SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Scalability and Performance Optimization in Video Streaming Infrastructure

Carlo Bortolan

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Scalability and Performance Optimization in Video Streaming Infrastructure

Skalierbarkeit und Leistungsoptimisierung von Video Streaming Infrastruktur

Author:	Carlo Bortolan
Examiner:	Prof. Dr.-Ing. Jörg Ott
Supervisor:	Dr. rer. nat. Andreas Paul
Submission Date:	15.10.2024

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15.10.2024

Carlo Bortolan

Acknowledgments

Special thanks to my advisor, Andreas Paul, for the support, feedback and encouragement throughout this thesis; Joscha Henningsen for introducing me to the GoCast project and helping me get started and understand its architecture; the TUM-Live and TUM-Dev groups, especially Sebastian Wörner, Dawîn Yurtseven and Andreas Jung for their feedback and technical assistance; and last but not least my family and friends for their support and patience during the last few months.

Abstract

Video streaming platforms are used on a daily basis by millions of users and need to be able to provide high-quality videos without interruptions. TUM-Live[1], used daily by thousands of students at the Technical University of Munich (TUM) is a perfect example for such a system that consists of multiple services that need to perform reliably with as little human intervention as possible. In my bachelor's thesis, I researched different approaches to scaling such a system and increasing stability by extending the current TUM-Live infrastructure to a distributed network managed by different organizations that can share computing resources and storage. Additionally, I analyzed different API[2] design approaches by comparing the current REST[3] API with a prototype for a gRPC[4] API that reduced latency by 3x and improved throughput by 8x. The overall goal was to understand, analyze and improve the current infrastructure and assess its potential to scale and identify potential future limitations, such as performance bottlenecks within the subsystems or the problem of having an in-memory chat system.

[1] `https://tum.live`
[2] Application Programming Interface (API)
[3] Representational State Transfer (REST)
[4] Google Remote Procedure Calls (gRPC)

Contents

1 Introduction

In recent years, the demand for video streaming has grown unprecedentedly, driven by platforms like YouTube, Netflix and Twitch, which have fundamentally transformed how people consume and share content globally.

This exponential growth in viewers, combined with an increasingly broad range of devices, network conditions, user expectations and costs to be able to maintain such platforms, have made the challenge of scalability and optimization one of the most critical areas in the field of distributed systems. Central to this challenge is the ability to balance computational resources, dynamically distribute streams and develop new hardware and software optimizations to provide a solid viewing experience to users in real-time.

GoCast, developed by students at TUM and currently in use at the School of Computation, Information and Technology (CIT) as TUM-Live is a perfect example of such a streaming service. With plans to expand it to other TUM schools and the potential for university-wide lecture streaming, there is a need to analyze and update the current system to handle distributed resources and distribute both the processing and storage of lecture streams as well as administrative tasks to the individual schools to reduce the load on the central servers and administrators.

We start with a chapter on its history and milestones over the years, using examples from three well-known video streaming services that display the general architecture, challenges and optimization potentials in this industry. Next, there is a chapter on TUM-Live, explaining its architecture and current state, followed by an overview of the technical concepts behind streaming services.

In the second half, we will focus exclusively on TUM-Live, showing a proposed solution that has been developed over the last few months to scale its architecture to support multiple organizations. This will be supported by a detailed analysis of the updated and newly designed systems and possible improvements and limitations. Finally, for the last part, we will take a look at solutions to optimize the current system with regard to common errors and how they are handled, as well as an optimization approach for database queries and a comparison of different API designs.

In short, this thesis aims to answer the question of how existing infrastructure behind video streaming services such as TUM-Live can be optimized and scaled while maintaining reliability, stability and high streaming quality.

2 Examples from Industry

This chapter gives a broad overview of the history of video streaming (see Subsection 4.1.1 for the definition of *video streaming*) and then presents three aspects of video services in detail using examples from the well-known video streaming services: First a look at the system architecture of a streaming service, using the example of Twitch, then a look at how Netflix works together with Internet Service Providers (ISPs) to deliver content as efficiently as possible and lastly a section on how YouTube handled different scalability challenges and developed its own video transcoding system.

2.1 History

The video streaming industry has evolved dramatically over the past decades, transitioning from old-fashioned technologies to highly complex, scalable infrastructures capable of simultaneously delivering large amounts of content to hundreds of millions of users. The origins of video streaming can be traced back to the late 1990s and early 2000s, with the introduction of streaming technologies like RealNetworks' RealPlayer and Apple's QuickTime. These early platforms allowed users to stream audio and video content, although it was of low quality and had significant buffering issues due to limited bandwidth and server capacities. Another important precursor to modern streaming was Multicast Backbone (MBone), a virtual network introduced in the early 1990s to support audio and video broadcasts over the internet, showing the potential behind internet-based multimedia transmission. Around the same time, web conferencing (often called webcast) became popular for video-chat tools, multiuser application-sharing and real-time polls. After the widespread adoption of Adobe Flash in the early 2000s, the interest in a single, unified streaming format started the development of a Flash-based streaming format, which was then used by Flash video players on most streaming sites. The real breakthrough in video streaming came in 2005 with the launch of YouTube, which introduced a user-friendly platform for uploading, sharing and streaming videos online. YouTube's success not only showed the willingness of users to produce, upload and consume hours of original video content via the internet but also the potential for scalable video streaming infrastructure, increasing the demand for online video content. At the same time, Netflix's pivot from a mail-based DVD rental business to streaming in 2007 set another milestone in the industry.

As internet speeds increased and cloud computing became more prevalent, the scalability of video streaming services improved (Netflix closed its last physical data center in mid-2016 [IVM16]). Additionally, the introduction of Adaptive Bitrate Transcoding (ABR) streaming allowed platforms to dynamically adjust video quality depending on the user's internet connection, reducing loading times and improving the overall viewing experience [Ben+18]. Meanwhile, using a Content Delivery Network (CDN) became increasingly important for caching video data closer to users, thereby reducing latency and server load [Adh+12].

2.2 Example: Twitch

Twitch launched in 2011 as a spin-off of the general interest streaming platform Justin.tv. Since then, it has become synonymous with live streaming, particularly within the gaming community and is still growing rapidly across different categories.

One of Twitch's most significant challenges is managing vast amounts of user-generated content in real-time and scale with high fluctuations in viewer numbers, particularly during major events like e-sports competitions. Twitch's architecture is highly complex and needs to support millions of concurrent video streams, real-time interactions through chat and extensive data processing such as predictive modeling for personalized recommendations, spam detection for chat messages and targeted campaigns based on in-app user behavior [Web17]. In the following, there is a detailed breakdown of the key components of its architecture (see also Figure 2.1):

1. **Video System**

   The video system begins with the **video ingest** process, where Real-Time Messaging Protocol (RTMP) live streams are received from streamers. Once the video stream is ingested, it is transported to the **transcode sub-system**. This sub-system, implemented in a combination of C/C++ and Go, transcodes the incoming RTMP stream into multiple HTTP Live Streaming (HLS) streams to allow viewers to switch between different resolutions when watching a stream. After transcoding, the streams are distributed through Twitch's global **Distribution and Edge** network consisting of multiple Points of Presence (POPs). The POPs cache the HLS streams and deliver them to users from geographically optimal locations, minimizing latency and buffering. The distribution system, also largely written in Go, is designed to scale massively to ensure high availability even during peak usage times. Additionally, Twitch archives all live streams through its **Video on Demand (VOD) system**, making content available for later viewing, either immediately after the live broadcast or as part of a long-term archive [She15].

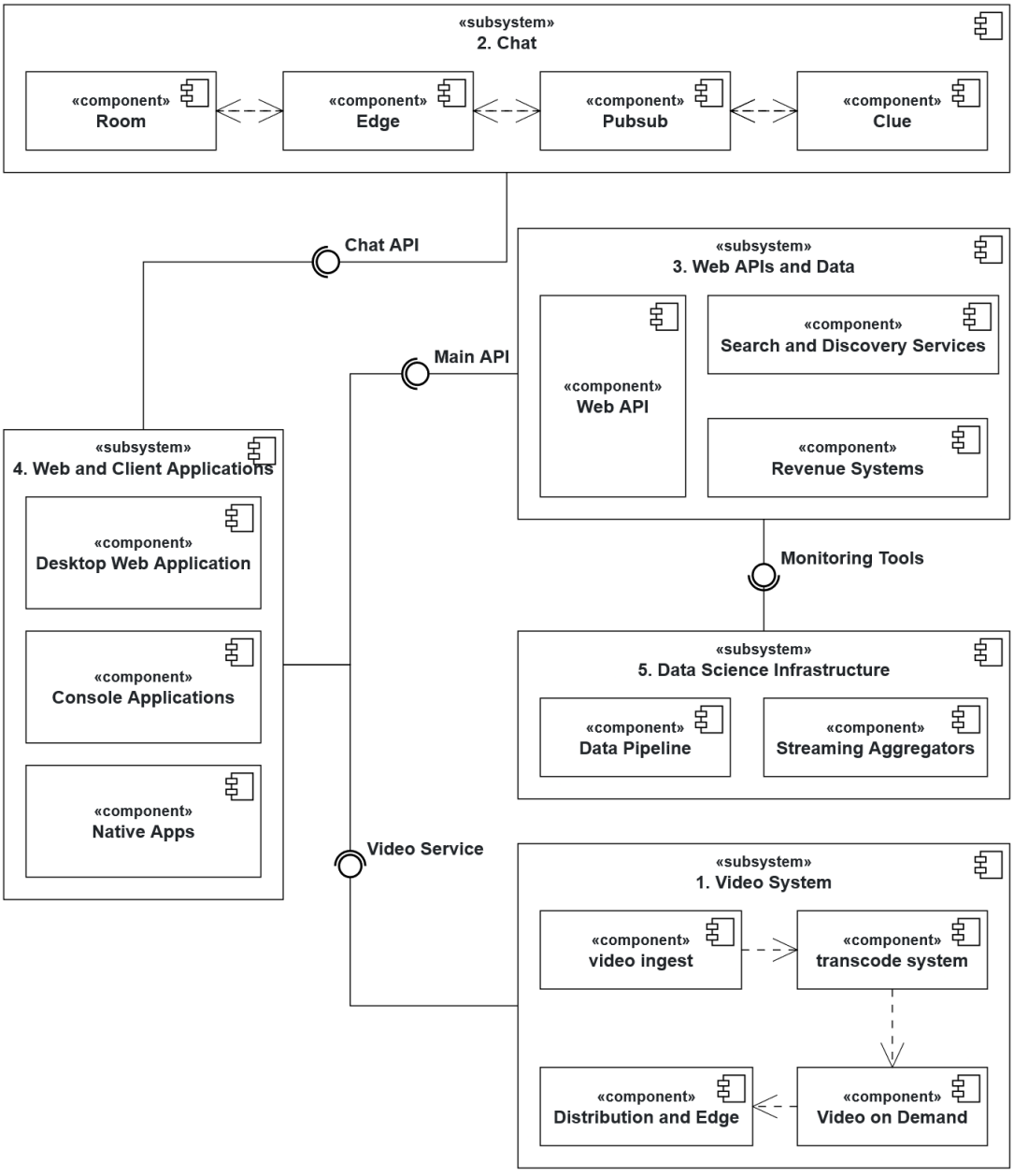

Figure 2.1: System Architecture of Twitch

2. **Chat**

   The chat system is a real-time distributed system, primarily written in Go, and designed to handle real-time interaction between viewers and streamers. The **Edge** component of the chat system is responsible for receiving and distributing messages between clients and backend services.

   Next, the **Pubsub** subsystem is used for the internal distribution of chat messages across various Edge nodes. Together, they create a hierarchical message distribution system capable of executing massive fanout to guarantee that all users in a chat room receive messages quickly and reliably. The **Clue** component handles the application of business logic to these chat interactions. For example, it authorizes the user's message by checking if they are banned from a channel, whether they are a subscriber, or if they are showing abusive behavior. Clue achieves this by accepting messages forwarded by the Edge nodes and then aggregating data from various databases, internal APIs and caches. Using this data, the viewers' messages are evaluated against existing rules in real time. Finally, the **Room** component manages the viewer list for each chat room. It aggregates, stores and queries membership data across all Edge nodes to provide accurate and up-to-date viewer lists, which are crucial for both moderation and user interaction [Swa16].

3. **Web APIs and Data**

   Twitch's platform also includes a set of web APIs and data analysis services for various purposes, e.g., from user profile management to stream discovery. These **Web APIs** are built using a combination of Ruby on Rails, Go and other open-source frameworks, designed to handle high request volumes, with Twitch's services processing over 50,000 API requests per second on average. These APIs allow users to manage their profiles, customize subscriptions and interact with other services on the platform [She15].

   Additionally, Twitch also has various microservices for specific use cases, such as **Search and Discovery Services** to help users find streams and content that match their interests or **Revenue Systems** that manage all aspects of advertising and subscriptions, ensuring that revenue is accurately tracked and distributed to partners [She15].

4. **Web and Client Applications**

   Twitch's **Desktop Web Application** began as a vanilla Rails application but has developed into an Ember.js application. For mobile users, Twitch offers **Native Apps** on iOS and Android as well as **Console Applications** for major gaming systems, including Xbox One, Xbox 360 and PlayStation 4 [She15].

5. **Data Science Infrastructure**

   Next to its operating infrastructure, Twitch's data science infrastructure plays an important role in optimizing the platform, improving user experiences and driving business decisions. At the core of this infrastructure is the **Data Pipeline**, which is responsible for collecting, cleaning and loading over a billion events per day into Twitch's data warehouse [She15].

   The platform also uses so-called **Streaming Aggregators** to summarize key metrics in near real-time and provide broadcasters with direct feedback on their stream performance [She15].

6. **Tools and Operational Infrastructure**

   Quality Assurance (QA) is critical, with Twitch utilizing both **Automated Testing Frameworks** such as Jenkins to allow for continuous integration and testing to maintain high code quality across all services.

   In addition to that, there are also several **Deployment and Rollback Tools**, as well as **Monitoring and Alerting Systems**, including Ganglia[1], Graphite[2] and Nagios[3] that monitor the status and performance of the infrastructure, providing real-time alerts and insights that help engineers quickly identify and prevent problems [She15].

   Lastly, Twitch's **Network Infrastructure** mostly operates on bare-metal POPs worldwide while an increasing number of services are being migrated to Amazon Web Services (AWS), which helps reduce operational overhead while benefiting from the on-demand scalability and flexibility of cloud services.

[1] `https://ganglia.info`
[2] `https://graphiteapp.org`
[3] `https://www.nagios.org`

2.3 Example: Netflix

2.3.1 Overview of Netflix Open Connect

Netflix Open Connect was developed in 2011 and officially launched in 2012 as a response to the rapidly increasing scale of Netflix's streaming service [Nai17]. Before this, Netflix relied on third-party CDNs to deliver content to its users. However, as Netflix's global internet traffic grew, it became evident that a custom-built CDN could provide more efficiency and better performance adapted to Netflix's specific needs. The core of Netflix Open Connect is its global network of Open Connect Appliances (OCAs), which consists of more than 8,000 specialized servers located in over 1,000 locations around the world [Flo16]. These OCAs are placed strategically in ISPs' data centers to allow Netflix to deliver content directly to the users without relying heavily on the general infrastructure of the internet. When a user signs into Netflix or performs other account-related actions, the user's client makes a request to the central server(s), which can potentially be far away from the user's location. However, when a video is requested, the request is sent to one of the nearby OCAs, which then serves the large video files directly to the user. Hence, this reduces latency and minimizes the amount of traffic, as the cached video data only needs to travel from the OCA to the user.

2.3.2 Content Delivery Mechanism

Netflix's content delivery strategy involves pre-positioning content on OCAs before users request it. This is achieved through advanced algorithms that analyze and predict popular content to make sure that the most likely to be watched content is already stored close to where it will be consumed [IVM16]. For example, in regions with limited internet capacity, such as Australia, Netflix minimizes the use of undersea cables by pre-loading content onto OCAs during off-peak hours, reducing the need for real-time content transmission over these expensive and bandwidth-limited connections.

2.3.3 Performance Optimization and Energy Efficiency

Over the years, Netflix has continued to optimize its OCAs, increasing their efficiency by an order of magnitude since the program's creation. For instance, the throughput of a single server has increased from 8 Gbps in 2012 to over 100 Gbps[4] in 2017 of encrypted traffic from a single OCA [Gal17], largely due to improvements in both hardware and software [Flo16]. These advancements have also led to smaller and more power-efficient OCAs.

[4] For FHD (1080p) resolution, which requires a download speed of around 5 Mbps (see `https://help.netflix.com/node/306`), 100 Gbps would support 20,000 simultaneous streams.

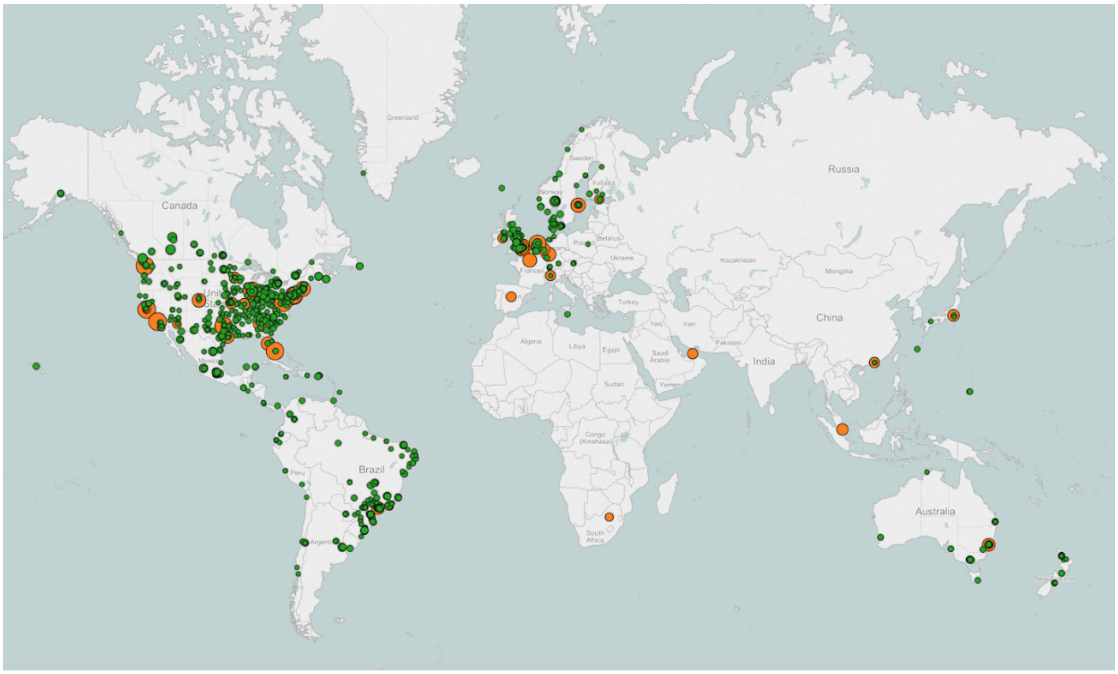

Figure 2.2: Netflix Open Connect Network as of 2016 [Flo16]

2.4 Example: YouTube

YouTube, launched in 2005, is the world's largest video-sharing platform, with more than 500 hours of videos uploaded every minute and approximately 694,000 hours of video content streamed per minute [Sta22].

2.4.1 General Architecture

The following system design model (see Figure 2.3) will not be explained in detail and is meant to help understand and visualize components referenced in this section.

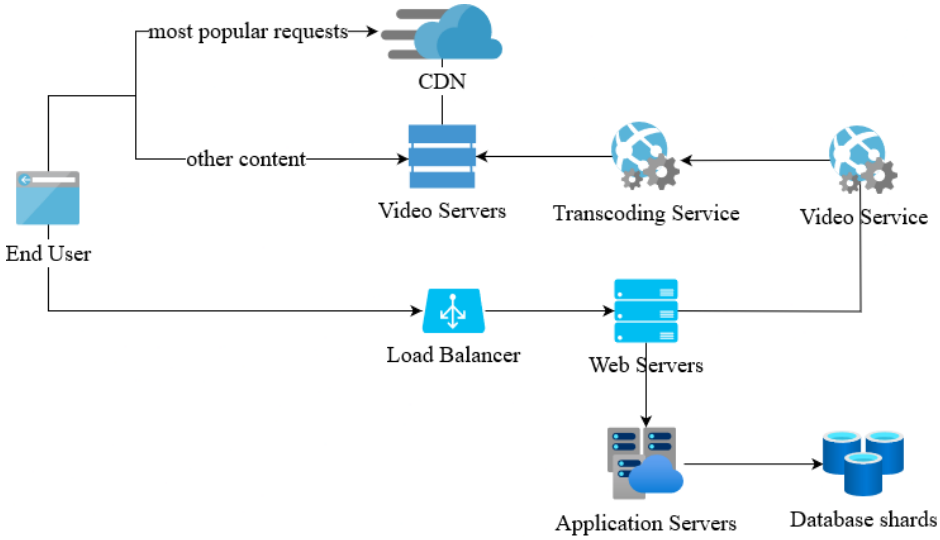

Figure 2.3: System Architecture of YouTube

YouTube's infrastructure is built on Google's global network of data centers and CDNs: The most popular content for a certain region is cached in global CDNs that make sure that there are multiple copies of a video to guarantee quality of service, reliability and closeness to the user while the other videos are stored on separate servers in various locations, mainly in the US [Hof08]. The platform uses advanced video coding formats, such as VP9[5] to optimize bandwidth usage while maintaining high video quality, even

[5]VP9 is a video coding format developed by Google and published in 2013 to compete with H.265, designed to handle high resolutions up to 65536×65536 pixels and mainly used by YouTube. It works by compressing video content using block-based transformation, where the image is divided into coding units of 64×64 pixels, which are further subdivided into smaller blocks based on the content complexity. A draft of the specification can be found at `https://storage.googleapis.com/downloads.webmproject.org/docs/vp9/vp9-bitstream-specification-v0.6-20160331-draft.pdf`

at lower bitrates [Ran+21]. In recent years, YouTube has also expanded its feature offering to include live streaming and 360-degree videos [Wil16].

2.4.2 Scalability Challenges

While there is not much information on current scalability issues published by YouTube, there are several records of problems the YouTube team faced up to 2012. These challenges included managing rapid and unpredictable user growth with limited resources, as there was insufficient budget to maintain excess machines to handle such growth. At the same time, the introduction of compute-heavy features, such as social graphs and algorithms for recommender systems, did not make it easier to find resources for video processing and storage. The team also pushed the limits of the hardware and software of that time, encountering limitations like performance bottlenecks in database structures before database partitioning (more details in Section 4.3) [Cuo12]. As the platform continued to evolve, identifying these bottlenecks became more difficult, as they often were not due to an issue by YouTube itself, but rather by limitations of other libraries and third-party software they used [Cuo08].

2.4.3 Video Acceleration at Scale

With Moore's Law slowing down, specialized hardware accelerators optimized for large-scale video transcoding are needed to meet the demand for video processing. In 2021, the YouTube team presented an accelerator called the Video Coding Unit (VCU) and showed that by scaling it within Google data centers, it achieved 20-33x improved efficiency[6] compared to the previous well-tuned non-accelerated systems (see Table 2.1) [Sil21].

Table 2.1: Throughput and Performance Comparison of Systems [Ran+21]

System	**Throughput [Mpix/s]**		**Perf/TCO**	
	H.264	VP9	H.264	VP9
Skylake	714	154	1.0x	1.0x
4xNvidia T4	2,484	–	1.5x	–
8xVCU	5,973	6,122	4.4x	**20.8x**
20xVCU	14,932	15,306	7.0x	**33.3x**

The core component of the VCU system is the encoder core, which serves as a special hardware component acting as an accelerator built for large distributed clusters. While

[6]Measured by performance-per-TCO (total cost of ownership) with values above 1.0x indicating better performance or cost-efficiency compared to the current non-accelerated reference system (Skylake).

the VCU design optimizes for throughput and system balance, allowing it to adapt to changing workloads and infrastructure demands, its major challenges are handling multiple resolutions and formats and ensuring system reliability in a large-scale environment [Ran+21]. Also, the VCU integrates multiple encoder cores and a memory system optimized for data center workloads, which allows it to be adjusted dynamically depending on network bandwidth, storage and live/VOD demand workloads. This hardware-software co-design approach helps efficiently transcode uploaded videos while being flexible for future workloads. The system is highly parallelized, allowing for efficient handling of multi-output transcoding where a single input video is processed into multiple resolutions and formats at the same time [Ran+21].

The VCU system enables otherwise infeasible VP9 compression at scale, caused by VP9's larger block sizes and limited hardware support. This results in new use cases related to live streaming (e.g., using VP9 to encode many short (2-second) segments in parallel to increase overall throughput). "*As a concrete example, a 2-second 1080p chunk could be encoded in 10 seconds, the encoding system would transcode 5-6 chunks concurrently to achieve the needed throughput of a 1 video-sec/second*" [Ran+21]. Currently, this system is mainly used by YouTube for a wide range of video workloads, including video sharing, live streaming and cloud gaming.

3 Fundamentals of TUM-Live

3.1 GoCast Lecture Streaming Service

3.1.1 Overview

GoCast is a fully self-hosted platform for live streaming and recording of lectures developed by students at TUM. The source code is open-source, accessible at github.com/TUM-Dev/gocast and licensed under the MIT license. Its main features include:

- Automatic live streaming from auditoriums based on lecture schedules imported from CAMPUSonline (campus management system used at TUM as TUMOnline).
- Self-service interface for lecturers to schedule and manage their VODs and streams.
- Automated import of lecture schedules and enrollments from CAMPUSonline.
- Self-streaming via third party streaming software such as OBS[1], Zoom[2], etc.
- Automatic recording of live streams.
- Manual VOD uploads.
- Automatic post-processing of recordings.
  - Detect silence in videos and make them skipable.
  - Transcribe live streams and VODs using the Whisper LLM[3].
  - Generate thumbnails.
- Optional live chat for viewers to ask questions.
  - Polls can be created by lecturers.
  - Questions can be upvoted by viewers and answered or hidden by lecturers.
  - Optional moderation features for lecturers.

[1] `https://obsproject.com`
[2] `https://zoom.us`
[3] `https://github.com/openai/whisper`

3.1.2 Current System Architecture

The current system architecture of GoCast can be divided into three parts:

1. User-, course- and task-management:

At the core of the GoCast system, there is the main API built on the Gin-Gonic[4] Framework and connected to a MariaDB[5] Database. Its main functionality is to manage users, courses & streams, pull events from CAMPUSonline and schedule tasks. For user authentication, it can use Single Sign On (SSO) and Security Assertion Markup Language (SAML) to allow users to authenticate themselves with their university credentials.

2. VOD upload related components:

Next, whenever a lecture is uploaded as a VOD, the video data is sent to a Worker, which then transcodes and segments the video into MPEG-2 compressed video transport stream files using FFmpeg[6]. These segments are then copied to a shared storage using the VOD Service component, which is later distributed by the Edge Server to the end user. There are plans to replace the Worker subsystem with a more robust and efficient Runner system. However, as the Runner is still in the development phase and not tested yet (see Subsection 6.3.3), we will mainly refer to the Worker subsystem throughout this thesis (although in principle, the Worker and Runner concepts can be used interchangeably).

3. Lecture recording and live streaming:

Lastly, for live streaming, GoCast depends on external streaming infrastructure. In TUM's case this is the live streaming infrastructure of the Leibniz Supercomputing Centre (LRZ) which internally runs the Wowza Streaming Engine[7]. Whenever a lecturer starts a live stream from a lecture hall, the produced RTMP stream is sent via the Worker to the LRZ streaming service, which then processes the live stream and makes it available for the viewer in real time. When the viewer now watches a live stream, the video is streamed directly to the user's browser client from the LRZ streaming service.

[4] `https://github.com/gin-gonic`
[5] `https://mariadb.org`
[6] `https://ffmpeg.org`
[7] `https://doku.lrz.de/allgemeines-526418642.html`

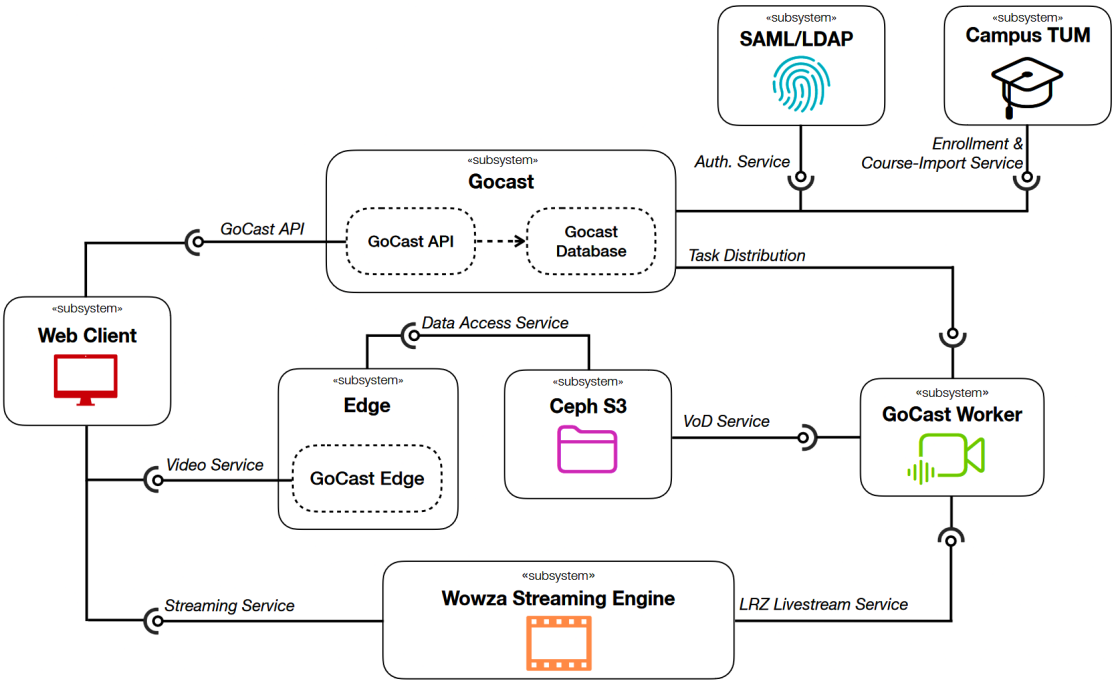

Figure 3.1: System Architecture of GoCast

3.2 TUM-Live

GoCast is in use at the TUM as TUM-Live. This section aims to group and structure existing information related to TUM-Live's history, technical setup and statistics in a concise manner.

3.2.1 Technical Setup of TUM-Live

To professionally stream hundreds of lectures every semester, TUM-Live has integrated GoCast into a complex infrastructure of hardware components deployed in separated VLANs that allow for easy streaming from a lecture hall without having to do additional configuration at the beginning of each lecture. First, there needs to be at least one pan–tilt–zoom camera equipped in a lecture hall to record and follow the lecturer and the white-/blackboard. Then, every lecture hall needs to be equipped with a so-called Streaming Media Processor (SMP)[8] which is an all-in-one recording and streaming processor that captures, switches, scales and distributes audio and video sources and presentations. In addition to that, it also provides multiple concurrent streams, including flexible two-window layouts and full-screen views. This is especially useful as it allows sending video data in three different formats to the main TUM-Live instance: `CAM` for the camera pointing at the lecturer, `PRES` for the currently shown screen of the lecturer's laptop or `COMB` for a two-window view of both. The SMP can also be used as a backup device for the VOD in case something goes wrong in a later stage (e.g., a Worker has an outage and the recorded stream is "lost") as the SMP device creates a local copy of the recording. The main problem with SMPs is their high cost, as one device can cost over 10,000 EUR.

3.2.2 Progress of TUM-Live

Originally, TUM-Live was started in 2019 by the multimedia group of the former faculty of informatics to stream overcrowded lectures of popular courses into different lecture halls as some courses had more students than lecture hall seats. A lecturer would create a RTMP stream of his current lecture and publish the stream in a private network to a different lecture hall which then would display the streams to the students. At the same time, this system started being used more and more to provide a public live stream and VOD portal and archive for students of selected courses. The system itself was - in comparison to the current system - rather simple as it displayed a list of streams that would show the currently live stream or link to uploaded .mp4 files (see Figure 3.2).

[8] `https://www.extron.com/article/smp`

Figure 3.2: TUM-Live as of 2019[9]

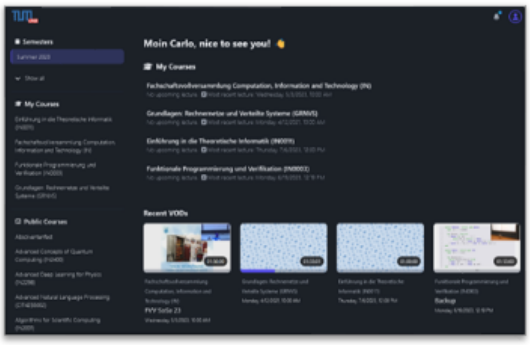

Figure 3.3: TUM-Live as of 2024[10]

This quickly changed in 2020 as the COVID-19 pandemic created a high demand for online lecture video streams, resulting in the development of GoCast. Starting in fall 2021, the old TUM-Live system switched to using GoCast, as it provided a user-friendly and easily accessible user interface for students and a cost-efficient and privacy-focused alternative to other lecture streaming platforms for TUM's media group "ProLehre."

Nowadays, TUM-Live offers live and on-demand videos of lectures and events from TUM's CIT[11]. It is maintained by members of the CIT multimedia group and the open-source TUM-Dev community. In 2023, a prototype for a mobile app was developed but not released yet[12].

Some of TUM's other schools have also shown interest in joining the CIT in using TUM-Live, but do not have the budget to equip all lecture halls with SMPs. However, this issue might soon be resolved, as a student has developed his own open-source SMP called Virtual Media Processor (VMP) that aims to re-implement the functionality of the SMP in software, using the GStreamer Multimedia Framework[13]. The target hardware is a small single-board computer with accelerated Multimedia encoding/decoding and additional HDMI capture capabilities, costing only around 500 EUR. Given the low cost of such a device, it is very likely that other TUM schools and possibly even other universities will join TUM-Live in the future.

[9] `https://web.archive.org/web/20191001114650/https://live.rbg.tum.de`
[10] `https://live.rbg.tum.de`
[11] `https://www.cit.tum.de`
[12] `https://gstreamer.freedesktop.org`
[13] `https://gstreamer.freedesktop.org`

3.2.3 User Statistics of TUM-Live

Since its creation in February 2021, TUM-Live has been used to stream thousands of hours of video every semester for more than 1,300 courses, 20,000 streams and 30,000 students. The following plots (see Figure 3.4) display a broad overview of viewer metrics from the current system. As the *VoD activity throughout the day* plot shows, the hours at which the users watch recorded VODs is normally distributed, with the mean being around 4PM. Most students use TUM-Live throughout the entire lecture week (see *VoD activity per day of week* plot showing an evenly distributed VOD activity over the week), meaning that the Edge Servers need to be fully functional at all times. At its peak, there are nearly 6,000 VOD replays per day (see *VoD activity per day*), while at times - mostly during the semester breaks - there are weeks with nearly no VOD activity at all.

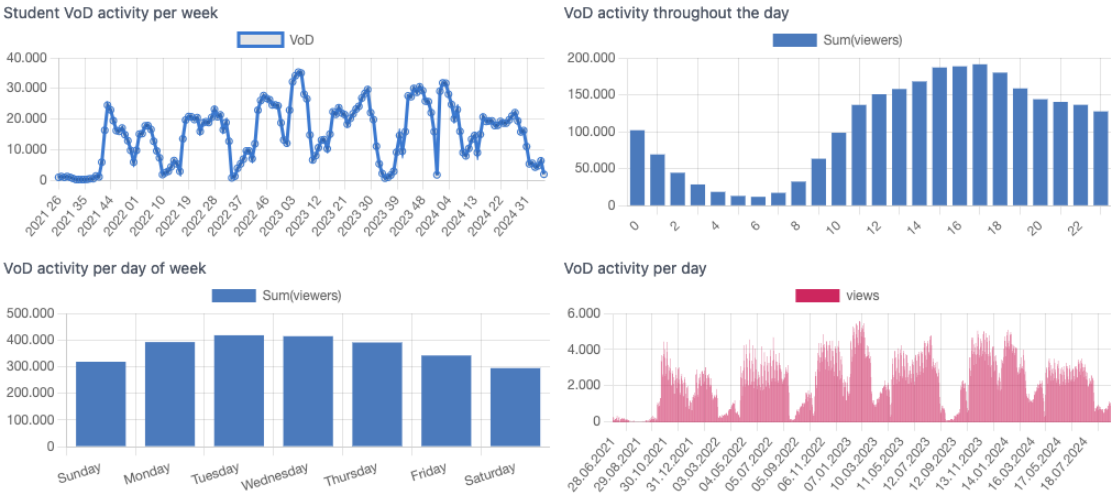

Figure 3.4: TUM-Live Viewing Statistics

What's especially interesting is the development of the number of users, courses and streams over time. As can be seen in Figure 3.5, at the beginning of each semester, there is a clear spike in new users, created courses, and streams. This is mainly due to the automatic import of lecture data and enrollments from CAMPUSonline. Between semesters, the increase in new users and courses is rather flat, with a slight increase in the number of streams (mostly streams created manually).

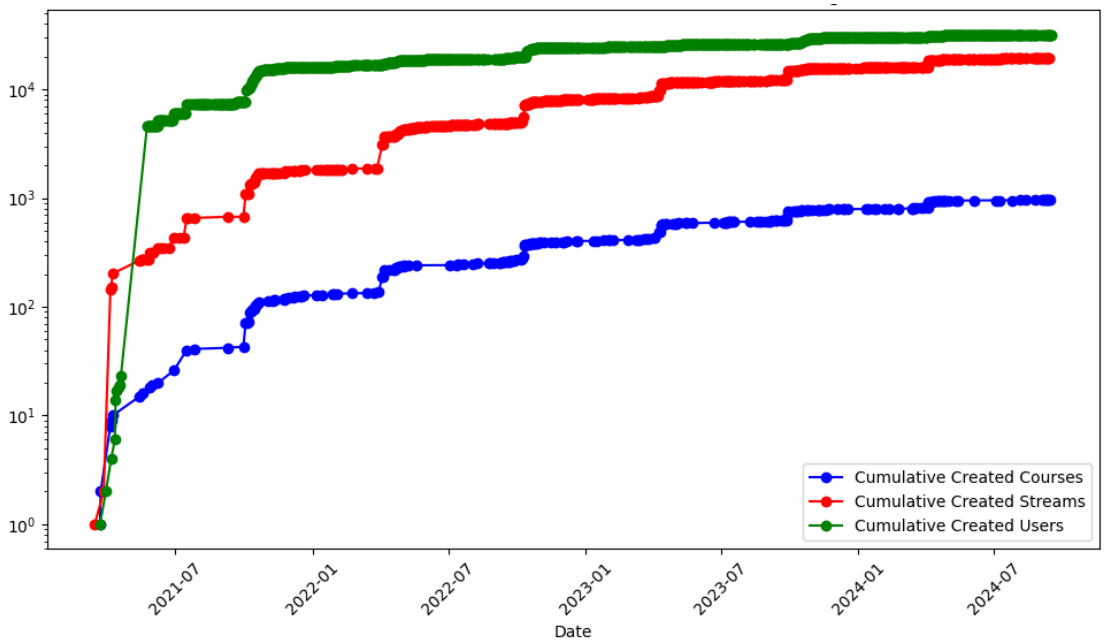

Figure 3.5: Cumulative Number of Users, Courses and Streams over Time (Log Scale)

4 Scaling Video Streaming Architecture

This chapter focuses on how video streaming actually works, what processes are involved in designing and deploying the infrastructure of a video streaming service and what its potentials and challenges with regard to scalability and performance are.

4.1 Video Streaming

The following section explains and compares the most important technical concepts of video streaming.

4.1.1 Video Streaming Types

The concept of video streaming was already present during the early years of television. However, nowadays, video streaming is mostly used to describe the transmission of video and audio data over the internet. In the following, the most important types of video streaming are described with their specific technical requirements and challenges:

- **Live Streaming:** This type involves real-time broadcasting of video content, often used for events, gaming and news. As mentioned in the overview of Twitch's architecture, live streaming requires a robust architecture capable of handling many users accessing the same content simultaneously while maintaining low latency. Technologies such as RTMP and HLS are commonly used. Ingest servers then process incoming streams, encode them in multiple bitrates, and distribute them through CDNs to the viewers.
- **Video on Demand:** VOD platforms (such as Netflix) allow users to browse and watch video content at any time. This type of streaming relies heavily on efficient storage systems and CDN to manage large content libraries and ensure quick access. VOD systems often use progressive streaming techniques, where large video files are segmented and streamed in small chunks as the user watches the video. This allows the user to start watching the video without requiring the entire video to be downloaded upfront, saving time and bandwidth, as only the segments that the user watches are downloaded [Sai+24].

- **Peer-to-Peer Streaming:** In Peer-to-Peer (P2P) streaming, viewers share the content they are streaming with others, reducing the load on central servers as the load is distributed to the viewers. P2P streaming can be difficult to manage because sophisticated algorithms are needed to optimize peer selection and data distribution and to ensure low latency and high reliability across the network [ZCL07]. This type of streaming is often used for decentralized news broadcasting and content sharing (e.g., BitTorrent Live).

- **Webcasting:** Lastly, there is also webcasting, which - to be precise - needs to be separated from the previous streaming types. While normal live streaming or VOD platforms serve a video in a 1:N relationship (1 Video is distributed to many viewers), webcasting serves video in an N:M relationship (all webcast participants are sharing their own videos with each other). As webcasting is mainly used for structured events like corporate events, webinars and online meetings (e.g., via Zoom, BBB, or Teams), maintaining a low latency of typically 150ms or less is very important. Higher latency values will result in noticeable delays between video and audio and make real-time communication in online conferences and meetings difficult (in comparison with normal live streaming, where a latency of up to 30 seconds will not be noticed by the viewers). It often involves additional features such as viewer authentication, interaction capabilities and detailed analytics [Sai+24].

4.1.2 Video Transcoding

Once a video has been received, it needs to be transcoded into different formats to ensure compatibility across platforms while maintaining as much of its original quality as possible. There are different types of transcoding; the most important are:

- **Standard Transcoding** involves changing the video compression standard of the video, e.g., by switching between different codecs such as H.265/HEVC to support different devices. It is important to note that when changing codec, it has to decode the bitstream with the old codec first and then encode it again with the new codec, making the entire operation very compute-intensive. [Gro+13].

- **Bitrate Transcoding/Transrating** adjusts the bitrate of a video stream to balance the quality of the video and subsequent bandwidth usage. For instance, a high-resolution video may need to be transcoded to a lower bitrate for users with limited bandwidth. ABR is often used for this, as multiple bitrate streams are generated, leaving the decision of which frame rate to select up to the client based on current network conditions [VCS03].

- **Spatial Transcoding/Transsizing** modifies a video's resolution to match the viewing device's capabilities. For example, a 4K video may be downscaled to 1080p or 720p for playback on devices that do not support higher resolutions [LDB20].
- **Frame Rate Transcoding** modifies the number of frames per second in a video file - usually by reducing the number of frames by removing frames in a certain interval so that the viewer does not notice a difference [VCS03].

Also, it is important to mention that there are two main compression classes used as part of the transcoding process:

- **Lossy Compression:** Compression algorithms of this class remove parts of a video's data to reduce its total file size. While this can affect the video quality, it decreases the overall bandwidth usage, as the video files are now smaller, making them easier to cache and requiring less data to be streamed to the end user. Popular lossy codecs include H.264[1], H.265[2] and VP9[3] [ET22].
- **Lossless Compression:** This compression class reduces file size without losing data, preserving the original quality. However, the compression ratios are typically much lower compared to lossy methods. Codecs like Apple ProRes[4] and FFV1[5] are examples of lossless compression used in professional environments such as film productions or studio-music recordings [ET22].

While Transcoding changes the file's contents, converting the video file format (e.g., from .avi to .mp4) is often referred to as Format Transcoding/Transmuxing. However, strictly speaking, this is not part of Transcoding as this operation changes the container and not the actual content, although it often includes re-encoding audio streams to formats like AAC or Opus, depending on the target platform [VCS03].

4.1.3 Delivery Networks

Delivery networks ensure that video content reaches end-users quickly and reliably. The two main approaches for this are CDNs and P2P networks:

CDNs are a network of geographically distributed servers that cache video content close to end-users. When a request is made (e.g., a user wanting to stream a video), it is

[1] `https://www.itu.int/rec/T-REC-H.264`
[2] `https://www.itu.int/rec/T-REC-H.265`
[3] Developed by Google and published 2013 to compete with H.265 and used mainly by YouTube. See also: `https://developers.google.com/media/vp9` and `https://www.webmproject.org/vp9`
[4] `https://www.apple.com/final-cut-pro/docs/Apple_ProRes.pdf`
[5] `https://github.com/FFmpeg/FFV1/blob/master/ffv1.md`

routed to the closest CDN node, which delivers the content with minimal latency. CDNs like Amazon CloudFront, Azure CDN and Google Cloud CDN are designed to handle massive amounts of traffic and can scale dynamically based on demand [Adh+12]. Technically, CDN works by replicating video content across multiple nodes in different locations. Each node (or edge server) holds a copy of the content, reducing the distance data must travel to the end-user, improving load times and reducing latency. Advanced CDNs also use techniques such as Anycast routing, where a single IP address is shared by multiple servers and requests are automatically routed to the nearest or least-loaded server [Adh+12].

As mentioned before, P2P streaming uses viewers' bandwidth to deliver content, reducing the load on central servers [Adh+12], as instead of relying on one central server, each user in the network shares parts of the video stream with others. However, P2P networks require complex algorithms to ensure optimal peer selection and data distribution. Algorithms like BitTorrent's Tit-for-Tat, which motivate users to share data by rewarding them with faster downloads[6], are often used for this. Additionally, to keep the latency as low as possible, P2P networks use buffer management strategies and protocols designed to minimize the delay in data exchange between peers [ZCL07].

4.1.4 Security

Most video streaming services work with licensed data such as movies or copyrighted music and must guarantee that their content is protected from unauthorized access, piracy and other malicious activities. There are three factors that need to be considered when evaluating the security of a streaming service:

First, ensuring that only authorized users can access the video content by using JSON Web Token (JWT) based authentication that generates a unique token for each session, or implementing access delegation mechanisms such as OAuth. What's important is that the authentication and access control is strictly applied on all levels - not just when a user accesses the streaming service, but also on an individual request basis to avoid an authorized user accessing forbidden scopes. A relevant example of this is streaming services that require authentication for a user to access the platform but then do not require any authentication for accessing specific content. A malicious user could exploit this to access hidden content or content for which he is not authorized. Also, using the example of a VOD platform, this could be used to download and share original high-resolution videos by checking the source URL of a video and accessing the full video file from there. Alternatively, there are also other attack vectors such as possible file and directory discovery where "*adversaries may enumerate files and directories*

[6] P2P clients may prefer sending data to peers that send data back to them, "rewarding" them with faster download speeds which reduce loading times and allow the viewer to increase the video resolution.

or may search in specific locations of a host or network share for certain information within a file system" [Cla17]. A simple solution to defend from this kind of access would be to ensure that any content on the site is only accessible with a valid JWT (or any other form of token) that is limited with regard to the scope and validity.

Secondly, to prevent piracy or any unwanted activities, one can use Digital Rights Management (DRM) tools like Microsoft's PlayReady[7], Apple's FairPlay[8] and Google's Widevine[9] protect content by controlling how it is accessed and used. DRM typically involves encrypting the content and then using licenses to grant authorized users the ability to decrypt and play the content. The encryption process usually involves Advanced Encryption Standard (AES) with 128-bit or 256-bit keys. When a user attempts to play DRM-protected content, the video player requests a license from a DRM server. If the user is authorized, the server provides the license, which includes the decryption key. The player then decrypts the content and plays it [JW11]. However, as many of these DRM services cost up to several cents per DRM license or several thousand dollars per year, one needs to consider if it is worth the additional effort and expense.

Lastly, protecting the infrastructure itself is critical. Beyond DRM, video streams are often encrypted during transmission (e.g., using Transport Layer Security (TLS)) to protect against interception. Additionally, standard security practices such as IP whitelisting, port restriction, the use of firewalls and regular security audits can help mitigate potential threats.

4.2 Cloud-Based Video Streaming

In 2008 Netflix started to migrate all its services to the cloud (AWS) to move away from vertically[10] scaled single points of failure, like relational databases, towards highly reliable, horizontally[11] scalable, distributed systems in the cloud [Cia10]. Since then, deploying and scaling services using third-party cloud providers has become the standard. The next section shows how this can be especially useful for a video streaming infrastructure.

[7] `https://www.microsoft.com/playready/documents`
[8] `https://developer.apple.com/streaming/fps`
[9] `https://developers.google.com/widevine`
[10] *vertically* refers to scaling up by increasing the current resources' power (e.g. upgrading GPUs or Storage)
[11] *horizontally* refers to scaling out by adding additional nodes or machines to the current infrastructure

4.2.1 Architecture

Cloud-based video streaming works on multiple layers, each responsible for a different step of the streaming process:

- **Content Storage:** Platforms like AWS S3, Google Cloud Storage and Azure Blob Storage offer scalable and reliable storage solutions for video content. These platforms use object storage systems designed to handle massive amounts of unstructured data. Data is often stored in multiple copies across different geographic locations to ensure durability and availability [LDB20]. (see also Table 4.1)
- **Encoding and Transcoding:** Cloud services like AWS Elemental MediaConvert, Google Cloud Video Intelligence and Azure Media Services handle the encoding and transcoding of video into various formats and bitrates. These services are designed to scale automatically depending on the volume of the processed content so that even large video collections can be encoded efficiently [LDB20].
- **Content Delivery Networks (CDNs):** As mentioned earlier, cloud-based CDNs, such as AWS CloudFront, Google Cloud CDN and Azure CDN are often used to deliver video content globally. [LDB20].
- **APIs and Microservices:** Cloud-based architectures often rely on microservices, with each service serving a specific purpose, such as user authentication, recommendation engines and analytics. These microservices communicate through APIs, which can be easily deployed, scaled and updated independently using systems cloud engines, such as Azure Kubernetes Service or Google Kubernetes Engine [LDB20].

4.2.2 Comparison: In-house Storage vs. Cloud Storage

When designing the architecture of a video streaming service, an important decision is to decide how and where the actual video data should be stored. The following table (Table 4.1) and subsection summarize the key characteristics of these three approaches in the context of video streaming.

In-House Storage involves managing physical storage infrastructure. This allows the owner full control over how the data is managed and allows also for highly customizable configurations, which can be a hard requirement for organizations bound to strict compliance regulations. However, it demands significant upfront capital expenditure and ongoing maintenance. Also, its scalability is limited by the physical infrastructure, requiring careful planning and potentially costly upgrades making short-term adjustments to handle flexible loads very difficult.

Cloud Storage, on the other hand, offers a flexible and on-demand scalable solution managed by third-party providers like AWS, Google Cloud, or Microsoft Azure. It allows video streaming providers to scale their storage needs dynamically without significant upfront investment or configuration effort, as most use cases are already covered by default configurations and extensive documentation and support if necessary. However, while often operating on a pay-as-you-go model, it comes at the cost of limited control over data and potential security concerns that are dependent on the cloud provider's policies. Cloud storage also introduces variability in latency depending on the location of data centers relative to users. While it is cost-effective in the short term, long-term costs and migration issues due to vendor lock-in can accumulate as data usage grows [Dar17].

While not exactly being an alternative storage method, **P2P** systems are highly scalable and can be a very cost-effective way to distribute and store data. However, they come with challenges related to security, data integrity and performance consistency, as they rely on the cooperation of peers [ZCL07]. Latency and quality of service can vary significantly depending on network conditions and peer availability, making P2P suitable for environments where cost savings are prioritized and some variability in performance is acceptable.

Table 4.1: Comparison of In-House Storage, Cloud Storage and P2P for Video Streaming

Factor	**In-House Storage**	**Cloud Storage**	**P2P**
Control	Full	Limited	Distributed
Security	High, customizable	Provider-dependent	Lower, peer-based
Scalability	Hardware-dependent	Virtually unlimited	Highly scalable
Costs	High upfront, low ongoing	Pay-as-you-go	Low infrastructure cost
Performance	Low latency	Variable	Variable
Management	High maintenance	Low maintenance	Complex, peer-managed
Redundancy	Customizable, local	High, managed by provider	Peer-based, variable
Compliance	Easier to customize	Provider-dependent	Difficult to enforce
Latency	Low	Moderate to high	Variable
Use Case	Enterprise	Scalable, flexible	Cost-sensitive, user-driven

4.3 Database Sharding

While the previous sections described general concepts of video streaming, this section is meant to focus on a more specific problem that services face when scaling up. With an increasing number of uploaded videos, there is the issue of running out of storage space (see also Subsection 2.4.2). This can easily be solved by buying either a larger storage system or extending the current storage system with a cluster of storage

systems. However, the options are more limited when running out of space in a relational database. One can still upscale the current database, but for large amounts of data - especially for services such as YouTube or Netflix - a single database is not enough. Adding additional databases can be a complex task, especially when trying to maintain Atomicity, Consistency, Isolation, Durability (ACID) properties. A possible solution to this is database sharding: a technique where large databases are distributed across multiple servers. In the context of video streaming, sharding can help store large amounts of content metadata, data generated by user interactions, and playback logs more efficiently.

4.3.1 Overview of Database Sharding

Database sharding, also known as horizontal partitioning, involves dividing a large database into smaller databases, so-called shards. Now the data can be processed and accessed in parallel, as each shard is stored on a separate server. This technique is especially useful for scaling databases that handle massive amounts of data, as is common in video streaming platforms. Database Sharding can be implemented in a shared-nothing architecture, where each shard operates independently, thus avoiding the contention issues typically associated with shared-disk clustered databases [BN15a]. This independence ensures that the failure of one shard does not affect the others, improving fault tolerance and availability.

A sharded database architecture divides data among multiple data nodes based on a partitioning scheme. Some of the most common partitioning strategies include range-based sharding, where data is divided based on the value range of a key and hash-based sharding, where a hash function is applied to a key to determine the shard placement [BN15a]. This allows the distribution of the data in a more balanced way and can significantly improve read and write operations, as each server now only needs to manage a reduced amount of data.

4.3.2 Advantages of Sharding

One of the main advantages is scalability, as with increasing data, new shards can be added without requiring significant changes to the application or database architecture. This would allow video streaming platforms to handle increasing loads efficiently, whether it be due to more users, more content, or both [BN15a].

Another advantage of sharding is fault tolerance. If one shard becomes unavailable or has a data loss due to any kind of hardware or network failures, the replicated data on another available node can take over, ensuring uptime of service, as the data is typically replicated across multiple nodes [BN15a].

Sharding also improves manageability and maintainability by dividing the database into smaller, more manageable units. Database admins can perform maintenance tasks, such as migrations, backups or schema updates on individual shards without affecting the entire system. This modular approach reduces the risk of large-scale disruptions [BN15a]. Of course, there remains the risk that a misconfiguration in one database can lead to errors across the systems, and unnoticed errors may spread from one database to others, complicating the debugging and repair of the databases.

4.3.3 Challenges and Considerations

The most critical challenge is maintaining consistency across shards. In a distributed environment, ensuring that all shards reflect the most recent data state can become difficult, especially in case of network partitions or node failures. To try to counter this, techniques such as distributed transactions and eventual consistency models are often used, but they will again add additional complexity to the system [Sun+08].

Another challenge is re-balancing shards as the data grows or usage patterns change. Over time, some shards may become "hot" (i.e., they handle a disproportionate amount of traffic compared to other shards), potentially leading to new performance bottlenecks. Re-balancing would require redistributing data across shards to ensure even load distribution [BN15b].

Lastly, sharding complicates query processing leading to potentially slower execution of cross-shard queries, where data from multiple shards needs to be processed. While there are special approaches to optimize distributed queries such as *Reference/Distributed Table Joins* or *Remote Distributed Table Joins*, careful query planning and optimization are still required to minimize the impact [Pro17].

To conclude this section, it is important to repeat that database sharding is a great option to scale database systems while improving fault tolerance and manageability. But at the same time, it also introduces challenges such as consistency management, shard re-balancing, and complex query processing. In the end, most companies or services with database scalability problems will more likely increase their current database compute and storage power - and worry about other approaches, such as sharding only when they reach a limit with their current system [PA16].

5 Scaling TUM-Live

This chapter focuses on applying the concepts discussed in the previous sections to the architecture of TUM-Live, discussing the development approach, results and challenges along the way.

5.1 Process, Preparation, Methods and Environments

The thesis spanned 5 months, from 15.05.2024 to 15.10.2024. The first weeks were spent familiarizing with the current system considering different approaches to scale its architecture and finding potential bottlenecks or issues. Following the initial analysis of the system, before being able to try scaling individual components, the original user and role system first needed to be updated together with new database models. After that, the most important components were gradually updated to be usable and manageable by different organizations. To develop and test the prototype of a distributed architecture for TUM-Live for this thesis, the following resources were used:

- 3 Virtual Machine (VM)s with: 2 GB RAM, Intel(R) Xeon(R) CPU E5-2697A v4 @ 2.60GHz
- 1 VM with: 20 GB RAM, AMD EPYC 7452 32-Core Processor
- 1 AWS Elastic Kubernetes Service (EKS) Cluster
- 1 selfhosted VM with 32 GB RAM, AMD Ryzen 7 PRO 6850U @ 2.70GHz

After the target architecture had been deployed (on a smaller scale) using given resources, a set of performance tests and comparisons was made to find potential limits and breakpoints of each component. Additionally, in parallel to the development of the new architecture, a dedicated documentation has been created to facilitate the setup of GoCast for lecturers or new schools, which can be found at tumlive-docs.pages.dev. The documentation was created using Meta Opensource's Docusaurus[1]. Static pages were deployed using Cloudflare. All relevant source code for the thesis, new architecture, other mentioned prototypes and documentation can also be found at github.com/carlobortolan/thesis.

[1] `https://github.com/facebook/docusaurus`

5.2 Proposed System

As explained in Section 3.2, to use TUM-Live, currently, each lecture hall needs to be equipped with a SMP device, which can cost up to 10,000 EUR. With the creation of the VMP, which costs only around 500 EUR per device, there is the demand for scaling TUM-Live to other TUM schools and possibly other universities. However, currently, this is not possible for the following reasons:

1. **High maintenance** (not enough human resources to provide a central support);
2. **TUM-Live can only be centrally hosted**;
3. **No support for multiple organizations**;
4. **"Flat" user structure** (no way to enforce ownership rules over lectures, users, etc.);
5. **Complex and not user-friendly features** (e.g., self streaming).

The following sections will explain a proposal to solve all five issues (see also Figure 5.1) and scale TUM-Live to handle lectures from other organizations.

5.2.1 Target System Architecture

To distribute TUM-Live to different schools and universities, the subsystems and components responsible for processing and storing video data need to be distributed and hosted by each individual organization. The main TUM-Live API instance however, will remain managed by the CIT IT Operations (ITO) or TUM so that users have a single point of access instead of having to switch instances when wanting to watch lectures of different schools. To achieve this, each organization needs to host at least three components: the Worker component, the VOD Service component and an Edge Server. Each organization can decide how many resources it wants to allocate to each service depending on the expected load. The following minimum requirements are set:

- At least 1 VM as an Edge Server. This server serves the videos to the users. Throughput is important, so, to serve many users, more instances are needed.
- At least 1 Worker VM to receive the stream and transcode the VOD. On the same node, for every Worker a VOD Service needs to be deployed to expose a simple HTTP interface that accepts file uploads and packages them to a HLS stream in a configured location. This stream may then be distributed by the Edge Server.
- Optionally, an organization can add additional VMs for monitoring (Grafana[2], Prometheus[3], etc.) or for deploying services such as the Voice Service for subtitling live streams and VODs using the Whisper LLM[4].

[2] `https://github.com/grafana/grafana`
[3] `https://github.com/prometheus/prometheus`
[4] `https://github.com/openai/whisper`

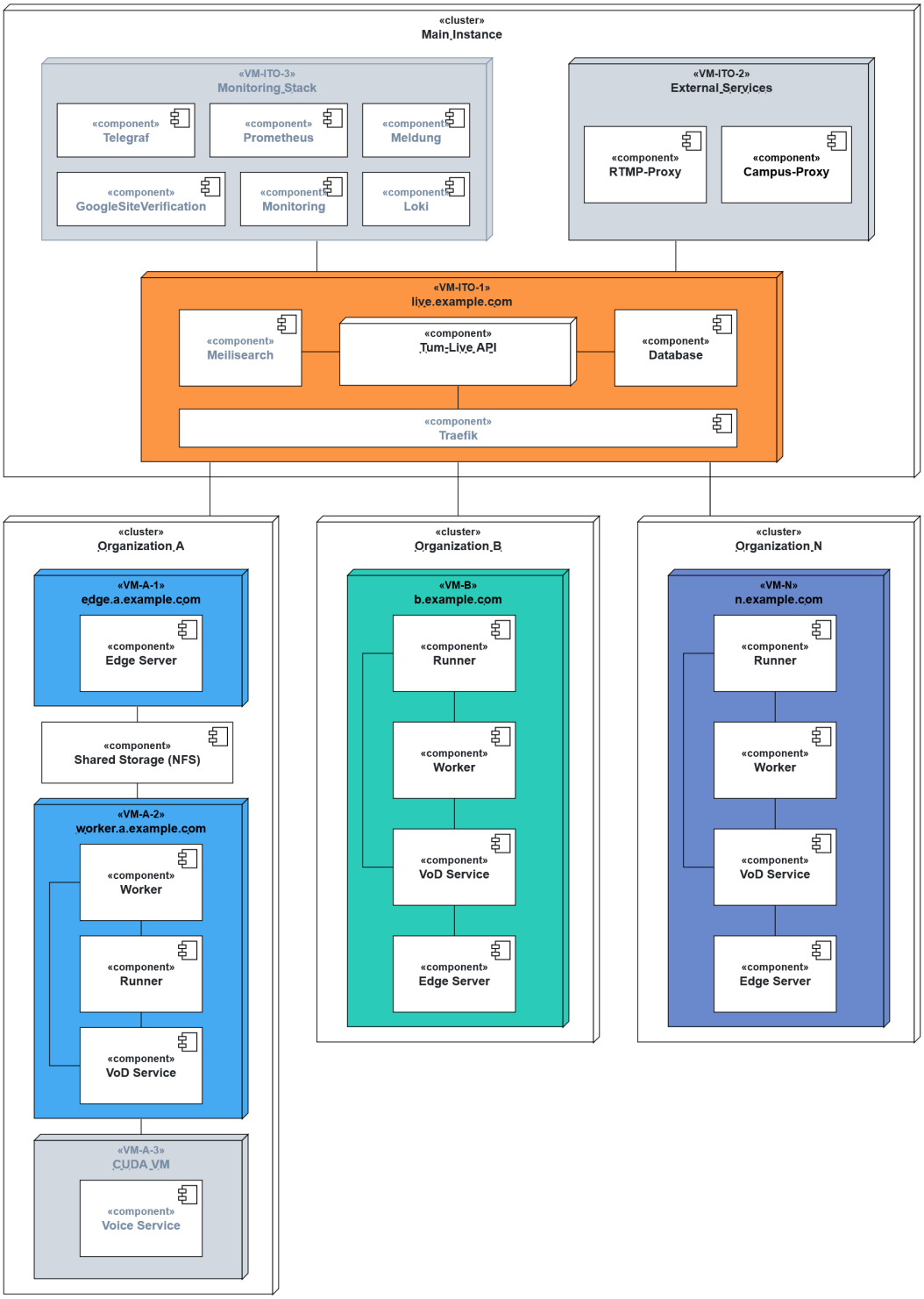

Figure 5.1: Target Deployment Diagram of TUM-Live[5]

5.3 Delegated Administration of Resources

Given the proposed system (see Figure 5.1), this section documents how this system has been implemented and what the current limitations and potential improvements of the main user system are.

5.3.1 Updating Architecture and Core API

As explained before, with increasing demand, GoCast must be extended for university-wide lecture streaming. The solution to this are **organizations** that allow for delegated partial administration of resources.

The first step of implementing this structure was updating GoCast's role system. Previously, it contained four roles: `admin`, `lecturer`, `student`, `visitor`. While admins can manage the entire system (e.g., create and delete lectures and users, as well as perform maintenance tasks), lecturers can only manage their own or create new lectures and students only manage their own profiles and preferences. However, with the introduction of organizations, this is not sufficient, as each organization needs to have its own organization-scoped admins that can manage their organizations' resources without being able to interfere with other organizations. To do this, a new `maintainer` role has been introduced (see Figure 5.7). With this, an organization using GoCast is managed by a set of maintainers of this organization. A user with the `maintainer`-role can be the maintainer of multiple organizations and also has maintainer rights for all sub-organizations of his organizations. Maintainers also have some basic administrative functionality that is limited to their organizations' scope (e.g., create, update and delete courses and streams only for those organizations that are administered by that maintainer).

5.3.2 GoCast Organizations

TUMOnline has a strict hierarchical structure for its organizations (one school has multiple departments; one department has multiple chairs; one chair has multiple courses ...). While TUM-Live is mainly used by the TUM, in principle it does not need to differentiate between organizational types that strictly. Organizations are only relevant when it comes to distributing the live streams and recordings of a certain entity to that entity's resources (e.g., Workers and VOD Services). Hence, the introduction of GoCast's organizations which represent an entity responsible for processing data. In practice, this is most of the time a TUMOnline school, however, one can also

[5] In comparison to a *System Architecture Model* (e.g., Figure 2.1 or Figure 5.1), a *Deployment Diagram* shows a concrete instance of an abstract system architecture.

create a GoCast organization for a department, chair or smaller organization which is subordinated to another organization, depending on the specific situation.

Here is an example to illustrate this in a more detailed way: The TUMOnline "School of Management" (SOM) wants to start using TUM-Live. Hence, the SOM's IT team contacts the admins of TUM-Live who then create a new *SOM organization* in TUM-Live and assign the SOM IT team as maintainers. The subordinated "Chair of Financial Management and Capital Markets" (FA), however, has its own data center and wants to host its lectures with its own resources. In this case, either one of the SOM maintainers or the CIT ITO can create a new organization in TUM-Live as a sub-organization of the *SOM organization* and accordingly assign new maintainers from the FA-team. Now, the FA-maintainers have full control over their sub-organization and can connect their own resources from their data center with TUM-Live, independently of the SOM.

5.4 Distributed Resources

Now that the user role system had been updated, the next step was to find a solution to have the different resources, such as Workers connected to the main cluster of Workers independent of the other organizations' resources. However, at the same time, they should be able to process and distribute requests between each other regardless of the organization they are in. This section explains in detail how each subsystem works as part of the distributed GoCast system.

5.4.1 Workers and Runners

To set up a distributed network of Workers, the first step was to update the system in such a way that Workers could be connected by an organization's maintainer to the main network of GoCast. To do this, an organization's maintainer can create a new organization-token, a JWT that expires after seven hours and allows its owner to connect new resources for the organization.

When a maintainer starts a new Worker, he can include this token in the container's environment (e.g., via a `docker-compose.yml` file or by passing it as an argument directly to Docker using `-e Token=<...>`). When starting up, the Worker sends a request to join the Worker pool of GoCast using the organization-token and additional data (e.g., host name, IP or FQDN address, VM workload, etc.) and - if successful - receives a token without expiration which is then used to validate all subsequent requests from and to the Worker. Maintainers can then see and manage the current status of their registered resources in the Resource-Dashboard of GoCast (see Figure 5.2).

Now, whenever a new lecture is being received or uploaded, the main GoCast API selects an available Worker for this organization and addresses it, given the hostname

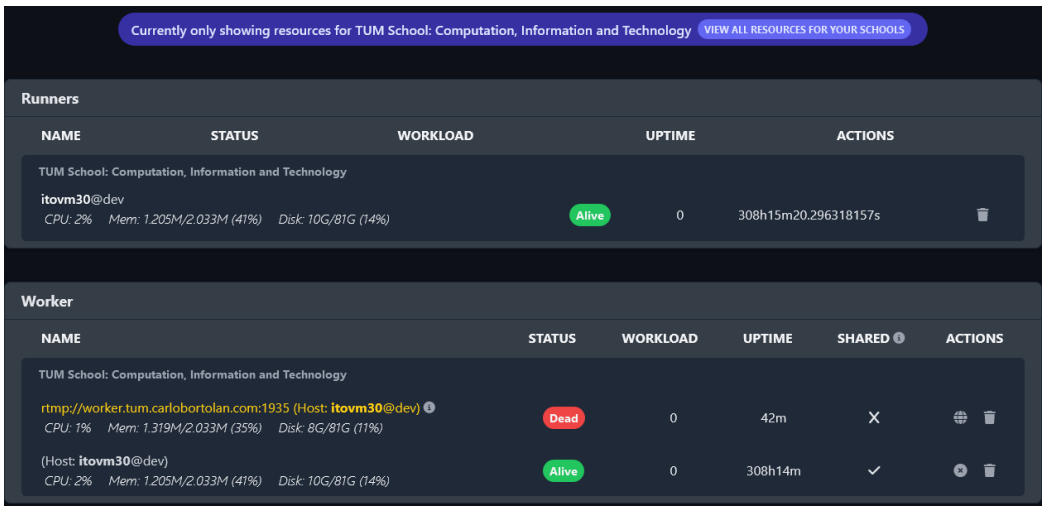

Figure 5.2: Resources Dashboard

and IP address that the Worker has registered when first connecting to GoCast. For more details on how the distribution of tasks works, see Section 5.5.

When the Worker receives such a request, it initiates the video transcoding process using FFmpeg. The transcoding is performed based on the specific stream version[6]. The process is monitored in real-time, with progress being reported back to the GoCast system, to ensure that any failed actions are retried with backoff strategies and that the final transcoded video meets the expected duration and quality standards.

5.4.2 VOD Service and Edge Server

Once the Worker has completed the transcoding process, the VOD Service starts detecting silent sections in the recordings so that the user will be able to skip these and jump directly to the start of the actual lecture. Then, it packages them into a HLS stream and copies the packaged stream to the organization's recording storage.

To ensure easy access to these recordings, each organization also needs to have its own Edge Server set up. The VOD Service at each organization is responsible for managing the processing, packaging and storage of the recordings server-side, while the Edge Server is responsible for distributing the streams to the end users. This also has the advantage that each organization maintains full control over the storage of its

[6] `CAM` streams use a higher compression level (CRF 26) and lower priority (niceness 10), while `PRES` streams use lower compression (CRF 20) and higher priority (niceness 9), with specific tuning in FFmpeg for still images in presentations. `COMB` streams, are assigned moderate compression (CRF 24) and the highest priority (niceness 8).

streaming data. Additionally, the Edge Server acts as a local cache for an organization's content. When students access recordings, the Edge Server distributes the content directly from the organization's storage, reducing the load on the organization's servers.

To allow for rapid post-processing and upload of VODs after a stream has ended, it is necessary that an organization has enough Workers to support the number of recorded lectures. If an organization expects many concurrent viewers, it might need to consider having multiple Edge Servers running, as they have a limited bandwidth. When network traffic to an Edge node exceeds the available bandwidth, the architecture might look like the example shown in Figure 5.3.

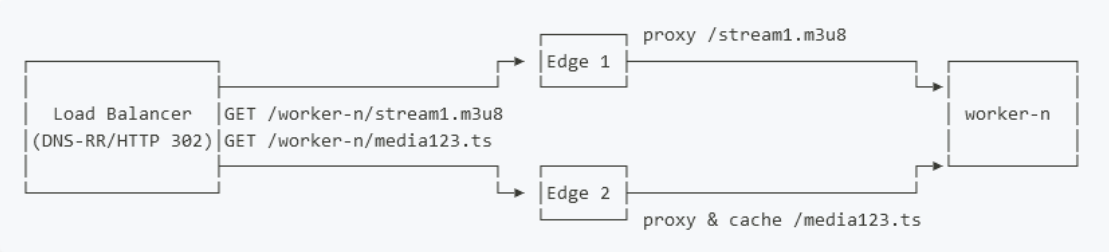

Figure 5.3: Edge Server as Proxy and Cache Node

5.4.3 RTMP-Proxy

Limitaitons of GoCast's Selfstreaming

GoCast supports not just recordings from the lecture hall or manual uploads but also so-called self-streams. The idea behind self-streams is that any lecturer can go live at any given time and stream from his own device without having to be in the lecture hall. In past, to do this, a lecturer had to go open TUM-Live, go to one of his courses' page, select a lecture and copy a link (such as: `rtmp://worker.example.com/cs-123?secret=5caa9d6447564cb5822995888e224f9a`) together with a secret stream key into a streaming software like OBS Studio. With the introduction of the organization system, there are several reasons why this cannot work:

1. The URL might change depending on the availability of Workers.
2. Having to copy and paste a new URL every time a lecturer wants to start a stream can become annoying.
3. For lecturers who teach at multiple organizations (e.g., a mathematics professor teaching lectures at the CIT and at the School of Engineering and Design), managing different URLs for each organization would be inconvenient.

Proposed Solution and Data-Flow

The solution for this is the RTMP-Proxy microservice. In a nutshell, the RTMP-Proxy acts as a router-like service that accepts all RTMP self-stream requests and redirects them to an organization's Worker depending on the stream's course. With this, all a lecturer has to do is create a new personal token (see Figure 5.4) and copy the token together with the target URL of the RTMP-Proxy into a streaming software only once. After this initial setup, the lecturer can go live whenever they want by simply using the pre-configured URL and token.

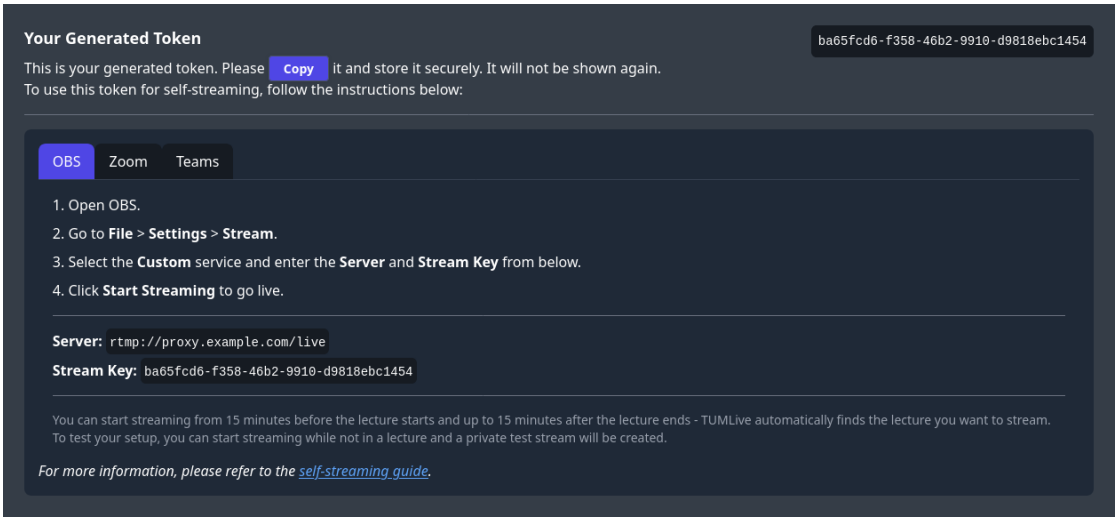

Figure 5.4: Creation of Personal Token in TUM-Live and RTMP-Proxy URL

The internal process of the RTMP-Proxy is as follows (see also Figure 5.5):

1. An incoming RTMP request is received:
   `rtmp://proxy.example.com/live/<personal-token>`.
2. The RTMP-Proxy sends a self-stream request to the main GoCast API with the provided `personal-token`.
3. The GoCast API validates the `personal-token` and uses it to detect whether the lecturer has a scheduled lecture for the current time slot or a soon upcoming lecture and automatically starts the live stream with a waiting screen for viewers. Then it checks the stream's course's organization and returns the RTMP URL of the ingest Worker currently available for that organization together with the course-slug and secret stream key for this stream:
   `rtmp://worker.example.com/<course-slug>?secret=<secret>/<secretKey>`.

4. The RTMP-Proxy redirects the RTMP request to the retrieved RTMP URL.
5. The organization's ingest Worker now receives the proxied RTMP request as if it were sent directly to it by the lecturer's streaming software.

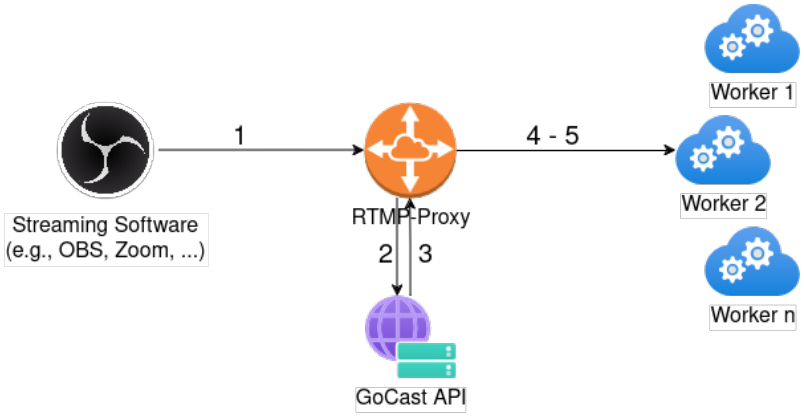

Figure 5.5: RTMP-Proxy Data Flow

Architectural Challenges and Limitations with RTMP

Initially, for the prototype of the RTMP-Proxy the idea was to develop a microservice similar to the Edge Server or VOD Service written in Go that would redirect the RTMP request (similar to an HTTP 302 redirect). However, after some testing, it quickly became obvious that Go does not yet have libraries to handle RTMP requests with the necessary granularity. Hence, to proceed with this approach, it would require to use Go's native libraries to:

- Handle the incoming RTMP request
- Perform a RTMP handshake
- Extract the user token from the RTMP request
- Find out the destination address
- Establish a connection with the destination host
- Streaming the video chunks directly to the destination address

This might work short term, but in the long term, it would lead to other problems with potential bugs, scalability issues, complex error handling and unknown security vulnerabilities if not properly maintained.

A better solution was to use the nginx-rtmp-module[7] that extends Nginx to handle RTMP streaming requests natively [Kho21]. Now, whenever a stream request arrives at the proxy server, Nginx accepts the request and invokes a script that extracts the stream key (the personal token) and requests the actual destination URL from the GoCast API. Upon receiving the destination URL, Nginx pushes the stream to the target server.

Implementation Details: FFmpeg vs. OBS Compatibility

During the testing phase, it was observed that at times streaming via FFmpeg worked perfectly fine, while attempts using OBS failed:

- **Encoding Mismatch**: FFmpeg explicitly specifies H.264 for video and AAC for audio. OBS, however, may set different default codecs. The target RTMP server of the Worker required H.264 video and anything else would cause a rejection.
- **Keyframe Interval**: Workers generally expect a keyframe interval of around 2 seconds. OBS settings needed to be adjusted to enforce this interval.
- **Bitrate Mismatch**: High bitrate settings in OBS were causing overloads, which led to connection issues with the target RTMP server. Lowering the bitrate resolved the issue.
- **URL Composition**: The self-streaming URL used to call the proxy is structured like: `<protocol>://<host>:<port>/live/<stream_key>`. OBS, however, requires the URL and stream key to be set separately (see Figure 5.6).

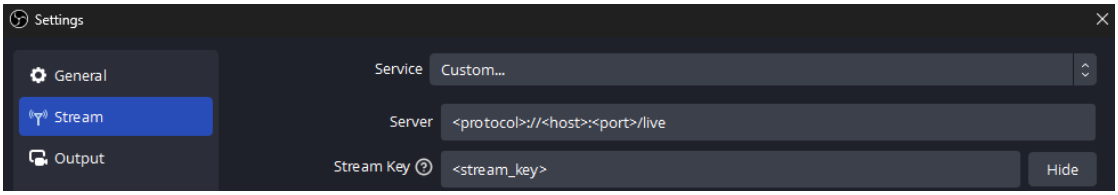

Figure 5.6: OBS Settings for Self-Streaming

This was later fixed by changing the settings in OBS and setting the video encoder to `x264` (H.264), the audio encoder to `AAC`, adjusting the bitrate to a more reasonable level and settings the keyframe interval to 2 seconds for better compatibility with the Nginx RTMP server.

[7] `https://github.com/arut/nginx-rtmp-module`

Final Words on Scaling the RTMP-Proxy

While the RTMP-Proxy is far from perfect, it is meant as a prototype of a possible solution to stream via RTMP using OBS in a distributed network where the final target address of the stream request is unknown when making the request. As the RTMP-Proxy is independent of other services in the GoCast environment (besides the main API to receive the destination URL), it can be scaled very easily by deploying multiple RTMP-Proxies and having, for example, a Round-robin DNS distribute the load accordingly.

5.5 Shared Resources

This section explains different approaches for the allocation of resources in the GoCast network. The main focus is on the task distribution to the available Workers when receiving new video uploads or stream requests.

5.5.1 Shared Resources

Before explaining how resources in the GoCast Network are dynamically allocated to different tasks, it is first important to explain in detail how an organization actually "owns" a certain resource such as a Worker. As described in Section 5.3, organizations are structured in a hierarchical manner. One organization can have multiple sub-organizations, which can again have multiple sub-sub-organizations and so forth. When an organization has deployed its own resources, by default, accessible only by the organization itself and its sub-organizations. Using the example shown in Figure 5.7, this would mean that if there is a lecture uploaded by *School 1*, only public resources and the organization's own private resources are considered. For a lecture of *School 2a*, all public resources, the organization's own private resources, as well as its parent organization's resources (and recursively so on) are considered.

Additionally, for certain resources, for example, Workers, one can decide to set the `Shared` flag. When a resource has that flag enabled, it means that even though it is part of an organization (and hence by default only available for lectures of that organization or sub-organizations in that organization's hierarchy), it now becomes available to all organizations and shares its compute power with the entire network of organizations. To use the previous example, assuming that *School 2a* has allowed to share its resources, when *School 1* uploads a video or starts a stream, in addition to its own private resources and the public resources, it also considers the shared resource of *School 2a*.

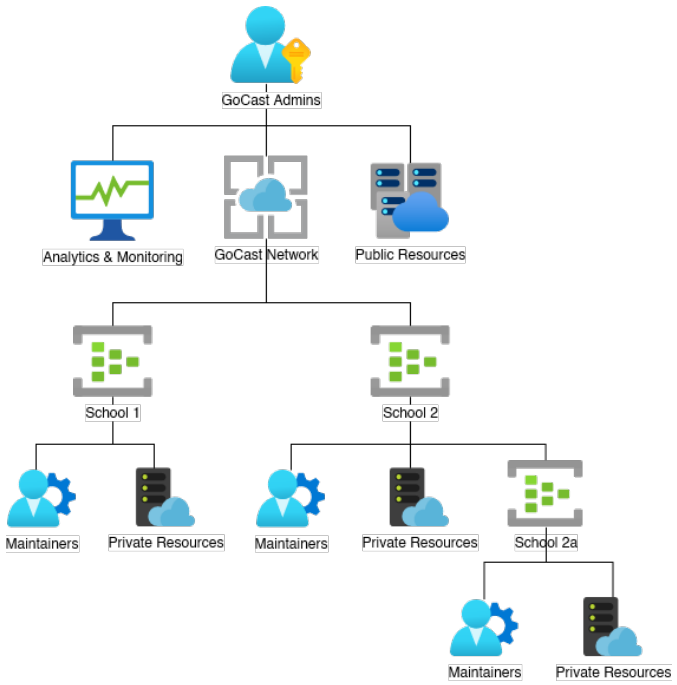

Figure 5.7: Example Organization Hierarchy

5.5.2 Four Approaches for Task Distribution

The following are four approaches considered for the decision algorithm behind the allocation of resources - especially of Workers - in the GoCast Network. The current implementation for the task distribution of Workers uses the second approach. For the Runners, it remains to be seen which approach will be chosen - most likely a combination of the second and third approaches.

1. Randomized Allocation

Tasks are assigned to available resources randomly. This method is simple and helps to distribute tasks across resources in an unbiased manner. However, it only considers an organization's own and public resources, ignoring the hierarchical structure of organizations or resource workloads. Randomized allocation can be effective if resource capabilities were relatively uniform, but it would most likely lead to inefficiencies if the networks got more complex.

```
-- 1. Randomized allocation example query
SELECT *
FROM workers
WHERE shared = TRUE OR org_id = ?
ORDER BY RANDOM()
LIMIT 1;
```

2. Recursive Organization-Based Allocation

When a task is started, the system recursively checks for available resources, starting from the organization's own resources and moving up the hierarchy to include parent organizations and shared resources as described in the previous subsection. This guarantees that each task is allocated to the most appropriate resource, improving resource utilization across the network by allowing sub-organizations to benefit from the resources of their parent organizations while respecting the resource ownership of each organization. One could implement different optimizations, such as prioritizing one's own resources before shared resources or limiting how many levels the query should follow the parent-child relationship upwards. Also, one organization might not want to use resources shared by other organizations and, therefore, include only resources from its own organization's hierarchy. The code snippet below is an example of how a recursive query might look to find all available Workers in an organization's hierarchy, including shared Workers.

```sql
-- 2. Recursive organization-based allocation example query
-- First, create a recursive query to follow the parent-child hierarchy upward
WITH RECURSIVE org_hierarchy AS (
    SELECT id, parent_id FROM orgs WHERE id = ?
    UNION ALL
    SELECT o.id, o.parent_id FROM orgs o
    INNER JOIN org_hierarchy oh ON o.id = oh.parent_id
)
-- Then, select all Workers that are in the hierarchy or shared
SELECT w.* FROM workers w
LEFT JOIN org_hierarchy oh ON oh.id = w.org_id
WHERE oh.id IS NOT NULL OR w.shared = true;
```

3. Priority-Based Allocation

Tasks are assigned to Workers based on a dynamically adjusted priority queue. The tasks would be prioritized based on factors such as viewer count, failure rate, VM workload or the type of content being processed. For example, live streaming tasks could be given a higher priority over VOD uploads and hence be processed by a Worker that runs on a more powerful VM. This approach would assign critical tasks (e.g., live streaming) first and re-schedule less urgent tasks (e.g., VOD uploads or thumbnail generation) for later. The example below shows one possible implementation of the selection process, assuming that the `taskQueue` contains tasks sorted by priority.

```
// 3. Priority-based allocation pseudocode
function processTaskQueue(taskQueue) {
    while not taskQueue.isEmpty() {
        task = taskQueue.dequeue() // Get task with highest priority
        worker := getNextAvailableWorker() // Find available Worker
        if worker != nil {
            worker.assignTask(task)
        } else {
            taskQueue.enqueue(task) // Requeue task if no Workers are available
        }
    }
}
```

4. Linear Programming Optimization Approach

By solving a Linear Program (LP), the tasks can be distributed evenly among Workers, minimizing the maximum load on any single worker and reducing bottlenecks.

1. **Variables and Parameters**

   Let:

   - n be the number of tasks.
   - m be the number of Workers.
   - t_i be the processing time of task i estimated using past averages or heuristics based, for example, on the scheduled stream's duration.
   - x_{ij} be a binary decision variable, where $x_{ij} = 1$ if task i is assigned to Worker j and $x_{ij} = 0$ otherwise.
   - L_j be the total workload on Worker j, calculated as $L_j = \sum_{i=1}^{n} t_i \cdot x_{ij}$.

2. **Objective Function**

   The goal is to minimize $L_{\max}$ across all Workers:

   $$\text{Minimize } L_{\max} \tag{5.1}$$

3. **Constraints**

   The LP is subject to the following constraints:

   a) **Task Assignment:** Each task i must be assigned to exactly one Worker:

   $$\sum_{j=1}^{m} x_{ij} = 1 \quad \forall i = 1, \ldots, n \tag{5.2}$$

   b) **Load Calculation:** The workload of each Worker j is the sum of the processing times of the tasks assigned to that Worker:

   $$L_j = \sum_{i=1}^{n} t_i \cdot x_{ij} \quad \forall j = 1, \ldots, m \tag{5.3}$$

   c) **Bounding the Maximum Load:** The workload on each Worker must be less than or equal to the maximum load $L_{\max}$:

   $$L_j \leq L_{\max} \quad \forall j = 1, \ldots, m \tag{5.4}$$

   d) **Binary Variables:** The assignment variables x_{ij} are binary:

   $$x_{ij} \in \{0, 1\} \quad \forall i = 1, \ldots, n \quad \text{and} \quad \forall j = 1, \ldots, m \tag{5.5}$$

4. **Linear Programming Formulation:**

   Given the objective function and the constraints, the solution to the LP can be formulated as:

$$\text{Minimize} \quad L_{\max} \tag{5.6}$$

$$\text{subject to} \quad \sum_{j=1}^{m} x_{ij} = 1 \quad \forall i = 1, \ldots, n \tag{5.7}$$

$$L_j = \sum_{i=1}^{n} t_i \cdot x_{ij} \quad \forall j = 1, \ldots, m \tag{5.8}$$

$$L_j \leq L_{\max} \quad \forall j = 1, \ldots, m \tag{5.9}$$

$$x_{ij} \in \{0, 1\} \quad \forall i = 1, \ldots, n \quad \text{and} \quad \forall j = 1, \ldots, m \tag{5.10}$$

5. **Example Implementation:**

```
# Using Gurobi as an example - any other solver also works
from gurobipy import Model, GRB, quicksum
model = Model("Minimize_Max_Load")

# Decision variables: x[i, j] = 1 if task i is assigned to Worker j, 0 otherwise
x = model.addVars(n, m, vtype=GRB.BINARY, name="x")

# Variable for maximum load across all Workers
L_max = model.addVar(vtype=GRB.CONTINUOUS, name="L_max")

# Objective: Minimize the maximum load L_max
model.setObjective(L_max, GRB.MINIMIZE)

# Constraint 3.a) Each task is assigned to exactly one Worker
for i in range(n):
    model.addConstr(
        quicksum(x[i, j] for j in range(m)) == 1, name=f"Task_Assignment_{i}")

# Constraint 3.b) and 3.c) Load on each Worker does not exceed L_max
for j in range(m):
    model.addConstr(
        quicksum(t[i] * x[i, j] for i in range(n)) <= L_max,
        name=f"Worker_Load_{j}")

# Run the model
model.optimize()
```

6 Performance Optimization and Error Analysis

This chapter analyses the performance of the Workers, based on collected TUM-Live log and error metrics and presents an optimization approach for the main GoCastAPI that compares the current REST API with a prototype of a gRPC API.

6.1 Monitoring Stack

The GoCast system uses a rather complex stack of monitoring, logging and analytics tools to collect metrics. The main technologies include Grafana, Prometheus, Telegraf and InfluxDB, which are deployed in a containerized environment using Docker and managed by Traefik as a reverse proxy. The collected data is then organized into Grafana dashboards. The metrics collection and logging system for TUM-Live is based on the following main components:

- **Grafana**[1]: This is used as the main dashboard for visualizing metrics and performance of connected services in real-time.
- **Prometheus**[2]: As a monitoring and alerting toolkit, Prometheus is responsible for scraping metrics from the various services within the system.
- **Telegraf**[3]: Collects metrics and logs from Docker containers, the host system and various applications. It works together with InfluxDB to store time-series data.
- **InfluxDB**[4]: A time-series database, InfluxDB is utilized for storing logs and metrics collected from Telegraf, which are then visualized through Grafana.
- **Loki** and **Promtail**[5]: These tools are used for centralized logging, allowing for real-time querying and log analysis through Grafana's Loki integration.

[1] `https://github.com/grafana/grafana`
[2] `https://github.com/prometheus/prometheus`
[3] `https://github.com/influxdata/telegraf`
[4] `https://github.com/influxdata/influxdb`
[5] `https://github.com/grafana/loki`

6.2 Metrics of the GoCast API

The main data sources for the GoCast API are Loki and Traefik, as well as internal server logs. This section will give some insights on common errors and patterns found in the main API and (given the limited amount of logging data) focus on two main aspects: The number of errors and the observed error types. The Python scripts for the plots for this and the following sections be found at github.com/carlobortolan/thesis/analytics.

6.2.1 Error Volume Analysis

Plotting the logged errors of the API over time (see Figure 6.1) shows an increase in the sum of total API errors with a constant, positive slope. This indicates that errors were logged at a relatively similar rate throughout the observed period. The fact that there are no obvious sudden spikes or drops in this data suggests a continuous, recurring issue rather than isolated incidents. Since the slope did not flatten over time, it shows that the error volume did not escalate or improve significantly over time, implying that either no major fixes were implemented to resolve the underlying causes during this time frame or new bugs were introduced with prior fixes.

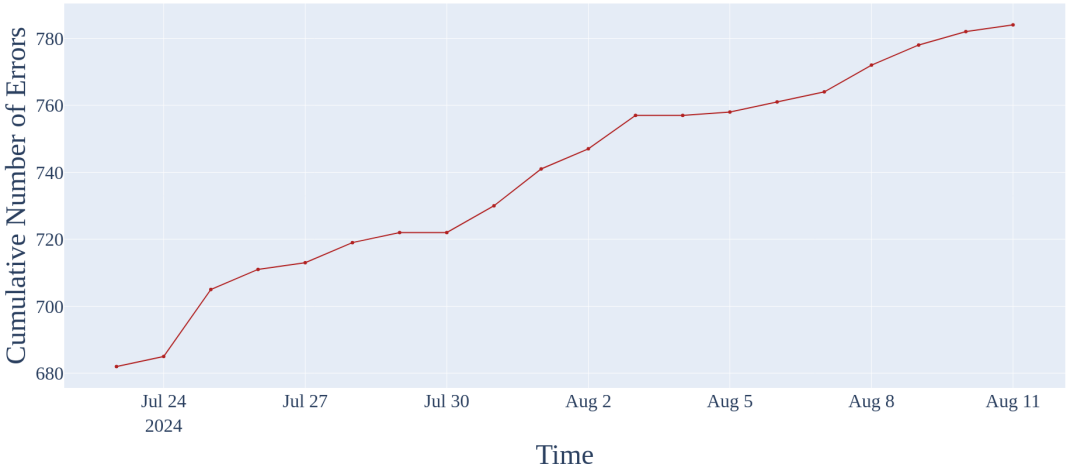

Figure 6.1: Cumulative API Errors over Time

6.2.2 Categorization of Error Types

The errors are not logged with a specific error type, but they contain a JSON String from which one can infer the origin of the error (e.g., see Figure 6.2).

```
{
  "time": "2024-07-23T06:38:08.282288184+02:00",
  "level": "ERROR",
  "msg": "Could not generate live preview",
  "service": "api",
  "err": "rpc error: code = Unknown desc = exit status 1"
}
```


Figure 6.2: Example API Error JSON

As seen in Figure 6.3, the most common error logged by the API was *'Could not generate live preview'* with 677 occurrences. To understand this error, first, it is necessary to explain that GoCast allows admins to trigger certain cron jobs, which then periodically perform a certain action, such as importing course assignments from CAMPUSonline or, in this case - fetching the thumbnails of live streams (Live Previews) from the GoCast Workers. Hence, this error is a result of the `FetchLivePreviews` cron job, which is performed every minute to get a live thumbnail from a Worker. If this request returns an error, it logs the error, ends the connection and decreases the Worker's workload. Possible reasons for this could be network issues preventing communication with the Worker, general Worker unavailability, or invalid or incomplete parameters passed to the `getLivePreviewFromWorker` function.

Figure 6.3: API Error Types

When aggregating the errors by their hour and plotting them in a bar chart grouped by error type (Figure 6.4), there is an interesting observation that the *'Could not generate live preview'* error, which is the most common error, mainly occurs during normal lecture times from 6AM to 5PM. As the number of reported occurrences of this error is constant over these hours, it can be assumed that it is either an issue with an individual Worker that might have a connection issue or is running an outdated docker image (the log data does not provide information on which VM the error occurred), or a bug that affects all Workers. However, since we know that the API fetches the previews once every minute from all Workers, and the number of errors per hour is around 60, it is most likely an issue with one individual Worker. If the issue was a random connection error, the number of occurrences should vary significantly and if the issue was a system-wide misconfiguration or bug in the Worker microservice, the error should occur on all Workers, meaning that the number of errors should be: `number of Workers * 60`.

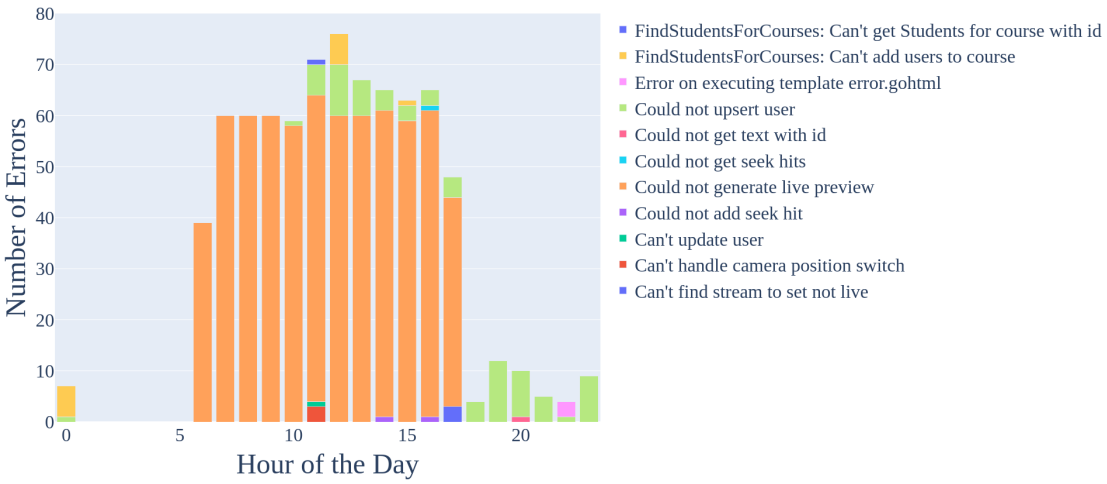

Figure 6.4: API Errors by Hour and Type

6.3 Performance Bottleneck: Worker

Again, using data collected from various logging services and exported via GoCast's Grafana dashboard, this section will give some insights on common errors of the Worker microservice and try to find possible causes as well as potential solutions to either avoid these errors or make it easier to debug them in the future.

6.3.1 Errors over Time

Similarly to the main API, the Worker logs too show a constant number of errors over time (see Figure 6.5), indicating that there are either bugs or network issues that continue to occur without being solved. With an average of approximately 30 Worker-related errors per day and days with as many as 75 logged errors (see Figure 6.6, it shows that the Worker system can become a bottleneck in the future. Especially considering that - in contrast to the main API - there is not just one Worker deployed, but rather a few dozen, meaning that the number of errors will increase proportionally to the number of Workers deployed.

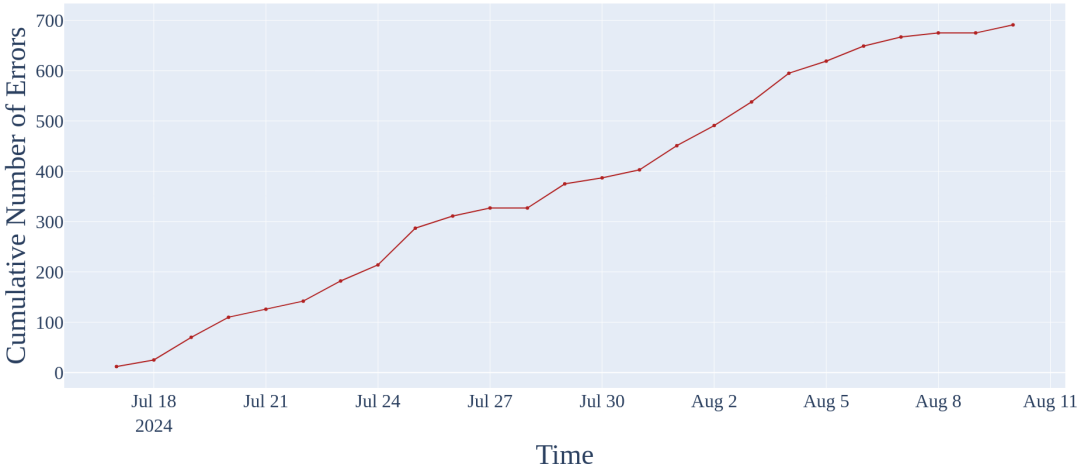

Figure 6.5: Cumulative Worker Errors over Time

6.3.2 The Main Cause Behind Worker Errors

To find out what the main issue behind the Worker errors is, it is again helpful to take a look at the most common error types (see Figure 6.7). This shows that more than 90% of the errors have the error log *"Sending Heartbeat failed"*. A Worker sends a *Heartbeat* HTTP request once per minute to the main API with its current status and workload so the API has a current overview of all available Workers and can decide which Workers to give tasks to. This error occurs, however, when a Worker tries to send such a request and the request fails because the API is unreachable. This can either occur because the API has become unavailable or because the Worker's VM has a network issue. As the main API (besides two notable outages that lead to the API being offline for a few

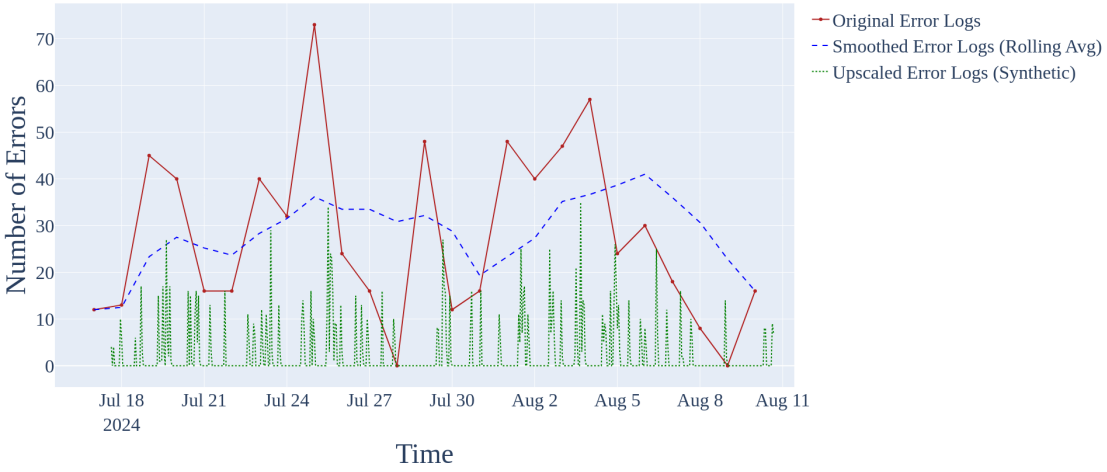

Figure 6.6: Upscaled Worker Error Logs

minutes) had an uptime of 99%, the error is most likely with the Worker system itself. When taking a closer look at the actual logs, especially at other more critical error types that occur less often, such as *"Error uploading stream"*, it becomes clear that most of these errors are always reported by the two same Workers VMs:

- `labels={container=live_worker..., error=Post "http://vodservice:8089": dial tcp 10.0.1.2:8089: i/o timeout, host=A, level=error, msg=Error uploading stream, stream=..., time=...}`
- `labels={container=live_worker..., error=Post "http://vodservice:8089": dial tcp 10.0.1.2:8089: i/o timeout, host=B, level=error, msg=Error uploading stream, stream=..., time=...}`

Note that for simplicity, some information has been redacted from the logs and the hosts have been renamed to `A` and `B`. However, the main point remains that it appears that the errors occur due to a timeout when trying to reach the VOD Service. As all Workers run on the same Worker-version and on similar VMs, and only the two Workers on host `A` and `B` report timeout error, it might be due to a misconfiguration or separate issue with the VM itself. A possible solution to improve debugging this error would be to include more details of the host VM or Worker name in the error log.

Figure 6.7: Worker Error Types

6.3.3 Future Improvement: Runners

At the time of this thesis, GoCast is developing a more robust and scalable alternative to the old Worker system: The Runner. Compared to the Worker system, which receives a task (e.g., to upload a lecture) and then performs it straightaway, the Runner system works based on queues and jobs. With this, incoming tasks are not just performed, resulting in either success or an error, but rather scheduled and handled internally in such a way that a task consists of multiple jobs that can be prioritized accordingly and, if failed, can be automatically retried multiple times before returning an error. Such a system would most likely solve errors such as the *"Error uploading stream"* error mentioned before, as the error would be handled internally and retried within a certain time interval before reporting an error and aborting the task.

6.4 A Technical Comparison of REST and gRPC APIs

While analyzing the performance and limitations of the individual microservices of GoCast can help to resolve bottlenecks, at the core of the entire system, there is the main GoCast API. Most services depend on the main API to fetch up-to-date information on streams and courses, to send updates on their current status and to notify it whenever there are new streams or VODs. Also, from the end-user's perspective, the API needs to handle many concurrent requests as efficiently as possible. Hence, in this section, we will focus on two different API design approaches: gRPC and REST.

	REST	**gRPC**
Data Format	Plain-text formats like JSON or XML, which are human-readable but less efficient.	Protocol Buffers (Protobuf), a more space-efficient binary format that is faster to (de)serialize but not human-readable.
Communication Patterns	Follows a unary request/response model (client sends one request, the server responds with one reply).	Supports multiple communication patterns: unary, server streaming, client streaming, and bidirectional streaming.
Underlying Protocol	Uses HTTP/1.1, which is common for web applications, but slower than HTTP/2.	Uses HTTP/2, offering advanced features like streaming and multiplexing, leading to reduced latency and higher throughput.
Code Generation	Lacks built-in code generation, requiring third-party tools for code generation.	Native code generation for both clients and servers in various languages via Protobuf.
Data Validation	Requires manual data validation for formats like JSON, increasing processing overhead.	Protobuf ensures strongly typed data "contracts", automatically validating messages.
Bidirectional Streaming	Not supported natively.	Allows bidirectional streaming, where both client and server can exchange messages over a single connection.
Use Case	Simple, public-facing APIs where ease of integration and human-readability (e.g., JSON) are prioritized.	High-performance, microservice-based architectures with low-latency needs, especially for real-time data and high-throughput applications.
Client-Server Coupling	Loosely coupled, making it easier for client and server to evolve independently.	Tightly coupled due to shared Protobuf files, requiring both client and server to have the same schema.
Design Approach	Resource-oriented: Use HTTP methods (`GET`, `POST`, `PUT`, `DELETE`, `OPTIONS`) to operate on resources.	Service-oriented: Invoke the gRPC methods on the server as if they were local functions.

Table 6.1: Comparison between REST and gRPC [Nal20; Dou22; Pos23; Ama24]

6.4.1 Why gRPC?

gRPC, released in 2015 by Google, is an open-source Remote Procedure Call (RPC) framework that can run cross-platform in any environment. Its strengths are that it can efficiently connect services and supports load balancing, tracing, health checking and authentication. Another particularity of gRPC is that it automatically generates idiomatic client and server bindings for a service in various languages and platforms using protofiles [Nal20]. As it will become clear in the next subsections, its main advantage over classical HTTP/REST APIs is that as is based on Protobuf, it has a noticeably stronger performance [Dou22] and supports HTTP/2-based transport and bi-directional streaming, which is difficult with HTTP/1.1 and more efficient than using WebSockets for such purposes [Fen19].

6.4.2 GoCast's gRPC Prototype and JMeter Test Setup

GoCast's current REST API is written in Go using the Gin-Gonic HTTP web framework. To test if a switch to gRPC would be useful, a prototype for a gRPC API V2 was developed and can be found in the enh/api_v2 branch of github.com/carlobortolan/thesis. The prototype is based on the official Go gRPC library and uses protoc used to compile and generate Go code from the `.proto` files as well as buf for managing and generating Protobuf files. Currently, only the most important original REST API endpoints related to the user, course and stream management are supported by the prototype, as it is mainly intended for test purposes.

JMeter Configuration and Data Collection

To perform the performance tests that resulted in the data used in the following section, Apache JMeter[6] - an open-source load tester and performance measurer was used. The main advantage of it is that it exclusively works at the protocol level, meaning that it appears and behaves like a browser (or rather, multiple browsers) to web services while not performing all the actions supported by browsers. The performed tests were structured using two thread groups - one for REST and one for gRPC - with each having three requests for the respective REST endpoints and gRPC methods:

- REST: `GET:/courses`, `POST:/user/settings`, `PATCH:stream/bookmarks`
- gRPC: `getPublicCourses`, `postSettings`, `putBookmarks`

[6] `https://jmeter.apache.org`

Each thread group used a thread pool (users) of size 150, a ramp-up period of 10 seconds and a repetition of 10,000 loops. To measure, compare and export the response time, JMeter-listeners like *Graph Results*, *Summary Report* and *Simple Data Writer* were used. To handle errors and check that the requests were performed successfully, JMeter's *View Results Tree* was used, which captured any failed requests or error codes.

6.4.3 Comparison of GoCast's gRPC Prototype and Current REST API

The tests had a duration of 20 to 30 minutes and performed 1,196,180 HTTP requests with a mean response time of 644.55 ms and 1,171,342 gRPC requests with a mean response time of 162.69 ms. A more detailed overview can be found in Table 6.2.

Table 6.2: Statistical Analysis of REST and gRPC Data

Metric	**REST API**	**gRPC API**
Count	1,196,180	1,171,342
Elapsed Time		
Mean	644.55 ms	162.69 ms
Standard deviation	60.38 ms	36.13 ms
Min	101 ms	101 ms
25th percentile	628 ms	148 ms
Median (50th percentile)	642 ms	152 ms
75th percentile	659 ms	175 ms
Max	4,562 ms	2,090 ms
grpThreads (allThreads for REST)		
Mean	149.90	149.21
Standard deviation	2.94	2.28
Min	2	11
25th percentile	150	149
Median (50th percentile)	150	149
75th percentile	150	150
Max	150	150

To evaluate whether gRPC provides significant enough benefits, the focus was on collecting and analyzing the following metrics:

- **Latency**: Response Time of an individual request - see Figure 6.8.
- **Throughput**: How many requests per minutes the API can handle - see Figure 6.9

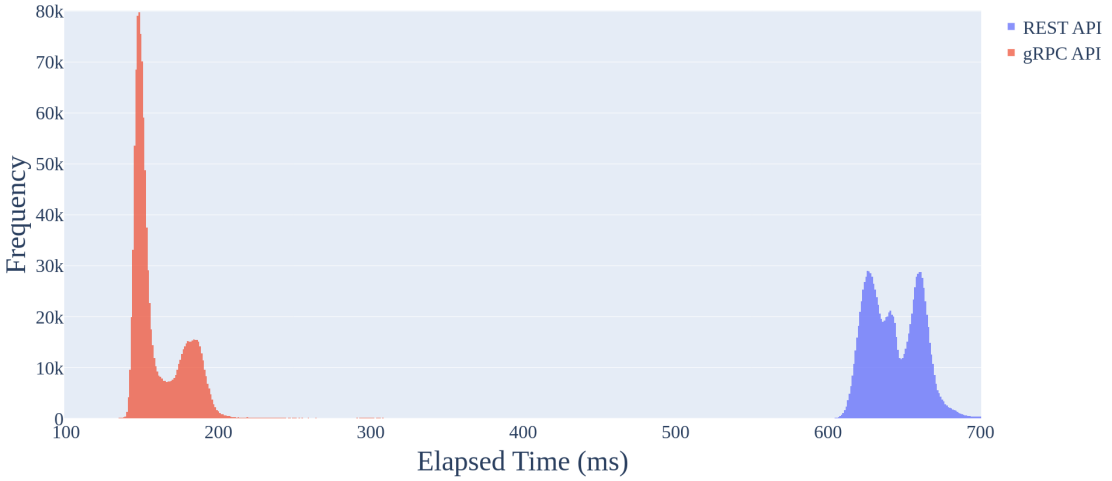

Figure 6.8: Response Time Distribution

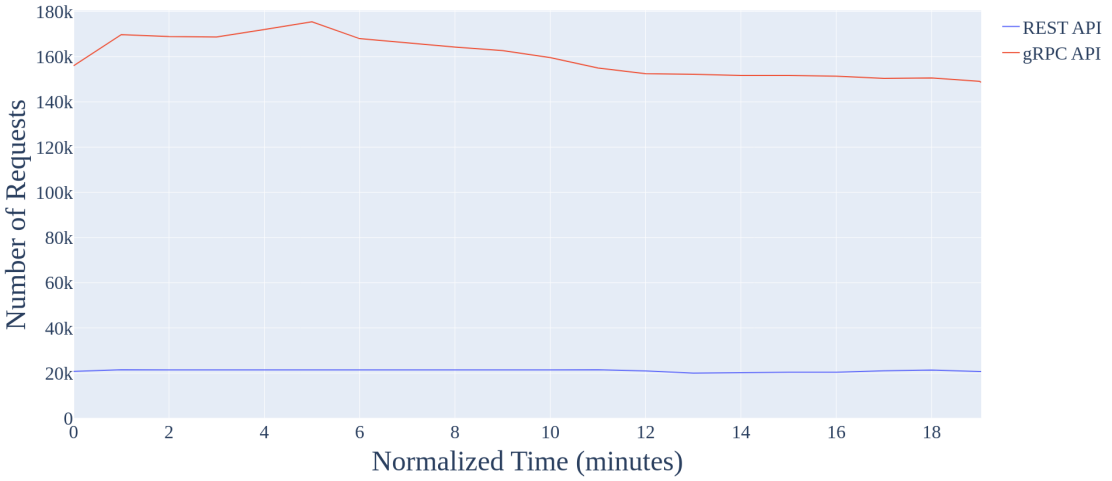

Figure 6.9: Throughput over Time

6.4.4 Results

As expected, gRPC is significantly faster due to its use of Protobuf and HTTP/2, which reduces overhead compared to HTTP/1.1. To evaluate whether gRPC justifies the effort of switching the API Design, there are three main considerations:

1. **Performance Benefits**: gRPC offers approximately a 3-times significant reduction in latency and an 8-times increase in throughput. It can also be seen that the standard deviation of the response time with gRPC requests (36.13 ms) is nearly half that of HTTP requests (60.38 ms) which could lead to an overall more stable API performance. When considering outliers, the gRPC API again outperforms the REST API, having a maximum latency of 2,090 ms for slow requests compared to 4,562 ms for the REST API.

2. **Browser Compatibility**: While the lack of native browser support for gRPC might make REST a better fit, the current gRPC prototype exposes an HTTP Gateway that can be accessed just like a REST API by browser-based clients. Also, the gRPC prototype only contained "normal" endpoints. Still, considering the many real-time use cases that come with streaming (e.g., the live chat), gRPC's bidirectional streaming could be a big advantage over REST.

3. **Migration Effort**: Migrating to gRPC would require a complete re-implementation from scratch of the GoCast API with significant changes to the clients and infrastructure. While tools for REST are widely spread (e.g., Postman, cURL), the gRPC ecosystem currently has fewer options for debugging, monitoring and testing, often requiring additional third-party add-ons to make commonly used tools compatible with gRPC.

6.5 Limitations of the Current System

Given the analyzed data and review of GoCast's codebase, some limitations have been found that could become major issues when scaling up the current system.

6.5.1 GoCast's In-Memory Chat System

A major limitation of GoCast is its chat system. Currently, it is implemented as a combination of TCP and WebSockets to transmit real-time updates to viewers but saves the actual chat channels in memory. This means that if the main API were to be scaled to multiple instances and two users were to connect to two separate instances, their chat messages would never reach the other instance. A possible solution for this problem

would be to refactor the chat system on the server side to store the chat messages in a separate in-memory database such as Redis[7] or use distributed messaging services such as NATS.io[8]. Alternatively, it is also possible to fix this issue by always having a certain GoCast instance being responsible for a certain chat or defining a hash function that determines to which instance messages of a certain stream should be posted.

6.5.2 N+1 Queries

Another weakness of the current system is the N+1 query problem, which is the result of multiple database queries executed to fetch data between Object-relational mapper (ORM) models. The N+1 query problem occurs when one initial query is followed by N additional queries for associated data, leading to performance degradation when N is large. In the current implementation, certain database operations, especially when retrieving courses and their associated streams or users, have this issue, as instead of creating one longer query that returns the given object and associations, first one query retrieves the object itself and then N queries retrieve the associated objects individually. This results in many queries being executed (N+1 instead of only 1) and negatively impacts performance, especially at scale.

Example of N+1 Problem in the Current Code:

In the `liveStreams` method, for each live stream retrieved by the `GetCurrentLive` query, additional queries are executed to fetch the associated course and lecture hall.

```
// returns all streams that are live
func (r streamRoutes) liveStreams(c *gin.Context) {
    streams, err := r.StreamsDao.GetCurrentLive(c)
    // ...
    for _, s := range streams {
        course, err := r.CoursesDao.GetCourseById(c, s.CourseID)
        lectureHall, err := r.LectureHallsDao.GetLectureHallByID(s.LectureHallID)
        // ...
```

1. The first query retrieves all currently live streams.
2. For each stream, a separate query fetches its associated course.
3. For each stream, an additional query fetches its associated lecture hall.

[7] `https://redis.io`
[8] `https://github.com/nats-io/nats-server`

This leads to N additional queries for the courses and lecture halls for each live stream, resulting in a total of 2N queries plus one initial query for fetching the streams. As we know from Subsection 3.2.3, TUM-Live can have up to 100 active streams with more than 1,000 viewers a day, resulting in a significant number of unnecessary queries.

Possible Solutions:

1. **Use Preloading to Eagerly Load Associations**: By preloading the associated data (courses and lecture halls) in a single query, the number of database queries can be reduced to a minimum. For example, the `GetCurrentLive` method could be refactored to preload the `Course` and `LectureHall` fields as shown below:

```
func (d *StreamsDao) GetCurrentLive(ctx context.Context) ([]model.Stream, error) {
    var streams []model.Stream
    err := DB.Preload("Course") // eagerly load stream's course
            .Preload("LectureHall") // eagerly load stream's lecture halls
            .Where("live_now␣=␣?", true)
            .Find(&streams).Error
    return streams, err
}
```

   Now, the `liveStreams` endpoint does not need to make additional queries for each stream:

```
func (r streamRoutes) liveStreams(c *gin.Context) {
    streams, err := r.StreamsDao.GetCurrentLive(c)
    // ...
    for _, s := range streams {
        lectureHall = s.LectureHall
        course = s.course
        // ...
    }
}
```

2. **Batch Queries**: Instead of loading each object one by one, the `GetCurrentLive` method could collect the IDs of the current live stream first and then batch the queries, reducing the number of database calls.

3. **Caching**: While not particularly fitting for this example, N+1 queries can be resolved by caching frequently accessed data (with careful invalidation) to reduce overall database queries.

4. **GraphQL or DataLoader**: A more advanced solution would be to use batching mechanisms like DataLoader[9], often used in GraphQL environments, to batch multiple requests for related data into a single query.

6.5.3 Other Factors for API Design

While the gRPC prototype showed promising results, when it comes to comparing the different API design approaches, there are additional factors that can be considered:

- **Resource Usage (CPU, Memory, Network)** It might be interesting to compare how efficiently each API handles resources under load. gRPC, with its smaller binary payloads, may consume less bandwidth but may use more CPU. Also, for potential bandwidth issues, network traffic could be analyzed to understand the differences in payload size and bandwidth usage.
- **Error Rates and Reliability** In a production-like environment, for testing purposes, it could be evaluated how both systems behave under load or failure scenarios. Does one API lead to more errors under heavy load, or can both handle the same level of traffic? One could measure the error rate (=percentage of failed requests during the test) or the time to recover from errors or failures (e.g., retries, handling errors, etc.).

[9] `https://github.com/graphql/dataloader`

7 Outlook and Alternative Technologies

In the previous sections, we looked at possible optimization approaches and limitations of TUM-Live. This chapter goes one step further and takes a brief look at current developments, trends and future technologies in the field of video streaming.

7.1 Machine Learning for Cloud Video Streaming Services

One promising application is predicting the transcoding time of video segments before scheduling, given an input video stream and its transcoding parameters, using the transcoding time as a random variable based on past observations. Simulation results show that such prediction methods can lead to significantly better load balancing of transcoding jobs than traditional methods [Den+14]. This would be especially useful for cloud environments, as accurate task execution time estimations can lower overall usage costs and resource provisioning. Other possible application fields of machine learning are for cost and storage optimization of deployed streaming systems, which could lead up to 15% to 20% compared to current methods [DAB23].

7.2 Blockchain Technology for Video Streaming

Blockchain is a technology that allows secure P2P communication through a shared database called a distributed ledger. In this system, every computer (node) in the network has a copy of the blockchain, which contains the full history of all transactions. Before a transaction can be added, all nodes must agree on it. Blockchain could potentially have a significant impact on the video streaming industry, as it could be used to create an allowlist of approved publishers or distributors to control user identities in a decentralized way. Currently, the video streaming industry is mainly controlled by quality standards and algorithms set by streaming platforms. For example, on YouTube, the search and recommendation systems are controlled by YouTube, not the creator. A Blockchain-supported system changes this by allowing publishers to share their content [LDB20].

7.3 Media Over QUIC

Media over QUIC is built on top of the QUIC protocol[1], focusing on live video delivery and taking advantage of the same benefits that QUIC offers over regular TCP. Like QUIC, Media over QUIC provides reduced connection setup times, encryption by default and multiplexing to avoid head-of-line blocking and congestion control, leading to significantly reduced latency compared to other standards [Bre24]. It uses one unified protocol for transmitting and receiving media, including audio, video and time-synchronized metadata like captions. Internally, Media over QUIC works by allowing *"media to flow through multiple relays, which can help fan out the data to many downstream users [...] [and] can be placed throughout the data transportation route, including at the CDN level, within a 5G network, and even on a user's local Wi-Fi. As media travels, copies of the data are stored in these relays"* [Bre24]. Therefore, lost media packets don't need to be re-requested and re-sent from and to the original server but only from and to the next relay (e.g., the user's local Wi-Fi).

Currently, Media over QUIC is still in the drafting stage, but there already are early available implementations and it can be expected that once it gets standardized and becomes more widely spread, many streaming services will incorporate it into their platform's infrastructure [DSF23].

7.4 Environmental Considerations

As explained in previous sections, video streaming services are resource-intensive, performing computing- and storage-intensive tasks that consume large amounts of computational power and storage. Given the environmental focus nowadays, it might require companies to consider their ecological footprint first. Advancements in video compression standards can reduce overall bandwidth usage but also shift environmental responsibilities from data centers and streaming providers to end users and hardware manufacturers, as more efficient video compression algorithms require additional power to encode and decode video, which increases the energy consumption of end-user devices [JK23].

[1] QUIC (Quick UDP Internet Connections), published in 2013, is a transport protocol designed to address the performance and security limitations of traditional protocols like TCP. It is built on top of UDP and combines features from protocols such as TLS and HTTP/2 to provide low-latency and secure data transmissions [IET24].

8 Conclusion

Video streaming infrastructure is a fascinating topic often taken for granted by users who open Netflix or YouTube and start streaming videos for hours without an idea of the complexity of what happens behind the screen when they press the "play" button. This thesis has not only shown how enterprise-grade streaming architecture is built and optimized to its limit, but also demonstrated the hands-on process of scaling TUM-Live. Besides scaling it to a distributed system that allows organizations to share computing resources and storage, microservices like the RTMP-Proxy now made features such as self-streaming more accessible. With this, TUM-Live will be able to be used not just by the CIT, but also by other TUM schools and universities. Additionally, we have found that there are still areas with room for improvement, including performance bottlenecks, complicated error management, N+1 queries and the issue of having an in-memory chat. Lastly, this thesis also provided concrete solutions and optimizations, such as the gRPC API that reduced latency by 3x and improved throughput by 8x compared to the current system.

Hopefully this thesis will serve as a useful resource for developers working on the GoCast architecture, researches exploring new streaming technologies, professionals looking for alternative scaling solutions and all individuals interested in understanding what really happens when they press "play".

Abbreviations

ABR Adaptive Bitrate Transcoding

ACID Atomicity, Consistency, Isolation, Durability

AES Advanced Encryption Standard

API Application Programming Interface

AWS Amazon Web Services

CDN Content Delivery Network

CIT School of Computation, Information and Technology

DRM Digital Rights Management

EKS Elastic Kubernetes Service

gRPC Google Remote Procedure Calls

HLS HTTP Live Streaming

ISPs Internet Service Providers

ITO IT Operations

JWT JSON Web Token

LP Linear Program

LRZ Leibniz Supercomputing Centre

MBone Multicast Backbone

OCAs Open Connect Appliances

ORM Object-relational mapper

P2P Peer-to-Peer

POPs Points of Presence

Protobuf Protocol Buffers

QA Quality Assurance

REST Representational State Transfer

RTMP Real-Time Messaging Protocol

SAML Security Assertion Markup Language

SMP Streaming Media Processor

SSO Single Sign On

TUM Technical University of Munich

TLS Transport Layer Security

VMP Virtual Media Processor

VM Virtual Machine

VOD Video on Demand

VCU Video Coding Unit

List of Figures

List of Tables

Bibliography

[Adh+12] V. Adhikari, Y. Guo, F. Hao, V. Hilt, and Z.-L. Zhang. "A tale of three CDNs: An active measurement study of Hulu and its CDNs." In: Mar. 2012. DOI: 10.1109/INFCOMW.2012.6193524.

[Ama24] Amazon. *What's the Difference Between gRPC and REST?* Accessed: 2024-10-06. 2024. URL: https://aws.amazon.com/compare/the-difference-between-grpc-and-rest/.

[Ben+18] A. Bentaleb, B. Taani, A. Begen, C. Timmerer, and R. Zimmermann. "A Survey on Bitrate Adaptation Schemes for Streaming Media over HTTP." In: *IEEE Communications Surveys & Tutorials* 21 (Aug. 2018), pp. 1–1. DOI: 10.1109/COMST.2018.2862938.

[BN15a] S. Bagui and L. Nguyen. "Database Sharding:: To Provide Fault Tolerance and Scalability of Big Data on the Cloud." In: *International Journal of Cloud Applications and Computing* 5 (Apr. 2015), pp. 36–52. DOI: 10.4018/IJCAC.2015040103.

[BN15b] S. Bagui and L. T. Nguyen. "Database Sharding: To Provide Fault Tolerance and Scalability of Big Data on the Cloud." In: *Int. J. Cloud Appl. Comput.* 5.2 (Apr. 2015), pp. 36–52. ISSN: 2156-1834. DOI: 10.4018/IJCAC.2015040103. URL: https://doi.org/10.4018/IJCAC.2015040103.

[Bre24] B. Brett. *What's the deal with Media Over QUIC?* Accessed: 2024-10-06. 2024. URL: https://www.ietf.org/blog/moq-overview.

[Cia10] J. Ciancutti. *Four Reasons We Choose Amazon's Cloud as Our Computing Platform*. Accessed: 2024-10-06. 2010. URL: https://netflixtechblog.com/four-reasons-we-choose-amazons-cloud-as-our-computing-platform-4aceb692afec.

[Cla17] A. Clark. *The MITRE Corporation: File and Directory Discovery*. Accessed: 2024-10-06. 2017. URL: https://attack.mitre.org/techniques/T1083.

[Cuo08] D. Cuong. *Powered By YouTube - Scaling up YouTube*. Accessed: 2024-10-06. 2008. URL: https://www.youtube.com/watch?v=HXevnuOOy48.

[Cuo12] D. Cuong. *Seattle Conference on Scalability: YouTube Scalability*. Accessed: 2024-10-06. 2012. URL: https://www.youtube.com/watch?v=w5WVu624fY8.

[DAB23] M. Darwich, T. Alghamdi, and M. Bayoumi. "Deep Learning Approach for Cost and Storage Optimization of Video Streaming in Cloud Environments." In: *2023 IEEE 8th International Conference on Smart Cloud (SmartCloud)*. 2023, pp. 80–85. DOI: 10.1109/SmartCloud58862.2023.00022.

[Dar17] M. Darwich. "Cost-Efficient Video On Demand (VOD) Streaming Using Cloud Services." PhD thesis. Dec. 2017.

[Den+14] T. Deneke, H. Haile, S. Lafond, and J. Lilius. "Video transcoding time prediction for proactive load balancing." In: *2014 IEEE International Conference on Multimedia and Expo (ICME)*. 2014, pp. 1–6. DOI: 10.1109/ICME.2014.6890256.

[Dou22] I. Douglas. *How to choose HTTP or gRPC for your next API*. Accessed: 2024-10-06. 2022. URL: https://blog.postman.com/how-to-choose-http-or-grpc-for-your-next-api.

[DSF23] M. Duke, Z. Sarker, and A. Frindell. *Media Over QUIC Project Overview*. Accessed: 2024-10-06. 2023. URL: https://datatracker.ietf.org/group/moq.

[ET22] S. Elakkiya and K. S. Thivya. "Comprehensive Review on Lossy and Lossless Compression Techniques." In: *Journal of The Institution of Engineers (India): Series B* 103.3 (June 2022), pp. 1003–1012. ISSN: 2250-2114. DOI: 10.1007/s40031-021-00686-3. URL: https://doi.org/10.1007/s40031-021-00686-3.

[Fen19] A. Fenster. *HTTP and gRPC Transcoding*. Accessed: 2024-10-06. 2019. URL: https://google.aip.dev/127.

[Flo16] K. Florance. *How Netflix Works With ISPs Around the Globe to Deliver a Great Viewing Experience*. Accessed: 2024-10-06. 2016. URL: https://about.netflix.com/en/news/how-netflix-works-with-isps-around-the-globe-to-deliver-a-great-viewing-experience.

[Gal17] D. Gallatin. *Serving 100 Gbps from an Open Connect Appliance*. Accessed: 2024-10-06. 2017. URL: https://netflixtechblog.com/serving-100-gbps-from-an-open-connect-appliance-cdb51dda3b99.

[Gro+13] D. Grois, D. Marpe, A. Mulayoff, B. Itzhaky, and O. Hadar. "Performance comparison of H.265/MPEG-HEVC, VP9, and H.264/MPEG-AVC encoders." In: Dec. 2013, pp. 394–397. ISBN: 978-1-4799-0294-1. DOI: 10.1109/PCS.2013.6737766.

[Hof08] T. Hoff. *YouTube Architecture*. Accessed: 2024-10-06. 2008. URL: https://highscalability.com/youtube-architecture.

[IET24] IETF. *IETF QUIC Working Group*. Accessed: 2024-10-06. 2024. URL: https://quicwg.org.

[IVM16] Y. Izrailevsky, S. Vlaovic, and R. Meshenberg. *Completing the Netflix Cloud Migration*. Accessed: 2024-10-06. 2016. URL: https://about.netflix.com/en/news/completing-the-netflix-cloud-migration.

[JK23] M. Jancovic and J. Keilbach. "Streaming against the Environment: Digital Infrastructures, Video Compression, and the Environmental Footprint of Video Streaming." In: *Situating Data: Inquiries in Algorithmic Culture*. Amsterdam University Press, 2023, pp. 85–102. ISBN: 9789463722971. URL: http://www.jstor.org/stable/jj.362382.9 (visited on 10/03/2024).

[JW11] H. Jin and Y. Wu. "Analysis and design of scalable DRM in VOD." In: *International Journal of Digital Content Technology and Its Applications* 5.8 (2011), pp. 407–414. DOI: 10.4156/jdcta.vol5.issue8.47.

[Kho21] S. Khorenyan. *NGINX-based Media Streaming Server*. Accessed: 2024-10-06. 2021. URL: https://github.com/arut/nginx-rtmp-module/wiki/Directives.

[LDB20] X. Li, M. Darwich, and M. Bayoumi. "A Survey on Cloud-Based Video Streaming Services." In: Elsevier, 2020. Chap. 1.

[Nai17] M. Nair. *How Netflix works: the (hugely simplified) complex stuff that happens every time you hit Play*. Accessed: 2024-10-06. 2017. URL: https://medium.com/refraction-tech-everything/how-netflix-works-the-hugely-simplified-complex-stuff-that-happens-every-time-you-hit-play-3a40c9be254b.

[Nal20] M. Nally. *gRPC vs REST: Understanding gRPC, OpenAPI and REST and when to use them in API design*. Accessed: 2024-10-06. 2020. URL: https://cloud.google.com/blog/products/api-management/understanding-grpc-openapi-and-rest-and-when-to-use-them.

[PA16] A. Pavlo and M. Aslett. "What's Really New with NewSQL?" In: *SIGMOD Rec.* 45.2 (Sept. 2016), pp. 45–55. ISSN: 0163-5808. DOI: 10.1145/3003665.3003674. URL: https://doi.org/10.1145/3003665.3003674.

[Pos23] Postman. *gRPC vs. REST*. Accessed: 2024-10-06. 2023. URL: https://blog.postman.com/grpc-vs-rest.

[Pro17] A. Prout. *Scaling Distributed Joins*. Accessed: 2024-10-06. 2017. URL: https://www.singlestore.com/blog/scaling-distributed-joins.

[Ran+21] P. Ranganathan, D. Stodolsky, J. Calow, J. Dorfman, M. Guevara, C. W. Smullen IV, A. Kuusela, R. Balasubramanian, S. Bhatia, P. Chauhan, A. Cheung, I. S. Chong, N. Dasharathi, J. Feng, B. Fosco, S. Foss, B. Gelb, S. J. Gwin, Y. Hase, D.-k. He, C. R. Ho, R. W. Huffman Jr., E. Indupalli, I. Jayaram, P. Kongetira, C. M. Kyaw, A. Laursen, Y. Li, F. Lou, K. A. Lucke, J. Maaninen, R. Macias, M. Mahony, D. A. Munday, S. Muroor, N. Penukonda, E. Perkins-Argueta, D. Persaud, A. Ramirez, V.-M. Rautio, Y. Ripley, A. Salek, S. Sekar, S. N. Sokolov, R. Springer, D. Stark, M. Tan, M. S. Wachsler, A. C. Walton, D. A. Wickeraad, A. Wijaya, and H. K. Wu. "Warehouse-scale video acceleration: co-design and deployment in the wild." In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 600–615. ISBN: 9781450383172. DOI: 10.1145/3445814.3446723. URL: https://doi.org/10.1145/3445814.3446723.

[Sai+24] T. Saini, P. Sharma, J. Tanwar, H. Alsghier, S. Bhushan, H. Alhumyani, V. Sharma, and A. Alutaibi. "Cloud-based video streaming services: Trends, challenges, and opportunities." In: *CAAI Transactions on Intelligence Technology* 9 (Mar. 2024). DOI: 10.1049/cit2.12299.

[She15] E. Shear. *Twitch Engineering: An Introduction and Overview*. Accessed: 2024-10-06. 2015. URL: https://blog.twitch.tv/en/2015/12/18/twitch-engineering-an-introduction-and-overview-a23917b71a25.

[Sil21] S. Silver. *Reimagining video infrastructure to empower YouTube*. Accessed: 2024-10-06. 2021. URL: https://blog.youtube/inside-youtube/new-era-video-infrastructure.

[Sta22] Statista. *Hours of video uploaded to YouTube every minute*. Accessed: 2024-10-06. 2022. URL: https://www.statista.com/statistics/259477/hours-of-video-uploaded-to-youtube-every-minute.

[Sun+08] D. Sun, S. Wu, J. Li, and A. Tung. "Skyline-join in distributed databases." In: May 2008, pp. 176–181. ISBN: 978-1-4244-2161-9. DOI: 10.1109/ICDEW.2008.4498313.

[Swa16] B. Swartz. *Improving Chat Rendering Performance*. Accessed: 2024-10-06. 2016. URL: https://blog.twitch.tv/en/2016/08/08/improving-chat-rendering-performance-1c0945b82764.

[VCS03] A. Vetro, C. Christopoulos, and H. Sun. "Video transcoding architectures and techniques: An overview." In: *Signal Processing Magazine, IEEE* 20 (Apr. 2003), pp. 18–29. DOI: 10.1109/MSP.2003.1184336.

[Web17] B. Weber. *Productizing Data Science at Twitch*. Accessed: 2024-10-06. 2017. URL: https://blog.twitch.tv/en/2017/06/01/productizing-data-science-at-twitch-67a643fd8c44.

[Wil16] K. Wilms. *We'll do it live — a new chapter in YouTube's live stream*. Accessed: 2024-10-06. 2016. URL: https://blog.youtube/news-and-events/well-do-it-livea-new-chapter-in.

[ZCL07] Y. Zhou, D. M. Chiu, and J. C. Lui. "A Simple Model for Analyzing P2P Streaming Protocols." In: Nov. 2007, pp. 226–235. ISBN: 978-1-4244-1588-5. DOI: 10.1109/ICNP.2007.4375853.