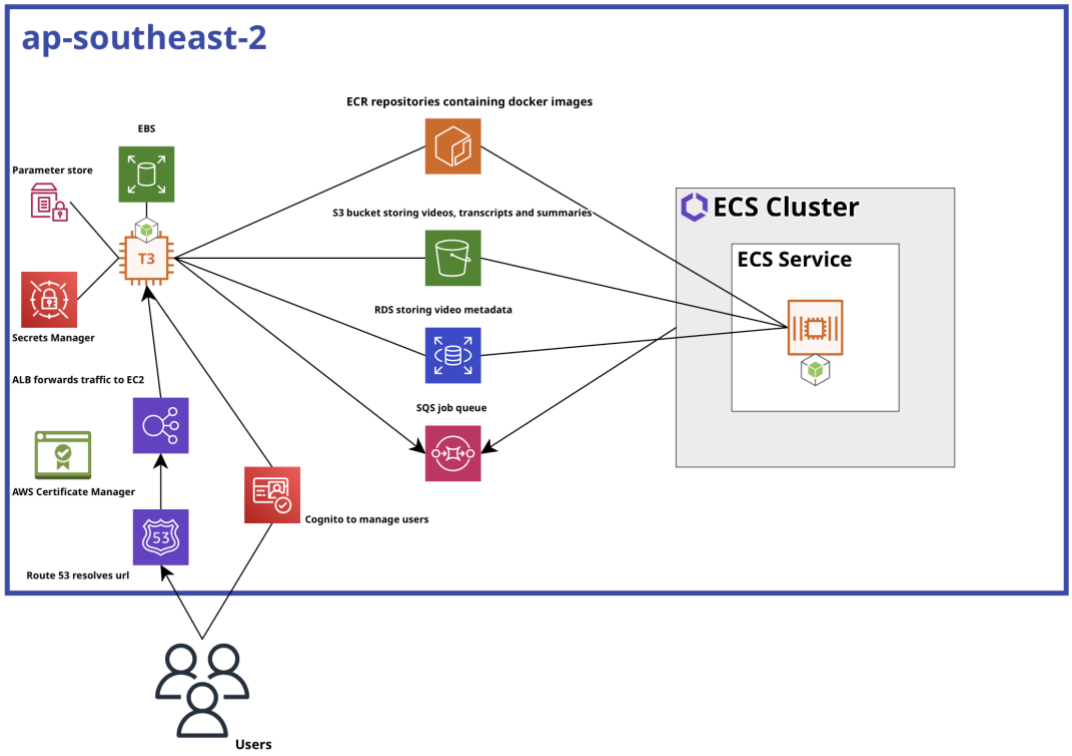
CAB432 Project Report

Daniel Crook - n12160334

Application overview

My application allows users to upload, view and summarise videos. Once a user has logged in or signed up to an account, they are displayed a dashboard of all the videos they have uploaded, they are also given the option to upload more videos. If they chose to do so they can upload a locally stored video onto the server and give it a name, it will then appear in their videos dashboard. Once a video is uploaded the user can play the video from the dashboard and it will stream it to them. The user is also given the option to generate a summary for the video, if they do so then the application will transcribe the video then using an AI API generate a summary of the video. Once the processing is complete the user can view or hide the summary for all the videos that they have processed. Videos only have to be processed once even if they log out of their account the summary is saved with the video and can be accessed again.

Application architecture



- **Route 53**: Used to manage DNS routing for my registered domains forwarding the requests to the application load balancer.
- **Certificate Manager**: Used to provide SSL/TLS certificates for secure HTTPS between the client and the application.
- **Application Load Balancer**: Forwards user requests to the backend target group that consists of EC2 instances.
- **EC2 instance**: Runs the node.js backend responsible for serving the website as well as handling the video upload process.
- **Elastic Container Registry**: Stores the docker images that contain the backend application as well as the worker process that processes the videos.
- **Secrets Manager and Parameter Store**: Stores environment variables and secrets that are loaded at run time.
- **S3**: Stores the video files uploaded by the users as well as their corresponding transcripts and summaries once they have been processed.
- **RDS**: Hosts the MySQL database that stores video metadata such as file names and uploader.
- **Cognito**: Manages user account details as well as user groups for regular and admin users.
- **SQS**: Used to store the job queue of videos waiting to be processed which the processing units can use to begin their next task.
- **ECS**: Runs the worker service as a serverless container which pulls the docker container from ECR that runs the node.js application that summarises the video. It also manages the autoscaling based on CPU usage.

Justification of architecture:

I chose to split my application into two microservices where one would handle the web backend, user interaction and uploading of videos and the other microservice would handle the processing of the uploaded videos which is the CPU intensive task. By separating these resources, it ensures that the user interaction in not impeded by the CPU intensive task and that the CPU intensive task is given maximum CPU resources in order for it to most efficiently process the videos. Separating the application also means that the CPU intensive task can scale independently of the web application which increases efficiency as the web application can handle many more users/requests on a single instance then the processing can.

For the web backend I chose to use EC2 for the compute this was because it provides a persistent environment which in necessary for a web application to have constant availability. EC2 also offers advanced control of the environment with good integration with the other AWS services I was planning on using. For the video processing service I chose to use ECS this was mainly due to its ability to scale serverless components meaning that amount of compute assigned to processing could vary with load. By using Fargate it removes the need to manage the servers that run the application which helps simplify the deployment. Lambda would not have been appropriate for any of these functions as it has a maximum execution time which is less than some of these processes will last. Additionally, its CPU limitations do not meet what is required for the processing task.

The communication of the jobs is handled through an SQS queue this was chosen as it allows me to decouple the web service from the processing making them independently scalable. It also allows for jobs to be continuously added without having to wait for the previous to finish. SQS also effectively handles varying loads seamlessly and improves fault tolerance by acting as a buffer between the different microservices.

The application load balancer allows me to integrate with the certificate manager allowing me to serve the site securely through SSL termination enhancing the security of the site.

I also made use of other AWS managed services such as Cognito and Secrets Manager these were chosen for their simple integration and cohesion with the other AWS services that I had already implemented.

Project Core - Microservices

- **First service functionality:** Process videos to generate a summary
- **First service compute:** ECS
- **First service source files:**
  - Worker/worker.js

- **Second service functionality:** Serve the backend, user interaction and uploading
- **Second service compute:** EC2
- **Second service source files:** web-backend directory

- **Video timestamp:** 0:00 – 1:00

Project Additional - Additional microservices

- **Third service functionality:**

- **Third service compute:**

- **Third service source files:**
- **Fourth service functionality:**
- **Fourth service compute:**

- **Fourth service source files:**
- **Video timestamp:**

Project Additional - Serverless functions

- **Service(s) deployed on Lambda:**
- **Video timestamp:**
- **Relevant files:**

Project Additional - Container orchestration with ECS

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

Project Core - Load distribution

- **Load distribution mechanism:** SQS
- **Mechanism instance name:** n12160334-processing-quue
- **Video timestamp:** 1:01 – 1:31
- **Relevant files: -** worker/worker.js/line 40-80
  - Web/backend/routes/api.js/line 156-190

Project Additional - Communication mechanisms

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

Project Core - Autoscaling

- **EC2 Auto-scale group or ECS Service name:** n12160334-assessment3-cluster/Worker-task-service
- **Video timestamp:** 1:32
- **Relevant files:** worker

Project Additional - Custom scaling metric

- **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:** https://videosum.cab432.com
- **Certificate ID:** 50bd2288-1a9b-4b46-aa3f-a6bc3c086225
- **ALB/API Gateway name:** ALB: n12160334
- **Video timestamp:** 4:55
- **Relevant files: n/a**

Project Additional - Container orchestration features

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

Project Additional - Infrastructure as Code

- **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:**
- **Content cached:**
- **Rationale for caching:**
- **Video timestamp:**
- **Relevant files:**

Project Additional - Other (with prior permission only)

- **Description:**
- **Video timestamp:**
- **Relevant files:**

Cost estimate

Calculator share link:

https://calculator.aws/#/estimate?id=2742361157024225c3e669f5310a210704493118

Monthly cost for 50 users:

- Amazon Route 53: 0.93 USD
- Elastic Load Balancing: 24.24 USD
- Amazon EC2: 20.04 USD
- Amazon Elastic Container Registry: 0.40 USD
- AWS Secrets Manager: 0.40 USD
- AWS Systems Manager: 0.00 USD
- Amazon Simple Storage Service: 4.15 USD
- Amazon RDS for MySQL: 43.48 USD
- Amazon Cognito: 0.00 USD
- Amazon Simple Queue Service: 0.00 USD
- AWS Fargate: 108.04 USD

Total monthly cost 201.68 USD

Scaling up

In order to scale to 10000 concurrent user one of the changes I would make would be to further decompose into more scalable microservices. These separate microservices would include a user management and authentication service, this would be responsible for user authentication and account management, this would be required to handle the login spikes keeping the application responsive to all users as well as increasing security by isolating this process. I would also add an upload service responsible for manging the upload of videos onto S3, this would allow it to scale based on demand as well as let it be optimized to perform just this task for high data throughput. I would also ensure that the web server on EC2 could scale to meet the user requirements and remove the restrictions on the maximum number of ECS tasks running at the same time.

To orchestrate these additional services I would implement an API gateway that would route the API requests to the appropriate microservice. This would unify the services as well as providing other benefits such as the ability to implement caching and increased monitoring.

In order to handle the increased load I would also upgrade the instance type to something with more and faster cores such as the m5.large as well as upgrading the RDS instance type to an m5.large.

To increase performance, I would implement a caching layer to cache frequently accessed data so that application response times are improved.

The application load balancer would also need to be adjusted to distribute the load across multiple instance targets stopping one target from becoming a bottleneck.

Security

Data confidentiality and integrity is ensured by using that all data in transit is secured over HTTPS, by provisioning SSL/TLS certificates for the Application Load Balancer all data in transit is encrypted which ensure that the sensitive information is protected even in the case of interception and is protected from tampering. Data protection could be ensured by encrypting all stored data in S3 and RDS so in the case of a data breech the data is not compromised. Sensitive information is all stored using secrets manager reducing the risk of credentials being leaked.

Strong authentication and authorization is ensured through the use of Cognito which enhances user access security. Network security is ensured through the use of security groups that only allow the necessary to pass between the resources reducing the risk of unauthorised access.

The applications defence could be improved by implementing a firewall such as AWS Web Application Firewall that would prevent common web based attacks from impacting the application. Implementing input validation so that all uploaded videos and there names are validated and checked thoroughly before being uploaded to the server.

Sustainability

On the software level the sustainability of the application can be improved by increasing the efficiency of the processing algorithms particularly the transcription algorithm. By reducing the complexity, it will reduce the CPU usage and therefore the energy that is needed to perform the processing. Reducing data transfers such as when videos are uploaded then immediately redownloaded for transcription instead of going straight to a preprocessing storage will reduce network bandwidth usage and energy consumption. Implementing data compression to reduce file sizes will reduce transfers and processing time saving energy. Implementing caching will also save energy as frequently used data will have to be downloaded less often.

On the hardware level ensuring that the chosen instance types are optimal for the task they are performing and do not leave a large overhead wasting energy. Chose more efficient graviton based instances over the current x86 architecture that is currently being used. Ensure that unused resources are removed and an efficient policy is in place to remove unused compute.

In the data centre by picking to host the resources in data centres that are powered by renewable energy will reduce the carbon footprint of the application.

Resources should be managed to efficiently meet demand and scale down optimally when the demand is decreased. Energy is optimally used when resources auto scale to meet the

demand that is currently required. Implement policies to remove unused data so that it is not being stored unnecessarily.