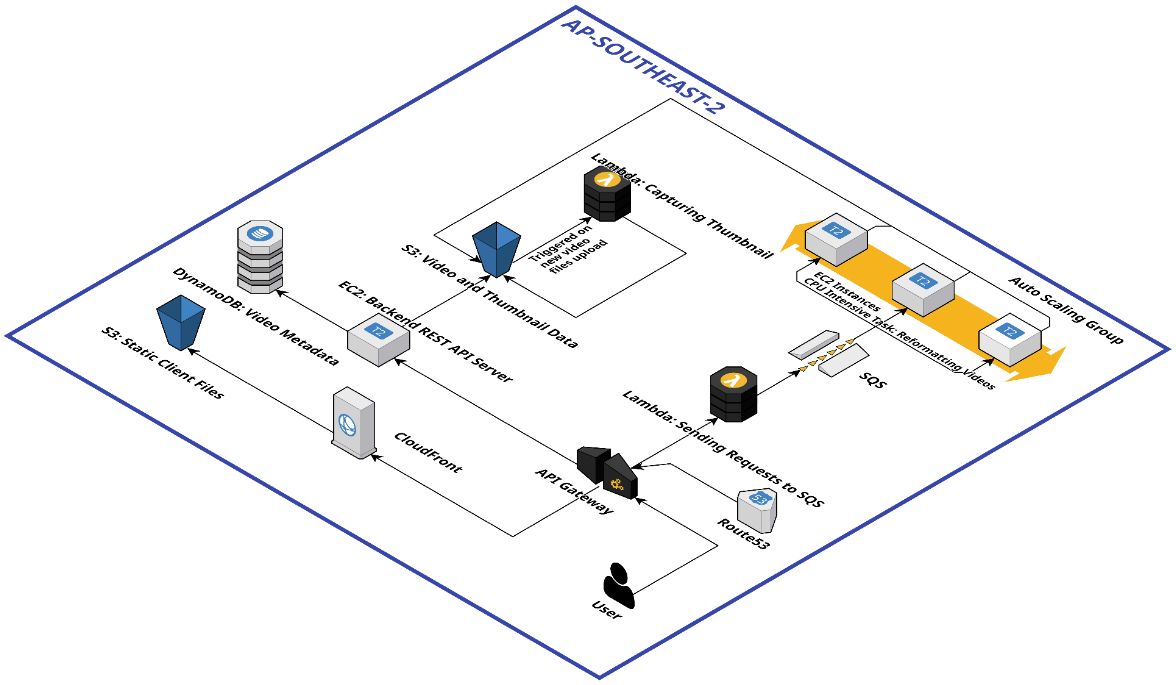
CAB432 Project Report

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 backend, hosted on an EC2 instance, handles core functionality 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 maintained 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 content is uploaded to S3. These thumbnails are also stored in S3 for easy access. CloudFront is used to deliver static frontend assets and ensure low-latency access to the web 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


AP-SOUTHEAST-2
Lambda: Capturing Thumbnail
Triggered on new video files upload
S3: Video and Thumbnail Data
DynamoDB: Video Metadata
EC2: Backend REST API Server
S3: Static Client Files
T2
EC2 Instances CPU Intensive Task: Reformatting Videos
Auto Scaling Group
SQS
Lambda: Sending Requests to SQS
CloudFront
API Gateway
Route53
User

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.

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 arrangement of the application is still largely monolithic, with a single backend REST API server handling critical functions such as user authentication, fetching videos, and managing uploads. This centralised structure can create bottlenecks under heavy traffic because all services are closely coupled and dependent on the performance of a single EC2 instance. To address this, I would further break up the backend into smaller, independently deployable services. Each microservice would be responsible for a distinct function—such as user authentication, video uploads, and metadata management—allowing each service to be scaled independently based on its specific demand. This decoupling improves fault tolerance, as the failure of one service would not affect the others, and enhances resource efficiency. Additionally, each microservice could be linked to its own application load balancer, providing granular control over how traffic is distributed.

For lightweight services such as authentication and video thumbnail generation, I would shift to a serverless architecture using AWS Lambda. Lambda functions are highly cost-effective for such workloads, and they scale automatically based on the volume of incoming requests, eliminating the need for manual infrastructure management. For compute-intensive tasks such as video processing and reformatting, I would containerize these services and deploy them using Amazon ECS (Elastic Container Service) or EKS (Elastic Kubernetes Service). Containers offer efficient use of compute resources, faster deployments, and seamless scaling across multiple instances, which is crucial for handling high levels of concurrent traffic.

Auto-scaling also needs to evolve beyond basic CPU utilisation tracking. Currently, the system employs target tracking scaling based on average CPU usage, but this could be augmented by implementing predictive scaling. Predictive scaling leverages historical traffic patterns to anticipate surges and automatically adjust infrastructure capacity ahead of time, thus reducing the risk of delays or service interruptions during peak loads.

In terms of load distribution, the current setup, which relies on API Gateway as the sole entry point for most requests, may not suffice for handling 10,000 concurrent users. I would increase API Gateway throttling limits by requesting a service limit increase from AWS, ensuring that it can handle the expected high volume of traffic. Additionally, SQS (Simple Queue Service), which currently manages video processing tasks, could be optimised by employing multiple SQS queues. This would allow better distribution of tasks to the appropriate compute resources, ensuring that no single queue becomes overwhelmed.

Finally, comprehensive load balancing is essential. By distributing traffic intelligently across multiple microservices, containers, and Lambda functions, we can avoid overloading any individual service or instance, ensuring a highly responsive and resilient application capable of scaling to support 10,000 concurrent users.

Security

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

Least Privilege

This principle ensures that each component and user in the application only has the minimum level of access required to perform its tasks. For instance, AWS Identity and Access Management (IAM) roles are assigned with least privilege in mind. EC2 instances, Lambda functions, and even users would only have access to specific S3 buckets, DynamoDB tables, or API endpoints necessary for their operation. For example, the Lambda function responsible for processing S3 uploads would only have write permissions to the S3 bucket where thumbnails are stored and read access to the video file bucket. To securely grant users or services access to video files or thumbnails stored in S3, S3 presigned URLs are used. These URLs allow temporary, limited-access permissions for uploading or retrieving specific objects, ensuring that access is granted only when necessary and only for a limited time. This minimises the risk of unauthorised access and helps enforce the principle of least privilege.

Separation of Concerns

Adopting a microservices architecture enforces the principle of separation of duties by isolating each service to handle a specific function. For instance, authentication, video uploads, and video processing should be managed by different services, ensuring that a failure or breach in one component does not compromise the entire system. 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

Multiple layers of security are 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 VPC (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

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.

Security by Design

From the ground up, the application is built with security as a core design principle. The OAuth2.0 protocol, implemented via AWS Cognito, secures user authentication flows by design, ensuring that tokens are securely generated and managed. The use of token-based authentication limits the exposure of sensitive data like passwords. To handle user file uploads and downloads securely, S3 presigned URLs are generated for specific objects, allowing temporary access to S3 resources without exposing the bucket publicly. This ensures that even if a presigned URL is leaked, it would only be valid for a limited period, reducing the window of risk. By using managed services like DynamoDB, much of the security responsibility, such as patching and availability, is handled by AWS, allowing the application to benefit from AWS's built-in security controls.

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.