

Music streaming architecture: A dot framework research paper  
A Software Engineering Study Project

---

Yardi van Nimwegen  
Complex software semester 6  
Fontys University of applied sciences  
2025-09-19

## Document Version History

Version	Date	Author	Description
1		Yardi van Nimwegen	Initial draft created

# Table of Contents

## Contents

Document Version History .....	2
Table of Contents .....	3
Abstract .....	4
Introduction .....	4
The problem .....	4
Objectives .....	5
Principles of back-end architecture .....	5
Case study: Spotify .....	5
Spotify .....	5
How do servers store music files .....	5
Which Object storage to use .....	6
Does MinIO have better throughput than the Spring application .....	6
.....	8
Bibliography .....	9

## Abstract

This report presents a technical analysis and architectural design shift to address high latency in a walking skeleton music streaming application. The cause of the bad performance has been identified as a fundamental architectural flaw, which will never allow the application to be scalable. The flaw was the use of synchronous

## Introduction

*The problem.* The main challenge for any media streaming application is to deliver the content to the users with low latency and uninterrupted playback without buffering. When this user expectation is not being met, it can lead to a huge loss of user engagement with the platform, since most users will not tolerate more than 90 seconds of poor quality until they abandon your platform[1]. The current implementation is suffering from precisely this issue, as shown in Figure 1



Figure 1: K6 test which shows the application is suffering from latency and bad throughput

This test shown in Figure 1 shows the unacceptable amount of latency which is limiting the current implementation. When investigating this further and running a CPU time profiler on the application, as shown in Figure 2. It became clear that the cause was a problem in the architecture which is not optimized for the I/O-bound nature of streaming large music files.

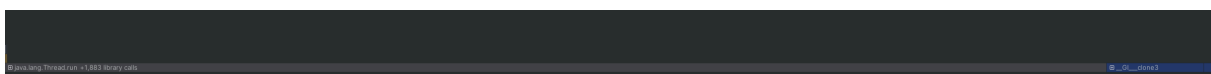


Figure 2: Flamegraph CPU time springboot application at start of the project

The current implementation implements a synchronous, thread per request model. This model selects a thread and keeps this thread dedicated to handle the request from the start to the end. When a request for

a song arrives, the thread gets used to retrieve and eventually serve the song to the user that has requested it. During this process, the thread will not be available for other tasks, when a lot of users are connecting at the same time the threads that are available can quickly be used up[2]. This will increase response times a lot as seen in Figure 1 when looking at the min and max time.

To solve this issue there needs to be a complete shift in architecture, which can pave the way to a highly scalable application.

*Objectives.* This research has been required to address critical architectural issue as described in the problem. The objects of this research are:

- Investigate the cause of the high latency and find a solution
- Propose a new architecture for the application which closely follows cloud best practices and can be run fully local.

This investigation will reach conclusions which are evidence-based and will provide the reader with theoretical concepts to a new architectural design which is future proof and scalable.

## Principles of back-end architecture

The performance of any back-end application is critical and is massively influenced by how it's architecture has been set up, how it handles a large amount of concurrent requests and manages it's resources to optimize performance[2]. The traditional server model, which is currently being used in the Springboot MVC application, operates on a thread per request model[3]. This means that each incoming HTTP request to stream a song receives a dedicated thread that processes the task from start to finish. Streaming music is however is very I/O intensive, since it is fetching and sending the files[4]. During this whole process the thread remains in a blocked state, simply waiting till the I/O operation is finished[5].

Currently the primary cause of the performance weakening when a lot of users are connecting is the system simply just waiting till I/O tasks are finished, as shown in Figure 2. Because of the high amount of users the threads which can be used, quickly run out and can only be assigned to a new user once they have fully completed the request[3]. The solution to solve this issue is to move to a non-blocking async architecture. These non blocking frameworks, instead of waiting for an I/O task to finish, free the thread that was being reserved for another task to execute on this thread. Once the I/O task has been finished the task gets assigned to a thread again, this non-blocking is perfect for achieving high concurrency and low latency which is required for a streaming platform.

*Case study: Spotify.* When looking at the back-end architecture of a streaming giant, you can see that the principles of back-end architecture are not just theoretical but are crucial to have a true scalable platform[6],[7].

*Spotify.* Spotify's back-end is built upon a microservices architecture, with most of their services written in Java[7]. For streaming the music Spotify doesn't stream it through any Java application like was done for the walking skeleton application. Instead Spotify utilizes content delivery networks[8], these servers connect to the closest location where a song is stored on a server. And the actual application logic such as managing playlists and the recommendation of music, is done by a Java service[7].

*How do servers store music files.* Instead of using an actual file system like computers meant for end-users which are used for daily usage such as browsing the web and typing some documents, cloud providers use object storage.

Object storage is perfect for storing a large amount of files which can be of varying size. This is due to that object storage stores every file as a self-contained object rather than within a folder structure. This makes it much easier to handle the management and retrieval of the files, since every object has it's own unique key, it also becomes possible to have songs with the same title since they get a unique key so they are still easily identifiable[9].

Storing files like objects is designed for huge scalability with no limits on how much storage you can get. This allow to just keep adding files as the platform grows. Furthermore, it is also very cost effective

for a cloud service, since you only pay for the storage that you use and makes it much easier to scale compared to normal file storage[9]. This is perfect for storing an ever growing music library.

To interact with object storage there is an API which has become the standard for cloud and local object storage, this is the S3 API[10], [11].

### Which Object storage to use

Since development of this project is mainly focussed on local development first, this is due to this project being a student project with no school provided cloud provider account. This means that the object storage should be self-hostable and also adhere to the S3 API standards, this would make it easier to eventually migrate to the cloud at the end of the project.

With these specific requirements in mind i made the choice to use MinIO[10]. Which is completely self-hostable through any containerization framework[12]. Minio also makes it easy to transition to a cloud provider[13],[14],[15]. Because it is self-hostable while in development, so there are no costs which will deplete the free credits cloud providers provide the user. And the fact that it would be possible to transition from MinIO to a cloud provider if needed, makes it a perfect consideration to use as the object storage for this music streaming platform project.

*Does MinIO have better throughput than the Spring application.* After setting up MinIO to run in a local docker environment and stress testing this software for music streaming, it became very clear that this is a huge improvement over using the Springboot framework to stream music files. This software has a much higher throughput and is being limited by the hardware which it is run on an would have much greater performance on a server.

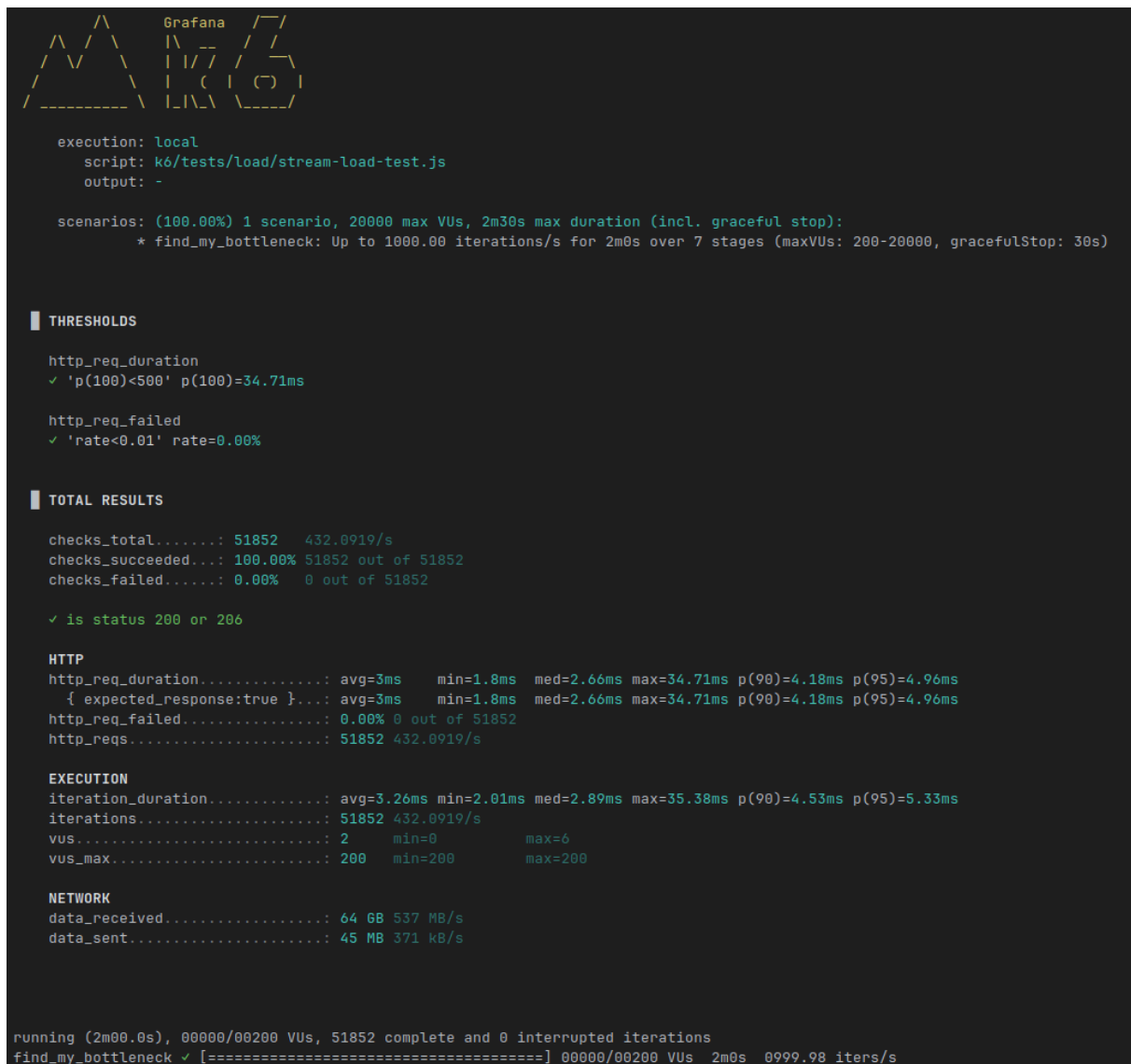
It is very clear that MinIO is the right choice as a storage system for music, it has great throughput and can handle an extremely high amount of concurrent connections and it severely being limited on my localhost hardware. To find the limit of the localhost system there were multiple stress tests executed which give insight into the bottleneck, and it is most certainly not the software that is being used.

The first test that was executed was a stress test to prove if MinIO was the right choice in making the software architecture of this project more scalable, the results of this test can be seen in Figure 3.

From the results in Figure 3 it already becomes clear that MinIO is a much better solution when comparing the results to the old results as seen in Figure 1

Version	AVG	Min	Med	Max	p(90)	p(95)
Spring mvc	1.11s	3.42ms	666.16ms	8.12s	2.69s	3.3s
MinIO	3ms	1.8ms	2.66ms	34.71ms	4.18ms	4.96ms

The average response time has increased by 370 times by switching to using an S3 compatible bucket.



```

execution: local
  script: k6/tests/load/stream-load-test.js
  output: -

scenarios: (100.00%) 1 scenario, 20000 max VUs, 2m30s max duration (incl. graceful stop):
  * find_my_bottleneck: Up to 1000.00 iterations/s for 2m0s over 7 stages (maxVUs: 200-20000, gracefulStop: 30s)

THRESHOLDS

http_req_duration
✓ 'p(100)<500' p(100)=34.71ms

http_req_failed
✓ 'rate<0.01' rate=0.00%

TOTAL RESULTS

checks_total.....: 51852 432.0919/s
checks_succeeded...: 100.00% 51852 out of 51852
checks_failed.....: 0.00% 0 out of 51852

✓ is status 200 or 206

HTTP
http_req_duration.....: avg=3ms min=1.8ms med=2.66ms max=34.71ms p(90)=4.18ms p(95)=4.96ms
  { expected_response:true }...: avg=3ms min=1.8ms med=2.66ms max=34.71ms p(90)=4.18ms p(95)=4.96ms
http_req_failed.....: 0.00% 0 out of 51852
http_reqs.....: 51852 432.0919/s

EXECUTION
iteration_duration.....: avg=3.26ms min=2.01ms med=2.89ms max=35.38ms p(90)=4.53ms p(95)=5.33ms
iterations.....: 51852 432.0919/s
vus.....: 2 min=0 max=6
vus_max.....: 200 min=200 max=200

NETWORK
data_received.....: 64 GB 537 MB/s
data_sent.....: 45 MB 371 kB/s

running (2m00.0s), 00000/00200 VUs, 51852 complete and 0 interrupted iterations
find_my_bottleneck ✓ [=====] 00000/00200 VUs 2m0s 0999.98 iters/s

```

Figure 3: Stress test MinIO with 1000 users concurrently

The performance improvement is gigantic and is currently being limited by the hardware it is being ran on. As you can see in Figure 3 when looking at the p(95) and then looking at the max, signs of a bottleneck existing are starting to show.

After some rigorous stress testing it became clear that the issue is being caused by the CPU of the localhost where MinIO is being ran for testing. Especially when stress-testing with above the 1000 users the CPU of the host would start running at very high percentages and would eventually top out of 100% and then performance declined very hard as seen in Figure 4, and the system resources monitor can be seen in Figure 5

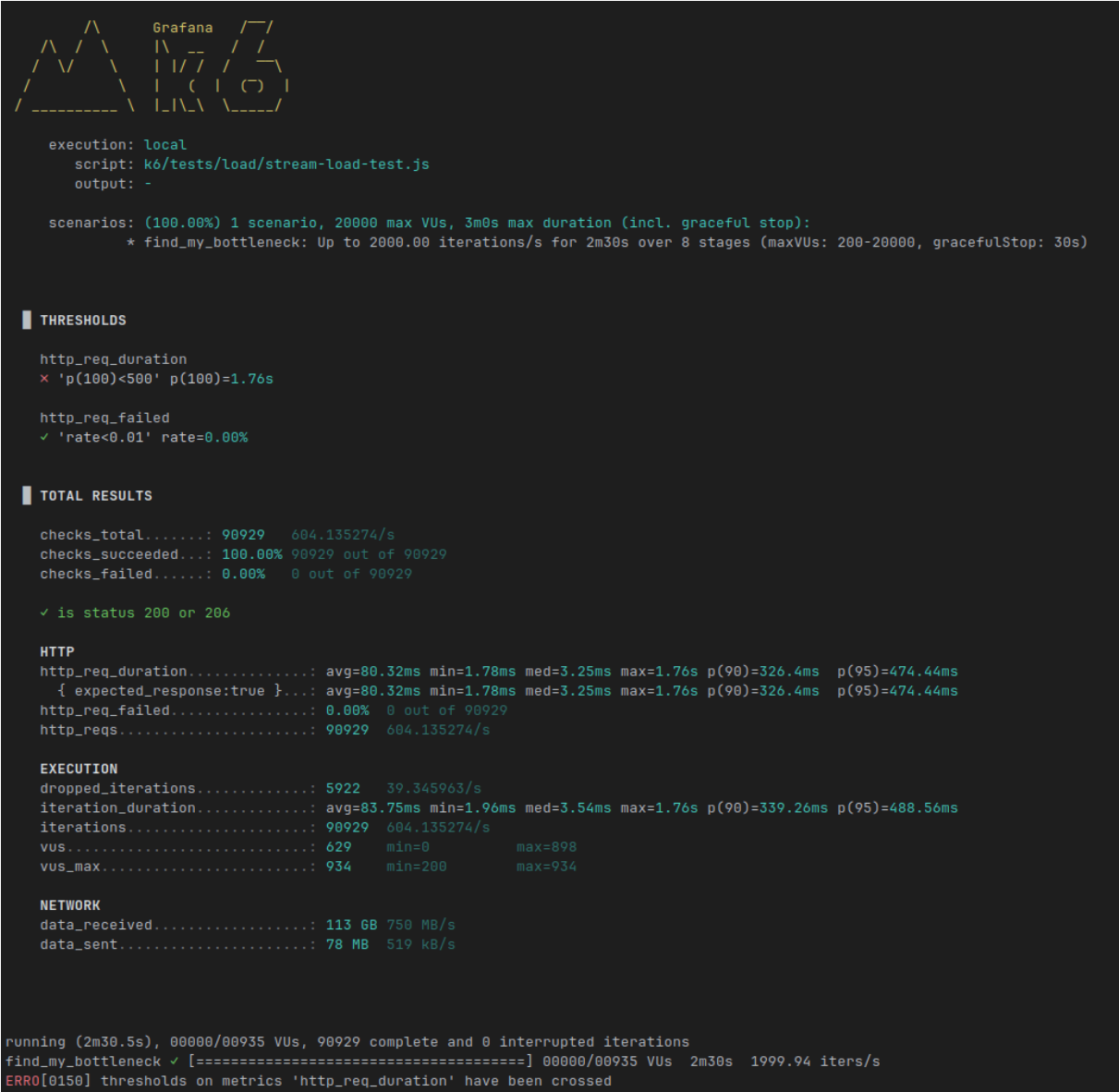


Figure 4: Stress test MinIO with 2000 users concurrently, limited by hardware

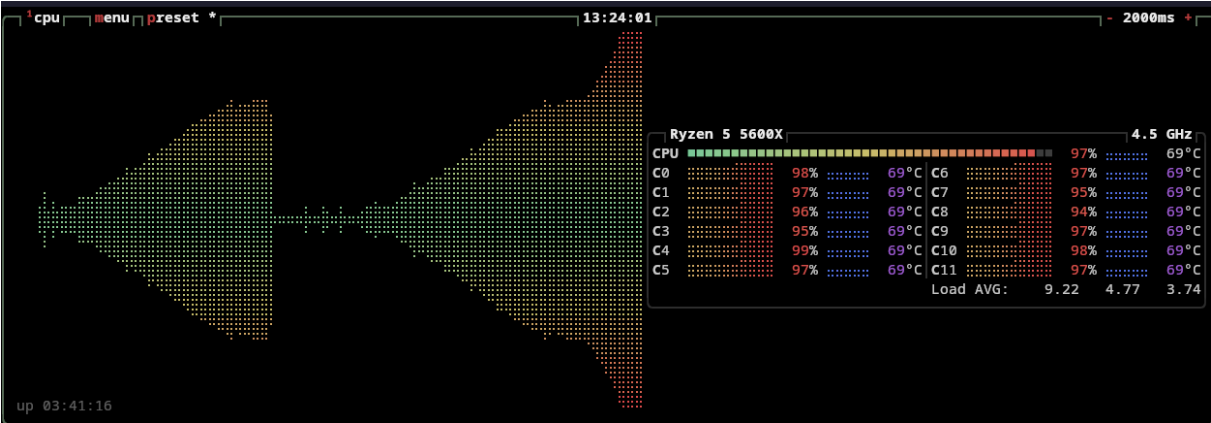


Figure 5: System running into performance issues which makes the true limit unknown



## Bibliography

- [1] “Video streaming system design: key principles for building a reliable streaming application architecture | RST Software.” Accessed: Sep. 18, 2025. [Online]. Available: <https://www.rst.software/blog/video-streaming-system-design-key-principles-for-building-a-reliable-streaming-application-architecture>
- [2] “Spring Boot Performance Tuning: 5 Common Issues and How to Fix Them.” Accessed: Sep. 18, 2025. [Online]. Available: <https://www.cogentuniversity.com/post/spring-boot-performance-tuning-5-common-issues-and-how-to-fix-them>
- [3] “Spring MVC vs WebFlux: When to Use Which Framework?.” Accessed: Sep. 19, 2025. [Online]. Available: <https://www.geeksforgeeks.org/blogs/spring-mvc-vs-spring-web-flux/>
- [4] A. Smith, “Is your backend CPU-intensive or I/O intensive?.” Accessed: Sep. 19, 2025. [Online]. Available: <https://codewithflash.com/is-your-backend-cpu-intensive-or-io-intensive>
- [5] “Spring MVC vs WebFlux: When to Use Which Framework?.” Accessed: Sep. 19, 2025. [Online]. Available: <https://www.geeksforgeeks.org/blogs/spring-mvc-vs-spring-web-flux/>
- [6] “System Design Netflix | A Complete Architecture.” Accessed: Sep. 19, 2025. [Online]. Available: <https://www.geeksforgeeks.org/system-design/system-design-netflix-a-complete-architecture/>
- [7] “Discuss Spotify system architecture..” Accessed: Sep. 19, 2025. [Online]. Available: <https://www.designgurus.io/answers/detail/discuss-spotify-system-architecture>
- [8] G. Andersen, “Exploring Back-End Development in the Music and Media Streaming Industry.” Accessed: Sep. 19, 2025. [Online]. Available: <https://moldstud.com/articles/p-exploring-back-end-development-in-the-music-and-media-streaming-industry>
- [9] “How Object vs Block vs File Storage differ.” Accessed: Sep. 19, 2025. [Online]. Available: <https://cloud.google.com/discover/object-vs-block-vs-file-storage>
- [10] “AWS S3 Compatible Object Storage | MinIO.” Accessed: Sep. 19, 2025. [Online]. Available: <https://www.min.io/product/aistor/s3-compatibility>
- [11] “S3 API Reference - Amazon Simple Storage Service.” Accessed: Sep. 19, 2025. [Online]. Available: [https://docs.aws.amazon.com/AmazonS3/latest/API/Type\\_API\\_Reference.html](https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html)
- [12] Accessed: Sep. 19, 2025. [Online]. Available: <https://hub.docker.com/r/minio/minio>
- [13] “Transition Objects from MinIO to S3 — MinIO Object Storage (AGPLv3).” Accessed: Sep. 19, 2025. [Online]. Available: <https://docs.min.io/community/minio-object-store/administration/object-management/transition-objects-to-s3.html>
- [14] “Transition Objects from MinIO to GCS — MinIO Object Storage (AGPLv3).” Accessed: Sep. 19, 2025. [Online]. Available: <https://docs.min.io/community/minio-object-store/administration/object-management/transition-objects-to-gcs.html>
- [15] “Transition Objects from MinIO to Azure — MinIO Object Storage (AGPLv3).” Accessed: Sep. 19, 2025. [Online]. Available: <https://docs.min.io/community/minio-object-store/administration/object-management/transition-objects-to-azure.html>