



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék

Videó streaming szolgáltatások implementációja

DIPLOMATERV

Készítette

Piller Trisztán

Konzulens

Kövesdán Gábor

2025. május 15.

Tartalomjegyzék

Kivonat	i
Abstract	ii
1. Bevezetés	1
1.1. Az első lépések	1
2. Elméleti háttér és irodalomkutatás	3
2.1. A videó formátumai	3
2.1.1. Konténerformátumok	3
2.1.2. Mozgóképek kódolási módszerei	4
2.1.3. Hang kódolási módszerei	4
2.2. Videó streaming kiszolgálása	4
2.2.1. Az élő közvetítés és Video-on-Demand különbségei	5
2.2.2. Adaptive Bitrate Streaming	5
2.3. Videó streaming hálózati protokolljai	7
2.3.1. Real-Time Messaging Protocol	7
2.3.2. HTTP Live Streaming	8
2.3.3. Dynamic Adaptive Streaming over HTTP	9
2.3.4. WebRTC	9
3. Követelmények	10
3.1. Funkcionális követelmények	10
3.2. Nem funkcionális követelmények	11
4. Felhasznált technológiák	13
4.1. FFMpeg szoftvercsomag	13
4.2. Open Broadcaster Software	13
4.3. Amazon Web Services	14
4.3.1. AWS Elemental és Amazon IVS	14
4.3.2. AWS-szolgáltatások hálózatiépítésre	17
4.3.3. Amazon S3	17
4.3.4. Amazon CloudFront	17
4.3.5. AWS-szolgáltatások számítási kapacitásra	18

4.3.6.	Kiegészítő AWS-szolgáltatások	18
4.4.	A webes komponensek technológiái	19
4.4.1.	TypeScript és JavaScript nyelvek	19
4.4.2.	React	20
4.5.	Üzemeltetési technológiák	21
4.5.1.	Docker	21
4.5.2.	GitHub	21
4.5.3.	Terraform	22
5.	A tervezett architektúra	23
5.1.	Logikai felépítés	23
5.1.1.	Összehasonlítás egy hasonló rendszerrel	24
5.2.	Fizikai felépítés AWS-re specializáltan	25
5.2.1.	Video-on-Demand kiszolgálás folyamata	27
5.2.2.	Live streaming folyamata	29
5.3.	Konfigurációmenedzsment	30
5.4.	A projekt felépítése	31
6.	Kliensközeli komponensek implementációja	32
6.1.	A CDN és a hozzácsatolt erőforrások	32
6.2.	A statikus weboldal	36
6.2.1.	A weboldal telepítésének CI/CD-folyamata	37
7.	Rétegeken átívelő szolgáltatások implementációja	40
7.1.	Single Sign-On (SSO) integrációja	40
8.	Szerveroldali folyamatok implementációi	44
8.1.	A virtuális privát felhő komponensei	44
8.2.	A Node.js-alkalmazás fejlesztése	47
8.2.1.	Az adatbázisséma	47
8.2.2.	Környezeti változók	48
8.2.3.	A videófeltöltés üzleti logikája	49
8.3.	A konténerizált környezet	51
8.3.1.	A szerveralkalmazás CI/CD-folyamatai	54
8.4.	Elemental MediaConvert felhasználása	55
8.4.1.	A MediaConvert-jobot indító Lambda-függvény	56
8.4.2.	A MediaConvert-job státuszváltozásának kezelése	59
8.5.	A MediaLive és MediaPackage összekötése	60
9.	Elkészült funkciók tesztelése	61
10.	Összegzés	63
10.1.	Továbbfejlesztés lehetőségei	63

HALLGATÓI NYILATKOZAT

Alulírott *Piller Trisztán*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2025. május 15.

Piller Trisztán
hallgató

Kivonat

Az IT-szakma meghatározó kihívása a magasan teljesítőképes, könnyen skálázódó és stabil infrastruktúra létesítése különféle üzleti célok megvalósítására. A hírközlés, a média és a szórakoztatás iparágaiban is kiemelt figyelmet kap ez a kihívás.

Ezek a virágzó és feltörekvő iparágak folyamatosan igénylik az olyan szoftverfejlesztőket, akik ezekre specializáltan is folyamatosan képzik magukat szakmailag, valamint mélyen ismerik a szakterület technológiáit.

Diplomatervem célja demonstrálni egy az Amazon Web Services platformján futó felhőalapú médiaszolgáltatás-rendszer alapos tervezésének, implementálásának, valamint üzemeltetésének folyamatát. A rendszer a videó streaming szolgáltatások területén nyújt weben elérhető megoldást, és a felhasználók számára lehetővé teszi, hogy saját időbeosztásuknak megfelelően férjenek hozzá videótartalmakhoz (Video-on-Demand), valami élő adásokat is tudjanak megtekinteni (élő közvetítés).

A megoldás ismertetése során kiemelt fókuszot kap a komponensek közötti laza kapcsolás kialakítása, az IT-biztonsági kockázatok kezelése, az konfigurációmenedzsment fenntarthatósága. Felhasználásra kerülnek modern webes technológiák és DevOps-technikák, mint a konténerizáció, a serverless függvények, a CI/CD-csővezetékek és az Infrastructure as Code.

Abstract

A key challenge for the IT profession is to build highly performant, easily scalable and stable infrastructure to support a variety of business goals. This challenge is also a major focus in Telecommunications, also in the Media & Entertainment industry.

These booming and emerging industries are in constant need of software developers who train themselves continuously for these industries and have a deep knowledge of the technologies in the field.

My thesis project aims to demonstrate the process of thoroughly designing, implementing and operating a cloud-based media delivery system running on the Amazon Web Services platform. The system will provide a web-based solution for video streaming services, allowing users to access Video-on-Demand content and live streams.

The solution will focus on the design of loose coupling between components, the management of IT security risks and the sustainability of configuration management. Modern web technologies and DevOps techniques such as containerisation, serverless functions, CI/CD pipelines and Infrastructure as Code will be used.

1. fejezet

Bevezetés

A média és szórakoztatás egy óriási szeletét tölti ki az internet teljes forgalmának. A videó streaming szolgáltatók, mint például a Netflix, a Twitch, vagy a YouTube, több százmilliós – vagy akár milliárdos – aktív felhasználói bázissal rendelkeznek, az globálisan kiterjedt és folyamatos forgalom kiszolgálására a világ legnagyobb szerverparkjait üzemeltetik. Kiterjed a felhasználásuk az élet minden területére: az oktatásra, a szórakozásra, a mindennapi és munkahelyi kommunikációra, a hírek és információk terjesztésére, a kulturális és művészeti élmények megosztására, össze tud kötni embereket a világ minden tájáról.

Végponttól végpontig tartó média streaming szolgáltatásoknak a fejlesztése és üzemeltetése számos komplex, informatikai területeket áthidaló kihívásokat hordoz magában. Ki kell tudni alakítani egy fenntartható, globális kiszolgálásra optimalizált és biztonságos hálózati infrastruktúrát. Folyamatosan kell tervezni a skálázhatóság biztosításával növekvő felhasználói bázissal. Ki kell tudni használni a legfrissebb hálózati protokollok adta lehetőségeket. Mélni kell a metrikákat a kitűnő felhasználói élmény biztosítása érdekében. Ki kell tudni szolgálni termékefélé végfelhasználói hardvert – például mobiltelefonok, különféle böngészők, AR- és VR-eszközök. A szabványok területén is tájékozottnak kell lennie a mérnököknek.

Már csak a kísérletezéssel felszedhető ismeretek is a piacon óriási előnyt jelentenek az ilyen irányba elköteleződő szakembereknek. Ezen indokok nyomán választottam magam is ezt a témát további vizsgálatra, eredményeim osztom meg jelen diplomamunkában.

1.1. Az első lépések

A téma bejárásának megkezdéseképp az Önálló laboratórium 2 című tárgy keretében egy olyan full-stack webes rendszer tervezését és implementációját vállaltam, amely lokálisan is futtatható szabad szoftvereket alkalmaz egy média streaming szolgáltatás alapjainak lefektetésére. Ennek köszönhetően megismerkedtem az FFMpeg szoftvercsomag videókonvertálási lehetőségeivel, videók kliensoldali lejátszásával HLS-protokollon továbbítva, valamint az NGINX webszerver RTMP-moduljának beállításával, amely lehetővé teszi a videók élő közvetítését.

Az elkészült rendszerből a diplomatervezés során alakítottam ki egy natív AWS-felhőalapú szoftverrendszert, némely komponens újrafelhasználásával előzőből – mint például a React-alapú kliensoldali weboldal, a Docker-konfiguráció, illetve a szerveroldali kód CRUD-funkciói.

2. fejezet

Elméleti háttér és irodalomkutatás

Ebben a fejezetben kerül bemutatásra minden jelentősebb alapfogalom, valamint az azokhoz szorosan kapcsolódó technikák és folyamatok, amelyek ismerete nélkülözhetetlen a videó streaming megértéséhez, illetve annak szoftveres megvalósításához.

2.1. A videó formátumai

A videó egy multimédiás eszköz auditív és mozgó vizuális információ tárolására, visszajátzására. Fontos felhasználási területei a bevezetésben is ismertetett média- és a szórakoztató ipar.

Egy videó tartalmazhat különböző nyelvű hanganyagokat, mozgóképet, és egyéb metaadatokat – például feliratokat és miniatűr állóképeket – mind egy fájlban. Ezek közös tárolására konténerformátumokat alkalmazunk, amelyek megadják, az egyes adatfolyamok hogyan, milyen paraméterekkel, kódolással, tömörítéssel kerüljenek tárolásra, és hogyan kerülhetnek majd lejátszásra.

Szokásos összekeverni, de a konténerformátumoktól függetlenül a mozgóképkódolás és a hangkódolás különálló folyamatok. A kódolás egy algoritmus nyomán az adatot tömöríti, hogy a tárolás és a továbbítás hatékonyabb legyen.

2.1.1. Konténerformátumok

A legelterjedtebb és legszélesebb körben támogatott konténerformátum a Moving Picture Experts Group (MPEG) gondozásában specifikált MP4 – avagy a sztenderdben használt nevén: MPEG-4 Part 14 –, amely az MPEG-4 projekt részeként született 2001-ben.

Ugyancsak az MPEG gondozásában, az MPEG-2 projekt részeként született 1995-ben az MPEG Transport Stream (MPEG-TS) konténerformátum, amely elsősorban a digitális televíziózásban használatos, és így az internetes videó streaming során is. Felbontja a videóadatokat kisebb, fix hosszú adatcsomagokra, ezzel is előkészítve a tulajdonképpeni kis késleltetésű azonnali továbbítására a videóanyagnak a hálózaton keresztül.

További elterjedt konténerformátumok közé tartozik a Matroska Video (MKV), amely a hibátűréséről ismert; Apple vállalat által macOS-re és iOS-re optimalizálva fejlesztett QuickTime Movie (MOV) formátuma; valamint a WebM, egy szabad felhasználású we-

bes kiszolgálásra optimalizálódott formátum, amelyet nagyobb jelentőségű webböngészők mind támogatnak. Régebbi, már kevésbé használt vagy kivezetett formátumok közé tartozik az Audio Video Interleave (AVI) és a Flash Video (FLV).

2.1.2. Mozgókép kódolási módszerei

Az MPEG-4 projekt keretében született az Advanced Video Coding (AVC) – avagy H.264 – kódolási szabvány mozgókép kódolására, amely 2004-ban vált elérhetővé. Továbbra is ez az egyik legelterjedtebb szabvány, a videó streamingben használt konténerformátumok is ezt a kódolást alkalmazzák.

Természetesen azóta több új szabvány is megjelent hasonlóan az MPEG projektjei alatt, mint például az 2013-ban megjelent High Efficiency Video Coding (HEVC) – avagy H.265 –, amely az AVC-től jobb tömörítést és jobb minőséget ígér, de a licencdíjak miatt nem vált annyira elterjedtté, mint az elődje.

Ezen modern problémák kiküszöbölésére terjedt el a VP8 és a VP9 – a WebM formátumnak tagjaként –, illetve az AV1 szabad felhasználású kódolási szabványok.

A szabványok konkrét megvalósításával (kodekek) nem foglalkozunk részletesebben, de egy szabad felhasználású és nyílt forráskódú megvalósítása a H.264-nek a x264, amelyet például az FFmpeg szoftvercsomag is használ videó kódolására és dekódolására.

2.1.3. Hang kódolási módszerei

Hanganyag kódolására is több szabványt tudunk megvizsgálni. Ilyen az MPEG Audio Layer III – rövid nevén: MP3 – 1991-ből, ezt 1997-ben az Advanced Audio Coding (AAC) szabvány váltotta le. Ezen szabványok az MPEG által kerültek kifejlesztésre, valamint mindkettő veszteséges tömörítést alkalmaz, azaz a kódoláson és dekódoláson átesett hanganyag minősége nem lesz azonos az eredeti hanganyaggal.

Főleg a mozivilágban használt kódolás a Dolby AC-3 – ismertebb nevén: Dolby Digital. Azonos bitrátánál is az előbbieknél jobb minőséget ígér. 2017-ben már lejárt a szabadalmi védelme, így azóta szabadon felhasználható. Egy másik, születése (2012) óta szabad felhasználású kódolás az Opus, amelyet a Skype és a Discord is használ VoIP-alapú kommunikációra, és az összes eddig említett kódolásnál jobb minőséget produkál.

Természetesen találkozhattunk tömörítetlen (pl.: WAV), illetve veszteségmentes kódolásokkal is (pl.: FLAC) is, azonban a streaming világában ezek nem használatosak, mivel a nagyobb fájlméretük meghaladná a sáv szélesség és a tárhely korlátait. Az AC-3 és AAC szabványok közül szokás választani a videó streaming során, széles körben kompatibilisek mindenféle lejátszó eszközzel, míg az Opus még nem eléggé támogatott.

2.2. Videó streaming kiszolgálása

A média streaming egy olyan folyamat, amely során az médiaadatokat – mi esetünkben videóadatot – egy adott hálózati protokoll felett, egy adott konténerformátumban, adott kódolással továbbítjuk a végfelhasználók számára. Elsősorban az azonnali elérhetőségre

összpontosít, azaz a lejátszásnak a lehető legkisebb késleltetéssel kell megtörténnie, kevésbé fontos a streaming során a minőség megtartása, mint ahogy az fontos lenne teljes médiumok egyben való letöltésekor.

A streaming során az adatfolyamba helyezés előtt a videóadatokat újrakódoljuk, majd kisebb adatcsomagokra – úgynevezett „packetekre” – bontjuk, és ezeket a csomagokat a hálózaton keresztül továbbítjuk a végfelhasználók felé. Az adatcsomagok önmagukban is értelmezhetőek, és a végfelhasználók lejátszó alkalmazásai képesek az adatcsomagokat a megfelelő sorrendben és időzítéssel lejátszani. A streaming könnyen reagál a lejátszás során ugrálásokra, előre- és visszatekerésre, mivel a videót nem kell teljes egészében letölteni a végfelhasználói eszközre, hanem a feldarabolt videócsomagot a lejátszás során továbbítjuk abban a pillanatban, amikor arra szükség lesz.

2.2.1. Az élő közvetítés és Video-on-Demand különbségei

Annak megfelelően, hogy az adat egésze mikor áll rendelkezésünkre, kettő fő streaming típust különböztetünk meg: az élő közvetítést (live streaming) és a Video-on-Demand (VOD) streaminget. A VOD esetében a videóadatokat előre rögzített formában tároljuk, és a végfelhasználók a videóadatokat a saját időbeosztásuknak megfelelően nézhetik meg. Az élő közvetítés esetében a videóadatokat valós időben továbbítjuk a végfelhasználók felé, és a végfelhasználók a videóadatokat a közvetítés során nézhetik meg.

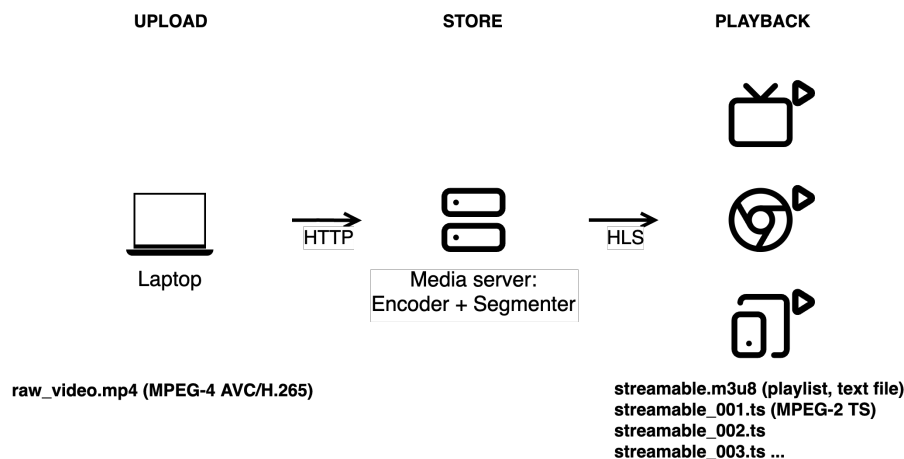
Különbözik mindkettőnél, hogy milyen fizikai infrastrukturális tervezést kell végrehajtani a legjobb kiszolgálás érdekében. Élő adásoknál a késleltetés a középponti kihívás, mivel a videóadatokat a lehető leggyorsabban kell továbbítani a végfelhasználók felé, hogy a közvetítés valóban élőnek tűnjön, ehhez magas számítási teljesítmény szükséges. A VOD esetében a hálózati sávszélességből adódó problémák leküzdése a központi kihívás, mivel a videót a felhasználók sokkal nagyobb közönsége kívánja elérni, azt kiváló minőségben szeretné megtekinteni, ennek ellenére a globális sávszélesség korlátozott. Itt a caching és a tartalomterjesztés optimalizálása a kulcs, ekkor jönnek képbe a Content Delivery Networkök (CDN-ek), azaz a tartalomterjesztő hálózatok.[9]

Üzleti szempontból a bevétel az élő adások során a közbeiktatott reklámokból származik főleg és a pay-per-view rendszerekből, míg a VOD esetében a közbeiktatott reklámokon kívül a felhasználók előfizetési díjából, tranzakcionális egyszeri vásárlásból – amennyiben a reklámokat kerülni szeretnék.

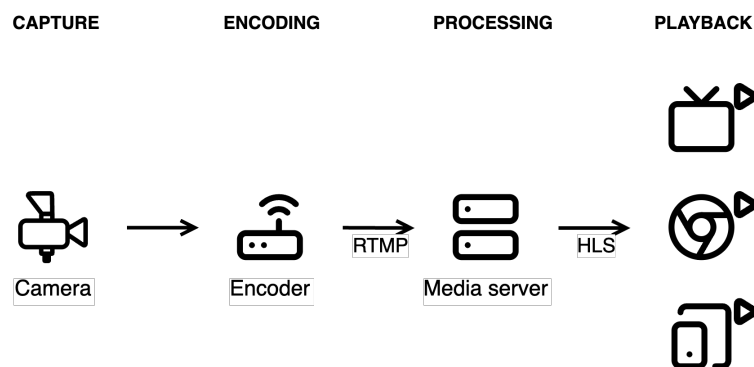
Lentebb két ábrát (2.1. ábra és 2.2. ábra) láthatunk, amelyek absztrakt példákat mutatnak egy VOD-kiszolgálás és egy élő közvetítés résztvevő médiaeszközeire. A rajta feltüntetett protokollok és fájlnevek a későbbi alfejezetekben olvasása során érthetőek lesznek.

2.2.2. Adaptive Bitrate Streaming

A streamelést erősen befolyásoló tényező a hálózati feltételek változása, amelyek a videólejátszás minőségét és késleltetését is befolyásolják. Az Adaptive Bitrate Streaming (ABR)



2.1. ábra. Példa egy VOD-kiszolgálás résztvevő eszközeire.

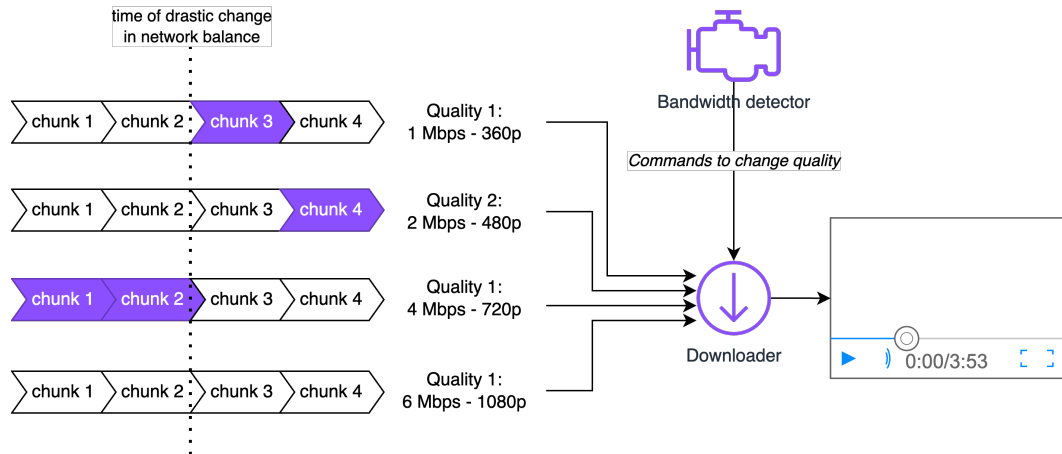


2.2. ábra. Példa élő közvetítés résztvevő eszközeire.

egy a hálózati kiszolgálás során alkalmazott technika, amely megoldást jelenthet erre a problémára.

Az ABR során a forrásvideót több különböző bitrátával dolgozó kodekekkel és különböző felbontással kódoljuk a médiaszerveren, majd lejátszáskor a lejátszó alkalmazás a hálózati feltételek változására reagálva, valamint a fogadó fél számítási kapacitásától függően valós időben választja ki a megfelelő videóstreamet ezek közül, onnan válogatja a packeteket (2.3. ábra).

Tiszta, hogy az ABR-t megvalósító rendszer előnyös mind a VOD, mind az élő közvetítés esetében, mivel a hálózati feltételek változása mindkét esetben előfordulhat, az automatikus streamváltogatás beavatkozás nélkül sokkal nagyobb Quality of Experience (röviden QoE, magyarul *az élmény minősége*) lehetőségét tudja biztosítani[7]. Az ABR-t alkalmazó rendszerek a videóadatokat több különböző bitrátával kódolják, ez természetesen a feldolgozási idejét a rendszereknek megnöveli, sőt élő közvetítés során egységnyi idő alatt jóval több számítási terhelést kell kibírnia a rendszernek.



2.3. ábra. Az Adaptive Bitrate Streaming működése.

2.3. Videó streaming hálózati protokolljai

Videó streamelésére – másképp fogalmazva: valamely korábban ismertetett formátumban tárolt videófájl adatfolyamba való illesztésére – különböző hálózati protokollok palettája áll rendelkezésünkre.

Néhány ismertebb célprotokoll létrejöttük időrendjében[23]:

- Real-Time Messaging Protocol (RTMP, 1996)
- Microsoft Smooth Streaming (MSS, 2008)
- Adobe's HTTP Dynamic Streaming (HDS, 2009)
- HTTP Live Streaming (HLS, 2009)
- WebRTC (2011)
- Dynamic Adaptive Streaming over HTTP (DASH, 2012)

Jelen alfejezet ezen alkalmazásrétegben alkalmazott protokollok (Layer 7) közül vizsgálja meg a legelterjedtebbeket, kitérve arra is, milyen szállítási rétegű protokollokkal (Layer 4) tudnak együttműködni.

2.3.1. Real-Time Messaging Protocol

A Real-Time Messaging Protocolt (RTMP) nevű protokollt még a Macromedia nevű cég fejlesztette ki, amely vállalatot később az Adobe felvásárolta. Az RTMP egy teljes szerver-től kliensig tartó protokoll, amely a Flash Player és a Flash Media Server közötti kommunikációra lett kifejlesztve eredendően. Közvetlen dolgozik a TCP felett, ennek nincs köze a HTTP-hez.[23] Az RTMP-t még támogatja sok-sok platform, mert egész alacsony késleltetést lehet vele elérni, egészen alacsony „költségű”, a Twitch és a YouTube is támogatja, hogy RTMP pull üzemmódjában tudjunk ezeken a platformokon élő adást fogadni, tehát szerveroldalon még használt, persze a Flash Player támogatásának 2020-as kivezetése miatt a protokoll használata kliensoldalon pedig visszaszorult.

2.3.2. HTTP Live Streaming

A HTTP-alapú streaming protokollok közül a legelterjedtebb a HTTP Live Streaming (HLS), amelyet az Apple fejlesztett ki 2009-ben. A HLS eredetileg az Apple jogvédett kereskedelmi protokolljaként indult, azóta viszont már szabad felhasználásúvá vált. Az Apple eszközök – macOS, iOS – alapértelmezetten támogatják ezt a protokollt, és a modern böngészők is támogatják a Media Source Extensions (MSE) API-n keresztül. Az HLS a HTTP felett dolgozik, egymástól független packetekre szedi a teljes kiszolgálandó videót, és az RTMP-hez képest ez így állapotmentes adatforgalmazást tud megvalósítani.[23]

HLS használata során H.264 formátumban kell a videóadatokat kódolni, a hangot AAC, MP3 vagy Dolby szabványokkal lehet kódolni. A konténerformátumot tekintve is kötött, MPEG-2 Transport Stream (MPEG-TS) formátumot használhatjuk, vagy pedig az MP4-et – fMP4 technikával, azaz *fragmented MP4-gyel*, ehhez pedig a Common Media Application Format (CMAF) konténerformátumra kell átalakítani.[21] A darabokra szedést követően a packeteket egy lejátszási listában (.m3u8 kiterjesztésű szöveges indexfájl) tartja számon a szerver, amelyet a lejátszó alkalmazások letöltenek, és a lejátszás során a megfelelő sorrendben és időzítéssel lejátszák.[5] Lásd a példát egy 720p-s streamet leíró indexfájltra a 2.1. kódrészletben.

A HLS egy olyan protokoll, amely magában hordozza az ABR implementációját is, kliensoldalon modern Media Source Extensions funkcionalitást támogató böngészőkben elterjedt használni a *hls.js*¹ nevű JavaScript-könyvtárat, amely implementálja a HLS protokollt. A .m3u8 indexfájl definiálhat több másik ilyen indexfájlt is, amelyek a különböző bitrátájú videóstreameket képviselik (pl. egy *1080p.m3u8* fájl és egy *720p.m3u8* fájl írja le a playlistjét egy-egy bitrátájú streamnek).

```
1 #EXTM3U
2 #EXT-X-VERSION:3
3 #EXT-X-TARGETDURATION:4
4 #EXT-X-MEDIA-SEQUENCE:1
5 #EXTINF:4.000000,
6 skate_phantom_flex_4k_2112_720p1.ts
7 #EXTINF:4.000000,
8 skate_phantom_flex_4k_2112_720p2.ts
9 #EXTINF:4.000000,
10 skate_phantom_flex_4k_2112_720p3.ts
11 #EXTINF:4.000000,
12 skate_phantom_flex_4k_2112_720p4.ts
13 #EXTINF:4.000000,
14 skate_phantom_flex_4k_2112_720p5.ts
```

2.1. kódrészlet. Részlet egy .m3u8 indexfájlból.

¹<https://github.com/video-dev/hls.js>

2.3.3. Dynamic Adaptive Streaming over HTTP

A Dynamic Adaptive Streaming over HTTP – DASH vagy MPEG-DASH, tekintve a fejlesztője ennek is az MPEG csoport volt – egy 2012-ben szabványosított protokoll. Hasonlít a HLS-hez, HTTP felett forgalmaz, sorozatba illeszt egymástól független packeteket. Ezeket a szegmentált packeteket egy manifestfájlban tartja számon a szerver (Media Presentation Description, MPD-fájl).[26]

A HLS-hez képest ez a protokoll kodekagnosztikus, ami annyit jelent, hogy nem kötődik videókodekhez, használható H.264, H.265, akár VP9 is.[15] Igyekeztek ezzel a protokollal egyezményesíteni a tartalomvédelmet is, Common Encryption (CENC) használ titkosításra. Digitális jogkezelésre (Digital Rights Management, DRM) is agnosztikus. Amióta a HLS támogatja az CMAF konténerformátumban való szállítást, azóta könnyen át lehet arról állni akár DASH protokollra is, ugyanis a DASH is CMAF-alapú. A DASH a HTTP/2 protokollt is támogatja, sőt HTTP/3-at is már UDP felett.

A HLS-hez hasonló elvekkel dolgozik a DASH is, illetve ez is egy ABR technológiájú protokoll. A DASH implementációjára JavaScript motorral rendelkező kliensekben – böngészők, mobiltelefonok stb. – a *dash.js*² könyvtárat használják legtöbb esetben.

2.3.4. WebRTC

A WebRTC (Web Real Time Communications) egy nyílt forráskódú projekt részeként alapult – támogatják kódbázisának fenntartását mind a Google, a Mozilla és az Opera csapatai is –, egy protokoll böngészőalapú valós idejű kommunikáció kialakítására. Ezt használja a Discord, a Google Chat és egyéb webes videóchat-alkalmazások.[23]

A források és a streamet figyelők számának kardinalitása szempontjából ez eltér az előzőekben ismertetettektől – amelyek egy forrás és több befogadóra optimalizált –, ez viszont peer-to-peer (P2P) alapú, tehát oda-vissza jellegű streamlést kell biztosítson, emiatt a WebRTC a legjobb választás, amennyiben a felhasználási célja az, hogy a felhasználók közvetlenül egymással kommunikálhassanak, és nem szükséges közbeiktatni egy központi médiaszervert. Ezenkívül a WebRTC-t egyre szélesebb körben próbálják alkalmazni a videó streaming területén is one-to-many közvetítésre is. Szabad felhasználású kodekeket alkalmaz videó- és hangkódolás terén (pl.: Opus, VP9).[12]

²<https://dashjs.org/>

3. fejezet

Követelmények

Egy nagyszabású, YouTube- vagy Netflix-szintű webes videó streaming szolgáltatás megvalósítása rendkívül összetett feladat, amely köré így kiterjedt technikai és üzleti követelményrendszert tudunk kialakítani. Egy ilyen rendszernek kezdetben biztosítani kell az alapvető funkciókat, amelyet hétköznapi webalkalmazások is megvalósítanak, mint például a videók lejátszása során felmerülő interakciók kezelése. Azonban ahogy a platformunk népszerűsége növekedhet, újabb és újabb infrastrukturális igények merülhetnek fel: ezek teszik ki a nem funkcionális követelményeket, amik kiterjednek például a tartalomterjesztés minőségére, a biztonsági felkészültségére a webalkalmazásnak.

A következőkben részletezem azokat a követelményeket, elvárásokat, amelyeket szem előtt tartottam a streaming szolgáltatás megtervezésekor és megvalósításakor.

3.1. Funkcionális követelmények

A streaming szolgáltatást alapvetően egy központosított kliensoldali webes felületen keresztül lehet elérni, azon keresztül tudják adminisztrátorok kezelni a tartalmat. Ennek a közvetlen frontoldali webes felületnek és a szolgáltatásainak a követelményeit érdemesnek tartottam csoportokba szedni:

- Felhasználókezelésre vonatkozó funkciócsoport
 - Habár a videók megtekintéséhez nincs szükség felhasználókezelésre és bejelentkezésre, viszont a videók kezeléséhez szükséges megvalósítani a felhasználók azonosítását és jogosultságkezelését.
 - A felhasználók AuthSCH¹ segítségével tudnak bejelentkezni a rendszer „stúdió” nevezetű oldalára, ahol a rendszer azonosítja őket AuthSCH-s profiljuk alapján, automatikusan jön létre felhasználói fiókjuk, regisztrációra külön nincs szükség.
 - A felhasználók közötti különbséget a rendszerben adminisztrátorok és normál felhasználók között teszem meg, az előbbieknél jogosultságuk van a videók

¹<https://vik.wiki/AuthSCH>

feltöltésére és kezelésére, az utóbbiaknak csupán bejelentkezésre későbbi adminisztrátorrá válásra, amennyiben másik adminisztrátor úgy dönt.

- Videóprojektek kezelésére vonatkozó funkciócsoport
 - Az adminisztrátorok a „stúdió” aloldalon tudnak új videóprojekteket létrehozni, amelyekhez a rendszer automatikusan generál egy egyedi azonosítót.
 - Egy videóprojekt létrehozása után tudunk a videókhoz tartozó metaadatokat adni, azokon módosítani, mint például a cím, leírás, borítókép, kategória, résztvevő stábtagnak.
 - A videóprojektben lehetőség van egy darab MP4 konténerformátumú videó feltöltésére.
 - Van lehetőség a videóprojekt teljes törlésére.
 - A feltöltés után a felhasználói felület visszajelzést kell adjon arról, hol tart a videó feltöltési folyamata, illetve a videókonvertálás folyamata a hálózati adatfolyamra való felkészítéshez.
 - A „stúdió” kívüli oldalon a videóprojektek listázva vannak, ahol a nézők megtekinthetik a videóprojektek konkrét videóit stream formájában.
- Élő közvetítés kezelésére vonatkozó funkciócsoport
 - Az oldal kezdetlegesen csupán egy élő közvetítés adását támogatja, az adminisztrátor a „stúdió” aloldalon tudja ezt az egyetlen élő közvetítést indítani.
 - A rendszer biztosítja, hogy fogadja egy külső forrásból (pl.: OBS Studio alkalmazásból) a felstreamelt videót a felhőn át és azt továbbítja a nézőknek.
 - Az oldalon kell, legyen egy útvonal a nézők számára, ahol a közvetítés élőben megtekinthető.

3.2. Nem funkcionális követelmények

A rendszerrel szemben támasztott nem funkcionális követelmények megállapításakor igyekeztem olyan dolgokra a hangsúlyt tenni, amelyek inkább számomra jelentenek kihívást, mivel nem terveztem a webalkalmazást úgy elkészíteni, hogy az valódi használatra készüljön – azaz valódi haszna legyen és legyenek élő felhasználói a nagyvilágból, csupán a kísérletnek volt része. Ezeket az elvárásokat a következőkben állapítottam meg:

- Biztonság
 - A megoldás kihasználja az AWS-felhő adta biztonsági lehetőségeket mind a hálózati struktúra kialakításakor, a webalkalmazás védelmezésére és a biztonságos kommunikáció (pl. HTTPS) kialakítására.
 - A webalkalmazásban a felhasználók autentikációját és autorizációját a JWT tokenek segítségével oldom meg.

- Hordozhatóság és könnyű karbantarthatóság
 - Konténerizálom a szerveralkalmazást, hogy könnyen lehessen telepíteni és futtatni, egy egységként lehessen kezelni.
 - A konténer és a buildelendő kódok automatikusan fordulnak és települnek a CI/CD-folyamat részeként.
 - A frissítések könnyedén kezelhetők, erre alkalmazásra kerül konténerképeket tároló Docker Registry is.
 - Eseményvezérelt architektúra az alkalmazás egyes folyamatainak megvalósítására és a szerveralkalmazás köré, hogy könnyen kiterjeszthető lehessen új funkcionalitásokkal.
 - Az infrastruktúra kód formájában (Infrastructure as Code, IaC) is dokumentált, hogy könnyen lehessen újraépíteni a rendszert.
- Elaszticitás
 - Alkalmazásra kerül olyan futtatókörnyezet, amelyben könnyedén lehet az erőforrásokat növelni és csökkenteni, hogy a rendszer mindig a szükséges kapacitással rendelkezzen.
 - Alkalmazásra kerül a CDN használata, hogy a videók gyorsan és megbízhatóan érhetőek legyenek el a nézők számára.
- Költséghatékonyság
 - Olyan szolgáltatások használata, amelyek csak a valóban szükséges erőforrásokat használják fel, és csak akkor, amikor azokra szükség van.
 - A szolgáltatásokat úgy tervezem meg, hogy a lehető legolcsóbban lehessen őket üzemeltetni a kísérletezés során.

4. fejezet

Felhasznált technológiák

A fejezet célja, hogy bemutassa a videó streaming szolgáltatások implementációjához felhasznált konkrét szoftvereket, webes és felhőszolgáltatásokat, és az azok közötti kapcsolatokat.

4.1. FFMpeg szoftvercsomag

Az FFMpeg¹ egy nyílt forráskódú és GPL-licenzelésű szoftvercsomag, amely képes videók és hangok kódolására, dekódolására, átalakítására (konvertálás), valamint streamelésre.[16] Az FFMpeg a legtöbb operációs rendszeren elérhető, és számos különböző formátumot, modern kodekeket támogat. Az FFMpeg a videók és hangok kódolására és dekódolására szolgáló kodekeket tartalmazza, valamint számos különböző formátumot támogat, beleértve az MPEG-4, H.264, H.265, VP8, VP9, AV1, AAC, AC-3, Opus, és sok más formátumot. Folyamatosan frissen tartják a kodekeket, aktív fejlesztőbázissal rendelkezik.

A lokális videókonvertálásra és streamelésre az FFMpeg-csomagból az azonos nevű ffmpeg parancssori interfészt használtam. Beépítettem a webszerver alkalmazásba egy külön szolgáltatásrészt, amely kihív a webszerver folyamatából és egy megfelelően felparaméterezett ffmpeg parancsot futtat a videókonvertálásra és streamelés megkezdésére aszinkron módon, a kimeneti fájlokat a megfelelő helyre menti.

A felhőalapú megoldásban már nem került felhasználásra az FFMpeg, mivel az AWS Elemental szolgáltatásokat használtam a videó kódolásra és streamelésre, viszont a szoftvercsomag közvetlen megismerése a videókonvertálás folyamatának megértését és a konvertálási folyamatok felparaméterezési lehetőségeinek mélyebb átlátását segítette.

4.2. Open Broadcaster Software

Az Open Broadcaster Software Studio (OBS Studio²) egy nyílt forráskódú, szabad szoftver, amelyet elsősorban élő közvetítésekhez és képernyőörögzítéshez használnak. Elterjedt Twitch-felhasználók körében. A program támogatja a Windows, macOS és Linux operáci-

¹<https://www.ffmpeg.org/>

²<https://obsproject.com/>

ós rendszereket, és számos beállítási lehetőséget biztosít a felhasználók számára. Az OBS lehetővé teszi több videó- és hangforrás kombinálását, ennek köszönhetően webkamera felvétele, mikrofon inputja, az éppen használt képernyő képe vagy előre rögzített videók is kombinálhatóak a stúdió műszerfalán.

A szoftver kompatibilis a legnépszerűbb streamingplatformokkal, így például a YouTube-ra és a Twitchre is lehet feltölteni vele, és lehetőséget biztosít saját egyedi RTMP-szerverekhez való csatlakozásra is annak felkonfigurálásával. Az OBS Studio megbízható eszköz azok számára is, akik professzionális szintű élő közvetítést szeretnének megvalósítani.

4.3. Amazon Web Services

Az Amazon Web Services (AWS) a világ egyik legjobban elterjedt, legnagyobb szerverfarmjait fenntartó, nagy hírév vállalatok által is megbízható felhőszolgáltatója. Felhasználói számára számítási, hálózati, adattárolási célokat megvalósító szolgáltatások széles palettáját kínálja. Felhasználják az AWS-t a mesterséges intelligencia területén; valamint kiterjedt adatbázisok, adatfeldolgozó rendszerek építésére; megbízható és könnyen skálázható webes szoftverrendszerek kialakítására.

A felhasználók a szolgáltatásokhoz az AWS Management Console webes felületen keresztül, vagy az AWS Command Line Interface (AWS CLI) parancssori interfészén keresztül férhetnek hozzá. Az egyes felhasználók fel tudnak állítani maguknak egy vagy több AWS-fiókot, amelyek a számlázás és a jogosultságkezelés szempontjából elkülönülhetnek egymástól.

A fiókon belül lehetőséget kapunk granulás jogosultságkezelésre, azaz az egyes felhasználók, szolgáltatások, vagy szolgáltatásrészek számára különböző alacsony szintű jogosultságokat adhatunk meg.

Az AWS regionális adatközpontokat üzemeltet a világ számos pontján, amelyek közül a felhasználók választhatnak, hogy melyik adatközpontban szeretnének szolgáltatásokat futtatni.

A költségeket „pay-as-you-go” alapelv alapján számolják fel, azaz a felhasználók csak az általuk használt szolgáltatások számítási kapacitásáért, a tárhelyért, az adatközpontból kifelé történő hálózati forgalomért fizetnek.

4.3.1. AWS Elemental és Amazon IVS

Az Elemental Technologies 2006-ban indította vállalkozását streamingmegoldások eladására, egy fő mérőldkövük volt, amikor a szolgáltatásaikkal került közvetítésre a 2012-es nyári olimpiai játékok Londonban. Az Elemental Technologies 2015-ben került az Amazon Web Services tulajdonába, azóta az AWS Elemental néven futó szolgáltatásokat kínálja az AWS-felhőben. A fő célja a szolgáltatáscsomagnak, hogy óriási célközönségek számára is megbízható streamközvetítési megoldásokat kínáljon, amelyeket könnyen lehet skálázni, és amelyek a legújabb videó kódolásokat és -technológiákat alkalmazzák.[8]

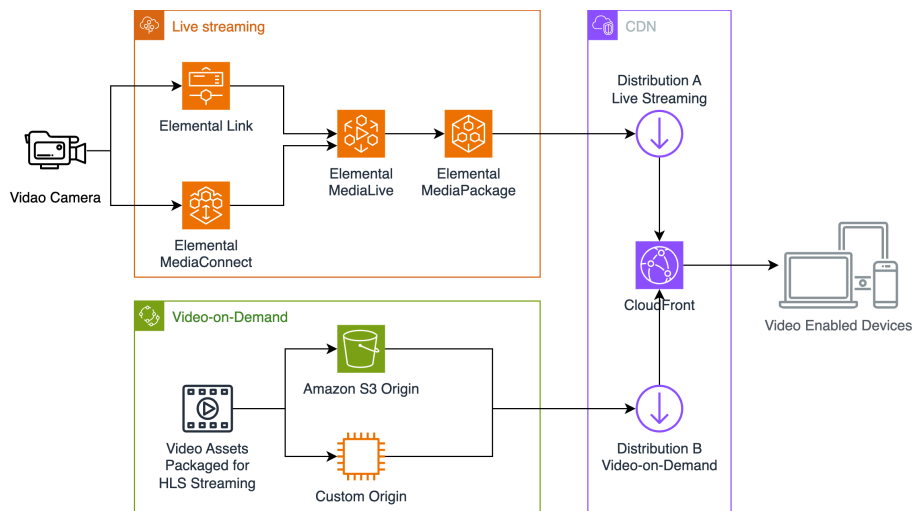
Szoftveres megoldásaik közé tartoznak a következők:

- **Elemental MediaConvert:** A MediaConvert egy felhőalapú videó kódoló szolgáltatás, Software-as-a-Serviceként viselkedik, egy API-t ad, amelyen keresztül kódolási munkafolyamatokat („jobokat”) indíthatunk. A MediaConvert támogatja a legnépszerűbb videóformátumokat, mint például a H.264, H.265, és a VP9, valamint a legújabb HDR (High Dynamic Range) és Dolby Vision technológiákat is. HLS streamre is képes felkészíteni a videókat. Csupán fel kell tölteni a forrásvideót egy S3-vödörbe, majd a konvertálás után a kimeneti videók számára is egy S3-vödröt tudunk megadni.
- **Elemental MediaLive:** Az Elemental MediaLive egy élő videó kódoló szolgáltatás, amely lehetővé teszi a felhasználók számára, hogy élő videóadásokat fogadjanak és kódoljanak át a felhőben. A MediaLive támogatja a legnépszerűbb élővideó feltöltési protokollokat, így az RTMP-t is. Ennek a használata már bonyolultabb, mint a MediaConverté, nem szimplán csak egy API meghívásaként kell elképzelni. Külön csatornákat lehet benne definiálni, azokhoz inputot/inputokat rendelni, ezután pedig a kódolási munkafolyamatokat felkonfigurálni. Az AWS sokféle nyelvben garantál SDK-kat, amelyek segítségével könnyen lehet automatizáltan MediaLive-csatornákat indítani külön-külön adásokhoz.
- **Elemental MediaPackage:** Az Elemental MediaPackage készíti elő, csomagolja a videófolyamat hálózati protokollokon szállítmányozásra, garantálja a biztonságos és folyamatos tartalomtovábbítást. Biztosíthatja VOD-ok S3-ból való továbbbocsztását, vagy élők továbbbocsztását a MediaLive-ből. A MediaPackage támogatja a legnépszerűbb kliensfelőli protokollokat, mint például az HLS, DASH és a Microsoft Smooth Streaming. Könnyedén integrálható CloudFront-disztribúciókba.

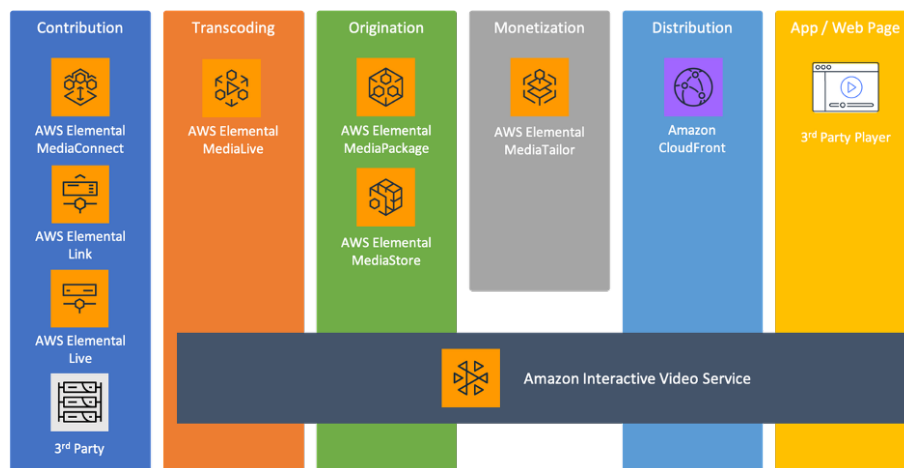
Érdemes még a szoftveres megoldások közt megemlíteni az Elemental MediaConnect, amely egy Quality of Service (QoS) réteget biztosít a streamet fogadók és az AWS-felhő között, megbízható és biztonságos hálózati kapcsolatot biztosít. Ismert lehet még az Elemental MediaTailor, amely lehetővé teszi a reklámok beillesztését a videófolyamainkba. Korábban még a felhozatalba tartozott, azonban kivezetésre kerül már az Elemental MediaStore 2025. november 13-áig, amely egy objektumtároló szolgáltatás volt, viszont már az Amazon S3 kiváltotta, mivel az már erős read-after-write konzisztenciát tud biztosítani 2020 óta.[13]

Ezekon kívül az AWS szolgáltató még fizikai hardvereket is a streaming könnyítésére és a nagy számításigények kiszolgálására, ezek közé tartozik például a AWS Elemental Link, amely egy HDMI- és SDI-portokkal rendelkező eszköz, lehetővé teszi a helyszíni videóforrások közvetlenül a felhőbe való továbbítását. A szoftveres és hardveres megoldások összekötésére egy példát szolgáltató mind VOD-ok és élő adások kiszolgálására a 4.1. ábra.

Az Amazon Interactive Video Service (IVS) egy teljesen AWS-kezelt, skálázható, és megbízható élő videó streaming Software-as-a-Service (SaaS), amely lehetővé teszi a fejlesztők számára, hogy gond nélkül integrálják az élő streaming funkcionalitását a saját alkalmazásaikba. Az IVS az Elemental szolgáltatásokhoz képest end-to-end megoldást kínál



4.1. ábra. Példa AWS Elemental szolgáltatások architektúrába kötésére.



4.2. ábra. Amazon IVS és az AWS Elemental szolgáltatások összehasonlítása.

kis késleltetésű többnézős alkalmazásra: HLS-alapú kiszolgálás, körülbelül 5 másodperces késleltetést szokott biztosítani; valamint valós idejű alkalmazásra is: WebRTC-alapú kiszolgálás.[28] A forrásvideó-kódolástól a tartalomkiszolgálásig minden szükséges funkciót biztosít (4.2. ábra³), nekünk csupán a Software Development Kitjét (SDK) kell használni a saját alkalmazásunkban, és a többit az AWS-re bízhatjuk. Ezenkívül biztosít olyan funkciókat, mint chatszobák és szavazások szolgáltatása, amelyeket könnyen integrálhatunk ezekbe az adásokba.

Ezen szolgáltatás ismertetése a megértést és a választás indoklását szolgálja későbbi fejezetekben, az Amazon IVS nem került felhasználásra a konkrét lefejlesztett rendszerben.

³A kép forrása: <https://aws.amazon.com/blogs/media/awse-choosing-aws-live-streaming-solution-for-use-case/>

4.3.2. AWS-szolgáltatások hálózatiépítésre

Az Amazon Virtual Private Cloud (Amazon VPC) egy virtuális hálózati környezet. Benne megvalósítható, hogy az AWS publikus felhőjén belül is privát és publikus saját hálózatokat hozzunk létre. Az Amazon VPC segítségével a felhasználók teljes kontrollt gyakorolhatnak a virtuális hálózati környezetük felett, beleértve a VPC-k alatti alhálózatok IP-tartományainak konfigurálását, útvonaltáblák kitöltését, a hálózati interfészek/portok korlátozásait, egyéb hálózati eszközök beillesztését (NAT Gateway-ek, Internet Gateway-ek, valamint Transit Gateway-ek), felülvizsgálható benne a hálózati teljesítmény.

Különböző eszközöket kínál a felhasználók számára, hogy biztonságossá tegyék az AWS-felhőn belüli hálózati környezetüket. Priváttá tehetik alhálózataikat, bevezethetnek állapotmentes korlátozást ki- és bemenő forgalomra (engedélyezett IP-tartományok és portok megadásával), erre jók alhálózatokra alkalmazható Network Access Control Listek (NACL), vagy a konkrét erőforrásokra alkalmazható Security Groupok. Állapottartó megoldás az AWS Network Firewall. VPC Flow Logok bekötésével monitorozható válik a hálózati forgalom is.

Az Application Load Balancer (ALB) az Amazon Elastic Load Balancer (ELB) egy fajtája, ISO–OSI Layer 7 szinten, azaz alkalmazásrétegek szintjén működő terheléselosztó hálózati eszköz. Automatikusan skálázódik, lehetővé teszi a felhasználók számára, hogy egy vagy több szerver példány között egyenletesen eloszthassák a beérkező HTTP- és HTTPS-kéréseket. Az ALB képes a kéréseket a kérések fejlécei alapján vagy a kérések útvonala alapján elkülöníteni a forgalmat.

4.3.3. Amazon S3

Az Amazon Simple Storage Service (Amazon S3) egy objektumtároló szolgáltatás, amely lehetővé teszi a felhasználók számára, hogy nagy mennyiségű adatot tároljanak az AWS-felhőben „vödörökben” (bucketokban). Az objektum egy fájl és a fájl leíró metaadatok közösen. A vödör az objektumok tárolója.

4.3.4. Amazon CloudFront

Az előző fejezet egy szekciójában már említésre került a CDN mint a Video-on-Demand alapú streaming szolgáltatások egyik kulcsfontosságú eleme. Az Amazon CloudFront egy globális kiterjedésű CDN, amely lehetővé teszi a felhasználók számára, hogy a tartalmat hozzájuk közelebbi szervereken tárolt cache-ből töltsék le, ezáltal csökkentve a késleltetést, csökkentve a központi szerverek terhelését, és növelve a letöltési sebességet. Tartalmaink csoportosítására CloudFront-„disztribúciókat” használunk.

A disztribúciók különböző URI-útvonalakon akár különböző CDN-forrásokból – úgynevezett „originekből” – tudnak tartalmat kiszolgálni: ilyen origin lehet egy S3-vödör, AWS Elemental MediaPackage-alapú élő adást sugárzó csatorna, Amazon Application Load Balancer (ALB) példány, vagy akár egy egyéni HTTP-szerver is saját doménnevvel.

4.3.5. AWS-szolgáltatások számítási kapacitásra

Az Amazon Elastic Container Service (ECS) arra szolgál, hogy konténeralapú alkalmazásokat, szoftvercsomagokat futtathassunk a felhőszolgáltatónál. Az ECS segítségével a felhasználók könnyen futtathatnak és skálázhatnak konténereket anélkül, hogy a konténerek futtatásához szükséges infrastruktúra mélyén futó szervergépeket, valamint azok életciklusát, operációs rendszerének patch-elését kellene kezelniük – ezek menedzselését az AWS Fargate motor veszi át, mi csupán a környezeti paramétereket kell felkonfiguráljuk az igényeinknek megfelelően.

Ilyen paraméterek a konténerek képei, a konténerek alapvető számítási erőforrásai (CPU-magok száma, memória mérete), a konténerek hálózati beállításai (porttovábbítások, alkalmazott Security Group), a konténerek naplózása (hova továbbítódjanak a futtatás során a naplók), és a konténerek hozzáférési jogosultságai az AWS-felhőn belüli más szolgáltatásokhoz. Könnyedén kapcsolható össze Amazon ALB-példánnyal.

Tipikusan alkalmazott az ECS párban az Amazon Elastic Container Registry (ECR) szolgáltatással, amely egy konténerképek tárolására szolgáló privát Docker Registry, amely lehetővé teszi a felhasználók számára, hogy a konténerek képeit biztonságosan tárolják és kezeljék az AWS-felhőben.

Az Amazon Relational Database Service (RDS) egy relációs adatbázis szolgáltatás, amely segít, hogy könnyen és hatékonyan hozhassunk létre, üzemeltessünk és skálázzunk relációs adatbázisokat az AWS-felhőben. Az RDS támogatja a legnépszerűbb relációs adatbázis motorokat, mint például a PostgreSQL, MySQL, MariaDB, Oracle, és SQL Server.

Képes automatikusan kezelni az adatbázisok frissítéseinek telepítését és a folyamatos biztonsági mentéseket. Mivel ezek is konkrét szervereket igényelnek, az RDS is könnyen integrálható az Amazon VPC hálózati környezetébe, a hálózati védelme is biztosítható.

4.3.6. Kiegészítő AWS-szolgáltatások

A konténerek orkesztrációjának kiegészítésére számos könnyen élesíthető és ECS-hez integrálható szolgáltatás áll rendelkezésre az AWS-felhőben, amelyek közül a legelterjedtebbek az Amazon CloudWatch Logs, az AWS Lambda és a Amazon EventBridge.

Az Amazon CloudWatch Logs egy naplózó és monitorozó szolgáltatás, amely lehetővé teszi a felhasználók számára, hogy a konténerek futtatása során keletkező naplókat gyűjtsék, tárolják, és vizsgálják az ezekből származó metrikákat is akár.

Az AWS Lambda egy serverless Function-as-a-Service (FaaS) szolgáltatás, amely lehetővé teszi kód függvényszerű futtatását anélkül, hogy szükség lenne a szerverek vagy a futtatási környezet menedzselésére. A Lambda-függvény eseményekre reagálva kerül meghívásra, például HTTP-kérésekre, adatbázis-eseményekre, vagy más AWS-szolgáltatások eseményeire.

Ezzel kapcsolatban kerül a képbe az EventBridge, az AWS központi eseménykezelő szolgáltatása, amely lehetővé teszi az egyes AWS-felhőszolgáltatásokon futó alrendszerek közötti kommunikációt. Segítségével szűrhetünk eseményekre, azokat könnyen továbbít-

hatjuk az egyes AWS-szolgáltatások között, a célpontja egy EventBridge által elkapott eseménynek ennek megfelelően egy Lambda-függvény is lehet.

4.4. A webes komponensek technológiái

A különböző felhőszolgáltatásokon futó kódbázisokat elterjedt webes technológiák segítségével fejlesztettem. Ezen technológiák kerülnek bemutatásra a következő szekciókban.

4.4.1. TypeScript és JavaScript nyelvek

A JavaScript egy dinamikusan és gyengén típusos, interpretált programozási nyelv, amelyet webes alkalmazások fejlesztésére használnak. A böngészőben is JavaScript fut legtöbbször modern keretrendszerek (React.js, Vue.js vagy Angular) támogatásával a Document Object Model (DOM) renderelésére, manipulálására, ezzel tudjuk lehetővé tenni a kliensoldali webes alkalmazások interaktív működését, a felhasználói események kezelését, a HTTP-kérések küldését.

Ezenkívül ez a nyelv használható szerveroldali környezetben is, az erre használatos *Node.js* egy futtatókörnyezet, amely lehetővé teszi a JavaScript-kód futtatását a szerveroldalon is. A Node.js a V8 JavaScript-motorra épül, amely a Google Chrome böngészőben is fut, eseményvezérelt architektúrában szolgál ki függvényhívásokat, és aszinkron I/O-működést biztosít, ami lehetővé teszi a blokkoló műveletek nélküli működést, maximalizálja a skálázhatóságot.[24]

A Node.js biztosítja különböző könyvtárakkal a HTTPS-alapú hálózati kommunikációt, a fájlrendszerműveleteket, a processzkezelést, a környezeti változók olvasását. A funkcionalitások kiterjesztésére szokás használni JavaScript-modulokat, ekkor kerül közép pontba az Node Package Manager (NPM) ökoszisztémája. Az NPM csomagkezelő segítségével könnyen telepíthetünk többek között Model-View-Controller alapú (MVC) keretrendszereket is (pl.: Express.js, NestJS), Object Relational Mapping (ORM) eszközöket (pl.: Prisma, TypeORM), SDK-kat (pl.: AWS SDK), vagy akár különböző adatbázis- és cache-kezelőkhöz (pl.: PostgreSQL, Redis) drivereket a webszerverünk kiegészítésére.

A TypeScript egy superhalmaza a JavaScriptnek – azaz a JavaScript szintaxisát bővíti ki –, amely szigorú és statikus típusosságot ad hozzá. A nyelvben írt kód a TypeScript-fordító (*tsc* – TypeScript Compiler, CLI-alapú eszköz) segítségével JavaScript-kóddá alakítható. A TypeScript segítségével a fejlesztők könnyebben tudják a kódjukat karbantartani, mivel a típusok segítenek a hibák felismerésében, statikus analízisben, és a kódolás során a fejlesztőknek segítségére lehet a kód kiegészítésében is. Mind kliens- és szerveroldalon is használatos, a Node.js natívan nem, de vannak futtatókörnyezetek, amelyek fordítás nélkül is már támogatja a TypeScript futtatását (pl.: Deno).

A TypeScript-alapú technológiákból felépülő „stackek” előnye, hogy a kliens- és szerveroldali kódokat ugyanabban a nyelvben írhatjuk meg, így a fejlesztőknek nem kell külön-külön nyelveket és környezeteket tanulniuk, és a kódok könnyebben átírhatók, újrahasznosíthatók, és könnyebben karbantarthatók.

Mellékesen érdemes még megemlíteni, hogy az AWS CloudFront szolgáltatása lehetőséget nyújt az felhasználóhoz közel futó edge szerverfarmokon nagyon kicsi számításigényű függvényeket futtatni, amelyeket a felhasználói kérésekre lehet közvetlen ráereszteni. Ezeknek két típusa is létezik, a CloudFront Function-függvények felprogramozása egy korlátozottabb nyelvi lehetőségekkel rendelkező JavaScriptben történik, míg a Lambda@Edge-függvények logikája lehet Node.js futtatókörnyezet feletti JavaScriptben, illetve akár Python nyelven megírva.

4.4.2. React

A React⁴ egy nyílt forráskódú JavaScript-könyvtár, amelyet a Facebook (ma Meta) vállalata fejlesztette ki még akkoriban belső fejlesztőeszközként. Single Page Applicationök (SPA) fejlesztésére használt. A React a komponensalapú fejlesztést támogatja, amivel úgy tudunk építkezni, hogy az felhasználói felületet (angolul *User Interface*, röviden UI) kisebb, újrahasznosítható építőelemekre bonthassuk. Az egyik fő előnye a *virtuális DOM*, amely hatékonyan kezeli a változásokat és javítja a teljesítményt.

A React alapvetően klienoldali renderelést (Client-Side Rendering, CSR) használ, ami azt jelenti, hogy az alkalmazás a böngészőben fut, és a szerver csak egy alapszintű HTML-t küld, hozzá a JavaScriptet. Azonban nagyobb alkalmazásoknál gyakran szükség van más renderelési módszerekre: Server-Side Rendering (SSR) esetén A React-alkalmazás HTML-jét a szerver generálja le és küldi el a böngészőnek. Ez javítja a teljesítményt és a keresőoptimalizálást (Search Engine Optimization, SEO), mert a keresőmotorok számára az oldal már előre renderelve érkezik. Egy másik ilyen módszer a Static Site Generation (SSG), amely során az oldalak statikusan generálódnak a buildelési folyamat során. Ez gyors betöltési időt eredményez. A Next.js egy React köré épített keretrendszer, amely mind SSG- és SSR-funkcióval is rendelkezik.

Gyakori, hogy a keretrendszer nélkül csupán statikus weboldalakat generálunk React felhasználásával, a Vite egy olyan buildelési eszköz, amely gyorsítja a fejlesztési folyamatot, és képes React- – és akár Vue.js- vagy egyéb – könyvtárral írt TypeScript- vagy JavaScript-kódot is statikus weboldalakká generálni. A TypeScriptben írt React-kód fájlkiterjesztése `.tsx`, illetve `.jsx`, ha JavaScriptben írodik.

A React egy nagyon népszerű keretrendszer, amelyet a fejlesztők széles körben használnak, és amelynek számos kiegészítő könyvtára és eszköze van, amelyek segítségével gyorsan és hatékonyan lehet webes felületeket fejleszteni. Ennek megfelelően széleskörben támogatott és szeretett könyvtárakat lehet beépíteni a React-alapú alkalmazásokba, mint például a React Router, SPA-n belüli routingra; a TanStack Query, egyszerűsített állapotkezelő aszinkron kérésekre; a React Hook Forms, gyors és hatékony formkezelésre.

⁴<https://react.dev/>

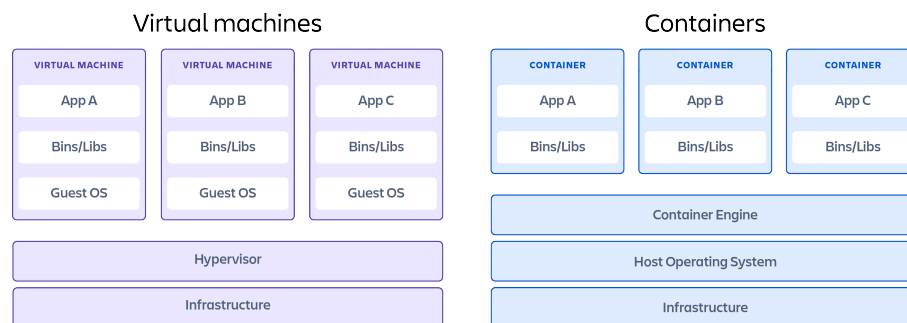
4.5. Üzemeltetési technológiák

Végül pedig a fejlesztés során használt üzemeltetési technológiákat ismertetem, amelyek segítik a karbantarthatóságot, amelyek segítségével a fejlesztők könnyen tudják a kódbázisból az alkalmazásokat futtatni, az infrastruktúrát felhúzni.

4.5.1. Docker

A virtualizáció egy típusa a *konténerizáció*, amely lehetővé teszi a fejlesztők számára, hogy az alkalmazásokat virtuális gépeknél egyszerűbb „konténerekbe” csomagolják, abból képet generáljanak, azt pedig könnyen osszák tovább, és ezekből a képekből konténereket futtathassanak a számítógépükön vagy épp a felhőben. Egy virtuális gép a gazdagép hardvereit virtualizálja, a konténer pedig a gazda operációs rendszert virtualizálja, azaz a konténerek alatt közös a *kernel* (4.3. ábra⁵). Ezzel elveszik a teljes izoláció, azaz a biztonság, de a konténerek könnyebbek, gyorsabban indulnak – hisz nincs *bootolási idő* –, és kevesebb erőforrást használnak.

A konténer egyfajta szabványosított egység, amely tartalmazza az alkalmazás kódját, a szükséges függőségeket (könyvtárakat), a konfigurációs fájlokat, és amire az alkalmazásnak szüksége van a futtatáshoz.



4.3. ábra. Virtuális gépek és konténerek architektúráis összehasonlítása.

A Docker egy konténerizációs fejlesztői környezet, a mélyén a *containerd* névre keresztelt motor fut konténerizációs futtatókörnyezetként.[19] Lehetővé teszi a konténerek létrehozását, indítását, kezelését és átvitelét. Virtuális tárhelyet és hálózatot biztosít a konténerek számára, és lehetővé teszi a konténerek közötti biztonságos kommunikációt. A Docker egy nyílt forráskódú projekt, amelyet a fejlesztők széles körben használnak a konténerizált alkalmazások fejlesztésére és futtatására.

4.5.2. GitHub

Szoftverrendszerek, alkalmazások fejlesztése során szinte elengedhetetlen a verziókezelés, amelynek segítségével a fejlesztők nyomonkövethetik a kódbázis változásait, visszaállíthatják az előző verziókat, és könnyen együtt tudnak dolgozni a kódon. Erre a munkafolyamatra

⁵A kép forrása: <https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms>

az egyik legelterjedtebb Source Code Management (SCM) eszköz a Git verziókezelő. A Git egy elosztott verziókezelő rendszer. Minden fejlesztő saját gépén tárolja a teljes kódbázisát, majd a módosításokat a felhőben lévő tárolóval szinkronizálhatja.

A Git szoftver köré széleskörben találhatunk felhőtárhely-szolgáltatókat, ezek közül a legnépszerűbb a GitHub⁶. A GitHub a tárhelyen kívül sok más funkcionalitást is szolgáltat a hatékony együttműködés és kódgondozás kivitelezésére, egy ilyen szolgáltatása a GitHub Actions, amely CI/CD-folyamatok kezelésére egy eszköz, olyan folyamatokra hasznosítható, mint a statikus ellenőrzés, a build folyamatok automatizálása, és például a kód AWS-re való kiélesítése is.

4.5.3. Terraform

A Terraform⁷ egy elterjedt Infrastructure as Code (IaC) eszköz, amely lehetővé teszi a felhasználók számára, hogy infrastruktúrát definiáljanak kódban, és ezt az infrastruktúrát automatizáltan hozzák létre, módosítsák és töröljék. A Terraform a felhőszolgáltatók API-jait, illetve Cloud Development Kitjét (CDK) használja az változtatások érvényre juttatására. Go nyelven íródott a motorja, a Terraform IaC-ra pedig a saját HashiCorp Configuration Language (HCL) nyelvét ajánlja, amely egy deklaratív nyelv.

⁶<https://github.com/about>

⁷<https://www.terraform.io/>

5. fejezet

A tervezett architektúra

A következőkben részletezem a tervezett architektúrát két nézetben: első körben a logikai felépítést mutatom be, majd a fizikai felépítést az AWS-felhőben. A logikai felépítés a követelmények alapján készül függetlenül egy választott platformtól, tükrözi az alapszintű kommunikációs modelljét az egyes részkomponensei közt a rendszernek, amikre lehet bontani a teljes egészet. Segíti a megértését a mérnök terveinek a megrendelő számára is, aki nem feltétlen teljesen tapasztalt a területen. A fizikai felépítés konkrét szoftverkomponenseket definiál, amelyeket fejleszteni szükségeltetik a rendszer megvalósításához. Inkább a fejlesztőknek szól, akik a rendszer megvalósításáért felelősek.

5.1. Logikai felépítés

A logikai felépítést legjobban a 5.1. ábra tudja jól bemutatni. A rendszer három fő komponenscsoportra bontható a követelmények alapján: a kliensközeli csoportra, a szerveroldali csoportra és az „orkesztrációs” csoportra. A kliensközeli csoport szolgálja ki a felhasználókat tartalommal, a szerveroldali csoport kezeli az hagyományos üzleti logikát, ahogy az egy többretegű webalkalmazás megvalósításánál is megszokott az iparban. Az utolsó csoportot „orkesztrációs” csoportnak neveztem el, mivel ez szereli fel a két csoportot tartalommal, kezeli események hatására a videófeldolgozást a háttérben.

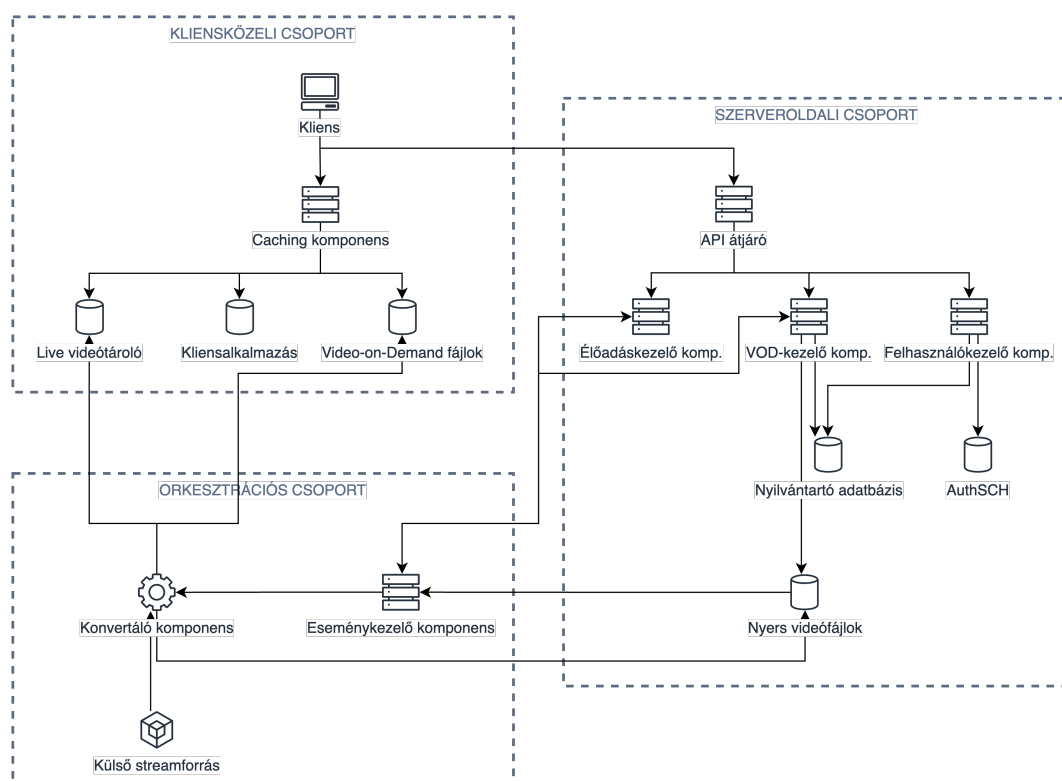
Érdekes lehet megfigyelni, hogy az egyes csoportok konkrét megvalósítása akár kicserélhetővé válik, egy-egy csoport mögötti teljes szoftvercsomag könnyen lekapcsolható a másik kettőről, csupán jól definiált és agnosztikus API-okra van szükség.

A kliensközeli csoportban a felhasználók a webalkalmazást saját kliensükre egy erre szolgáló tárolóból kell letöltsék, hasonlóképp érik el a videókat két tárolóból. Ezzel a funkcionális követelmények megvalósulnak az élők elérésére, a VOD-ok elérésére, a weboldalon a videóprojektek UI-jára vonatkozó követelmények. A nem funkcionális követelményekből pedig teljesül a caching komponenssel, hogy a videók gyorsan és megbízhatóan érhetőek el a nézők számára.

A szerveroldali csoportban darabokra szedve tulajdonképpen egy REST API helyezkedik el élő adást kezelő, VOD-kezelő és felhasználókezelő komponensekkel közösen. Ez API-átjáró mögé van helyezve, amely a biztonságos kommunikációt tudja biztosítani, a

felhasználók autentikációját és autorizációját tudja kezelni együttműködve a felhasználókezelő komponenssel, valamint a kliensnek a megfelelő interakciós lehetőséget tud szolgáltatni. A szerveroldali csoportban a videók feltöltése történik csupán, az egyes entitásokról a változások nyilvántartása kerül még tárolásra.

A szerveroldali csoport események formájában kommunikál az élőadás- és VOD-kezelő az „orkesztrációs” csoportban lévő eseménykezelővel. Az orkesztrációs réteg fogja a két csoportot össze, események hatására indítat konvertálást a konvertáló komponenssel. Ez a komponens tölti fel a videókat a kliensközeli csoport felé, tart összeköttetést a live streaming külső forrásával.



5.1. ábra. Logikai felépítés a követelmények alapján.

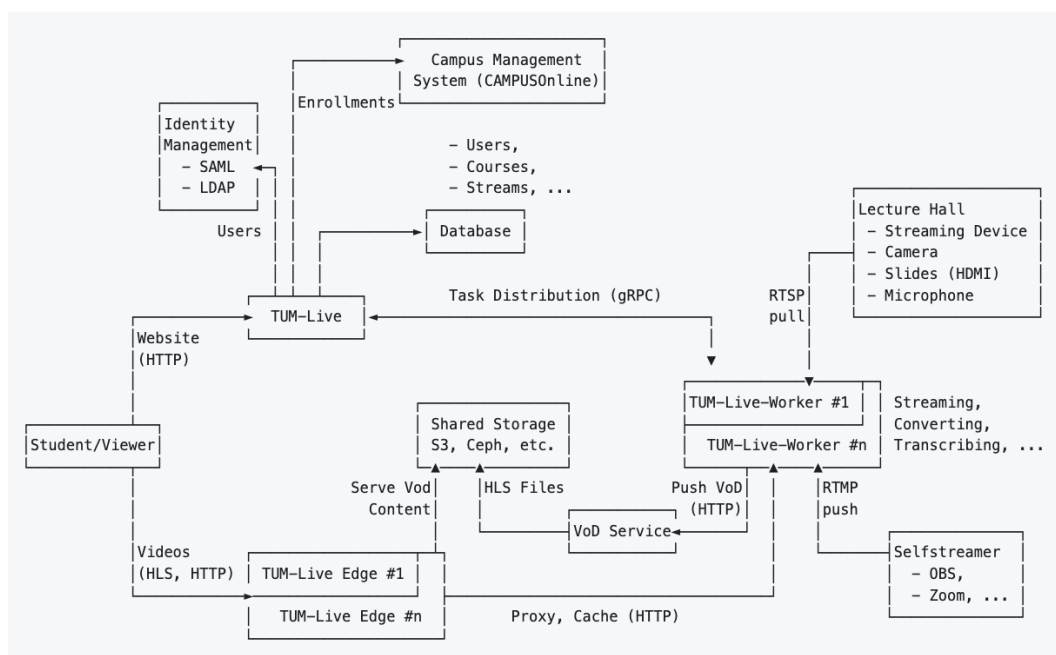
Megjegyzés: az itt feltüntetett komponensek élőadás-, VOD- és felhasználókezelő moduljai a későbbiekben a konkrét szoftverarchitektúrában monolit struktúrában egy közös szerveralkalmazásba kerültek bele, azon belül kerültek modularizálásra az üzleti logikában.

A korábbi 2.2. alfejezetben megismert protokollok közül a tervezés során az egyszerű implementáció és a jól támogatottság szempontjából a HTTP Live Streaming (HLS) protokollt választottam a VOD és live streaming fogadó oldalán. Az élő közvetítéshez a Real-Time Messaging Protocol (RTMP) protokollt választottam, ehhez az OBS Studiót használtam a felstreameléshez.

5.1.1. Összehasonlítás egy hasonló rendszerrel

A tervek igazolásához segítségül kerestem az interneten nyílt forráskódú hasonló megoldásokat is. Megtaláltam a Technische Universität München (TUM) egy hallgatói csoportja,

a TUM-Dev által fejlesztett az egyetemen is használt VOD és live streaming szolgáltatását, a GoCastot¹. Ez a rendszer önállóan hosztolható szoftvereket komponál össze, nem felhőnatív. A rendszerben hasonló absztrakt tervek lehet megfigyelni az én megoldásaimhoz (5.2. ábra), ugyanígy HLS-sel szolgálják ki a tartalmakat (lásd *TUM-Live Edge* példányok), viszont a live streamingre saját többportos workereket alkalmaznak (lásd *TUM-Live-Worker* példányok), lehetőség ad viszont saját streamerből RTMP-n keresztül feltölteni élő közvetítést, ahogy én is megvalósítom a saját megoldásomban. Külön mikroszolgáltatás biztosítja a live és VOD streamingen kívüli funkciókat, a TUM-Live. Ez a megoldás nem teszi lehetővé, hogy a rendszeren kívül készült videót lehessen feltölteni és VOD-ként elérhetővé tenni rajta keresztül.



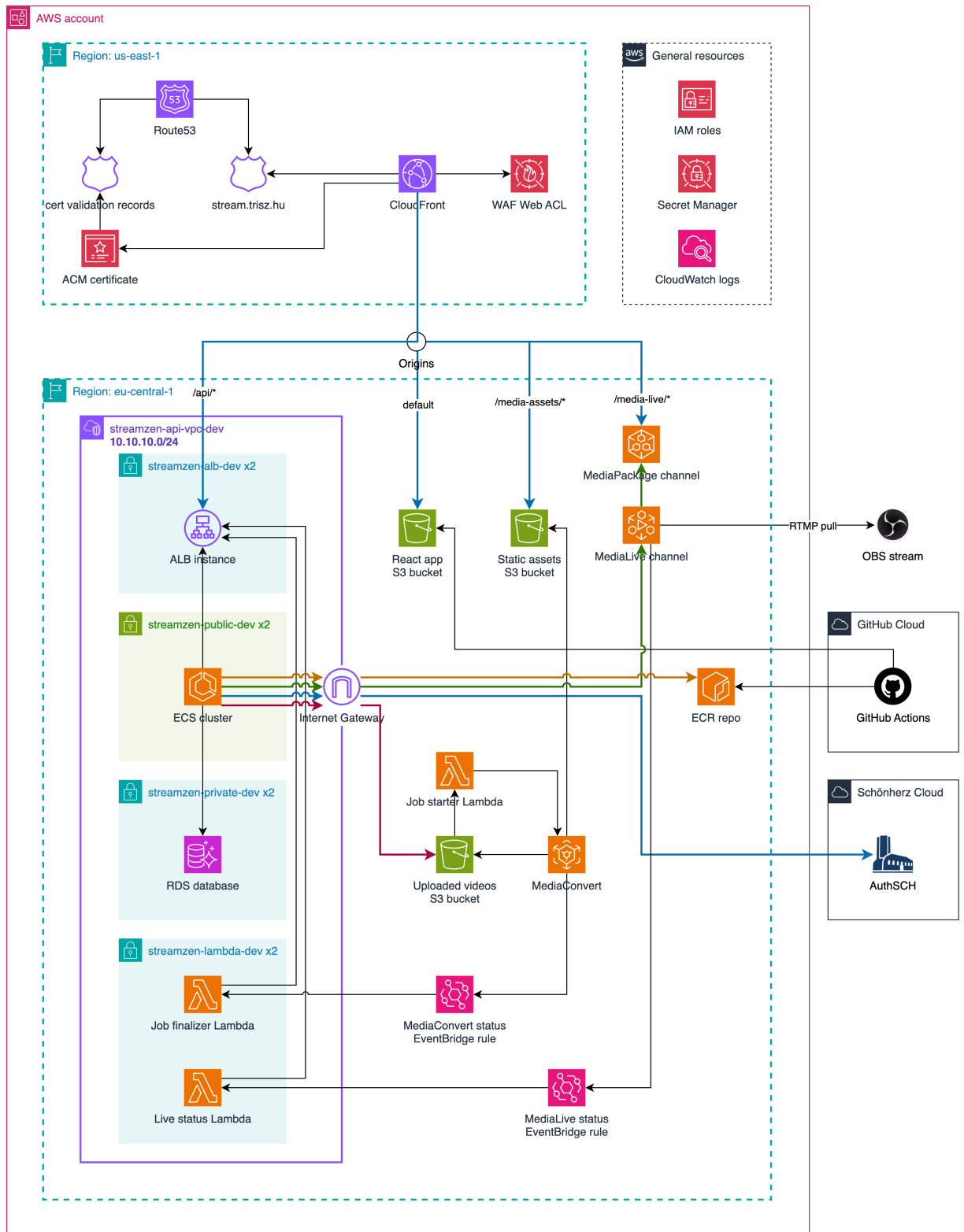
5.2. ábra. A GoCast architektúrája a dokumentációból.

5.2. Fizikai felépítés AWS-re specializáltan

A rendszert az átlátható kezelés érdekében az AWS-felhőben egy teljesen erre külön készített AWS-fiókba helyeztem, így terveztem meg az architektúrát is. A rendszer egyszerűsített fizikai felülnézetét az AWS-fiókban, az AWS-erőforrások összeköttetését jól összefoglalja az 5.3. ábra.

Az egyes erőforrások csoportokra bonthatóak. A kliensközeli csoportot a CloudFront CDN szolgálja ki, készítettem számára egy `stream.trisz.hu` domain alatti Route 53 DNS-zónát is, illetve egy SSL-tanúsítványt. A védelmezésre egy AWS Web Application Firewall (WAF) web Access Control List (web ACL) is bekerült a disztribúció elé. Ezen erőforrások habár fizikailag az edge szerverfarmokra kerülnek ki, logikailag őket a `us-east-1`, azaz az észak-virginiai régióban kell elhelyezni.

¹<https://github.com/TUM-Dev/gocast>



5.3. ábra. Az AWS-fiók erőforrásainak logikai kapcsolata.

A disztribúció után jönnek egy új rétegben először egy ALB-példány, ez és a mögötte lakó erőforrások az eu-central-1 (frankfurti) régió belül is egy saját hálózatba, azaz AWS VPC-be kerültek. Ezenkívül videók S3-vödörét, a React alkalmazás vödörét és a live csatornát mind külön originként tettem a disztribúció mögé, külön-külön útvonalak mintázatokra illeszkedve.

A RTMP-alapú live stream fogadását OBS Studióból a MediaLive kezeli, ami a MediaPackage segítségével továbbítja egy csatornán a tartalmat. A VOD-tartalmakat a MediaConvert konvertálja HLS-adatfolyamba illeszthető formátumba, a kimenetét pedig a S3-vödörbe helyezi. A serveralkalmazás egy Node.js-alapú web app a terveim alapján, amely NestJS keretrendszerrel kerül kialakításra, Dockerrel konténerizálom és az ECS-be telepítem, azzal menedzselem életciklusát; az ECR-be kerülnek a konténer képei. Az adatbázis egy PostgreSQL-példánnyal kerül megvalósításra, amelyet az RDS szolgáltatásban helyezek el menedzselésre.

Az orkesztrációs eseményekre Lambda-függvények reagálnak, az eseményeket központilag az EventBridge hallgatja le szabályokkal és kötteti össze a megfelelő Lambda-függvényekkel. Felhasználásra kerülnek a biztonságos kezelésre IAM-szerepkörök, monitorozásra és naplózásra a CloudWatch, illetve az érzékeny paraméterek tárolására az AWS Secrets Manager.

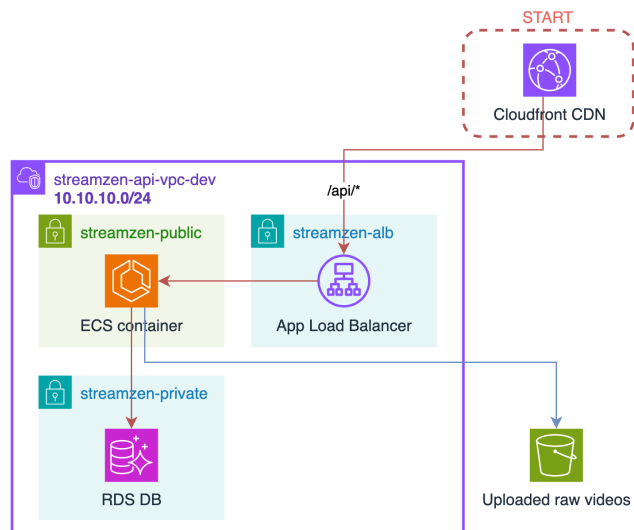
A kód bázis GitHubon kerül verziókezelésre, ott a CI/CD-folyamatokat GitHub Actions segítségével automatizálom a serveralkalmazás élesítésére, a React-alkalmazás statikus fájljainak feltöltésére. Terraformot terveztem bevezetni, amely segítségével az infrastruktúrát kód formájában kezelhetem.

5.2.1. Video-on-Demand kiszolgálás folyamata

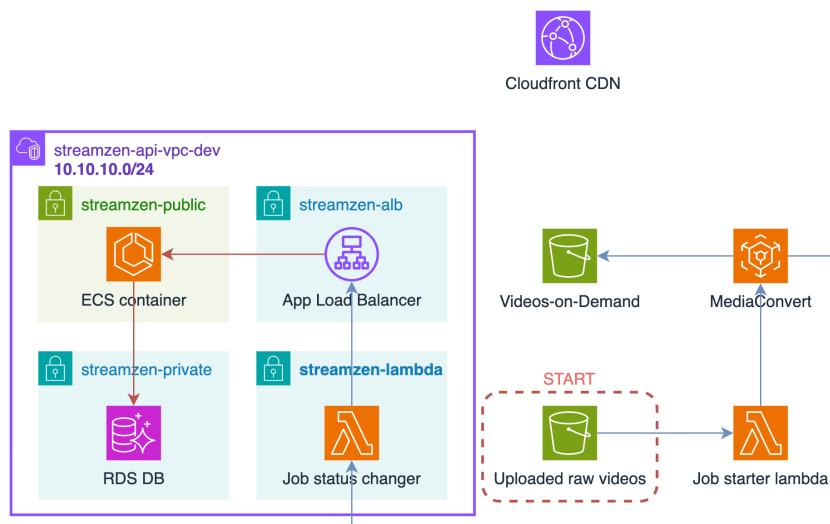
A VOD-ok kiszolgálásának folyamatát a következőképp terveztem meg. A 5.4. ábra mutatja be a VOD-tartalom feltöltésének folyamatát, ahol a serveralkalmazás a nyers videót buffer formájában fogadja HTTPS-en keresztül a böngészőből. A webszerver az S3-vödörbe helyezi, illetve lenyugtázza az adatbázisban, hogy a konvertálási folyamat ezzel elindult. A folyamatot a felhasználói felületen a felhasználók követhetik, ahol a folyamat állapotát mutatja a serveralkalmazás.

A 5.5. ábra mutatja be, hogy fut le a feltöltés utáni folyamat. Egy konvertálási jobot indító Lambda-függvény feliratkozik a feltöltött videókat tároló S3-vödörre, amely a feltöltés után felkonfigurál egy jobot a MediaConvert számára, megadja a forrásfájlt és az S3-vödört, ahova majd a job után kell kerüljön a HLS-kompatibilis fájlcsomag. Egy másik Lambda-függvény, amely a MediaConvert-job állapotváltozásaira van feliratkozva, a folyamat végén értesíti a webszervert az ALB-n keresztül, hogy nyugtázza a folyamat jelenlegi státuszát.

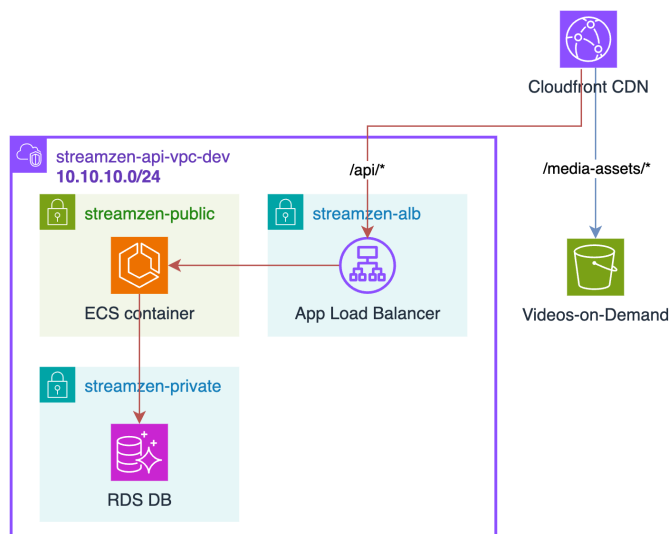
A 5.6. ábra fejti ki egy részről, hogy hogy értesül az adminisztrátor a videó feldolgozottságának állapotáról az API-n keresztül, illetve a UI-on értesülés után mi történik, ha meg is nyitja a már streamelhető videót, amely a CloudFront-disztribúció Video-on-Demand S3-alapú originjén keresztül érhető el.



5.4. ábra. Folyamatábra a nyers videó feltöltéséről.



5.5. ábra. Folyamatábra a feltöltés utáni videófeldolgozásról.

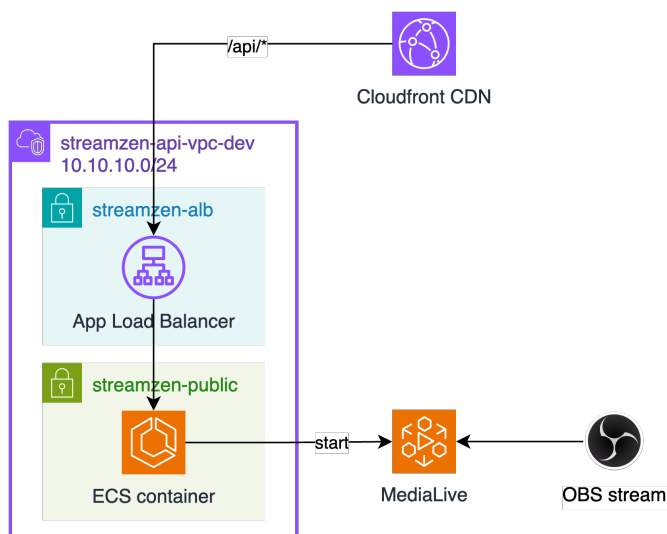


5.6. ábra. Folyamatábra a VOD-tartalom lejátszásáról.

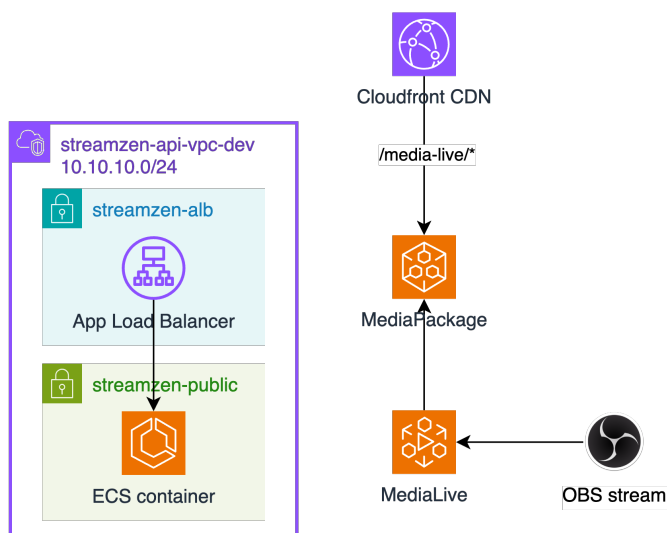
5.2.2. Live streaming folyamata

A live streaming indítását mutatja be a 5.7. ábra. Az adminisztrátor a stúdióban gombnyomásra megnyitja API-n keresztül a live streamet, amellyel a MediaLive szolgáltatásban a csatorna is elindul, aktívan húzza RTMP-n keresztül a felcsatlakoztatott forrásból a videóanyagot.

A 5.8. ábra pedig bemutatja, miután a live stream elindult, a MediaPackage-csatorna húzza át a konvertált videót a CloudFront-disztribúció elé, így ezen az originjén keresztül lesz elérhető a Cloudfront-disztribúciónak a felhasználók számára, akik a webalkalmazásban a megfelelő útvonalon érik el a live streamet.



5.7. ábra. Folyamatábra a live stream indításáról.



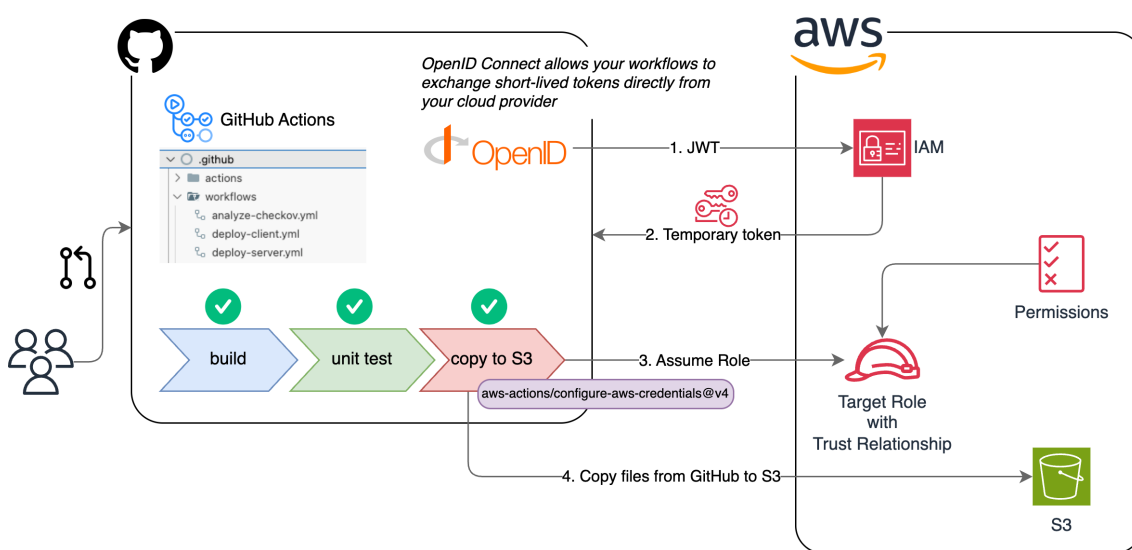
5.8. ábra. Folyamatábra a live streamre való kapcsolódásról.

5.3. Konfigurációmenedzsment

Nagyobb rendszerek tervezése igényli, hogy megfelelő konfigurációmenedzsmentet teremtsen köré a tervezőmérnök, hogy a rendszer könnyen karbantartható legyen. A Terraform lehetővé teszi az infrastruktúra építését és az annak felkonfigurálását kód formájában, a kódban való élesítések nyomán friss és dokumentált marad az állapota is ezeknek.

Egyetlen környezetet terveztem kialakítani, egy *development*, azaz fejlesztési környezetet, a Terraform modulokat egy befoglaló Terragrunt gyökérmodulba szerveztem, a streamzen-core mappába. Az erőforrásokat igyekeztem olyan módon elnevezni, hogy azok tartalmazzák a streamzen prefixet és a környezet nevét, például dev is tartalmazzák, hogy könnyen lehessen azonosítani őket a későbbiekben. Az erőforrások alapvetően az „eu-central-1” régióban helyezkedtem el, a globális erőforrások kivételével.

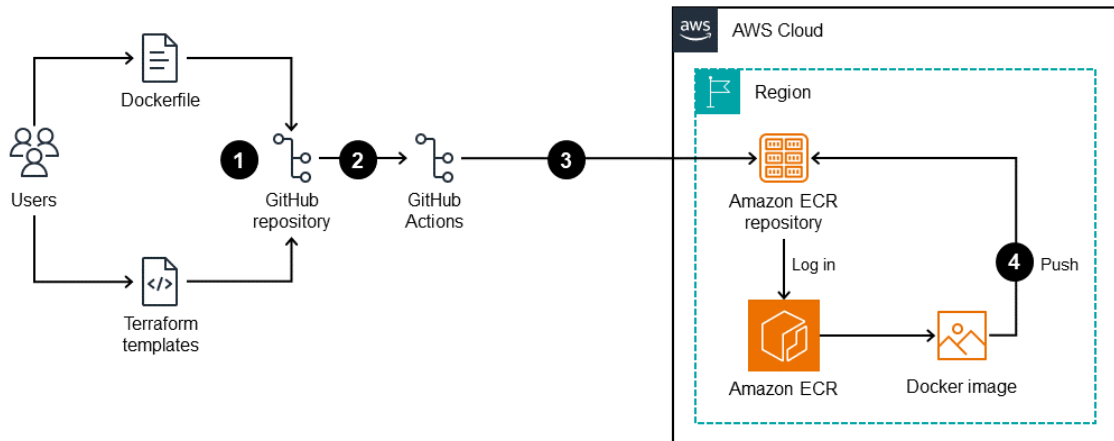
Terraformban menedzselt infrastruktúra tipikus életciklusa áll a kódból származó tervek előállításából (plan), annak manuális átolvasásából, majd pedig a változtatások aktiválásából (apply). Az automatizálás érdekében GitHub Action-munkafolyamatokba terveztem szervezni az élesítési tervek előnézetének generálását – amely a terraform plan parancs kiadásával kezdeményezhető –, ami minden Pull Request (PR) UI-ján kommentként kerül hozzáadásra a PR-hez, viszont a tervek élesítését (erre használt parancs a terraform apply) saját kézzel a saját parancssoromból terveztem megtenni, tekintettel arra, hogy csupán egyedül dolgoztam a kódbázissal.



5.9. ábra. Workflow autorizálása AWS-szerepkörre OIDC-val.

Hogy az AWS-fiók erőforrásaihoz hozzáférést kaphassanak az egyes munkafolyamatok, azok számára OpenID Connect (OIDC) felállításával terveztem az erre szánt AWS-szerepkör felvételét megvalósítani (5.9. ábra).[20]

A szerveralkalmazás egységként való kezelése érdekében és a könnyű telepíthetőségért – ahogy ezt a nem funkcionális követelmények is megkívánták – konténerizálni terveztem a Node.js-szerveralkalmazást Docker-konténerbe való komponálással, a buildelési folyamatot Dockerfile-lal kívántam megvalósítani hozzá. GitHub Action került alkalmazásra



5.10. ábra. Folyamatábra a Docker-kép ECR-be feltöltéséről.

az buildelt Docker-kép ECR-be való automatizált feltöltésére (5.10. ábra), a React-kód buildelésére és S3-vödörbe való feltöltésére.

5.4. A projekt felépítése

A projekt egészét egy közös GitHub-repository-ba terveztem kivitelezni, tehát „monorepo” jelleggel. A TypeScript-alapú projektekre bő eszköztárat és könnyű kezelést biztosít a Visual Studio Code (röviden VSCode²) szövegszerkesztő, amelyben a projekt könnyű átláthatóságára pedig egy VSCode-os (streamzen.code-workspace fájl névvel) munkatér-konfigurációt alakítottam ki 5 fő mappából álló struktúrával: client, server, infra, infra-bootstrap és .github alatt.

A client mappában a React-alprojekt található, a server mappában a szerveralkalmazás Node.js-alprojektje, az infra mappában a Terragrunt-konfiguráció, az infra-bootstrap mappában az egyszer aktiválandó Terraform-konfiguráció található (ez állította fel az S3-vödröt a Terraform-állapot tárolására és a CI/CD-csővezeték kapcsolódását OIDC-n keresztül az AWS-fiókba), a .github mappában pedig a GitHub Action-munkafolyamatok találhatók a csővezetékre.

²<https://code.visualstudio.com/>

6. fejezet

Kliensközeli komponensek implementációja

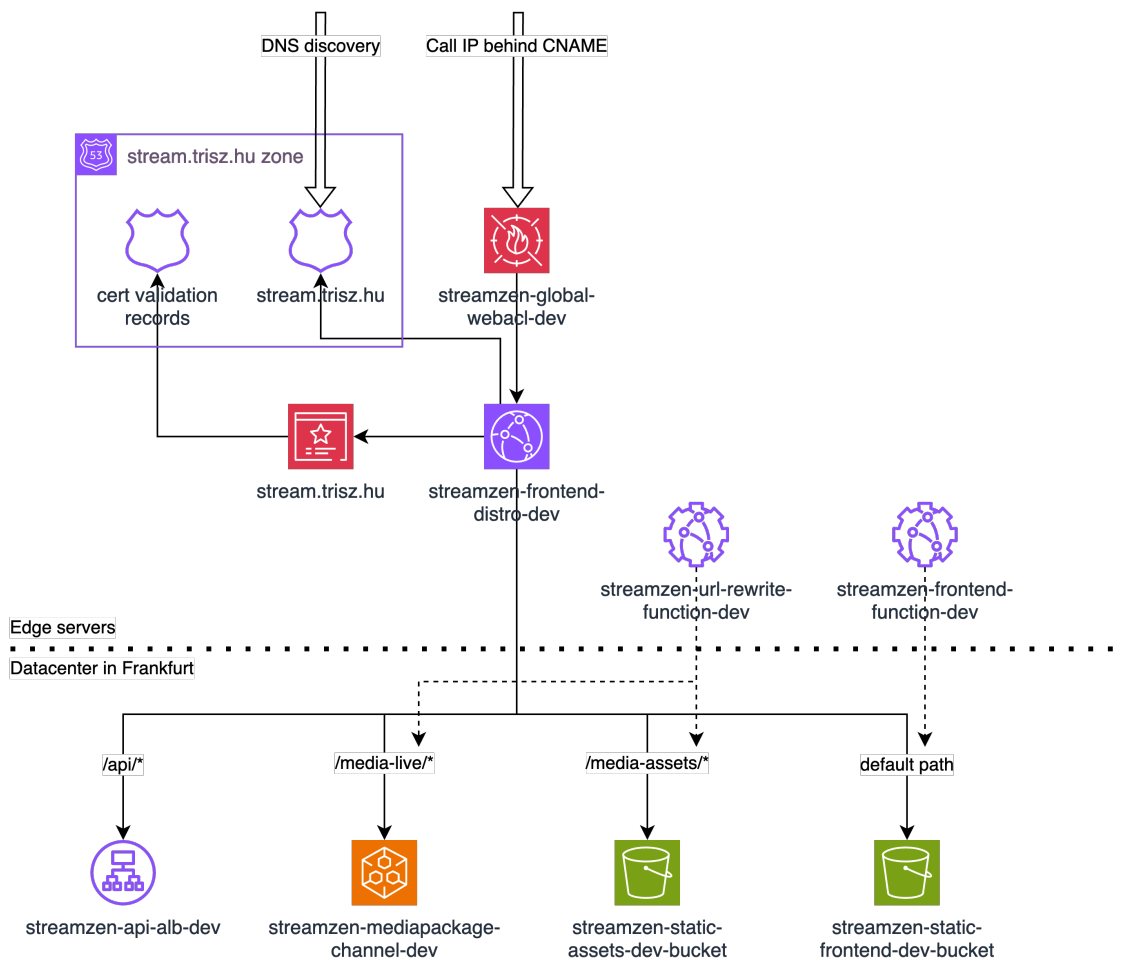
A következőkben részletezem a klienseket kiszolgáló infrastrukturális komponensek konfigurációját, valamint a legfelső megjelenítési réteg szoftveres komponenseinek implementációját. A forgalom először a CDN-nel ütközik, amelyen keresztül lesznek elérhetőek a statikus weboldal erőforrásai, valamint a média-erőforrások csatornái.

6.1. A CDN és a hozzácsatolt erőforrások

A CDN és az ahhoz tartozó erőforrások jelentik az első belépési pontját egy a rendszerhez intézett kérésnek. A rendszer és a CDN – azaz a CloudFront-disztribúció – számára a `stream.trisz.hu` doménnevet rendeltem, amelyet a saját `trisz.hu` doménem aldómenjeként jegyeztem be a Route 53 szolgáltatásban egy külön DNS zónaként. A böngészőből, azaz kívülről indított kérések minden esetben a DNS feloldásával kezdődnek, a Route 53 névszervereire oldódik fel, ezzel szerzi meg a kliens az IP-jét a CloudFront edge szerverfarmjának. A CloudFront-disztribúcióknak különleges CNAME rekordjaik vannak, biztosítják hogy a kliens a legközelebbi edge szerverfarmhoz csatlakozhasson, a konkrét működést az AWS elrejt a háztető alatt előlünk. A biztonságos, HTTPS-alapú server-kliens kommunikációt a CloudFront-disztribúcióhoz tartozó SSL-tanúsítvánnyal biztosítja a rendszer. A tanúsítványt az AWS Certificate Manager szolgáltatásban generáltam és kezeltem, amely automatikusan megújítja a lejáró tanúsítványokat, amennyiben a domainhez tartozó DNS-zónát a Route 53 szolgáltatásban – azaz az AWS szolgáltatásában kezeljük.

A CloudFront-disztribúcióhoz hozzácsatoltam egy WAF ACL-t, amely a webalkalmazás szintű tűzfal szerepét tölti be. Minden kérést ez a Layer 7 rétegbeli logika szűri meg. Mivel nem volt élő forgalomra készítve az alkalmazás, így egy egyszerű AWS menedzselte szabályt tettem csak rá a WAF-ra demonstrációképp, amely megvizsgálja a kérést, hogy tartalmaz-e SQL injection támadást vagy egyéb megszkott webalkalmazásokra jellemző ilyen jellegű támadást (pl.: ismert kihasználható URI-útvonalak, Java webalkalmazásokra jellemző exploitok) – a szabálycsoportot név szerint `AWSManagedRulesKnownBadInputsRuleSet` név alatt tartja számon az AWS WAF.

Az edge-en kerül kiértékelésre a kapott kérés útvonala (angolul *path*) alapján az, hogy melyik origin felé kell továbbítsa a disztribúció a kérést.



6.1. ábra. A kliensoldali architektúra részletesebben.

A 6.1. ábra adatközpont (angolul *datacenter*) felőli oldalán láthatóak a nyilak végén, hogy milyen útvonalak alapján kerül a kérés melyik originhez. A kiértékelés során a disztribúció figyelembe veszi a szabályok sorrendjét, az szabályokban definiált útvonalmintákra mintaillesztés történik, és ha kapott útvonal illeszkedik a sorban következő mintára, ott véget és a kiértékelés. A disztribúcióhoz tartozó originok közül a legfontosabb a statikus weboldal, amely az S3-vödörben található, ez lett az alapértelmezett útvonal, ahova a kérések mennek, amennyiben egyik előbbi útvonalmintára se illeszkedik a kérésben található útvonal.

A CloudFront biztosítja, hogy kis számításigényű logikát tudjuk még az edge rétegében futtatni a kéréseken, erre CloudFront Function-függvényeket alkalmaztam a MediaPlayer-csatorna előtt és a VOD-okat kiszolgáló S3-vödör (static-assets) előtt. Ez az url-rewrite kis függvény egy `url-rewrite.js` nevű fájlt futtat (6.1. kódrészlet). A függvény a kérés URI-ját vizsgálja, és ha a kérés a `/media-assets/` vagy `/media-live/` előtaggal kezdődik, akkor eltávolítja ezeket az előtagokat, és a kérés így megtisztított URI-ját továbbítja a konkrét origin felé.


```

1  function handler(event) {
2      const request = event.request;
3      ["/media-assets/", "/media-live/"].forEach((prefix) => {
4          request.uri = request.uri.replace(prefix, "/");
5      });
6      return request;
7  }

```

6.1. kódrészlet. url-rewrite.js fájl tartalma.

A backend /api előtagú útvonalakon keresztül elérhető, a CloudFront erre egy VPC Origin típusú originként csatlakozik rá, azon keresztül továbbítja a kérést, ezzel leegyszerűsítve a kérések hálózati biztonsági kezelését. Ennek az originnek a cache behaviorje nem igényli, hogy hagyjuk a Host fejléctet, ugyanis az ALB mögötte akár fel tudja használni a forgalomirányításhoz. A többi origin esetén a Host fejléctet le kell hagyni a kérésekről (működésük ezt igényli az AWS-dokumentáció alapján dolgozva), ezért is került használatra ezeken az útvonalakon a Managed-AllViewerExceptHostHeader nevű Origin Request Policy, azaz ez a kéréseket minden fejléccel együtt továbbítja, kivéve a Host fejléctet. A behaviorök mindegyikénél beállítottam, hogyha HTTP-val jönne a kérés, akkor dobja vissza a kliens felé, hogy kezdje újra HTTPS-en a kérést. A cachelési szabályzásokat nem volt célom túlbonyolítani, így egyszerű menedzselte optimalizált cache beállítást használtam, amely a legjobban optimalizálja a cachelési időt. Az API-n kívül a többinél be kellett állítsam, hogy az alap CORS-fejléceket (Managed-CORSHeaders) visszaküldje a kliens felé a disztribúció, hogy a böngésző ne blokkolja a válaszokat. Az API esetében maga a NestJS-alkalmazás kezeli ezt. Ezen beállításokat mutatja be a 6.2. ábra is.

EI2GXKTZOVAL8

General Security Origins Behaviors Error pages Invalidation Tags Logging							
Behaviors							
<div> <input type="button" value="Save"/> <input type="button" value="Move up"/> <input type="button" value="Move down"/> <input type="button" value="Edit"/> <input type="button" value="Delete"/> <input type="button" value="Create behavior"/> </div> <div> <input type="text" value="Filter behaviors by property or value"/> </div>							
	Preced...	Path pattern	Origin or ori...	Viewer protocol ...	Cache pol...	Origin request p...	Response ...
<input type="radio"/>	0	/api/*	vpc-origin	Redirect HTTP to ...	Managed-Cachi	Managed-AllViewer	-
<input type="radio"/>	1	/media-assets/*	assets-origin	Redirect HTTP to ...	Managed-Cachi	Managed-AllViewerExc	Managed-CORS
<input type="radio"/>	2	/media-live/*	live-origin	Redirect HTTP to ...	Managed-Cachi	Managed-AllViewerExc	Managed-CORS
<input type="radio"/>	3	Default (*)	frontend-origin	Redirect HTTP to ...	Managed-Cachi	Managed-AllViewerExc	Managed-CORS

6.2. ábra. Képernyőkép a különböző útvonalak cache szabályairól.

A VPC Origin típusú origin a CloudFront-disztribúcióban egy Amazon VPC-n belüli erőforrást jelent – a mi esetünkben ez az ALB-példányunk –, amelyet a CloudFront közvetlenül elérhet. Nem működik *cross-account* módon, tehát amennyiben a célpont egy

másik AWS-fiókban helyezkedne el. Az origin a VPC-n belül található, és elsősorban privát alhálózaton, Security Groupokkal van védve hálózati szinten. A VPC Origin típusú origin használata lehetővé teszi, hogy a CloudFront közvetlenül kommunikáljon az erőforrással, anélkül hogy nyilvános doménnevet kelljen rendelni hozzá. Ezzel megspórolhatjuk az ALB-példány Internet Gateway-re való kötését, az ALB-példány számára domén bejegyzését, valamint SSL-tanúsítvány felvételét annak, a TLS-kapcsolat is terminálhat a CloudFront-disztribúcióban, házon belül már elég HTTP-alapon forgalmazni komponensek között.

Block public access (bucket settings)

Edit

Public access is granted to buckets and objects through access control lists (ACLs), bucket policies, access point policies, or all. In order to ensure that public access to all your S3 buckets and objects is blocked, turn on Block all public access. These settings apply only to this bucket and its access points. AWS recommends that you turn on Block all public access, but before applying any of these settings, ensure that your applications will work correctly without public access. If you require some level of public access to your buckets or objects within, you can customize the individual settings below to suit your specific storage use cases. [Learn more](#)

Block all public access

On

► Individual Block Public Access settings for this bucket

Bucket policy

EditDelete

The bucket policy, written in JSON, provides access to the objects stored in the bucket. Bucket policies don't apply to objects owned by other accounts. [Learn more](#)

Public access is blocked because Block Public Access settings are turned on for this bucket

To determine which settings are turned on, check your Block Public Access settings for this bucket. Learn more about [using Amazon S3 Block Public Access](#)

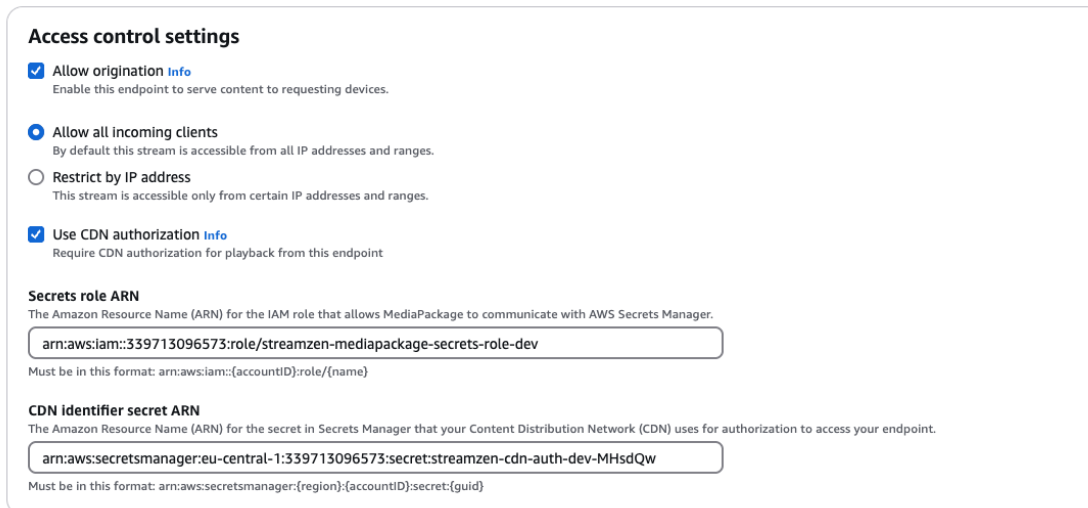
```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowCloudFrontServicePrincipal",
      "Effect": "Allow",
      "Principal": {
        "Service": "cloudfront.amazonaws.com"
      },
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::streamzen-static-assets-dev-bucket/*",
      "Condition": {
        "StringEquals": {
          "AWS:SourceArn": "arn:aws:cloudfront::339713096573:distribution/EI2GXKTZOVAL8"
        }
      }
    }
  ]
}
```

Copy

6.3. ábra. Képernyőkép a vödrök hozzáférési beállításairól.

Az S3-vödrök publikus elérését teljes blokkolásra állítottam (6.3. ábra). Úgy tettem őket elérhetővé a CloudFront-disztribúció számára, hogy egy-egy „bucket policy”-t csatoltam hozzájuk[14], amely az S3 originre egyedi, arra csatolt Origin Access Controllal (OAC)[17] együtt lehetővé teszi, hogy csupán az az AWS-beli principal – azaz a mi disztribúciónk – kapjon az objektumolvasásokra (és csak arra) hozzáférést, amelynek a megadott

Amazon Resource Number (ARN) azonosítója van. Az ábrán megadott `AWS:SourceArn` a `streamzen-frontend-distro-dev` disztribúció ARN-je. A fent gyakorolt beállítás is biztosítja az IT-biztonságban is elterjedt és az AWS által is ösztönzött *"principle of least privilege (PoLP)"* elvét.



The screenshot shows the 'Access control settings' panel in the AWS MediaPackage console. It contains three sections: 'Access control settings', 'Secrets role ARN', and 'CDN identifier secret ARN'. In the first section, 'Allow origination' and 'Use CDN authorization' are checked, while 'Allow all incoming clients' is selected with a radio button. The second section shows the 'Secrets role ARN' as `arn:aws:iam::339713096573:role/streamzen-mediapackage-secrets-role-dev`. The third section shows the 'CDN identifier secret ARN' as `arn:aws:secretsmanager:eu-central-1:339713096573:secret:streamzen-cdn-auth-dev-MHsdQw`.

Access control settings

- ☒ **Allow origination** Info
Enable this endpoint to serve content to requesting devices.
- ☒ **Allow all incoming clients**
By default this stream is accessible from all IP addresses and ranges.
- ☐ **Restrict by IP address**
This stream is accessible only from certain IP addresses and ranges.
- ☒ **Use CDN authorization** Info
Require CDN authorization for playback from this endpoint.

Secrets role ARN
The Amazon Resource Name (ARN) for the IAM role that allows MediaPackage to communicate with AWS Secrets Manager.

`arn:aws:iam::339713096573:role/streamzen-mediapackage-secrets-role-dev`

Must be in this format: `arn:aws:iam::{accountID}:role/{name}`

CDN identifier secret ARN
The Amazon Resource Name (ARN) for the secret in Secrets Manager that your Content Distribution Network (CDN) uses for authorization to access your endpoint.

`arn:aws:secretsmanager:eu-central-1:339713096573:secret:streamzen-cdn-auth-dev-MHsdQw`

Must be in this format: `arn:aws:secretsmanager:{region}:{accountID}:secret:{guid}`

6.4. ábra. Képernyőkép a HLS-végpont hozzáférési beállításairól.

A MediaPackage-csatorna védettségét pedig a publikált HLS-végpontra beállított *CDN Authorization*[1] segítségével biztosítottam, amely a CloudFront-disztribúciótól érkező kérésekben a `X-MediaPackage-CDNIdentifier` fejlécbe várja el egy titkos kulcs-érték párból az értéket. Ezt a fejléct a disztribúció felkonfigurálásakor a megfelelő originra rá tettem. A MediaPackage-csatorna HLS-végpontjának hozzáférési beállításait, a felhasznált Secret Managerből származó kulcs-érték pár ARN-jét, és az azt elérő IAM-szerep ARN-jét a 6.4. ábra mutatja be.

6.2. A statikus weboldal

A statikus weboldal HTML-, JavaScript- és CSS-fájlokból, valamint a weboldal statikus tartalmát képező médiafájlokból (képek, betűtípusok) tevődik össze. Ezek összeállításához a React keretrendszerben írt SPA-alkalmazásokat is jól kezelő Vite.js eszközt használtam, amely a React-alkalmazásokat egyetlen kiindulási `index.html` HTML-fájlba csomagolja, és a készülő JavaScript-kódot is optimalizálja. A Vite.js a fejlesztési környezetben gyorsítótárazza a fájlokat, így gyorsabbá téve a fejlesztést, míg a gyártási környezetben (angolul *in production*) optimalizálja azokat, hogy a lehető legkisebb méretűek legyenek.

Korábban bemutatásra került két különböző CloudFront Function-függvény, amelyek az S3-vödrök elérése előtt futnak minden kérésen. Ezek közül a statikus weboldal előtti függvény (6.2. kódrészlet) csupán arra hivatott, hogy a kéréseknél az egyes prefixekkel kezdődő kéréseket átirányítsa az alapértelmezett / útvonalra. Ezekre azért volt szükség, hogy a statikus oldal által kezelt útvonalak mind a CDN-re való belépés után az alapértelmezett útvonalon elérhető `index.html`-re oldódjanak fel, hiszen az indexoldalon behivatkozott Ja-

vaScriptből pedig majd a betöltés után a React Router megfelelően kezeli le a böngészőben az eredetileg kért útvonalat. Ez a megoldás természetesen magával vonzza azt az igényt, hogy akármikor ha új aloldalt vezetünk be, akkor a CloudFront Function-függvényben ezt a prefixet hozzá kell adnunk a `spaInternalRoutingPrefixes` tömbhöz, amely a kód elején található.

```
1  const spaInternalRoutingPrefixes = ["/videos", "/live", "/events", "/"
2    members", "/courses", "/about", "/studio", "/login"];
3  function handler(event) {
4    const request = event.request;
5    if (spaInternalRoutingPrefixes.some((pref) => request.uri.startsWith(
6      pref))) {
7      request.uri = "/";
8    }
9    return request;
10 }
```

6.2. kódrészlet. frontend-request-default.js fájl tartalma.

A Vite.js ökoszisztémájának részét képezik különböző fejlesztők által összerakott kód-generáló szkriptek, a Vite.js hivatalos dokumentációja által ajánlott szkriptek közül választottam a kezdőprojekt felállítására egy olyat, amely TypeScript nyelven írt és React keretrendszerre felkészítve rak össze egy kliensoldali NPM-projektet.

Ezek után telepítettem ebbe a projektbe a könnyebb fejlesztéshez szükséges könyvtárakat, ilyenek például a React Router, a React Hook Forms, az Axios és a Hls.js, utóbbi kettő jelentős szerepet fog még betölteni később implementációs kifejtéseimben. Ezek mellett UI-komponensek kódját húztam be a *shadcn/ui*¹ könyvtárából, amelyek a felhasználói felület megjelenítéséhez szükségesek.

A Vite.js képes fejlesztői módban indítani egy webszervert arra, hogy folyamatosan figyelje a fájlokat (úgynevezett *watch mode*-ban), és ha változás történik, akkor újrabuildelje a fájlokat, és újraindítsa a webszervert.

6.2.1. A weboldal telepítésének CI/CD-folyamata

Előkészítettem egy CI/CD-folyamatot `lint-client.yml` néven a `.github` mappa munkafolyamatai alatt, amely akkor fut le, ha a változtatások giten való feltöltése után készítünk egy Pull Requestet. Ez a munkafolyamat a következő lépéseket hajtja végre: statikus ellenőrzés ESLint² használatával, kódformattálás ellenőrzése Prettier³ használatával, valamint a webalkalmazás lebuildelese. A három lépés hibátlan lefutása jelzi azt, hogy a változtatások után a webalkalmazás telepíthető lesz az S3-vödörbe.

¹<https://ui.shadcn.com/>

²<https://eslint.org/>

³<https://prettier.io/>

Egy másik folyamat pedig a Pull Request elfogadása és a main branchbe való beolvasztása után indul el, amely a `deploy-client.yml` néven található. A 6.3. kódrészlet mutatja be a kódjának fontos részletét. Két job jellemzi a folyamatot, amelyből a második, a konkrét telepítés függ az előző lefutásától, valamint kézzel kell elindítani, amint készen áll (ezt az `environment: production` sor valósítja meg). Az elkészült teljes csomag a Node.js 20-as verziójával buildelődik, majd pedig az elkészült alkalmazás csomagja artifaktként kerül átadásra a következő jobnak. Letöltés után a telepítő szkript az AWS CLI segítségével a megadott S3-vödörbe tölti fel a fájlokat. A `aws s3 sync` parancs használatával a fájlok feltöltése előtt törli a vödörből azokat a fájlokat, amelyek már nem találhatók meg a buildelt fájlok között, így biztosítva azt, hogy mindig csak a legfrissebb fájlok kerüljenek ki a vödörbe.

A GitHub Actionból való AWS-hez való hozzáférést egy külön kompozit akció valósítja meg (`setup-aws` néven), amely az AWS által fenntartott hivatalos „Configure AWS Credentials” nevű GitHub Marketplace-en publikált akció⁴ v4-es verzióját használja. Ez az akció a megadott AWS IAM-szerepkörhöz tartozó hitelesítő adatokat állítja be a környezeti változóknak, amelyeket a következő lépésben használhatunk.

A szerepkör, amelyet a környezet felvesz a `github-oidc-pipeline` névre hallgat. Ezt a szerepkört az `infra-bootstrap` alprojektből telepítettem korábban az AWS-fiókba. A szerepkört úgy konfiguráltam be, hogy csupán a teljes projektet tároló `streamzen-monorepo` nevű GitHub-repository számára adjon jogosultságot, azaz csak az innen futó GitHub Actionök tudják felvenni a szerepkört. Ez a szerepkör az, amelyet 5.3 alfejezet is a tervekben bemutatott, amely OIDC nyomán kap hozzáférést.

⁴<https://github.com/marketplace/actions/configure-aws-credentials-action-for-github-actions>

```

1  build:
2    runs-on: ubuntu-latest
3    defaults:
4      run:
5        working-directory: client
6    steps:
7      - name: Checkout code
8        uses: actions/checkout@v4
9      - name: Setup Node.js
10       uses: actions/setup-node@v4
11       with:
12         node-version: 20.x
13      - name: Install dependencies
14        run: corepack enable && yarn install
15      - name: Build
16        run: yarn build
17      - name: Save bundle
18        uses: actions/upload-artifact@v4
19        with:
20          name: bundle
21          path: ./client/dist/
22  deploy:
23    runs-on: ubuntu-latest
24    needs: build
25    environment: production
26    steps:
27      - name: Download bundle
28        uses: actions/download-artifact@v4
29        with:
30          name: bundle
31          path: /tmp/bundle/dist/
32      - name: Setup AWS
33        uses: ../.github/actions/setup-aws
34      - name: Deploy
35        run: aws s3 sync --delete /tmp/bundle/dist s3://streamzen-static-frontend-dev-bucket

```

6.3. kódrészlet. Részlet a deploy-client.yml fájl tartalmából.

7. fejezet

Rétegeken átívelő szolgáltatások implementációja

Ebben a fejezetben kerül kifejtésre az implementációja azon szolgáltatásoknak az alkalmazásban, amelyek a kliens és szerver rétegén átívelnek, így külön-külön nem lenne érdemes bemutatni őket.

7.1. Single Sign-On (SSO) integrációja

A videók feltöltésére azok jogosultak, akik a stúdióba be tudnak lépni a `/studio` aloldalon, a bejelentkeztetéshez pedig a kari hallgatói közösségünk által nyílt forráskóddal fejlesztett AuthSCH nevű SSO-rendszert integráltam be a weboldalba, ennek az esszenciális tokenkezelési implementációs részeit a backendre bízam, a kliensoldal csupán a megfelelő útvonalakra irányításért felel.

A 7.1. kódrészlet mutatja be a React-alkalmazásban használt AuthCtx nevű kontextust, amely JSON Web Tokeneket (JWT)[10] használ a bejelentkeztetés során a felhasználói adatok biztonságos és állapotmentes átpasszolására. A useMe nevű hook a backend `/api/auth/me` végpontjáról kéri le a bejelentkezett felhasználó adatait, a végpont mögötti logika csupán annyiból áll, hogy kibontja a kódolt JWT-t és visszaadja abból a releváns profiladatokat. Az AuthProvider nevű React-függvénykomponens biztosítja a kontextust az alkalmazás többi részének, a useAuth hookon keresztül lehet indítani a többi komponensekben bejelentkezést és kijelentkezést (login és logout függvények), lehet megtudni, van-e bejelentkezett felhasználó a rendszerben (authenticated boolean változó).

```

1  import { createContext, PropsWithChildren } from "react"
2  import { useMe } from "../use-me.hook"
3
4  type AuthCtxType = {
5    authenticated: boolean
6    isLoading: boolean
7    login: () => void
8    logout: () => void
9  }
10 export const AuthCtx = createContext<AuthCtxType | undefined>(undefined)
11
12 export function AuthProvider({ children }: PropsWithChildren) {
13   const { data, isLoading } = useMe()
14   const onLogin = async () => {
15     window.location.href = import.meta.env.VITE_BACKEND_URL+"/auth/login"
16   }
17   const onLogout = async () => {
18     window.location.href = import.meta.env.VITE_BACKEND_URL+"/auth/logout"
19   }
20   const value = {
21     authenticated: !!data,
22     isLoading,
23     login: onLogin,
24     logout: onLogout,
25   }
26   return <AuthCtx.Provider value={value}>{children}</AuthCtx.Provider>
27 }
28 export function useAuth() {
29   const context = useContext(AuthContext)
30   if (context === undefined) {
31     throw new Error("useAuth must be used within an AuthProvider")
32   }
33   return context
34 }

```

7.1. kódrészlet. auth-context.tsx fájl tartalma.

A szerver oldalán egy AuthModule névre hallgató NestJS-modul került létrehozásra, amely a AuthController és AuthService osztályokat tartalmazza. Az AuthController osztályban kerülnek definiálásra a forgalmat lehallgató HTTP-végpontok/függvények, míg az AuthService osztály elkülöníti a konkrét üzleti logikát. A 7.2. kódrészlet bemutat két fontos HTTP-végpontot, a /api/auth/login és a /api/auth/callback útvonalakon GET metódusra hallgatóakat sorrendben.


```

1  @UseGuards(AuthSchGuard)
2  @Get("login")
3  @ApiResponse({
4    description: "Redirects to the AuthSch login page.",
5  })
6  login() {}
7
8  @Get("callback")
9  @UseGuards(AuthSchGuard)
10 @ApiResponse({
11   description: "Redirects to the frontend and sets cookie with JWT.",
12 })
13 @ApiQuery({ name: "code", required: true })
14 oauthRedirect(@CurrentUser() user: UserDto, @Res() res: Response): void {
15   const jwt = this.authService.login(user)
16   res.cookie("jwt", jwt, {
17     httpOnly: true,
18     secure: true,
19     domain: process.env.NODE_ENV === "production" ? getHostFromUrl(process
20       .env.FRONTEND_CALLBACK) : undefined,
21     maxAge: 1000 * 60 * 60 * 24 * 7, // 7 days
22   })
23   res.redirect(302, process.env.FRONTEND_CALLBACK + "?authenticated=true")
24 }

```

7.2. kódrészlet. Az AuthController osztály fontos függvényei.

A AuthSchGuard egy NestJS-dekorátor, amely fel kell kerüljön a oauthRedirect és login függvényekre. E mögött a dekorátor mögött egy Passport.js-stratégia[18] van, amely megvalósítja az AuthSCH-val való OAuth-alapú beazonosítási logikát az OAuth-token megújítását, ehhez kell beállítja a kapott OAuth-kliens azonosítóját és titkos kulcsát, ezen értékek forrásáról később a 8.2.2. alfejezet beszél.

A login függvényt maga az AuthSchGuard dekorátor kiváltja, a függvény törzsébe nem kell semmit se írni, a mögöttes logika a felhasználót átirányítja az AuthSCH bejelentkezési oldalára. Az oauthRedirect függvény a sikeres bejelentkezés után visszairányítja a felhasználót a frontendre, a kérésben a jwt nevű HTTP-only biztonságos sütit beállítja, amely tartalmazza a bejelentkezett felhasználóra jellemző JWT-t 7 napos lejáratú idővel. A @CurrentUser() dekorátor a bejelentkezett felhasználó adatait injektálja a megadott user paraméterbe (ezt az AuthSchGuard oldja meg a háttérben, kiolvassa a kapott autorizációs kódot, amit az AuthSCH-től kapott). Az AuthService osztály login függvénye a bejelentkezett AuthSCH-s felhasználó adatait kapja meg, amelyből a JWT-t generálja, ezt a 7.3. kódrészlet mutatja be. A createOrUpdateUser függvény a Prisma ORM segítségével megkeresi a felhasználót az adatbázisban, ha nem találja, akkor létrehozza azt, ha megtalálja, akkor frissíti a felhasználó adatait, ez a függvény kerül felhasználásra az

AuthSchGuard mögötti stratégiában is a bejelentkeztetés validációs „mellékhatásaként”, hogy a saját adatbázisunkban is létrejöjjön a felhasználó.

```
1 login(user: object): string {
2   return this.jwtService.sign(user, {
3     secret: this.configService.get<string>("JWT_SECRET"),
4     expiresIn: "7 days",
5   })
6 }
7 async createOrUpdateUser(prof: AuthSchProfile): Promise<UserDto> {
8   const gravatarUrl = this.getGravatarUrl(prof.email, 200)
9   return this.prisma.user.upsert({
10     where: { authSchId: prof.authSchId },
11     update: {
12       fullName: prof.fullName,
13       firstName: prof.firstName,
14       email: prof.email,
15       imageUrl: gravatarUrl,
16     },
17     create: {
18       authSchId: prof.authSchId,
19       fullName: prof.fullName,
20       firstName: prof.firstName,
21       email: prof.email,
22       imageUrl: gravatarUrl,
23     },
24   })
25 }
```

7.3. kódrészlet. Az AuthService osztály fontos függvényei.

Lefejlesztésre került egy másik *guard* típusú NestJS-dekorátor, ez a JwtGuard, amelyet minden más a szerveren található végpontra be kell vezetni, amennyiben az bejelentkezett felhasználót igényel, ez a dekorátor ellenőrzi a jwt süti jelenlétét, a felhasználót beazonosítja. Az ilyen függvényekben is ha a paraméterre egy @CurrentUser() dekorátort helyezünk, akkor a felhasználó adatai automatikusan be lesznek injektálva abba a paraméterbe.

8. fejezet

Szerveroldali folyamatok implementációi

A megjelenítési réteg alatt található szerveroldali folyamatok implementációja során a kiszolgáló infrastruktúra kialakítására, a Node.js-alkalmazás fejlesztésére, a konténerizált környezet kialakítására, valamint a videófeldolgozásra fókuszálunk ebben a fejezetben.

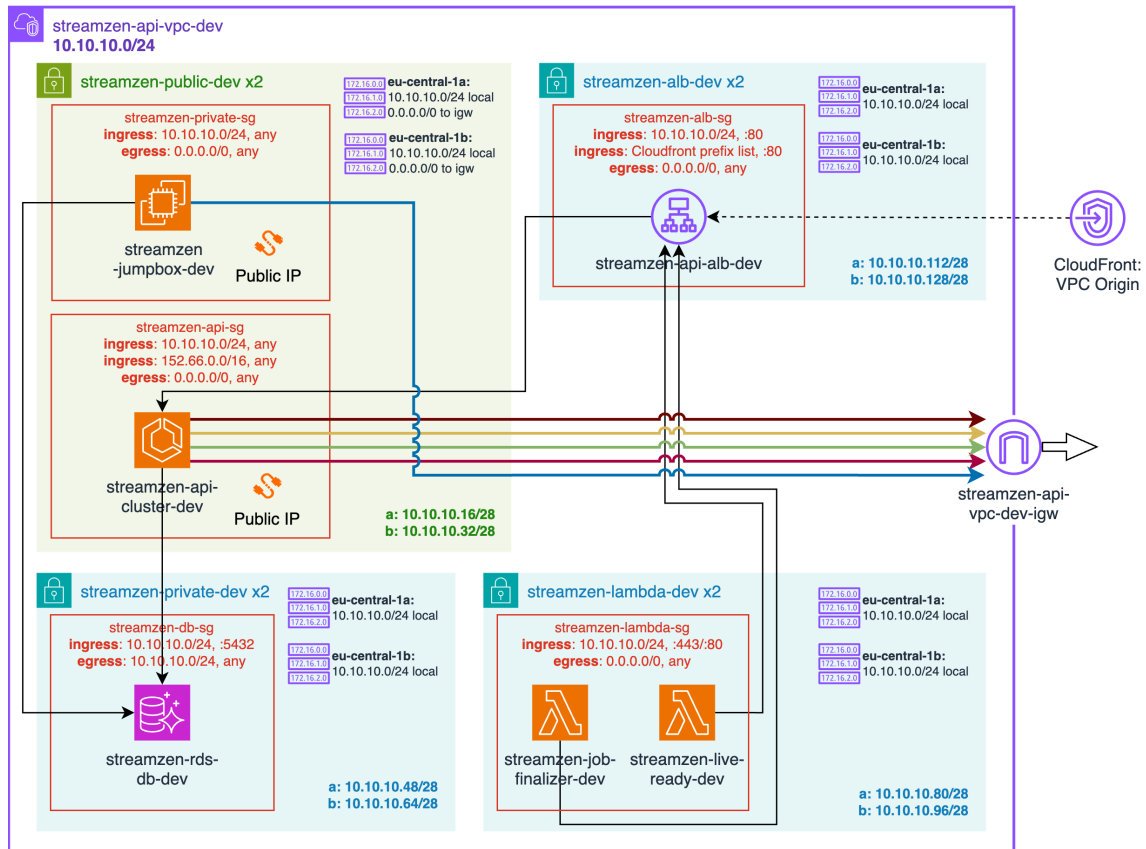
8.1. A virtuális privát felhő komponensei

A szerveroldal erőforrásait igyekeztem mindet egy közös VPC-be szervezni, amely lehetővé teszi, hogy a komponensek egymást lássák, és biztonságosan kommunikáljanak egymással alhálózataik között. A VPC-n belül az összetartozó elemekhez két-két alhálózatot hoztam létre. A legtöbb AWS-szolgáltatás a magas rendelkezésreállás érdekében legalább kettő konkrét adatközpontba kell kitelepítése kerüljön, azaz az AWS saját terminológiáját használva: két *Availability Zone*-ba (AZ) kell elhelyezésre kerüljenek a konkrét erőforrások. Ennek megfelelően az eu-central-1 régió alatti *eu-central-1a* és *eu-central-1b* AZ-ba szerveztem az egyes alhálózataim. Az alhálózatok közötti forgalom irányítását útválasztó táblák (angolul *route table*) segítségével végeztem, amelyek biztosítják, hogy a komponensek közötti kommunikáció megfelelően működjön. Az útválasztó táblák konkrét bejegyzéseit, valamint a felhasznált IP-tartományokat is tartalmazza a 8.1. ábra.

Az ábráról leolvashatóak még a Security Groupok, azaz virtuális állapotmentes tűzfalak, ezek a komponensek vörös keretként látszódnak az ábrán és *-sg* végződésűek neveik. Ezek a tűzfalak úgy kerültek kialakításra, hogy a csupán a ténylegesen szükséges IP-tartományokat és portokat engedélyezzék a kommunikációhoz.

A legelső belépési pontja egy kérésnek az ALB-példány, amely privát alhálózatra lett bekötve. Privátnak minősül egy alhálózat, amennyiben nincs közvetlen internetkapcsolata, nem kerül bekötésre az útválasztó táblájába IGW.

Habár bevezetésre került egy Internet Gateway (IGW), fontos megjegyezni, hogy nem azért, hogy a kliensoldali erőforrás – a CloudFront-disztribúció – elérhesse a szerveroldalt, hiszen azt megvalósítja egy VPC Origenen keresztül, amely működési lényege, hogy számára nem szükségeltetik bevezetni publikus alhálózatot, a disztribúció közvet-



8.1. ábra. Részletes architektúraábra a VPC-ről.

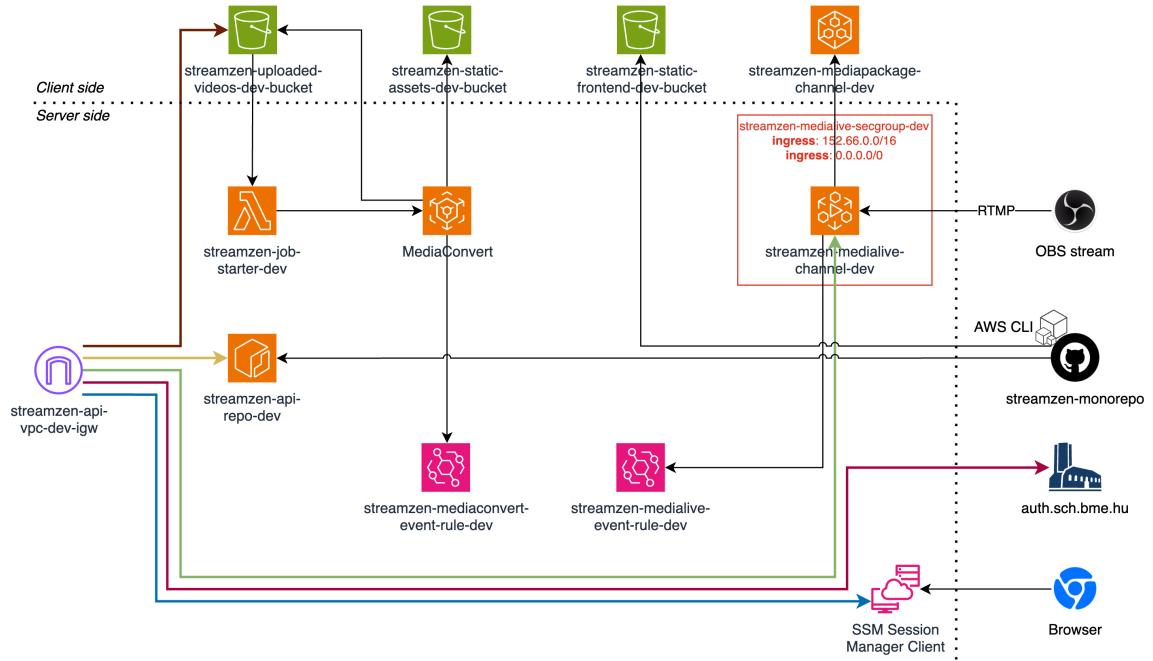
len összeköttetést tud összehozni VPC-n belül elrejtett hálózati erőforrásokkal, azaz a mi ALB-példányunkkal is. Az IGW bevezetésének indokait a következő bekezdések tisztázzák.

Az ECS-klaszter, amely a Node.js-alkalmazást futtatja és összeköttetésre kerül az ALB-vel, viszont már publikus alhálózatban helyezkedik el. A Security Groupja úgy lett beállítva, hogy a VPC-n belüli eszközöket engedje be, illetve azt az IP-tartományt, amelyben az AuthSCH is fut (ez a Schönherz Kollégium hálózati tartománya, a 152.66.0.0/16).

A szervernek fontos a kapcsolódása az adatbázisra, amely egy külön privát alhálózatot kapott, abba a Security Group csupán a PostgreSQL-re jellemző 5432 porton keresztül enged és csak a VPC-belülről forgalmat, hasonlóképp kifelé is csak a VPC-n belülre enged.

Az ECS-klaszterben működő szervernek több külhálózaton működő szolgáltatás felé kell tudnia kommunikálni. Ezek a VPC-n kívüli komponensek a 8.2. ábra segítségével kerülnek vizuálisan ismertetésre.

A 8.2. ábra ábra és a 8.1. ábra együttesen mutatja be az Internet Gateway-en keresztüli kommunikációs útvonalakat, azonos színű vonalak egyazon kommunikációs útvonalat jelölik. Ennek megfelelően az előző bekezdésben jelölt AuthSCH-t elérő kommunikációs útvonalat jelöli a piros vonal. A sárga vonal jelöli azt, ahogy az ECS-klaszter eléri az ECR-repository-t a Docker-kép letöltésére. A zöld vonal jelöli a szerver és a MediaLive-csatorna közti utat, amelyen keresztül a szerver elindítja az vételt a csatornán. Végül pedig a barna vonal jelöli a videófeltöltés folyamatát az S3-vödörbe a szerveren keresztül.



8.2. ábra. Részletes architektúraábra a VPC-ből kifelé és befelé kommunikáló komponensekkel.

Léteznek különféle hálózati erőforrástípusok, hogy privát alhálózatban helyezzünk el egy ECS-ben futó szervert, illetve akár Lambda-függvényeket, amelyek kommunikálnak VPC-n kívüli eszközökkel. Ilyen erőforrástípus a *NAT Gateway*, amely Network Address Translation (NAT) szolgáltatás, és amennyiben adunk neki egy állandó publikus IP-címet (Amazon Elastic IP szolgáltatással), úgy képes azon keresztül az internet felé forgalmazást biztosítani a VPC erőforrásai számára, kinti erőforrásoknak viszont befelé már nem. Ez megoldotta volna az AuthSCH felé kommunikálást. Egy másik lehetőség lett volna a *VPC Endpointok* alkalmazása, amelyek interfészt szolgáltatnak bizonyos AWS-en belüli szolgáltatások felé anélkül, hogy az elhagyná az adatközpontot. A bizonyos szolgáltatások közé tartozik a CloudWatch Logs, az ECR, az ECS-hez tartozó egyéb szolgáltatások, az EventBridge és az S3 is. Ezt a biztonsági kockázatot a kísérletem szempontjából nem kívántam megelőzni, az igazán szükséges védelmi réteget a Security Group is megvalósítja, a privát alhálózat csupán plusz egy réteget jelent nagy kockázatú, vegyes forgalmat kezelő enterprise rendszerek kivitelezése során, az én esetemben a költségek és a bonyolultság erősen megnőtt volna jelölt erőforrások bekötésével.

A hálózat építése tégláról téglára került kivitelezésre, a fejlesztési/tervezési fázisban ezért érdemesnek tartottam a tesztelés során „*jumpbox*” jelleggel bevezetni egy EC2-példányt, amely a publikus alhálózatban helyezkedik el. Viszont az elérése nem kívántam jelszavakat, SSH-kulcsokat kezelni, ezért az AWS Systems Manager (SSM) Session Manager szolgáltatását használtam, amely lehetővé tette, hogy a webes konzolon keresztül SSH-kapcsolatot létesítsek az EC2-példánnyal anélkül, hogy közvetlenül elérné azt. Az EC2-es példány létrehozása után egy ágenset kellett volna telepítenem, amely erre felkészíti magát a virtuális gépet, azonban ezt a telepítési folyamatot is automatizáltam a Systems

Manager Host Management szolgáltatásával, amely lehetővé teszi az EC2-példányok központi kezelését. A jumpbox segítségével tudtam pingelgetni a hálózati eszközök interfészeit, illetve a Security Groupok beállításait is tesztelni, és akár az RDS-adatbázison is futtatni egyszerű lekéréseket, sémamigrációkat.

8.2. A Node.js-alkalmazás fejlesztése

A NestJS keretrendszerben írt Node.js-alkalmazások modulokra bomlanak, az egyes modulok *controller* rétegre és *service* rétegre válnak ketté, a javaslat, hogy az előbbi réteg osztályaiban a forgalomirányítást valósítsuk meg, a későbbiben pedig a konkrét szolgáltatásszintű üzleti logikát, a számításokat, adattisztítást, adatelérést valósítsuk meg. A modulok implementációjában az egyes *service* és *controller* típusú osztályok példányainak beinjektálását, illetve azok életciklusáról való gondoskodást a NestJS által biztosított Dependency Injection (DI) rendszer valósítja meg.

Az elkészült webszerver-alkalmazás egy REST API-t valósít meg. A benne kialakításra került modulok a következő funkciókat látják el:

1. AuthModule: A felhasználók azonosításáért és jogosultságkezeléséért felelős modul.
2. PrismaModule: Csupán behúzza a rendszerbe a projektbe telepített Prisma ORM-et mint service.
3. UsersModule: A felhasználók kezeléséért felelős modul, amely a felhasználók CRUD-műveleteit valósítja meg.
4. VideoModule: A videók kezeléséért felelős modul, amely a videók feltöltését és metaadatainak kezelését valósítja meg.

8.2.1. Az adatbázisséma

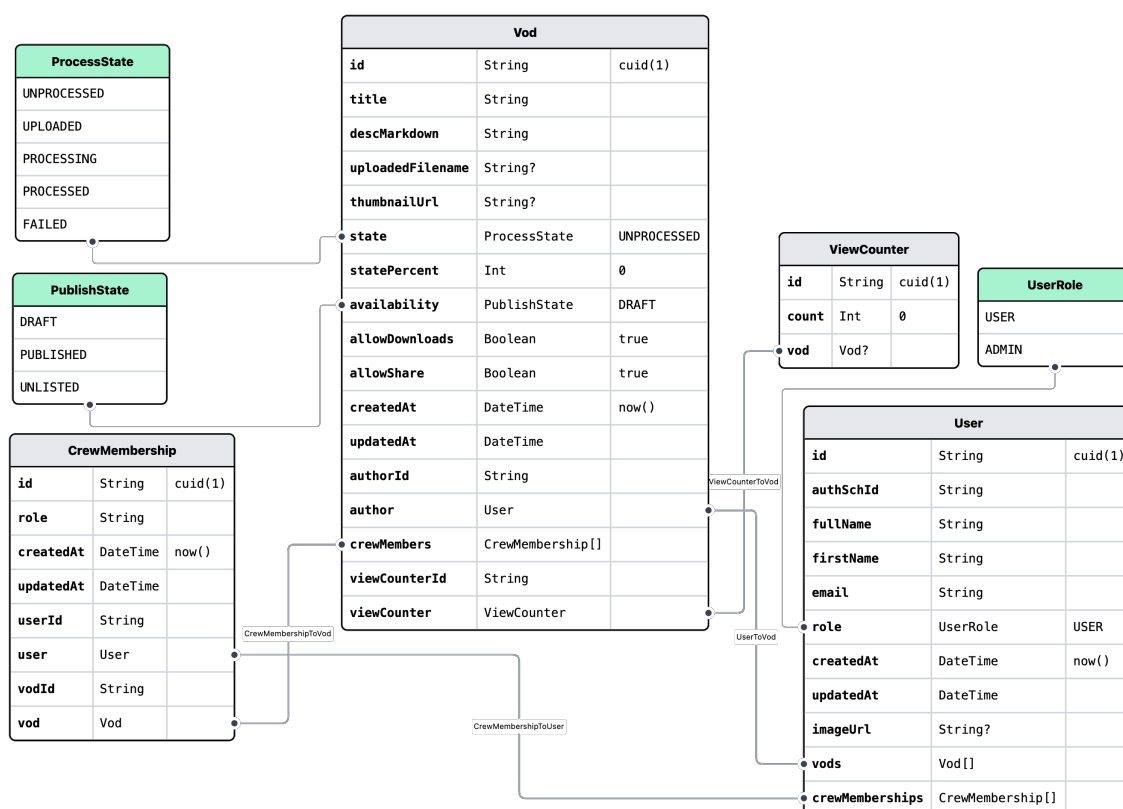
A 8.3. ábra mutatja be a Prisma-ban írt Prisma-sémát. A séma tartalmaz jövőbeli felhasználásra szánt elemeket, az egyszerűsége törekszik a séma, csupán a lényeges adatokat tároljuk le.

A User entitás reprezentálja a felhasználókat, akik megfelelő *role* attribútumérték esetén feltölteni tudnak a weboldalra. A *role* attribútum típusát enum típusra állítottam, amelynek értékei a USER és ADMIN. A User-nek számított attribútuma a *vods* és a *crewMemberships*.

A Vod entitás a feltöltött videók metaadatait reprezentálja. A Vod entitásnak meg kell adni létrehozáskor egy User kapcsolatot, amely a feltöltő felhasználót reprezentálja. Ezenkívül fontos attribútuma a *state*, amely a videó állapotát reprezentálja. A *state* attribútum enum típusú, lehet értéke UNPROCESSED (ez a létrehozáskori érték), lehet UPLOADED (amikor feltöltésre került), PROCESSING (amikor már a MediaConvert dolgozik az átalakításán), PROCESSED (amikor véget ért a MediaConvert-job), vagy pedig FAILED (amennyiben hiba történt valamelyik fázisban). Segít a videó állapotának nyomon követésében a

statePercent attribútum is, amely a MediaConvert által visszaadott százalékos értéket tárolja le.

A CrewMembership entitáshoz tartozó tábla a Vod és a User entitások közötti illesztő tábla (angolul *join table*), saját attribútumokat tud ehhez a kapcsolathoz hozzáadni, kardinalitás szempontjából egy CrewMembershipnek csak egy User-t és csak egy Vod-ot kell kötnie. Egy Vod-nak persze több CrewMembership-je is lehet, ugyanígy User-nek is. Ez az entitás és a ViewCounter – amely a megtekintések számlálóját lett volna hivatott reprezentálni – aktívan nem került felhasználásra a fejlesztés során, tervezésben maradt csupán, nem volt esszenciális a végső cél megvalósításához.



8.3. ábra. A Prisma-séma vizuális reprezentációja.

8.2.2. Környezeti változók

Az alkalmazásnak korábbi alfejezetekben már kiderült, hogy vannak külső függőségei. Ezen függőségekhez való hozzáféréshez a környezeti változók használata elengedhetetlen. A ECS-ben futó konténer környezeti változóinak beállítására az ECS is ad lehetőséget a 8.1. kódrészlet mutatja be a Terraform-kód egy részletét, amely átpasszolja az alkalmazás számára a következő fontos paramétereket.

Az AuthSCH SSO-ban felvett OAuth-kliensünk azonosítóját és titkos kulcsát a AUTHSCH_CLIENT_ID és AUTHSCH_CLIENT_SECRET környezeti változók tárolják, ezek értékeit ahogy a kódból is látható egy-egy SSM Parameter Store-ból szerzi meg a CI/CD-folyamat futtatása során a Terraform-kliensünk (értsd: a referált erőforrás elején data

kulcsszó jelöli, hogy *data source*-ként éri kéri le a Terraform az AWS CDK-n keresztül az értékét). Ezen Param Store kulcs-érték párokhoz az értéket az AuthSCH saját oldalán szereztem meg `auth.sch.bme.hu` oldalon található webes felületén, ahol korábban már megszerzett SCHAcc-fiókomba való belépés után új OAuth-kliens beregisztrálása után tudtam magamnak generálni a kívánt értékeket (ID és client secret).

Egy felhasználónevet és jelszót (POSTGRES_USER és POSTGRES_PASSWORD) is meg kell tudjuk adni környezeti változón keresztül az alkalmazásunknak, amellyel autentikál az adatbázisba, ezek értékét is SSM Param Store-okba tettem bele, viszont a felhasználónevet már magam találtam ki, a jelszót pedig magam generáltam kézzel.

Az alkalmazásnak tudnia kell bizonyos átirányítások lekezelésére a kliensünk bázis URL-jét, így azt megadtam környezeti változóban (FRONTEND_CALLBACK). A doménnevet változtathatónak véltem hagyni, ezért is került Terraform-változóba az értéke (`var.domain_name`).

Ezenkívül még szükség van egy titkos kulcsra, amely a JWT-tokenek aláírására szolgál (JWT_SECRET), ezt is SSM Param Store-ból szerzi be az alkalmazásunk, illetve ezt is már én töltöttem be, miután kézzel generáltam egyet.

A feltöltött videókat tároló S3-vödör nevét (AWS_S3_UPLOADED_BUCKET) az általam kitalált nevezéktani konvenciók alapján töltöttem be, illetve az S3-vödör régióját (AWS_S3_REGION) pedig természetesen arra a régióra állítom, ahová az erőforrások főképp telepítésre kerültek (és ahova az S3-vödör is került).

```
1 task_environment = {
2   AUTHSCH_CLIENT_ID = data.aws_ssm_parameter.these["authsch-client-id"].
      value
3   AUTHSCH_CLIENT_SECRET = data.aws_ssm_parameter.these["authsch-client-
      secret"].value
4   POSTGRES_USER = data.aws_ssm_parameter.these["db-username"].value
5   POSTGRES_PASSWORD = data.aws_ssm_parameter.these["db-password"].value
6   FRONTEND_CALLBACK = "https://${var.domain_name}"
7   JWT_SECRET = data.aws_ssm_parameter.these["api-jwt-secret"].value
8   AWS_S3_REGION = var.region
9   AWS_S3_UPLOADED_BUCKET = "streamzen-uploaded-videos-${var.environment}-
      bucket"
10 }
```

8.1. kódrészlet. Az ECS-szerveralkalmazás környezeti változóinak feltöltése a `main.tf` fájlban.

8.2.3. A videófeltöltés üzleti logikája

A szerveroldalon kialakításra kerülő konkrét üzleti logikai folyamatok közül a feltöltésre kerülő nyers videók S3-vödörbe való továbbítása az egyik legérdekesebb. A fentebb ismer-

tett `VideoController` és `VideoService` osztályokból emeli ki rendre a 8.2. kódrészlet és a 8.3 kódrészlet a fontosabb részeket.

```
1  @Post("/:id/upload")
2  @UseGuards(JwtGuard)
3  @UseInterceptors(FileInterceptor("file"))
4  async upload(
5    @Param("id") id: string,
6    @UploadedFile(new ParseFilePipe()) file: Express.Multer.File
7  ) {
8    const { originalname, buffer } = file
9    const res = await this.videoService.upload(id, originalname, buffer)
10   return await this.videoService.afterUpload(id, res.fileName)
11 }
```

8.2. kódrészlet. Videófeltöltés kezelése a `VideoController` osztályban.

A feltöltéshez szükséges fájlok a kliensoldalon kerülnek kiválasztásra, és a POST metódusú `/api/videos/:id/upload` végponton keresztül kerülnek feltöltésre. Ezt a végpontot hallgatja le a `VideoController` osztálybéli `upload` függvény is. A függvény dekorátorai közül a `@Post` jelöli azt, amiképp hallgat a kérésekre (POST metódussal és a jelölt útvonalon). A `@UseGuards` dekorátor az autentikáció szükségességét biztosítja a kérések végbemeneteléhez, amelyet a mi esetünkben a `JwtGuard` osztály valósít meg. A `@UseInterceptors` dekorátor pedig egy interceptort állít be a függvény elé, a fájlok feltöltését segíti elő, amelyet a `FileInterceptor` osztály valósít meg. Ez az osztály a NestJS alatt működő Express keretrendszerben működő *Multer* nevű Node.js-middleware-t hasznosítja, amellyel így tehát képes az osztály mögötti logika a HTTP-alapú bájtfolyamot hatékonyan feldolgozni/parseolni, és elrejtteni előlünk a sok-sok kihívását egy fájlfeltöltésnek. A függvény paramétereiben a `file` paraméter a `@UploadedFile` dekorátoron keresztül válik a feltöltött fájl reprezentációjává, típusa is a `File` interfész lesz.

A `VideoService`-ben működő `upload` függvény felé lesz a kapott fájl bufferje továbbítva, végül a service függvény eredményéből született fájlnevével kerül az adatbázisba is lementésre az entitásként a videó is a `afterUpload` függvény által.

A `VideoService` az AWS SDK S3 API-ját használja a fájlok vödörbe való feltöltésére. Ehhez létrehoztam tagváltozóként a `s3Client`-et. A `upload` függvény először átalakítja a fájlnevet, megtartja a kiterjesztést, viszont a dátumot szerkeszti bele a konkrét névbe. Majd pedig készít egy `PutObjectCommand` típusú objektumot, amellyel a megfelelő helyre és névvel kerül feltöltésre az `s3Client`-en keresztül a fájl.

```

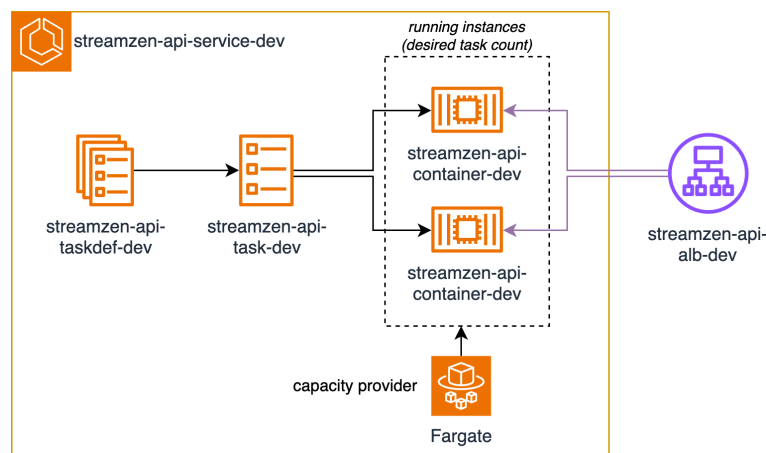
1  private readonly s3Client = new S3Client({
2      region: this.configService.getOrThrow("AWS_S3_REGION"),
3  })
4  async upload(id: string, fileName: string, file: Buffer) {
5      const ext = fileName.split(".").slice(-1)[0]
6      const baseName = new Date()
7          .toISOString().slice(0, 19).replaceAll(":", "-")
8
9      const response = await this.s3Client.send(
10         new PutObjectCommand({
11             Bucket: this.configService.getOrThrow("AWS_S3_UPLOADED_BUCKET"),
12             Key: `${id}/${baseName}.${ext}`,
13             Body: file,
14         })
15     )
16     return { ...response, fileName: `${baseName}.${ext}` }
17 }

```

8.3. kódrészlet. Videófeltöltés függvénye a VideoService osztályban.

8.3. A konténerizált környezet

A konténerizált webalkalmazás kialakítására a Docker ökoszisztémájának eszközeit vettem alkalmazásba. A fejlesztés során a *Docker Desktop* alkalmazást használtam, amely lehetővé tette a konténerek helyi futtatását, a *Docker Compose* segítségével pedig a Node.js-alkalmazás mellé felvettem egy PostgreSQL-adatbázis konténerét is, azok között a Compose-kóddal a konténerek közötti kommunikációt is könnyen meg tudtam valósítani.



8.4. ábra. Az ECS-klaszter elemei.

Az élesítése a Node.js-alkalmazásnak a korábban is ismertetett módon, egy ECS-klaszterba telepítve került kivitelezésre. A szolgáltatás felépítő elemeit, azok működését a 8.4. ábra mutatja be. A felkonfiguráláshoz szükséges Terraform-kódot a 8.4. kódrészlet

mutatja be. A klaszterbe csupán egy szolgáltatás került be: a REST API-t kiszolgáló webszerver.

Egy taszkdefiníciója van a teljes szolgáltatásnak, emiatt egyfajta taszk fog futni is, az fogja kitelepíteni a konkrét konténerpéldányokat. A taszk a konténerek képét természetesen a korábban ismertetett ECR-tárolóból tölti le, majd buildeli le. A konténerek erőforrásmenedzsmentjét a Fargate szolgáltatásra hagytam (lásd: `launch_type`). Az ECS-szolgáltatáshoz definiálja kapcsolatát az ALB-példánnyal (lásd: `load balancer` blokk a Terraform-kódban), az ALB pedig képes automatikusan felderíteni a konkrét konténereket, azokra pedig ráirányítani a forgalmat.

```
1  resource "aws_ecs_cluster" "this" {
2      name = "streamzen-api-cluster-${var.environment}"
3  }
4  resource "aws_ecs_service" "this" {
5      name                = "streamzen-api-service-${var.environment}"
6      cluster              = aws_ecs_cluster.this.arn
7      task_definition      = aws_ecs_task_definition.this.arn
8      desired_count        = var.ecs.desired_task_count
9      launch_type          = "FARGATE"
10
11      health_check_grace_period_seconds = 300
12      network_configuration {
13          subnets          = var.api_subnet_ids
14          security_groups    = var.api_secgroup_ids
15          assign_public_ip   = true # false if you have a NAT GW
16      }
17      load_balancer {
18          target_group_arn = aws_lb_target_group.this.arn
19          container_name    = "streamzen-api-${var.environment}"
20          container_port     = var.ecs.port_mapping
21      }
22  }
```

8.4. kódrészlet. A klaszter és szolgáltatás Terraform-kódja.

Az ECS egy orkesztrációs környezet is, a segítségével be tudjuk konfigurálni, hogy a konténerek miképp naplózzanak, milyen portokat nyissanak meg, milyen erőforrásokat használjanak, illetve milyen környezeti változókat kapjanak. A 8.5. kódrészlet mutatja be a taszkdefiníciót. A taszkdefinícióban található `container_definitions` blokkban található a konténerre vonatkozó beállítások többsége. Naplózásra az AWS CloudWatch Logs szolgáltatását használja a konténer.

```

1  resource "aws_ecs_task_definition" "this" {
2      family                = var.ecs.family_name
3      container_definitions = jsonencode([
4          {
5              volumes         = []
6              mountPoints     = []
7              healthCheck    = try(var.ecs.health_check, {})
8              portMappings   = local.port_mappings
9              environment    = [for k, v in var.ecs.task_environment : { name = k,
10                  value = v }]
11              memory         = var.ecs.memory,
12              cpu            = var.ecs.cpu,
13              image          = "${aws_ecr_repository.this.repository_url}:latest",
14              essential      = true,
15              name           = "streamzen-api-${var.environment}",
16              logConfiguration = {
17                  logDriver = "awslogs",
18                  options   = {
19                      awslogs-group      = aws_cloudwatch_log_group.this.name
20                      awslogs-region    = data.aws_region.current.name
21                      awslogs-stream-prefix = "ecs-streamzen-api-${var.environment}"
22                  }
23              }
24          }
25      ])
26      network_mode          = "awsvpc"
27      requires_compatibilities = ["FARGATE"]
28      memory                = var.ecs.memory
29      cpu                   = var.ecs.cpu
30      execution_role_arn    = aws_iam_role.ecs_service_install.arn
31      task_role_arn         = aws_iam_role.ecs_service.arn
32  }

```

8.5. kódrészlet. Az ECS-taszk Terraform-kódja.

Fontos két elemet kiemelni a `aws_ecs_task_definition` attribútumaiból: ez a `task_role_arn` és a `execution_role_arn`. A `execution_role_arn` az az IAM-szerepkör, amelyet az ECS-motor fog használni ahhoz, hogy AWS-en belüli hívásokat intézzon a felhasználó nevében, például a CloudWatch Logs szolgáltatásba való naplózásra vagy az ECR-ről való letöltésére a konténerképnek. A `task_role_arn` pedig az a szerepkör, amelyet konkrétan a taszk kap meg, a Node.js-app fogja használni, például az S3-vödörbe való feltöltéshez.

A szerveralkalmazás kódjának a telepítés szempontjából fontos része a Dockerfile, amely leírja a konténerkép felépítéséhez szükséges lépéseket. Ennek kódját mutatja be a 8.6. kódrészlet.

```

1  # Stage 1: Build the application
2  FROM node:20-alpine AS build
3  ENV NODE_ENV=development
4  WORKDIR /app
5  COPY package.json ./
6  COPY yarn.lock ./
7  COPY .yarnrc.yml ./
8  COPY prisma ./prisma/
9  RUN corepack enable
10 RUN yarn install
11 COPY . .
12 RUN npx prisma generate
13 RUN yarn build
14
15 # Stage 2: Create a lightweight container with the built app
16 FROM node:20-alpine AS production
17 ENV NODE_ENV=production
18 WORKDIR /app
19 COPY --from=build /app/dist ./dist
20 COPY --from=build /app/package.json ./package.json
21 COPY --from=build /app/yarn.lock ./yarn.lock
22 COPY --from=build /app/.yarnrc.yml ./yarnrc.yml
23 COPY --from=build /app/prisma ./prisma
24 RUN corepack enable
25 RUN yarn install --immutable
26 RUN npx prisma generate
27 CMD ["npm", "run", "start:migrate:prod"]

```

8.6. kódrészlet. Dockerfile tartalma.

A Dockerfile két szakaszból áll, az első szakaszban a Node.js-alkalmazás buildelése történik csak meg – standard Node.js-appokra jellemző folyamatot mutat be –, a második szakaszban pedig a buildelt alkalmazás kódja kerül átmásolásra kerül egy újabb Node.js-konténerbe, annak végén még a Prisma ORM-hez szükséges JavaScript-modulokat is legenerálja, az alapértelmezett parancs pedig, amely a konténer indulásakor a fő folyamatot indítja, az a `npm run start:migrate:prod` parancs. Ezt a futtató parancsot a `package.json` fájlban magam definiáltam, a Prisma ORM által biztosított migrációs eszközt hívja meg (`prisma migrate deploy`), amely a PostgreSQL-adatbázisban migrációt hajt végre, létrehozza a megfelelő táblákat vagy frissíti azokat, aztán pedig indítja a Node.js folyamatát (`node dist/main`).

8.3.1. A szerveralkalmazás CI/CD-folyamatai

A kliensoldalon is került ismertetésre 6.2.1 alfejezetben egy olyan GitHub Actions-alapú CI/CD-munkafolyamat, amely az AWS-fiókba lép be GitHub OIDC-t használva. Azonos-

képp a szerveroldali konténerkép telepítése a változtatások main főágba való olvasztása után az ott ismertetett autentikációs módszerrel kerül feltöltésre AWS-re.

Ennek a folyamatnak az esetében egyszerűbb volt a build- és telepítő folyamatot egybeépíteni, egy job végzi a kettőt. Ennek megfelelően a munkafolyamat miután megszerezte az AWS-fiókhoz hitelesítő adatokat, a következő lépéseket hajtja végre: belép az ECR-beli Docker Registrybe, majd a Docker-képfájlt buildeli, végül pedig feltölti a saját ECR-képtárolónkba. A 8.7. kódrészlet mutatja be a `deploy-server.yml` fájl releváns részét, amely a kifejtett lépéseket hajtja végre.

A szerveroldali Node.js-kód ellenőrzésére is készült egy CI/CD-folyamat, amely a `lint-server.yml` fájlban található és Pull Requestek létrehozásakor fut le. Ez a munkafolyamat a kliensoldali ellenőrzéshez hasonlóan a `eslint` és a `prettier` eszközöket használja a kód statikus ellenőrzésére és formázására. A munkafolyamat végén utolsó lépésként pedig az NPM-projekt buildelése van ellenőrzésképp.

```
1  deploy:
2    runs-on: ubuntu-latest
3    environment: production
4    defaults:
5      run:
6        working-directory: server
7    steps:
8      - name: Checkout code
9        uses: actions/checkout@v4
10     - name: Setup AWS
11       uses: ../.github/actions/setup-aws
12     - name: Login to ECR
13       run: aws ecr get-login-password --region eu-central-1 | docker login
14           --username AWS --password-stdin 339713096573.dkr.ecr.eu-central-1.
15           amazonaws.com
16     - name: Build image
17       run: docker build -t streamzen-api-repo-dev .
18     - name: Tag image
19       run: docker tag streamzen-api-repo-dev:latest 339713096573.dkr.ecr.
20           eu-central-1.amazonaws.com/streamzen-api-repo-dev:latest
21     - name: Push image
22       run: docker push 339713096573.dkr.ecr.eu-central-1.amazonaws.com/
23           streamzen-api-repo-dev:latest
```

8.7. kódrészlet. Részlet a `deploy-server.yml` fájl tartalmából.

8.4. Elemental MediaConvert felhasználása

A videófeltöltés és live indítás alfolyamatai a rendszernek implementációi eseményvezérelt architektúrában kerültek kivitelezésre (*event-drive architecture*, EDA[3]). Ennek az architektúrának egy eleme a Node.js-szerver is, amely a korábbi alfejezetekben került bemu-

tatásra. A feltöltés megtörténik a Node.js REST API-ján keresztül, majd pedig beindul egy másodlagos folyamat, amely a feltöltött fájlok átkonvertálásáért felelős. A feltöltés az S3-vödrön egy "s3:ObjectCreated" eseményt generál, amelyet egy Lambda-függvény kap el, a streamzen-job-starter-dev (8.2. ábra).

A Lambda-függvény célja megkomponálni az konvertáláshoz szükséges job definícióját, amely a feltöltött fájlt átkonvertálja HLS-folyamba helyezhetővé, és az elkészült fájlokat egy S3-vödörbe helyezi el (streamzen-static-assets-dev-bucket).

8.4.1. A MediaConvert-jobot indító Lambda-függvény

A függvénynek hat környezeti változót adtam meg, ezek mind kellenek a MediaConvert-job felépítésére, a legfontosabb kerülnek bemutatásra. Az AWS-fiókra egyedileg generálódik egy MediaConvert API HTTP-végpont, és ahhoz, hogy ezt megkapjuk, az AWS CLI segítségével hívni kell a következő parancsot: `aws mediaconvert describe-endpoints`. A kapott URI-t a `MEDIA_CONVERT_ENDPOINT` környezeti változóban passzolom át. Az `IAM_ROLE_ARN` az IAM-szerepkör ARN-je, feljogosítja arra a függvényt, hogy CloudWatch-ba naplózzon és az `OUTPUT_BUCKET_URI` és `INPUT_BUCKET_URI` változóknak jelölt kimeneti és bemeneti S3-vödrökben tudjon olvasni, létrehozni.

A 8.8. kódrészlet emeli ki jelentős részeit a Lambda kódjának. A `assembleJobCommand` függvény építi fel a MediaConvert API-ja számára a job paramétereit egy óriási JavaScript-objektumban. A következőket lehet leszűrni az objektumból:

- **Kimenetek száma:** 4
 1. Kimenet: 1080p képmagasság, max. videóbitráta 10 Mbps, hangbitráta 256 kbps
 2. Kimenet: 720p képmagasság, max. videóbitráta 5 Mbps, hangbitráta 192 kbps
 3. Kimenet: 480p képmagasság, max. videóbitráta 2 Mbps, hangbitráta 128 kbps
 4. Kimenet: 360p képmagasság, max. videóbitráta 1 Mbps, hangbitráta 96 kbps
- **Kimenetekre azonos mozgóképtulajdonságok:** HLS formátum, H.264 kódolású, 6 másodperces szegmensek, *Quality-Defined Variable Bitrate* (QVBR) bitráta[2].
- **Kimenetekre azonos hangtulajdonságok:** AAC audiókódolás, konstans bitráta (CBR), 48 kHz mintavételezés, 2 csatorna.

Minden más adat/konténertulajdonság, amit nem közöltem, az az alapértelmezett maradt vagy öröklődik a forrás videófájlból.

```

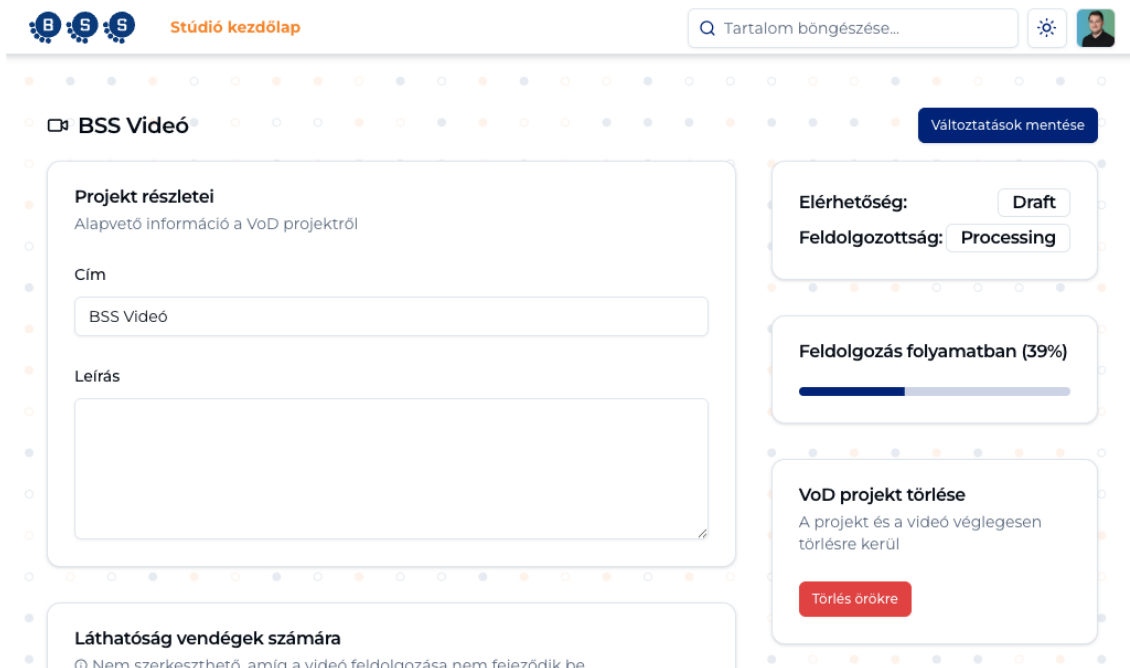
1  const assembleJobCommand = (config) => ({
2    Queue: config.jobQueueArn,
3    UserMetadata: {
4      id: `${config.id}`,
5      uploadedFilename: `${config.uploadedFilename}`,
6    },
7    Role: config.iamRoleArn,
8    StatusUpdateInterval: "SECONDS_20",
9    Settings: {
10     OutputGroups: [{
11       Outputs: [
12         { ... }, // 1080p settings
13         { ... }, // 720p settings
14         { ... }, // 480p settings
15         { ... }, // 360p settings
16       ],
17       OutputGroupSettings: {
18         Type: "HLS_GROUP_SETTINGS",
19         HlsGroupSettings: {
20           SegmentLength: config.segmentLength,
21           Destination: `${config.dest}`, ...
22         },
23       }, ...
24     ]},
25     Inputs: [{ FileInput: `${config.origin}`, ... }],
26   },
27 });

28

29 export const handler = async (event, context) => {
30   const callerInput = event.Records[0].s3;
31
32   const config = {
33     jobQueueArn: process.env.JOB_QUEUE_ARN,
34     iamRoleArn: process.env.IAM_ROLE_ARN,
35     origin: `${process.env.INPUT_BUCKET_URI}/${callerInput.object.key}`,
36     dest: `${process.env.OUTPUT_BUCKET_URI}/${callerInput.object.key}`,
37     id: callerInput.object.key.split("/")[0],
38     uploadedFilename: callerInput.object.key.split("/")[1],
39     segmentLength: process.env.SEGMENT_LENGTH ?? 6,
40   };
41
42   const data = await emcClient.send(
43     new CreateJobCommand(assembleJobCommand(config))
44   );
45 };

```

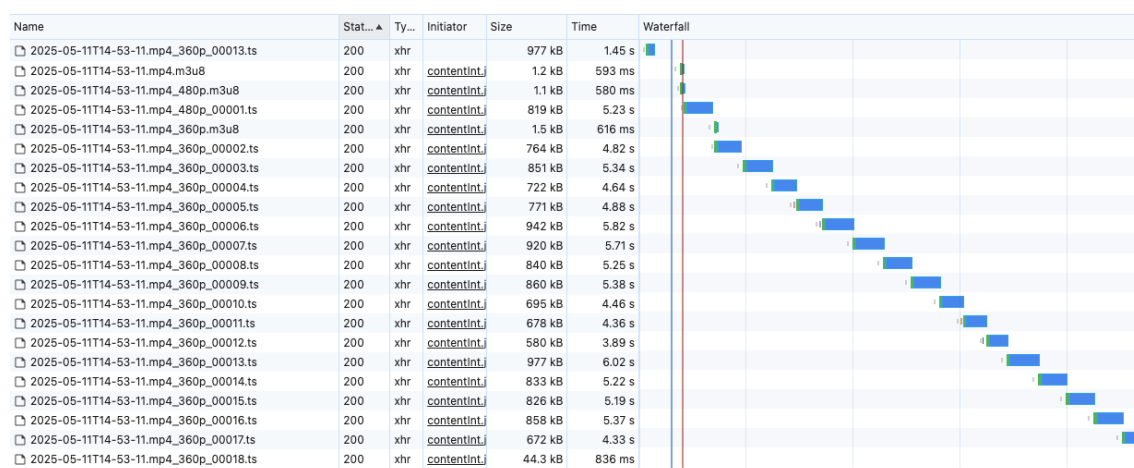
8.8. kódrészlet. Részlet a job-starter függvény JavaScript-kódjából.



8.5. ábra. Képernyőkép egy videóprojekt szerkesztői nézetében.

A 8.5. ábra bemutatja, miképp kerül visszajelzésre a felületen az, hogy hol áll a feldolgozottsága egy feltöltött videónak. A képernyőképen jelzett „BSS Videó” nevű projektbe egy 91 MB-os 1 perc 43 másodperc hosszú videófájl került feltöltésre. A feldolgozást megvizsgáltam a MediaConvert konzoljáról, a job 42 másodpercig tartott. A feltöltött videó további paraméterei ezek voltak:

- **Mozgóképanyag:** H.264 kódolású, 1920x1080p felbontás, 7 Mbps átlagos bitráta
- **Hanganyag:** AAC kódolás, 256 kbps bitráta, 2 csatorna, 48 kHz mintavételezés



8.6. ábra. Képernyőkép a HTTP-paketek érkezéséről (Slow 4G).

A 8.6. ábra mutatja be, ha átnavigálunk a weboldalon a streamelhető videó oldalára, akkor a böngészőben milyen packetek kerültek letöltésre a videófolyamokból. A böngészőben beállítottam sávszélességhatárítást (*Slow 4G*), és azt tapasztaltam, hogy a böngésző

elkezd a 480p-s folyamat letölteni, majd az első packet után 360p-sre vált. A videót az első packet megérkezése után rögtön elindítottam, a 360p-s videófolyam bufferelés nélkül tudott lejátszódni.

Kipróbáltam 3G-s beállítást is a Chrome böngészőben, azon a sávszálon már erősen hosszú bufferelésre volt szükség még a legalacsonyabb 360p-s felbontásnál is. Vártam másfél percet, hogy betöltsön néhány packetet előre a böngésző, utána már simább volt az élmény. Ez jelentős élményromlást tud jelenteni, ha egy körülbelül 2 perces videó lejátszásához is muszáj megállnom és bevárni pár packetet. Igény merülhet fel újabb és spórolósabb kimenetek bevezetésére a konvertálás folyamatába. A 360p-s packetek átlagban 1,2 MB-osak lettek, a CDN a legtöbbet átlagban 800 kB-osra tudta letömöríteni még. Az 1080p-s packetek átlagban 10 MB-osak lettek, a 720p-sek 6 MB-osak, a 480p-sek 2,4 MB-osak. Ezek esetében is a CDN 30-40% körüli tömörítést tudott elérni.

8.4.2. A MediaConvert-job státuszváltozásának kezelése

A 8.9. kódrészlet mutatja be a MediaConvert-job eseményeire való feliratkozást. A job-finalizer modulbeli Lambda-függvény roppant egyszerű kóddal rendelkezik, csupán behív az eseményből kapott paraméterekkel (job ID és feldolgozottság százaléka) az ALB-n keresztül a Node.js-appba a /api/videos/:id/progress végponton, a webszerver pedig frissíti az adatbázisban a videó állapotát.

```
1  resource "aws_cloudwatch_event_rule" "this" {
2      name          = "streamzen-mediaconvert-event-rule-${var.environment}"
3      description    = "Capture MediaConvert job state changes"
4      event_pattern = jsonencode({
5          source      = ["aws.mediaconvert"],
6          detail-type = ["MediaConvert Job State Change"]
7          detail      = {
8              status = ["COMPLETE", "ERROR", "STATUS_UPDATE"]
9              userMetadata = {
10                  application = ["streamzen-${var.environment}"]
11              }
12          }
13      })
14  }
15
16  resource "aws_cloudwatch_event_target" "this" {
17      rule          = aws_cloudwatch_event_rule.this.name
18      target_id     = "streamzen-mediaconvert-event-target-${var.environment}"
19      arn           = module.job_finalizer.arn
20  }
```

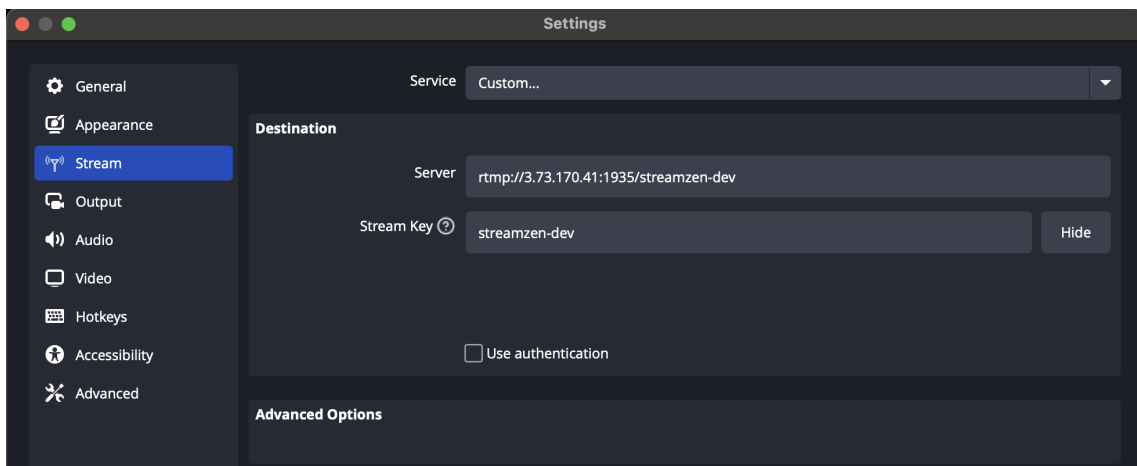
8.9. kódrészlet. A streamzen-mediaconvert-event-rule-dev Terraform-kódja.

8.5. A MediaLive és MediaPackage összekötése

A MediaLive szolgáltatásban fellőttem a tervezett egyetlen csatornát, amelyre egy RTMP push alapú inputot kötöttem be. A létrehozott inputhoz az AWS rendelt IP-t, portot, a kapott URL így ez lett: `rtmp://3.73.170.41:1935/streamzen-dev`. A csatorna úgy lett felkonfigurálva, hogy maximum 20 Mbps bitráta mellett 1080p-s felbontású videófolyamot tudjon fogadni, a hangot pedig AAC kódolásúként, 256 kbps bitrátaival. A MediaLive-csatornából átvezetett folyam két kimenetet képez, egy 8 Mbps célbitrátájú 1080p-s kimenetet ad, illetve egy 3,3 Mbps célbitrátájú 720p-s kimenetet. A MediaPackage szolgáltatásban is egyetlen csatornát hoztam létre, amely a MediaLive-csatorna kimeneteit fogadja be, és az átkonvertálja HLS-formátumú folyamra.

A MediaLive-csatornának még szüksége volt egy IAM-szerepkörre, amelyben megadtam a szerepkörnek minden olyan jogosultságot, amely szükséges a MediaPackage-csatornába való átvezetéshez számára, illetve a CloudWatch-ba való naplózásra és a szükséges hálózati/EC2-hez kapcsolódó jogokat is megkapta (a MediaLive háttérében menedzselte EC2-példányok futnak).

A MediaLive-csatorna indítása után lehetséges az input URL-re felcsatlakozva kezdeni a streamelést. Az RTMP streameléshez az OBS Studio nevű alkalmazást használtam[4], abban a beállításoknál a „Stream” menüben kitöltöttem az egyedi RTMP-szerverem adataival a mezőket (lásd 8.7. ábra). Nem került felhasználásra egyelőre a kísérlet alatt autentikáció a végpontra.

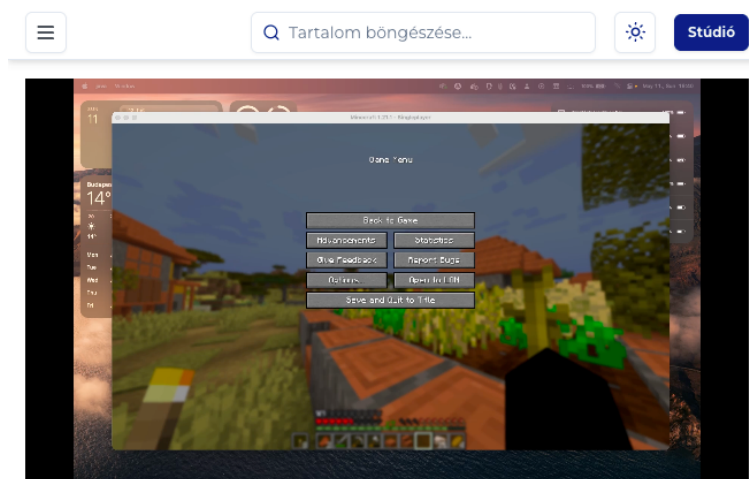


8.7. ábra. Képernyőkép az OBS stream beállításairól.

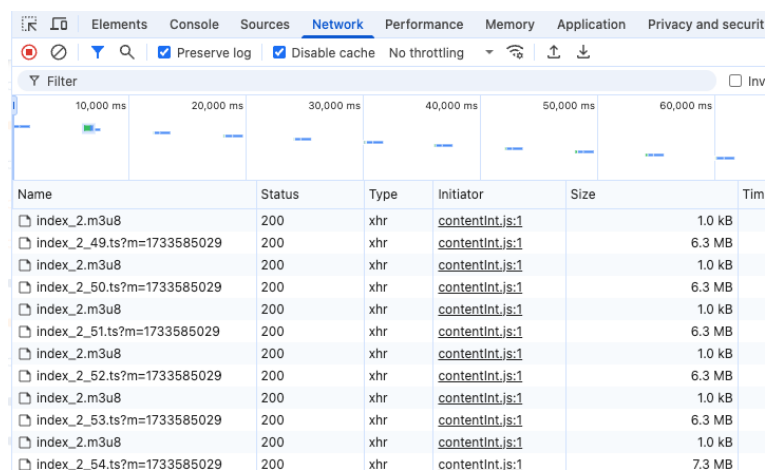
9. fejezet

Elkészült funkciók tesztelése

A weboldalamon a /live útvonalon megtekintettem az előzőleg (8.5. alfejezet) beindított streamet (9.1. ábra). Egy próbálkozást mutat be a 9.2. ábra képernyőkép, amely a böngészőben a HTTP-kérések vízésését[6] mutatja be.



9.1. ábra. A /live útvonalon látható aloldal.



Name	Status	Type	Initiator	Size	Tim
index_2.m3u8	200	xhr	contentInt.js:1	1.0 kB	
index_2_49.ts?m=1733585029	200	xhr	contentInt.js:1	6.3 MB	
index_2.m3u8	200	xhr	contentInt.js:1	1.0 kB	
index_2_50.ts?m=1733585029	200	xhr	contentInt.js:1	6.3 MB	
index_2.m3u8	200	xhr	contentInt.js:1	1.0 kB	
index_2_51.ts?m=1733585029	200	xhr	contentInt.js:1	6.3 MB	
index_2.m3u8	200	xhr	contentInt.js:1	1.0 kB	
index_2_52.ts?m=1733585029	200	xhr	contentInt.js:1	6.3 MB	
index_2.m3u8	200	xhr	contentInt.js:1	1.0 kB	
index_2_53.ts?m=1733585029	200	xhr	contentInt.js:1	6.3 MB	
index_2.m3u8	200	xhr	contentInt.js:1	1.0 kB	
index_2_54.ts?m=1733585029	200	xhr	contentInt.js:1	7.3 MB	

9.2. ábra. HTTP-kérések vízésése a /live oldalon.

Az SPA-alkalmazás a környezeti változóból beállított HLS-végponton kéri le először a .m3u8 playlistet, amelyből megtudja, mi lesz a következő szegmens, amelyet le kell töltsen. Az ábrából látszik, hogy az alapértelmezett 6 másodperces részekre vannak felbontva a szegmensek. Nagyjából 6–7 MB-os szegmensek érkeznek a böngészőbe (már tömörítve a CDN-en keresztül).

A MediaPackage v2-es verziója támogatja a Low Latency HLS (LL-HLS) alapú streamelést is, azonban én még a v1-es verziót használtam és a kliensoldali implementációs is meghagytam a hagyományos HLS-en. A fent ismertetett élő stream MediaLive és MediaPackage közti integrációjának manuális tesztelése folyamán körülbelül 10 másodperces késleltetést éreztem.

Teljesítmény szempontjából a fenti streamelés késleltetését két dolog befolyásolja jelentősen: a kódolási beállítások (mekkora szegmensekre bontsa, hányféle kimenet legyen, milyen hibaarányal érkezik a forrásvideó) és a lejátszó bufferelési logikája. Előbbi befolyásoló tényezőnek a mögöttes forrása a MediaLive maga, ami a kódolást intézi. Mind a MediaLive és a MediaPackage menedzselt szolgáltatások, magunk nem tudunk hatással lenni az alatti erőforrások eloszlására, annak növelésére, ugyanis automatikusan skálázódnak, ahogy arra a szolgáltatás elköteleződik is.

A késleltetés mérésére a nézők számának növelésével nem tudnánk hasznos teljesítményteszteket készíteni, a live betöltődik a MediaLive-ba, utána a kész szegmensek kiszórása az, ami kihívás a rendszer számára, viszont ezt ahogy fentebb kifejtettem, a MediaPackage automatikusan skálázódással oldja meg, csupán abban az esetben érheti teljesítményromlás a rendszert a MediaPackage hatáskörén kívül, ha az már jelentős terhelést fejtenek ki a felhasználók magára az AWS hálózatra.[27]

TODO: szerver teljesítménytesztje.

10. fejezet

Összegzés

A streaming rendszereket kiszolgáló technológiák fejlődése közel sem ért véget, a QUIC protokoll előretörésével, új modern CDN-funkcionalitásokkal, szerverfarmokban használt alkalmazásspecifikus chipek fejlődésével, illetve hatékonyabb kodekek megjelenésével egyre magasabb minőséget lehet szolgáltatni. Az általam bemutatott rendszer egy egyszerűsített példája a streaming rendszereknek, amely lehetőséget ad arra, hogy megértsük a mögöttes technológiákat és azok működését, betekintést nyújt néhány tipikusan streamingre használt AWS-erőforrás világába, a tervezési technikák palettájába.

A fejlesztés ideje alatt ahogy egyre több komponenst kötöttem a rendszerbe, nőtt a felhőszolgáltatások költsége is. Az dolgozat írásához többször újratesteltem a futó rendszert, a márciusi összköltsége AWS-ben \$67.98 lett, áprilisban \$73.29, ezekben az összegekben a legjelentősebbek közé tartozott az RDS-adatbázis, az ALB-példány és az ECS-klaszter költsége, ezek nem on-demand jellegűek, jó pár szolgáltatás pedig a *free tier*, azaz ingyenes átkategóriában tudott maradni.

10.1. Továbbifejlesztés lehetőségei

Architekturális szempontból a webservert átszervezhető mikroszolgáltatásos alapokra, amennyiben a jövőben a rendszer bővítése, új funkciók bevezetése indokoltá teszi, ezzel is jobban támogatva a már megkezdett event-driven architecture (EDA)[3] irányvonalat.

Biztonság szempontjából a rendszer továbbfejlesztése érdekében érdemes lenne megnövelni a naplózást kiterjeszteni a CDN és a WAF szintjén „access logging”-ra is, a VPC-n belül a Flow logok bekapcsolására. Élő, nagy forgalmú rendszerben kihagyhatatlan kötelességünk volna a WAF ACL-ben foglalt szabályokat finomhangolni veszélyes forgalom kiszűrésére. A 8.1. alfejezetben említett NAT Gateway és VPC Endpoint megoldások alkalmazása is javallott volna.

A kiszolgálás minőségének javítására érdemes lehet akár a CloudFront-disztribúció cache-beállításait is optimalizálás szempontjából újraszemlézni. Felmerülhet egy nagyobb rendszernél a webservert felé irányított forgalom hálózati terheléséből származó kihívásokra felkészíteni az ALB-példányt is, erre lehet igénybe venni Load balancer Capacity Unit (LCU) foglalat, amely biztosít megfelelő számítási kapacitást a forgalom fogadására.

Emellett az ECS-klaszterbe is be lehet vezetni egy automatikus skálázási megoldást, amely a forgalom növekedésével automatikusan új taszkokat indít el az ECS-szolgáltatásban, ezzel biztosítva a megfelelő számítási kapacitást.

A rendszer erős kódmigrálást, infrastruktúra-átalakítást igényelne, amennyiben cél volna a *vendor lock-in*[22] kockázatainak mitigálása. Ehhez javasolt lenne a Kubernetes-re való áttérés ECS-ről[25], Ceph¹ használata S3 helyett, a Kubernetes alá hozható az adatbázis, illetve a Lambda-függvények is kis mikroszolgáltatások formájában. A Media-Convert, a MediaLive és MediaPackage szolgáltatások kiváltása már bonyolultabb lehet, ebben segíthet az általam korábbi projektben megismert FFMpeg és az NGINX RTMP-modulja[11], ahogy azt a 1.1 alfejezetben is említettem.

Nem utolsó sorban a live streaming kiterjesztése még egy feladat, ami várat magára. A rendszer jelenleg nem engedi több élő adás indítását, ezt lehetne kiterjeszteni kódból történő dinamikus MediaLive- és MediaPackage-csatornák felkonfigurálására.

¹<https://ceph.io/en/>

Irodalomjegyzék

- [1] Amazon Web Services: CDN authorization in AWS Elemental MediaPackage. <https://docs.aws.amazon.com/mediapackage/latest/ug/cdn-auth.html>. [Hozzáférés dátuma: 2025-04-27].
- [2] Amazon Web Services: Quality-Defined Variable Bitrate (QVBR). <https://aws.amazon.com/media/tech/quality-defined-variable-bitrate-qvbr/>. [Hozzáférés dátuma: 2025-05-11].
- [3] Amazon Web Services: What is an Event-Driven Architecture? <https://aws.amazon.com/event-driven-architecture/>. [Hozzáférés dátuma: 2025-05-12].
- [4] Amazon Web Services: OBS Studio to AWS Elemental MediaLive to AWS Elemental MediaPackage – Workflow Example. Jelentés, 2020, Amazon Web Services. <https://d2908q01vomqb2.cloudfront.net/fb644351560d8296fe6da332236b1f8d61b2828a/2020/04/14/Connecting-OBS-Studio-to-AWS-Media-Services-in-the-Cloud-v2.pdf>.
- [5] Apple Inc.: HTTP Live Streaming | Apple Developer Documentation. <https://developer.apple.com/documentation/http-live-streaming>. [Hozzáférés dátuma: 2025-03-24].
- [6] Kayce Basques–Sofia Emelianova: Chrome DevTools – Network features reference. <https://developer.chrome.com/docs/devtools/network/reference>. [Hozzáférés dátuma: 2025-05-12].
- [7] Abdelhak Bentaleb–Bayan Taani–Ali C. Begen–Christian Timmerer–Roger Zimmermann: A survey on bitrate adaptation schemes for streaming media over http. *IEEE Communications Surveys & Tutorials*, 21. évf. (2019) 1. sz., 562–585. p. <https://ieeexplore.ieee.org/document/8424813>.
- [8] Andra Christie–Shyam Arjarapu–Ravi Tallury: Choose the right aws video service for your use case. Jelentés, 2021, Amazon Web Services. <https://aws.amazon.com/blogs/iot/choose-the-right-aws-video-service-for-your-use-case/>.
- [9] Cloudflare Inc.: What is a content delivery network (CDN)? | how do CDNs work? <https://www.cloudflare.com/learning/cdn/what-is-a-cdn/>. [Hozzáférés dátuma: 2025-03-25].

- [10] Flavio Copes: Jwt authentication: Best practices and when to use it. Jelentés, 2024, LogRocket. <https://blog.logrocket.com/jwt-authentication-best-practices/>.
- [11] F5 NGINX: RTMP – NGINX Docs. <https://docs.nginx.com/nginx/admin-guide/dynamic-modules/rtmp/>. [Hozzáférés dátuma: 2025-05-12].
- [12] Frequent Questions | WebRTC. <https://webrtc.github.io/webrtc-org/faq/>. [Hozzáférés dátuma: 2025-03-24].
- [13] Dan Gehred: Support for aws elemental mediastore ending soon. Jelentés, 2024, Amazon Web Services. <https://aws.amazon.com/blogs/media/support-for-aws-elemental-mediastore-ending-soon/>.
- [14] Henry Golas: S3 bucket policies: A practical guide. Jelentés, 2021, Cloudian. <https://cloudian.com/blog/s3-bucket-policies-a-practical-guide/>.
- [15] ISO Central Secretary: Information technology – Dynamic adaptive streaming over HTTP (DASH) – part 1: Media presentation description and segment formats. ISO/I-EC 23009-1:2022. Standard, Geneva, CH, 2022, International Organization for Standardization / International Electrotechnical Commission.
URL <https://www.iso.org/standard/83314.html>.
- [16] Csaba Kopiás: Ffmpeg - the ultimate guide. Jelentés, 2022. <https://img.ly/blog/ultimate-guide-to-ffmpeg/>.
- [17] Ben Lee – Anuj Butail – Igor Kushnirov: Amazon cloudfront introduces origin access control (oac). Jelentés, 2022, Amazon Web Services. <https://aws.amazon.com/blogs/networking-and-content-delivery/amazon-cloudfront-introduces-origin-access-control-oac/>.
- [18] Kamil Myśliwiec: Passport (authentication). <https://docs.nestjs.com/recipes/passport>. [Hozzáférés dátuma: 2025-05-11].
- [19] Savannah Ostrowski: containerd vs. docker: Understanding their relationship and how they work together. Jelentés, 2024, Docker Inc. <https://www.docker.com/blog/containerd-vs-docker/>.
- [20] David Rowe: Use iam roles to connect github actions to actions in aws. Jelentés, 2023, Amazon Web Services. <https://aws.amazon.com/blogs/security/use-iam-roles-to-connect-github-actions-to-actions-in-aws/>.
- [21] Sydney Roy: What is CMAF? Jelentés, 2022, Wowza Media Systems, LLC. <https://www.wowza.com/blog/what-is-cmaf>.
- [22] Andrii Ryzhokhin: Understanding vendor lock-in and strategies to avoid it. Jelentés, 2023, Ardas International Inc. <https://ardas-it.com/understanding-vendor-lock-in-and-strategies-to-avoid-it>.

- [23] Servers.com: The history of streaming told through protocols. Jelentés, 2023, Servers.com. https://www.servers.com/docs/whitepaper/history_of_streaming_told_through_protocols.pdf.
- [24] Hezbollah Shah – Tariq Soomro: Node.js challenges in implementation. *Global Journal of Computer Science and Technology*, 17. évf. (2017. 05), 72–83. p.
- [25] Cody Slingerland: Ecs vs. kubernetes: A detailed guide to container solutions. Jelentés, 2024, CloudZero. <https://www.cloudzero.com/blog/ecs-vs-kubernetes/>.
- [26] Thomas Stockhammer: Dynamic adaptive streaming over http: Standards and design principles. 2011. 02, 133–144. p. <https://dl.acm.org/doi/10.1145/1943552.1943572>.
- [27] Nicolas Weil: Part 4: How to compete with broadcast latency using current adaptive bitrate technologies. Jelentés, 2018, Amazon Web Services. <https://aws.amazon.com/blogs/media/part-4-how-to-compete-with-broadcast-latency-using-current-adaptive-bitrate-technologies/>.
- [28] Christer Whitehorn: Choosing the right aws live streaming solution for your use case. Jelentés, 2021, Amazon Web Services. <https://aws.amazon.com/blogs/media/aws-e-choosing-aws-live-streaming-solution-for-use-case/>.