesn't store the counter in the object itself, instead serving as the field to which these values are assigned when getting a short. The total number of views for each short is incremented based on blob downloads.

Likes:

Added attribute "`id`": An id field was added to the data structure to also comply with Cosmos DB's requirements.

Design Choices

CosmosDB NoSQL Containers

Data storage is organized into containers with unique document ID formats for each entity:

- **users**: Each user has a unique document ID.
- **shorts**:
  - Uses "`userId+UUID`" as the document ID.
- **likes:**
  - Uses "`userId_shortId`" as the document ID.
- **following**:
  - Uses "`userId1_userId2`" as the document ID, representing follower-followee relationship.
- **stats**:
  - Follows the same ID format as Shorts, associating stats with specific shorts for easy access.

Azure Functions

Due to the superior performance of CosmosDB NoSQL over PostgreSQL, we chose to implement our Azure Functions with exclusive support for NoSQL.

The specifics of each implemented function are documented in the next section.

Database & Cache

Initially, we ran tests for every endpoint available in our application, just to get an understanding of what to expect with more complex testing flows and usage patterns. Then, to determine the final implementation of our database system that we should settle on, we ran tests focused on two distinct use cases, one focused on the creation of users and content, and the other focused on accessing said content.

We ran these tests for four different deployment scenarios, CosmosDB for PostgreSQL and CosmosDB for NoSQL, each with and without Redis caching enabled. They all had the same configuration for the phases:

```
target: https://tukano7420.azurewebsites.net/rest
phases:
  - name: Warm up
    rampTo: 5
    duration: 10
    arrivalRate: 1
  - name: Main test
    duration: 20
    arrivalRate: 10
```

The graphs presented in the section before the conclusion help understand the reasoning behind our choice of database implementation. In these graphs, and focusing on HTTP response times, we can observe that NoSQL outperformed PostgreSQL in every test, which immediately pointed us in the direction we wanted to take. The

difference in performance when using Redis is also considerable, marking a noticeable improvement compared to not using it for either database, in both scenarios.

Regarding the creation tests specifically, we can see that there were more errors when using CosmosDB NoSQL for user creation than there were when using Postgres, we suspect these are directly related to the way we generate users randomly in those tests, as we can't reliably reproduce the 500 errors otherwise; The 409 Conflict errors are common in both Database implementations because of that same random user generation that turned out to not be as random as we thought.

For these reasons, we didn't consider the errors as a significant factor when determining which implementation to go with, focusing instead on performance factors such as requests per second and their response time, which ultimately led to our decision of using CosmosDB for NoSQL, with Redis caching, as the final implementation in our application.

Azure Functions

Counting Views (HTTP Trigger)

We implemented an HTTP Trigger function that is triggered by an HTTP request after each blob download method. This function is responsible for incrementing the view counter of the corresponding Short.

To ensure that view counters grow monotonically, we used the **`patchItem`** method from the Azure Cosmos DB API. This method applies atomic patch operations and executes them sequentially.[1]

Delete User Information (HTTP Trigger)

Another HTTP Trigger function, this time triggered by the deleteAllShorts method once a user is deleted from the system. It ensures the deletion of all that user's shorts, likes, following relationships, and even invalidates the cache for any shorts that might have been present, as well as clear their statistics, namely the views, so TukanoRecommends doesn't think they still exist. This was done to make the process more efficient and reduce the workload on the main application server.

Follow Tukano Recommends (HTTP Trigger)

Another HTTP Trigger function to automatically make users follow the "TukanoRecommends" system user. When a new user is created, this function is invoked with their user ID.

This ensures that all users will see the TukanoRecommends shorts in their feed from the start.

Tukano Recommends (Timer Trigger)

To periodically recommend "popular shorts" to users, we implemented an Azure Timer Trigger function. Running once a day at midnight, it retrieves the top 5 most liked shorts from the Likes container and the top 5 most viewed shorts from the Stats container in CosmosDB. It then combines these results, removing any duplicates, to create a single list of shorts to recommend.

For each of these shorts, the function copies the original blob content to a new one named "tukRecs" + the original short ID. It also creates a new Short document in the Shorts container, linking to the copied blob and setting "tukRecs" as the owner.

Test Results Visualization

In this section we present the aforementioned graphs to help illustrate why we went in the direction we did. We've separated them into two subsections, one for each test scenario, with respective indications of which ones are using Redis caching.

The final test presented was written after the submission, so it serves more to illustrate that we made the right choice, as it's only a comparison between NoSQL and Postgres, both with cache, in a scenario that is a combination of both previous scenarios. This test was more complex to write hence why we didn't use it to inform our decision but instead to confirm it.

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.

This workload simulates real-world use to evaluate system performance across alternative database configurations.

1.1. Without Cache

NoSQL

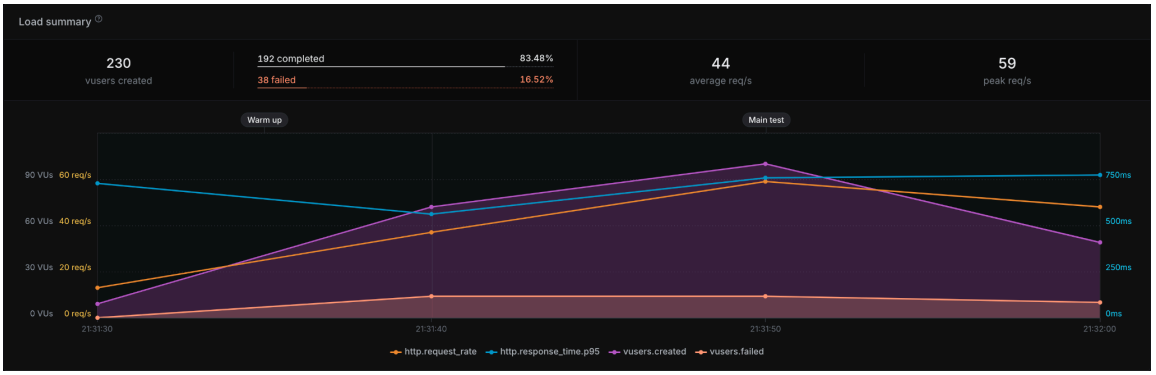

Fig. 1.1 a) — Load Summary

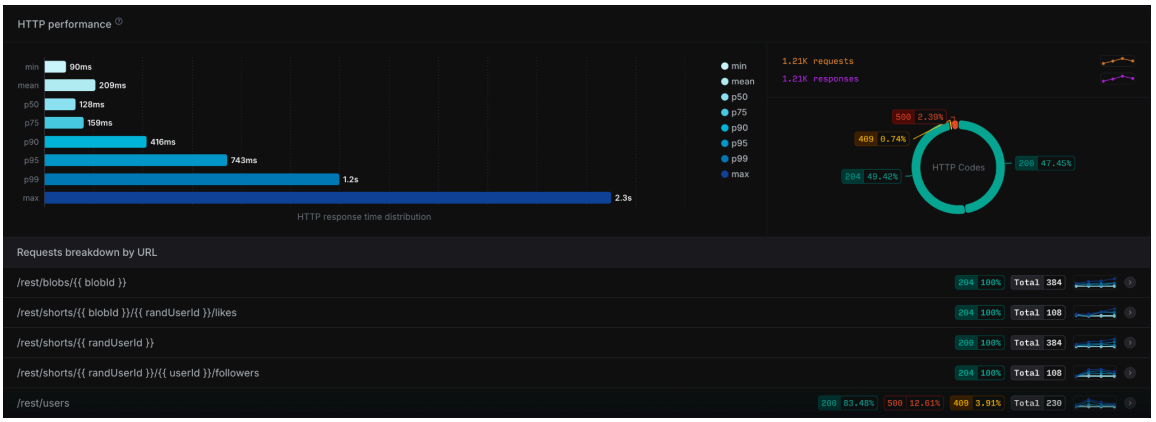

Fig. 1.1 b) — HTTP Performance

PostgreSQL

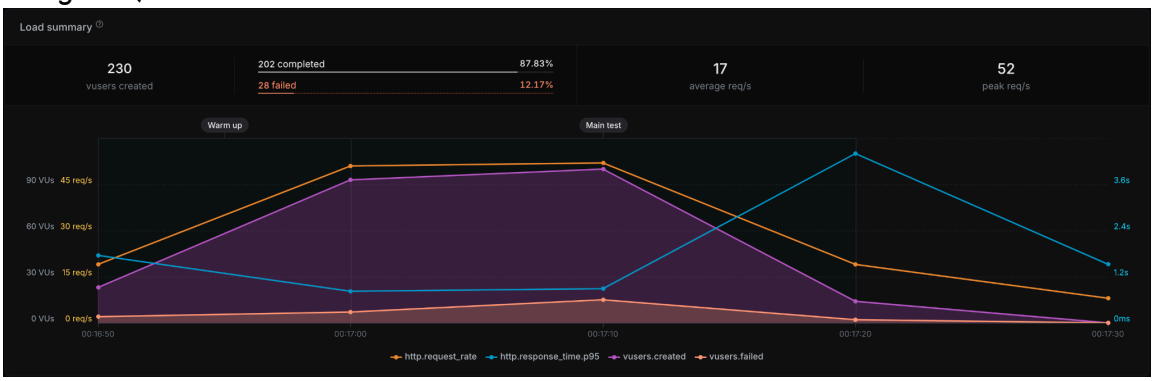

Fig. 1.1 c) — Load Summary

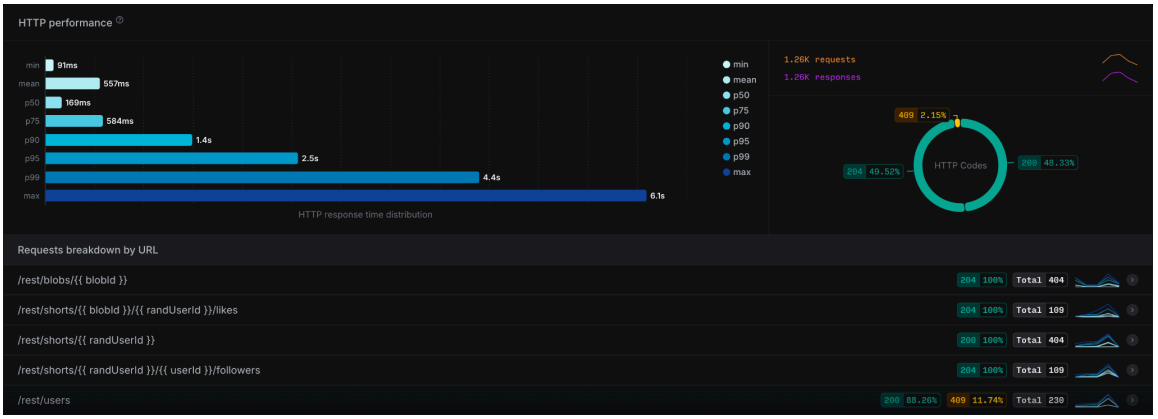

Fig. 1.1 d) — HTTP Performance

1.2. With Cache

NoSQL

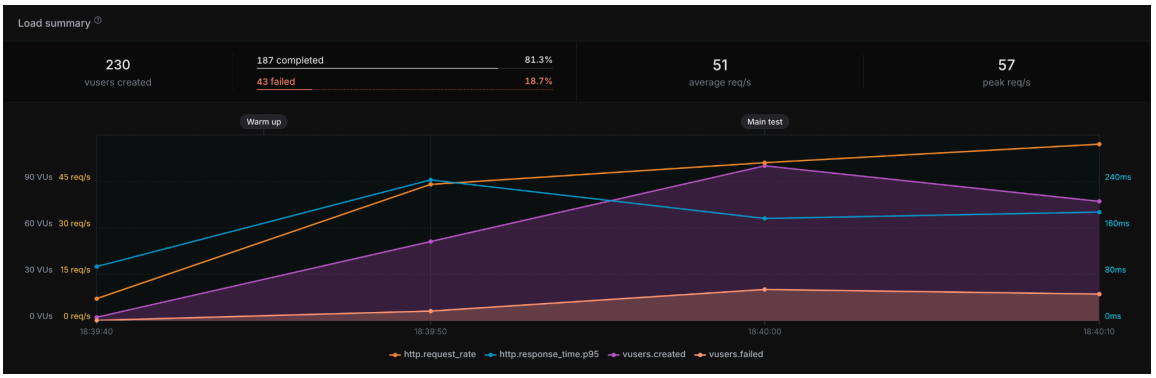

Fig. 1.2 a) — Load Summary

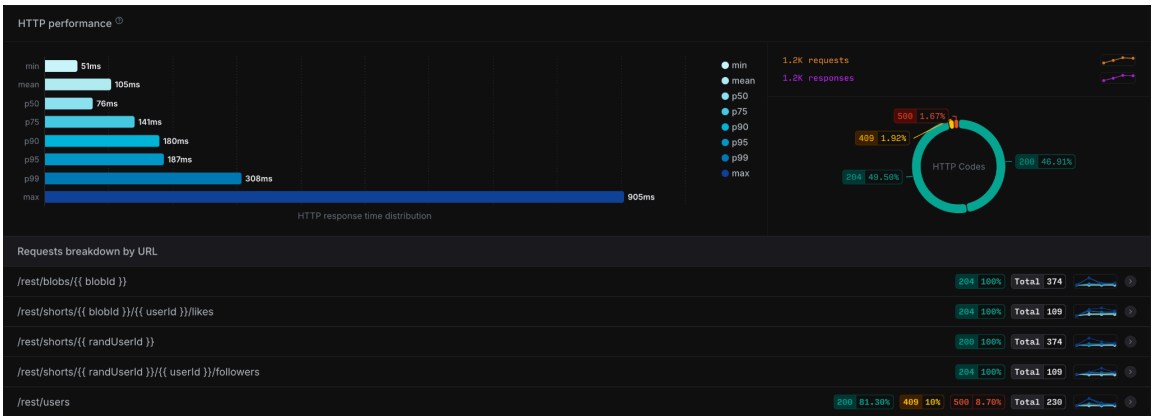

Fig. 1.2 b) — HTTP Performance

PostgreSQL

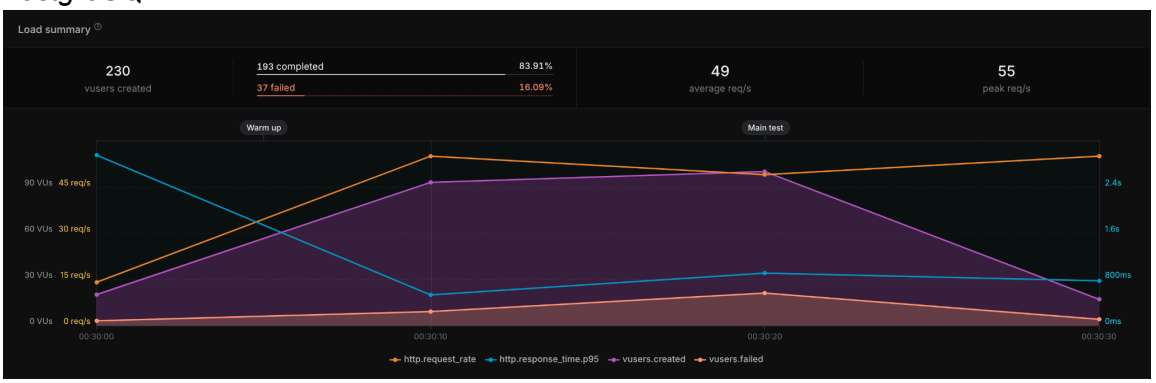

Fig. 1.2 c) — Load Summary

Fig. 1.2 d) — 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.

Just like the previous test scenario it aims to simulate a real-world usage scenario to evaluate how the system performs.

2.1. Without Cache

NoSQL

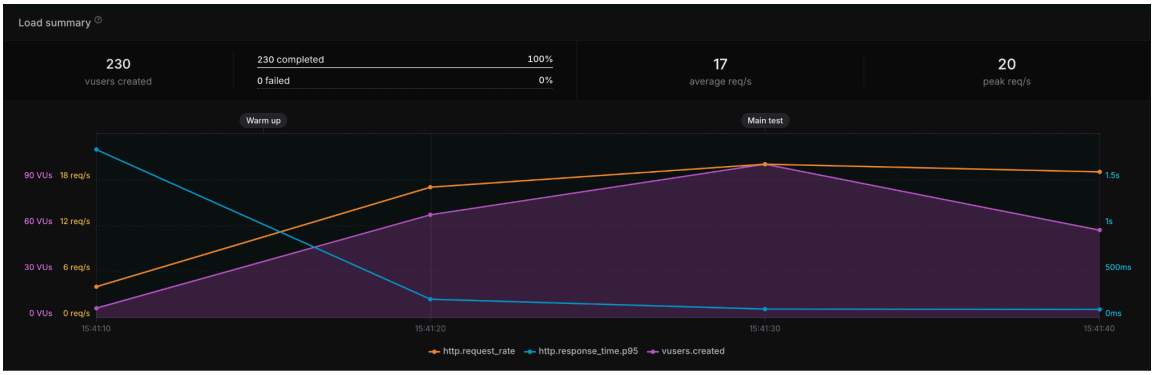

Fig. 2.1 a) — Load Summary

Fig. 2.1 b) — HTTP Performance

PostgreSQL

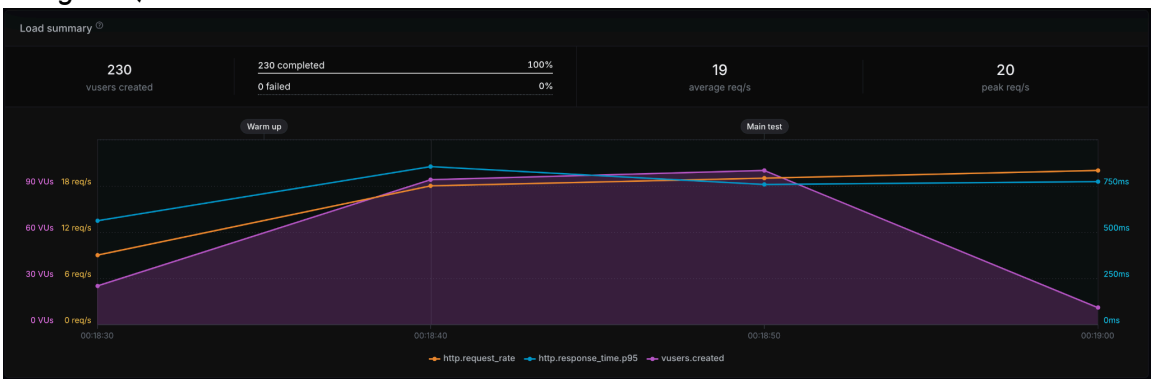

Fig. 2.1 c) — Load Summary

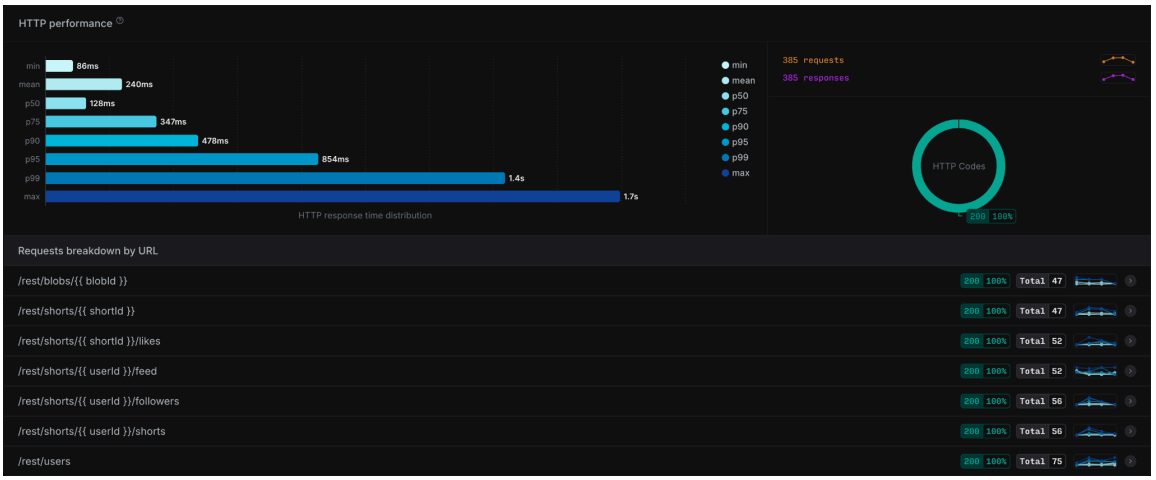

Fig. 2.1 d) — HTTP Performance

2.2. With Cache

NoSQL

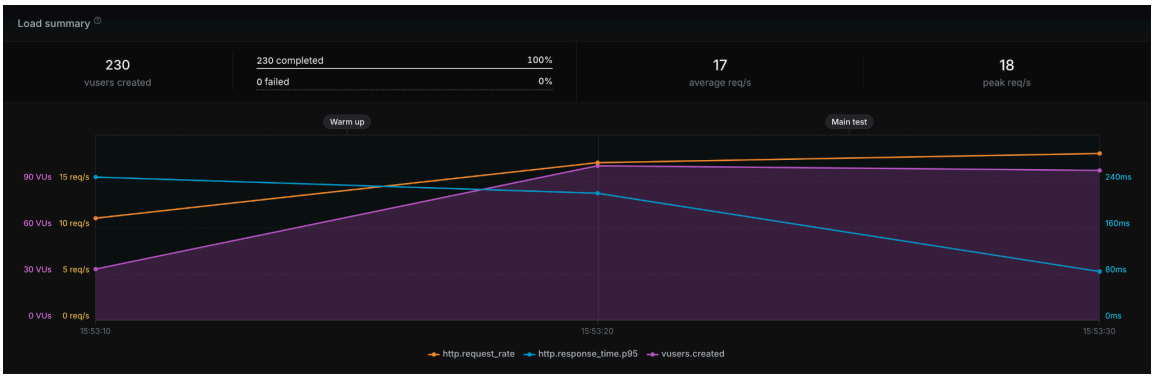

Fig. 2.2 a) — Load Summary

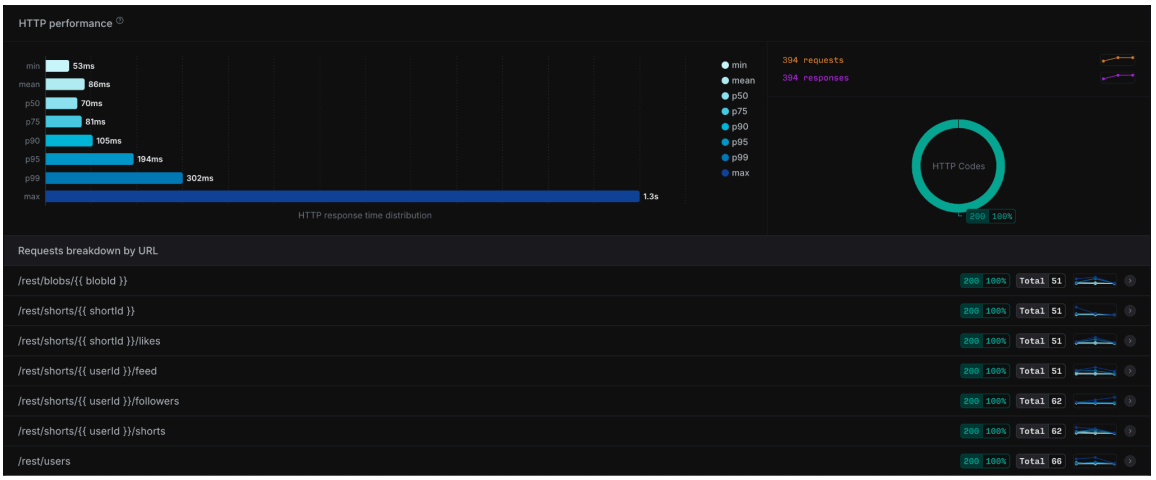

Fig. 2.2 b) — HTTP Performance

PostgreSQL

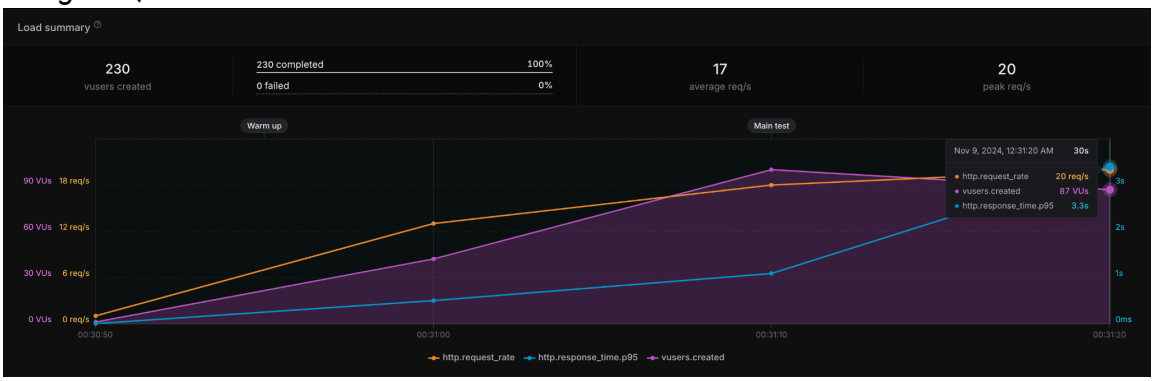

Fig. 2.2 c) — Load Summary

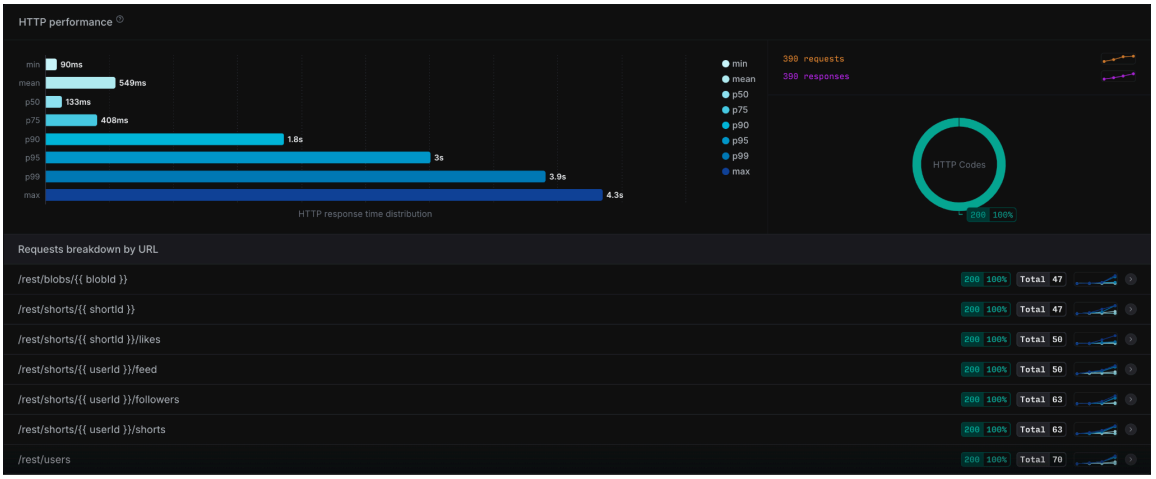

Fig. 2.2 d) — HTTP Performance

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, it skips over testing without Redis because of our assumption that, based on the previous tests, it's always beneficial.
The configuration for this test's phases was a little different, to allow for more requests to come in:

```
target: https://tukano7420.azurewebsites.net/rest
phases:
  - name: Warm up
    rampTo: 5
    duration: 10
    arrivalRate: 1
  - name: Main test
    duration: 30
    arrivalRate: 10
```

This workload simulates real-world usage under a much heavier load than previous tests, to evaluate system performance in a more realistic way.

3.1. With Cache

NoSQL

Fig. 3.1 a) — Load Summary

Fig. 3.1 b) — HTTP Performance

PostgreSQL

Fig. 3.1 c) — Load Summary

Fig. 3.1 d) — HTTP Performance

Conclusion

To conclude, we now have a better understanding of the challenges and benefits of using Cloud providers when developing applications. While the results we obtained align with what we expected when comparing NoSQL vs SQL databases, with and without a Cache layer, this project revealed that the decision between the two isn't as clear-cut as we thought, mostly due to the complexity that arises from non-relational data storage.

While the structure of our code could perhaps be improved, especially in the parts where we make direct calls to CosmosDB or Hibernate from outside of our DB class, we're confident in the performance of our application and in the final submission.

Also worth noting that the graphs presented were generated by Artillery on their Cloud Dashboard, when running the following command: **`artillery run our_test.yaml --record --key our_api_key`**.[2]

This was particularly useful in not only visualizing our test results, but also in corroborating our implementation decisions with more clarity than what we had in mind before discovering the command.

Bibliography

[1] — CosmosPatchOperationsClass | Microsoft Learn.
(n.d.). [Consult. 09 nov. 2024].
Available at:
https://learn.microsoft.com/en-us/java/api/com.azure.cosmos.models.cosmospatchoperations?view=azure-java-stable

[2] —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