AWS VIDEO STREAMING WEB SERVICE GUIDE

SHAFI UDDIN

INTRODUCTION

The aim of this project is to build a serverless video streaming and sharing service with Amazon Web Services (AWS) infrastructure to provide fast, scalable video transcoding and hosting capabilities.

Key features:

- Secure video transcoding
- Scalable content delivery

Tech Stack:

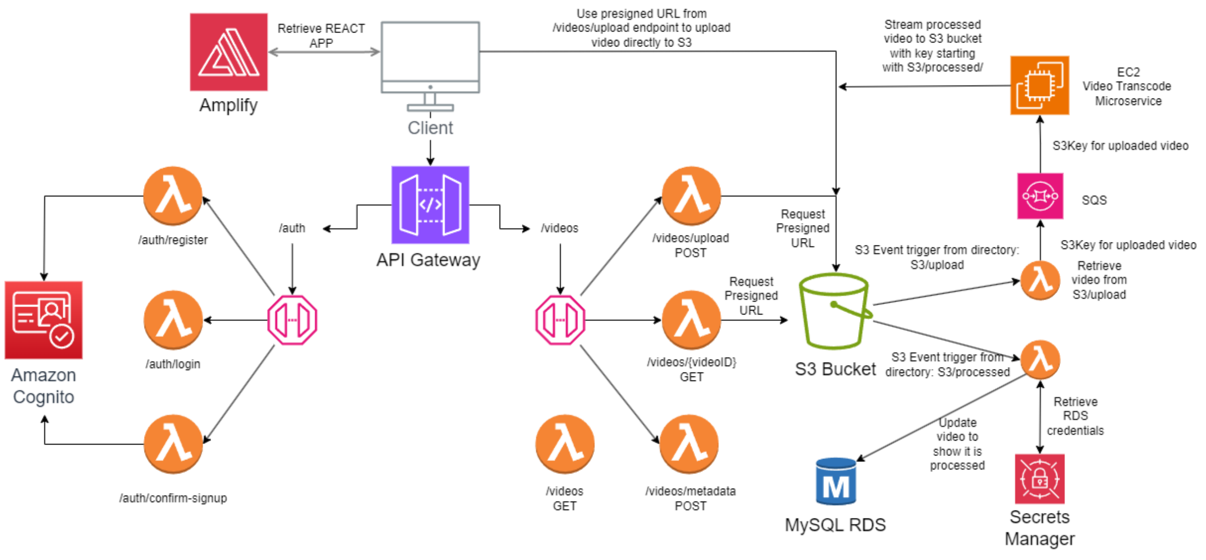
- **Frontend:** React
- **Backend:** Node.js, Lambda, API Gateway, EC2, SQS, Docker
- **Identity Management:** Cognito
- **Data storage:** MySQL, S3
- **CDN:** Cloudfront (through Amplify)
- **CI/CD:** Amplify
- **DevOps:** Cloudwatch, Secrets Manager, IAM, GitHub

A serverless architecture model, with decoupled services is used wherever possible, by leveraging AWS infrastructure. This allows for a service that can be easily scaled and delivered to consumers seamlessly whilst ensuring security and reliability.

ARCHITECTURE

A serverless and microservices architecture is used to deliver an easily scalable and secure service. This approach promotes separation of concerns, where each component of the architecture is modular and decoupled.

Architecture Layout



API Overview

API gateway in this web service uses a REST API architecture to ensure reliable and secure functionality of client facing services. The API follows the below architecture:

Purpose	Resource		Request Type	Authentication Required?
All authentication related endpoints.	/auth			
After registering, users receive a onetime code via email to confirm their registry to the service.		/ confirm-register	POST	No
Allows user to log in to the system and they receive web tokens that allow them to access endpoints that require authentication.		/login	POST	No
Allows user to register with the service. A onetime code is sent to the provided email to confirm their registry.		/register	POST	No
All authentication related to videos on the web service. GET request allows all videos to be fetched from RDS to allow users to choose which video they want to watch.	/videos		GET	NO
Receive a presigned S3 URL to watch a particular video, along with all relevant video metadata.		/{videoID}	GET	NO
Receive a presigned URL to directly upload a video to S3 from the client system.		/upload	POST	YES
Post all metadata for the videos (separately) to RDS.		/metadata	POST	YES

RDS Table Schema

The current RDS schema for the MySQL service is designed to be as simple as possible with the currently simple requirements of the video streaming application. The current schema is as follows:

Column Name	**Data Type**	**Constraints**	**Description**
videoKey	VARCHAR(255)	PRIMARY KEY, NOT NULL	Unique identifier for each video.
displayName	VARCHAR(255)	NOT NULL	Human-readable name or title of the video.

uploadTime	DATETIME	NOT NULL	Timestamp indicating when the video was uploaded.
isTranscoded	TINYINT(1)	NOT NULL, DEFAULT 0	Flag indicating whether the video has been transcoded (0 = No, 1 = Yes).
email	VARCHAR(255)	NOT NULL	Email address of the uploader or associated user.

Summary of how cloud services are used

Blob Storage	S3	Used to store video data
SQL storage	RDS (MySQL)	Used to store video metadata
CDN	CloudFront	Used to manage distribution and caching for front end react service
Server compute	EC2 with Docker	Used to host transcoding microservice. Containerised with Docker to ensure consistency with micro service
	Lambda	Used to execute simple functions to manage back-end infrastructure
API management	API Gateway	Used to manage all communication between client and all back-end services.
Queue System	SQS	Used to retain videos to be transcoded, so they can be processed by the transcoder microservice when it is available.

Measures to Ensure Service Resilience and Persistence

Although the entire service is designed a Serverless and decoupled architecture, some measures are still taken to ensure the resilience of the service in the event of an unforeseen crash of a service.

1. **Front end React resilience against failed API calls**
   React frontend has been designed to handle **failed API calls** gracefully, by retrying requests or displaying appropriate error messages when a service is unavailable.
2. **Backups of MySQL RDS instance**
   Automated backups are enabled for the MySQL RDS instance, ensuring data persistence even in case of failures.
3. **Creation of Amazon Machine Image (AMI) of transcode service**
   To quickly start up an EC2 instance for the transcode microservice, an AMI of the EC2 instance was generated, allowing the service to be brought up quickly, requiring set up of the service to only be done once.
4. **Docker command to restart Docker container in the event of a crash**
   A Docker command has been implemented to automatically restart the transcoding service container in case of a crash.

CHALLENGES AND SOLUTIONS

Several challenges were encountered during development, and a systematic approach of isolating these issues was used such as through console logs, and debugging was adopted:

1. **Authentication issues with AWS services:**
    - **Problem:** AWS services would often fail due to authentication issues when connecting services such as Lambda and RDS, or S3.
    - **Solution:** Allowing IAM role used by Lambda function to access the RDS and S3 instances.

2. **Docker container crashing during AMI creation:**
    - **Problem:** The docker container running the transcode microservice would crash when the EC2 AMI was created. This means upon every boot up of an EC2 instance, the transcode microservice would crash upon startup.
    - **Solution:** The Docker `--restart always` flag was used to ensure the microservice starts up every time an EC2 instance is brought online.
3. **Unable to fetch data from Secrets Manager**:
    - **Problem**: The relevant Lambda functions are designed to securely fetch RDS credentials from secrets manager. This design decision was made to ensure such sensitive details are secured with IAM roles, preventing them from being hardcoded into Lambda code or as environment variables. It also allows for a centralised management system of credentials so they may be securely rotated, changed, and yet still be fully and seamlessly accessible from all subscribing services.
    - **Current Issue:** This aspect of the architecture is not functional and must be fixed in future revisions of the web service. The current issue being faced is secrets manager not returning any credentials to subscribing lambda functions, despite the lack of returned errors.
    - **Temporary Solution:** Usage of environment variables for relevant lambda functions.

FUTURE IMPROVEMENTS

Front-end - Client Facing Features

The current web service is designed to allow users to share and view video content. However, the user experience can be significantly enhanced with additional features such as:

- **Video description:** The ability to add a description to a video when posting to provide viewers additional context of the content.
- **Likes and dislikes:** The ability to like and dislike content to provide empirical feedback to content creators.
- **Commenting:** The ability for viewers to make comments to provide content creators direct feedback to creators, or to discuss the content with other viewers.
- **Notification system:** The ability for users to be notified when their video has been fully processed and is ready to be viewed, namely through in-application notifications (via AWS SNS) or emails (via AWS SES).

Back-end - Enhance Security and Reliability

CI/CD:

- Usage of AWS CodeDeploy, GitHub Actions or AppRunner to create a CI/CD pipeline for the transcode microservice. This microservice currently requires a direct interface with the EC2 system files to implement changes.

Orchestration:

- Useage of services such as Terraform or ECS to provide higher level management of the service to allow for greater cost saving measures. Allows for the service to be more flexible with regards to deployment and overall management of systems.

Auto-scaling:

- Use of auto scaler on EC2 instance to account for increased load, or to reduce the number of instances to 0, to ensure costs are incurred when the web service is idle. This can also be achieved by using AWS AppRunner which includes auto scaling as part of its regular service.

Edge caching:

- Use of edge caching to reduce the number of API calls, to save costs and allow for a more responsive application.

Revamped MySQL schema:

- Currently only uses one table to keep track of all application data. Additional tables can be used to track more data, such as additional metadata, comments, user related data, etc.