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. március 26.

Tartalomjegyzék

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. március 26.

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 felhő alapú médiaszolgáltatás-rendszer alapos tervezésének, implementálásának, valamint tesztelésének 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 fókuszt 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 testing 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

1.1. A téma ismertetése

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.

A média streaming szolgáltatások fejlesztése és fenntartása komoly kihívások elé állítja a tervezőmérnököket, és a szakma legjobbjai a világ minden tájáról dolgoznak azon, hogy a felhasználói élményt folyamatosan javítsák, és a szolgáltatásokat a lehető legnagyobb számú felhasználó számára elérhetővé tegyék.

1.2. A témaválasztás indoklása

Véngponttól 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. Mérni kell a metrikákat a kitűnő felhasználói élmény biztosítása érdekében. Ki kell tudni szolgálni termérdekféle 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.

1.3. 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 kliens oldali 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-felhő alapú 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, visszajátszására. 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 szabványban 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 hibatűréséről ismert; Apple vállalat által macOS-re és iOS-re optimalizálva fejlesztett QuickTime Movie (MOV) formátuma; valamint a WebM, egy szabad felhasználású webes 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 sávszélesség 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 médiastreamelés egy olyan folyamat, amely során az médiaadatokat – mi esetünkben videóadatot – egy adott hálózati protokoll felett, egy adott konténerformátumban, adott kódolásssal 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

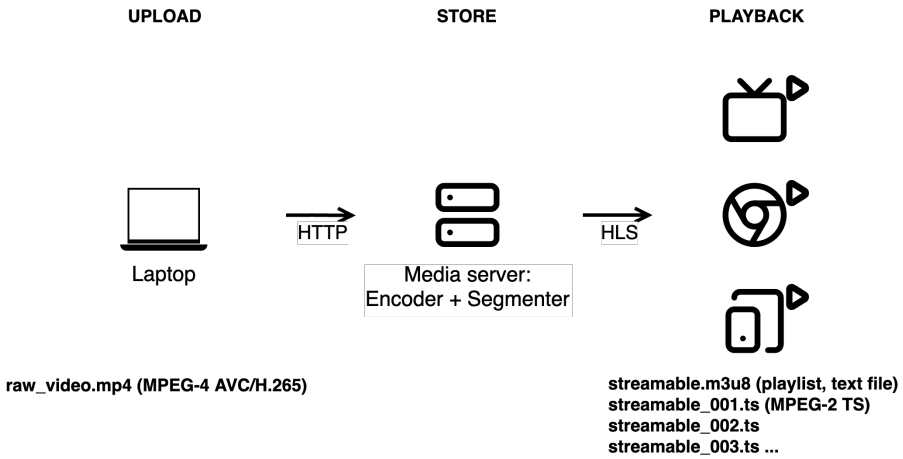
Annak megfelelően, hogy az adat egésze mikor áll rendelkezésünkre, kettő fő streaming típust különböztetünk meg: az élő közvetítést (live streaming) és a video-on-demand (VOD) streaminget. A VOD esetében a videóadatokat előre rögzített formában tároljuk, és a végfelhasználók a videóadatokat a saját időbeosztásuknak 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 vég-

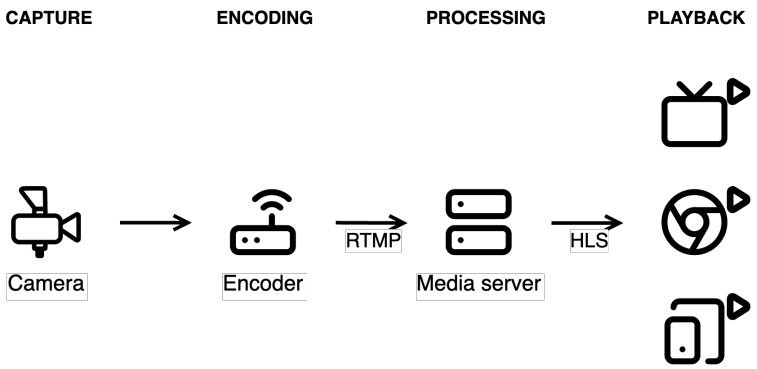
felhasználók felé, hogy a közvetítés valóban élőnek tűnjön, ehhez magas számítási teljesítmény szükséges. A VOD esetében a hálózati sávszélességből adódó problémák leküzdése a központi kihívás, mivel a videót a felhasználók sokkal nagyobb közönsége kívánja elérni, azt kiváló minőségben szeretné megtekinteni, 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.

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.



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

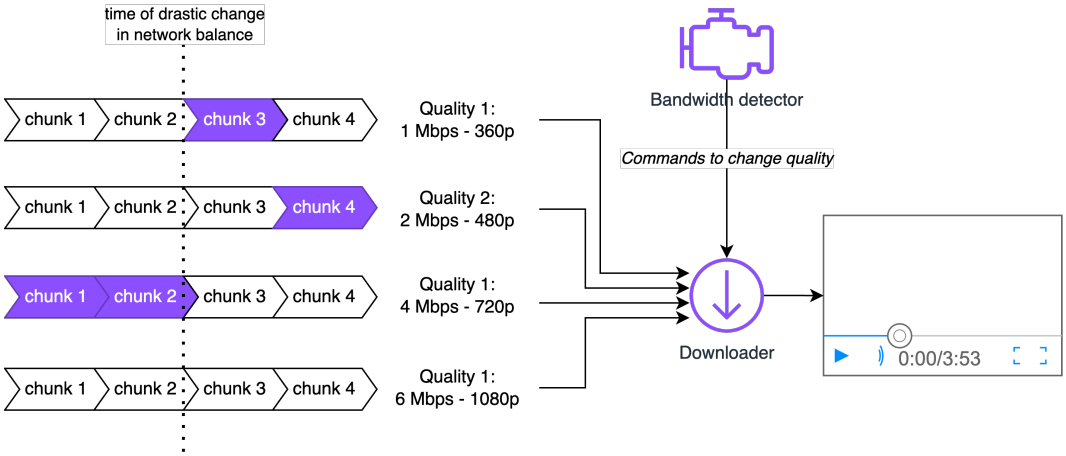


2.2. ábra. Példa egy élő közvetítés résztvevő eszközeire.

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) 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).



2.3. ábra. Az Adaptive Bitrate Streaming működése.

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 [2]. 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. Videó streaming hálózati protokolljai

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

Néhány ismertebb streaming használatára kitalált protokoll létrejöttük időrendjében [11]:

- 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 legelterjedtebb protokollokat, megemlítve, 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 szervertől 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. [11] 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 élőadást fogadni, tehát szerver oldalon még használt, persze a Flash Player támogatásának 2020-as kivezetése miatt a protokoll használata kliens oldalon 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 függelten packetekre szedi a teljes kiszolgálandó videót, és az RTMP-hez képest ez így állapotmentes adatforgalmazást tud megvalósítani. [11]

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. [10] 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. [1] Lásd a példát egy 720p-s streamet leíró indexfájlra 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.

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). [13]

A HLS-hez képest ez a protokoll kodekagnosztikus, ami annyit jelent, hogy nem kötődik videókodekhez, használható H.264, H.265, akár VP9 is. [6] Igyekeztek ezzel a protokollal egyezményesíteni a tartalomvédelmet is, Common Encryptiont (CENC) használ titkosításra. Digitális jogkezelésre (Digital Rights Management,

[1] `https://github.com/video-dev/hls.js`

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 böngésző alapú valós idejű kommunikáció kialakítására. Ezt használja a Discord, a Google Chat és egyéb webes videóchat-alkalmazások. [11]

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 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). [4]

[2] `https://dashjs.org/`

3. fejezet

Követelmények

Egy nagyszabású, YouTube- vagy Netflix-szintű webes videóstreaming-szolgáltatás megvalósítása rendkívül összetett feladat, amely köré így kiterjedt technikai és üzleti követelményrendszert tudunk kialakítani. Egy ilyen rendszernek kezdetben biztosítania 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 AuthSCH[1] segítségével tudnak bejelentkezni a rendszer „stúdió” nevezetű aloldalára, ahol a rendszer azonosítja őket AuthSCH-s

[1] `https://vik.wiki/AuthSCH`

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 előbbieknek jogosultságuk van a videók 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 stábtagok.
  - 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 „stúdión” 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 kezelhetőek, 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 FFMpeg[1] egy nyílt forráskódú és GPL-licenszelésű szoftvercsomag, amely képes videók és hangok kódolására, dekódolására, átalakítására (konvertálás), valamint streamelésre. [7] 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 felhő alapú megoldásban már nem került felhasználásra az FFMpeg, mivel az AWS Elemental szolgáltatásokat használtam a videókódolásra és streamelésre, viszont a szoftvercsomag közvetlen megismerése a videókonvertálás folyamatának megértését és a konvertálási folyamatok felparaméterezési lehetőségeinek mélyebb átlátását segítette.

[1] `https://www.ffmpeg.org/`

4.2. Open Broadcaster Software

Az Open Broadcaster Software Studio (OBS Studio[2]) egy nyílt forráskódú, szabad szoftver, amelyet elsősorban élő közvetítésekhez és képernyőrögzítéshez használnak. Elterjedt Twitch-felhasználók körében. A program támogatja a Windows, macOS és Linux operációs 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 hírű 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 granuláris 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.

[2] `https://obsproject.com/`

4.3.1. AWS Elemental

Az Elemental Technologies 2006-ban indította vállalkozását streamingmegoldások eladására, egy fő mérföldkövük 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 videókódolásokat és -technológiákat alkalmazzák. [3]

Szoftveres megoldásaik közé tartoznak a következők:

- **Elemental MediaConvert**: A MediaConvert egy felhőalapú videókódoló 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.

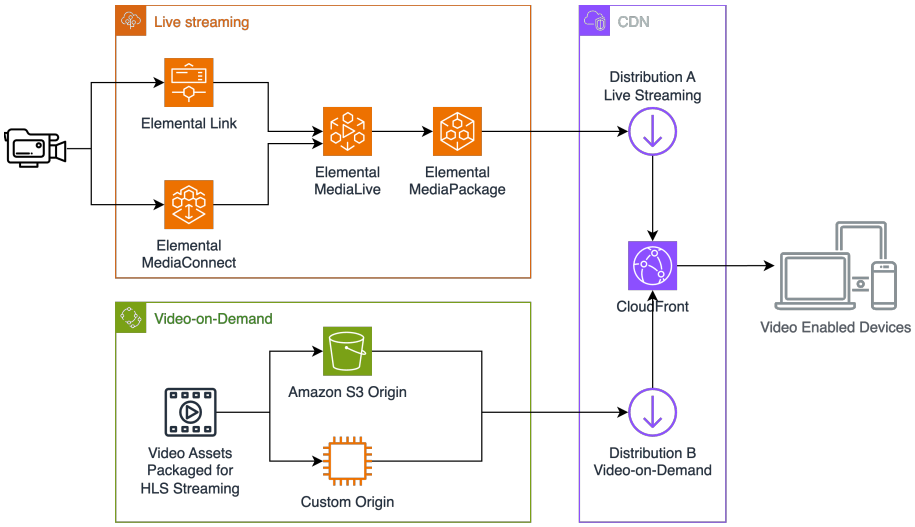
- **Elemental MediaLive**: Az Elemental MediaLive egy élő videókódoló szolgáltatás, amely lehetővé teszi a felhasználók számára, hogy élő videóadásokat fogadjanak és kódoljanak át a felhőben. A MediaLive támogatja a legnépszerűbb élővideó feltöltési protokollokat, így az RTMP-t is. Ennek a használata már bonyolultabb, mint a MediaConverté, nem szimplán csak egy API hívásaként 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 videófolyamot hálózati protokollokon szállítmányozásra, garantálja a biztonságos és folyamatos tartalomtovábbítást. Biztosíthatja VOD-ok S3-ból való továbbosztását, vagy élők továbbosztását a MediaLive-ból. 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 MediaConnectet, 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. [5]

Ezeken kívül az AWS szolgáltat 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 szolgáltat mind VOD-ok és élőadások kiszolgálására a 4.1. ábra.



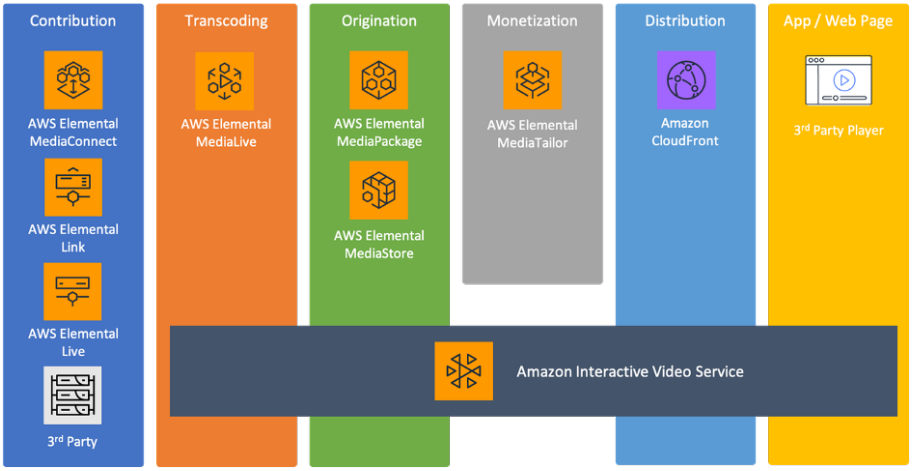
4.1. ábra. Példa AWS Elemental szolgáltatások architektúrába kötésére.

4.3.2. Amazon IVS

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 kis késleltetésű többnézős alkalmazásra: HLS-alapú kiszolgálás, körülbelül 5 másodperces késleltetést szokott biztosítani; valamint valós idejű alkalmazásra is: WebRTC-alapú kiszolgálás. [14] A forrásvideó-kódolástól a tartalomkiszolgálásig minden szükséges funkciót biztosít (4.2. ábra[3]), nekünk csupán a Software

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

Development Kitjét (SDK) kell használni a saját alkalmazásunkban, és a többit az AWS-re bízhatjuk. Ezenkívül biztosít olyan funkcionalitásokat, mint chatszobák és szavazások szolgáltatása, amelyeket könnyen integrálhatunk ezekbe az adásokba.



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

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

4.3.3. Amazon VPC

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.

4.3.4. Amazon ALB

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.

4.3.5. 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 „vödrökben" (bucketokban). Az objektum egy fájl és a fájlt leíró metaadatok közösen. A vödör az objektumok tárolója.

4.3.6. Amazon CloudFront

Az előző fejezet egy szekciójában már említetésre kerültek a CDN mint a video-on-demand alapú streaming szolgáltatások egyik kulcsfontosságú eleme. Az Amazon CloudFront egy globális 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-útvonalaikon akár különböző CDN-forrásokból – úgynevezett „originekből" – tudnak tartalmat kiszolgálni: ilyen origin lehet egy S3 vödör, AWS Elemental MediaPackage alapú élőadás-csatorna, Amazon Application Load Balancer (ALB) példány, vagy akár egy egyéni HTTP-szerver is saját doménnévvel.

4.3.7. Amazon ECS

Az Amazon Elastic Container Service (ECS) arra szolgál, hogy konténer alapú 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 patchelését 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.

4.3.8. Amazon RDS

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.

4.3.9. 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-felhőszolgátatásokon futó alrendszerek közötti kommunikációt. Segítségével szűrhetünk eseményekre, azokat könnyen továbbíthatjuk 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 a blokkoló műveletek nélküli működést, maximalizálja a skálázhatóságot. [12]

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

A TypeScript egy szuperhalmaza a JavaScriptnek – azaz a JavaScript szintaxisát bővíti ki –, amely szigorú és statikus típusosságot ad hozzá. A nyelvben írt kód a TypeScript fordító 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 React[4] egy nyílt forráskódú JavaScript-könyvtár, amelyet a Facebook (ma Meta) fejlesztett 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 (User Interface, 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/`

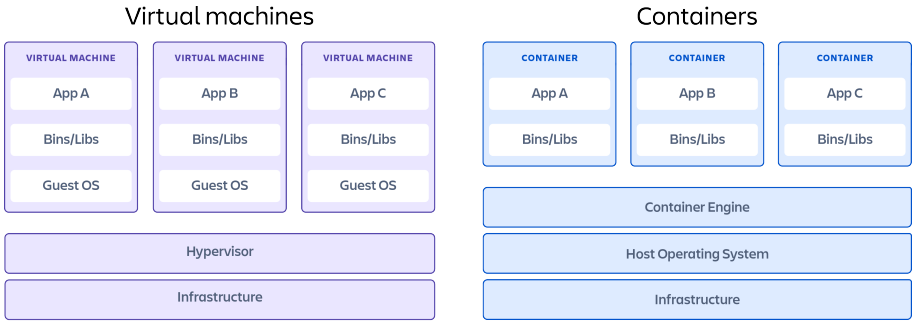
4.5. Üzemeltetési technológiák

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 oszák tovább, és ezekből a képekből konténereket futtathassanak a számítógépükön vagy épp a felhőben. Egy virtuális gép a gazda gép hardvereit virtualizálja, a konténer pedig a gazda operációs rendszert virtualizálja, azaz a konténerek alatt közös a kernel (4.3. ábra[5]). Ezzel elveszik a teljes izoláció, azaz a biztonság, de a konténerek könnyebbek, gyorsabban indulnak (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 architekturális összehasonlítása.

A Docker egy konténerizációs fejlesztői környezet, a mélyén a containerd motor fut konténerizációs futtatókörnyezetként. [8] 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.

[5] A kép forrása: `https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms`

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 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 GitHub[6]. 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 building folyamatok automatizálása, és például a kód AWS-re való kiélesítése is.

4.5.3. Terraform

A Terraform[7] egy elterjedt Infrastructure as Code 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, CDK-jait (Cloud Development Kit) használja az változtatások érvényre juttatására. Go nyelven íródott a motorja, a Terraform IaC-ra pedig a saját HCL (HashiCorp Configuration Language) 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 felépítését mutatom be, majd a fizikai felépítését az AWS-felhőben. A logikai felépítés a követelmények alapján készül függetlenül egy választott platformtól, türközi 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 többrétegű 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.

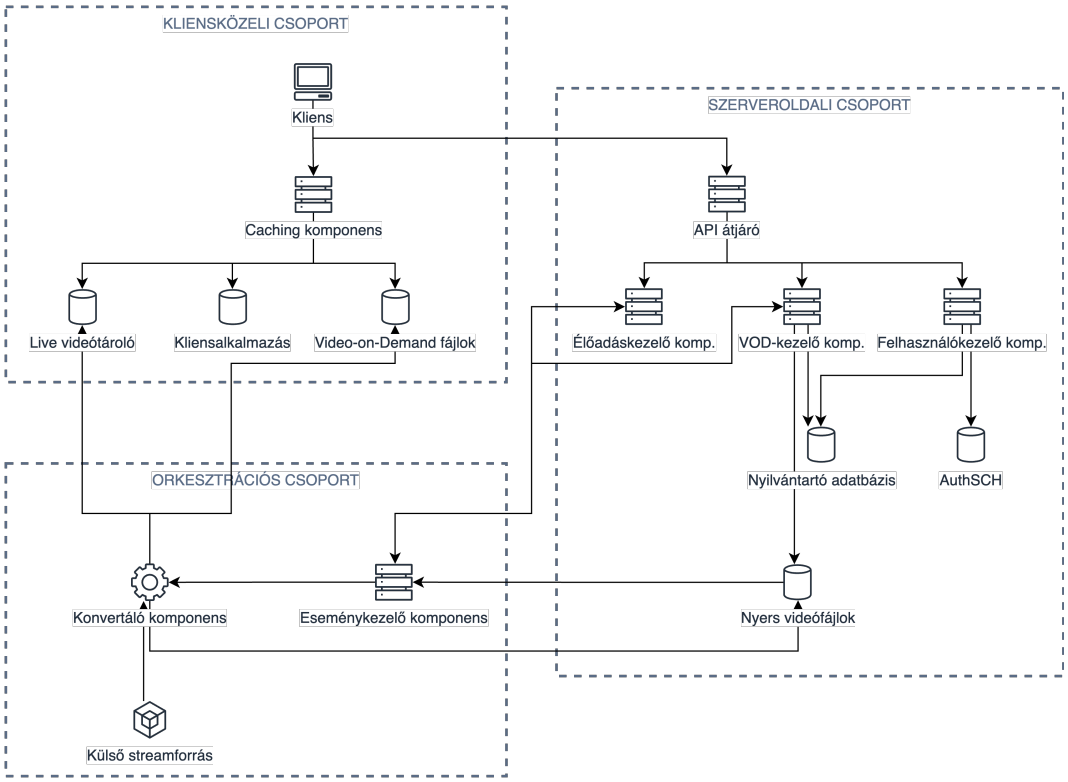
Érdekes lehet megfigyelni, hogy az egyes csoportok konkrét megvalósítása akár kicserélhetővé válik, egy-egy csoport mögötti teljes szoftvercsomag könnyen lekapcsolható a másik kettőről, csupán jól definiált és agnosztikus API-okra van szükség.

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 előadás-, VOD- 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 előadás- és VOD-kezelő az „orkesztrációs” csoportban lévő eseménykezelővel. Az orkesztrációs réteg fogja a két csoportot össze, események hatására indíttat 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.

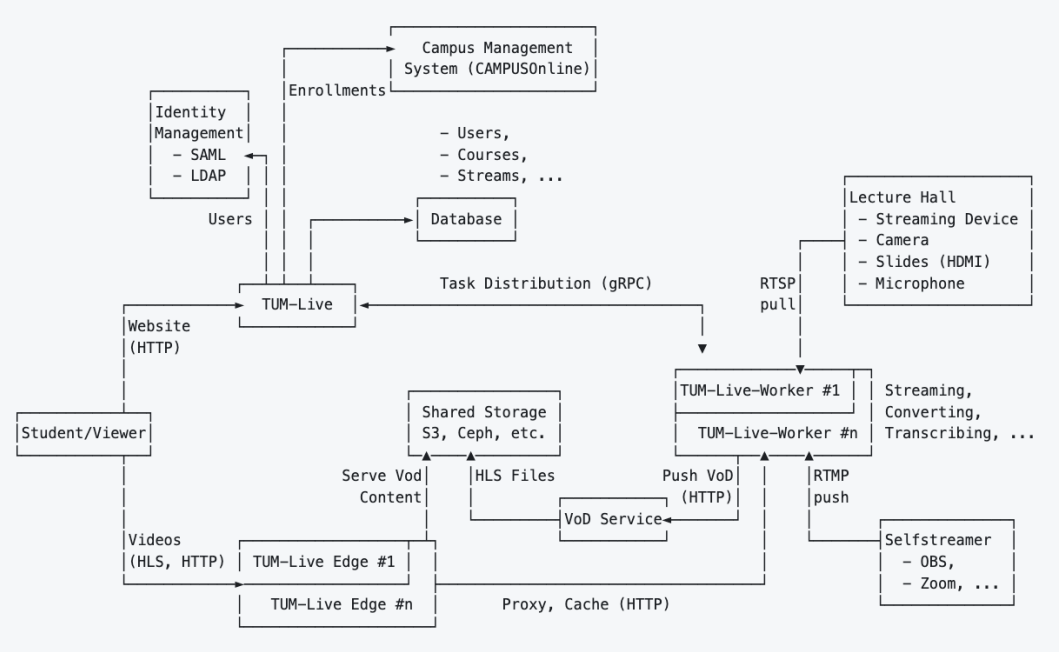
Megjegyzés: az itt feltüntetett komponensek előadás-, VOD- és felhasználókezelő moduljai a későbbiekben a konkrét szoftverarchitektúrában monolit struktúrában egy közös szerveralkalmazásba kerültek bele, azon belül kerültek modularizálásra az üzleti logikában.

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 GoCastot[1]. Ez a rendszer önállóan hosztolható szoftvereket komponál össze, nem felhőnatív. A rendszerben hasonló absztrakt terveket 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 több portos 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 funkcionalitásokat, 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.

[1] https://github.com/TUM-Dev/gocast

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.

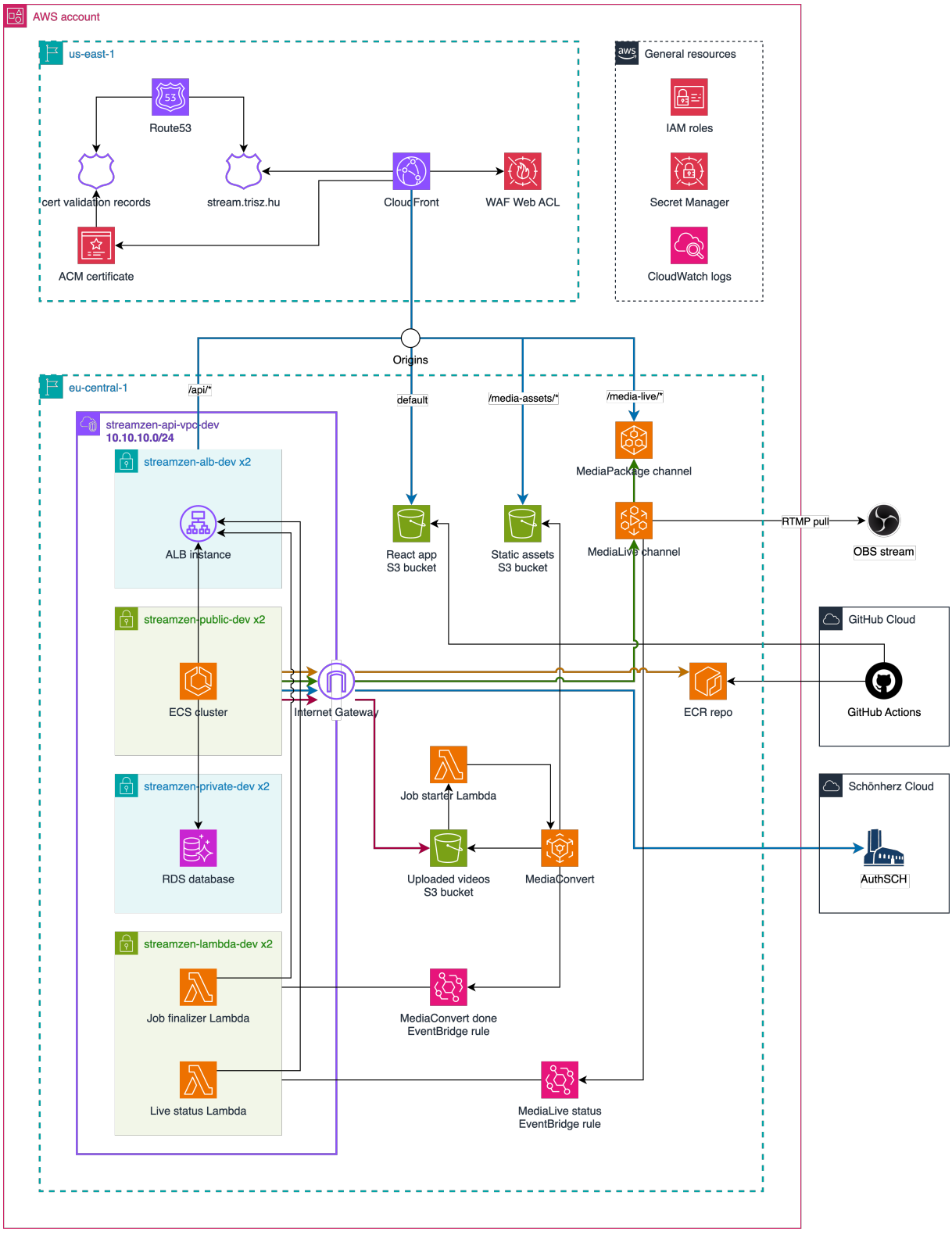
Az ábrát olvasva látható, hogy egészen jól elkülönülhetnek ebben a nézetben is csoportokra az egyes erőforrások. A kliensközeli csoportot a CloudFront CDN szolgálja ki, ennek a disztribúciónak adtam domainnevet, így használatba került egy stream.trisz.hu domain alatti Amazon Route53-zóna is, illetve egy SSL-tanúsítvány is hozzáadásra került. A védelmezésre egy AWS Web Application Firewall web Access Control List – röviden: egy WAF web ACL – is bekerült a disztribúció elé. Ezen szolgáltatások az edge szerverfarmokon működnek, az AWS ehhez azt írja elő, hogy az erőforrásokat a „us-east-1", azaz az észak-virginiai régióban kell elhelyezni.

A disztribúció után jönnek egy új rétegben először egy ALB-példány, ez és a mögötte lakó erőforrások az „eu-central-1" (frankfurti) régión belül is egy saját hálózatba, azaz AWS VPC-be kerültek. Ezenkívül videók S3-vödrét, a React alkalmazás vödrét 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 szerveralkalmazás 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.

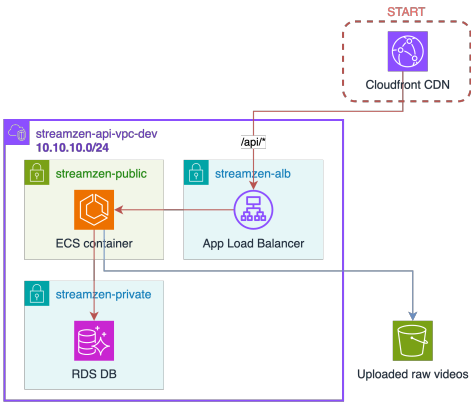
A kódbázis GitHubon kerül verziókezelésre, ott a CI/CD-folyamatokat GitHub Actions segítségével automatizálom a szerveralkalmazás élesítésére, a React-alkalmazás statikus fájljainak feltöltésére. A Terraform segítségével az infrastruktúrát kód formájában kezelem, a Terragrunt – amely egy Terraform-kódkezelést segítő kiegészítő eszköz – pedig a Terraform modulokat kezeli, hogy a kódbázis ne legyen túl bonyolult.



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

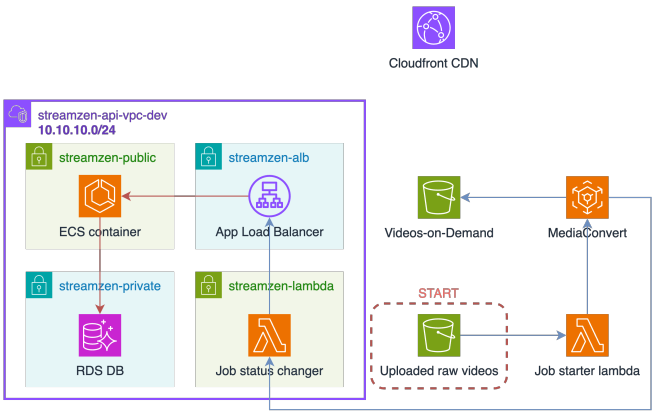
5.2.1. Video-on-Demand kiszolgálás folyamata

Vizsgáljuk meg közelebbről a VOD-tartalom kiszolgálásának folyamatát az AWS-felhőben. A 5.4. ábra mutatja be a VOD-tartalom feltöltésének folyamatát, ahol a szerveralkalmazás 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 szerveralkalmazás.



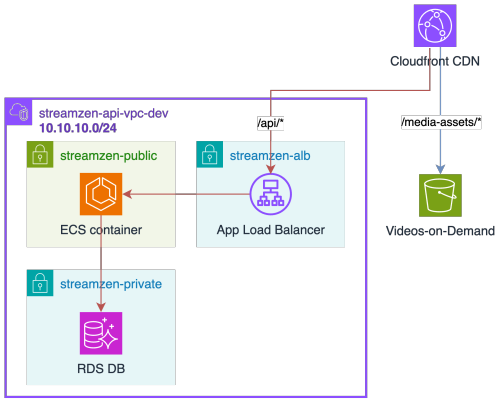
5.4. ábra. Folyamatábra a nyers videó feltöltéséről.

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-vödröt, 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.



5.5. ábra. Folyamatábra a feltöltés utáni videófeldolgozásról.

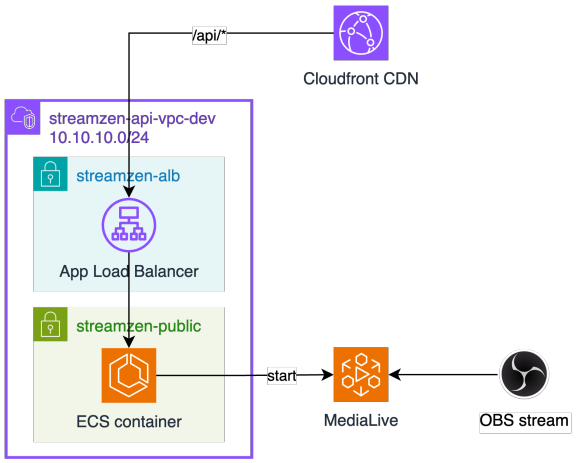
A 5.6. ábra fejti ki egy részről, hogy hogy értesül az adminisztrátor a videó feldolgozottságának állapotáról az API-n keresztül, illetve a UI-n értesülés után mi történik, ha meg is nyitja a már streamelhető videót, amely a CloudFront disztribúció Video-on-Demand S3-alapú originjén keresztül érhető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



5.6. ábra. Folyamatábra a VOD-tartalom lejátszásáról.

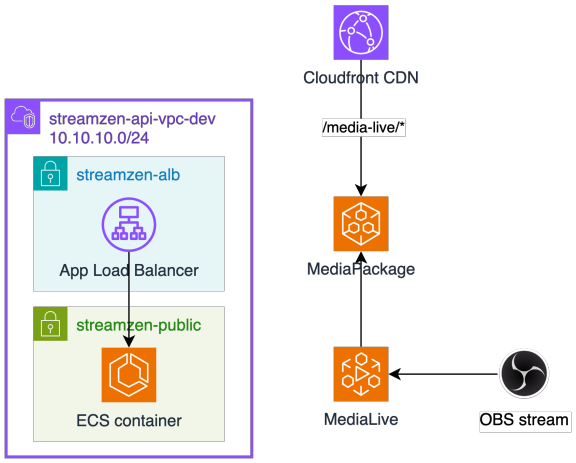
5.2.2. Live streaming folyamata

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.



5.7. ábra. Folyamatábra a live stream indításáról.

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.



5.8. ábra. Folyamatábra a live streamre való kapcsolódásról.

5.3. Konfigurációmenedzsment

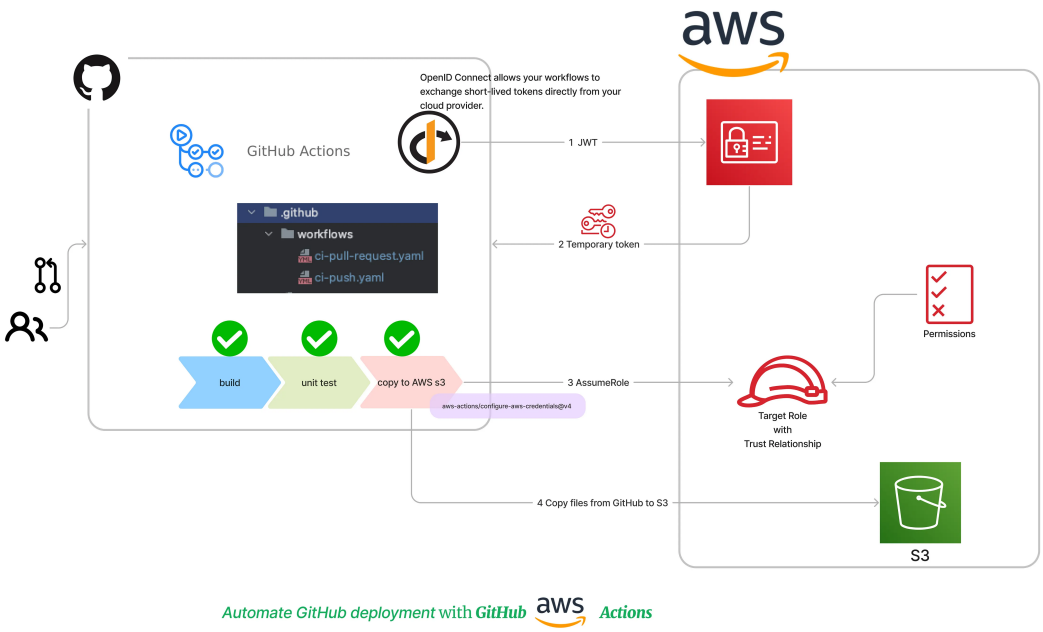
Nagyobb rendszerek tervezése igényli, hogy megfelelő Konfigurációmenedzsmentet teremtsen köré a tervezőmérnök, hogy a rendszer könnyen karbantartható legyen. A Terraform lehetővé teszi az infrastruktúra építését és az annak felkonfigurálását kód formájában, a kódban való élesítések nyomán friss és dokumentált 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 a Terraform tervek előnézetének generálását – amely a `terraform plan` parancs kiadasával 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 parancs-

soromból terveztem megtenni, tekintettel arra, hogy csupán egyedül dolgoztam a kódbázissal.

Hogy az AWS-fiók erőforrásaihoz hozzáférést kaphasson a munkafolyamat is, a GitHub Action munkafolyamata számára OpenID Connect (OIDC) felállításával terveztem az erre szánt AWS-szerepkör felvételét megvalósítani (5.9. ábra[2]). [9]



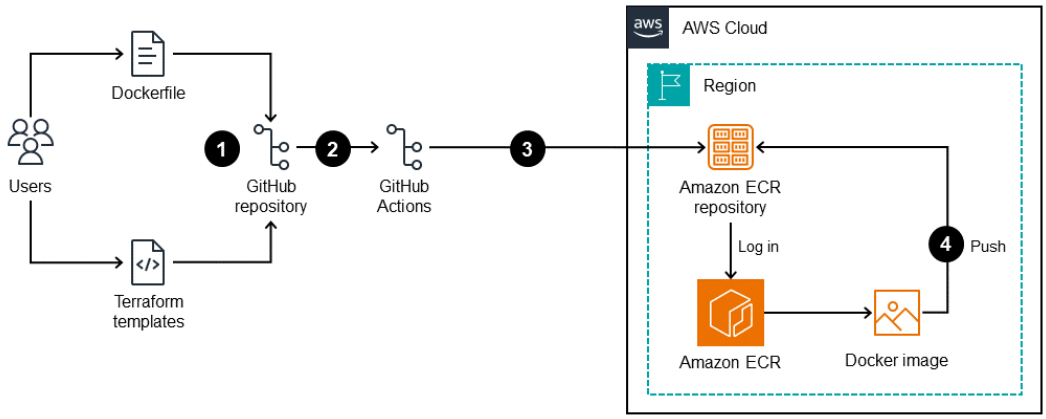
5.9. ábra. Workflow autorizálása AWS-szerepkörre OIDC-val.

A konfigurációmenedzsment részeként még a változtatások hibamentes élesítése érdekében statikus ellenőrzést terveztem bevezetni hasonlóan GitHub Action munkafolyamatokból a React-kód és a szerveralkalmazás Node.js-kódjára is.

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 az buildelt Docker-kép ECR-be való automatizált feltöltésére (5.10. ábra[3]), a React-kód buildelésére és S3-vödörbe való feltöltésére.

[2]A kép forrása: `https://mahendranp.medium.com/configure-github-openid-connect-oidc-provider-in-aws-b7af1bca97dd`

[3]A kép forrása: `https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/build-and-push-docker-images-to-amazon-ecr-using-github-actions-and-terraform.html`



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

5.4. A projekt felépítése

A projekt egészét egy közös GitHub-repositoryba terveztem kivitelezni, tehát „monorepo” jelleggel. A TypeScript-alapú projektekre bő eszköztárat és könnyű kezelést biztosít a Visual Studio Code (röviden VSCode[4]) szövegszerkesztő, amelyben a projekt könnyű átláthatóságára pedig egy VSCode-os (`streamzen.code-workspace` fájlnévvel) 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 találhatóak a csővezetékre.

[4] `https://code.visualstudio.com/`

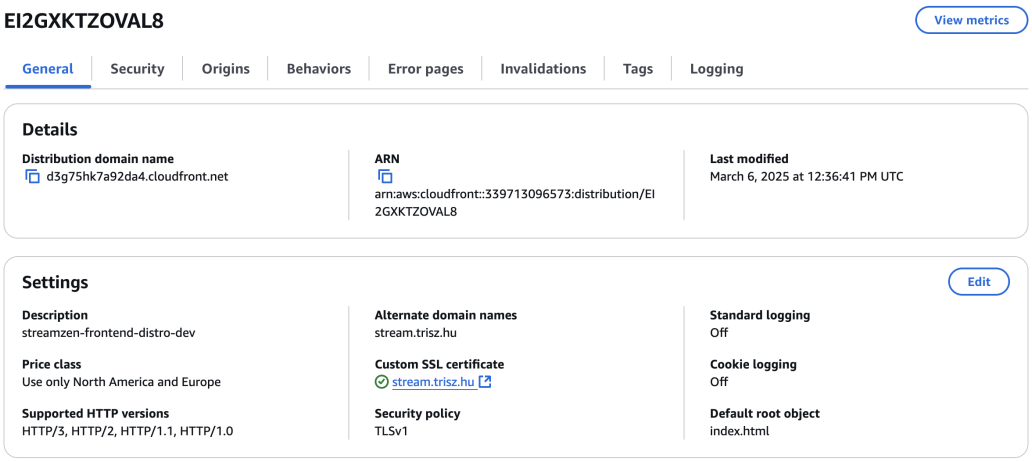
6. fejezet

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.

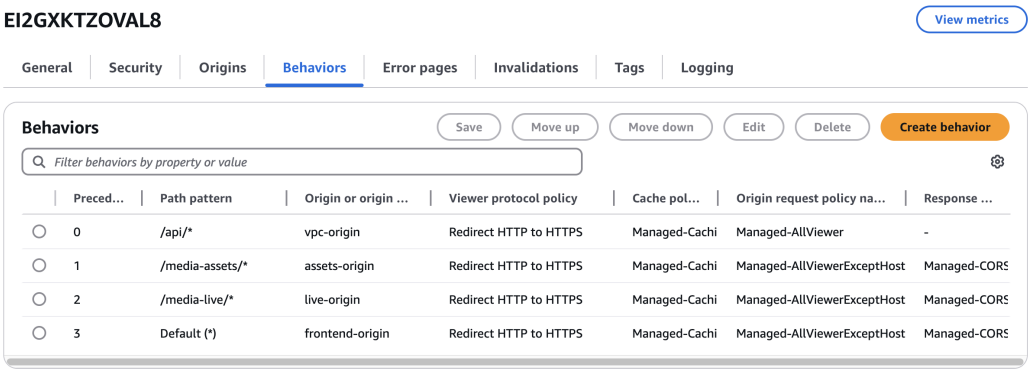
6.1. A CDN és a hozzácsatolt erőforrások

TODO: A CDN és az originjeinek tárgyalása, melyik origin mire való. VPC Origin tárgyalása, annak haszna. A WAF szerepe, a felkapcsolt cert és WAF web ACL szerepe.



6.1. ábra. Képernyőkép a disztribúció alapvető beállításairól az AWS-konzolon.

TODO: ez meg az.



6.2. ábra. Képernyőkép a különböző útvonalakra illesztett cache behavior-ökről.

```
function handler(event) {
  const request = event.request;
  ["/media-assets/", "/media-live/"].forEach((prefix) => {
    request.uri = request.uri.replace(prefix, "/");
  });
  return request;
}
```

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

6.2. A statikus weboldal

TODO: Az S3 bucket. A statikus website kiszolgálásának módja.

```
const spaInternalRoutingPrefixes = ["/videos", "/live", "/events", "/members", "/
    courses", "/about", "/studio", "/login"];
function handler(event) {
  const request = event.request;
  if (spaInternalRoutingPrefixes.some((prefix) => request.uri.startsWith(prefix))
    ) {
    request.uri = "/";
  }
  return request;
}
```

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

```
module "api" {
  source = "./modules/api-stack"
  environment = var.environment
  vpc_id = module.vpc.vpc_id

  alb_tg_port_mapping = 80
  alb_secgroup_ids = [module.vpc.secgroups["streamzen-alb-sg"].id]
  alb_subnet_ids = [module.vpc.subnets["streamzen-alb-1a"].id, module.vpc.subnets
    ["streamzen-alb-1b"].id]
  alb_internal = true # does not need to be internet-facing
  db_secgroup_ids = [module.vpc.secgroups["streamzen-db-sg"].id]
  db_subnet_ids = [module.vpc.subnets["streamzen-private-1a"].id, module.vpc.
    subnets["streamzen-private-1b"].id]
  api_secgroup_ids = [module.vpc.secgroups["streamzen-api-sg"].id]
  api_subnet_ids = [module.vpc.subnets["streamzen-public-1a"].id, module.vpc.
    subnets["streamzen-public-1b"].id]
  api_subnet_route_table_ids = [for s in values(module.vpc.subnets) : s.
    route_table_id]

  ecs = {
    family_name = "streamzen-api"
    port_mapping = 80
    task_environment = {
      AUTHSCH_CLIENT_ID = data.aws_ssm_parameter.these["authsch-client-id"].value
      AUTHSCH_CLIENT_SECRET = data.aws_ssm_parameter.these["authsch-client-secret
    "].value
      POSTGRES_USER = data.aws_ssm_parameter.these["db-username"].value
      POSTGRES_PASSWORD = data.aws_ssm_parameter.these["db-password"].value
      POSTGRES_PRISMA_URL = "postgresql://${data.aws_ssm_parameter.these["db-
    username"].value}:${data.aws_ssm_parameter.these["db-password"].value}
    @streamzen-rds-dev.czw6iqm8461h.eu-central-1.rds.amazonaws.com:5432/streamzen
    ?schema=public"
      FRONTEND_CALLBACK = "https://${var.domain_name}"
      JWT_SECRET = data.aws_ssm_parameter.these["api-jwt-secret"].value
      AWS_S3_REGION = var.region
      AWS_S3_UPLOADED_BUCKET = "streamzen-uploaded-videos-${var.environment}-
    bucket"
    }
    memory = 512
    cpu = 256
    desired_task_count = var.enable_ecs ? 1 : 0
  }
  db = {
    engine = "postgres"
    engine_version = "16.4"
    instance_class = "db.t3.micro"
  }
}
```

6.3. kódrészlet. API stack moduljának felparaméterezése a main.tf fájlban.

6.2.1. A React alkalmazás fejlesztése

TODO: ejlesztés részletei. Nem annyira lényeges most ebben a dolgozatban, de néhány szép kihívást jelentő kidolgozott form és egyebeket be lehet itt mutatni.

6.2.2. A weboldal telepítésének CI/CD folyamata

TODO: GitHub Actions a smoke tesztekre, workflowk, a deploymentek. Értsd itt: S3 telepítés.

6.3. Média erőforrások objektumtárolói

TODO: Hogy lettek felkonfigurálva és miért az egyes S3 bucket-ok (bucket policy, CORS policy).

6.4. A MediaLive és MediaPackage bekötése

TODO: Az Elemental stack részeinek felkonfigurálása, a MediaLive channel és a MediaPackage channel felépítése, a MediaPackage endpoint konfigurálása. Miként kerül kiszolgálásra, melyiket mire használom. OBS bekötésének módja.

7. fejezet

Szerver oldali 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.

7.1. A virtuális privát felhő komponensei

TODO: A VPC-beli (Virtual Private Cloud) subnetek, a security groupok, route táblák. A biztonság vizsgálata.

7.2. A Node.js alkalmazás fejlesztése

TODO: Az elkészült alkalmazás felépítése, a különböző rétegek, a routing, a middleware-ek, a kontrollerek, a service-ek, a REST API.

TODO: Itt lehet szó az adatbázisbeli entitásokról is.

TODO: Kódrészletei. Kitérve arra, hogy miképp könnyíti a munkát a Prisma, milyen egyéb szolgáltatások kerültek be, mi a felépítése a reponak, miért választottam ezt a stacket.

TODO: S3 bucketba való mentése a videónak egy érdekes rész.

7.3. A konténerizált környezet

TODO: Leírás, hogy miért választottam a konténerizált környezetet, a konténerizálás előnyeit, hátrányait. Hogy használható ki a legjobban a konténerizáció az Applica-

tion Load Balancer-rel együtt. Miképp kapcsoltam ezt a kettőt össze (ECS service, ALB).

7.3.1. A Node.js szerveralkalmazás ECS-en

TODO: Az ECS orkesztrációs toolsetjének kialakítása, a konténer rétegződés felépítése ECS-ben, a konténer registry (ECR) bekötése. Környezeti változók, portok, ALB-re való kötése. Miből állt a dockerizálás nekem (Dockerfile, registry, image build, push, networking).

TODO: Milyen IAM role-okat kellett feltenni rá, mikkel kommunikál kifelé, mi indokolta, hogy publikus subnetbe kerüljön. Hogy hív meg más külső rácsatlakozó erőforrásokat (S3 bucket, RDS instance, Lambda függvény, MediaLive channel).

TODO: Jumpbox használata, illetve miért került ki, hol volt egy hibázás a Dockerfile-lal.

7.3.2. A szerveralkalmazás CI/CD folyamatai

TODO: GitHub Actions a smoke tesztre, a deploymentre: Docker build és ECR-be telepítés.

7.4. Elemental MediaConvert felhasználása

TODO: Az Elemental MediaConvert API használata a Lambdából, illetve hogy hogy hívódik meg a Lambda, milyen triggerrel, milyen környezeti változókkal, milyen IAM role-al. Hogy kellett felkonfigurálni a MediaConvert job, hogy kellett magát a Lambdát felkonfigolni, hogy tudja is hívni.

8. fejezet

Tesztelés és mérés

A média streaming szolgáltatások fejlesztése és üzemeltetése során a mérési eredmények alapján lehet a legjobban optimalizálni a rendszert, kiszámolni a költségeket, és a felhasználói élményt folyamatosan javítani. Jelen fejezetben megvizsgáljuk közelebbről egyszerűbb tesztesetek segítségével, milyen eredményekre lehet számítani egy AWS-felhő alapú streaming szolgáltatás fejlesztése után.

8.1. Az infrastruktúra terhelése

TODO: Szimpla load tesztelés összeállításáról fogok itt értekezni, és a mérési eredmények kiértékelése.

8.2. Népszerű szolgáltatók metrikái

TODO: Keresni kell cikkeket Netflix blogban vagy valahol arról, a népszerű szolgáltatók milyen SLA-val, milyen statisztikával dolgoznak.

9. fejezet

Összegzés

TODO: Tanulságok, a rendszer működésének és fejlesztési élmények értékelése. Média streaming jövője saját meglátások szerint.

9.1. Továbbfejlesztés lehetőségei

TODO: Min lehetne javítani, mikroszolgáltatásos architektúra stb.

9.1.1. Vendor lock-in jelensége

TODO: Miként befolyásolja egy üzlet működését a vendor lock-in, és hogyan lehet ezt kezelni. Miképp lehetne a jövőben a vendor lock-in-t csökkenteni ebben a rendszerben. S3 helyett Ceph, konténert kiemelni, akár K8s-re felkészíteni, hogy ne ECS-től függjön.

Irodalomjegyzék

[1] Apple Inc.: HTTP Live Streaming | Apple Developer Documentation. https://developer.apple.com/documentation/http-live-streaming. [Hozzáférés dátuma: 2025-03-24].

[2] 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.

[3] 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/.

[4] Frequent Questions | WebRTC. https://webrtc.github.io/webrtc-org/faq/. [Hozzáférés dátuma: 2025-03-24].

[5] 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/.

[6] ISO Central Secretary: Information technology – Dynamic adaptive streaming over HTTP (DASH) – part 1: Media presentation description and segment formats. ISO/IEC 23009-1:2022. Standard, Geneva, CH, 2022, International Organization for Standardization / International Electrotechnical Commission. URL https://www.iso.org/standard/83314.html.

[7] Csaba Kopiás: Ffmpeg - the ultimate guide. Jelentés, 2022. https://img.ly/blog/ultimate-guide-to-ffmpeg/.

[8] 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/.

[9] 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/.

[10] Sydney Roy: What is CMAF? Jelentés, 2022, Wowza Media Systems, LLC. https://www.wowza.com/blog/what-is-cmaf.

[11] Servers.com: The history of streaming told through protocols, 2023. https://www.servers.com/docs/whitepaper/history_of_streaming_told_through_protocols.pdf.

[12] Hezbullah Shah – Tariq Soomro: Node.js challenges in implementation. *Global Journal of Computer Science and Technology*, 17. évf. (2017. 05), 72–83. p.

[13] 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.

[14] Christer Whitehorn: Choosing the right aws live streaming solution for your use case. Jelentés, 2021, Amazon Web Services. https://aws.amazon.com/blogs/media/awse-choosing-aws-live-streaming-solution-for-use-case/.