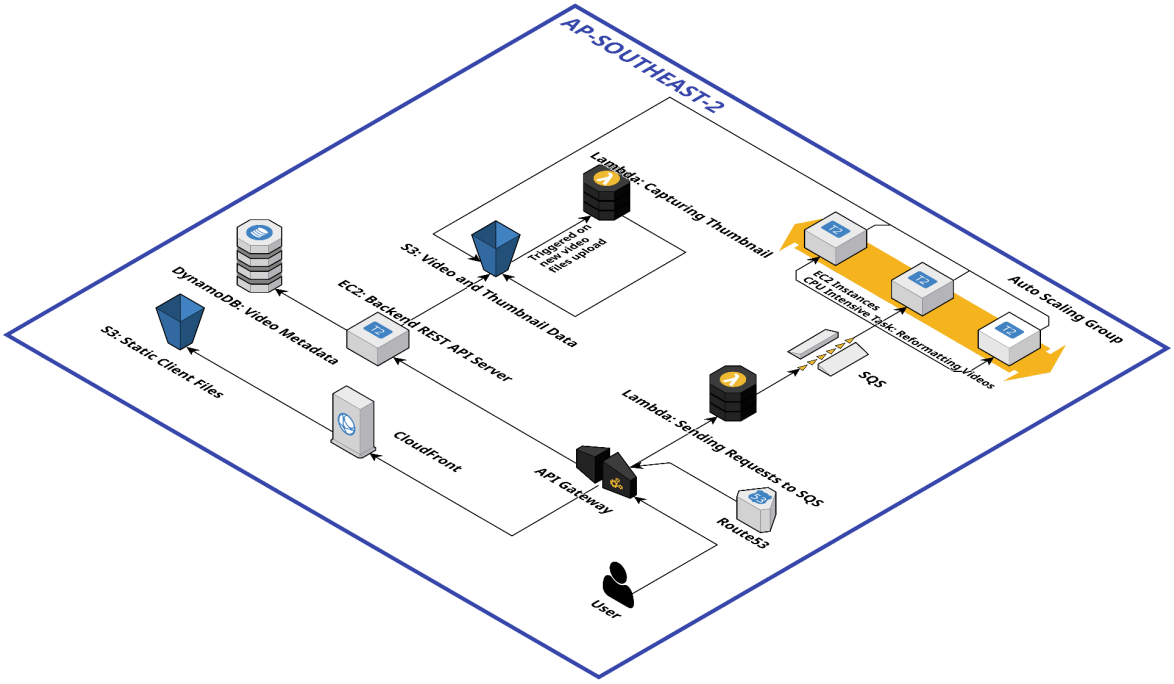
CAB432 Project Report

Dennis Chen - n12150801

Application overview

The application is a comprehensive cloud-based video management platform designed to handle video uploads, processing, and content delivery. Users access the platform through a custom domain managed by Route 53, which routes traffic to an API Gateway that directs requests to various backend services. The main backend REST API server is hosted on an EC2 instance and it handles most of the core functionality of the application such as user authentication, fetching users' video libraries, and managing video uploads. Uploaded videos are stored in S3 buckets, along with their corresponding thumbnails and reformatted versions. Video metadata is stored in DynamoDB, ensuring scalable and efficient storage of key information. The application integrates with Lambda functions for event-driven tasks, such as automatically generating video thumbnails when new video is uploaded to S3. CloudFront is used to deliver static frontend assets and ensure low-latency access to the client interface, providing users with a seamless experience. For video reformatting tasks, incoming requests are queued in SQS and processed by EC2 instances in an auto-scaling group, which allows the system to dynamically handle varying workloads. Once videos are reformatted, they are uploaded back to S3 for storage and retrieval. This architecture provides a scalable, reliable, and efficient system for managing user videos and delivering a smooth user experience.

Application architecture



View the application architecture from this link here.

User Access:
Users access the web application via a custom domain name managed by Route 53.

Route 53 and API Gateway:
Route 53 is configured to route traffic to the API Gateway's domain name.

API Gateway Integration:
The API Gateway routes requests to three key components:

- **Backend REST API** hosted on an EC2 instance.
- **Frontend static client** delivered through CloudFront.
- **Lambda function** that processes requests and sends messages to SQS.

Backend Server:
The EC2 instance handles most of the application's requests, which includes login, getting user's videos and uploading of videos.

DynamoDB:
DynamoDB stores metadata for the uploaded and reformatted video files.

S3 Buckets:

- **Video Storage**: S3 contains the original video files, thumbnails, and reformatted videos.

- **Static Client Files**: Another S3 bucket holds the static assets for the frontend, delivered via CloudFront.

Lambda Function for S3 Trigger:
Another Lambda function is triggered when a new video is uploaded to S3. This function generates a thumbnail for the video and saves it back to S3.

CloudFront and S3:
CloudFront is linked to the S3 bucket containing the frontend static files, enabling content delivery.

SQS and EC2 Auto Scaling Group:
SQS queues incoming requests that are processed by the EC2 instances in an auto-scaling group. Once the video reformatting process is complete, the EC2 instance uploads the reformatted video to S3.

Justification of Architecture

Choice of Microservices

The current services are the main backend REST API server, CPU intensive process - video transcoding service, capturing thumbnail service and queuing of message services.

The main backend REST API server handles requests pertaining to user interactions or databases. Having this as a microservice makes it such that this is the only compute resource that users of the application are generally communicating with and nothing else. This service requires reliability as it is the core of the entire architecture, thus it should be separated from other services such as the video transcoding service which might take up compute resources and cause the main REST API server to not be able to function properly.

Next, having a microservice that is dedicated to the video transcoding service is because it requires a lot of computational power. This microservice is the only service that is being scaled up horizontally in the current architecture and this is suitable because video transcoding service is the only process in this application that takes up a longer time and requires more resources. By isolating this process from other services, we ensure that this service does not affect the functionality of other services.

The decision to have a service dedicated to capturing thumbnails is because it does not have any relationship to either users or video transcoding. Additionally, this separation ensures that thumbnail creation, a relatively lightweight and fast operation, does not interfere with the more resource-intensive video transcoding process.

Lastly, having a message queuing service in the architecture stems from the need to decouple the flow of communication between different components and to handle asynchronous tasks effectively. In an application with CPU-intensive processes like video transcoding, directly processing tasks synchronously can lead to performance bottlenecks, system overload, and

poor user experience. A message queue service helps mitigate these risks by introducing controlled and orderly processing.

Choice of Compute

The decision to use EC2 instead of ECS for the backend REST API server was because the current architecture does not rely heavily on containers. The current backend is still largely monolithic and could potentially still be broken down into smaller services such as a service for user authentication or a database service.

The video transcoding service uses EC2 as video processing involves handling large files and intense computations over extended periods of time. ECS would require careful management of resources, including container memory, CPU allocation, and storage requirements, which can lead to additional complexity. EC2 provides a more straightforward approach, where the instance type and storage configuration can be tuned specifically for the video processing needs without worrying about container orchestration overhead.

For capturing thumbnails and sending messages, both are lightweight processes which can be completed within a few seconds and are event driven which makes it suitable for Lambda. Lambda functions are suitable because they execute based on these events and do not require any persistent compute resources. This eliminates the need for maintaining container instances or worrying about idle resources in ECS or EC2.

Choice of Load Distribution Mechanism

For this architecture, I have chosen to use Amazon Simple Queue service in order to handle the load distribution video transcoding requests to the EC2 Auto Scaling group. The choice to use SQS instead of traditional load balancing is mainly due to the video transcoding process can be time consuming and computationally expensive especially when video files are large. As such at any point of time, each instance should only handle a single video transcoding request at a time in order to ensure optimal performance and avoid overload to the computation resources. SQS is suitable in this case as it stores all the requests inside the queue which the video transcoding microservice would be the one who gets the requests from the queue. Only after each instance finishes the current video transcoding requests would they be able to get another message from the queue, unlike traditional load balancing which just pushes the video transcoding requests straight to the instances and does not wait for current processes to finish. Thus, SQS helps to keep the workload manageable for all instances within the Auto Scaling group.

Choice of service abstraction

API Gateway makes it easy to integrate the client requests with backend microservices as it enables seamless routing to these services and a single point of entry. This streamlines communication between the frontend and backend and enables users to not worry about handling requests, or cross origin resource sharing problems. Furthermore, API Gateway has security features such as authentication and rate limiting without having the need to manually implement it.

Project Core - Microservices

- **First service functionality:** Public facing REST API server
- **First service compute:** EC2 instance - i-094944d2063ed5f34
- **First service source files:**
    - Assignment 3/server
- **Second service functionality:** CPU Intensive task, video transcoding
- **Second service compute:** EC2 instance - i-0b522e1841558e89e (Image is build on this EC2 instance to be used for the auto scaling group)
- **Second service source files:**
    - Assignment 3/scale
- **Video timestamp:** 0:23

Project Additional - Additional microservices

- **Third service functionality:** Capturing thumbnail from uploaded video
- **Third service compute:** Lambda - n12150801-captureThumbnail
- **Third service source files:**
    - Assignment 3/lambda-captureThumbnail
- **Fourth service functionality:** Sending request to SQS
- **Fourth service compute:** Lambda - n12150801-sendMessage
- **Fourth service source files:**
    - Assignment 3/lambda-queue
- **Video timestamp:** 1:20

Project Additional - Serverless functions

- **Service(s) deployed on Lambda:** Capturing thumbnail from uploaded video, and sending reformat requests to queue
- **Video timestamp:** 1:20
- **Relevant files:**
    - Assignment 3/lambda-captureThumbnail
    - Assignment 3/lambda-queue

Project Additional - Container orchestration with ECS (Not attempted)

- **ECS cluster name:**
- **Task definition names:**
- **Video timestamp:**
- **Relevant files:**

Project Core - Load distribution

- **Load distribution mechanism:** SQS
- **Mechanism instance name:** n12150801-assessment
- **Video timestamp:** 2:10
- **Relevant files:**
    - Assignment 3/server/aws/sqs.js
    - Assignment 3/scale/aws/sqs.js
    - Assignment 3/lambda-queue/sqs.mjs

Project Additional - Communication mechanisms (Not attempted)

- **Communication mechanism(s):** [eg. SQS, EventBridge, …]
- **Mechanism instance name:** [eg. n1234567-project-sqs]
- **Video timestamp:**
- **Relevant files:**

Project Core - Autoscaling

- **EC2 Auto-scale group:** n12150801-autoscale-assessment
- **Video timestamp:** 5:13
- **Relevant files**
    - Assignment 3/scale

Project Additional - Custom scaling metric (Not attempted)

- **Description of metric:** [eg. age of oldest item in task queue]
- **Implementation:** [eg. custom cloudwatch metric with lambda]
- **Rationale:** [discuss both small and large scales]
- **Video timestamp:**
- **Relevant files:**

Project Core - HTTPS

- **Domain name:** n12150801.cab432.com
- **Certificate ID:** 061c1139-a871-42de-850d-8312830fe30b
- **ALB/API Gateway name:** API Gateway - n12150801-api-assessment
- **Video timestamp:** 4:00
- **Relevant files:**
    - Assignment 3/client -> routed to /
    - Assignment 3/server -> routed to /api
    - Assignment 3/lambda-queue -> routed to /api/reformat-queue

Project Additional - Container orchestration features (Not attempted)

- **First additional ECS feature:** [eg. service discovery]
- **Second additional ECS feature:**
- **Video timestamp:**
- **Relevant files:**

Project Additional - Infrastructure as Code (Not attempted)

- **Technology used:** [eg. CloudFormation, Terraform, …]
- **Services deployed:** [eg. ALB, SQS,... Only Block 3 services need to be listed]
- **Video timestamp:**
- **Relevant files:**

Project Additional - Edge Caching

- **Cloudfront Distribution ID:** E3KB31PXOXCXW2
- **Content cached:** Client build with create react app
- **Rationale for caching:**
    - **Reduce server load**: When static assets are cached, they no longer need to be fetched directly from the origin server for every request.
    - **High Availability**: Edge caching ensures content is available even if the origin server has downtime or experiences heavy load, improving the reliability and uptime of the application.
    - **Limited Changes**: The client is built with static files that are not updated frequently, making it suitable for caching as the content remains consistent for long periods.
    - **Accessed Frequently**: The client will be accessed frequently, making edge caching ideal for reducing load times and delivering content to users efficiently.
- **Video timestamp:** 2:55
- **Relevant files:**
    - Assignment 3/client

Project Additional - Other (with prior permission only) (Not attempted)

- Description:
- Video timestamp:
- Relevant files:

Cost estimate

The total cost estimate can be found here.

	AWS Service	Upfront Cost (USD)	Monthly Cost (USD)
1	Amazon EC2	0	25.93
2	Amazon Cognito	0	8.26
3	Amazon API Gateway	0	1.29
4	Amazon Simple Storage Service	0.01	2.50
5	Amazon DynamoDB	205.20	15.43
6	Amazon Route 53	0	3.00
7	Amazon Simple Queue Service	0	1.14
8	AWS Lambda	0	0
9	Amazon Cloudfront	0	2.00

Scaling up

The current design of video transcoding application is still considered to be rather inflexible and monolithic as the most of the application core functionality and logic in the main backend REST API server is being handled by a single EC2 instance. This can be rather problematic when scaling up as it is very likely to create bottlenecks when it is under heavy traffic, since services like user authentications, getting user's video metadata and more are all being performed in a single EC2 instance. In order to address this problem, I should look to separate the main REST API server into smaller, independent services. Each service will only be concerned with their own distinct functionality such as user authentication, or getting video's metadata. As such, by transitioning into even more of a microservices architecture, this should improve the fault resilience of the entire application, since the failure of one service does not result in the failure of the entire system.

Furthermore, each service should be linked to its own unique application load balancer and is able to be scaled up based on their current demand and this should improve resource usage efficiency. There would also be better control over distributing traffic as in the current application architecture, if I were to scale up the main REST API server, this will scale up all services within the REST API server, instead of just a specific service that is experiencing heavier traffic at that point of time.

Some services such as user authentication or video thumbnail generation which does require much computing power can be implemented with a serverless architecture using AWS Lambda. This is beneficial as Lambda functions are cost effective for such workloads and they are able to scale up automatically without having the need to set up an auto scaling group. For services that require more computing power such as video reformatting, instead of using an auto scaling group which takes time to start up EC2 instances, we should aim to containerise these services and deploy them using Amazon ECS. Using containers not only makes better use of compute resources, it also is faster at deploying since adding and removing containers is faster than starting and terminating EC2 instances.

Other than that, the way that services are scaled up should no longer just look at basic CPU utilisation. The current system employs a target tracking scaling based on average CPU usage, but this should be changed to implement predictive scaling instead. By looking at past traffic patterns and learning from these patterns, the application should be able to anticipate when it is likely to have experienced more traffic and automatically adjust the infrastructure capacity ahead of time, reducing delays or service interruptions during these peak periods.

Lastly, the current set up for load distribution may not be suitable to handle 10,000 concurrent users. For instance, there is currently only one queue being used to manage all the video processing tasks, but this could be improved by using multiple SQS queues. This ensures that no single queue becomes overwhelmed. I should also look to increase the API Gateway throttling limits in order to ensure that it can handle the expected high volume of traffic.

Security

To secure the application in a commercial setting and mitigate the risks of cyberattacks, several cloud security principles would be applied.

Understand risks, roles and entities

The first step in securing the application is understanding what are the entities of the application and what are their responsibilities within the application. As such, we are able to understand and identify where there could be potential security breaches within the application as it will most likely only occur at the points whereby these entities interact with the application. This allows us to conduct risk assessments regularly to identify vulnerabilities within the cloud environment such as attacks to S3 buckets or API endpoints. Furthermore, when we have identified what are the responsibilities of the entities, we are able to provide role-based access control to the users, giving them authorization only to services that they require to do their job.This minimises access to services and reduces the exposure of these services.

Least Privilege

The principle of least privilege is to give only the privileges necessary for an entity to do its job. This will protect against any entities that have been compromised by attacks by external parties and limits what these attackers are able to reach. To improve security of cloud applications, we should use AWS Identity and Access Management roles to assign entities authorisation to specific services that they require to do their job. EC2 instances, Lambda functions and even users should only be able to access specific S3 buckets, or DynamoDB tables that they need for their job. A way to implement this in my current application is in my capture thumbnail Lambda function, it should only have write permissions to the S3 bucket where thumbnails are stored and read access to the video file bucket. Another way that I have implemented the principle of least privileges is to utilise S3 presigned URLs. What these URLs do is that they provide temporary access to a specific object within the S3 bucket, and entities must complete their tasks that require them to interact with the object within this time limit. This minimises the risk of unauthorised access and helps enforce the principle of least privilege.

Separation of Concerns

The principle of separation of concerns is about building systems with small entities and each of these entities have a job that is distinct to them. As such, this provides the opportunity to apply the principle of least privilege. For my cloud application, I have attempted to adopt a microservices architecture that enforces the principle of separation of duties by isolating each service to handle a specific function. For instance, the backend EC2 instance focuses solely on handling user requests, CloudFront delivers the static frontend, and Lambda functions handle specific tasks like video processing and SQS interactions. This modular approach ensures that each aspect of the application is independently managed, reducing the risk of cross-service failures. In terms of permissions, each service is responsible for different tasks that require distinct permissions. For example, S3 handles video storage and thumbnails, while DynamoDB

manages metadata storage. By separating these functions into distinct components, the system ensures that different responsibilities are clearly divided, reducing the chance that a vulnerability in one service could affect another. Logging and monitoring systems are also kept separate, preventing any service that can modify system data from tampering with logs, further reinforcing the security architecture. This separation also allows independent scaling, so services that require more resources can be adjusted without impacting others.

Defence in Depth

The principle of defence in depth is about reducing the chances that a single mistake or bug within the application can result in a successful attack to the entire cloud application. Hence, in order to achieve this principle, multiple layers of security should be applied to protect the application. At the perimeter, API Gateway provides a first layer of defence by validating incoming requests and blocking malicious traffic through Web Application Firewall rules. For the backend services, the EC2 instances reside within a Virtual Private Cloud, with strict security group rules limiting inbound and outbound traffic to only required ports and IP ranges. Network security is further strengthened with private subnets for sensitive services like DynamoDB, ensuring that they are not directly accessible from the internet. Additionally, S3 presigned URLs offer another layer of security for accessing media files, as they provide controlled access without needing to make S3 buckets public or granting long-term access. Only authenticated requests or authorised users via API Gateway can request presigned URLs, further securing the workflow. Moreover, encryption at rest is enforced for data stored in S3 and DynamoDB using AWS-managed keys, and TLS ensures secure transmission of data.

Security by Default

The principle of security by default is about always taking the most secure option that is available when given a choice. AWS services are configured with secure defaults to minimise the risk of vulnerabilities. For instance, all S3 buckets containing sensitive data, such as video files and user metadata, have public access blocked by default, ensuring that only authenticated and authorised requests can access them. S3 presigned URLs are also generated in a secure manner, granting temporary and limited access to specific objects, further reducing exposure. Access to resources like Lambda functions and DynamoDB is controlled through tightly scoped IAM roles and policies, ensuring that only the minimum required permissions are granted. All API endpoints are protected by default, with API Gateway enforcing authorization checks using AWS Cognito for user identity management. By default, HTTPS is enabled for all communications, ensuring that data in transit between the user, API Gateway, and backend services is encrypted, mitigating the risk of data interception. Additionally, CloudTrail logs are enabled by default, capturing all API calls and activities for auditing and anomaly detection. This ensures that all interactions with AWS services can be monitored and reviewed, providing a strong layer of visibility into the system's operations and security events.

Sustainability

Software Layer

At the software layer, reducing compute, storage, memory, and network requirements directly contributes to sustainability. JavaScript code can be optimised to minimise computation needs by ensuring efficient algorithms and avoiding unnecessary loops or re-computations. Minimising the number of API calls and adopting GraphQL instead of REST APIs can reduce network traffic, as GraphQL enables fetching only the necessary data. To lower RAM usage, care can be taken to optimise memory allocation, such as reducing the size of variables, cleaning up unused objects, and minimising large in-memory datasets. Additionally, caching mechanisms (both client-side and server-side) can significantly reduce network usage and computation by preventing redundant requests. Compressing video files before storage and transferring only essential data across the network can also reduce both storage and bandwidth requirements.

By adopting compiled languages for certain performance-critical backend components (e.g., WebAssembly for parts of the application that require heavy processing), we can reduce CPU cycles and memory usage. Furthermore, edge caching using AWS CloudFront reduces the load on central servers, lowers network latency, and minimises the volume of data travelling across the network.

Hardware Layer

At the hardware level, utilising efficient processor architectures like ARM or AMD EPYC for EC2 instances could provide better performance-per-watt than using older or less efficient CPUs. Additionally, switching to serverless architectures such as AWS Lambda or Fargate allows for dynamic scaling, ensuring that resources are only consumed when necessary, reducing idle resources and power consumption. Serverless automatically shuts down instances when not in use, optimising hardware utilisation and energy efficiency. By scaling down capacity to match real-time demand and using auto-scaling groups in AWS, we can avoid over-provisioning of resources and thus minimise energy usage during low-traffic periods.

For video processing and transcoding tasks, leveraging GPUs or specialised hardware accelerators can significantly improve performance efficiency for compute-intensive operations. These specialised processors use less energy compared to CPUs for these specific workloads.

Data Center Layer

The data centre layer presents an opportunity to optimise sustainability by selecting data centres located in regions with cooler climates. Cooler environments reduce the energy required for cooling hardware, which is a significant contributor to overall power consumption. By choosing AWS regions where renewable energy is more prevalent (such as Northern Virginia or Oregon) or cooler regions, we can reduce the environmental impact associated with cooling and power delivery. Time-shifting non-urgent tasks (e.g., video transcoding, backup tasks) to

off-peak hours, when temperatures are cooler and renewable energy availability is higher, can further reduce the energy required for cooling and grid power demand.

Additionally, the application can benefit from deploying workloads to data centres that support more efficient power conditioning and uninterruptible power supplies (UPS), further optimising energy consumption at the infrastructure level. AWS provides tools such as the AWS Sustainability Pillar to guide application deployment in more energy-efficient regions.

Resources Layer

At the resource layer, focusing on the carbon intensity of electricity powering the data centres is crucial. Choosing AWS regions powered by renewable energy, or regions with a higher proportion of solar, wind, or hydroelectric power, reduces the carbon footprint of the application. The AWS Carbon Footprint Tool can provide insights into the environmental impact of running the application, allowing us to make informed decisions about which data centres to use based on their renewable energy mix.

We can also adopt a strategy of purchasing carbon offsets or paying for matched renewable energy for data centre usage. This helps to ensure that renewable energy is generated in amounts equivalent to the energy consumed by the data centre, reducing the overall environmental impact. While not a direct reduction in emissions, such offsets can contribute to the broader goal of sustainability. However, as offsetting is not a perfect solution, the primary focus should remain on reducing direct energy consumption by improving software and hardware efficiency.