UNIVERSITÀ DI PAVIA

Department of Electrical, Computer and Biomedical Engineering

Sunomi

A Cloud-Native Video Sharing Platform

Authors: Federico Cignoli, Davide Ferrari, Andrea Frigatti, Alessandro D. Rotta

Abstract

This report details the design, development, and deployment of Sunomi, a cloud-based video platform inspired to mirror the core functionalities of popular video-sharing services. Sunomi enables users to upload, share, and interact with video content, offering features like user authentication, automated video transcoding for adaptive streaming and playback, content searching and social interactions such as liking, commenting and user subscriptions. We present the architectural choices, technology selections, implementation details, challenges encountered and potential future enhancements, emphasizing the reasoning behind each design decision and exploring the advantages and trade-offs considered throughout the project's lifecycle. The project's primary objective was to construct a fully functional cloud-based video platform deployed on AWS that demonstrated a clear understanding of cloud computing best practices, scalability considerations, and modern web development techniques and workflows.

Contents

1 Introduction

This report serves as a comprehensive documentation of Sunomi's development lifecycle, covering its initial requirements, architectural decisions, and implementation details. It details the deployment process, highlighting the use of Infrastructure as Code (IaC) and CI/CD pipelines to automate and streamline operations. Additionally, the report explores the rationale behind adopting a cloud-native architecture, discussing how these choices influenced the project's technology stack and overall design. Lastly, we examine the challenges faced during development, the solutions implemented, and potential future improvements to enhance the platform's scalability, reliability, and functionality.

2 Requirements and Use Cases

The development of Sunomi was guided by a set of core requirements and use cases, carefully chosen to represent the essential functionalities of a video-sharing platform. These requirements were categorized into business requirements and technical requirements.

2.1 Business Requirements

- **User Account Management:** The platform supports user registration and login through two authentication methods: traditional username/password and GitHub OAuth [2], catering to different user preferences.
- **Video Upload and Management:** Registered users can upload videos with a title and description. The system automatically transcodes videos into multiple resolutions for adaptive streaming. Users can also delete their uploaded videos.
- **Video Playback:** The platform needed to provide a seamless video playback experience. This included support for adaptive bitrate streaming, allowing the video player to dynamically adjust the quality based on the user's network bandwidth and device capabilities.
- **Content Search:** Users need a way to easily find videos of interest. A search functionality was required, allowing users to search for videos based on keywords in their titles.
- **Content Moderation:** To maintain a safe and appropriate environment, the platform needed to incorporate a mechanism for content moderation.
- **Real-time Progress Updates:** To enhance the user experience during video processing, the platform was required to provide real-time updates on the transcoding progress. This allows users to monitor the status of their uploads and know when their videos are ready for viewing.

2.2 Technical Requirements

- **Scalability:** The system architecture needed to be designed to handle a large and potentially fluctuating number of users, videos, and concurrent requests. This requirement dictated the use of scalable cloud services and a microservices architecture.
- **High Availability:** The platform must ensure resilience and continuous operation through redundancy, failover mechanisms, and strategic deployment. AWS managed services, auto-scaling, and load balancing enhance availability.
- **Security:** To ensure the security of user data and video content, the platform employs HTTPS for secure communication, protecting data during transmission. Additionally, an authorization protocol is in place to manage and control user access, ensuring that only authorized individuals can perform actions based on their permissions.
- **Performance:** Video processing, upload, and delivery should be efficient to minimize latency and provide a smooth user experience.
- **Automation:** The deployment and infrastructure management were fully automated using Infrastructure as Code (IaC) with Terraform [3] and CI/CD pipelines via GitHub Actions [4] to ensure consistency and minimize errors.

- **Cloud-Native Design:** The system should be designed to leverage the benefits of cloud computing, specifically utilizing AWS services. This includes using managed services where appropriate to reduce operational overhead and improve scalability.

While Sunomi does not intend to replicate the full breadth of features found in established commercial video platforms, its scope encompasses the core functionalities essential for a video-sharing service. This focused approach allowed us to concentrate on mastering the underlying technologies and design principles, creating a solid foundation upon which further features could be built.

3 Architecture

3.1 Introduction

Sunomi is built upon a microservices architecture. This architecture was chosen as the foundational approach for several compelling reasons. Primarily, microservices offer greater scalability. Each service, representing a distinct functional unit of the platform (e.g., user management, video transcoding, content delivery), can be scaled independently based on its specific resource demands. This contrasts starkly with a monolithic architecture, where scaling often involves replicating the entire application, leading to inefficient resource utilization. From the start, we wanted this project to run on AWS and to take advantage of its services. This understanding of cloud-native principles led us to design and implement the system as a collection of services from the very beginning. We started with local development using Docker [6] to simulate the cloud environment, ensuring a seamless transition to AWS for deployment.

Microservices enhance resilience by isolating failures to individual services, preventing full system outages. They offer flexibility in technology choices, allowing each service to be developed using the programming language, framework, and database technology best suited to its particular task. Additionally, they enable independent development and deployment, enabling separate teams to work concurrently on different services and accelerating the overall development process. This agility is crucial in a rapidly evolving environment.

While microservices offer significant advantages, they also introduce complexities. Communication between services must be carefully managed, for Sunomi, the benefits of scalability, resilience, technology diversity, and independent development were deemed to outweigh the added complexities.

To enable rapid development while maintaining a cloud-native mindset, the project was initially built and tested in a local environment using Docker [6] and Docker Compose [7]. This approach allowed for fast iteration while ensuring a clear separation between development and deployment concerns. Several tools were used to replicate AWS services locally, including MinIO [5] as an alternative to Amazon S3 for object storage, MySQL [8] as a substitute for Amazon RDS, and RabbitMQ [9] to simulate Amazon SNS for asynchronous task processing. Despite beginning with a local-first approach, the architecture was designed from the start with cloud deployment in mind, ensuring a smooth transition to AWS as the project evolved.

3.2 Development Process

Development began with the implementation of core functionalities, including the controller, frontend, and transcoder. One of the first major tasks was designing the database schema. Several database architectures were considered, including NoSQL solutions, graph-based databases like Neo4j [10], and traditional relational databases. After evaluating their respective advantages and limitations, a relational database was chosen due to its structured data model and strong consistency guarantees. Once this decision was made, the focus shifted to defining the entities and relationships necessary to support the platform's functionality.

To ensure a well-organized and transparent development process, we adopted an agile approach, using GitHub Issues [11] and GitHub Projects [12] as our primary tools for task management and team coordination. GitHub Issues served as a central repository for outlining tasks, documenting bugs, and proposing new features. Larger tasks were broken down into smaller, manageable issues to promote iterative progress and facilitate easier code reviews. To ensure accountability, specific team members were assigned to individual issues. Regular team meetings were crucial for discussing progress, collaboratively resolving roadblocks, and ensuring seamless coordination across the team.

3.3 Frontend

The Sunomi frontend is a single-page application (SPA) built with React [13] and TypeScript [14]. Vite [15] is employed as a bundler and build tool, providing fast development builds and hot module replacement.

The frontend is responsible for:

- **User Interface:** The frontend is responsible for rendering all aspects of the user interface. It uses a component-based approach, breaking down the UI into reusable pieces. The Material UI (MUI) [16] component library is used to provide a consistent and visually appealing user experience.
- **API Communication:** The client communicates with the backend API (controller) via HTTP requests. It leverages an auto-generated API client produced by OpenAPI Generator [17]. This tool processes the OpenAPI specification [18] of our backend server and generates type-safe request methods that abstract the communication layer. This approach significantly speeds up development cycles compared to manually writing and maintaining the API interface and minimizes the risk of discrepancies between frontend and backend implementations, facilitating seamless updates when the backend API evolves and ensuring the controller remains the single point of truth for the API contract definition.
- **State Management:** The application manages client-side state using a combination of React's built-in useState and useContext hooks, and the react-query library [19]. React Query is used for managing asynchronous data fetching, caching, and paginated endpoints, simplifying the process of interacting with the backend API and keeping the UI in sync with the server-side data.
- **Routing:** The frontend uses React Router [20] for client-side routing, enabling navigation between different views within the SPA without full page reloads.
- **Resource Discovery and Configuration:** The application dynamically discovers the locations of other infrastructure components through a config.json file. This file defines URLs that are mapped to the appropriate servers via Amazon Route 53.

3.3.1 Deployment

- **Local Environment:** Our frontend runs in a Docker container [6], listening on specific ports. It uses Vite [15] for development mode, enabling fast hot module replacement.
- **Cloud Environment:** For cloud deployment, the frontend's static files are stored in an Amazon S3 bucket and distributed globally via CloudFront, ensuring fast and reliable access. The deployment process is automated using a GitHub Action [4], which triggers a build and pushes updated versions to the S3 bucket whenever changes are committed to the repository.

Initially, a traditional web server running in a container on ECS was considered for hosting, as it would have provided easier configurability. However, this approach came with higher costs and less scalability. Ultimately, the S3-CloudFront setup proved to be a more efficient and cost-effective solution for static content delivery.

The frontend is accessible via the sunomi.eu domain managed through AWS Route 53. To enable secure access over HTTPS, we generated an SSL certificate using AWS Certificate Manager, ensuring encrypted and authenticated communication.

3.4 Transcoder

The transcoder is a critical component of the Sunomi video platform, it is responsible for converting uploaded videos into multiple resolutions and formats suitable for adaptive bitrate streaming using the HLS (HTTP Live Streaming) protocol [21]. Implemented in Python [24], this service leverages the power and versatility of FFmpeg [22], a leading open-source multimedia framework for encoding, decoding, and transcoding media. The use of Python was chosen for its ease of development, extensive libraries, and readily available FFmpeg bindings.

The core function of the transcoder is to process a single video upload and generate multiple HLS output streams, each at a different quality level. This ensures compatibility with a wide range of user devices and network conditions, enabling adaptive bitrate streaming. The process is designed to be robust, efficient, and scalable, and follows these steps:

1. **Job Initiation:** The transcoding process begins with a trigger event.
    - **Local:** The controller publishes a message to a designated RabbitMQ [9] queue. This message contains the videoId, the MinIO [5] bucket name (bucket), and the object key (path) of the newly uploaded video. The transcoder, running as a persistent worker, consumes this message using the pika library.
    - **Cloud:** An Amazon S3 event notification, triggered by a new video object being uploaded to the designated S3 bucket, publishes a message to an Amazon SNS topic. An AWS Lambda function subscribed to this topic is invoked which launches a new AWS ECS Fargate task, passing the videoId, bucket, and path as environment variables to the transcoder container.
2. **File Download:** The transcoder uses the boto3 library to download the original video file from S3/MinIO to a temporary local directory (./tmp).
3. **Video Analysis:** The transcoder utilizes ffprobe (part of FFmpeg [22]) to extract video metadata such as resolution, frame rate and video duration. This metadata is essential for determining the appropriate transcoding settings and selecting target quality levels.
4. **Quality Level Determination:** Based on the original video's resolution, the transcoder determines the target quality levels to generate. This ensures the transcoder does not attempt to upscale the video beyond its original resolution and generates a suitable range of qualities for adaptive streaming.
5. **FFmpeg Execution:** The transcoder constructs and executes FFmpeg commands for each target quality level. The python-ffmpeg library [23] provides a convenient way to run and monitor the progress of each FFmpeg process.
6. **HLS Playlist Generation:** FFmpeg [22] generates HLS playlists for each quality level, referencing the corresponding video segments. Once all quality levels are processed, a master playlist is created, linking to the individual quality playlists and enabling adaptive bitrate streaming.
7. **Thumbnail Generation:** The transcoder utilizes FFmpeg [22] to extract a single frame from the original video and save it as a thumbnail image.
8. **File Upload:** The transcoded video segments, HLS playlists and the thumbnail image are uploaded to S3/MinIO.
9. **Progress and Status Reporting:** Throughout the transcoding process, the transcoder sends status updates.
    - **Local:** Progress updates are published to a RabbitMQ [9] queue. The controller consumes these updates and forwards them to clients.
    - **Cloud:** Progress updates are published to an SNS topic. A Lambda function, triggered by SNS, retrieves connection IDs from a DynamoDB table (populated by the WebSocket $connect handler) and sends updates to connected clients via the API Gateway WebSocket connection.
10. **Completion/Error Notification:** Once the transcoding process is complete (or if an error occurs), the transcoder sends a final status message.
11. **Temporary File Cleanup:**
    - **Local:** The transcoder deletes the temporary files and directories created during the transcoding process to prevent excessive disk usage.
    - **Cloud:** Temporary file cleanup is unnecessary as the ECS task runs on Fargate, which provides ephemeral volumes that are automatically discarded when the transcoding task is deprovisioned.

3.4.1 Parallelization

We designed the transcoder to handle all quality levels for a given video in a single process, where FFmpeg [22] processes multiple quality levels in parallel within the same container. This approach was chosen for several key reasons.

- It minimizes network traffic by downloading the source video only once from S3/MinIO, rather than requiring multiple downloads if we were to spawn separate ECS tasks for each quality level.
- It simplifies coordination by having a single process responsible for all aspects including transcoding, thumbnail generation, and publishing.
- It eliminates the complexity of tracking completion status across multiple ECS tasks and removes ambiguity about which task should handle shared responsibilities like thumbnail generation and final video publishing

While parallelization at the ECS task level was considered, we ultimately chose process-level parallelism as the operational complexity and additional network overhead of task-level parallelism outweighed its potential benefits.

3.4.2 Deployment

The transcoder is built as a Docker image [6], ensuring consistent and reproducible deployments across both local development and the AWS cloud environment. This Docker image is used in both environments, minimizing configuration differences and streamlining the deployment process. The primary difference between the local and cloud deployments lies in how transcoding jobs are initiated and how progress updates are communicated.

- **Local Environment:** The transcoder runs as a container within a Docker Compose environment [7]. It communicates with RabbitMQ [9] for message queuing and MinIO [5] for object storage.
- **Cloud Environment:** The transcoder is deployed as a containerized service on AWS ECS Fargate. Each transcoding job runs on-demand as a separate ECS task, removing the need for a queue. This allows dynamic scaling of transcoding capacity based on the number of video uploads. AWS Lambda functions, triggered by S3 and SNS, manage the initiation of this task.

3.4.3 Custom-built Transcoder vs. AWS Elemental MediaConvert

While AWS Elemental MediaConvert offers a comprehensive and readily available transcoding solution, supporting HLS, thumbnail generation, and progress notifications, we opted to develop a custom transcoder using Python [24] and FFmpeg [22]. This decision was primarily driven by the desire for complete control over the transcoding process. As a core component of a video streaming platform, the transcoder's functionality is critical, and a proprietary solution allows for fine-grained customization and optimization tailored specifically to Sunomi's requirements. This includes precise control over encoding parameters, codec selection (beyond the standard MediaConvert options), and the ability to implement custom processing steps or effects if needed. Furthermore, while MediaConvert's "Pro" tier offers extended features, a custom solution offers greater long-term flexibility and avoids potential vendor lock-in and costs associated with higher tiers.

3.5 Controller

The controller serves as the backend API for the Sunomi platform, built using Node.js and the Fastify [25] web framework. It handles client requests and manages data persistence.

3.5.1 Key Responsibilities

- **API Endpoint Handling:** The controller exposes a well-defined RESTful API, automatically documented using OpenAPI specification [18] (formerly Swagger specification). This documentation provides an interactive specification, including available endpoints, request/response schemas, and data types. Endpoints are grouped logically into modules (e.g., /auth, /upload, /user, /video).
- **User Authentication & Authorization:** The controller manages user authentication using JSON Web Tokens (JWT) [27]. Users can register/login either via username/password or through GitHub OAuth [2].

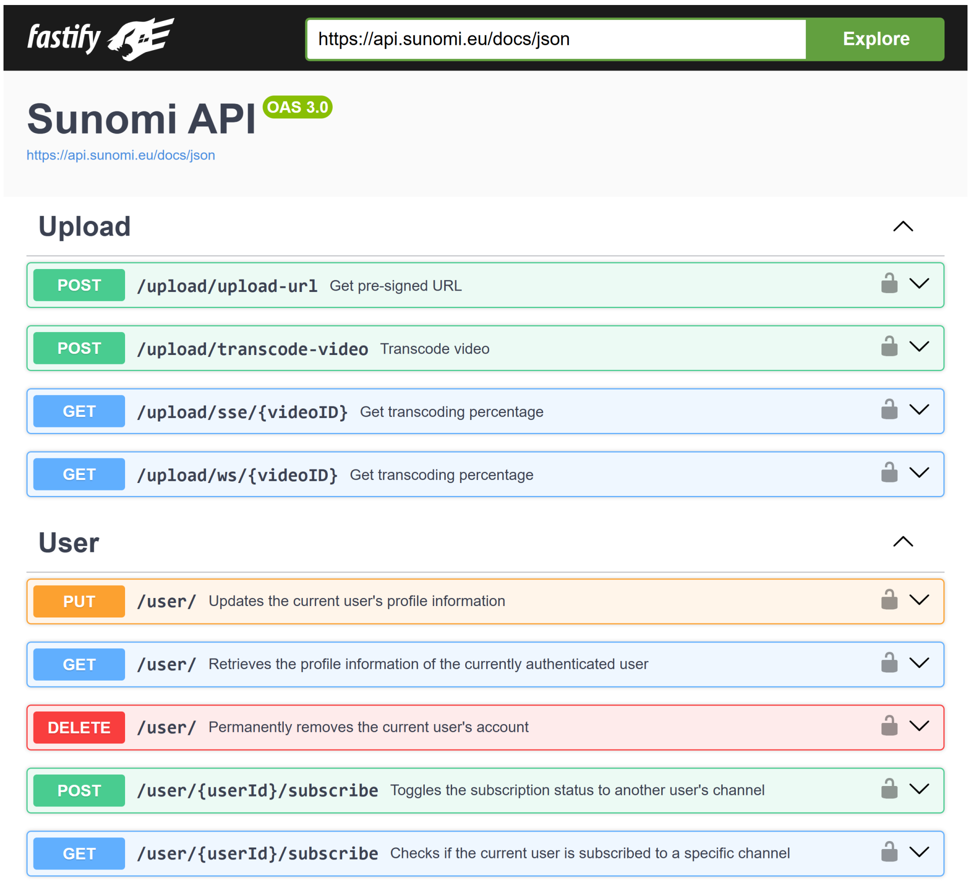
- **Data Management:** The controller uses Prisma ORM [26] for all interactions with the MySQL database [8]. Prisma provides a type-safe and developer-friendly way to query and manipulate data, abstracting away the complexities of raw SQL and improving code maintainability.
- **Paginated Endpoints:** Several endpoints, like /video/all-videos and /video/videoId/comments, have been implemented with pagination, this ensures that the platform can efficiently handle large datasets and maintain optimal performance.
- **Error Handling:** Robust error handling is implemented throughout the controller. Errors are caught, logged, and appropriate HTTP status codes and messages are returned to the client.

3.5.2 API Structure and Documentation

The controller's API is structured around resource-oriented URLs. Key endpoints include:

- **/auth:** Handles user registration, login (with both username/password and GitHub OAuth), and token validation.
- **/upload:** Provides endpoints for obtaining pre-signed S3/MinIO URLs for video uploads and initiating the transcoding process.
- **/user:** Manages user profiles, including updating user information and handling user subscriptions.
- **/video:** Provides endpoints for retrieving video details, searching for videos, managing likes and comments, and retrieving lists of videos (all videos, subscriptions).

The entire API is formally defined using an OpenAPI (Swagger) specification [18]. This specification serves as the single source of truth for the API contract, enabling automatic generation of client SDKs (used by the frontend) and interactive documentation.



Sunomi API OAS 3.0

https://api.sunomi.eu/docs/json

Upload

POST `/upload/upload-url` Get pre-signed URL

POST `/upload/transcode-video` Transcode video

GET `/upload/sse/{videoID}` Get transcoding percentage

GET `/upload/ws/{videoID}` Get transcoding percentage

User

PUT `/user/` Updates the current user's profile information

GET `/user/` Retrieves the profile information of the currently authenticated user

DELETE `/user/` Permanently removes the current user's account

POST `/user/{userId}/subscribe` Toggles the subscription status to another user's channel

GET `/user/{userId}/subscribe` Checks if the current user is subscribed to a specific channel

Figure 1: Interactive API documentation in Swagger UI

3.5.3 Authentication and Authorization

User authentication is handled using JWT (JSON Web Tokens) [27]. This approach was chosen for its stateless nature, which is well-suited for a microservices architecture and allows the controller to scale horizontally without needing to maintain a shared session store. The stateless nature of JWTs was a key factor in this choice. The process works as follows:

1. **Login**: When a user attempts to log in (using username/password or a third-party provider like GitHub OAuth [2]), the controller receives the credentials.
2. **Credential Validation**: The controller verifies the provided credentials against the stored user information (e.g., by comparing the hashed password).
3. **JWT Generation:** Upon successful authentication, the controller generates a JWT that includes a unique user ID (id). The JWT is digitally signed using a secret key retrieved from AWS Secret Manager.
4. **Token Delivery**: The generated JWT is sent back to the frontend client and stored in the browser local storage.
5. **Subsequent Requests:** For all subsequent requests to protected API endpoints, the client includes the JWT in the Authorization header.
6. **JWT Verification & Authorization**: The controller intercepts incoming requests to protected routes, validates the JWT's signature, extracts the user ID, and performs authorization checks before allowing access.

By utilizing JWT, the system benefits from a secure, scalable, and efficient authentication process that is well-suited for cloud-based and microservices architectures. Since each request carries its own authentication credentials, services can operate independently without requiring session replication or database lookups for every request, leading to improved response times and system resilience.

3.5.4 Deployment

The controller is deployed as a containerized service on ECS Fargate. This setup allows us to run Docker containers [6] without managing EC2 instances, reducing operational overhead while ensuring high availability, scalability, and resilience.

An Application Load Balancer (ALB) fronts the ECS service, distributing incoming traffic among container tasks. This ensures that if any individual controller instance fails, the ALB seamlessly redirects traffic to healthy instances. Additionally, ECS integrates with Auto Scaling to dynamically adjust the number of controller tasks based on traffic demand, scaling out during peak loads and scaling in when demand decreases.

For deployments, we utilize a Blue/Green strategy. When a new version of the controller is ready, ECS launches tasks with the updated container image alongside the existing ones. The ALB then gradually shifts traffic from the old (blue) tasks to the new (green) tasks, allowing us to monitor performance and ensure stability before fully transitioning and decommissioning the old version.

3.6 Database

Sunomi's data persistence layer is built upon MySQL [8], a widely-used, open-source relational database management system (RDBMS). MySQL was selected for its robustness, maturity, strong community support, and excellent compatibility with Prisma [26], our chosen Object-Relational Mapper (ORM). While we initially considered NoSQL databases and graph-based databases like Neo4j [10], a relational database was chosen due to its structured data model and strong consistency guarantees.

The database schema, defined within a schema.prisma file, consists of the following key tables:

- **users:** Stores user information (userID, hashed password, name, profilePictureUrl). Includes a foreign key relationship to githubUsers for GitHub OAuth authentication.
- **githubUsers:** Maps Sunomi users to their GitHub accounts (userId, githubId).
- **videos:** Stores video metadata (id, uploader's userId, title, uploadDate, status, description). Includes foreign key relationships to users, comments, likes, and views.

- **comments:** Stores video comments (id, videoId, userId, content, timestamp).
- **likes:** Represents the many-to-many relationship between users and videos they've liked (composite primary key: videoId, userId).
- **subscriptions:** Represents user subscriptions (composite primary key: subscriberId, subscribedToId, both referencing users).
- **views:** Tracks individual user views of videos (composite primary key: videoId, userId).
- **video_moderation:** Stores content moderation results from AWS Rekognition (videoId, moderation label).

Relationships between tables are enforced using foreign keys, ensuring data integrity and enabling efficient querying of related data. Cascade constraints are used to automatically remove or update dependent entities when related records are modified or deleted, maintaining consistency within the database.

3.6.1 Object-Relational Mapper

Prisma [26] serves as a crucial intermediary between the application and the MySQL database [8]. Instead of writing raw SQL queries, we interact with the database using Prisma's type-safe client, generated from the schema.prisma file. This offers several significant advantages:

- **Type Safety:** Prisma provides compile-time type checking for all database interactions. This eliminates a large class of potential runtime errors related to incorrect data types or query syntax.
- **Database Migrations:** Prisma Migrate handles schema changes in a controlled and predictable manner. We define the desired schema state in schema.prisma, and Prisma generates the necessary SQL migrations to update the database structure. This simplifies database evolution and reduces the risk of manual errors during schema updates.
- **Abstraction:** Prisma abstracts away the low-level details of database connection management, including connection pooling. This is particularly important in serverless environments like AWS Lambda, where improperly managed connections can lead to resource exhaustion. Prisma's built-in connection pooling optimizes connection reuse and prevents connection leaks, a critical consideration for scalability and stability.
- **Readability:** Prisma Client's query builder makes database interactions more readable and understandable compared to raw SQL, improving code maintainability.
- **Reduced SQL Injection Vulnerabilities:** By using parameterized queries, Prisma Client helps protect against SQL injection attacks, a common security concern with raw SQL.

3.6.2 Database Deployment: RDS and High Availability

The database is deployed using AWS Relational Database Service (RDS), a managed database service that handles many administrative tasks, such as backups, patching, and scaling. We employed two distinct RDS configurations, reflecting our development and production needs:

Development/Testing (Single Instance):

Initially, we used a free-tier RDS instance. This was a cost-effective solution for early development and testing, allowing us to experiment without significant financial commitment. This setup utilized a single database instance, located within a defined set of private subnets and protected by a security group allowing inbound traffic on port 3306 from within the VPC. This configuration is not highly available. If the single instance fails, the database becomes unavailable.

Production (Multi-AZ Aurora MySQL Cluster)

For production, we transitioned to an Amazon Aurora MySQL cluster, configured for high availability and performance.

- **Multi-AZ Deployment:** The cluster spans multiple Availability Zones (AZs) within the AWS region. Three cluster instances are created, each in a different AZ. This ensures that if one AZ experiences an outage, the database remains operational.

- **Read Replicas:** We created one writer instance and two reader instances. This allows us to offload read traffic from the primary writer instance, improving overall performance and scalability.
- **Automated Backups:** We enabled automated backups, providing a point-in-time recovery capability.
- **Encryption:** Encryption ensures that the data stored in the database is encrypted at rest.

	Development (Single Instance)	**Production (Aurora Cluster)**
Instance Type	db.t4g.micro (free tier)	db.r5.large (performance)
High Availability	No	Yes (Multi-AZ)
Read Replicas	No	Yes (2 reader instances)
Engine	mysql	aurora-mysql
Backups	Manual/None	Automated (7-day retention)
Encryption	(Optional)	Enabled
Cost	Low (free tier eligible)	Higher (pay-as-you-go)
Scalability	Limited	High

Table 1: Comparison of Development and Production Database Configurations

Database Initialization

Initially, the database schema was initialized locally using a SQL script as part of the Docker Compose setup [7]. For the AWS deployment, we created a dedicated Lambda function. This function executes the same SQL script against the RDS MySQL [8] instance during the initial Terraform deployment [3], ensuring that the database schema is correctly set up before any other services attempt to connect.

Secrets Management

Database credentials, along with other sensitive information, are securely stored in AWS Secrets Manager. This avoids hardcoding credentials directly in the codebase or configuration files, enhancing security and simplifying credential rotation.

3.7 Object Storage and Distribution

We utilize object storage for storing large binary files, primarily video files, thumbnails, and transcriptions.

For local development, MinIO [5], an S3-compatible object storage server, was used. This allowed testing of upload and storage functionality without incurring in AWS costs and without requiring an internet connection. For cloud deployment, Amazon S3 is used, providing highly scalable, durable, and cost-effective object storage. MinIO's compatibility with the S3 API played a major role in its selection, enabling a seamless transition from local to cloud environments by serving as a drop-in replacement for S3. The decision to use object storage instead of storing large video files in a relational database like MySQL [8] was driven by several factors, including scalability, cost-effectiveness, and performance. Storing this data in MySQL is neither practical nor efficient, as it leads to performance bottlenecks and increased storage costs due to the way relational databases manage large binary objects. Instead, object storage services like Amazon S3 provide optimized high-throughput file retrieval, especially when paired with a CDN like AWS CloudFront for seamless video playback. Its redundancy across Availability Zones ensures durability and availability while reducing the burden on the database, which remains focused on managing structured data. This separation of concerns enhances system responsiveness and efficiency, with video files accessed through a CDN rather than direct database queries.

The system maintains a clear, consistent file structure within the S3 bucket, following the naming structure below:

```
[videoId]/
├── master.m3u8 (Master HLS playlist)
├── original.mp4 (Original uploaded video)
├── thumbnail.jpg (Thumbnail image)
├── transcripts/
│   └── transcription.vtt (VTT transcription)
├── 480p/
│   ├── playlist.m3u8 (Variant HLS playlist)
│   ├── s_000.ts (Video segments)
│   ├── s_001.ts
│   └── ...
├── 720p/
│   ├── playlist.m3u8
│   ├── s_000.ts
│   ├── s_001.ts
│   └── ...
└── [quality_label]/
    ├── playlist.m3u8
    ├── s_000.ts
    ├── s_001.ts
    └── ...
```

Each uploaded video has a unique videoId directory storing all related files. The structure is designed for HLS [21], with a master playlist referencing variant playlists and .ts segments. Transcoded files are organized by quality (e.g., 480p, 720p, 1080p) for easy retrieval and adaptive streaming.

To efficiently handle video uploads, pre-signed URLs are used to grant temporary, time-limited access to a specific destination within the S3 bucket for writing video files. When a user requests to upload a video, the controller generates a pre-signed URL, allowing the client to upload the file without exposing AWS credentials.

To optimize content delivery, Amazon CloudFront is used as a content delivery network (CDN) to cache and distribute both video content and the frontend application at edge locations worldwide, ensuring lower latency and improved performance for users globally.

Leveraging CloudFront for CDN delivery offers multiple benefits:

- **Improved Latency and User Experience:** Content is delivered from geographically closer edge locations, reducing latency and improving load times for users globally.
- **Reduced Load on S3:** CloudFront caches content, decreasing the number of direct requests to S3, which helps lower costs and prevent S3 from becoming a bottleneck.
- **Scalability and Availability:** CloudFront is a highly scalable and reliable service. It can handle large traffic spikes without impacting performance.
- **Enhanced Security:** CloudFront can be configured to use HTTPS, protecting data in transit and restricting direct access to the S3 bucket using Origin Access Identities (OAI) / OAC.

In summary, video playback is exclusively handled via CloudFront, which caches content from the S3 video bucket. Video uploads use temporary, tightly controlled pre-signed URLs that grant write access to specific locations within the bucket. This combined approach provides a secure, scalable, and performant solution for managing video content.

3.8 Lambda and Event-Driven Design (SNS)

Within Sunomi's architecture, several Lambda functions operate within a completely event-driven design, primarily orchestrated by the use of Amazon SNS (Simple Notification Service). This pattern allows for asynchronous, decoupled communication between various services, promoting scalability and resilience.

As shown in Section 4.1, one set of functions is triggered by messages published to a central "video upload" SNS topic. This topic receives notifications directly from S3 whenever a new video file with a specific suffix (original.mp4) is uploaded to the designated bucket. Subscriptions to this topic automatically invoke three separate Lambda functions. One initiates the video transcoding process by creating a new ECS task, another starts a transcription job using AWS Transcribe, and the third triggers a content moderation analysis using AWS Rekognition. Another Lambda function is subscribed to a "moderation results" SNS topic. This topic receives notifications from Rekognition upon completion of the content moderation analysis. The Lambda function, triggered by this SNS subscription, processes the Rekognition results and stores them in the database.

For real-time transcoding updates, a dedicated function is subscribed to the transcoder-status SNS topic. Messages published by the transcoder (containing video ID and status updates) trigger this function, which then retrieves relevant WebSocket connection IDs from DynamoDB and sends updates to connected clients. Finally, a publication lambda is subscribed to transcoder_status SNS, but with a filter policy, to receive messages only with a specific prefix. All of these Lambda functions benefit from SNS's publish-subscribe model; they are invoked only when relevant events occur, and they don't need to poll for changes, creating a highly responsive and efficient system.

The combination of AWS Lambda, SNS, and an event-driven architecture within Sunomi unlocks a multitude of benefits, extending beyond just asynchronous communication:

- **Serverless Efficiency and Cost Optimization:** Lambda functions are inherently serverless. This means Sunomi doesn't need to provision, manage, or scale any servers to handle events like video uploads, transcoding completion, or moderation results. AWS automatically manages the underlying compute resources, scaling them up or down precisely in response to the number of incoming events. This "pay-per-use" model drastically reduces operational overhead and often results in significant cost savings, as Sunomi only pays for the compute time consumed when the Lambda functions are actually executing. Idle time incurs no cost. This contrasts sharply with traditional server-based approaches, which require maintaining servers even during periods of low activity.
- **Enhanced Scalability and High Availability:** The event-driven nature, powered by SNS, naturally promotes high scalability and availability. SNS is a highly available, distributed service designed to handle massive message volumes. When an event occurs (e.g., a video upload), SNS reliably delivers the message to all subscribed endpoints (Lambda functions, in this case). Lambda automatically scales the number of concurrent function executions to match the incoming event rate. This ensures that Sunomi can handle spikes in video uploads, transcoding requests, or content moderation analyses without performance degradation.
- **Decoupling and Fault Isolation:** The event-driven design, using SNS as a message broker, completely decouples the different components of Sunomi. The service that produces an event (e.g., S3 when a video is uploaded) doesn't need to know anything about the services that consume the event (e.g., the Lambda functions that start transcoding, transcription, and Rekognition). This loose coupling makes the system more resilient to failures. If the transcoding Lambda function is temporarily unavailable, the S3 upload event still gets published to SNS, and the Lambda function will be invoked when it recovers. The failure of one component doesn't cascade and bring down the entire system.

3.9 Other AWS Services Used

In addition to the core services described above, Sunomi leverages several other AWS services:

- **AWS API Gateway:** Provides a managed WebSocket API for real-time retrieval of upload status updates coming from the transcoder, ensuring that users receive immediate feedback on their upload processing. The API Gateway handles connection management, message routing, and scaling. It is integrated with Lambda functions to handle connection events ($connect, $disconnect). A custom domain name (ws.sunomi.eu) is configured for the WebSocket API.

SSE/WebSocket connection management was moved from the controller (in local environment) to API Gateway when moving to AWS in order to achieve service decoupling. This ensures that when controllers scale, especially in a scale-in scenario, connections are not lost. This follows the principles of event-driven architecture, allowing for better fault tolerance and scalability.

- **AWS Certificate Manager:** ACM is used to provision and manage SSL/TLS certificates for secure HTTPS communication. Two separate SSL/TLS certificates are provisioned using ACM to meet different service requirements. One certificate is issued in eu-west-1 and used in API Gateway and the Application Load Balancer for the controller ECS service. The other certificate is issued in us-east-1, as required by CloudFront, to enable secure HTTPS communication for distributed content delivery.
- **Route 53:** Route 53 is used to manage DNS records for the custom domain names used by the platform (e.g., api.sunomi.eu, video.sunomi.eu, ws.sunomi.eu, www.sunomi.eu). Route 53 is configured to route traffic to the appropriate AWS resources.
- **Secrets Manager**: Secrets Manager is used to store sensitive information, such as database credentials, JWT secrets [27], and GitHub OAuth client secrets [2]. This avoids hardcoding sensitive information in the codebase or configuration files, improving security.
- **DynamoDB:** DynamoDB, a NoSQL database, is used by the API Gateway and WebSocket Lambdas to efficiently store and manage active WebSocket connections.
- **AWS Rekognition:** Provides video analysis services, primarily used for detecting moderation labels in videos to identify sensitive content.
- **AWS Transcribe:** Used for generating VTT transcriptions of videos, enabling automatic speech-to-text conversion.

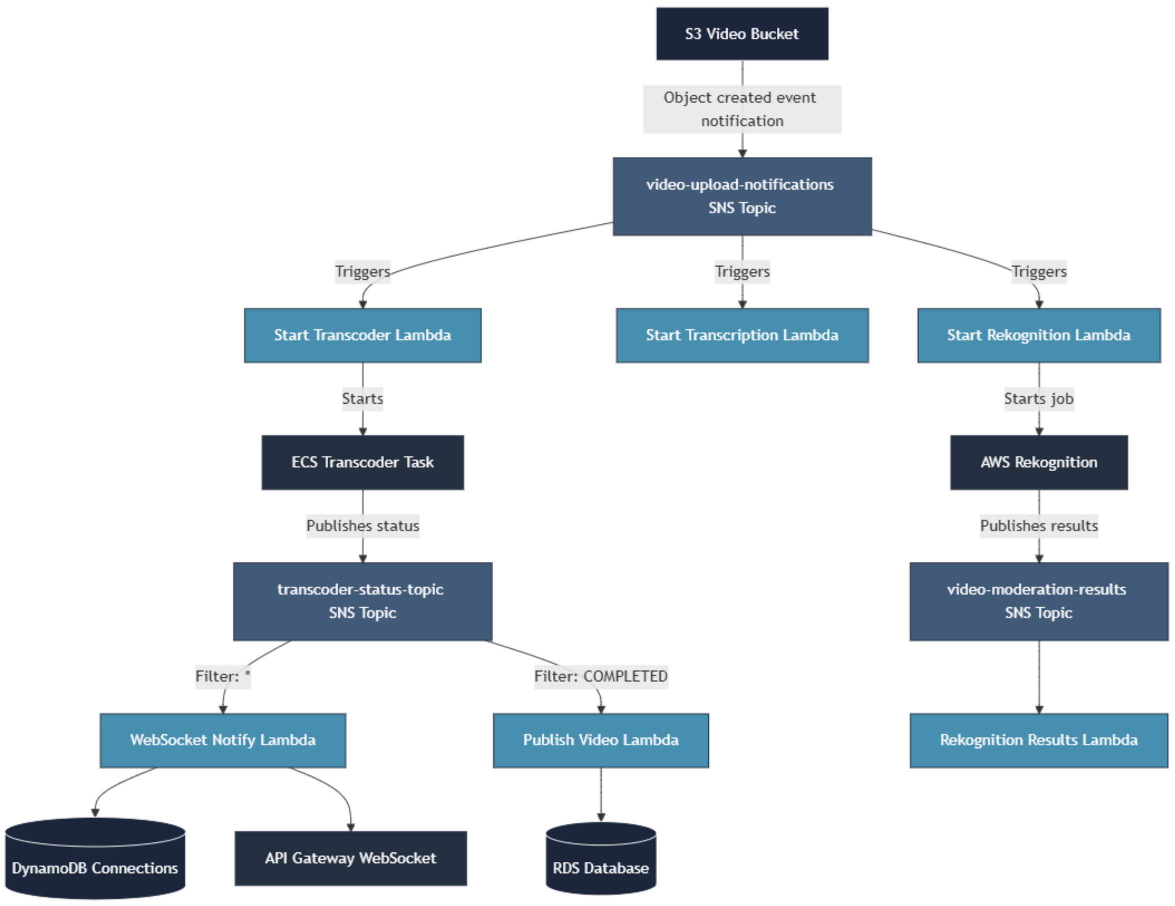
3.10 Networking

The Sunomi video platform's AWS networking setup, managed via Terraform [3], prioritizes security, scalability, and availability. A dedicated VPC (10.0.0.0/16) hosts public and private subnets across three availability zones.

- **Private Subnets:** Three private subnets are created, one in each availability zone.
  - 100.0.0.0/24
  - 100.0.1.0/24
  - 100.0.2.0/24
- **Public Subnets:** Three public subnets are created, mirroring the availability zone distribution of the private subnets.
  - 100.0.128.0/24
  - 100.0.129.0/24
  - 100.0.130.0/24
- Internet Gateway enables public subnet internet access.
- NAT Gateway in a public subnet allows outbound internet access for private subnets while blocking inbound traffic, requiring an Elastic IP.
- Routing: Public subnets use an Internet Gateway, while private subnets route external traffic through the NAT Gateway.
- Security Groups: Control access for database, Lambdas, ECS controller and ECS transcoder.
- VPC Endpoint: An S3 Gateway Endpoint enables private subnet access to S3 without using the NAT Gateway, reducing costs and improving performance.

4 Flows

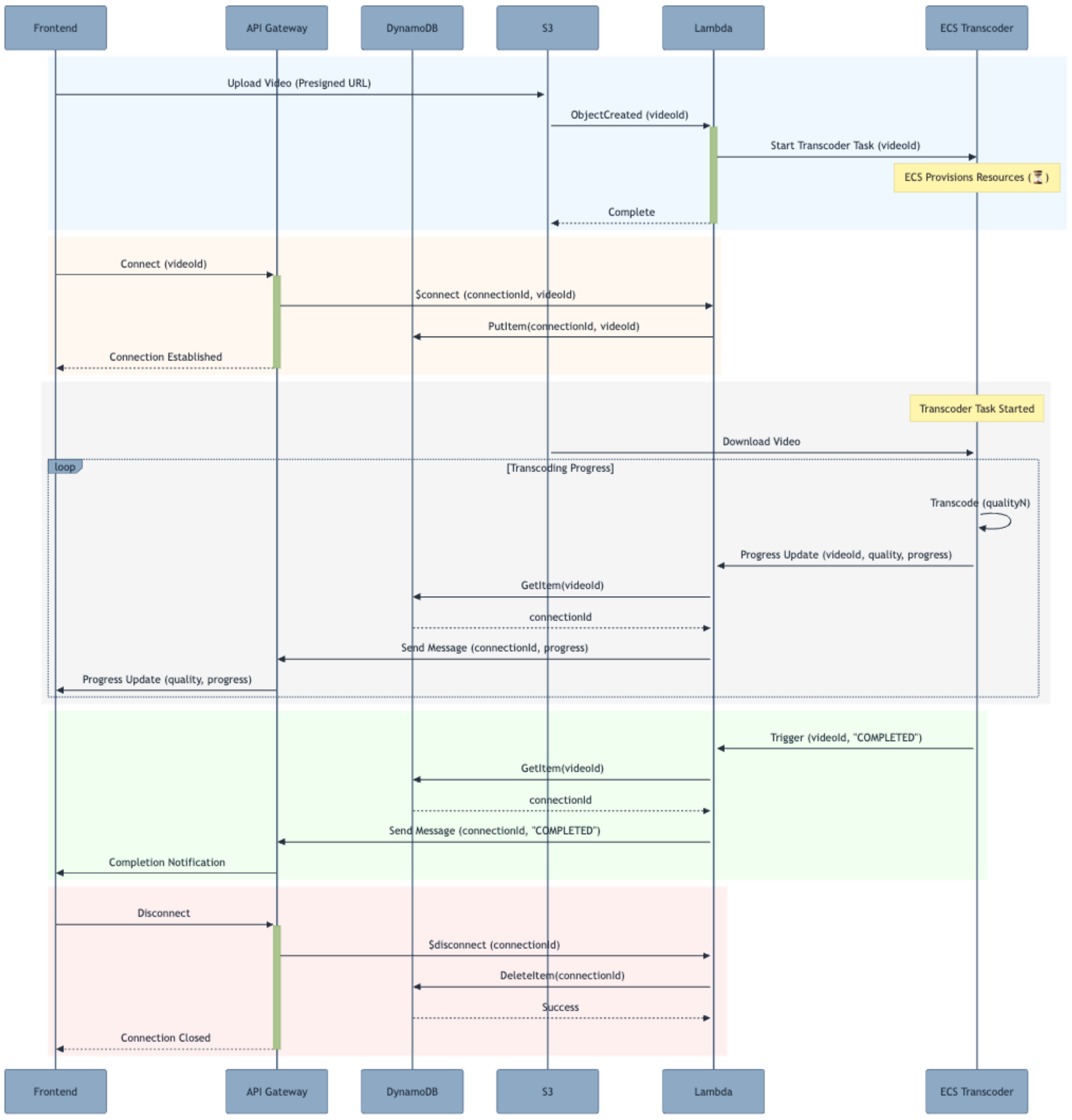
4.1 AWS Video Upload and Processing Workflow



This diagram illustrates the AWS-based upload flow for a video processing system, showcasing the interaction between various AWS services. When a user uploads an original video file to the designated S3 Video Bucket, an `"ObjectCreated"` event triggers an SNS topic named `"video-upload-notifications"`. This SNS topic, in turn, simultaneously starts three separate Lambda functions: "Start Transcoder", "Start Transcription", and "Start Rekognition." The "Start Transcoder" Lambda initiates an ECS Transcoder Task, which handles the video transcoding process and publishes status updates to another SNS topic, "transcoder-status-topic". The "Start Transcription" Lambda utilizes AWS Transcribe to generate a `.vtt` transcription of the uploaded video, and the "Start Rekognition" Lambda starts a content moderation job using AWS Rekognition.

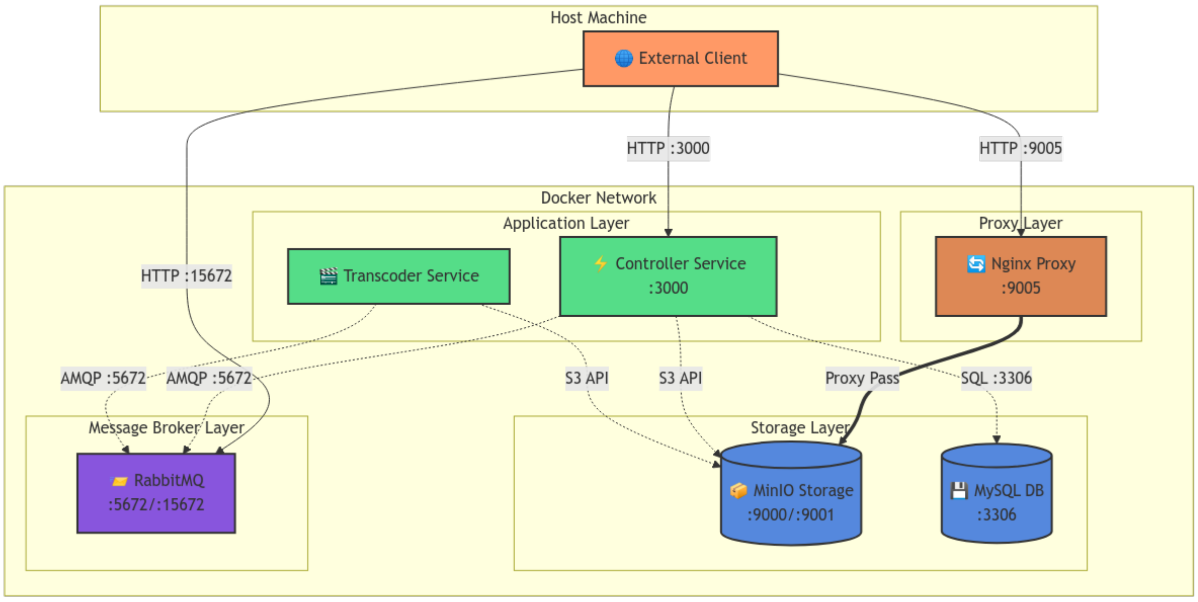
The `"transcoder-status-topic"` SNS topic has two filters. The first filter, a wildcard ("*"), forwards all status updates to the "WebSocket Notify" Lambda function. This Lambda function interacts with a DynamoDB table ("DynamoDB Connections") to manage WebSocket connections and relays these updates to connected clients via API Gateway, providing real-time progress information. The second filter specifically targets `"COMPLETED"` status messages, triggering the "Publish Video" Lambda, which updates the video's status in the RDS database to mark it as ready for public viewing. Finally, AWS Rekognition publishes moderation results to the `"video-moderation-results"` SNS Topic, which triggers the "Rekognition Results" Lambda to process and store these findings, likely flagging inappropriate content. This flow ensures efficient parallel processing of uploaded videos, including transcoding, transcription, and content moderation, while also providing real-time progress updates and a mechanism for database management.

4.2 Asynchronous Video Transcoding Progress Reporting


Frontend
API Gateway
DynamoDB
S3
Lambda
ECS Transcoder
Upload Video (Presigned URL)
ObjectCreated (videoId)
Start Transcoder Task (videoId)
ECS Provisions Resources (⏳)
Complete
Connect (videoId)
$connect (connectionId, videoId)
PutItem(connectionId, videoId)
Connection Established
Transcoder Task Started
Download Video
loop
[Transcoding Progress]
Transcode (qualityN)
Progress Update (videoId, quality, progress)
GetItem(videoId)
connectionId
Send Message (connectionId, progress)
Progress Update (quality, progress)
Trigger (videoId, "COMPLETED")
GetItem(videoId)
connectionId
Send Message (connectionId, "COMPLETED")
Completion Notification
Disconnect
$disconnect (connectionId)
DeleteItem(connectionId)
Success
Connection Closed

The sequence diagram portrays an event-driven, asynchronous video processing workflow orchestrated within AWS. It details the process from the client's initial video upload to S3 using a presigned URL, which then triggers a Lambda function to initiate an ECS task for transcoding. It illustrates the establishment of a WebSocket connection via API Gateway, managed through connection mappings stored in DynamoDB, enabling real-time progress updates to the client during transcoding. Upon completion, a notification is sent, and the WebSocket connection is terminated, highlighting the architecture's reliance on serverless components and event-based triggers to achieve scalability and responsiveness.

4.3 Local Development Environment



This diagram illustrates the local development architecture of the video streaming platform, showcasing the interaction between various services within a Dockerized environment. The system is composed of several key layers: Storage (MinIO for video files and MySQL for metadata), Message Broker (RabbitMQ for asynchronous communication), Application Services (Controller for API handling and Transcoder for video processing), and a Proxy layer (Nginx for routing external requests to MinIO). External clients, represented by a globe icon, interact with the system primarily through the Controller's API (port 3000) and the Nginx proxy (port 9005) for accessing video content stored in MinIO. Internal communication, depicted with dashed arrows, occurs between the Controller and the Storage/Message Broker layers, utilizing protocols like AMQP for RabbitMQ, S3 API for MinIO, and SQL for MySQL. The Transcoder service similarly interacts with RabbitMQ and MinIO.

Nginx acts as a proxy to make MinIO's presigned URLs accessible from outside Docker. Without it, the Controller would return URLs only valid within Docker's internal network (e.g., http://minio:9000/...). Nginx listens on port 9005 and forwards these requests to MinIO, allowing external clients to access the presigned URLs through localhost:9005 instead.

5 Infrastructure as Code

The infrastructure underpinning the Sunomi platform is fully managed using Terraform [3], embodying the principles of Infrastructure as Code (IaC). This declarative approach allows for the automated provisioning and consistent configuration of all required AWS resources, from networking components like the VPC and subnets to compute resources like ECS clusters and Lambda functions, and data stores like RDS and S3. Chosen for its maturity, robust dependency management, and ease of integration with the AWS ecosystem, Terraform facilitates version control, collaboration, and simplified change management. This means the entire infrastructure can be treated as code, stored in a repository, and easily replicated or modified. The configuration is organized into modular units, each dedicated to a specific aspect of the infrastructure, promoting maintainability, reusability, and a clear separation of concerns. This modularity makes updates and modifications easier and less prone to unintended side effects. These units encompass networking, compute, storage, security, and application-specific services, ensuring a well-structured and organized approach to infrastructure management.

The configuration includes provider setup, ensuring AWS connectivity and SSL/TLS compliance, and core networking elements such as VPCs, subnets, and routing for secure and efficient traffic management. Network gateways, including Internet and NAT Gateways, facilitate controlled internet access while security groups enforce firewall rules for different services.

The deployment structure encompasses ECS-based services for controllers and transcoders, each configured with defined compute resources, networking, and security settings. Database setup is managed via Lambda functions, which initialize schema creation and handle workflow automation for video processing. AWS services like SNS coordinate event-driven processes such as transcoding, transcription, and content moderation.

The API Gateway, specifically configured for WebSocket-based real-time communication, integrates with backend services. CloudFront ensures optimized content delivery, securing connections with managed SSL/TLS certificates. Object storage in S3 is used for videos, thumbnails, and static assets, with strict access policies and encryption.

Security measures include IAM roles for access control, Secrets Manager for sensitive data handling, and certificate management via AWS Certificate Manager. Input variables allow configuration customization for different environments, making the Terraform setup adaptable across development and production.

6 CI/CD Pipelines

We used GitHub Actions [4] to implement Continuous Integration and Continuous Deployment (CI/CD) pipelines, automating the build and deployment process. GitHub Actions was a natural choice due to its tight integration with our source code repository[1] and ease of use, allowing us to hook into repository events to trigger specific workflows.

The AWS credentials used by GitHub Actions belong to a specific AWS IAM user that has only the necessary permissions required for the operations performed in these workflows. These credentials are securely stored in GitHub repository environment secrets and configured within the workflows.

Our CI/CD setup consisted of two main pipelines:

6.1 Automated Docker Image Build and Deployment

This workflow is responsible for building Docker images [6] for the controller and transcoder-aws services, pushing them to AWS Elastic Container Registry (ECR), and redeploying the relevant Elastic Container Service (ECS) services if necessary.

The workflow is triggered automatically when changes are made to the source code of the controller or transcoder-aws components. Manual triggering is also possible when needed.

Steps:

1. **Checkout:** The repository is cloned to the runner environment.
2. **Configure AWS credentials and login to AWS ECR:** Credentials are retrieved, configured, and used to authenticate with AWS ECR.
3. **Build and push Docker images:** This step builds Docker images for both services (controller and transcoder-aws), tags them with both latest and the current Git commit SHA, and then pushes them to AWS ECR. Tagging with the Git commit SHA allows for rollbacks in AWS if the new deployment fails.

Additional steps for the controller service:

5. **Retrieve and update ECS task definition:** The existing ECS task definition is retrieved and updated with the new image tag. This shifts the responsibility away from the GitHub Action for defining the task definition, ensuring that it only updates the one already provisioned by the Terraform deployment [3].
6. **Start blue/green deployment:** The deployment follows a blue/green rollout strategy, where a new task set is created and gradually replaces the existing one. This ensures zero-downtime deployments by running the new version alongside the old one until the transition is complete.
7. **Handle rollback and failover:** If issues arise, traffic can quickly be redirected back to the previous version, minimizing disruptions and maintaining service stability.

6.2 Frontend Application Build and Deployment

This workflow is responsible for building the frontend React application [13] and deploying it to an AWS S3 bucket. It ensures that the frontend is always up to date with the latest changes from the repository. Steps:

1. **Checkout:** The source code is cloned.
2. **Setup Node.js and install dependencies:** The appropriate Node.js version is installed, and the frontend dependencies are then installed.
3. **Build Swagger documentation:** This ensures that the frontend and other services are always up to date with the latest controller API definitions [18], reducing the risk of integration issues. By generating the documentation at build time, potential mismatches between the frontend and backend can be detected early, improving development efficiency and reliability.
4. **Build React application:** The frontend is built into a production-ready bundle.
5. **Configure AWS credentials and deploy to S3:** Credentials are retrieved and configured, then the build output is synchronized to an S3 bucket, overriding the previous files.

By automating the entire build and deployment process, our CI/CD pipelines significantly improved our development workflow. Manual deployment efforts were reduced, minimizing human errors, while all changes were deployed in a consistent and reproducible manner. The pipelines enabled rapid iteration by allowing automatic and on-demand deployments.

7 Conclusions

7.1 Challenges and Solutions

The development of Sunomi presented several noteworthy technical challenges, requiring careful consideration and innovative solutions. This section details the most significant obstacles encountered and the strategies employed to overcome them.

Ensuring Robust and Scalable Video Transcoding: The implementation of a reliable and performant video transcoding pipeline presented a substantial initial challenge. The inherent complexity of FFmpeg [22], coupled with the need to support diverse input formats and ensure resilience against processing errors, demanded a meticulous approach. To address this, we leveraged the python-ffmpeg library [23] to provide a more structured interface for interacting with FFmpeg. The transcoding process was carefully orchestrated to handle multiple quality levels sequentially, and robust error handling mechanisms were incorporated to gracefully manage potential failures and facilitate retries. The use of AWS Fargate enabled the deployment of numerous transcoder instances, providing horizontal scalability.

Achieving Real-Time Transcoding Progress Updates: Providing users with real-time feedback on the progress of their video uploads was a critical requirement. Initial explorations with Server-Sent Events (SSE) [29] revealed limitations in standard implementations, particularly concerning the secure transmission of authentication credentials. Consequently, we transitioned to a WebSocket-based architecture, utilizing AWS API Gateway and Lambda functions to manage persistent connections. A DynamoDB table was employed to maintain a registry of active connections, indexed by video identifiers. This architectural shift, while introducing greater complexity, resulted in a more robust and scalable solution for real-time communication.

Coordinating Asynchronous Processing Operations: The video processing workflow involved a series of asynchronous operations, including transcoding, transcription generation, and content moderation analysis. Coordinating these tasks and ensuring the correct execution of each step required a robust and reliable mechanism. We employed AWS Simple Notification Service (SNS) to facilitate asynchronous communication between services. SNS topics and subscriptions, coupled with comprehensive error handling within Lambda functions, provided the necessary framework for managing this distributed workflow.

Harmonizing Local Development and Cloud Deployment: Maintaining consistency between the local development environment, which utilized MinIO [5] and RabbitMQ [9], and the production cloud deployment, which leveraged AWS S3 and SNS, presented a logistical challenge. Discrepancies between these environments could introduce inconsistencies in application behavior and complicate testing and debugging. To address this, we employed a combination of conditional logic and environment variables. This allowed the application to adapt its behavior based on the detected environment, ensuring consistent operation across both local and cloud contexts.

Pre-signed URL Accessibility in Local Environment: A significant hurdle arose from the interplay of Docker containerization [6] and the generation of pre-signed URLs for direct file uploads to MinIO [5] during local development. The Dockerized controller, operating within its isolated network, generated pre-signed URLs that were valid only for internal network clients. This presented a critical issue when the frontend, residing outside the Docker network, attempted to utilize these URLs. Consequently, the frontend was unable to access the designated storage location. To overcome this limitation, we deployed Nginx [28] as a reverse proxy. Nginx acted as an intermediary, effectively bridging the gap between the internal Docker network and the external environment. By routing traffic through Nginx, the externally facing URLs were correctly mapped to the internal Docker hostnames, enabling seamless and secure file uploads from the frontend, irrespective of its network location.

7.2 Future Developements

The current testing strategy for Sunomi primarily relies on manual testing alongside code quality checks using linters and formatters. Recognizing the need for a more robust and automated approach, the implementation of a comprehensive testing suite is planned as a future development. The ECS service for the controller incorporates rollback logic that can automatically revert to a previous stable version in the event that a new deployment encounters issues.

While Sunomi provides a solid foundation for a video platform, numerous opportunities exist for future enhancements and expansion. Examples include: implementing a comprehensive automated testing suite, improving error handling and logging capabilities, and enhancing overall security hardening. Other avenues for future development encompass frontend improvements, richer video metadata storage, and streamlining the video transcoding process for improved efficiency.

7.3 Final Conclusion

Sunomi is a functional and robust cloud-based video platform built upon a microservices architecture, demonstrating a practical application of modern software engineering principles. By leveraging key AWS services and a well-defined API, Sunomi achieves a scalable, resilient, and feature-rich application. The project embodies DevOps best practices through automated infrastructure provisioning with Terraform [3], CI/CD pipelines with GitHub Actions [4], and Docker-based containerization [6]. As Yoda would say, "*Difficult to see; always in motion is the future*". While the existing features provide a solid foundation, future enhancements such as live streaming and improved content moderation offer compelling opportunities for further growth and expansion, reinforcing Sunomi's potential as a competitive video platform.

References

[1] Sunomi Repository. (n.d.). https://github.com/ferraridavide/cloud

[2] GitHub. (n.d.). Authorizing OAuth Apps. GitHub Docs. https://docs.github.com/en/apps/oauth-apps/building-oauth-apps/authorizing-oauth-apps

[3] HashiCorp. (n.d.). Terraform. https://www.terraform.io/

[4] GitHub. (n.d.). Features – GitHub Actions. https://github.com/features/actions

[5] MinIO. (n.d.). MinIO. https://min.io/

[6] Docker. (n.d.). Docker. https://www.docker.com/

[7] Docker. (n.d.). Overview of Docker Compose. Docker Documentation. https://docs.docker.com/compose/

[8] MySQL. (n.d.). MySQL. https://www.mysql.com/

[9] RabbitMQ. (n.d.). RabbitMQ. https://www.rabbitmq.com/

[10] Neo4j. (n.d.). Neo4j. https://neo4j.com/

[11] GitHub. (n.d.). About Issues. GitHub Docs. https://docs.github.com/en/issues

[12] GitHub. (n.d.). About Projects. GitHub Docs. https://docs.github.com/en/issues/planning-and-tracking-with-projects

[13] React. (n.d.). React. https://react.dev/

[14] Microsoft. (n.d.). TypeScript. https://www.typescriptlang.org/

[15] Vite. (n.d.). Vite. https://vitejs.dev/

[16] Material UI. (n.d.). Material UI. https://mui.com/

[17] OpenAPI Generator. (n.d.). OpenAPI Generator. https://openapi-generator.tech/

[18] OpenAPI Initiative. (n.d.). OpenAPI Specification. Swagger. https://swagger.io/specification/

[19] TanStack. (n.d.). React Query. TanStack Query. https://tanstack.com/query/latest

[20] Remix. (n.d.). React Router. https://reactrouter.com/en/main

[21] Apple. (n.d.). HTTP Live Streaming. Apple Developer. https://developer.apple.com/streaming/

[22] FFmpeg. (n.d.). FFmpeg. https://ffmpeg.org/

[23] KKroening. (n.d.). ffmpeg-python. GitHub. https://kkroening.github.io/ffmpeg-python/

[24] Python Software Foundation. (n.d.). Python. https://www.python.org/

[25] Fastify. (n.d.). Fastify. https://www.fastify.io/

[26] Prisma. (n.d.). Prisma. https://www.prisma.io/

[27] JWT.io. (n.d.). JSON Web Tokens. https://jwt.io/

[28] NGINX. (n.d.). NGINX. https://www.nginx.org/

[29] Mozilla. (n.d.). Server-Sent Events. MDN Web Docs. https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events