National Institute of Applied Sciences and Technology

UNIVERSITY OF CARTHAGE

Professional Personal Project

Major : **SOFTWARE ENGINEERING (GL)**

A Live Streaming Platform.
An Implementation of a microservices-based architecture

Realized by

Zeineb BENABDALLAH
Raed ADDALA
Mohamed Hedi BEN ALI

Presented on : **04-06-2024**

SUPERVISOR

Mme Hajer TAKTAK

Academic Year: 2023-2024

Contents

General Introduction

Multimedia Processing was introduced and started during the 1970s, evolving into the complex algorithms and efficient dedicated hardware accelerators of today, providing the foundation of internet real-time communication from person-to-person calls like Skype to Online Meetings such as Zoom and Google Meet, and Live Streaming like in Twitch.
Live streaming started in the early 1990s when the first media players capable of live streaming were developed. This new technology allowed computer users to watch audio and video content in real-time, changing how we consume entertainment. With the emergence of new technologies, breakthroughs in networking capabilities and the formulation and improvements in Multimedia Protocols and Codecs made Live Streaming find the widespread usage it has today. This has created a huge new market with a huge room for potential.
The massive needs and special requirements of such a system made each approach to engineering it unique and different to any other. Hence, there is no right answer in the architecture of streaming services and it highly depends on the exact needs of the target users, as well as other constraints and the preferences of the engineers. There is room for experimenting with the architecture and coming up with our own.

Chapter I

General Project Context

Contents

1 Project Context

This project is our Professional Personal Project (PPP) for our 3^{rd} year of software engineering. Our work was supervised by Mrs Taktak Hajer. This report details the design, and implementation of our live-streaming platform. It covers the technical architecture, development process, and key considerations. Additionally, we discuss the future and potential development of this project.

2 Introduction

Live Streaming has become an essential medium for communication, entertainment, and education in the current digital era, allowing for real-time interaction and information delivery worldwide. It has gained a lot of attraction in the last few years, especially with the coronavirus crisis. Live streaming services are utilised for a variety of reasons and target groups, including corporate events, social media influencers, and online education. The goal of this project is to create a scalable and reliable live-streaming platform that will satisfy the needs of modern consumers while maintaining excellent user experience, security, and performance.

3 Methodology

In the development of our project, we followed a linear Project management Technique since our main needs are fixed, the scope is well-defined and the project has dependencies between phases and tasks.
This methodology also called the waterfall technique consists of these phases:

1. **Requirements:** The requirements phase states what the system should do. At this stage, you determine the project's scope, from business obligations to user needs. After market research and target analysis, the team must have documented functional and non-functional requirements formally and precisely.

2. **Software Design and Architecture:** After a careful requirements analysis and reformulation, the system architecture must be discussed and determined formally and with careful precision. A careful examination of what aligns perfectly with the project without overengineering or undermining the obligation for a correct, extensible and easily maintainable structure. Through this phase, Solutions that meet the requirements must be designed and developed: the application architecture is to be defined, as well as the exact deliverables and their corresponding design and diagrams.

3. **Implementation:** When the architecture is fully defined, revised and approved, then each team can start on their part and implement it. This is the most practical part of the project and can lead to some minor changes in design when some complexities arise or if better solutions are found while working on the implementation. However, these changes must stay as minimal as possible.

4. **Testing:** In this phase, QA (quality insurance) write tests for the product and documents any bugs or errors to be fixed by covering varieties of use cases and environments. Implementation and Testing go hand in hand in a cycle until reaching an MVP (minimum viable product) that can be deployed and distributed.

5. **Deployment and Maintenance:** The final phase of the project where the finalized solution is made accessible for clients to use in production. Commonly, this phase involves planning the deployment, monitoring and maintenance of the solution, producing a final report to summarize the project, and conducting a project retrospective as a final step. There is also the production infrastructure maintenance and site reliability that must be taken into consideration.

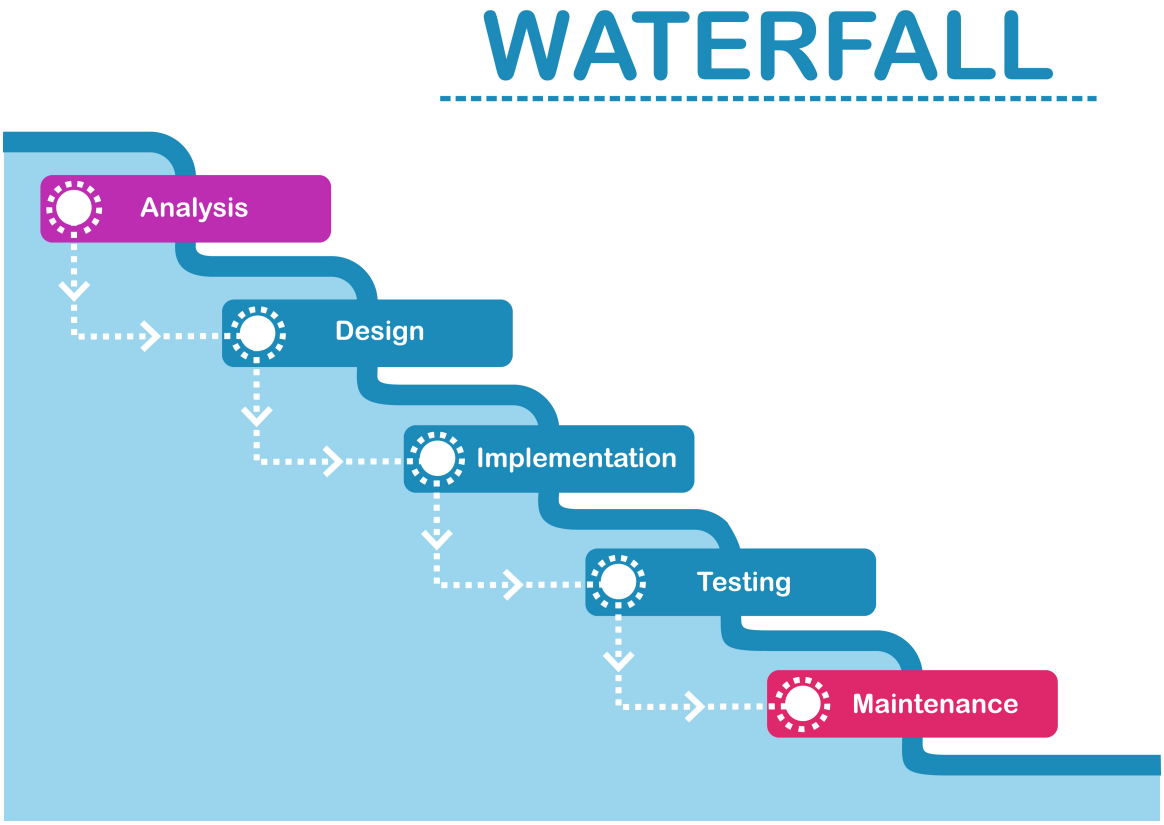

Figure I.1: Waterfall's Process Diagram

Chapter II

Specification of requirements

Contents

1 Functional Requirements

1.1 Identification of actors

Identifying different actors in a system is a crucial step in a product's development process and in identifying requirements for our system. For our live-streaming platform, we identify the following actors and their relationships:

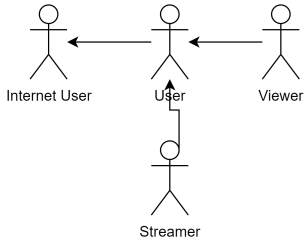

Figure II.1: Actors Relationship Diagram

1.2 Specification of functional requirements

In this phase, we will specify the function requirements of our system and the present use cases related to each actor.

Internet User: This Actor represents any internet user that visits our site independent of him being a user of our platform (has an account) or not.

This actor has the following use cases:

- join and watch a live stream of a user.
- make an account.
- consult profiles of existing users.

User: This Actor represents a user of our platform with an account. In addition to the use cases of an internet user, our users have these use cases:

- participate (ie: receive and send messages) in a chat associated with a streamer unrelated to his live stream.
- edit and personalize his profile.
- start a live stream.
- end a live stream.

2 Non-Functional Requirements

Non-functional requirements (NFRs) define the system's operation rather than its behaviours. They are crucial for ensuring the quality, performance, and usability of the system.
Here are some non-functional requirements relevant to our project:

Performance:

- Latency: The platform should maintain low latency for live streams to ensure real-time interaction.
- Throughput: The system should handle a high number of concurrent viewers and streamers without degradation in performance.
- Load Handling: The system should support peak load conditions, such as major events, with minimal performance loss.

Scalability:

- Realtime systems are renowned for their huge loads on the servers and the network. Hence, Horizontal Scalability is a must to ensure a highly available and smooth system for clients. This is crucial especially because of how computationally heavy multimedia processing is. In the case of Twitch, clusters are deployed for processing only. So it is clear how crucial it is for the success of the business.

Security and Privacy:

- Security and Privacy are a huge concern for both users and business owners and must be our top priority. In the case of Privacy, our policy is to hash sensitive data in the database such as passwords. Also, exchange data over the network through a protected medium using public and private encryption keys.
  Implementing robust authentication and role-based authorization through fresh tokens mechanisms to prevent unauthorised access and protect user data as well as going for a microservices architecture to minimize attack region, are strategies used in our Security Policy. Further policies can be added if proven necessary.

Maintainability:

- The Code should follow the SOLID principles and the design patterns, especially the Open/Closed Principle meaning that each component is open to extension and closed to modification. Good Software Design ensures easier extension in the future, facilitates debugging and solving issues as well as maintaining it. (The Code becomes self-documenting).
- The software should be well-documented and follow best practices for readability and modularity.
- Versioning must be taken seriously and the dependencies of each version must be well defined in the documentation.
- The system should support easy updates and patching without significant downtime. It would be perfect if the system supports hot code reloading in production so changes are shipped on the fly.

Compatibility:

- Keeping up with software updates while being backwards-compatible and supporting older platforms to ensure a good reach.
- The platform should be compatible with major web browsers and mobile devices.

Chapter III

System Design

Contents

1 System Architecture

1.1 Architecture Choice:

Looking at Well-known live streaming platforms such as Twitch, we found a certain pattern that made us ditch the idea of creating a monolith in favour of Microservices. But before commenting on this, let's first define what we mean by Microservices and Monoliths.

- **Monolith**: a single build of unified code. Every service and feature in the code is built into one large system. They usually share the same server, file system, and database. They all share the same resources.
- **Microservices**: smaller modules of code, each with its function, focus and resources. Every module is a separate application that usually lives on its server, communicating with each other through APIs.

A case study of monoliths Monoliths are tempting for new businesses. They are easier to start programming with and they are the perfect option if you want to build your product fast and ship fast. The infrastructure needed is nearly just configuring an HTTP Server like NGinx, a proxy for caching and improving performance like using HAProxy, as well as a database and that's nearly all. All resources are shared by all services so we don't have to think about resource allocation in our infrastructure.
This is exactly our problem: all services share the same resources. From the early stages, it is clear that some services will do most of the heavy lifting in the server and will be the performance bottlenecks that will hinder everything else. The solution would be to duplicate the system into two for example: just buy a new computer, create a new instance of

the server and put it in the new computer, now using a round-robin algorithm, channel each request to a corresponding server. This is so wasteful because the real problematic services need more computing power, more CPU time and more memory yet they have to share it with the other services each time. This is where we started to think about separating our main service into isolated microservices linked together through the API gateway working as a reverse proxy.
Most live streaming services started as Monoliths, where all functionalities are accessed through the same API and node. In the case of Twitch, it was a Ruby on Rails Monolith supporting all features including live streaming, live chat, an API for bots, 3rd party tools, etc, data analysis tools, recommender systems and more. Live streaming is itself a computationally heavy service (this will be discussed in detail later) and alongside the chatting system, they both represent a heavy load both on the servers and the network itself.
Hence, the Twitch team found itself with a Performance Bottleneck with a massive amount of concurrent views and a diverse audience that connects from different regions of the world even with a caching layer, the system reached a limit. The system can no longer be scaled vertically and has to be scaled horizontally. Hence, going for a microservices architecture to solve this problem.

Microservices Pros In a nutshell, we have chosen Microservices for these reasons:

- Separation of responsibilities and the decoupling of components.
- Making the system more fault-tolerant and resilient to failures and errors.
- Effective allocation of resources depending on the exact needs of each service.
- Handling heavy loads by load balancing, and repartitioning the server into different geographic locations.
- Simpler components: simpler to develop, maintain and extend.

Microservices Cons However, there is one drawback to this approach:

- Too much time is needed for setting up the infrastructure.
- The system is harder to design especially if the isolation is not evident.
- Even though maintaining microservices individually is relatively easier, maintaining the whole infrastructure is hard and costly.

With all these considerations, microservices are a must.

1.2 Overall Architecture

In Neon, we followed a microservices architecture. Our system is divided into 3 microservices:

- Authentication Service
- Live Chat Service
- Live Streaming Service

Each of these services is containerized using Docker and exposed through an API Gateway built on top of Kong.
In case of high demand for one of these services, we can duplicate one of these services and configure it in Kong so it now balances loads between different nodes of the same service.
This property of our microservices ensures horizontal scalability and fault tolerance by duplication.

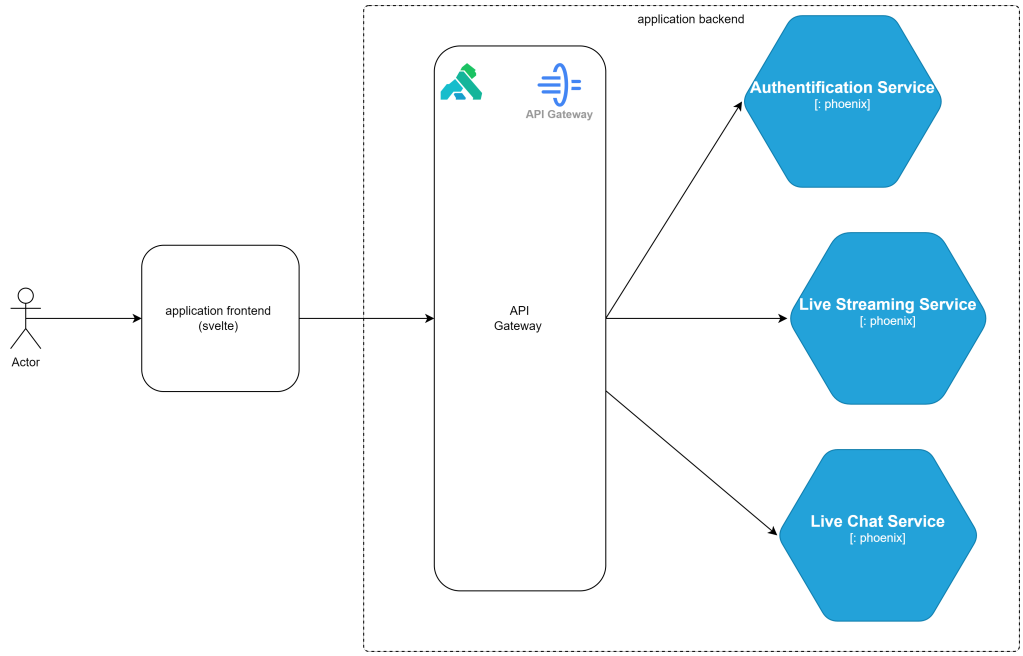

Figure III.1: System Architecture of Neon

2 Architecture and Service Logic

In this section, we will explore the logic behind each element of our live-streaming platform's architecture. Our platform is designed using a microservices architecture to ensure modularity, scalability, and maintainability. Each service is dedicated to a specific set of tasks, allowing for clear separation of concerns and efficient management of resources. Below, we will delve into the details of each service, outlining their roles, interactions, and the technologies used to implement them.

2.1 Live Streaming Service

2.1.1 Service Requirements and Needs

We are building a Live Streaming Service. Scalability, Low-Latency, Fault-Tolerance, Load-Tolerance, and Consistent Stream are a must.

So our system must:

- receive streams in high resolution without data loss and in low latency.
- perform real-time processing.
- provide high throughput for worldwide distribution at scale.

Meaning, it should be offering an end-to-end low-latency video experience.

2.1.2 Live Streaming Service Architecture Overview

These are the main components of the System:

- Video Ingestion:
  - First Mile Delivery on top of RTMP. Streamers must stream in high definition and high quality. No information must be lost. The protocol used must be used on top of TCP. RTMP is used because it is a TCP Protocol that is known for its wide support as well as its low latency.
- Transcoding:

  - Transcodes incoming high-definition video to different resolutions and protocols. (Compute-Intensive).
  - These are the chosen codecs:
    * AAC (Advanced Audio Codec).
    * H.264 (MPEG-4 AVC (Advanced Video Codec)).

- Distribution and Caching.
- Video on Demand:
  - Video transferred to clients through adaptative bitrate protocols:
    * HLS
    * MPEG-DASH

2.1.3 Streaming Service System Design

For the current implementation which is simplified, we have only one pipeline of data (video chunks) with one operator, the transcoder, else we have the source which is the RTMP ingestion and the sink which is the HLS distribution.

Our System will be then divided into 2 big parts:

- Source Bin: Video Ingestion + Transcoding. (RTMP).
- Sink Bin: Distribution and Caching + Video on Demand. (HLS).

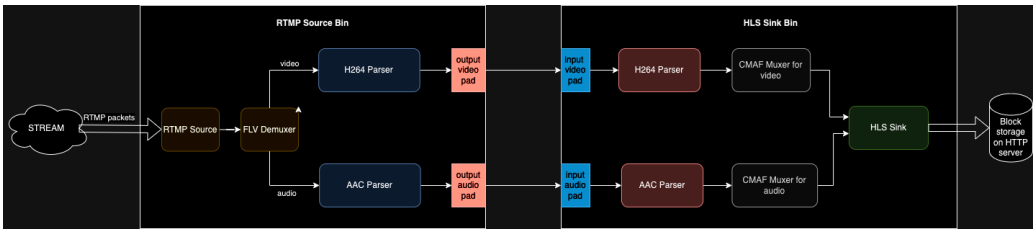

Figure III.2: System Design

2.1.4 Live Streaming Architecture Graph

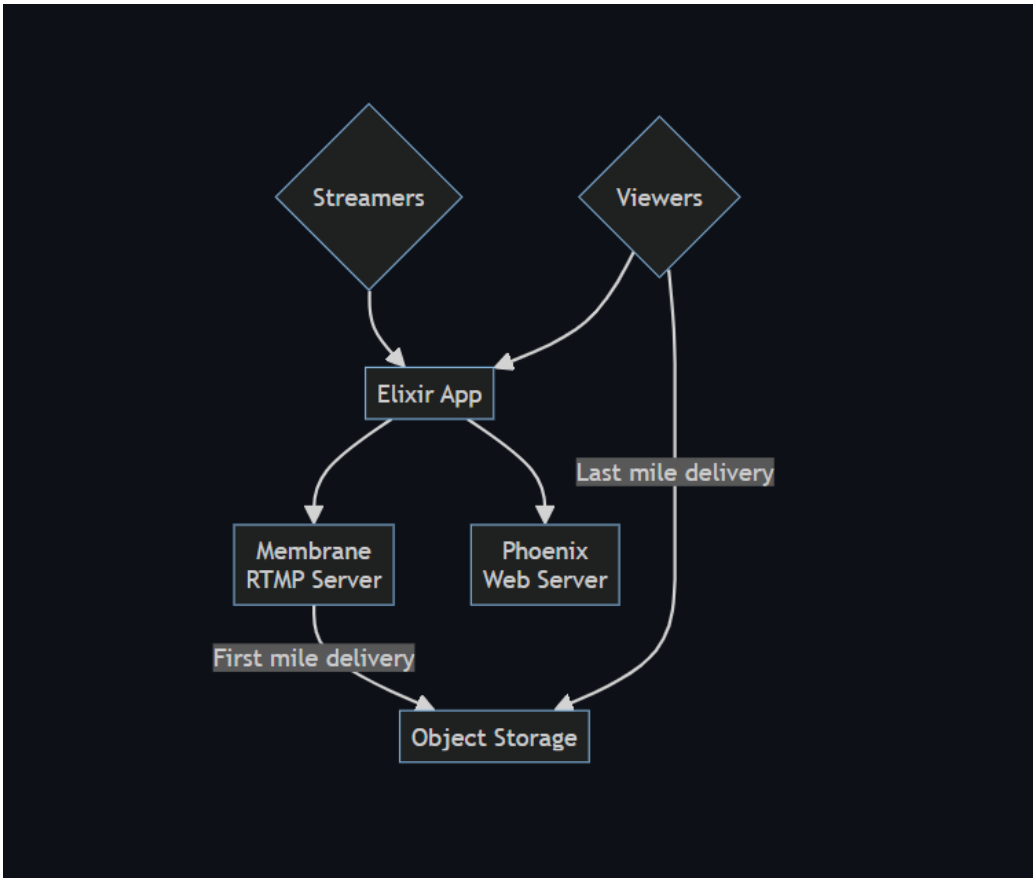

Figure III.3: Live Streaming Architecture Graph

2.1.5 Live Streaming Pipelines Graphes

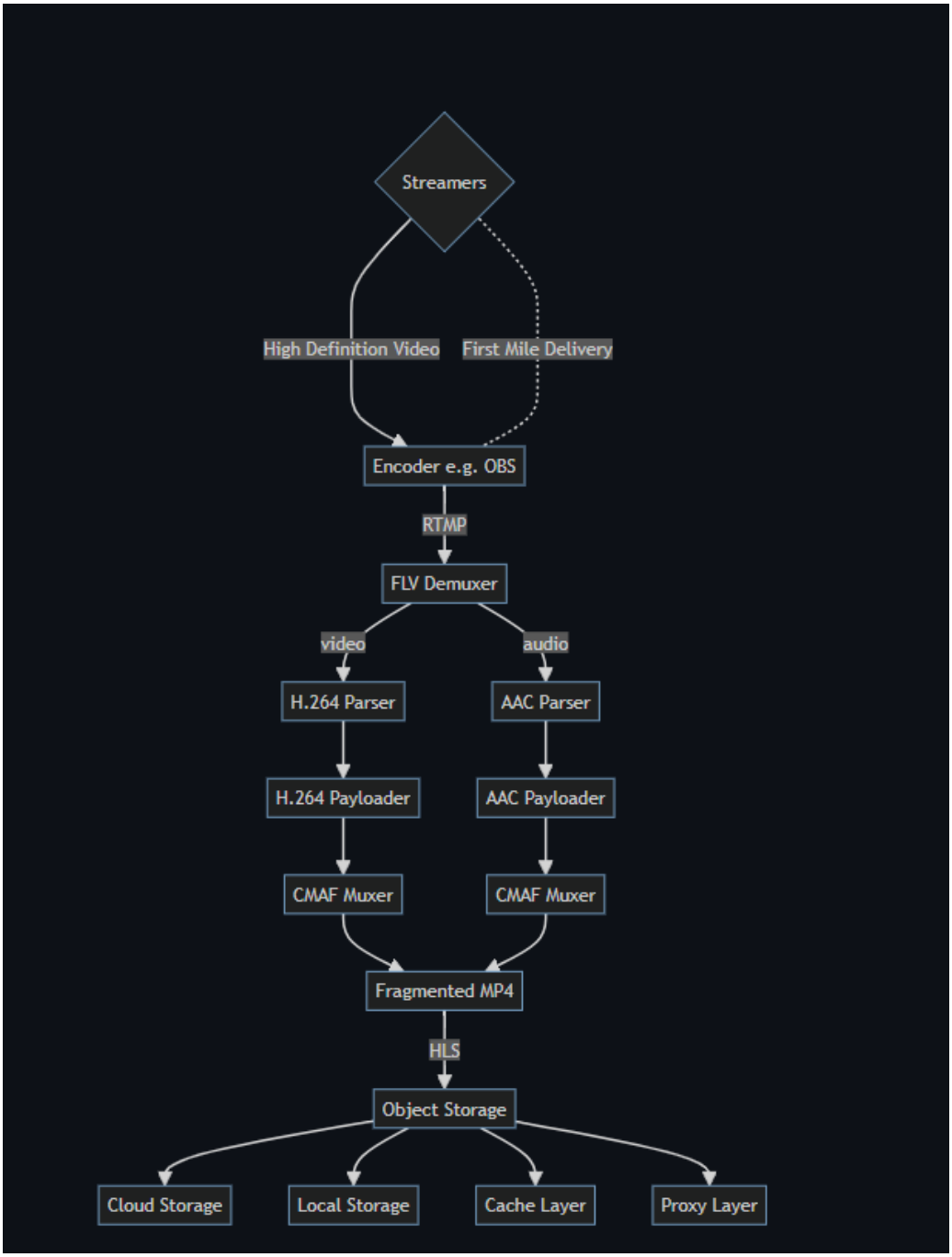

Figure III.4: First Mile Delivery Pipeline Graph

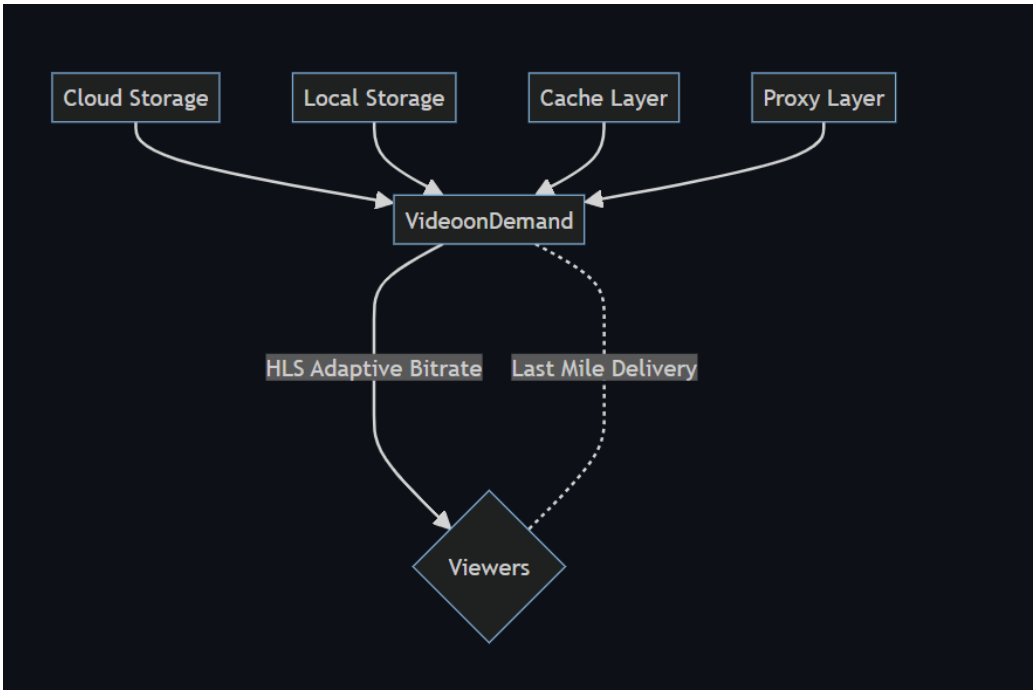

Figure III.5: Last Mile Delivery Pipeline Graph

2.2 Live Chat Service

The live chat plays a crucial role in a live streaming platform since it connects the streamer to his audience. Hearing the viewers' feedback in real-time allows the streamer to adapt to the situation to strengthen the connection between the two; hence improving the audience's engagement.

2.2.1 Communications Protocol: WebSocket

The real-time and bidirectional aspects of the chat dictated we use WebSocket protocol for our implementation for the advantages it provides mainly:

- **Bidirectional**: WebSocket data frames can be sent back and forth between the client and the server in full-duplex mode.
- **Low overhead**: the only HTTP request sent by the client is their initial handshake.
- **Low latency**: since the connection is always open, there's no delay in establishing a connection, leading to faster data transmission.

2.2.2 Workflow

2.2.2.1 Handshake: To establish a WebSocket connection, the client and server upgrade from the HTTP protocol to the WebSocket protocol during their initial handshake, as shown in the following example:-

```
GET /text HTTP/1.1
Upgrade: WebSocket
Connection: Upgrade
Host: www.websocket.org

HTTP/1.1 101 WebSocket Protocol Handshake
Upgrade: WebSocket
Connection: Upgrade
...
```

For authorization, the client sends his credentials in the query parameters of the handshake. The credentials are sent in the query parameters because the WebSocket browser API doesn't allow setting arbitrary headers with the HTTP handshake like Authorization. These credentials allow the server to determine if the current user is allowed to send messages in the channel or not. The server assigns the data acquired from the credentials to the socket connection.

2.2.2.2 Joining a channel: The client requests to join a channel with a "join" event. The channel is named using the username of the associated stream. The server responds with an "ok" event indication of success in adding the user to the requested channel.

2.2.2.3 Sending a message: The client sends his message to the server with an event of "shout". the server adds new data to the message like the sender and the current date then broadcasts the message to clients connected to the current channel.

2.3 Authentication Service

In this subsection, we will explore technologies used in our authentication service and the logic behind its implementation and our choices. But first, let's start with exploring basic concepts.

2.3.1 Basic Concepts

Authentication and authorization are fundamental concepts for managing interactions within our system.

Authentication. . In security terms, authentication verifies the identity of a party. Typically, we authenticate a user by requiring their username and password, assuming only the legitimate user knows this information. More advanced methods include biometric verification, such as using fingerprints or facial recognition on phones. In discussions about authentication, the entity being verified is often called the principal.

Authorization. Authorization, on the other hand, determines what actions a principal is permitted to perform. After a principal is authenticated, we receive information about them that helps us decide their access rights. For instance, knowing the department or office they belong to can guide our system in defining their permissions.

JWT (JSON Web Token). JWT is an open standard used to share security information between two parties — a client and a server. Each JWT contains encoded JSON objects, including a set of claims. JWTs are signed using a cryptographic algorithm to ensure that the claims cannot be altered after the token is issued. A JWT is a string made up of three parts, separated by dots (.), and serialized using base64. Once decoded, you will get two JSON strings:

- The header and the payload.
- The signature.

The JOSE (JSON Object Signing and Encryption) header contains the type of token — JWT in this case — and the signing algorithm.

The payload contains the claims. This is displayed as a JSON string, usually containing no more than a dozen fields to keep the JWT compact. This information is typically used by the server to verify that the user has permission to perform the action they are requesting.

There are no mandatory claims for a JWT, but overlaying standards may make claims mandatory. For example, when using JWT as a bearer access token under OAuth2.0, iss, sub, aud, and exp must be present. some are more common than others.

The signature ensures that the token hasn't been altered. The party that creates the JWT signs the header and payload with a secret that is known to both the issuer and receiver or with a private key known only to the sender. When the token is used, the receiving party verifies that the header and payload match the signature.

2.3.2 Authentication in a microservices architecture

In a monolith, Authentication and authorization are simpler modules to implement since our app is divided into modules which are deployed in the same place and are wrapped in a global module. But in a microservices architecture, things become trickier since each service is deployed separately and ease of use is crucial—we aim to streamline access to our system. Users should not need to log in separately for each microservice with different credentials. We looked at different architectures for authentication in microservices:

2.3.2.1 Option 1: Centralized Identity Provider: The authentication service (identity provider) manages all authentication related tasks, including login, registration, role management, and token validation. As a result, every request that requires role verification or authentication is routed through the authentication service to validate credentials and verify the token's validity before the request can be processed. This can lead to numerous unnecessary and frequent requests to the authentication service.

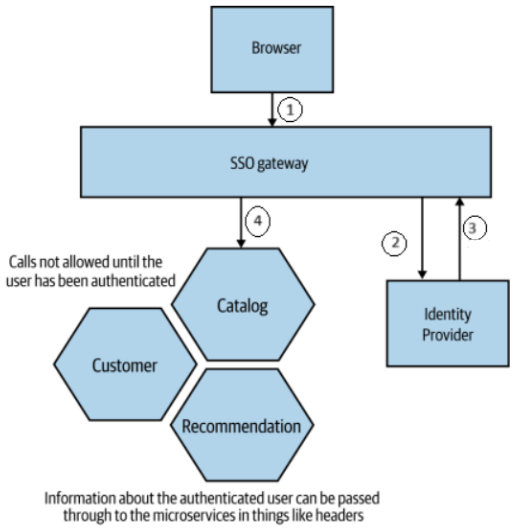

Figure III.6: Flow of a Request

2.3.2.2 Option 2: No Dedicated Identity Provider: There is no dedicated service to handle authentification. In the absence of a dedicated authentication service or identity provider, the API gateway assumes the responsibilities of handling identities, user management, and token creation and validation. This approach compromises the separation of concerns and introduces potential security risks. If an attacker breaches the gateway, they can gain unrestricted access to all microservices. Relying on the API gateway as the sole access point also violates the "defence in depth" principle, which advocates for multiple layers of security to protect against threats.

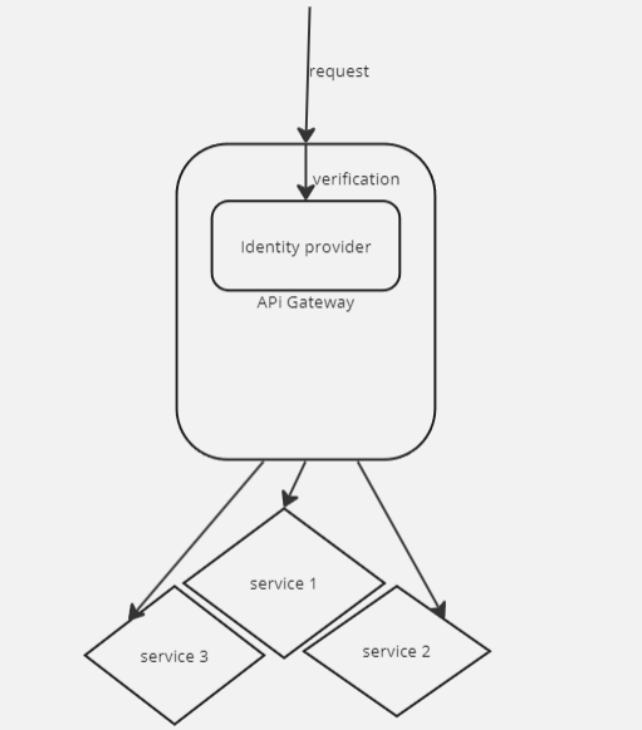

Figure III.7: API gateway as Identity Provider

2.3.2.3 Option 3: Our chosen Architecture: Since our architecture is based on microservices, we chose to use token-based (JWT) authentication with an asymmetric signing algorithm (RS256) to generate a private/public key pair. The pair of keys are saved in the authentication server and used to generate valid tokens; the public key is also saved in the API gateway and is used to verify incoming tokens in requests that require authentication. All in all, our authentication flow consists of creating the tokens in the authentication server and verifying request headers and bearer token validity at the level of the getaway. This choice enforces:

- **separation of concerns:** The responsibility of authenticating users and checking permission is delegated to a separate identity provider and the gateway is only responsible for proxying requests to their designated service.
- **security:** By using an asymmetric algorithm, we ensure that tokens can only be generated by the authentication server, while the public key can be safely distributed to verify tokens without exposing the private key.

We also chose to implement a worker that rotates the keys monthly to ensure security. Key rotation is a critical security measure that mitigates the risk of key compromise and limits the potential damage if a key is ever exposed. By periodically generating new private/public key pairs and retiring old ones, we enhance the security of our authentication system and our system in general.

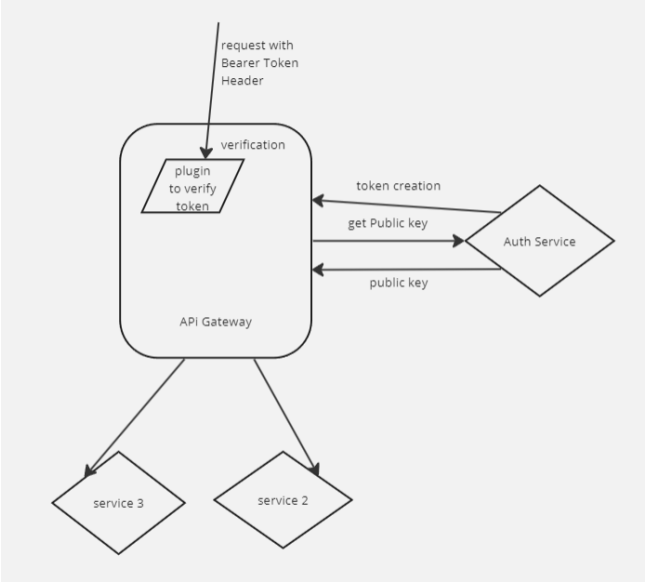

Figure III.8: Request flow in our architecture

2.3.3 Class Diagram

The authentication service is also responsible for persisting user data in a PostgreSQL database. The following is a class diagram of our user entity:

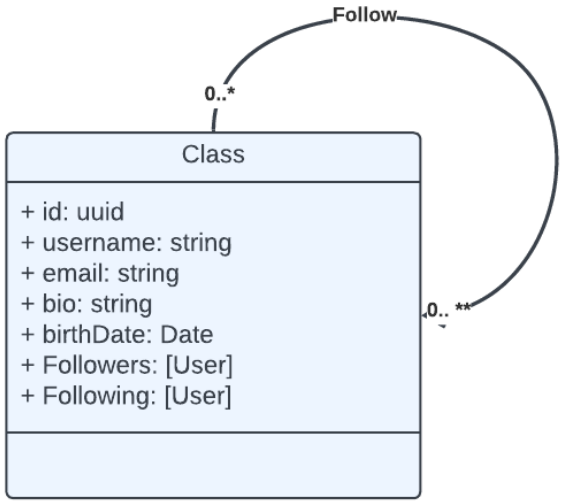

Figure III.9: Authentification service Class Diagram

2.4 API gateway

An API gateway is an API management tool that sits between a client and a collection of backend services. In this case, a client is the application on a user's device and the backend services are those on an enterprise's servers. We chose to use the open-sourced project: Kong. Kong API Gateway is a cloud-native, platform-agnostic, scalable API Gateway distinguished for its high performance and extensibility via plugins. By providing functionality for proxying, routing, load balancing, health checking, authentication (and more), Kong serves as the central layer for orchestrating microservices or conventional API traffic with ease. It is built on top of NGINX and openresty and optimized for high performance and low latency, making it suitable for managing and orchestrating traffic in complex microservice architectures.
Some benefits of using Kong:

- Improved Performance: Optimizes API traffic handling, reducing latency and improving response times.
- Simplified Management: Provides a single point of control for managing and monitoring APIs and microservices.
- Flexibility: Supports a wide range of configurations and plugins, allowing it to adapt to various use cases and requirements.
- Community and Support: Backed by a strong open-source community and a commercial offering (Kong Enterprise) for additional features and support.

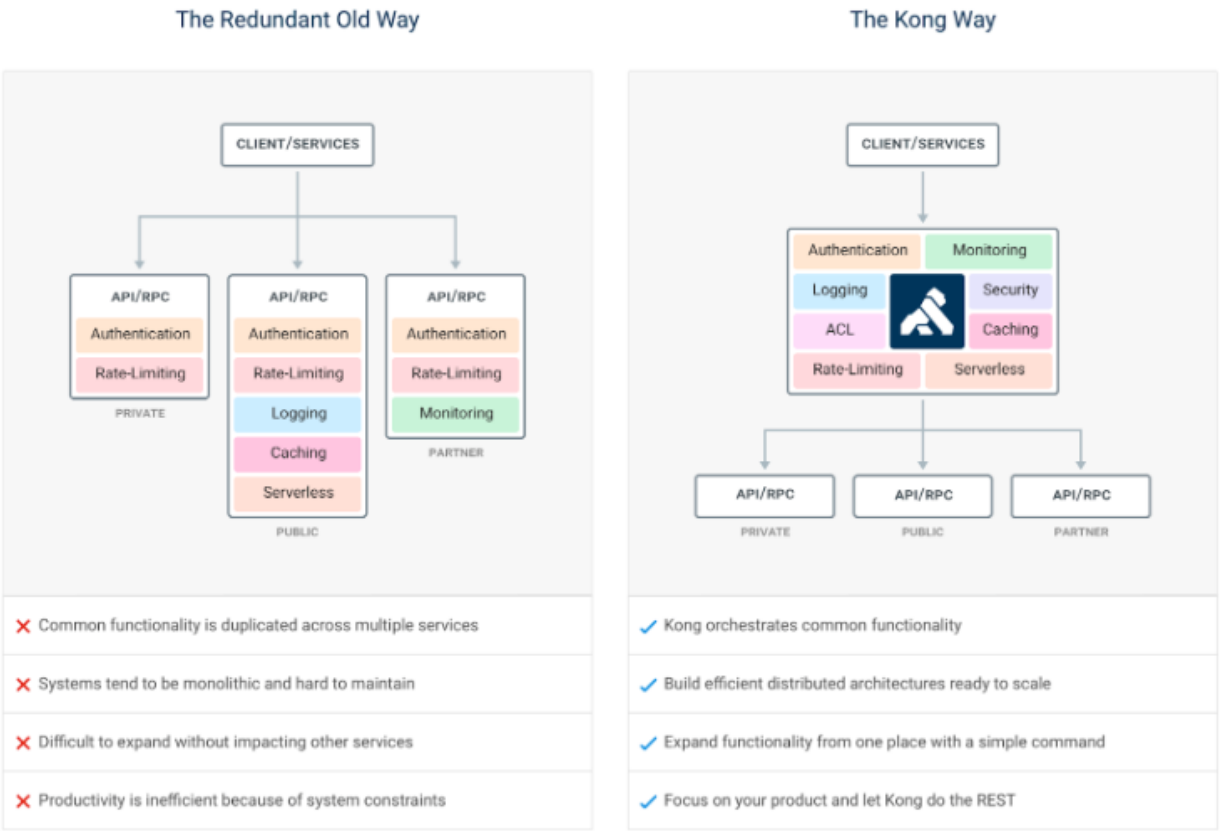

Figure III.10: Comparison old API gateways vs Kong

2.4.1 Authorization

We chose to handle the verification of the JWTs in the API Gateway for a better separation of concerns and also to reduce the workload on the services.

Kong JWT Plugin: Kong provides a plugin for JWTs that looks for the token in the header or the query parameters, verifies the token and sets headers with the data of the consumer (we add keys to the consumer for verification). This approach does not work with the WebSocket protocol since we don't have access to the headers of the handshake on the server.

Custom Auth Plugin: To mitigate this, we created a custom Kong plugin that verifies the token and sets data extracted from the token in the request where we found the token. So, if the token is in the "Authorisation" header we store the client ID and username in the "X-Client-Id" and "X-Client-Username" headers respectively and if the token is in the "token" query parameter we store the client ID and username in the "id" and "username" query parameters respectively. This allows to use of the same plugin with all the endpoints that need to verify if the client has a valid token.

3 Technology Choices And Argumentation

3.1 Functional Requirement

We are building a streaming platform that handles multiple concurrent connections and heavy data load and must endure high traffic. So our choice of technology must ensure High Availability, high concurrency support and high performance.

3.2 Language Choice

We have found that the BEAM system (the Erlang Virtual Machine) is perfect for our requirements. The BEAM System was engineered to support millions of concurrent connections for Telecom firms and to be fault tolerant in a way that faulty processes restart and resume their work without having problems propagate through the system or affect other connections. In terms of uptime, the BEAM system proved its viability and efficiency. Considering the concurrent nature of modern live-streaming platforms and the requirements needed, such a system proves to be the best choice. BEAM make sure that all requests are served. This is a highly available characteristic at the cost of performance yet it proves its efficiency when the system becomes complex and needs to scale.

Through our needs, we saw that the best software design approach would be a pipeline approach. It would be easier to maintain, debug, improve and code considering the inherent complexity of the system. Thus, it would be easier to have purely functional language for this usage. The Pure Functional Paradigm is a paradigm based on pure function and immutability meaning everything is constant and can't be changed, it is either copied or used but never changed and functions do not have side effects. For the same input, we always have the same output this is crucial also for caching which can improve performance for this reason we chose a purely functional paradigm and a data-oriented approach. Elixir is a popular programming language for web development due to its simplicity, efficiency, and powerful features. It offers an easy-to-understand syntax and allows developers to write less code while achieving complex operations. It is built on top of the BEAM and utilizes its capabilities while being optimized by design for concurrent services. Furthermore, Elixir proved to be perfect for our requirements:

- Elixir has a concise and expressive syntax that allows Elixir developers to write programs quickly with fewer lines of code than most alternatives.
- Elixir leverages powerful functional programming features like pattern matching, enabling developers to handle complex operations with minimal syntactic overhead.
- Elixir's key feature lies in its ability to achieve improved reliability via concurrency rather than failure prevention.
- All Elixir systems are built around completely concurrent processes running independently of one another, allowing them to recover from potential errors or disruptions without impacting the overall system performance.

3.3 Framework choice

An important discussion was about whether our Frontend and Backend should be tightly linked or separated. Mainly we cared about latency and performance, we wanted a smooth user experience and low latency for the live streaming and the live chat. Also, our web app would be very dynamic and require a lot of API calls and real-time data streaming. Even though SSR(server-side rendering) is easier to work with, program, improve and test at the early stages of a project, it will easily prove to be a burden to the team especially when it is just a service among many others. Also, Frontend microservices are hard to deal with and manage. Also for the goal of separation of concern, using CSR (client-side rendering) is more convenient for our case. Now the problem was, what js framework should we choose? There were multiple options to choose from: Angular, React, Vue, Svelte, Solid, Qwik and Emberjs. These were valid and well-founded options and we felt we experiment with what we saw fit the most. So our final choice was decided by what would work best for our system. First, due to the complexities of our system, we wanted a simple option that doesn't have complex state management. Second, due to the high load of data coming and going and the heavy traffic, we needed to squeeze every way we could to improve performance. In this scenario, frameworks that involved virtual DOM seemed in one way overengineered for our needs as we don't need all the state management and routing provided and also costly in terms of performance, we wanted something simple, effective and with as low performance overhead as possible For these reasons, we ruled out Angular, Vue, Emberjs and React. Qwik js was a valid option at first but it was quickly ruled out due

to one important point. Qwikjs design is more about lazy-loading javascript files meaning only loading code when needed by contrast to other big frameworks that load a huge javascript file from the start resulting in a long waiting time. This approach helps a lot with big projects that have a lot of code that may not be used by the user. in our case, the website is mostly made up of three/four pages so most of the javascript code will be used immediately or soon enough. There was no practical reason to use Qwik.js hence ruling it out. Svelte.js was our best option for many reasons:

- First, it is a well-documented framework with a good and evolving ecosystem of UI frameworks, utility libraries, etc.
- Svelte.js has a developer-friendly syntax and error handling and messages. It is a technology geared for developer productivity as much as React without having the huge performance overhead or complexity of its competitors.
- Sveltejs is a relatively lightweight framework compared to angular but has all the functionalities needed for front-end development in comparison to Reactjs which requires other libraries for state management for example which is too much of a learning curve.
- Svelte does most of the work on build time so it doesn't need to use VirtualDOM which improves the website performance and reduces javascript bundle size. The website only uses code that it needs any dead code is removed in the build process and reduces the initial load time significantly.
- Svelte's minimal runtime also means fewer JavaScript operations, which can greatly benefit users on slower devices or with limited bandwidth and makes it an excellent choice for building fast and responsive web applications.

4 Conclusion

Through this project, we demonstrated the effectiveness of combining proven technologies and best practices. The detailed architecture and service logic discussions provided insights into how each component contributes to the overall functionality and reliability of the platform. By addressing key considerations such as scalability, security, and maintainability, we ensured that the platform is well-equipped to handle current demands and future growth.

Chapter IV

Implementation

Contents

Introduction

After completing the conceptual study of our web application, we transition to the implementation phase. This chapter presents the final product of our web application, accompanied by screenshots illustrating its various functionalities and the user interface.

1 Software Environment

We now focus on the software tools crucial for its realization and documentation. Below is a list of software utilized for development and report writing:

- **Github** is an Internet hosting provider for software development and version control using Git. We used it to share the project among team members and collaborate on development
- **Visual Studio Code** is a source code editor created by Microsoft for all systems. It's a lightweight and user-friendly editor that supports many languages (through extensions).
- **Postman** is an API platform that allows developers to design, build, test, and iterate on their APIs. We used it to test all our APIs before integrating them into the frontend.
- **Docker Desktop** is a secure, out-of-the-box containerization software offering developers and teams a robust, hybrid toolkit to build, share, and run applications anywhere. We used it to monitor our docker images and containers.

2 Technologies Used

In this section, we will present the different technologies adopted for the implementation of our project.

2.1 Back-end Micro services

To ensure harmony and compatibility between microservices, we decided to implement them using a similar almost identical stack:

- **Elixir:** is a dynamic, functional language for building scalable and maintainable applications. Elixir runs on the Erlang VM, known for creating low-latency, distributed, and fault-tolerant systems.
- **Phoenix:** is a web development framework written in the functional programming language Elixir.
- **Postgresql:** is a free and open-source relational database management system emphasizing extensibility and SQL compliance.
- **Docker:** We used DockerFiles and Docker compose to run our services and our databases.

2.2 Front-end

For the frontend technologies, we discussed and looked at many options and finally settled on these technologies (for reasons discussed before):

- **Svelte:**A modern JavaScript framework that compiles components into highly efficient, imperative code that directly manipulates the DOM, resulting in fast and small web applications.
- **Svelte-kit:**A framework for building web applications with Svelte, offering features like server-side rendering, routing, and file-based configuration for creating fully-fledged applications.
- **Tailwind:** A utility-first CSS framework that provides low-level utility classes to build custom designs directly in the markup, promoting rapid and consistent styling.
- **Shadcn:** A highly customizable component library used for building user interfaces, with a focus on accessibility and design.

3 Application development

Moving on to the graphical section, we present various user interfaces with different functionalities.

3.1 Authentication pages

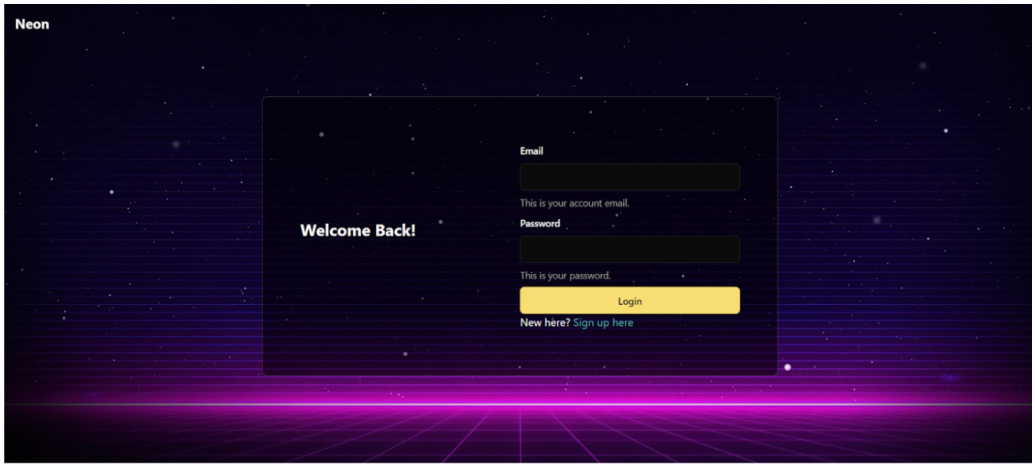

Figure IV.1: Login Page

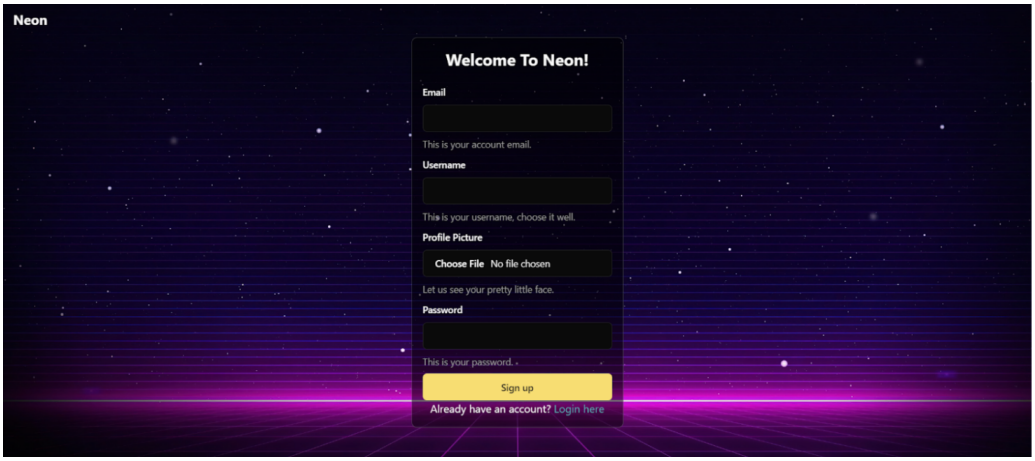

Figure IV.2: Sign-up Page

3.2 Profile page

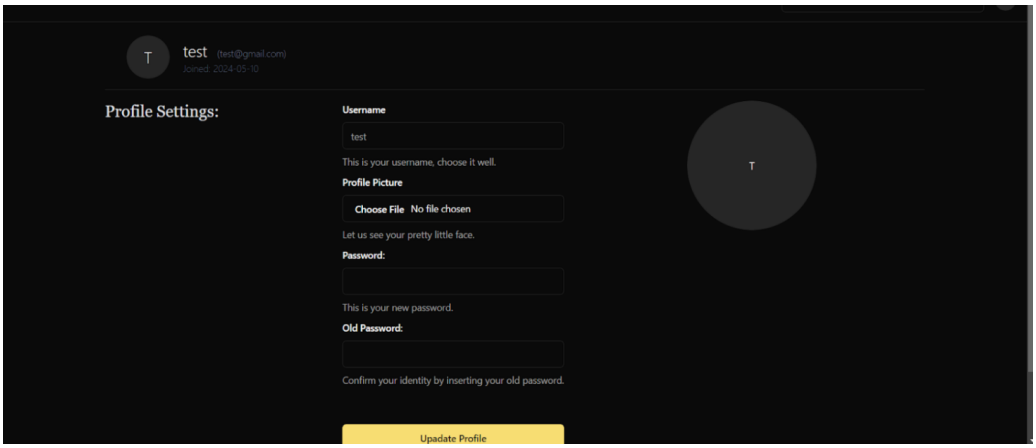

Figure IV.3: Profile Page

Conclusion

In implementing our live streaming platform, we meticulously selected and integrated a range of technologies to ensure a robust, scalable, and secure system. By leveraging a microservices architecture, we achieved modularity and flexibility, allowing each component to be developed, deployed, and scaled independently. T

General Conclusion and Perspectives

The development of our live streaming platform represents a significant achievement in combining modern technologies and best practices to deliver a high-quality, scalable, and secure service. By leveraging a microservices architecture, we achieved modularity and flexibility, allowing for independent development, deployment, and scaling of each service. This architectural choice was instrumental in addressing the diverse needs of streamers and viewers while ensuring robustness and efficiency.

The technologies used albeit having a steep learning rate and some having little to no documentation are the most suitable for our needs. They offer low latency and high throughput which are critical for live streaming. This report discussed three major parts. The first is the heavy task of understanding user and system requirements, analyzing them and determining the non-functional requirements. The second part was focused on the system design; in this part, we delved into our microservices architecture, explaining our choices and the logic behind them. We also explained communication Protocols such as HLS and RTMP. And lastly, a dedicated part for a run-through of the implementation phase.

A possible next step is to add other dedicated services such as a storage management system, a notifications system, and a subscription System ... we can also train and fine-tune a state-of-the-art recommender system to provide stream recommendations based on a user's behaviour or on the regional placement for new users.
Use Kafka as a data streaming pipeline to ensure further better uptime and fault tolerance as well as use Kafka S3 connectors to persist chunks of streams in S3 and then use Amazon S3 as a scalable, globally accessible, highly available caching layer lifting the load from the main server and enabling First Mile Delivery processing to multiple points without affecting the viewer experience.

Build an encoder for the new H.265 video encoding which proves to have better throughput and better compression ratio than its predecessor, as well as implement other encodings. HLS now has a multitude of encoding to choose from depending on the quality of the viewer's network, improving even further the smooth experience of live streaming. Adding new encodings will result on big loads of computationally intensive jobs. So, a new microservice can be added to improve the scalability of the system: a job scheduler with many worker clusters all responsible for multimedia processing. This can be implemented by using message queues and PubSub architecture.
Ids in a Distributed System can be very problematic due to possible collisions for this, we must use an algorithm that is distributedly friendly while also being an indexable ID. UUID fails to achieve the latter. A better option would be to use the Snowflake ID Generation algorithm instead.

Bibliography

1. Building Microservices, 2nd Edition by Sam Newman, published By O'Reilly
2. Phoenix Documentation: https://hexdocs.pm/phoenix/overview.html
3. Kong Documentation: https://docs.konghq.com/
4. Authentication and Authorization in Microservices Architecture - Part I:
5. Waterfall Methodology: A Comprehensive Guide By Atlassian
6. Rtmp streaming by restream
7. What is HTTP Live Streaming? | HLS streaming by Cloudfare
8. Svelte Documentation