Serverless/Event-driven Architectural Design Report

Thanh Nam Vu (ID: 104991276)
Alexander Rigato (ID: 105338913)
Phuong Bao Minh Nguyen (ID: 104339685)
Hoai Thuong Triet Nguyen (ID: 103522028)
Swinburne University of Technology
Tutorial time: Thursday 4:30pm
Date of submission: 20 October 2024

Contents

I. Introduction

THe goal of this project is to design and re-engineer a fast, scalable, and serverless cloud-based architecture for web applications. This new architecture will address the current business challenges while meeting both the growing demands and future requirements.

The Photo Album platform is experiencing rapid growth, with traffic doubling every six months, which is placing significant strain on the current infrastructure. The proposed architecture will improve scalability, lower costs, and enhance performance, with optimized global response times to better serve users across multiple regions. Additionally, it will support future expansion to handle video content, including transcoding and media processing capabilities.

This report outlines the architectural decisions made, explaining the rationale behind choosing each AWS service. The focus is not only on meeting current needs but also on building a foundation for future innovation and scalability.

II. Architecture Overview

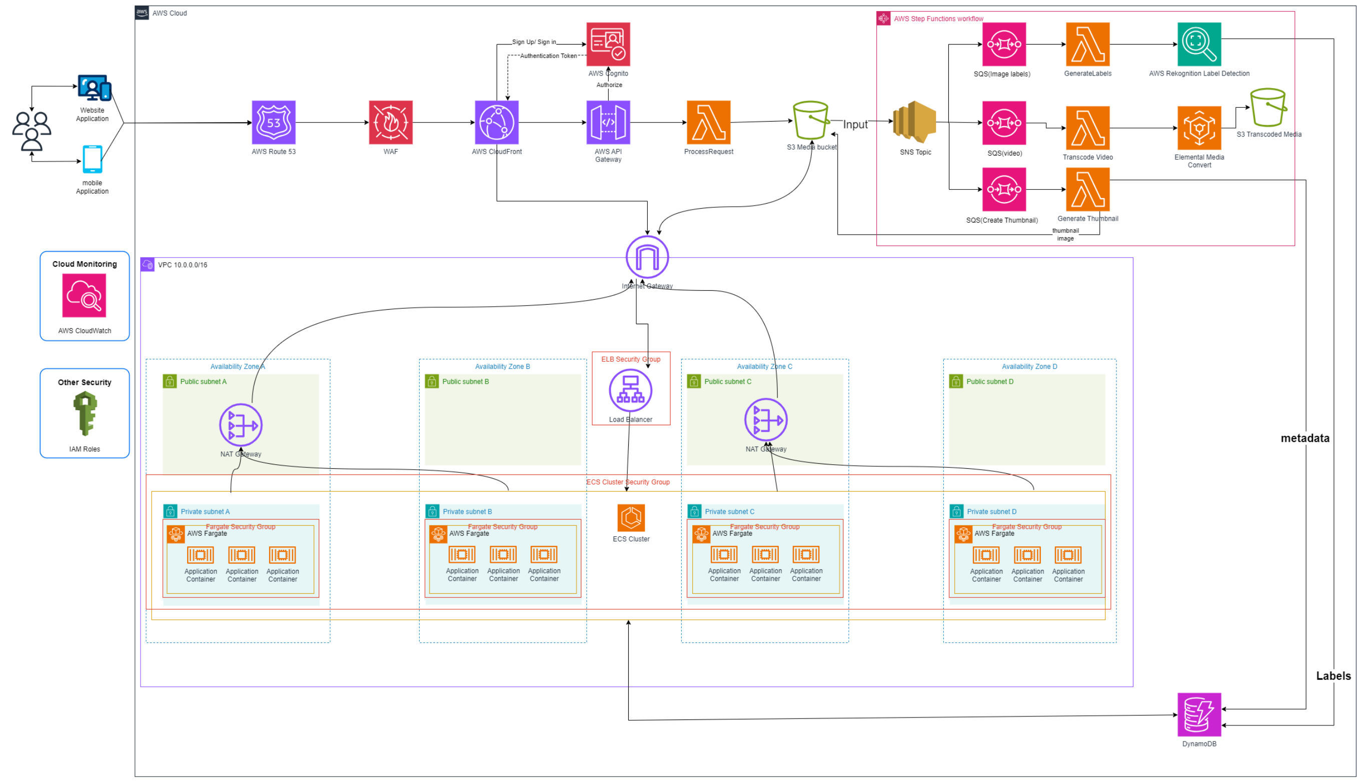

Fig. 1: Architecture Design

Here is the definition list of each AWS service we used on our diagram:

- **Route 53**: Provides DNS management and routing, using latency-based routing to direct users to application deployments in regions with the lowest latency. [1]
- **WAF (Web Application Firewall)**: Protects web applications from threats like SQL injection and cross-site scripting by filtering traffic at the edge when integrated with services like Amazon CloudFront. [2]
- **CloudFront**: Accelerates content delivery by caching static and dynamic assets at global edge locations, ensuring faster access for users worldwide. [3]
- **Cognito**: Handles user authentication and authorization, managing user sign-ups, logins, and providing temporary credentials for secure access to AWS resources. [4]
- **API Gateway**: Manages RESTful and WebSocket APIs, exposing backend services to front-end applications while handling request validation, throttling, and authentication. [5]
- **Lambda**: Processes backend logic without managing servers by executing code in response to events, such as API Gateway requests or S3 uploads. [6]
- **S3 Bucket**: Provides scalable object storage for media files, backups, and processed outputs, with features like versioning and lifecycle policies for efficient data management. [7]
- **SNS (Simple Notification Service)**: Sends real-time notifications to subscribers via email, SMS, or HTTP endpoints, facilitating event-driven architectures. [8]

- **SQS (Simple Queue Service)**: Ensures reliable, asynchronous communication between components by queuing tasks and delivering messages in order, supporting both standard and FIFO queues. [9]
- **SQS (Image Labels Queue)**: Stores tasks for image labeling requests, ensuring orderly processing of messages related to AWS Rekognition.
- **SQS (Video Queue)**: Holds video-related tasks like transcoding or analysis requests and manages workloads efficiently by processing them asynchronously.
- **SQS (Create Thumbnail Queue)**: Queues thumbnail generation tasks, ensuring asynchronous processing and preventing delays.
- **AWS Step Functions**: Orchestrates workflows by coordinating multiple AWS services, enabling complex processes with visual monitoring and built-in error handling. [10]
- **AWS Rekognition**: Analyzes images and videos to generate labels, detect objects and scenes, and perform facial recognition for content moderation and metadata generation. [11]
- **AWS Elemental MediaConvert**: Transcodes media files into various formats, enabling adaptive bitrate streaming for on-demand and live-streaming applications. [12]
- **NAT Gateway**: Allows resources from a private subnet in the VPC to access the internet for outbound services, while maintaining security by not exposing them directly to the internet. [13]
- **Internet Gateway**: Connects the VPC to the internet, enabling resources in public subnets to communicate with external services. [14]
- **Elastic Load Balancer (ALB)**: Distributes incoming traffic across multiple ECS containers, ensuring stability and efficient scaling as demand increases. [15]
- **ECS Cluster**: Distributes tasks effectively across container applications, optimizing resource utilization. [16]
- **ECS Fargate**: A compute engine for automated container management, providing automatic scaling and resource management without manual intervention. [17]
- **Application Containers(ECS)**: Containers deployed within ECS perform core functions, including data processing and task management. [18]
- **DynamoDB**: A NoSQL database for storing metadata, such as image tags and processing status, offering high scalability and performance. [19]
- **CloudWatch**: Monitors and logs the performance of AWS resources, providing real-time alerts and insights into system behavior. [20]
- **IAM Roles**: Defines different access roles to improve security, ensuring that each role has appropriate access levels within the system. [21]

III. Service Descriptions and Functions

Networking and Traffic Management

A. Route 53: DNS Management

In this architecture, **Route 53** acts as a scalable DNS service, directing user traffic by resolving domain names to the right IP addresses. It uses *latency-based routing* to send requests to the closest CloudFront distribution, cutting down response times. As the application grows globally, this ensures users from all regions can access the system quickly, meeting the need for faster response times worldwide.

B. WAF: Web Application Firewall to Protect Against Malicious Traffic

WAF enhances the application's security by filtering out malicious traffic before it reaches CloudFront or backend services. As the user base grows, it protects against common threats like *DDoS attacks* and *SQL injection*, ensuring only legitimate traffic gets through. This helps maintain system stability and prevents downtime, which is crucial as the application scales.

C. CloudFront: Content Delivery for Faster Global Response Times

CloudFront, the content delivery network (CDN) in this design, caches media files near users to provide faster access to frequently requested content. It enhances the user experience by quickly delivering photos and videos, even as global usage grows. For uncached content, CloudFront forwards requests to the backend via the *Internet Gateway*, ensuring smooth performance. This setup also eases the load on backend resources, helping the system scale efficiently with increasing demand.

Authentication and API Gateway

D. Cognito: User Authentication and Authorization

Amazon **Cognito** handles user authentication and authorization, ensuring only authorized users can access the application. It manages sign-up, sign-in, and authentication, issuing secure tokens to validate sessions between the front-end and backend.

As the user base grows, Cognito helps the system scale by securely managing identities without heavy in-house management, aligning with the company's goal of reducing operational overhead. This ensures seamless, secure access for users across multiple devices.

E. API Gateway: Exposing APIs for Front-end and Back-end Interaction

API Gateway exposes backend services to the front-end through secure APIs. It serves as the central hub for processing requests from web and mobile apps, forwarding them to backend services like *Lambda functions* or containers. As the company adopts a serverless, event-driven approach, API Gateway ensures smooth communication between components while maintaining high performance. It also supports future growth by enabling new features and APIs to be added without significant changes to the infrastructure.

Serverless and Compute Services

F. Lambda Function

Lambda helps minimize operational complexity and save expenses due to its serverless nature and ability to grow automatically on demand. In the proposed architecture, there will be 5 Lambda functions, each responsible for performing tasks such as labeling, classifying images, transcoding videos, and creating thumbnails. These Lambda functions are activated when a user uploads image or video content to the Amazon S3 bucket.

When an image is uploaded, a Lambda function is triggered to invoke AWS *Rekognition* to analyze the uploaded image and create a label. In the second scenario, when a video is uploaded, Lambda initiates the video transcoding process, converting the video into formats suitable for different devices. Additionally, having a Lambda function between API Gateway and S3 acts as a secure intermediary, enforcing access control and handling authentication logic.

G. ECS Fargate

Using **ECS Fargate** in this business scenario provides the benefit of ensuring steady development without worrying about infrastructure. Since there is no need to manually provision servers, containers operate in a serverless environment, automatically allocating resources and balancing workloads based on the complexity of the tasks and container requirements.

ECS Fargate also simplifies the scaling process by enabling the system to adapt efficiently to varying traffic demands without manual adjustments, making it an ideal choice to support growth and maintain business efficiency.

Storage and Databases

H. S3 Buckets: Storing Media and Processed Outputs

In the architecture, **Amazon S3** serves as the primary storage solution for both raw and processed media files, including images, videos, and their corresponding thumbnails. When a user uploads media, the files are stored in S3 buckets, providing a highly scalable, durable, and cost-effective way to handle large volumes of data.

S3 is designed to handle the increasing demands of media-heavy applications while ensuring data is accessible when needed. It also triggers events to initiate further processing, such as image resizing or video transcoding, with the processed results stored back into designated S3 buckets for future retrieval.

I. DynamoDB: Storing Metadata Such as Image Labels

For storing metadata related to the media, **Amazon DynamoDB** is used. DynamoDB stores crucial information like image labels generated by AWS *Rekognition* and the status of video processing. As a NoSQL database, DynamoDB is optimized for fast read and write operations, allowing it to handle large amounts of data with low latency.

It automatically scales to meet demand and, being serverless, requires no manual management. DynamoDB is ideal for storing metadata that supports real-time tasks like searching, filtering, and organizing media files.

Messaging and Workflow

J. SNS: Trigger Notifications When Media is Uploaded

Amazon SNS triggers notifications whenever new media is uploaded to the S3 bucket. In this architecture, it ensures that components are immediately notified when a user uploads photos or videos, starting the necessary processing workflows. For instance, SNS sends notifications to *SQS* queues to initiate tasks like transcoding or thumbnail generation. This real-time messaging keeps the media processing pipeline running smoothly and ensures the system stays responsive without requiring manual intervention.

K. SQS: Queueing Tasks for Asynchronous Processing

Amazon SQS queues tasks for asynchronous processing, keeping the system decoupled and scalable. In this architecture, three specialized queues manage distinct tasks: the *Image Labels Queue* sends media to AWS *Rekognition* for labeling, the *Video Queue* handles video transcoding with AWS *MediaConvert*, and the *Create Thumbnail Queue* generates thumbnails for uploaded images.

This setup ensures that tasks run independently, preventing system overload, with worker nodes processing tasks at their own pace. By distributing workloads across these queues, the application efficiently handles multiple media files simultaneously. SQS also supports future extensibility, allowing new tasks to be integrated easily, aligning with the company's goal of a scalable, flexible media processing system.

Media Processing

L. MediaConvert: Video Transcoding and Media Transformation

AWS Elemental MediaConvert manages video transcoding and media transformation, converting uploaded videos into multiple formats for compatibility across devices. It supports adaptive bitrate streaming for smooth playback and prepares the system for future live-streaming needs.

In this architecture, the *Video Queue* in SQS triggers MediaConvert jobs, ensuring efficient, asynchronous video processing. This service aligns with the company's goal of minimizing in-house infrastructure by offloading resource-heavy video tasks to the cloud. MediaConvert offers scalable and cost-effective processing, adjusting to handle fluctuating workloads seamlessly.

Transcoded videos are stored back in S3, ready for quick delivery via CloudFront, ensuring fast access for users worldwide. With MediaConvert, the system stays ready to meet future media demands, supporting growth while maintaining high performance and user satisfaction.

IV. Design Rationale

Fulfillment of Business Requirements

In designing the architecture for the Photo Album application, each service was carefully selected to meet the client's business requirements and support growth and operational efficiency. Below is an in-depth explanation of how each business requirement is fulfilled.

A. Managed AWS Services

The entire architecture utilizes managed AWS services to minimize the need for in-house infrastructure management. Managed services like ECS Fargate, Lambda, DynamoDB, API Gateway, S3, and others allow the company to focus on deployment, innovation, and scaling while AWS handles the underlying infrastructure.

- **Automatic Scaling:** AWS services such as ECS, Lambda, and DynamoDB can automatically scale based on demand, ensuring high availability and seamless growth.
- **High Availability and Maintenance:** Services like CloudFront, S3, and DynamoDB come with built-in redundancy and availability across multiple Availability Zones (AZs), reducing downtime. AWS also handles updates and patching.

B. Scalability

The architecture supports scalability to handle the growing demands of the application and global traffic efficiently.

- **ECS Fargate over EC2:** ECS Fargate provides a distributed and cost-efficient solution for running containers. Unlike EC2, which requires manual instance management, Fargate automatically scales containers based on workload, ensuring optimal performance with minimal operational overhead.
- **Lambda Functions:** Event-driven Lambda functions auto-scale based on incoming requests and perform critical tasks such as media processing and AI-based labelling.
- **DynamoDB:** This NoSQL database scales automatically to handle high read and write volumes, ensuring performance at any scale without administrative effort.
- **Route 53 and CloudFront:** Route 53's latency-based routing directs traffic to the closest edge locations, while CloudFront caches content globally, reducing response times and handling increased user traffic efficiently.

C. Serverless and Event-Driven Architecture

The architecture leverages serverless computing through Lambda and event-driven components such as SNS and SQS to respond to dynamic traffic and workloads.

- **Lambda Functions:** These functions handle media processing tasks like creating thumbnails, transcoding videos, and generating low-resolution images in response to S3 upload events.
- **SNS and SQS:** SNS delivers notifications upon media upload, and SQS queues distribute workloads to ensure asynchronous processing. This design decouples services, enabling independent scaling and reducing system bottlenecks.
- **ECS Fargate:** Additional workloads are offloaded to ECS Fargate containers, ensuring the architecture remains scalable and responsive.

D. Relational Database Alternatives

Instead of using RDS or Aurora, which are SQL databases, the architecture leverages DynamoDB for its performance, scalability, and cost-effectiveness.

- **DynamoDB as the Preferred Database:** DynamoDB, a NoSQL database, offers faster read and write speeds, automatic scaling, and lower operational costs compared to traditional SQL databases. This ensures high performance for workloads such as storing image metadata and other structured data.
- **AI and Metadata Processing:** DynamoDB integrates seamlessly with other services like Lambda and Rekognition, enabling efficient metadata management and AI-powered tagging.

E. Solving Slow Global Response Times

To address the need for global reach and low-latency performance, the architecture integrates several AWS services:

- **CloudFront:** Acts as the content delivery network (CDN), caching media and static content at edge locations to provide faster access to users worldwide.
- **Route 53:** Utilizes latency-based routing, ensuring traffic is directed to the nearest AWS region to minimize response times.
- **CloudFormation for Multi-Region Deployment:** Automates the deployment of the infrastructure across multiple regions, enhancing availability and ensuring consistent performance globally.

F. Handling Video Media

Handling video media efficiently is a key requirement of the application. The architecture leverages AWS Elemental MediaConvert for high-quality video transcoding.

- **Lambda and MediaConvert Integration:** When videos are uploaded to S3, Lambda functions trigger MediaConvert jobs to transcode the videos into multiple formats, ensuring compatibility across devices.
- **S3 Buckets for Processed Media:** The transcoded media is stored back into S3 buckets, ready for distribution via CloudFront to users globally.

G. Decoupled Architecture

Decoupling components ensures that the system is modular, scalable, and maintainable. This architecture achieves decoupling using SNS and SQS to handle different tasks independently.

- **SNS Topics for Notifications:** When media is uploaded, SNS sends notifications to multiple subscribers, triggering tasks such as AI-based labelling or thumbnail generation.
- **SQS for Asynchronous Task Handling:** Separate SQS queues are used to handle tasks like image labelling, video processing, and thumbnail creation independently, ensuring the system remains scalable and resilient.

Justification of Chosen Solution

The architecture is designed to meet all business requirements while providing scalability, reliability, security, and cost-efficiency.

Performance:

- SNS provides low-latency message delivery to multiple subscribers.
- SQS ensures high-throughput messaging without bottlenecks.
- ECS Fargate and Lambda guarantee fast and efficient workload execution.

Reliability:

- SNS uses at-least-once delivery with retry mechanisms, ensuring reliable message transmission.
- S3 offers highly available storage with redundancy across multiple AZs.

- SQS guarantees tasks are not lost during disruptions and are reliably processed.

Security:

- Cognito ensures secure user authentication, preventing unauthorized access and spam uploads.
- IAM roles control access to AWS services, following the principle of least privilege.
- WAF and security groups provide multiple layers of security by filtering malicious traffic.

Cost Efficiency:

- Lambda and DynamoDB are serverless, meaning costs are tied directly to usage, without the need for provisioning resources.
- SNS offers a free tier and affordable messaging rates, keeping operational costs low.
- ECS Fargate avoids the need for EC2 instance management, running containers only when required, further optimizing costs.

In summary, the architecture design fulfills the business scenario requirements by using managed, scalable, and serverless solutions. With ECS Fargate, Lambda, and DynamoDB at its core, the system ensures rapid scaling, high performance, and cost-efficiency. CloudFront and Route 53 improve global response times, while MediaConvert enables efficient video processing. Decoupling via SNS and SQS promotes a modular system that can grow and evolve without disrupting existing services. Additionally, the architecture is secure, reliable, and cost-effective, meeting the client's current and future needs.

Alternatives Comparison

H. SQL vs. NoSQL Databases

A SQL database, such as **RDS**, is relational and follows a fixed schema. SQL databases only support vertical scaling due to their relational structure, meaning they require a consistent view of the data across the entire system. This makes SQL databases ideal for transactional systems that demand data integrity but limits scalability.

In contrast, a NoSQL database, such as **DynamoDB**, is non-relational with a flexible schema and supports horizontal scaling. NoSQL databases are well-suited for handling unstructured or semi-structured data and excel in real-time analytics or workloads requiring high scalability. [22]

I. DynamoDB vs. RDS

DynamoDB is optimized for high throughput and low-latency access, making it ideal for large-scale workloads. It follows a pay-per-use pricing model where you only pay for the read and write capacity you consume, making it cost-efficient for dynamic workloads. On the other hand, **RDS** is a SQL-based database that excels in transactional operations but may struggle with large-scale read and write demands. RDS becomes more expensive for larger workloads since its pricing model is based on CPU, memory, storage, and I/O operations rather than read/write usage alone. [23]

DynamoDB Pricing:

- Writes: \$1.25 per million write requests
- Reads: \$0.25 per million read requests

[24]

RDS Pricing:

- Storage: \$0.10 per GB-month
- I/O: \$0.20 per million requests

[25]

J. Compute Options: ECS vs. EC2

In this solution, we chose **ECS** over **EC2** for compute resources. **ECS** offers a serverless container management solution, removing the need to manage underlying infrastructure. It provides seamless auto-scaling based on workload demand and is more cost-effective since you only pay for the active containers. In comparison, **EC2** requires manual management, including OS updates, configuration, and patching. EC2 also needs the setup of auto-scaling groups and is generally more costly since you pay for the instances regardless of their utilization.

ECS Cost per Month:

- Per vCPU: $720 \times \$0.04048 = \$29.1456$ per month
- Per GB: $720 \times \$0.004445 = \$3.2004$ per month

[26]

EC2 Cost (t3.medium):

- $720 \times \$0.0416 = \$29.952$ per instance per month
- 3 instances: $3 \times 29.952 = \$89.85$ per month

[27]

K. Media Processing: MediaConvert vs. Elastic Transcoder

We opted to use **MediaConvert** for media processing due to its advanced features and broader support for formats and codecs. MediaConvert offers a pay-as-you-go pricing model based on video duration and encoding features, such as HD or UHD output. It also provides more customization options, allowing for fine-tuning of encoding settings. While **Elastic Transcoder** is designed for simpler transcoding tasks, it offers fewer advanced features and is being gradually replaced by MediaConvert. [28] [29]

Elastic Transcoder: $0.0075 per minute [29]

MediaConvert:

- First 100,000 normalized minutes: $0.0075 per minute

[30]

L. AWS 2-Tier vs. 3-Tier Architecture

In a **two-tier architecture**, the application consists of a client layer (presentation tier) and a server layer (data tier). The client communicates directly with the database, making this architecture simpler but limiting scalability and security.

In a **three-tier architecture**, an additional middle layer, known as the application or logic tier, is introduced. This layer processes requests from the presentation tier, interacts with the data tier, and decouples logic from the user interface. This structure enhances scalability and security since the client doesn't interact directly with the database. However, the added complexity makes it better suited for larger applications that require independent scaling and more robust security. [31]

V. Use Cases: Media Upload and Processing Workflow

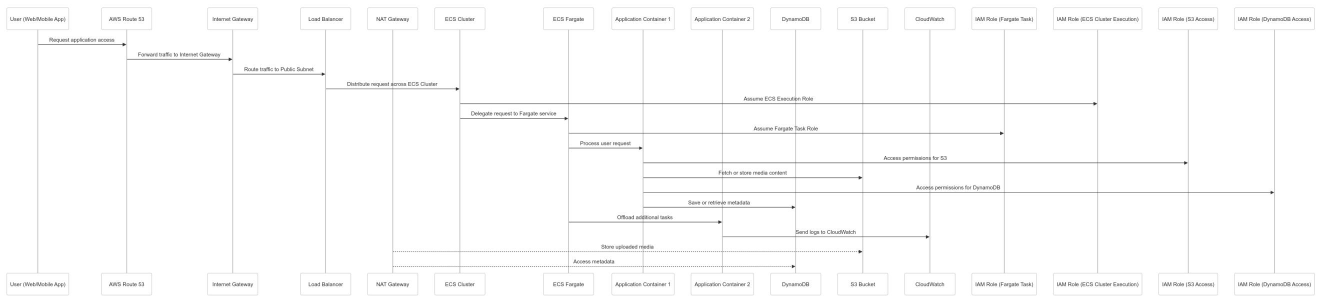
User (Web/Mobile App)
AWS Route 53
Internet Gateway
Load Balancer
NAT Gateway
ECS Cluster
ECS Fargate
Application Container 1
Application Container 2
DynamoDB
S3 Bucket
CloudWatch
IAM Role (Fargate Task)
IAM Role (ECS Cluster Execution)
IAM Role (S3 Access)
IAM Role (DynamoDB Access)
Request application access
Forward traffic to Internet Gateway
Route traffic to Public Subnet
Distribute request across ECS Cluster
Assume ECS Execution Role
Delegate request to Fargate service
Assume Fargate Task Role
Process user request
Access permissions for S3
Fetch or store media content
Access permissions for DynamoDB
Save or retrieve metadata
Offload additional tasks
Send logs to CloudWatch
Store uploaded media
Access metadata

Fig. 2: UML Diagram for Front-end workflow

The user initiates a request from a web or mobile app, which is resolved by **Route 53** and directed to **CloudFront**. If the requested content isn't available in the cache, **CloudFront** forwards the request through the **Internet Gateway** into the **VPC**. Once inside the VPC, the **Load Balancer** in the public subnet distributes the incoming traffic to the appropriate backend service running on the **ECS Cluster**. Since the **ECS Cluster** is deployed in a private subnet for security, the request flows through the **NAT Gateway**, ensuring secure communication between public-facing services and private resources.

The **ECS Cluster** assumes the **IAM Role for ECS Execution** to launch and manage tasks on **Fargate**. **Fargate** handles the execution of these tasks while assuming the **IAM Role for Fargate Task** to interact securely with other AWS services. If the request involves media files, **Container1** either stores new files or retrieves existing ones from **S3**, using the **IAM Role for S3 Access** to manage access. For any metadata operations, **Container1** interacts with **DynamoDB** by assuming the **IAM Role for DynamoDB Access**, allowing it to store or query metadata like file details or user activity records.

In cases where additional or intensive processing is required, **Container1** offloads tasks to **Container2**, ensuring the workload is distributed efficiently. **Container2** then sends logs and operational metrics to **CloudWatch**, allowing for continuous monitoring and troubleshooting of system health. The **NAT Gateway** ensures that containers in the private subnet can securely access **S3** and **DynamoDB** without exposing the underlying resources to the public internet, maintaining a high level of security throughout the data flow.

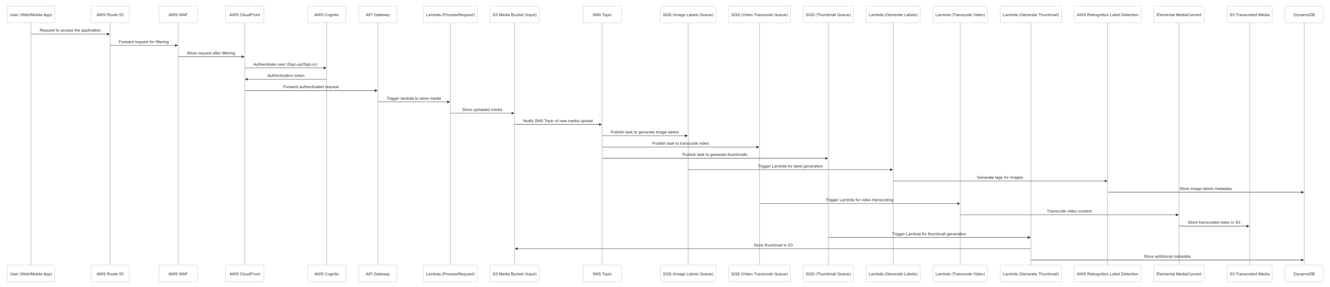
User (Web/Mobile App)
AWS Route 53
AWS WAF
AWS CloudFront
AWS Cognito
API Gateway
Lambda (ProcessRequest)
S3 Media Bucket (Input)
SNS Topic
SQS (Image Labels Queue)
SQS (Video Transcode Queue)
SQS (Thumbnail Queue)
Lambda (Generate Labels)
Lambda (Transcode Video)
Lambda (Generate Thumbnail)
AWS Rekognition Label Detection
Elemental MediaConvert
S3 Transcoded Media
DynamoDB
Request to access the application
Forward request for filtering
Allow request after filtering
Authenticate user (Sign-up/Sign-in)
Authentication token
Forward authenticated request
Trigger lambda to store media
Store uploaded media
Notify SNS Topic of new media upload
Publish task to generate image labels
Publish task to transcode video
Publish task to generate thumbnails
Trigger Lambda for label generation
Generate tags for images
Store image labels metadata
Trigger Lambda for video transcoding
Transcode video content
Store transcoded video in S3
Trigger Lambda for thumbnail generation
Store thumbnail in S3
Store additional metadata

Fig. 3: UML Diagram for Back-end workflow

The user sends a request through **Route 53** from a web or mobile app. **Route 53** forwards the request to **AWS WAF** to filter out any malicious traffic. Once the request passes the **WAF** filters, it is routed to **CloudFront**, which caches content and handles further processing. **CloudFront** checks if the user is authenticated by interacting with **Amazon Cognito**. If the user is not signed in, **Cognito** manages the login or registration. Once authenticated, **Cognito** sends a token back to **CloudFront**, which forwards the authenticated request to **API Gateway** for backend processing.

When the request reaches **API Gateway**, it triggers a **Lambda** function (`ProcessRequest`) to handle the media upload. The **Lambda** function stores the uploaded media in an **S3 Input Bucket**. After the media is saved, **S3** sends a notification to an **SNS Topic**, which informs other services about the new upload.

The **SNS Topic** distributes tasks to three different **SQS** queues. One task goes to the **Image Labels Queue**, which handles image label generation. Another task is sent to the **Video Transcode Queue** for processing videos. The final task goes to the **Thumbnail Queue**, where thumbnails for the uploaded media will be generated.

The **Image Labels Queue** triggers a **Lambda** function (`Generate Labels`) to generate tags for the uploaded images. This **Lambda** function uses **AWS Rekognition** to analyze the images and detect relevant labels. Once the labels are generated, they are stored in **DynamoDB** for easy retrieval later.

For video files, the **Video Transcode Queue** triggers another **Lambda** function (`Transcode Video`) to manage the transcoding process. This **Lambda** function calls **AWS Elemental MediaConvert** to transcode the video into different formats. The transcoded video is then saved in an **S3 Output Bucket**.

The **Thumbnail Queue** triggers another **Lambda** function (`Generate Thumbnail`) to create thumbnails for the media. These thumbnails are saved back in the **S3 Input Bucket**. The function also saves metadata about the thumbnails in **DynamoDB** for future reference.

This architecture uses **Route 53**, **WAF**, **CloudFront**, and **Cognito** to manage routing, security, caching, and authentication. **API Gateway** and **Lambda** handle media uploads and task processing. **S3** stores the input media, transcoded outputs, and thumbnails. **SNS** and **SQS** distribute tasks to **Lambda** functions, ensuring asynchronous processing. **AWS Rekognition** and **MediaConvert** handle image recognition and video transcoding, while **DynamoDB** stores metadata about the media and its labels. This design is event-driven, scalable, and fault-tolerant, ensuring efficient media handling across all components.

VI. Assumptions and Constraints

Assumptions

Traffic Growth: The company expects traffic to double every six months for the next two to three years. The architecture must support this exponential growth and handle peak loads reliably, favoring a scalable, serverless solution that can adjust to varying demand.

Global User Base: Since the application serves a global audience, using **AWS Route 53** for DNS management and **CloudFront** to cache content at edge locations ensures low-latency access for users worldwide.

Media Format Diversity: Users will upload various media formats, including JPEG, PNG, MP4, and MOV. To efficiently manage this diversity, **AWS MediaConvert** will transcode videos and optimize other media types for consistent delivery across devices.

Serverless Architecture: The system will leverage **AWS Lambda** for event-driven processing and **ECS Fargate** for containerized workloads. This approach ensures workloads like video transcoding and thumbnail generation scale automatically, reducing operational overhead.

Availability: The design relies on highly available services like **DynamoDB** and **S3**, ensuring uninterrupted access to media and metadata. These services also provide built-in replication to maintain data availability across regions.

Security: **IAM Roles** will enforce fine-grained access control, ensuring secure interactions between AWS services. **WAF** and **Cognito** will provide additional security by filtering malicious traffic and managing user authentication.

Growth: With expected increases in media uploads and metadata storage, **S3** and **DynamoDB** will scale horizontally to accommodate future growth without requiring additional infrastructure.

Constraints

Lack of Caching: Currently, the system does not include a caching layer for metadata stored in **DynamoDB**, which could result in slower data retrieval times. Implementing a cache, such as **ElastiCache**, in the future could improve read performance under high load.

AWS Service Limits: The architecture must operate within AWS service limits, such as Lambda invocation rates and SQS queue limits. During peak traffic, these limits may affect performance and require tuning or additional capacity planning.

Storage Costs: As media uploads increase, the cost of **S3** storage will rise. Adding lifecycle policies to archive infrequently accessed content to **S3 Glacier** can help optimize costs over time.

Cold Start Delays: Lambda functions may experience cold start delays during sudden traffic spikes, potentially slowing down workloads such as **AWS Rekognition** tagging or **MediaConvert** transcoding. This can impact user experience during peak periods.

Network Latency: High traffic between VPC subnets and AWS regions could introduce latency. **CloudFront** mitigates this by caching frequently requested content, but further optimization may be required for real-time interactions.

Access Control Complexity: Managing multiple **IAM Roles**, along with **Cognito** and **WAF**, introduces operational complexity. Misconfigurations in access policies could affect system availability or expose services to risks.

MediaConvert Bottlenecks: Processing large volumes of video files simultaneously using **MediaConvert** may result in delays, impacting real-time media delivery. Monitoring and optimizing encoding workloads will be essential.

Auto-Scaling Complexity: Using both **ECS Fargate** and **Lambda** introduces challenges in tuning auto-scaling thresholds to prevent under-provisioning or over-provisioning during dynamic traffic patterns.

DDoS Risks: While **WAF** offers protection against common threats, a large-scale DDoS attack could affect service availability. Implementing additional mitigation strategies may be necessary to ensure uptime.

VII. Work Distribution

Thanh Nam Vu

- Develop the Architecture Design
- Design Frontend Workflow UML
- Design Backend Workflow UML
- Fulfillment of Business Requirements
- Justification of Chosen Solution
- Use Cases: Media Upload and Processing Workflow
- Conclusion

Alexander Rigato

- Develop the Architecture Design
- Design Frontend Workflow UML
- Design Backend Workflow UML
- Fulfillment of Business Requirements
- Alternatives Comparison
- Assumptions and Constraints

Phuong Bao Minh Nguyen

- Architecture Overview
- Networking and Traffic Management
- Authentication and API Gateway
- Messaging and Workflow

Hoai Thuong Triet Nguyen

- Introduction
- Architecture Overview
- Service Descriptions and Functions
- Serverless and Compute Services
- Storage and Databases

VIII. Conclusion

The architecture presented in this report addresses the key business requirements of scalability, reliability, and cost-efficiency. By leveraging managed AWS services such as ECS Fargate, Lambda, DynamoDB, and S3, the system ensures high availability and operational simplicity.

The event-driven design, utilizing SNS and SQS, promotes scalability by decoupling workloads, allowing the system to handle peak traffic efficiently. Media processing is seamlessly integrated through MediaConvert, ensuring support for diverse media formats and scalable transcoding workflows. The architecture also improves global response times using CloudFront and Route 53, ensuring users around the world experience low-latency access to the application.

Security is prioritized by integrating services such as WAF and Cognito for traffic filtering and authentication, while IAM roles ensure secure communication between components. Additionally, the serverless approach minimizes operational costs by automatically scaling resources based on demand, avoiding unnecessary infrastructure expenses.

In summary, this architecture is designed to grow with the business, meeting both current and future demands. Its modular and scalable design ensures the system can evolve as requirements change, providing a strong foundation for long-term success.

REFERENCES

[1] "Amazon Route 53," https://aws.amazon.com/route53/, accessed: 18-Oct-2024.
[2] "Amazon web application firewall (waf)," https://aws.amazon.com/waf/, accessed: 18-Oct-2024.
[3] "Amazon cloudfront," https://aws.amazon.com/cloudfront/, accessed: 18-Oct-2024.
[4] "Amazon cognito," https://aws.amazon.com/pm/cognito/, accessed: 18-Oct-2024.
[5] "Amazon api gateway," https://aws.amazon.com/api-gateway/, accessed: 18-Oct-2024.
[6] "Aws lambda," https://aws.amazon.com/pm/lambda/, accessed: 18-Oct-2024.
[7] "Amazon s3," https://aws.amazon.com/s3/, accessed: 18-Oct-2024.
[8] "Amazon sns," https://aws.amazon.com/sns/, accessed: 18-Oct-2024.
[9] "Amazon sqs," https://aws.amazon.com/sqs/, accessed: 18-Oct-2024.
[10] "Aws step functions," https://aws.amazon.com/step-functions/, accessed: 18-Oct-2024.
[11] "Amazon rekognition," https://aws.amazon.com/rekognition/, accessed: 18-Oct-2024.
[12] "Aws elemental mediaconvert," https://aws.amazon.com/mediaconvert/, accessed: 18-Oct-2024.
[13] "Vpc nat gateway," https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html, accessed: 18-Oct-2024.
[14] "Vpc internet gateway," https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Internet_Gateway.html, accessed: 18-Oct-2024.
[15] "Elastic load balancing," https://aws.amazon.com/elasticloadbalancing/, accessed: 18-Oct-2024.
[16] "Amazon ecs clusters," https://docs.aws.amazon.com/AmazonECS/latest/developerguide/clusters.html, accessed: 18-Oct-2024.
[17] "Aws fargate," https://aws.amazon.com/fargate/, accessed: 18-Oct-2024.
[18] "Amazon ecs," https://aws.amazon.com/ecs/, accessed: 18-Oct-2024.
[19] "Amazon dynamodb," https://aws.amazon.com/dynamodb/, accessed: 18-Oct-2024.
[20] "Amazon cloudwatch," https://aws.amazon.com/cloudwatch/, accessed: 18-Oct-2024.
[21] "Iam roles," https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html, accessed: 18-Oct-2024.
[22] "The sql vs nosql difference," https://www.integrate.io/blog/the-sql-vs-nosql-difference/, accessed: 19-Oct-2024.
[23] "Amazon rds vs dynamodb," https://www.qa.com/resources/blog/amazon-rds-vs-dynamodb-12-differences/, accessed: 19-Oct-2024.
[24] "Amazon dynamodb pricing," https://aws.amazon.com/dynamodb/pricing/, accessed: 19-Oct-2024.
[25] "Amazon rds pricing," https://aws.amazon.com/rds/pricing/, accessed: 19-Oct-2024.
[26] "Amazon ecs pricing," https://aws.amazon.com/ecs/pricing/, accessed: 19-Oct-2024.
[27] "Amazon ec2 pricing," https://aws.amazon.com/ec2/pricing/, accessed: 19-Oct-2024.
[28] "Aws mediaconvert: The powerful alternative to elastic transcoder," https://www.cloudthat.com/resources/blog/aws-mediaconvert-the-powerful-alternative-to-elastic-transcoder, accessed: 19-Oct-2024.
[29] "Amazon elastic transcoder pricing," https://aws.amazon.com/elastictranscoder/pricing/, accessed: 19-Oct-2024.
[30] "Aws mediaconvert pricing," https://aws.amazon.com/mediaconvert/pricing/, accessed: 20-Oct-2024.
[31] "Difference between two-tier and three-tier database architecture," https://www.geeksforgeeks.org/difference-between-two-tier-and-three-tier-database-architecture/, accessed: 20-Oct-2024.