OST
Eastern Switzerland
University of Applied Sciences

Distributed Systems
& Ledgers Lab

Development of a Scalable and Distributed Streaming Platform

Technical Report

Manuel Metzler   Sascha Häring

Bachelor Thesis

Eastern Switzerland University of Applied Sciences
Campus Rapperswil-Jona

Spring 2024

Dr. Thomas Bocek	Sven Stucki	Prof. Stefan F. Keller
Advisor	*Procivis AG*	*Reader*
	Examiner	

Abstract

The increasing demand for video streaming services highlights a gap in the availability of free and open-source tools that support scalable and distributed streaming with video conversion capabilities. Existing platforms like YouTube and Vimeo offer scalability and conversion but are neither open source nor self-hostable. Conversely, open-source solutions such as Jellyfin and PhotoPrism lack the ability to convert videos upon upload and do not scale efficiently.

This thesis aims to develop a backend server that facilitates scalable and distributed video streaming with integrated conversion capabilities. The objective is to create a free and open-source solution that addresses the limitations of current offerings in the market.

The development process began with an evaluation of streaming technologies, focusing on factors such as client support, openness of the standard, and streaming requirements. Dynamic Adaptive Streaming over HTTP (DASH) was selected due to its open standard, extensive client support, and compatibility with royalty-free formats such as WebM-DASH. To achieve scalability, the server was designed to be stateless. Video files are automatically converted and published post-upload utilizing a message queue system for signaling, ensuring scalable and atomic conversion tasks.

Using DASH as the underlying streaming technology allowed the use of modern video formats, reducing storage requirements. The stateless design enabled horizontal scaling, while the message queue ensured efficient handling of video conversion tasks. Each conversion task is processed exactly once and in the order received, ensuring reliability and temporal decoupling.

The developed streaming platform successfully meets the needs for a scalable, distributed, and open-source video streaming service with integrated conversion capabilities. It fills the existing gap by providing a self-hostable solution that leverages modern streaming standards and scalable architecture, making it a valuable tool for communities and organizations seeking an alternative to proprietary platforms.

Acknowledgements

We would like to thank selected members of staff at the Eastern Switzerland University of Applied Sciences for their involvement in this work. In particular, we thank Dr. Thomas Bocek for encouraging our interest in distributed systems and providing countless mentoring sessions over the last three semesters. We also extend our gratitude to Prof. Dr. Olaf Zimmermann for his guidance during the development of the software architecture, as well as for his lecture, *Application Architecture*, which proved very helpful.

Management Summary

Problem

Video streaming is increasingly important for both education and entertainment. Existing video streaming solutions are often proprietary, lack self-hostability, or miss key features. Proprietary platforms like YouTube and Vimeo offer scalability and video conversion but are closed-source and cannot be self-hosted. Open-source alternatives such as Jellyfin and PhotoPrism do not support on-upload video conversion and lack scalability features. To bridge this gap, a free and open-source tool is required that provides both scalable video streaming and integrated video conversion in a self-hostable format.

Purpose

This bachelor thesis aims to develop a backend server that enables scalable and distributed video streaming with integrated video conversion capabilities. The goal is to create a system that combines the best aspects of proprietary platforms' scalability and automatic conversion functionality with the openness and self-hostability of open-source solutions.

This serves the following purposes:

1. Allow users to have complete control over their streaming infrastructure.
2. Maintain high performance, even under heavy load due to the system's scalability.
3. Provide the convenience of automatic video conversion upon upload.

Methods

To achieve this goal, the project involved several key steps:

Technology Evaluation Various streaming technologies were evaluated based on openness, client implementation support, and serving requirements. Dynamic Adaptive Streaming over HTTP (DASH) was selected for its open standard, wide client support, and use of royalty-free formats.

Architectural Design A stateless server design was adopted to handle numerous simultaneous requests without maintaining session information, allowing for horizontal scalability.

Message Queue Integration A message queue system was integrated for video conversion tasks to decouple upload and conversion processes. This ensured that the system remained responsive and efficient, with tasks processed independently and managed atomically.

Implementation and Testing The implementation of DASH allows the use of modern video formats. Strict testing and iterative improvements were conducted throughout the development process to ensure DASH compliance and reliability.

Results

The final product is a scalable, distributed, and open-source video streaming service that supports both efficient streaming and seamless video conversion. The stateless server design allows for easy scalability, and the integration of a message queue system ensures efficient processing of video conversion tasks. The use of DASH allows a wide range of clients to stream videos from our service, enabling adaptation for various purposes. This solution provides a valuable alternative to existing proprietary and open-source platforms, offering users complete control over their streaming infrastructure.

Contents

References

[1] *AV1 Bitstream & Decoding Process Specification*, Specification, version 1.0.0-errata1, Alliance for Open Media, 2019-01. [Online]. Available: https://aomedia.org/av1/.

[2] A. Beach and A. Owen, *Video Compression Handbook*. Peachpit Press, 2018.

[3] W. Eddy, *Transmission Control Protocol (TCP)*, RFC 9293, 2022-08. [Online]. Available: https://www.rfc-editor.org/info/rfc9293.

[4] R. T. Fielding, M. Nottingham, and R. Julian, *HTTP Semantics*, RFC 9110, 2022-06. [Online]. Available: https://www.rfc-editor.org/info/rfc9110.

[5] V. K. Garg, *Principles of Distributed Systems*. Kluwer Academic Publishers, 1996.

[6] M. Glinz, "On Non-Functional Requirements," in *15th IEEE International Requirements Engineering Conference (RE 2007)*, IEEE, 2007, pp. 21–26.

[7] *H.264: Advanced Video Coding for Generic Audiovisual Services*, Recommendation, International Telecommunication Union, 2021-08. [Online]. Available: https://www.itu.int/rec/T-REC-H.264.

[8] *H.265: High Efficiency Video Coding*, Recommendation, International Telecommunication Union, 2023-09. [Online]. Available: https://www.itu.int/rec/T-REC-H.265.

[9] *HTTP Live Streaming*, Specification, Apple Inc. [Online]. Available: https://developer.apple.com/streaming/ (visited on 2024-03-25).

[10] ISO/IEC 13818-3:1998, *Information technology – Generic coding of moving pictures and associated audio information*, Part 3: Audio, 1998-04. [Online]. Available: https://www.iso.org/standard/26797.html.

[11] ISO/IEC 14496-14:2020, *Information technology – Coding of audio-visual objects*, Part 14: MP4 File Format, 2019-12. [Online]. Available: https://www.iso.org/standard/79110.html.

[12] ISO/IEC 14496-3:2019, *Information technology – Coding of audio-visual objects*, Part 3: Audio, 2019-12. [Online]. Available: https://www.iso.org/standard/76383.html.

[13] ISO/IEC 23009-1:2022, *Information technology – Dynamic adaptive streaming over HTTP (DASH)*, Part 1: Media presentation description and segment formats, 2022-08. [Online]. Available: https://www.iso.org/standard/83314.html.

[14] ISO/IEC 23009-2:2020, *Information technology – Dynamic adaptive streaming over HTTP (DASH)*, Part 2: Conformance and reference software, International Organization for Standardization, 2020-09. [Online]. Available: https://www.iso.org/standard/79107.html.

[15] Q. Jiang, Y. C. Lee, and A. Y. Zomaya, "The Limit of Horizontal Scaling in Public Clouds," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 5, no. 1, 2020.

[16] J. Kalvenes and N. Keon, "The market for video on demand," *Networks and Spatial Economics*, vol. 8, pp. 43–59, 2008.

[17] R. C. Martin, *Clean Architecture*. Prentice Hall, 2017.

[18] T. Masternak and S. Pobiega, "Exactly-once message delivery," *Exactly Once: On Distributed Systems*, 2020. [Online]. Available: https://exactly-once.github.io/posts/exactly-once-delivery/.

[19] *Matroska Media Container*, Specification, Matroska.org, 2017-06. [Online]. Available: https://www.matroska.org/index.html.

[20] *Microsoft Smooth Streaming*, Specification, Microsoft Corporation. [Online]. Available: https://web.archive.org/web/20100615150921/http://www.iis.net/download/SmoothStreaming (visited on 2024-03-25).

[21] *open(2) Linux User's Manual*, Manual, The Linux man-pages Project, 2024-05. [Online]. Available: https://www.kernel.org/doc/man-pages/ (visited on 2024-04-20).

[22] R. Pantos and W. May, *HTTP Live Streaming*, RFC 8216, 2017-08. [Online]. Available: https://www.rfc-editor.org/info/rfc8216.

[23] *pipe(7) Linux User's Manual*, Manual, The Linux man-pages Project, 2024-05. [Online]. Available: https://www.kernel.org/doc/man-pages/ (visited on 2024-04-20).

[24] N. Poulton, *The Kubernetes Book*. Leanpub, 2024. [Online]. Available: https://leanpub.com/thekubernetesbook.

[25] *Recommended Settings for VOD*, Specification, Google LLC. [Online]. Available: https://developers.google.com/media/vp9/settings/vod (visited on 2024-03-08).

[26] L. Richardson, *Justice Will Take Us Millions Of Intricate Moves*, 2008. [Online]. Available: https://www.crummy.com/writing/speaking/2008-QCon/ (visited on 2024-03-26).

[27] M. Seufert, S. Egger, M. Slanina, T. Zinner, T. Hoßfeld, and P. Tran-Gia, "A Survey on Quality of Experience of HTTP Adaptive Streaming," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 1, pp. 469–492, 2015.

[28] K. Spiteri, R. Urgaonkar, and R. K. Sitaraman, "BOLA: Near-Optimal Bitrate Adaptation for Online Videos," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1698–1711, 2020.

[29] G. Starke, M. Simons, S. Zörner, and R. D. Müller, *arc42 by Example*, Software Architecture Documentation in Practice. Packt Publishing Ltd, 2019.

[30] M. A. Titmus, *Cloud Native Go*. O'Reilly Media, 2021.

[31] J.-M. Valin, K. Vos, and T. B. Terriberry, *Definition of the Opus Audio Codec*, RFC 6716, 2012-09. [Online]. Available: https://www.rfc-editor.org/info/rfc6716.

[32] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "A Kubernetes Controller for Managing the Availability of Elastic Microservice Based Stateful Applications," *Journal of Systems and Software*, vol. 175, p. 110 924, 2021.

[33] *Vorbis I Specification*, Specification, Xiph.Org Foundation, 2020-07. [Online]. Available: https://xiph.org/vorbis/doc/Vorbis_I_spec.html.

[34] *VP9 Bitstream & Decoding Process Specification*, Draft, version 0.7, WebM Project, 2017-02. [Online]. Available: https://www.webmproject.org/vp9.

[35] *WebM Container Guidelines*, Draft, WebM Project, 2023-10. [Online]. Available: https://www.webmproject.org/docs/container.

[36] *WebM Dash Specification*, Specification, WebM Project. [Online]. Available: https://wiki.webmproject.org/adaptive-streaming/webm-dash-specification (visited on 2024-04-01).

[37] A. Wiggins. "The Twelve-Factor App." (2012), [Online]. Available: https://12factor.net (visited on 2024-03-12).

[38] O. Zimmermann and M. Stocker, *Design Practice Reference*, Activities and Templates to Craft Quality Software in Style. Leanpub, 2023. [Online]. Available: https://leanpub.com/dpr.

Glossary

Adaptive Bit Rate Streaming (ABR) Data that a client wants to receive is provided in different bit rates and qualities, allowing the client to adapt its streaming to different network conditions. 15, 16, 20, 24, 31, 34, 38, 43, 67

Availability The ability of a system to continue operating and serving requests despite failures or issues. "Every request receives a (non-error) response, without the guarantee that it contains the most recent write." 3, 16, 18

Cloud Native Cloud native is an approach to software development that utilizes cloud computing to build and run scalable applications in modern and dynamic environments. Container technology, microservice architecture and deployment via declarative code are common elements of this architectural style.[1] 5, 18, 30, 39, 40, 41, 42, 45, 49, 61

Consistency The property of a system in which the state across all participants is the same at the same time. "Every read receives the most recent write or an error." 18, 22, 26, 27, 29, 40

Distributed System A system consisting of multiple computers which do not share a clock or memory, connected by a communications network. 2

Dynamic Adaptive Streaming over HTTP (DASH) A streaming technique that adjusts video quality in real-time based on network conditions to ensure smooth playback. 26, 31, 32, 36, 37, 38, 42, 43, 44, 45, 54, 57, 62, 63, 67

Frames per Second (fps) A measure of how many individual images are displayed per second in a video. 8, 43

GNU Affero General Public License (AGPL) A strong copyleft license that permits anyone to use the software licensed under it commercially or privately, and to distribute

[1]https://github.com/cncf/toc/blob/main/DEFINITION.md

and modify it freely. When a modified version is released or provides a service over a network, the complete source code of that modified version must be made available under the same license.[2] 61, 67

Open-Source Software that gives users the freedom to run, study, change, and distribute it. It is often referred to as free software, a term that emphasizes the additional freedoms that it grants. 3, 4, 26, 37, 53, 61, 62, 67, 68

Parallel System A multiprocessor (computer) system in which individual processors communicate via a shared memory. 2

Source-Available A term used to describe software for which the source code is made available to users, but the terms of distribution do not necessarily comply with the criteria for free software. Source-available software allows users to view the code and sometimes modify it, but it may come with restrictions to other freedoms. 4

State The data or condition of a system at a particular point in time. 2, 18, 22, 45, 62

Transmission Control Protocol (TCP) TCP provides reliable data transfer over an unreliable network (e.g. IP) by sending it in small parts and waiting for acknowledgement by the recipient. 14, 30

Video on Demand (VoD) Distributing video media without relying on a traditional video playback device (e.g. DVD Player) or conformance to a broadcasting schedule on the consumer side. 3, 4, 6, 7, 16, 25, 43

[2]https://www.gnu.org/licenses/agpl-3.0.txt

List of Figures

List of Tables

List of Code Listings

Part I.

Technical Report

Chapter 1.

Introduction

This report details the development process of the GoReeltime software, a bachelor thesis project at the Eastern Switzerland University of Applied Sciences (OST).

The aim of this chapter is to introduce the task assignment and explain the motivations and reasons for the inception of this project. Additionally, it provides an initial assessment of the project domain and outlines the established core values.

1.1. Assignment

The main goal of this project is the implementation of a media streaming platform as a distributed system, a single service provided by many different computers.

A key advantage of building a distributed system instead of a parallel system is the possibility of horizontal scalability to accommodate increased demand for a service. Unlike vertical scalability, where the resources of a single compute unit providing the service are increased (e.g., adding more memory to a system), horizontal scalability distributes requests across multiple compute units participating in the system [5]. This approach to software system design has gained popularity with the advent of virtualization technologies and has become a de facto standard for most applications in the ecosystem around container technologies [15].

While designing software as a distributed system offers many opportunities, such as increased fault tolerance and more flexible cost control, it also presents certain challenges that have to be overcome. The most notable ones are the absence of a shared state and temporal decoupling, as one system might not respond immediately to a request made by another [5].

A secondary goal of this project is the utilization and implementation of a streaming protocol to allow access to media provided by the system. In contrast to a computer that serves complete units of data (e.g., a file server), a streaming server allows for the gradual retrieval of certain parts of a file for direct consumption, commonly used in video streaming. The service developed during this project follows the Video on Demand (VoD) approach, prioritizing client compatibility over the reduction of storage needed on the server side [16].

The final goal is to implement the media distribution service in a manner that allows for the automatic conversion of uploaded media to the format required by the streaming protocol. This ensures high compatibility with the chosen protocol on the server side and optimizes performance by streamlining the data for consumption on the client side.

1.2. Motivation

At the time of writing, there is no VoD software available that fulfills *all* the following requirements:

Open-source The service qualifies as open-source software, promoting adoption, security by transparency, and community collaboration. Open-source software empowers developers to inspect, modify, and enhance the code, ensuring continuous improvement and customization to specific needs.

Self-hosted Users can deploy the service on their own infrastructure, providing data privacy and customization to exact needs. This approach mitigates the risks associated with third-party data breaches and ensures full control over content management and delivery.

Scalable The software can adapt to varying numbers of requests without compromising performance, making it suitable for large deployments. Scalability ensures that as the user base grows, the system can maintain high availability and responsiveness, essential for providing a seamless viewing experience.

The development of such a platform would enable the broad distribution of knowledge and other video media without relying on third parties or compromising one's freedoms, addressing a gap in the current landscape of VoD solutions. By eliminating the dependency on proprietary platforms, organizations and individuals can maintain autonomy over their content. Additionally, this solution would foster innovation through community-driven development and collaborative problem-solving, creating a robust and versatile VoD platform that meets diverse needs and use cases.

1.3. Conditions

The following is a list of conditions and constraints that apply during this project, including reasoning about why they should be in effect.

Open-source release The software developed during this project is to be released as open-source software. This not only supports educational purposes but also aims to attract additional contributors to the project in the future, as it is planned to be further maintained as an open-source project beyond the scope of this bachelor thesis.

Dependencies can be studied Any additional components required to run and operate the developed software need to be licensed under at least a source-available license. Since this project is conducted in an academic environment and will itself be licensed as open-source software, it is important that all its dependencies can also be studied.

1.4. Project Domain

Even though the implementation of this media streaming platform takes place in an educational environment, it does so with the goal of producing usable software. In order to work towards providing usability to a specific stakeholder, the *Distributed Systems and Ledgers Lab* (DSL) at the Eastern Switzerland University of Applied Sciences (OST) has agreed to act as a customer. They are looking for a solution to distribute video lectures to students enrolled in their various courses.

To further specify the environment this project is taking place in, a project domain can be described by combining these stakeholder requirements with the general goals and motivation of the project. Doing so, the following qualities emerge:

Self-Hosted The DSL at OST wishes to control their published content by providing their own video streaming service. This means the developed software needs to be easily installable and come pre-configured with reasonable defaults. It should also be able to run in larger environments and support gradual growth in environment size in accordance with the current demands regarding system load. In such larger environments, it should provide qualities like fault tolerance and load balancing across multiple machines.

The Twelve-Factor App With the growing popularity of the cloud native software architecture style, a popular methodology for software projects titled “The Twelve-Factor App” has also risen to fame, promoting technologies and techniques well-suited to such environments. Initially written by Adam Wiggins with contributions from many other engineers working on the Heroku application platform, it lays down twelve important factors that software written for cloud native environments should fulfill [37]. While some of these qualities may take some effort to implement and adhere to (e.g., dependency injection principle, stateless services), they have proven to enhance both maintainability and failure resiliency in software systems [17].

Chapter 2.

Analysis

During this chapter, further details of the previously introduced task are worked out. This encompasses a brief overview of the required functionality of the service, as well as insights from similar products in the market. The final section provides a technical introduction to video streaming, forming a foundational understanding for the core functionality of the service.

2.1. Requirements

The goal of this project is to produce a functional media streaming software, as presented in the assignment introduction. To further refine this definition, the general requirements for fulfilling the task are separated into two parts: functionality and compliance.

2.1.1. Functionality

The main functionality of the developed service is to provide a platform for video media streaming. Considering this, special care has to be taken when deciding on the video streaming protocol, as this will affect the types of clients that can consume media from the service. It should be possible to use either a third-party media player or an embedded built-in client.

The implementation values client compatibility over lower server resource consumption. For example, this could mean that more data is stored on the server side if this enables more clients to stream more optimally. This approach aligns with the VoD methodology, emphasizing content delivery above all else.

Table 2.1.: Evaluation of similar products that already exist in the market.

Product	OSS	SH	SC	Verdict
YouTube	✗	✗	✓	Reliable and stable service, but operated by a third party.
Plex	✗	✓[1]	✗	Good for managing a personal media collection, but neither open-source nor fully manageable internally.
Jellyfin	✓	✓	✗	A personal media collection manager not suitable for VoD use cases, does not scale.
PhotoPrism	✓	✓	✓	Only support short video media and is not built for VoD use cases.

[1] Plex only works with an external authentication service.

However, it is *not* a priority of the project to put extended effort into the user-facing frontend of the platform. Depending on the deployment environment, a user interface (e.g., website, lecture index) may already be present, in which case it would primarily rely on a robust backend implementation of the video streaming service.

2.1.2. Compliance

Given the top priority of client compatibility, the developed software should fully and optimally comply with the specifications of the chosen video streaming protocol. If there is a repeatable verification test for the chosen protocol, the software should pass it flawlessly, ensuring that no compatibility issues arise.

2.2. Market Analysis

Several software projects published on the internet fulfill at least one of the requirements mentioned in section 1.2, such as being *open-source* (**OSS**), *self-hosted* (**SH**), and *scalable* (**SC**). Table 2.1 shows an overview of existing projects, their compatibility with the requirements in the motivation section, and why they do not yet achieve what the software developed during this project aims to accomplish.

While all products listed in Table 2.1 fulfill their intended use case very well, they do not meet every aspect of the outlined schema.

Thus, developing a new solution that meets the defined requirements is warranted.

2.3. Domain Analysis

Having confirmed that the inception of a software project is justified, the process of gathering domain knowledge can begin. When building a video streaming service, understanding the underlying components and their interactions is crucial. This involves a deep dive into several key areas, including video encoding, bit rate management, and the mechanics of streaming itself. Each of these elements plays a vital role in ensuring the seamless and efficient delivery of video content to users.

2.3.1. Video

The key component of a video streaming service is the concept of digital video itself. At its core, a video is a sequence of images displayed in rapid succession in order to create the illusion of motion. However, storing an image for every single *frame* (unit in which visual updates are measured) the video is made up of would be highly inefficient, requiring (1) an excessively large amount of storage, and (2) making playback more resource-intensive, as a full image would need to be loaded multiple times per second. To address this issue, a video only contains full images at so-called *key frames*, with other visual changes only being recorded as per-frame changes.

To enable efficient digital video functionality, various specialized methods are employed:

Key Frames Key frames, also known as *intra-frames* or *I-Frames*, are frames in a video that are encoded without reference to any other frames. They save storage by serving as reference points for subsequent or previous frames, which are encoded based on the differences from these key frames rather than being encoded independently [2].

The placement of key frames can have an effect to the compatibility of video streaming technologies with the encoded video data. How and when these key frames are created can be approached in three different ways:

- *Fixed Interval*:

  In this approach, the interval between key frames is fixed by specifying either a time interval or a set number of frames between key frames. When choosing a time interval (Δ_t), the Frames per Second (fps) measure of the video (fps_v) is used to determine how many frames lay between key frames:

$$\Delta_t \cdot \mathrm{fps}_v = \text{number of frames per key frame}$$

This method ensures that key frames are placed at regular intervals, which can be beneficial for certain types of streaming due to its predictability. However, it does not leverage the optimizations that certain algorithms could apply.

- *Algorithmic/Content-Based*:

  Depending on the encoder used to compress the video, certain algorithms can be used to detect scene changes or significant differences between frames to decide when to insert a key frame.

  This method is more efficient in terms of compression, as it places key frames only when necessary, based on content changes, leading to potentially smaller file sizes. It is primarily used when the smallest possible file sizes are required, such as when sharing full files across a network. However, due to less predictable in key frame spacing, this method is not optimal for streaming applications.

- *Hybrid*:

  It is also possible to limit the maximum interval between key frames while allowing an algorithm to decide if smaller intervals are needed during more complex sections of the video. This method aims to balance the predictability of fixed intervals with the efficiency of content-based placement.

Predicted Frames Predicted frames, also known as *inter-frames*, are video frames that reference other frames, be that past or future frames. There are two types of predicted frames:

- P-Frames: These are *forward* ***p****redicted frames* which reference a previous frame of any type, not just key frames. By only storing the changes to the previous frame, using predicted frames can significantly reduce storage requirements for digital video.
- B-Frames: Being ***b****i-directionally predicted frames*, these inter-frames reference not only previous frames, but also subsequent ones. Doing so, they can save even more storage, allowing for data to be referenced in multiple directions.

This difference-based encoding (used in inter-frames like P-frames and B-frames)

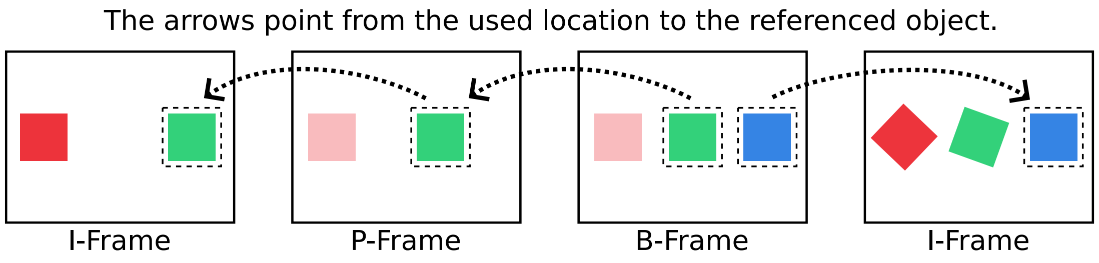
The arrows point from the used location to the referenced object.
I-Frame
P-Frame
B-Frame
I-Frame

Figure 2.1.: The different types of intra- and inter-frames, showing I-, P- and B-frames.

Table 2.2.: An overview comparing common modern video codecs.

Name	Creator	Notes
`H.264`	ISO/IEC (MPEG)	Advanced Video Codec (AVC), widely used [7].
`H.265`	ISO/IEC (MPEG)	High Efficiency Video Coding (HEVC), improvement to H.264 [8].
`VP9`	Google / WebM Project	Open standard, direct competition with H.265 [34].
`AV1`	Alliance for Open Media	Open standard, most efficient as of 2024 [1].

reduces the amount of data needed, as only the changes from the key frames are stored, leading to significant storage savings [2]. Figure 2.1 provides an overview of how predicted frames use references to other frames to minimize overhead in a video file.

Video Format The video format refers to the *codec* used to encode and decode video data. It provides the algorithms used to determine inter-frame types, what other frames they reference and how visual changes are reflected on them. The format also defines how video data is compressed and decompressed, impacting quality, compression efficiency, and computational requirements.

Some example codecs are shown in Table 2.2.

Streaming technologies often rely on specific codecs, making the choice of the codec a valuable decision when it comes to selecting a streaming technology.

Audio Format Digital video media often also includes audio. Similar to video, streaming technologies frequently depend on specific audio codecs, linking the choice of audio codec to the overall decision-making process. Likewise, the audio format (or audio *codec*) determines how audio data is sampled, quantized, and processed.

Some example codecs are shown in Table 2.3.

Table 2.3.: An overview comparing common modern audio codecs.

Name	Creator	Notes
`MP3`	Fraunhofer Society	MPEG-1 Audio Layer III, most common format for digital audio [10].
`AAC`	ISO/IEC (MPEG)	Advanced Audio Codec, successor to MP3, widely used [12].
`Vorbis`	Xiph.Org Foundation	Open standard [33].
`Opus`	IETF	Open standard to replace all others [31].

Container Formats Container formats, also called *wrappers*, are file formats that bundle video, audio, and sometimes other data (like subtitles and metadata) into a single file. While the video format dictates how the video data itself is encoded, the container format determines how different types of data streams (video, audio, etc.) are packaged together and synchronized within the file [2]. Common containers are:

- `AVI`: The default Microsoft Windows media container.
- QuickTime (`MOV`): The default video container by Apple Inc.
- `MP4`: Video and audio container for the MPEG codecs [7], based on QuickTime.
- Matroska (`MKV`): An open standard not limited to video and audio; it can hold virtually anything [19].
- `WebM`: A subset of Matroska, commonly used in web-based media distribution with open audio (`Vorbis`, `Opus`) and video formats (`VP9`, `AV1`) [35].

Header Information Header information in video files contains metadata about the video and audio streams, such as codec types, frame rates, resolution, and timing information. The location of this header information is crucial for video streaming because it allows a streaming client to understand and correctly interpret the incoming data before playing it. The container format of a file determines where header information is placed. Depending on the streaming technology used, it may be required to place the header at the start of the file, allowing the client to begin streaming as soon as the header is read [2].

In this project, header information is also relevant for the conversion of uploaded media. Some file types (e.g., QuickTime [2]) write header information at the end instead of the beginning of the file, which must be handled accordingly.

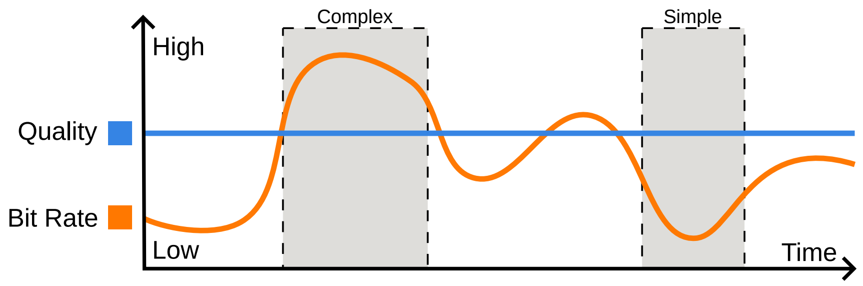

Figure 2.2.: The effects of VBR encoding on bit rate and quality of a video with visually complex and simple section.

2.3.2. Bit Rate

In order to fully understand video streaming and the opportunities it brings, bit rate needs to be covered first. Bit rate is the core concept that allows video streaming to adapt to the current network conditions, such as low or unstable bandwidth.

Generally the term *bit rate* refers to the number of bits processed over a given amount of time. In video streaming, this involves transmitting bits over a communications network. Additionally, bit rate is significant in video compression. During encoding, the *amount of information stored per unit of time* is also referred to as bit rate. This means that bit rate is in a constant trade-off with the quality of the encoded video. Using more bits yields higher quality video with more detail, while fewer bits reduce quality.

There are different methods to handle the *bit rate to quality balance*, each with their benefits and disadvantages [2]:

Variable Bit Rate (VBR)

> In VBR encoding, the bit rate varies throughout the encoding process, based on the complexity of the content, maintaining constant quality. This allows for high quality in detailed scenes and lower bit rates in simpler scenes, resulting in more efficient compression. The effect this has is shown in Figure 2.2.
>
> While VBR encoding results in close to optimal file compression, the varying size of processed data poses a problem for video streaming applications, where a more predictable bit rate is required.

Constant Bit Rate (CBR)

> CBR encoding maintains a constant bit rate throughout the entire video. While this provides predictable file sizes and thus data transfer rate, it may not always be the

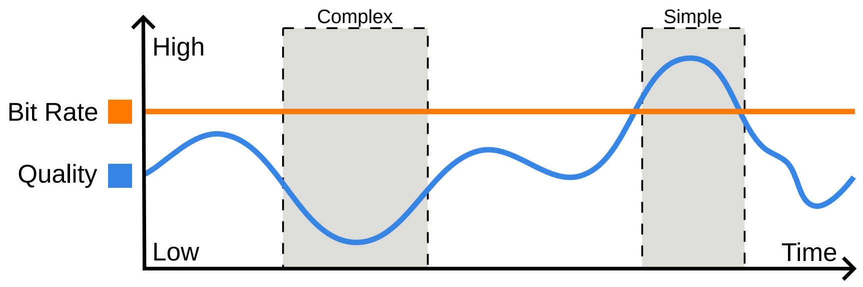

Figure 2.3.: The effects of CBR encoding on bit rate and quality for video sections with more or less detail.

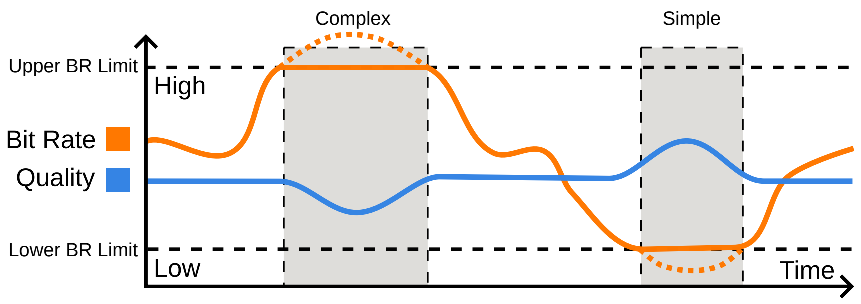

Figure 2.4.: The quality and bit rate implications of encoding with a constrained bit rate.

most efficient in terms of quality or minimizing file size. All content is encoded with the same bit rate, regardless to how much information is actually needed, resulting in poor visual quality for detail-rich sections, shown in Figure 2.3.

Constrained Bit Rate In constrained bit rate encoding, complex sections of the video may suffer a small quality loss, while simpler sections may use more information than necessary. It adds an upper and lower bit rate boundary, ensuring that the bit rate stays within a predictable range. This balance is shown in Figure 2.4.

Constrained bit rate is a good compromise between visual quality and keeping the amount of data to transfer predictable, making it an excellent choice for video encoding.

2.3.3. Streaming

After an introduction to digital video formats and the effect that bit rate has on their contents, the process of the actual video streaming can be explained properly.

Data distribution over a communications network aims to make content available on another computer as fast as possible, while keeping the process reliable. This was initially solved by using the Transmission Control Protocol (TCP) [3]. Without utilizing video streaming technologies, the process to download and play a video file from a server is as follows:

1. The client requests the file from a server.
2. The server begins transferring small increments of the file to the client, waiting for acknowledgment before transferring more.
3. The client reassembles the file as increments are received.
4. Once reassembled, the file can be played on the client.

Depending on how fast the network connection speed was, as well as how large the video file was, this transfer may have taken arbitrarily long. The recipient of the content also needs to plan ahead, as the file can only be played after being completely downloaded. From a technical perspective, this is equal to copying a file over a network to a remote machine before playback.

On the other hand, video streaming is the process of starting the playback as soon as content begins to arrive at the client, even though the rest of the data is still being transferred [13].

Header Information Before the client can start to interpret received data, it needs to know *how* to interpret it. This requires header information to be transferred first, so the client can be configured correctly.

Frame Consistency In order to interpret the received data, the client must ensure that all referenced key frames have been downloaded. If video data contains frames referencing other frames not yet received, playback will not function properly. TCP does not know anything about the layer it is providing a service to, so it does not know about key frames. This mechanism therefore needs to be handled on the application layer [3].

If these conditions are met, playback of a stream can start as soon as the first parts of content arrive at the target machine, unlike file copying which requires the entire file first. See Figure 2.5 for reference.

When incorporating bit rate concepts with streaming, it becomes possible to serve content under unpredictable network conditions. For example, a client is streaming content from

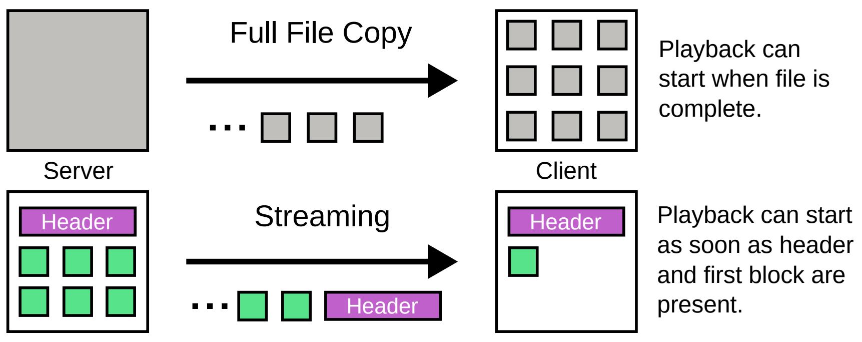
Full File Copy
Server
Client
Playback can start when file is complete.
Streaming
Header
Header
Header
Playback can start as soon as header and first block are present.

Figure 2.5.: Comparison of file copying and streaming, showing when playback is ready.

a server while experiencing changes in network quality. When the server offers the stream in different sizes, the client can select which stream to consume depending on the current state of the network. If the bandwidth is low, the client could select a stream with a low bit rate, and the opposite if the bandwidth were to increase again.

This is referred to as Adaptive Bit Rate Streaming (ABR), a streaming technique that allows the stream to adapt to varying network conditions by adjusting the quality of the video in real-time. Unlike traditional streaming methods that use a single bit rate throughout the playback, ABR dynamically switches between different bit rates based on the current network bandwidth and device capabilities. This ensures a smoother and more reliable viewing experience, even when network conditions fluctuate.

The following describes how ABR works:

1. **Encoding Multiple Versions:** The video content is encoded at multiple bit rates, resulting in several versions of the same video with different quality levels. These versions are then segmented into small chunks, typically a few seconds each.

2. **Manifest File:** A manifest file (sometimes also referred to as playlist) is created, listing all the different bit rate versions and their respective chunks. This file is used by the client to manage the streaming process.

3. **Initial Request:** When a user begins to play a video, the client initially requests the video to start streaming. The process of choosing which representation is used for the initialization depends on the actual client implementation.

4. **Continuous Monitoring:** The client continuously monitors relevant metrics, such as the available bandwidth, network conditions, and device performance.

5. **Dynamic Switching:** Based on the monitoring data, the client dynamically switches

to higher bit rate chunks if the network conditions are good, or to lower bit rate chunks if the bandwidth decreases. This switching occurs seamlessly, typically between video chunks, to avoid interruptions in playback.

This approach offers several benefits:

- **Improved User Experience:** Users experience fewer interruptions and buffering, as the video quality is adjusted in real-time to match the available bandwidth.
- **Optimal Use of Bandwidth:** By delivering the highest possible quality that the network can support at any given moment, ABR maximizes the use of available bandwidth without overwhelming the network.
- **Compatibility Across Devices:** ABR ensures that video playback is optimized for a wide range of devices with varying capabilities, from smartphones to high-definition TVs.

ABR is commonly used by popular streaming services to deliver high-quality video content globally, regardless of network conditions [13]. By adapting to changing conditions, ABR provides a resilient and flexible approach to video streaming that puts availability for the client first, optimal for VoD.

Chapter 3.

Solution Strategy

After establishing more familiarity in the domain of video streaming, as well as identifying certain challenges to be solved, it is now time to start developing solutions. This chapter will be separated into two parts: The first section reviews the current project goals and what problems need to be solved, based on the inputs of the previous chapters. The second section consists of an evaluation of concrete strategies on how to achieve these goals.

3.1. Overview

This project has three main goals, as initially introduced in chapter 1:

1. **Distributed Service:** The final product operates as a distributed system with the ability to scale according to the current demand.
2. **Streaming Protocol Implementation:** The developed software implements a streaming protocol.
3. **Automatic Conversion to Required Format:** In order to support the streaming protocol, uploaded media is automatically converted to the required format on the server side.

Each of these goals involves different problems and decisions that need to be addressed and solved in order to successfully begin the software development process. As some of these decisions will significantly influence the software architecture, it is important to come to conclusions early and document these choices in order to retrace the decision at a later point [38].

3.1.1. Distributed Service

Cloud native applications usually fall into one of two categories: *Stateful* or *stateless*. These terms refer to how the application handles state and whether it prioritizes consistency or availability [5].

Stateful A stateful application maintains an individual state per instance of the service itself. This is typical for applications like databases and shared caches, where each instance of the service has meaningful state that needs to be consistent across the system.

Another property of a stateful application is *identity*. Since only the specific instance itself can provide its state to the other system participants, it also needs to be addressable in a consistent manner (e.g., consistent IP address or DNS name). This is also the origin of the term *instance*, highlighting the association of an identity.

A stateful application prioritizes consistency over availability. While multiple instances can increase resilience to failure, there is only every one instance that holds a specific state [24].

Stateless A stateless service on the other hand does not maintain a state by itself. If there is a state, it is either managed by a backend service (e.g., a shared database) or is provided with the request to that service (e.g., JSON Web Tokens (JWT)), or a combination of both.

An instance of a stateless service does not have an identity, it does therefore not matter which instance replies to a request. This is not only a property, but a highly important condition when building a stateless service. Thus, the term *replica* is used to refer to a copy of a stateless service, to highlight the lack of an identity. While this condition may enforce some restrictions, it is a very powerful property to have during the operational phase of a service, allowing replicas to be added and removed without regard to consistency [15], [32].

Stateless applications prioritize availability above all. If a replica is not deemed healthy, it can be replaced. If more or fewer replicas are needed, their number can simply be adjusted without concern [24].

By detaching state management to either the backend or the request itself, a stateless service can add and remove replicas without regard to consistency or identity (See Figure 3.1). With seamless scalability of the service as a primary objective for the software

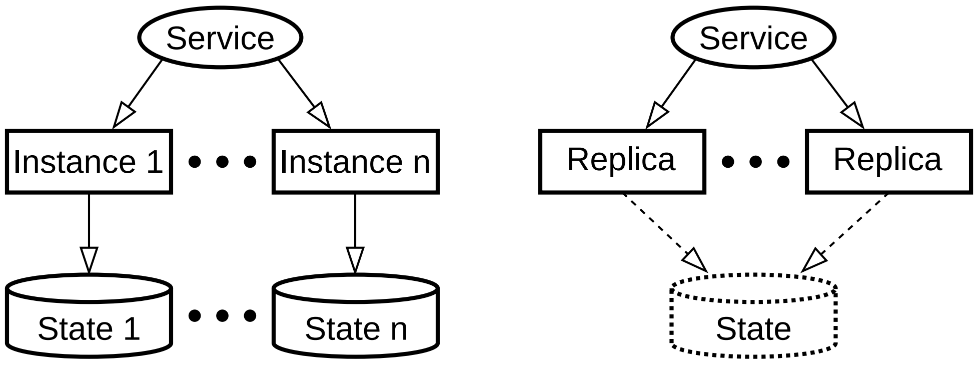

Figure 3.1.: Comparison of a stateful service's instances with their state and a stateless service and its replicas.

developed during this project, a *stateless* application has to be achieved.

3.1.2. Streaming Protocol

With video streaming being the main functionality of the software developed during this project, it is crucial to reason about the opportunities and risks of various approaches to video streaming. These factors will significantly influence decisions regarding what core values the streaming capability of the software should represent. The key objectives to be accomplished are as follows:

Client Independence Ensuring client independence is crucial for the flexibility and adaptability of the streaming software. Playback of content provided should not be restricted to a single client software, but should instead be available to all clients implementing the chosen streaming standard. This not only provides much more freedom of choice regarding the deployment pattern to the user of the software, but also promotes innovation and product diversity by allowing third-party client implementations. By adhering to this principle, the software can cater to a broader audience and support a wide range of devices and platforms [13].

Quick Loading Times A frustratingly slow start to video playback can deter users and negatively impact their overall experience. Seufert et al. [27] suggest that loading times greater than a few seconds can lead to increased abandonment rates, emphasizing the need for efficient and rapid content delivery. While there are many other factors that influence Quality of Experience (QoE), stalling (waiting for playback to start or interruptions of active playback) is considered the worst degradation of

QoE [27]. Optimizing the initial load time is therefore a priority to ensure user satisfaction and retention.

Client Driven Decisions In addition to promoting client software independence, clients should decide the quality of the stream if multiple qualities are available. Empowering clients to make these decisions based on their current network conditions and device capabilities enhances the overall user experience and ensures optimal performance. This approach aligns with user-centric design principles, where the needs and preferences of the end-user are prioritized in the software's functionality [27].

Adaptive Bit Rate Streaming Bringing together the best of both variable and constant bit rates, adaptive bit rate streaming offers both high-quality encoding for complex video sections and predictable bandwidth requirements. It is the foundation upon which reliable ABR can be built, allowing for dynamic, client-decided adjustments to the video quality based on the viewer's current network conditions, providing a seamless viewing experience.

Providing users with multiple qualities of the same content also enables endpoints in less performant or unreliable network environments to stream content. This adaptability ensures that all users, regardless of their connection speed, receive the best possible viewing experience without unnecessary buffering or interruptions [27].

High Compliance Finally, the bedrock for a *client-first* approach is the highest possible compliance level with the chosen video streaming standard.

Achieving high compliance guarantees that the software will work seamlessly with a wide range of client applications, ensuring broad compatibility and reducing the risk of playback issues. This commitment to compliance reflects a dedication to quality and reliability, reinforcing the software's reputation as a robust and user-friendly solution [14].

These qualities will be the primary influences on the decision-making process regarding video streaming going forward.

3.1.3. Automatic Conversion

In order to ensure that media uploaded to the developed service is in the correct format required by the chosen streaming protocol, an automatic conversion task should be initiated whenever media is uploaded. This approach mitigates the problem of expecting the

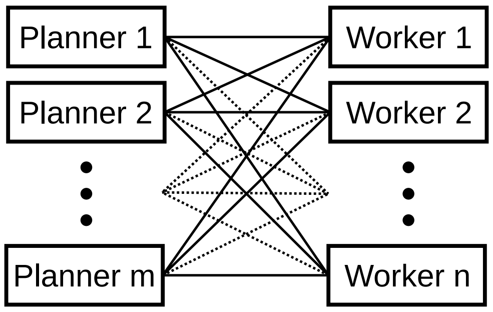

Figure 3.2.: The emerging connectivity pattern as the number of replicas of a service increases.

user to convert the media properly before uploading, which would be highly error-prone.

The read and write workloads of the service are expected to be asymmetric, as users will either be uploading media or streaming it [15]. Also considering that video conversion is a resource-intensive process, the decision was reached that a secondary service that can be scaled individually will be responsible for converting uploaded files [2].

This pattern is referred to as a *Distributed Job Scheduler* by the community, describing an architecture where tasks planned by one system (*planner*) are executed by other systems (*worker*) [18]. By adopting this architecture, three additional challenges need to be solved:

m-to-n Distributed Job Scheduler The expected workflow of a distributed job scheduler looks as follows:

1. A planner dispatches a job to the worker.
2. The worker executes the job. After completing the job, the worker informs the planner of the completion, including whether it was successful.
3. The planner processes that result.

In the current scenario, both planner and worker are made up of an arbitrary number of replicas, meaning m planners need to talk to n workers. Thus, their coordination becomes more difficult:

- Any planner must be able to schedule a task.
- Every worker must be able to handle it.
- All planner replicas must be able to process the result.

Figure 3.2 illustrates the emerging problem when there is an unknown number of

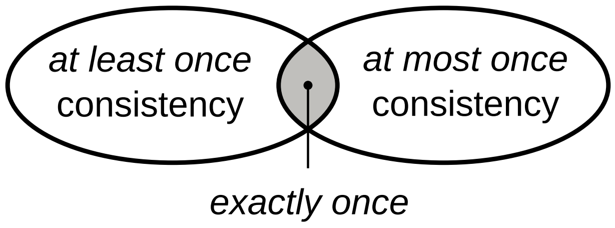

Figure 3.3.: A Venn diagram showing the nature of exactly-once semantics.

replicas, as $m \cdot n$ connections are needed in order to ensure that any replica of the planner group can communicate with any replica in the worker group [5]. It can therefore be concluded that an additional service must be responsible for handling the state of the tasks exchanged between planner and worker.

Temporal Decoupling It also needs to be considered that either group (planner or worker) may be unavailable during this exchange of state. A worker might not be available right away when a video needs to be converted, and similarly, a planner might not be present to receive a result.

This asynchronous behavior would also need to be handled by a mediator between the two groups or replicas, as there is state which has to be preserved.

The *Exactly-Once* Problem In the context of job scheduling, losing a job is costly, as this would mean that a video that a user has uploaded is not converted. Consistency is therefore a desired property. Similarly, performing a job more often than required is also expensive, as video conversion is a process that requires significant computing resources. This *uniqueness* can also be categorized under consistency, although stemming from the opposite end.

In combining both the *at least once* and *at most once* properties of this conversion job delivery, their intersection results in a third property: *Exactly once* (See Figure 3.3). This is the feature that has to be achieved in order to provide a consistent scheduling of video conversion jobs. *Exactly once* semantics are a common problem in distributed systems, which means that many solutions already exist [5], [18]. This is usually done by an intermediary service or system between planner and worker, as also discovered in the other two problems.

3.2. Technology Evaluations

Having created a detailed overview of goals to achieve and challenges to solve, more specific, architecturally relevant decisions can now be made, which will be difficult to change later. It is important to note that these decisions should encompass generic concepts, not concrete implementations and specific products or vendors. Switching between different products that serve the same function is easier than having to change the whole paradigm itself [38].

The primary decision drivers will be features and familiarity with the general paradigm. However, it is also crucial to consider the maturity and community support of the technologies being evaluated. Technologies with active development and a strong user base are more likely to be reliable and to offer a wealth of resources available for troubleshooting and optimization. Evaluating the compatibility of these technologies with other components in the system is also a vital step, as seamless integration can significantly reduce implementation time and effort.

3.2.1. Video Streaming

There are different video streaming technologies, each with their own strengths and weaknesses. This subsection will introduce some common video streaming standards and review their compatibility with the core values of this project, introduced in the previous section. Then, an informed decision can be made, in a process that includes requirements, possible shortcomings, and opportunities. While this decision is less general than the others in this section, having to change it later on will be equally hard as switching between paradigms in the other cases.

The video streaming standards reviewed for this project are:

DASH (Dynamic Adaptive Streaming over HTTP)

DASH is an international standard developed by the Moving Picture Experts Group (MPEG). It is designed to enable high-quality streaming of media content over the internet delivered by conventional HTTP web servers. DASH adapts the video quality based on the current network conditions, ensuring smooth playback even in fluctuating network environments [13].

Key features of DASH include:

- **Adaptive Bit Rate Streaming:** DASH is a prime example of using ABR. By providing different formats in multiple sizes and bit rates on the server side, it allows the client to switch between different quality streams based on network conditions.
- **Wide Support:** Compatible with a variety of devices and browsers, as it is codec-agnostic and supports different audio and video formats.
- **Segmented Media:** Media is divided into segments, making buffer control more granular and ensuring smooth playback. Unlike other standards, DASH does not split its streams into many small files but relies on *HTTP Range Requests* to request specific parts of a full file. This approach simplifies file handling, as there is a single file per stream, not thousands of small blocks that need to be managed.
- **Open Standard:** Being an open standard, DASH encourages widespread adoption and compatibility. There are many community projects and knowledge bases surrounding DASH, as well as numerous client implementations.

A notable subset of DASH developed by the WebM Project is called WebM-DASH. It focuses on using the `WebM` container format in combination with other open standards like the `VP8` and `VP9` video codecs and `Vorbis` and `Opus` audio codecs. This subset is most notably used by YouTube in their video streaming platform [36].

HLS (HTTP Live Streaming)

HLS is an ABR streaming protocol developed by Apple. It is widely used to stream media on Apple devices, though it is also supported on other platforms and is implemented by a diverse set of clients. HLS splits the overall stream into a sequence of small HTTP-based file downloads, each segment representing a short interval of playback time.

Key features of HLS include:

- **Chunked Streaming:** Video is divided into small chunks, allowing for adaptive bit rate streaming without the need to support additional request types on the server side.
- **Reliability:** HLS has built-in support for encryption and secure streaming, making it a robust choice for content protection. It is used extensively in the Apple ecosystem and is a key building block in many of their successful

technologies, such as AirPlay [9].

- **Live and On-Demand Streaming:** HLS is suitable for both live-streaming and VoD applications and is used in many internet video streaming services.

While there is less implementation work required on the server side with HLS, its approach involves managing numerous small files. This can lead to increased overhead and complexity in handling, as each segment is a separate file that must be managed individually [22].

MSS (Microsoft Smooth Streaming)

MSS is a protocol developed by Microsoft for adaptive streaming of multimedia content. It is designed to work with Internet Information Services (IIS) servers and Silverlight, a Microsoft framework for building internet applications. MSS provides a seamless streaming experience by adapting to changing network conditions in real-time.

Key features of MSS include:

- **Adaptive Bit Rate Streaming:** Automatically adjusts the quality of the video stream based on the viewer's network conditions. Client implementations communicate their capabilities as and current network condition to the server in order to receive a suitable stream.
- **Fragmented Media:** Media is divided into small fragments, allowing for smoother transitions between different quality levels. This is similar to the case of HLS, but the files are managed by IIS.
- **Integration with the Microsoft Ecosystem:** MSS is designed to work well with Microsoft products and services, such as IIS and Silverlight. In self-hosted environments it uses an existing IIS installation as its underlying web server and can be installed as a plugin. In cloud offerings, Microsoft used MSS in its Azure Media Service, which has, however, been retired[1].
- **Dynamic Manifest:** It uses a manifest file that is generated ad-hoc by the server according to client capabilities to deliver video segments dynamically. This enhances the streaming experience and reduces latency.

MSS is a powerful technology supported by many modern video players. However, it

[1]https://azure.microsoft.com/en-us/updates/retirement-notice-azure-media-services-is-being-retired-on-30-june-2024/

heavily relies on Microsoft technologies that are not available under an open-source software license, and can thus not be fully studied. With the retirement of their cloud offering, it is also unclear how the technology itself will progress [20].

Considering these standards, it was decided that Dynamic Adaptive Streaming over HTTP (DASH) will be used for the implementation of the video streaming platform. Specifically, the WebM-DASH subset will be implemented due to its proven reliability and large community support [36]. HLS requires storing each small video part of every quality in separate files, which would drastically increase file management overhead. MSS is also unsuitable due to its strong integration with other Microsoft products and the retirement of the Azure Media Service.

3.2.2. Metadata Storage

While video data will be the main content served by the developed platform, it will also provide a way of discovering said content. For this purpose, metadata further describing the available content must also be stored. This could be details ranging from a unique identification marker to more general information like a description provided by the uploader.

During the design phase, two potential solutions were evaluated for storing metadata:

Relational Database One approach is to store uploaded video details in a Relational Database Management System (RDBMS) such as PostgreSQL. The team has previous experience with relational databases, making it a familiar choice.

PostgreSQL: https://www.postgresql.org/

Advantages:

- *Easier to implement and lower maintenance*: Setup and operation of an RDBMS is relatively simple, especially considering existing experience.
- *Use of common SQL features*: Consistent IDs, indexing, and search features are already present.
- *High consistency and good indexing capabilities*: Relational databases ensure data consistency and integrity and provide robust indexing mechanisms to optimize query performance.

Disadvantages:

- *Scalability may be limited due to the transactional nature of RDBMS*: While relational databases excel at handling structured data and ensuring consistency, they might struggle with scalability issues as the volume of data grows.

  However, PostgreSQL-compatible installations like CockroachDB can mitigate this issue to some extend.

  CockroachDB: https://www.cockroachlabs.com/docs/

Metadata Indexing Another approach is to index existing videos on the platform using a search engine like Elasticsearch or ZincSearch. These systems are designed to handle large volumes of data and offer fast search capabilities.

Elasticsearch: https://www.elastic.co/elasticsearch

ZincSearch: https://github.com/zincsearch/zincsearch

Advantages:

- *Optimized for search*: Search engines are specifically designed to perform fast and efficient searches over large datasets, making it easier for users to discover content quickly.
- *Built for scalability*: Designed for horizontal scaling, search engines can handle increasing amounts of data and queries by adding more nodes to the cluster, thus accommodating platform growth [30].

Disadvantages:

- *Requires running an additional service*: Integrating a search engine requires additional infrastructure and maintenance, increasing the complexity of the system.
- *More overhead, less pragmatic*: Setting up and managing a search engine involves more overhead compared to using a relational database. As the primary focus of the project is not on enhancing search capabilities, this approach may seem impractical.

Both solutions have their advantages and drawbacks. However, it was decided to proceed with a relational database for this project, as the primary objective of this project is to achieve a working video streaming platform. Nonetheless, the implementation of metadata storage will be abstracted using a clean architecture approach, allowing it to be implemented using another method [17].

3.2.3. Storage

As streaming requires access to the video data, it must be stored in a fast and easily accessible manner. To achieve this, the following two storage options were evaluated:

File Level API Regardless of the platform on which the software runs, file system control is achieved through standard operating system calls. The specific file system can vary depending on the platform: If the streaming platform is running in a container, the file system may be provided by a shared volume mount. If it is running on bare-metal hardware, it may be provided by a file system stored on a built-in hard drive. Nevertheless, it can be accessed via simple OS-level calls.

- *Simple but efficient*: Issuing file system calls to the operating system may the simplest solution. However, considering the versatility that modern operating systems offer in terms of what can be used as a file system, it is by no means limiting. The possibilities range from shared network folders to a *ReadWrite-Many* shared volume in Kubernetes [24].
- *Performance dependent on platform*: While this approach involves less overhead compared to complex remote calls, it relies on the file system provided by the platform. This could potentially lead to performance issues or conflicts with other concurrent services running on the same platform.

Object Storage A modern approach to file storage is to use object storage, where files or parts of files can be quickly retrieved over a communications network (e.g., using HTTP requests). While there are many pure cloud offerings for this like Amazon S3, there are also API-compatible, self-hosted solutions available, such as MinIO [24].

Amazon S3: https://aws.amazon.com/s3/

MinIO: https://min.io/

- *Feature-rich and modern*: Object storage solutions are widely adopted in modern infrastructures for their scalability and comprehensive feature sets, including robust access management and seamless integration with cloud providers.
- *File access via network*: Accessing files over a network enhances scalability by enabling resource distribution across multiple locations. However, this approach may introduce latency and requires intricate setup, potentially rendering it less resilient to network issues.

Based on these considerations, the decision was made to adopt file level APIs for their rapid stability and sufficient flexibility. This choice also simplifies local deployments by avoiding additional dependencies for bare-metal setups. Furthermore, the storage implementation will be abstracted to allow for alternative implementations to be developed and integrated [17].

3.2.4. Job Scheduling

Finally, there is the matter of solving the scalability of the distributed job scheduler for the automatic conversion of uploaded video files. To recap the challenges for the scheduler: A conversion job should be delivered and completed *exactly once* to ensure that (1) no jobs are lost leading to unstreamable videos and (2) computing resources are used efficiently without redundant conversions.

The following possibilities were evaluated further:

Transactional Database The first option utilizes a database with strong consistency, typically ensured through transactions [18]. The workflow (using the terminology from subsection 3.1.3) is as follows:

1. The *planner* adds a job to a table using a consistent transaction to guarantee the job is added exactly once.
2. Each *worker* continually checks the table for *new* jobs. Upon finding a new job, it attempts to transition its state to *in process* using another transaction. If successful, the conversion process can be started on this worker.
3. Once the conversion process is finished, the *worker* attempts to mark the job as *finished* using a transaction.
4. Each *planner* also continually checks the table but for *finished* jobs. When a finished job is found, the video is published by the *planner*.

While this approach is viable, it introduces significant overhead. Both sides must continually monitor the job table, potentially straining the database. This could be especially problematic when scheduling numerous jobs in rapid succession or when service replicas increase.

Message Queue The most widely adopted and generally recommended solution to the *exactly once problem* is to use a messaging queue that supports this type of delivery (also called *exactly once semantics*) [18]. While this resembles the transactional database method, it uses acknowledgments similar to TCP in order to distribute jobs across a fleet of waiting workers.

The workflow looks as follows:

1. The *planner* publishes a job to the message queue.
2. A *worker* subscribed to the queue receives the job, *acknowledging* it. It then starts the conversion to processes the job.
3. Upon completion, the worker sends a result to the message queue, including the success status of the job.
4. The planner, also subscribed to the queue, receives the result to perform final steps, such as publishing the video. As only one *planner* can acknowledge a result, this prevents redundant work.

Based on that, the decision was made to go with a message queue to handle job scheduling. Not only is it the recommended approach, but there are many existing solutions in the cloud native ecosystem that solve this exact problem. Additionally, this still allows for temporal decoupling as the queue will only discard the jobs once they are acknowledged.

Given that it is an additional system, two additional requirements apply to this decision:

1. The use of a message queue must be optional to maintain the possibility of a *minimalistic deployment*, as desired by the main stakeholder (see chapter 1).
2. The implementation of video conversion job scheduling will be abstracted, allowing for future replacement with alternative implementations [17].

Chapter 4.

Implementation

Having established the project goals and decided on strategies in the preceding chapter, we now transition into the practical implementation phase. This chapter provides a comprehensive overview of the key technologies, architectural decisions, and design methodologies employed in the software development process.

The first section offers a detailed explanation of DASH, the selected ABR streaming protocol, which is integral to the platform's functionality. Next, specific technology choices are addressed, providing detailed reasoning to justify each selection and ensure an optimal implementation. The third section documents the architectural design and decisions, highlighting the considerations for scalability and efficiency. Finally, a retrospective evaluation of the thought process and solutions identified during the implementation is presented.

4.1. Dynamic Adaptive Streaming over HTTP

In the previous chapter, DASH was selected as the chosen ABR streaming protocol. As the primary technology used for developing the streaming platform, this section is dedicated solely to the explanation of how DASH works in detail, since the goal of the other components is to facilitate its operation.

4.1.1. HTTP Range Request

The Hypertext Transfer Protocol (HTTP) is used to transport data between web servers and clients, with a request being initiated by the client to retrieve the requested resources from the server.

In order to understand DASH, *HTTP range requests* are an important prerequisite. Range requests allow clients to request only a portion of a resource. This can be particularly useful for fetching only parts of a large file. In the case of DASH, this is what is used to stream video files by transferring small parts, not the entire file.

The client specifies the desired range using the `Range` header in the HTTP request, a possible syntax being [4]:

```
Range: bytes=start-end
```

In this example,

- `start` indicates the starting byte position.
- `end` indicates the ending byte position (inclusive).

For example, to request the first 500 bytes of a resource, the client would send:

```
GET /path/to/resource HTTP/1.1
Host: example.com
Range: bytes=0-499
```

The server responds with a `206 Partial Content` status code and includes the `Content-Range` header to specify the exact range being returned:

```
HTTP/1.1 206 Partial Content
Content-Range: bytes 0-499/12345
```

Here, `12345` is the total size of the resource [4]. By enabling a client request a portion of a resource, HTTP Range Requests are a key building block of DASH, allowing for the copying of segments.

4.1.2. Overview

Using HTTP range requests, it is possible to stream files in a controlled manner. In order to achieve successful video streaming, DASH employs the following sequence (see Figure 4.1 for reference):

1. Content Preparation (Server)

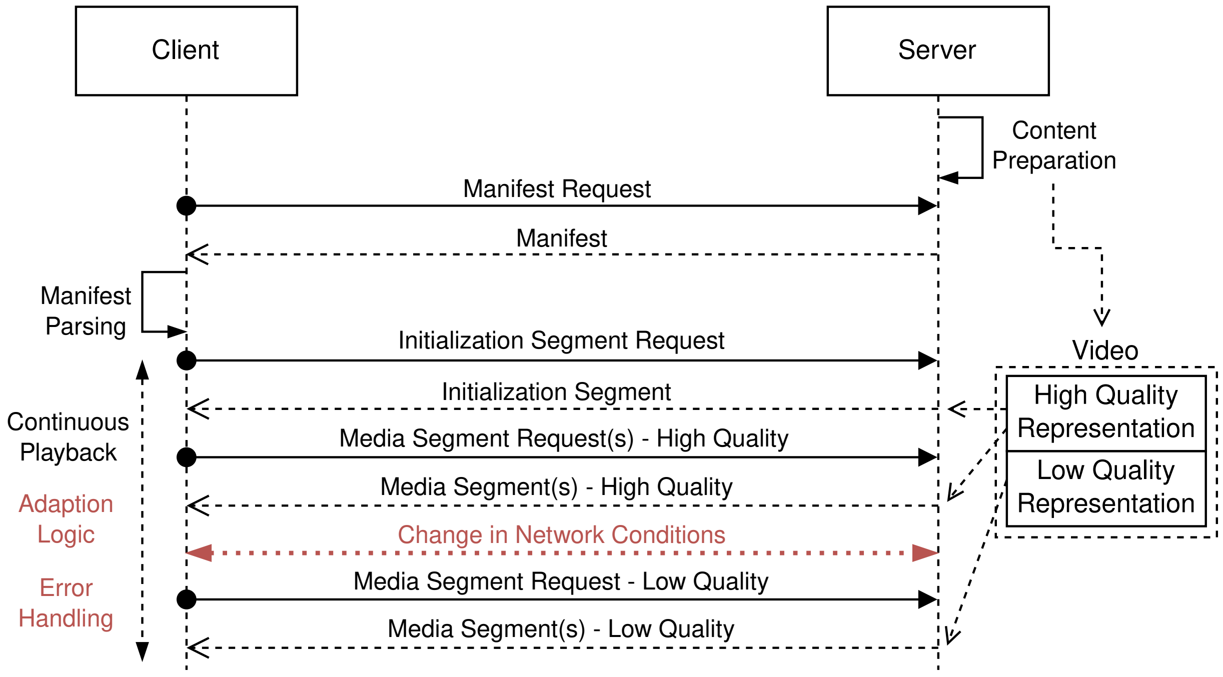
Client
Server
Content Preparation
Manifest Request
Manifest
Manifest Parsing
Video
Initialization Segment Request
Initialization Segment
High Quality Representation
Low Quality Representation
Continuous Playback
Adaption Logic
Error Handling
Media Segment Request(s) - High Quality
Media Segment(s) - High Quality
Change in Network Conditions
Media Segment Request - Low Quality
Media Segment(s) - Low Quality

Figure 4.1.: A sequence diagram illustrating the DASH workflow in order to achieve a video stream.

- *Encoding*: The video content is encoded at multiple bit rates and resolutions, generating different representations. Segments corresponding to key frames and a segment index containing this information are also added to the video file.
- *Manifest Generation*: An MPD (Media Presentation Description) file is created, which describes the different files and representations.

2. Initialization (Client)
   - *Manifest Request*: The client requests the MPD file from the server.
   - *Manifest Parsing*: The client parses the received MPD file to understand the available streams and representations.
3. Adaptive Streaming (Client)
   - *Initialization Segment Request*: The client sends an HTTP request to retrieve the initialization segment for the selected representation. This special segment informs the client how to play the video.
   - *Media Segment Request(s)*: The client requests subsequent media segments

  based on the current network conditions and buffer status. The client uses *HTTP range requests* to only request a specific segment of the representation.
- *Playback*: The client decodes and plays back the media segments as they are received.

4. Adaptation Logic & Error Handling (Client)
   - *Monitoring*: The client continuously monitors metrics such as the buffer status, playback smoothness, and network conditions. If a file transaction fails or network conditions change, it can trigger necessary actions.
   - *Bit Rate Adaptation*: The client dynamically switches between different representations (bit rates and resolutions) to ensure smooth playback and optimal quality.
   - *Retry Mechanism*: If a segment download fails, the client may retry the request or switch to a different representation. It may also implement fallback strategies to handle network errors or disruptions, ensuring continuous playback.

Initially, the server must invest compute resources in order to prepare the video files for streaming. Afterward, it needs to serve the files in a way that supports HTTP range requests, with the client managing ABR streaming and error mitigation.

4.1.3. Manifest

The video stream manifest, also called MPD (Media Presentation Description), is an XML file that describes the media content, including available qualities, codecs, and segment information. To stream a video, a client application first retrieves the manifest, which it uses to determine where to obtain the actual video data.

The XML schema in the MPD follows a predefined hierarchy. Each element can occur multiple times and can itself have multiple children (noted using ">"):

```
MPD > Period(s) > AdaptionSet(s) > Representation(s) > Segment(s)
```

Periods and segments come after each other, describing differences in time. Adaption sets and representations run in parallel, describing differences in content (either in adaption or quality) that is played (see Figure 4.2 for reference) [13].

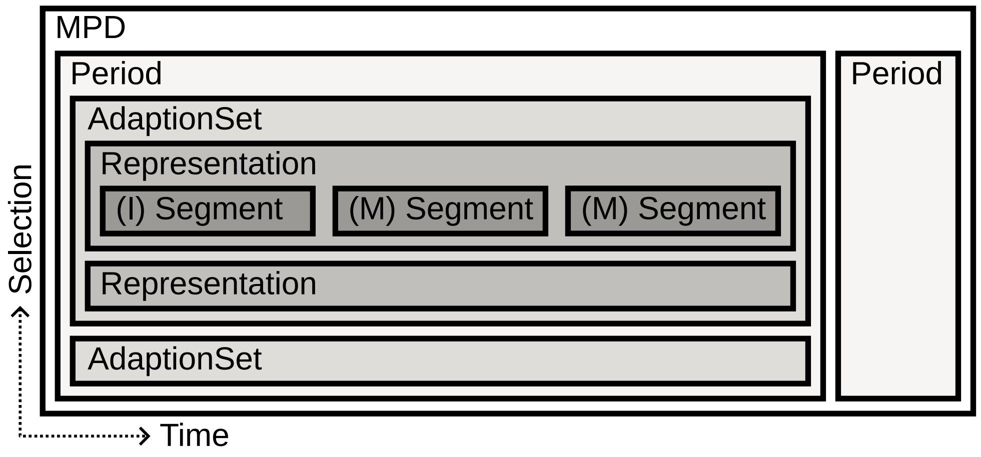

Figure 4.2.: DASH manifest hierarchy structure.

Period A Period is a time segment of content that can contain multiple adaptation sets. Periods allow for changes in content, such as advertisements or different episodes and parts of a series.

Adaptation Sets An adaption set groups a set of media content component representations. A common split is to have one adaptation set for audio streams and another for video streams. Different languages of content are also separated using adaption sets.

Representations Specific versions of media content within an adaptation set, characterized by bit rate, resolution, and codec. A client may switch automatically between different representations, as there should be no difference between them except quality, unlike with adaption sets where language might change.

Segments Segments are the actual blocks of data that are requested by the client. There are two types of segments:

- *Initialization Segments*, which contain initialization data for the media stream, allowing the client to interpret the following segments properly.
- *Media Segments*, smaller chunks containing the actual video or audio data. They are served individually, enabling for adaptive streaming.

In the XML-encoded form of a manifest, periods, adaption sets and representations are shown. The actual video segments are not shown, as they are the binary data which can be requested. Instead, a representation contains information on where the client can request video segments in a specific quality from (see Listing 4.1) [13]:

`BaseURL` Specifies the part of a URL where segments are stored, such as a file name.

`SegmentBase` Provides additional information about where the client can request segments. The field **indexRange** defines the bytes where the segment index is located, in the format that can be used directly in an HTTP range request. This segment index contains the location of all *media segments* in the representation. The **Initialization range** defines the bytes where the *initialization segment* lies, containing the metadata necessary to play the media segments.

```
...
<Representation id="0" ...>
    <BaseURL>file_eng144p15.webm</BaseURL>
    <SegmentBase indexRange="359103-359625">
        <Initialization range="0-499"></Initialization>
    </SegmentBase>
</Representation>
...
```

Listing 4.1: Representation block in an XML-encoded DASH manifest.

4.1.4. Usage

With a theoretical understanding of the key components in DASH, it is now time for the practical usage of it in the software developed during this project. Since most of the logic in DASH is controlled by the client, the choice of a robust client implementation is important for numerous reasons [14]:

1. *Development:* To develop a backend implementation that supports as many DASH clients as possible, having a client software that reports mistakes in the backend implementation is highly useful. This helps to avoid mistakes when programming thus aid software correctness during development.
2. *Compliance:* To ensure that the created software meets the qualities set by the DASH standard, a client that represents these values is required. While there are many production-grade clients available that focus on speed, the one used during development should prioritize correctness.

Video Player

On the client side, two options for the video streaming client have been evaluated for this project:

dash.js JavaScript Reference Client Developed by the DASH Industry Forum is the reference client implementation for DASH in the browser. Its main focus is on compliance with ISO/IEC 23009 (*Dynamic adaptive streaming over HTTP (DASH)*), the international standard [14].

Project: https://reference.dashif.org/dash.js/

It is licensed under the `BSD License`, making it open-source software.

Shaka Player This production-ready player, initially developed at Google, is now managed by the community under the Shaka Project. A key goal of the Shaka Project is to evolve the project and integrate modern web browser technologies as fast as possible.

Project: https://github.com/shaka-project/shaka-player

It is licensed under the `Apache License, Version 2.0`, also being open-source software maintained by the community.

Given these two choices, the *dash.js JavaScript Reference Client* was chosen for its focus on developer experience and implementation correctness, which aligns with the values of this project.

The dash.js ecosystem also offers various development tools and compliance checkers to verify whether the backend implementation meets the required standards:

- *DASH-IF Reference Player:* A hosted version of the dash.js reference client. This version has a graphical user interface for all possible options the client offers, which otherwise would need to be configured manually by loading the library and using JavaScript. It includes a built-in *Conformance Violations* report, showing warnings and errors if the backend implementation does not fully comply with the standard. By running completely in the browser, it can also be used to play streams running on the local computer, demonstrating the commitment of the DASH Industry Forum to providing valuable development tools.
- *DASH-IF Conformance Tool:* This tool checks the XML-encoded manifest for compliance with the standard. It validates the schema and reports on incorrect values

and violations of the standard. While it validates the manifest for MPEG-DASH, it has not yet been updated for the WebM-DASH subset used in this project.

Player Configuration

The dash.js reference player offers numerous configuration options covering the client's logical behavior and visual presentation.

An important decision regarding video streaming performance is the choice of algorithm the client uses to decide what quality a video segment should be requested in. The dash.js reference player offers three different modes for ABR streaming [14]:

Throughput The bit rate chosen by the client is based on the recent throughput speed (bandwidth) of the connection to the server. This method prioritizes reacting to changes in network quality.

BOLA The bit rate choice is based on the consistency of segments that have already been successfully downloaded (buffer). BOLA prioritizes video quality and performs the best at delivering high definition streams over low bandwidth [28].

Dynamic This mode combines the *Throughput* and *BOLA* modes and tries to achieve the best video quality while still being able to react to changing network conditions.

For this software, the *Dynamic* operation mode has been chosen. It provides a good balance between quality and reliability, and is also the default configuration for the dash.js reference player.

Problems

DASH-IF Conformance Tool does not include WebM-DASH The conformance validation tool that can be used to check if a DASH manifest is valid does not yet support the WebM-DASH specification. While the *WebM-DASH Specification* defines the XML name space

`urn:mpeg:dash:profile:webm-on-demand:2012`,

the conformance tool does not recognize this as valid [36]. Since WebM-DASH is a subset of the standard, it should be included as well.

There is an open issue on the conformance tool source code repository:

Issue: https://github.com/Dash-Industry-Forum/DASH-IF-Conformance/issues/90

4.2. Technology Choices

Having made conceptual choices regarding general technologies in the previous chapter, it is now time to decide on specific implementations of those technologies. Reasoning about these choices is also an important part of the development process in order to guarantee a justified course of action.

4.2.1. Programming Language

As the decision of the programming language for a project also influences other choices based on the available libraries and ecosystem, it should be made early. A key factor is also any previous experience the development teams has with the specific language.

Java Widely adopted by many enterprises and online communities alike, Java is a strong choice. It is also the primary programming language taught at the Eastern Switzerland University of Applied Sciences (OST), so there is a large knowledge pool available in the case of difficulties.

Execution Model: Hybrid (JVM) / *Package Manager:* ✓

C++ Another popular and widespread language, in which the team has some experience.

Execution Model: Compiled / *Package Manager:* ✗

Rust Currently the most *desired*[1] language that many developers want to work with, in which the team also has previous experience.

Execution Model: Compiled / *Package Manager:* ✓

Go When looking at the *cloud native landscape* by the CNCF[2], Go is the language that is most represented. Ease of use, a strong standard library and fast development iteration speeds make it perfect for the creation of scalable web applications.

Execution Model: Compiled / *Package Manager:* ✓

[1]https://survey.stackoverflow.co/2023/
[2]Cloud Native Computing Foundation Landscape: https://landscape.cncf.io/

Go is a modern programming language that offers a clean and efficient syntax. Its robust standard library and strong performance make it ideal for developing high-performance server-side applications. Being specifically designed for the development of cloud native applications, and with the team having substantial experience with it, Go the best choice [30].

4.2.2. Messaging System

The message queue used to schedule video conversion jobs across a fleet of different service replicas will have a large impact on how the system will perform at scale.

NSQ Initially developed at Bitly, the *new simple queue* has grown into one of the most liked projects on GitHub. It focuses on being a very performant *event store* that supports the *publish-subscribe* model. It also has first-class support for Go, providing official libraries.

Project: https://nsq.io/

Apache Kafka An event store adopted by many enterprises. There are no official Go libraries available, but there are some by third parties.

Project: https://kafka.apache.org/

NATS A project endorsed by the CNCF that implements both event store (PubSub) and message queue (JetStream) modes. It also has an official library for Go and is itself written in that language.

Project: https://nats.io/

Apache ActiveMQ A message queue focused on correct order of delivery. The team has previous experience with this product, although while using Java as the programming language. There are no official Go libraries available.

Project: https://activemq.apache.org/

To maintain consistency in the technologies used throughout this project, NATS was chosen as the messaging system for conversion job scheduling. It is a modern and well-established software project and offers high consistency with exactly-once delivery, as defined as required in the previous chapter.

In a deployment where no external video conversion instances are running, conversion jobs will run on the replicas locally, without the need for a messaging system.

4.2.3. Database

To maintain a record of available videos on the platform, a robust database is necessary.

Similar to the messaging queue, there should be the possibility of running the software without depending on an external database, although this is feasible only for single-replica deployments.

MariaDB A solid database with which the team has previous experience. If the need for greater scalability arises, *Vitess* can be used, which allows for large-scale deployments.

Projects: https://mariadb.org/ and https://vitess.io/

SQLite A very popular lightweight, single-file database with a powerful set of SQL features. There are also pure Go client implementations available, allowing for a dependency-free, statically compiled binary.

Project: https://sqlite.org/

DuckDB A new but very promising project introduced to the team during the development process. It is a lightweight, single-file database similar to SQLite, although it currently lacks a CGO-free implementation of the Go client.

Project: https://duckdb.org/

PostgreSQL The team has a lot of experience with this database, as its very powerful SQL dialect is the primary one taught at the Eastern Switzerland University of Applied Sciences. In the case an even more scalable solution is required, *CockroachDB* is a wire-compatible replacement designed for cloud native deployments.

Projects: https://postgresql.org/ and https://cockroach.io/

Due to familiarity with the technology, PostgreSQL was chosen for the large-scale deployment, while SQLite will be used for single-replica setups. The two also offer a very similar feature set in their SQL dialect, minimizing development overhead. DuckDB would have been the preferred choice for single-replica setups if a dependency-free implementation were available.

4.2.4. Storage

Since the decision was made to use a file-level API instead of an external storage provider, the underlying implementation of the storage does not matter. This evaluation is therefore mostly for completeness, as any solution could be used. The decision is left to the user of the software.

Local Storage The straight-forward way to provide storage is to simply let the software access it locally. When running in an environment where containers are preferred, other abstractions like Volumes can be used to persist storage [24].

Ceph A failure-resistant, highly scalable and distributed storage system that monitors its own state and repairs itself in case a replica of data is lost. In a cloud native environment, *Rook* is a well-established operator that manages Ceph.

Projects: https://ceph.io/ and https://rook.io/

iSCSI Another option is to create a deployment in which every participating computer provides storage to a large pool available to the running resources. In a cloud native setting, this could be achieved using software like *Longhorn*.

Projects: https://longhorn.io/

4.2.5. Video Conversion

Various options can be used to convert uploaded video files to the format required by DASH. This decision impacts both development effort and compatibility.

From Scratch Writing a video conversion program without using any prerequisite would be possible. However, due to the limited scope of this project, this is not a feasible choice. Additionally, this approach would shift the focus from building a distributed video streaming platform to building a video conversion software.

GStreamer Library Bindings GStreamer is a powerful multimedia framework, that provides libraries for many platforms. Using its libraries, a video conversion component could be implemented, although this would also require a high development effort, as the team lacks experience with GStreamer.

Project: https://gstreamer.freedesktop.org/

FFmpeg Binary FFmpeg is a popular, cross-platform tool with support for most media containers, codecs and technologies. It is highly configurable and also able to generate DASH manifests.

By building a wrapper around the command line program, a video conversion component could be implemented. The team has a previous experience with FFmpeg.

Project: https://ffmpeg.org/

FFmpeg Library Bindings Another possibility is the use of the FFmpeg libraries instead of the GStreamer libraries for an implementation. There are projects within the Go ecosystem that offer this functionality, but their feature set does not currently meet the minimum requirements for this project.

Creating an FFmpeg command line program wrapper is the most viable solution, as it clearly separates concerns (e.g., argument parsing and conversion handled by FFmpeg) and lets the team apply previous experiences due to the similar interaction.

Encoder Settings for VoD

In order to achieve ABR streaming capabilities, the platform has to provide video streams at different bit rates. There are many different variants in which video resolution, fps, bit rate, conversion speed and other factors could be combined to come up with a set of streams to provide as a platform.

For this project, we will use Google's *Recommended Settings for VOD*. To achieve a good balance between quality and encoding speed, these recommendations aim to accommodate a wide range of content types while minimizing the bit rate required to achieve a visually good quality. This approach ensures that the encoding process is efficient while still producing high-quality output that can handle diverse types of content [25].

These values align well with the goals of the project and can be adjusted later if necessary.

4.3. Design and Architecture

This section documents the architectural design decisions made while building a highly scalable video platform. Some of the parts are adopted from the *arc42 documentation template*, a well-established template for software documentation in the software engineering community [29].

The project team has settled on the name *GoReeltime* for the software developed over the course of this project. It is the goal of this name to reflect the key values of the software, being a fast, almost *real-time* streaming service, with the *reel*-part of the name in reference to old film reels used in early cinema.

4.3.1. Overview

GoReeltime delivers a video streaming platform with a high performance and flexibility to the operators. The most important quality goals can be summarized as follows:

Horizontal Scalability With the evolution of cloud computing and distributed systems architecture, the importance of *scaling out* by adding more computers instead of *scaling up* a single computer has become increasingly pronounced [15]. Therefore, GoReeltime is build as a stateless service that can be easily scaled up according to the current needs of the platform.

Ease of Adoption Many software projects struggle to achieve a high degree of adoption, as their operation can require complex setup procedures and additional supporting systems. It is possible to run GoReeltime in different sized deployments, from a large cloud landscape to a single, resource-constrained machine.

Streaming Performance Waiting for video playback to start or the interruption of a running video playback (referred to as *stalling*) is considered the worst degradation of quality in the context of user experience [27]. GoReeltime will provide a high degree of video streaming performance as well as quick response times, ensuring a pleasant user experience.

Smart use of Resources Despite the abundance and low cost of compute resources due to cloud computing, GoReeltime uses these resources efficiently [15]:

- A release version is shipped as a single distributable with minimal dependencies (e.g., no heavy underlying runtimes, statically compiled code).
- Loading large amounts of data into memory is to be avoided whenever possible.
- Resource-intensive jobs can be processed remotely by more suitable machines.

Client First GoReeltime provides a built-in video player, but any client supporting DASH can be used to play content on the platform. It serves as a backend providing video content, making it suitable for embedding on other websites.

The Twelve-Factor App As a cloud native service, GoReeltime follows the *Twelve-Factor App* methodology, a set of established practices when building web services that run in cloud environments [37].

Stakeholders

While the development of GoReeltime takes place in the context of a bachelor thesis, the following stakeholder is considered:

1. **Distributed Systems and Ledgers Lab:** The DSL at the Eastern Switzerland University of Applied Sciences (OST) seeks a new solution to publish their lecture recordings. This stakeholder is the primary source of the *Ease of Adoption*, *Streaming Performance*, and *Client First* goals for the project.

Development Process

The development of GoReeltime is divided into two different stages: *Core Functionality* and the *Proof-of-Concept* stage, with features related to core functionality taking the highest priority.

Core Functionality :

1. *Streaming*: A video can be streamed from server to client using DASH. Streamed refers to incrementally downloading small portions of the video and reassembling them on the client, not downloading the full file. The video files need to be present on the server in the streamable format.
2. *Scaling*: The server component can be scaled horizontally, resulting in a service consisting of multiple statless replicas. For this, state needs to be moved into database and file storage respectively.
3. *Uploading*: A video file can be uploaded to the server component from a client.
4. *Conversion*: Uploaded videos are automatically converted into the streamable format by the server. The scalability must not be lost in this step.

Proof-of-Concept :

1. *Frontend*: A simple frontend to manage uploaded videos is available. Since GoReeltime primarily provides a video streaming backend, a complex frontend application is out of scope.

2. *Authentication*: A system is in place that allows for the disabling of content modification.

4.3.2. Functional Requirements

This section documents functionality the platform needs to provide. These features are documented in the form of *user stories* in order to emphasize a user-centric approach to the development process [38].

🟩 **FR-01 – Content Overview**
As a user, I want to have an overview of available content so that I can easily navigate and find videos of interest.

🟩 **FR-02 – Basic Search**
As a user, I want to perform basic searches so that I can quickly find specific videos or content categories.

🟩 **FR-03 – Video Streaming**
As a user, I want to stream available videos seamlessly so that I can enjoy uninterrupted entertainment.

🟩 **FR-04 – View Metadata**
As a user, I want to view detailed metadata for each video so that I can understand its context, such as the title and description.

🟩 **FR-05 – View Thumbnail**
As a user, I want to see a thumbnail for each video so that I can get a quick visual preview of its content before deciding to watch it.

🟩 **FR-06 – Upload Video**
As a content provider, I want to upload new videos easily so that I can share my content with the audience.

🟩 **FR-07 – Uploaded Video Conversion**
As a content provider, I want the platform to handle various video formats upon upload so that I do not need to worry about compatibility issues.

🟩 **FR-08 – Video Metadata Upload**
As a content provider, I want to add metadata to my uploaded videos so that viewers can have detailed information and context about the content.

🟩 **FR-09 – Automatic Thumbnail Generation**
As a content provider, I want the platform to automatically generate thumbnails for my videos so that I have a default preview available without additional effort.

🟩 **FR-10 – Video Metadata Update**
As a content provider, I want to be able to update the metadata of my videos so that I can correct any mistakes or make improvements over time.

4.3.3. Non-Functional Requirements

Non-functional requirements focus on broader product attributes, like relating to performance or specific qualities [6]. They are usually harder to measure as they are not just present or absent like functional requirements.

🟪 **NFR-01 – Minimal Setup**
The platform should be deployable in a simple setup without relying on object storage, specific container APIs or heavy supporting services.

🟪 **NFR-02 – Maximal Setup**
The platform should be deployable in a highly scalable environment utilizing container orchestration.

🟪 **NFR-03 – Horizontal Scalability**
The software architecture of the platform should rely on statelessness where possible in order to enable horizontal scalability.

■ NFR-04 – The Twelve-Factor App

To promote modern deployment methodologies and maximum portability, the developed application should adhere to the Twelve-Factor App methodology wherever possible [37].

■ NFR-05 – API Usability

The HTTP API should be designed in a way that uses proper resource names and HTTP verbs. This is often referred to as *Level 2* of the *Richardson Maturity Model* for RESTful HTTP [26].

Example: `PUT /api/video/1337` updates the video with ID 1337 with the content contained in the request.

■ NFR-06 – Quality Attribute Scenario [38]: Handling of High User Load

Synopsis Performance of video streaming API in response to high user load. When considering the stakeholder *Distributed Systems and Ledgers Lab* the expected number of users for this requirement is around 100 simultaneous streams.

Business Goals Keeping the platform attractive to users by ensuring quick loading times for video streams.

Relevant Quality Attributes Performance (Response Time)

Scenario Components **Stimulus** A user requests a video segment of by sending an HTTP range request.

- **Stimulus Source** The user's web browser sends the request.
- **Environment** At runtime, high workload (100 active users), not under stress (external attack, system outages).
- **Artifact** Gateway, Network, Compute and Storage Resources, Orchestration System, System Architecture
- **Response** The backend is built for high performance in order to quickly respond even to non-cached requests.
- **Response Measure** The start of a video stream still conforms to NFR-07.
- **Questions** System is distributed, are there issues that occur at a very large scale?
- **Issues** How is the impact underlying hardware will have on system performance addressed during solution strategy?

■ **NFR-07 – Streaming Performance**
The action of starting a video stream manually conforms to the following agile landing zone [38]:
Minimal: 2s / Target: 0.75s / Outstanding: 0.3s

■ **NFR-08 – Testability**
Every software unit should employ the dependency injection pattern, promoting testability by mocking more complex or irrelevant components.

■ **NFR-09 – Persistence**
Content uploaded to the platform should be persisted in such a manner that a reboot of the system does not cause data loss except for caches.

■ **NFR-10 – Search Performance**
Searching for content happens in an indexed manner and does not rely on scanning every available video artifact on the platform.

4.3.4. Architecture Constraints

The following constraints apply to the design and development of GoReeltime:

Versioning GoReeltime is versioned using semantic versioning (SemVer), because it gives relevant meaning to version numbers, has a high degree of adoption and is easy to understand.

Reference: https://semver.org/

Programming Language GoReeltime is written in the Go programming language due of its first-class support for cloud native web applications [30].

The Go ecosystem includes a set of useful tools that are used during development. While the following decisions could be considered as constraining, they promote a streamlined and goal-focused development experience. These tools also integrate well with each other:

Code Documentation: `Godoc` Godoc-style comments are used in the source code where additional information is needed. This allows for automatic generation of documentation.

Reference: https://go.dev/blog/godoc

Test Framework: `go test` Go comes with a powerful, built-in test suite which is used for unit tests. Wherever possible, the concept of *table-driven tests*[3] is employed to achieve easy extensibility without additional logic.

Code Formatting: `gofmt` Go's built-in formatter is used for code formatting across the project to ensure a consistent style.

Code Linting: `golangci-lint` A bundler and runner or a very large list of supported linters in the Go ecosystem. It also produces code quality reports in many standardized formats, which can be used in other tooling, such as continuous integration pipelines.

Project: https://golangci-lint.run/usage/linters/

The project is checked using different categories of linters, ranging from typos to deep analysis using Go's powerful type system. See the project repository for more information.

Naming Convention: It is common best practice when writing Go to used single-word names for packages, which will be adhered to.

Examples: `config`; `enrichlet`

Small and Large Deployments GoReeltime is developed to switch between two types of database backend:

SQLite is used for single-replica deployments in resource-constrained environments.

PostgreSQL is used for deployments that require more than a single replica of the service.

Optional Remote Jobs For large-scale deployments, GoReeltime can use a messaging system to exchange jobs and results. However, this must be optional in a single-replica-deployment to avoid requiring additional, unnecessary resources.

[3]https://go.dev/wiki/TableDrivenTests

4.3.5. Architecturally Significant Decisions

In this section summarizes the decision made in section 4.2 and evaluates further and more fine-grained decisions that are hard to change later on. This includes choices of frameworks and paradigms that will have a significant impact on the development process.

Architectural Decision Records (ADRs) are captured in the form of *Y-Statements* in order to capture context, decisions and consequences concisely [38].

🟨 **ADR-01 – Video Streaming Standard**

In the context of choosing a video streaming standard, facing the need for client-driven adaptive streaming, we decided to use DASH and neglected HLS to use an open standard with individually playable video files, accepting that it has less community adoption.

🟨 **ADR-02 – RESTful HTTP**

In the context of web-based interaction, facing the need for ease of implementation and maintainability, we decided to use RESTful HTTP. This decision was made in order to achieve a standardized, resource-based interaction model that enable simple and flexible communication between client and server. We acknowledge that RESTful HTTP requires careful design and proper management of HTTP status codes.

🟨 **ADR-03 – Statelessness**

In the context of session state management, facing the need horizontal scalability, we decided to relocate application state to backend services, accepting that implementation complexity is higher.

🟨 **ADR-04 – Containers**

Containers will be used as the unit of distribution for the software, facing the need for different sizes of deployment. Using containers allows for different environment configurations while keeping the distribution unit the same.

■ **ADR-05 – Server-Side Rendering**
SSR (Server-Side Rendering) will be used for the web frontend implementation of the service, being that it is not part of the core components and can be disabled if not needed. This decision was made to keep interaction logic on the server side.

Web Server Performance Benchmarks

To make an informed decision on the web framework for the project, the performance of various options was evaluated while considering additional factors. The key objectives were:

1. The ability to handle HTTP range requests…
2. …and to do so effectively under high load.
3. High compatibility with the Go standard library to ensure ease of use.

The following Go libraries were evaluated:

Fiber Fiber was chosen because of its popularity and the possibility to use *preforking* (allowing multiple sockets to listen on the same address, enabling kernel-level load balancing).

Project: https://gofiber.io/

Gin Another popular framework that the team has previous experience with.

Project: https://gin-gonic.com/

Chi A newer, lightweight solution with full compatibility with the Go standard library (`net/http`). It does not use any custom types, unlike the other frameworks.

Project: https://go-chi.io/

The performance of these different options was evaluated using empirical measurements:

Each library was used to implement a minimal working example that supports HTTP range requests. Measurements were collected by continuously running a high number of randomly calculated (and therefore non-cacheable) requests against the example. A bare-metal, high-performance environment was used to gather meaningful performance data, shown in Figure 4.3.

The goal was to simulate a high number of clients requesting different parts of a file using

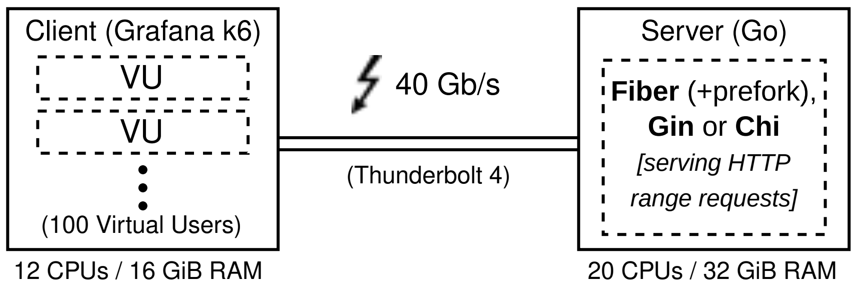

Figure 4.3.: The benchmark setup used to evaluate different Go web frameworks.

Option	Avg. Request Duration (ms)	Duration (s)	Data Received (MiB)
Fiber	11.77	30	190
Fiber (prefork)	9.7	30	230
Gin	33.53	30	60
Chi	4.31	30	828

Table 4.1.: Results of the Grafana k6 benchmark for Fiber, Gin and Chi.

HTTP range requests. To do this, *Grafana k6* (https://k6.io/), an open-source load-testing tool, was used. It can be configured as needed to simulate specific experiments.

Considering **NFR-06**, the tested scenario will consist of 100 concurrent users consistently performing random HTTP range requests over a time period of 30 seconds. The *Range* header is randomized, in order to prevent the server from simply caching previous requests, which could potentially falsify the results. The configuration used to achieve the results displayed in Table 4.1 is shown in Listing A.1.

Based on these experiments, Chi was chosen as the Go web framework for this project.

ADR-06 – Web Router Library

In the context of selecting a Go web router library, facing the need for high performance while keeping code maintainable, we decided to use Chi, in order to achieve standard library compatibility at a high performance rating, accepting that it does not employ *preforking*.

4.3.6. Building Block View

This section describes the interaction between the different components that make up the GoReeltime platform. This is done using the C4 model (https://c4model.com/) by Simon Brown to show these relationships on different levels of abstraction.

System Context

In the system context, the C4 model was not applied, as there are no explicit interactions with external entities except for the user. However, the system relies on some external dependencies:

User A user may be an operator of the platform, a content provider or a content consumer. Most of the computing performance required will be consumed by the content providers and consumers.

dash.js Reference Player The chosen DASH video player is embedded on the HTML frontend of the platform, meaning the user will need internet connectivity in order to download the player library.

FFmpeg The platform uses FFmpeg for automatic video conversion. Should the command line API of FFmpeg introduces breaking changes, the functionality of GoReeltime could be affected, potentially requiring additional software engineering efforts.

Container View

Inside the actual platform, different *containers* interact with each other. The term *container* is somewhat overloaded, as it refers to unique pieces of software communicate together in this context. But due to the distribution unit of GoReeltime being OCI-compatible containers (run by a container runtime), the terms describe the same units in this case [24].

Reeltime This is the main service, responsible for letting the user interact with the platform. It is responsible for presentation, business logic, job scheduling and content provisioning.

Reelconvert This service is responsible for the conversion of content on the platform (e.g., video encoding to other formats).

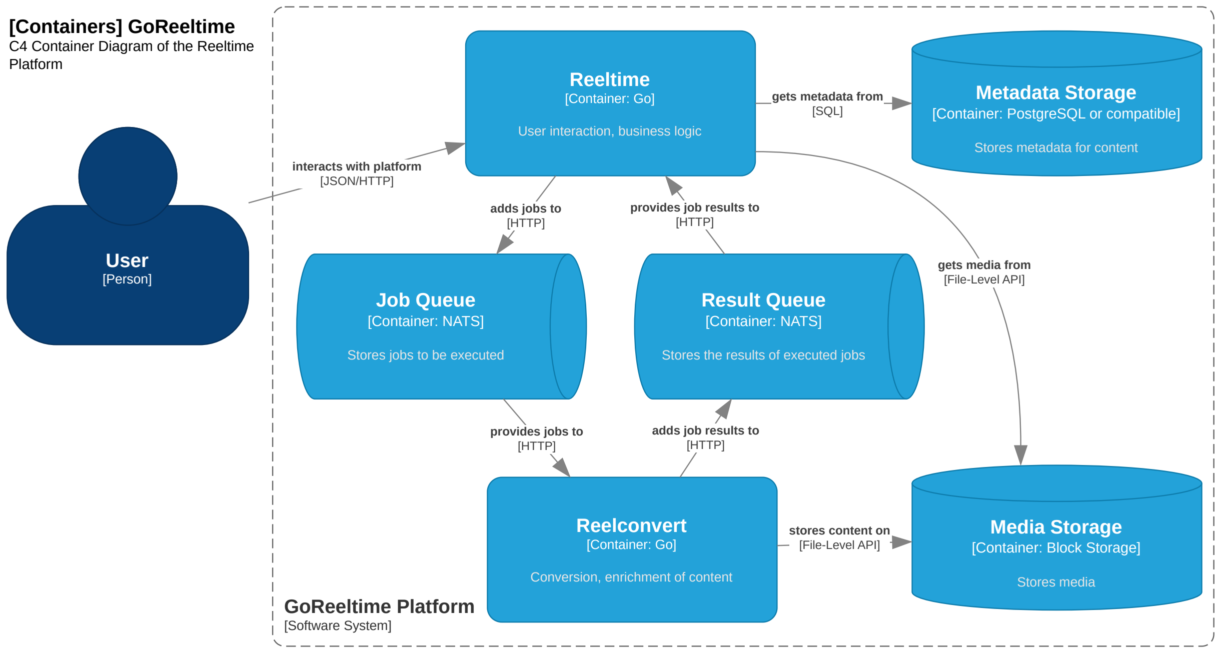

Figure 4.4.: C4 Container diagram of the GoReeltime platform.

Queue(s) In order to properly schedule jobs across multiple replicas of both Reeltime and Reelconvert, the messaging pattern using queues is used [38].

Media Storage This file-level API compatible storage allows the persistence of binary media files (video data).

Metadata Storage A database that satisfies the need for structured metadata that can be indexed in order to provide a fast overview of consumable content as well as responsive search functionality.

Figure 4.4 shows the relationship between the containers in a scalable setup.

Component View

In the C4 model, each container consists of components, different parts of the software that interact on separate concerns. Over the development period of GoReeltime, two different containers were built: Figure 4.5 shows the component interaction of the Reeltime container, while Figure 4.6 shows the interactions for Reelconvert.

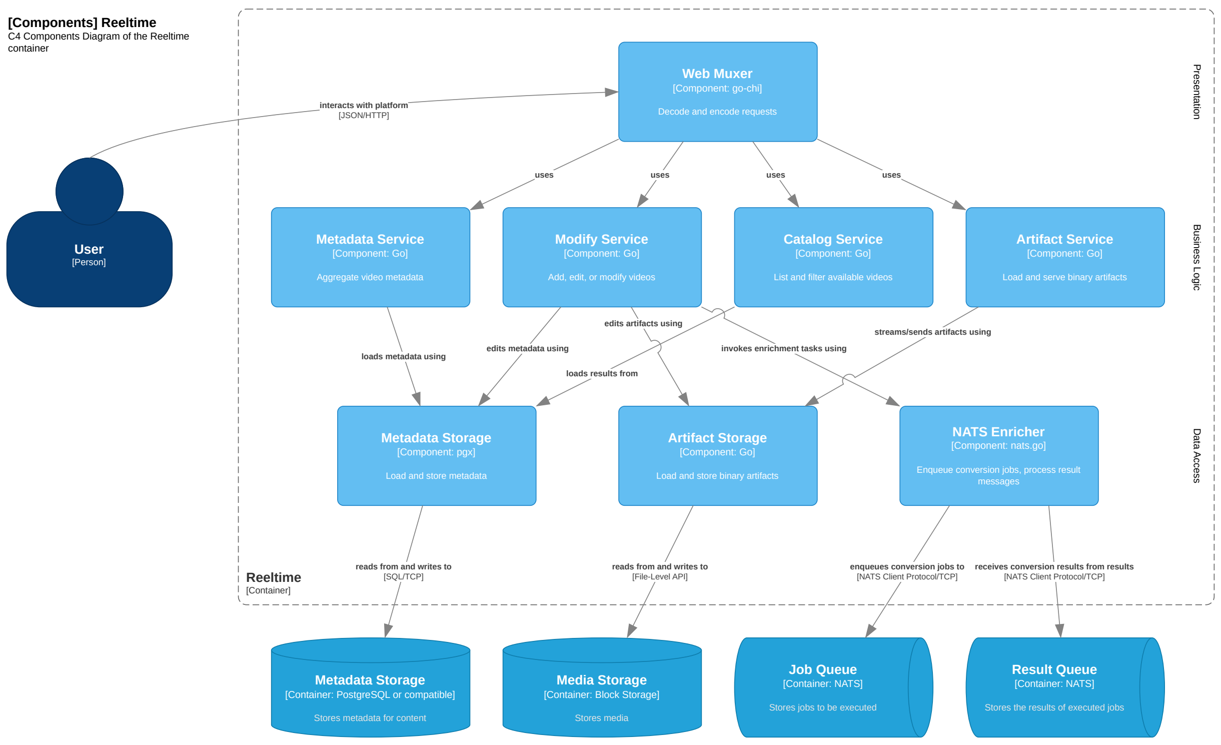

Figure 4.5.: C4 Component diagram of the Reeltime container.

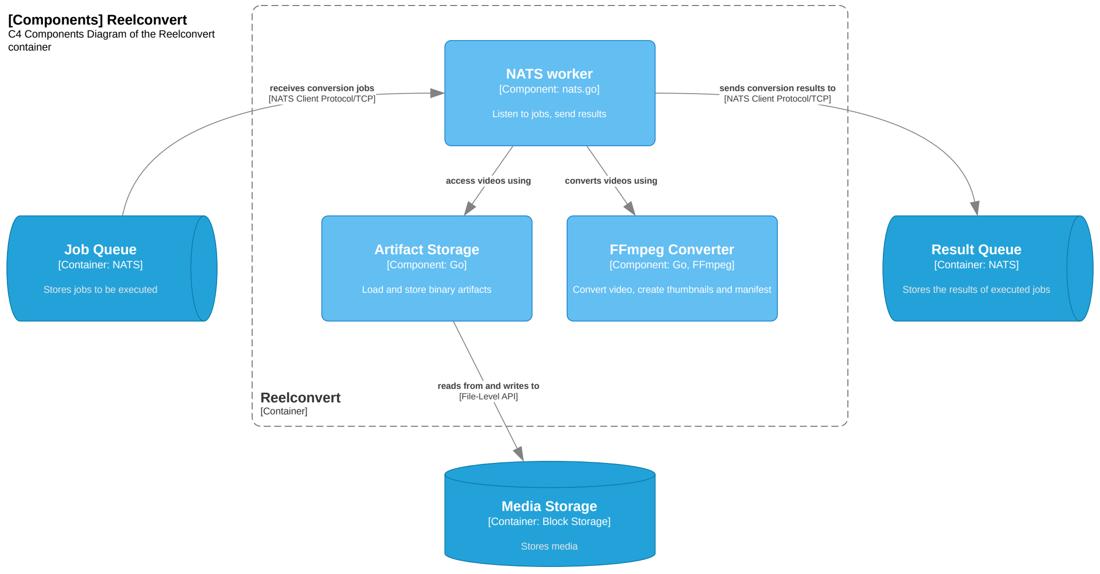

Figure 4.6.: C4 Component diagram of the Reelconvert container.

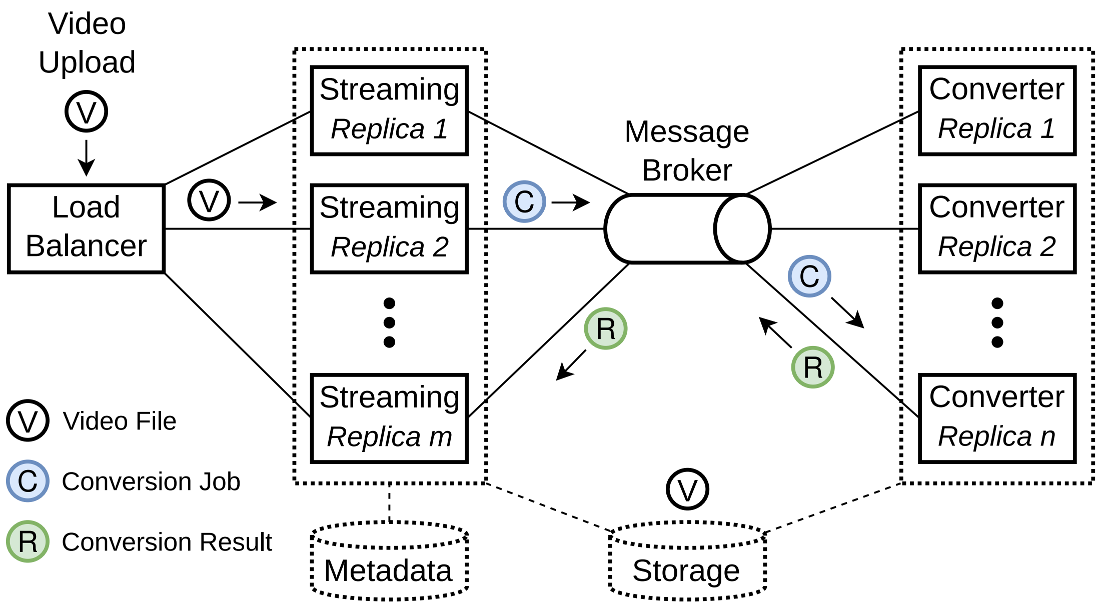

Figure 4.7.: Workflow of the automatic conversion process upon a video upload.

4.3.7. Runtime View

As some use cases of the system involve rather complex processes, this section aims to give a step-by-step explanation of the most prominent interactions and their effects inside the system.

Automatic Conversion

This use case addresses **FR-07**.

After a video file has been uploaded to the platform, it will most likely not be in the format required for DASH streaming. Consequently, it will not appear on the platform, as videos are not marked as *available* until they have been correctly encoded.

When a video is uploaded via HTTP, the following process takes place (see Figure 4.7 for reference):

1. Depending on the setup, a load balancer will decide which replica of the streaming service the video file will be uploaded to. Besides handling the upload request, the replica is responsible for claiming a unique video ID from the metadata store, which it will assign to the uploaded file.

2. After the upload request has been successfully completed, the same replica also dispatches a *job* to the messaging system (broker). This message will have the format

   ```
   job.<Video ID>
   ```

   , indicating that the file with this ID still needs to be converted.
3. As soon as a replica of the converter service detects that job, it will open a transaction to claim it. If it successfully claims and removes the job from the messaging system, a local video encoding job will be started on that converter replica. This can be done instantly, since the conversion service has access to the same storage the streaming service.
4. When the video encoding process is finished successfully, the replica sends a *result* message with the format

   ```
   result.<Video ID>.success
   ```

   to the message broker, indicating the conversion was completed without failure.
5. Similarly, if a replica of the streaming service detects a result on the messaging system, it will open a transaction to claim it. If it was able to claim and remove the result, it marks the video as *available* in the metadata storage, publishing it on the platform.

After this process has completed, a newly uploaded video is ready to be streamed on the platform.

4.3.8. Deployment View

GoReeltime is software that can be freely shared and distributed. It will not be running in just one single environment hosted by a *Software as a Service* (SaaS) provider, but many different types of environments, with individual requirements by the user. The goal of this section is to show two different configurations, each suited to its specific environment.

Single Container

GoReeltime uses OCI containers as its distribution unit, which can be run by any compliant runtime [24]. This basic single-container setup of GoReeltime has the following features:

- The web frontend is enabled.
- The upload feature is enabled, and conversion jobs are run inside the container.
- Files and the metadata database are stored on the host file system.

It can be started using the `docker` runtime as follows:

```
# Create a folder for the media and database
mkdir media
# Run the container
docker run --rm \
     --volume ./media:/media \
     --publish 8437:8437 \
     registry.gitlab.com/goreeltime/goreeltime/reeltime:0.1.0 \
     --web
```

Listing 4.2: Command line starting a single, perishable container.

The web interface of this local instance of GoReeltime can be accessed at http://localhost:8437/.

A more permanent installation of this deployment can be created using `docker-compose`, a declarative way of defining one or more containers [24]. In a file named `compose.yaml`, the container can be defined as shown in Listing 4.3.

```
# compose.yaml
---
services:
  reeltime:
    restart: unless-stopped
    image: registry.gitlab.com/goreeltime/goreeltime/reeltime
    :0.1.0
```

```
      command:
      - --web
      ports:
      - 8437:8437
      volumes:
      - ./media:/media
```

Listing 4.3: `docker-compose` example of a single-container deployment.

This configuration can then be launched and stopped using the following commands in the same directory as the newly created file:

```
# Start the configured container (and detatch from its output)
docker compose up -d
# Stop the configured container
docker compose down
```

Scalable Deployment

To run the platform as a scalable service across multiple replicas, a different configuration is required. The environment would need to have a database that can be accessed in parallel, as well as a messaging system in place that can schedule conversion jobs externally (see Figure 4.8).

A possible way of deploying this configuration is shown in the appendix in Listing A.2. It can also be replicated in other container orchestration solutions, like Kubernetes [24].

4.3.9. Cross-Cutting Concepts

Crosscutting concepts cover a range of key considerations that, by their nature, impact multiple aspects of the system. These concepts are fundamental to the development process and operation of GoReeltime, ensuring robustness and maintainability while meeting the demands of modern software engineering practices.

Twelve-Factor App We adhere to the principles of the Twelve-Factor App methodology, which guides the design of modern software-as-a-service applications. This approach

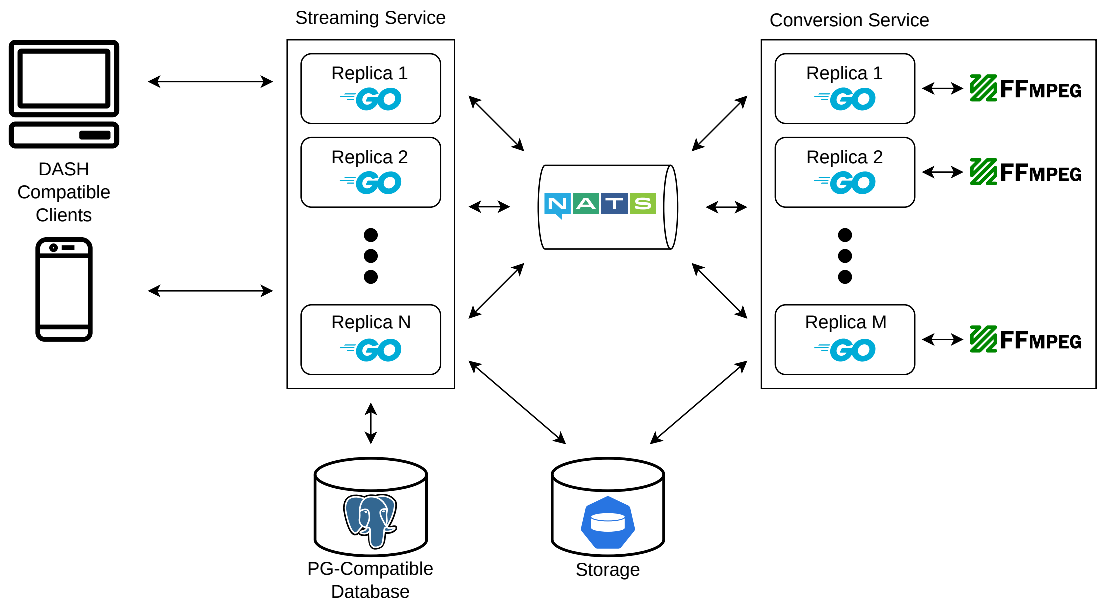

Figure 4.8.: High-level overview of GoReeltime in a scalable deployment.

promotes best practices for building scalable and maintainable cloud native applications. Key factors are declarative formats, isolation and portability [37].

Development Foundations Our development process follows industry best practices, encompassing build automation, continuous testing and deployment (CI/CD) and version control. These practices ensure code quality, rapid iteration, and reliable delivery of features.

Clean Architecture Wherever applicable and compatible with the ecosystem, we adopt Clean Architecture principles, emphasizing separation of concerns and dependency injection in order to promote testability and maintainability of the codebase [17].

4.4. Application

The product of the implementation phase was released on the 11th of June 2024, achieving one of the key motivations behind this project: creating an open-source solution. The release was licensed under the GNU Affero General Public License (AGPL), ensuring that it remains open and accessible for community contributions.

Repository: https://gitlab.com/goreeltime/goreeltime

This section highlights the key features and goals that were achieved, and reflects on some of the difficulties experienced with FFmpeg.

4.4.1. Features

Horizontally Scalable Thanks to its stateless design, GoReeltime can easily scale out by adding more replicas of the service. By moving state to an external metadata and video storage, the need for identifiable instances is eliminated.

Distributed Architecture With its ability to outsource resource-intensive video conversion tasks to another service, the streaming part of GoReeltime can continue to deliver video streams at high speeds. If the need for more parallel video conversion capability is required, the number of replicas performing this task can simply be increased.

Minimalistic Container Build with Ko Containers were chosen as the distribution unit for GoReeltime. To allow for fast deployment and minimal overhead, the container images are built using Ko (https://ko.build), a fast and lightweight build tool. By adding only a single layer containing the output of a Go compiler to the base image, it does not require unnecessary privileges and enhances the continuous integration experience.

Extensible Codebase By using a clean architecture methodology during software development, every component in GoReeltime can easily be exchanged by another implementation [17]. This approach allows the adaption of the software to specific needs and may attract additional developers to the open-source project in the future.

4.4.2. Compliance

Conformance to the DASH standard in order to support many different client implementations was noted as one of the key goals of this project.

The DASH-IF Conformance Tool by the DASH Industry Forum was used to validate both a generated manifest and the active playback of a video generated by GoReeltime. Except for the incompatibility of the validator with WebM-DASH, described in subsection 4.1.4, the implementation fully complies with the standard. The reference player also does not throw any errors.

Through empirical testing, it was also validated that various other software supports video playback from GoReeltime:

- Celluloid, a GTK+ frontend for the `mpv` media player, plays remote video flawlessly (https://celluloid-player.github.io).
- `yt-dlp`, a command-line video downloader application is able to find and parse manifests from GoReeltime and can successfully download videos hosted on the platform (https://github.com/yt-dlp/yt-dlp).

This concludes that the goal of being compatible with as many video player client implementations as possible was achieved.

4.4.3. Live Test

Our main stakeholder, the Distributed Systems and Ledgers Lab (DSL) at the Eastern Switzerland University of Applied Sciences (OST), gave us the opportunity to test the streaming of a converted video on their website. The last news segment of the lecture *Distributed Systems* can be viewed using DASH.

This test was performed to verify the possibility of embedding a video on another website. The following steps were taken:

1. We uploaded the video to our instance of GoReeltime, triggering a conversion process.
2. The output of the conversion (being the encoded video and manifest) was copied from the platform's storage to the web server of the DSL. Instructions on how to embed and configure the dash.js reference player on the website was also provided.
3. The news segment of lecture 14 can now be streamed at https://dsl.i.ost.ch/lect/fs24/.

4.4.4. The FFmpeg Postmortem

FFmpeg is a key building block of GoReeltime, responsible for video encoding tasks. It has proven to be an incredibly powerful tool but is not without issues. This is a summary of features in FFmpeg that were valuable over the course of this projectand the challenges encountered during implementation, along with the strategies employed to overcome them. For reference regarding video formats and encodings, see subsection 2.3.1.

File Descriptors instead of Named Pipes

It was the initial plan to use *named pipes* (also known as FIFOs) to pass video data directly to FFmpeg, in order to avoid the creation of intermediary files and be more memory efficient. Additionally, this would also allow complete independence from the storage implementation. However, named pipes are not *seekable*, they represent a simple stream of data [23].

Using named pipes proved unviable because video containers, particularly MP4, store required header information at the end of the file. This requires a file to be seekable during the conversion process in order to read metadata first. Although the MP4 container format option *faststart* allows metadata to be positioned at the start of the file, this is not the default [11].

Processing MP4 files with FFmpeg would therefore not be possible without the *faststart* option enabled, making most MP4 files incompatible with our system.

Instead, *file descriptors* were used. Unlike named pipes, a file descriptor represents a specific open file, thus making it seekable. This approach unfortunately reduces the possible storage backends, as it must return a valid file descriptor for FFmpeg to work with [21].

This is possible since FFmpeg supports file descriptors to be used as both input and output. However, being a relatively new feature, the available documentation is limited: https://ffmpeg.org/ffmpeg-protocols.html#fd. Listing 4.4 shows a proper usage example. The change was applied with commit 6e7c006 in January 2023 and released with FFmpeg 6.0.1 "Von Neumann" in September of the same year.

Bugs Encountered

The following are issues which were encountered while working on the command line wrapper for FFmpeg.

Audio Channel Format Needs to Be Named Explicitly This is a reported issue regarding the *libopus* encoder not automatically remapping some audio source channel layouts to destination channels, causing encoding failures. Previously, FFmpeg automatically handled this remapping, but a change to the software removed this functionality, requiring manual channel mapping specifications.

GoReeltime uses FFmpeg to encode uploaded video containing audio, so all possible channel layouts need to be specified for every audio format job.

Issue: https://trac.ffmpeg.org/ticket/5718

Analysis Timeout for Some Video Formats To calculate the required output quality formats for a video, GoReeltime uses FFmpeg to analyze the video and audio streams of uploaded files. As some video container formats store the header information at the end of the file, FFmpeg needs to process the entire file to find it. Depending on the size of the file, a timeout can be reached, in which case the process fails.

GoReeltime solves this problem by increasing the values of the `analyzeduration` and `probesize` arguments, leaving FFmpeg more time to analyze the uploaded file and determine the output.

Issue: https://trac.ffmpeg.org/ticket/10678

Manifest Generation Fails for Single-Key-Frame Videos For very short input files, there is not enough play time to allow for more than one key frame to be generated. FFmpeg does not generate MPD manifests for video files with a single key frame, causing the manifest generation process to fail.

The impact of this issue is reduced by reducing the key frame interval of shorter videos, this can however not be mitigated completely.

Issue: https://trac.ffmpeg.org/ticket/9999

Manifest Generation Uses File Descriptor Names GoReeltime uses file descriptors to the data of uploaded video files via the file descriptor protocol in FFmpeg (see Listing 4.4).

There is a currently unreported bug in FFmpeg in which the manifest generated from a command that uses the file descriptor protocol does not include the actual file names, but instead just puts `fd:` as the `BaseURL` (see subsection 4.1.3 for reference). This generates a valid but incorrect manifest, as all file names need to be adjusted in it.

```
# Generating a manifest using file names
# Results in <BaseURL>input.webm</BaseURL>
ffmpeg \
-f webm_dash_manifest -i input.webm \
-c copy -map 0 \
-f webm_dash_manifest -adaptation_sets "id=0,streams=0" \
manifest.mpd

# Generating a manifest using file descriptors
# Results in <BaseURL>fd:</BaseURL> instead of file name
ffmpeg \
-f webm_dash_manifest -fd 0 -i fd: \
-c copy -map 0 \
-f webm_dash_manifest -adaptation_sets "id=0,streams=0" \
-fd 1 fd:
```

Listing 4.4: Difference between running an FFmpeg command with file names and file descriptors.

Chapter 5.

Results

The primary goals of this thesis were to develop a horizontally scalable, standard-compliant video streaming server. It should implement automatic conversion upon upload and adhere to the Twelve-Factor Methodology. As an additional goal, the server should be released as an open-source project.

The results show that the developed platform achieves horizontal scalability through its stateless server design and the integration of a message queue system for handling video conversion tasks. This design ensures that the platform can efficiently manage increasing loads without degradation in performance. Compliance with the DASH standard was successfully validated through the *DASH-IF Conformance Tool* and empirical testing, confirming that the platform can deliver ABR streaming to a wide range of clients.

Regarding the development process, it adhered to the Twelve-Factor Methodology. Special care was taken to ensure that the platform logs to stdout, scales via the process model, and communicates with backing services through APIs, specifically using NATS for messaging. This methodology aided in creating a resilient and maintainable system architecture. The software has been released as free software under the GNU Affero General Public License (AGPL), ensuring that it remains open and accessible for community contributions and enhancements.

Moreover, through a test in a production environment, the platform's ability to manage video conversion processes while maintaining DASH compliance was confirmed. Users can seamlessly stream videos on various devices and clients while maintaining efficient handling of system resources, as requests are resolved immediately, resulting in a reduction of latency compared to streaming entire video files.

In conclusion, the developed streaming platform successfully meets the outlined objectives, providing a scalable, distributed, and open-source solution with integrated video conversion capabilities. It offers a viable alternative to proprietary platforms, empowering users with complete control over their streaming infrastructure. However, the system must still be further developed to ensure that it can be used standalone in a production setting.

5.1. Future Work

Moving forward, there are several areas for future research that can build on the developed systems and findings. First, this includes the evaluation and implementation of an authentication and authorization model that maintains the backend's stateless nature, potentially using JWT (JSON Web Tokens) with OpenID Connect and OAuth. This would enable more fine-grained access control over uploaded video files. Additionally, implementing a *cancel* function for conversion tasks would allow the graceful cancellation of conversions if a video is deleted before its completion, possibly by using a broadcast queue.

Further research into optimal parameters for the video format should continue, aiming to balance quality and file size. The platform could also benefit from user-configurable quality settings, such as high and standard quality options for different kinds of content. Finally, developing a robust frontend or client applications using the system's API would enhance usability and accessibility.

These future enhancements present opportunities for further student research projects and bachelor theses. Contributions from the community are encouraged to extend the software's capabilities to address the need for video streaming platforms.

Part II.

Appendix

Appendix A.

Code Listings

A.1. Grafana k6 Configuration

In order to configure Grafana k6 to perform continuous HTTP range requests for 100 users over a period of 30 seconds, the following configuration was used.

Execution: `k6 run k6-script.js`

```
import http from "k6/http";
import { check } from "k6";

export const options = {
    vus: 100,
    duration: "30s",
    thresholds: {
        // 95% of requests must complete within 500ms
        http_req_duration: ["p(95)<500"],
    },
};

function getRandomRange() {
    // Video is 12 MiB, request one MiB
    const ONE_MEBIBYTE = 2 ** 20;
    let startByte = Math.floor(Math.random() * 10 *
    ONE_MEBIBYTE);
    let endByte = startByte + ONE_MEBIBYTE;
    // Return byte range in 'Range'-header format
    return `bytes=${startByte}-${endByte}`;
}

export default function () {
    let rangeHeader = getRandomRange();
    let headers = { Range: rangeHeader };
    // Request a random byte range of the sample video
    let response = http.get(
        "http://192.168.69.2:8437/video/catvideo/
        catvideo_3840x2160_3000k.webm",
        { headers: headers },
    );
    // HTTP 206: Partial Content (range request successful)
    check(response, {
        "status is 206": (r) => r.status === 206,
    });
}
```

Listing A.1: Grafana k6 configuration.

A.2. Scalable Docker Compose Configuration

This configuration uses Traefik (https://traefik.io/traefik) for load balancing and stores data in Docker volumes.

- Execution: `docker-compose up`
- Add converter replicas: `docker-compose scale reelconvert=10`
- Reset converter replicas: `docker-compose scale reelconvert=1`

```
---
services:
  traefik:
    restart: unless-stopped
    image: traefik:v3.0
    command:
    - --providers.docker
    - --providers.docker.exposedbydefault=false
    - --entrypoints.web.address=:8437
    volumes:
    - /var/run/docker.sock:/var/run/docker.sock
    ports:
    - 8437:8437
    networks:
    - default
    - frontend
  reeltime:
    restart: unless-stopped
    image: registry.gitlab.com/goreeltime/goreeltime/reeltime
    :0.1.0
    command:
    - --web
    - --postgres-host=database
    - --postgres-user=postgres
    - --postgres-password=postgres
    - --nats-host=nats
    ports:
    - 8437:8437
    volumes:
    - media:/media
    networks:
    - frontend
    - backend
    labels:
      traefik.enable: true
      traefik.docker.network: frontend
      traefik.http.routers.reeltime.rule: PathPrefix(`/`)
      traefik.http.routers.reeltime.entrypoints: web
      traefik.http.services.reeltime.loadbalancer.server.port:
```

```
      8437
  reelconvert:
    restart: unless-stopped
    image: registry.gitlab.com/goreeltime/goreeltime/
    reelconvert:0.1.0
    command:
    - --nats-host=nats
    volumes:
    - media:/media
    networks:
    - backend
  database:
    restart: unless-stopped
    image: postgres:16.2
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: reeltime
    ports:
    - 5432:5432
    volumes:
    - postgres:/var/lib/postgresql/data
    networks:
    - backend
  nats:
    restart: unless-stopped
    image: nats:2.10.14-alpine3.19
    command:
    - --jetstream
    ports:
    - 4222:4222
    networks:
    - backend
volumes:
  media:
```

```
  postgres:
networks:
  frontend:
    name: frontend
    driver: bridge
    internal: true
    ipam:
      config:
      - subnet: 10.43.0.0/16
  backend:
    name: backend
    driver: bridge
    internal: true
    ipam:
      config:
      - subnet: 10.44.0.0/16
```

Listing A.2: Scalable deployment of the GoReeltime platform using Docker compose.