Smart Media Uploader – Cloud-Native Backend

A production-style, event-driven media processing backend built on AWS using **FastAPI, S3, SQS, Step Functions, Lambda, DynamoDB, and ECS Fargate**.

This system allows users to upload images, audio, or video files and processes them asynchronously (validation, classification, thumbnails, transcription, transcoding, metadata) with full observability, retries, and fault-tolerance.

🧠 High-Level Architecture

```
Client
  ↓
FastAPI (ECS Fargate)
  ↓
S3 (Presigned Upload)
  ↓
DynamoDB (Job record)
  ↓
SQS (Jobs Queue)
  ↓
Lambda (Dispatcher)
  ↓
Step Functions (Workflow Brain)
  ↓
Lambda / ECS Workers
  ↓
S3 (Processed Outputs)
  ↓
DynamoDB (Job Status + Metadata)
```

📦 Tech Stack

Layer	Technology
API	FastAPI on ECS Fargate
Storage	Amazon S3
Database	DynamoDB
Messaging	Amazon SQS
Orchestration	AWS Step Functions

Layer	Technology
Lightweight compute	AWS Lambda
Heavy compute	ECS Fargate
Infrastructure	Terraform
CI/CD	GitHub Actions
State & locking	S3 + DynamoDB backend

🚀 Development Phases

🏗 Phase 0 — Infrastructure Foundation

Goal: Create a production-grade AWS foundation using Terraform and CI/CD.

What we built

- Remote Terraform state:
  - S3 backend
  - DynamoDB state lock
- Modular Terraform structure:
  - networking
  - compute
  - messaging
  - data
  - orchestration
  - storage
- VPC, subnets, IGW
- S3 buckets:
  - `smmu-dev-raw-media`
  - `smmu-dev-processed-media`
- DynamoDB tables:
  - `smmu-dev-jobs`
  - `smmu-dev-media`
- SQS queues:
  - `smmu-dev-jobs-queue`
  - `smmu-dev-jobs-dlq`

- IAM roles with least-privilege
- GitHub Actions pipeline with remote state

Problems solved

- Terraform state was local → migrated to S3
- CI/CD could not access AWS → fixed credentials
- State lock conflicts → resolved via DynamoDB

What was learned

- How real teams manage Terraform state
- Why remote state + locking prevents infrastructure corruption
- How CI/CD deploys cloud infrastructure safely

Phase 1 — API Layer (FastAPI on ECS)

Goal: Expose a production-grade API.

What I built

- FastAPI service running in ECS Fargate
- Docker image pushed to ECR
- Application Load Balancer
- HTTPS-ready architecture
- ECS service auto-deployable via Terraform

What I validated

- ALB target group health checks
- ECS task networking
- IAM permissions to pull images from ECR
- CloudWatch logs

Problems solved

- ECS tasks stuck in PENDING → fixed VPC + NAT + ECR access
- ALB not routing → fixed target group & security groups

What was learned

- How containers run in AWS
- How ALB connects to ECS
- How production APIs are hosted without servers

Phase 2 — Asynchronous Workflow Engine

This is where the system became **distributed**.

What we built

SQS-driven pipeline

- API (later) sends jobs to SQS
- SQS triggers Lambda

Dispatcher Lambda

- Reads job from SQS
- Starts Step Functions execution

Step Functions (the brain)

```
ValidateInput → DetectType → Branch → UpdateStatus
```

Workflow Lambdas

- `validate_input`
- `detect_type`
- `update_status`

DynamoDB integration

- Job status updated by workflow
- Verified with real data

What I validated

```
{"jobId":"test-123","type":"image","inputKey":"s3://fake"}
```

Result:

```
SQS → Lambda → Step Functions → Lambdas → DynamoDB
```

And DynamoDB showed:

```
status = RUNNING
```

This proves:

- Queue → Workflow → Database → Orchestration works

Problems we debugged

Problem	What it taught
Invalid JSON from SQS	How SQS wraps messages
Lambda missing permissions	Real IAM debugging
Step Functions not triggering	Event source mapping
Missing fields in state	How Step Functions passes JSON
CI couldn't find ZIP files	Artifact build vs IaC separation

🧩 Phase 3 — Distributed ECS Workers (Media Processing Layer)

Goal

Move all heavy media processing out of Lambdas and into **containerized ECS workers** that can scale independently and handle CPU-intensive jobs like transcoding, image processing, and transcription.

This phase converts the platform from **serverless-only** to a **hybrid serverless + container compute system** — the architecture used by real production media pipelines.

What was built

1️⃣ Worker container system

We created **dedicated ECR repositories** for each worker type:

Worker	ECR Repository
Image Transcoder	`smmu-dev-transcode-worker`
Video Processor	`smmu-dev-video-worker` (future)
Audio Processor	`smmu-dev-transcribe-worker`

Each worker is:

- Built as a Docker image
- Pushed to ECR
- Executed on-demand via ECS Fargate

2️⃣ ECS worker infrastructure

We deployed:

- ECS Cluster
- Task Definitions for each worker
- IAM execution role for pulling from ECR
- IAM worker role for accessing S3 + DynamoDB
- CloudWatch log groups for worker logs

Workers do **not run 24/7**.
They are **started by Step Functions only when a job arrives**.

This is how Netflix-style pipelines save money.

---

3 Step Functions → ECS integration

The ImagePipeline state was upgraded from a placeholder to:

```
Step Functions → ECS RunTask (Fargate) → Worker Container
```

This means:

- Step Functions starts the container
- Passes job parameters (jobId, inputKey, outputKey)
- Waits for the container to finish
- Moves to the next state automatically

This turns ECS into a **true serverless worker pool**.

---

4 Real end-to-end flow

A full image job now executes:

```
API
→ S3 (upload)
→ SQS
→ Dispatcher Lambda
→ Step Functions
→ ECS Transcode Worker
→ S3 (processed file)
→ DynamoDB (status updated)
→ Step Functions → MarkCompleted
```

No manual triggers.
No polling.
No servers running idle.

5 What I debugged and fixed

This phase involved real production-grade issues:

Problem	What it taught
ECS couldn't pull images	ECR auth via IAM
Logs not appearing	CloudWatch log groups must exist
S3 403 / 404	Bucket IAM + correct keys
Step Functions failing	How ECS returns task status
Payload lost	`ResultPath` vs overwrite

🧠 What Phase 3 teaches

I understood:

- How Step Functions orchestrate containers
- How ECS Fargate works without servers
- How AWS IAM gates every service
- How to debug distributed systems using logs + state machines
- How production media pipelines really run

◇ Phase 4 — Multi-Worker Media Processing (ECS Integration)

In Phase 4 we transformed the system from a "single worker pipeline" into a **true distributed media processing platform**.

This phase added **multiple ECS workers** and wired them into the **Step Functions brain**, allowing different media types to be processed by different containers.

What we built

We introduced **two independent ECS workers**:

Worker	Purpose
Transcode Worker	Processes images (and currently MP4 test media)
Transcribe Worker	Extracts / transcribes audio from media

Each worker:

- Runs as an **ECS Fargate task**
- Pulls its own Docker image from **ECR**

- Uses **IAM roles** to securely access S3, DynamoDB, and CloudWatch
- Writes its own logs to **CloudWatch**

Step Functions Brain (Upgraded)

The Step Functions pipeline was upgraded from simple Lambdas to a **hybrid orchestration engine**:

```
SQS → Dispatcher Lambda → Step Functions
                                |
                        Detect Media Type
                                |
          ┌─────────────┬──────────────┬─────────────┐
          |             |              |
       Image          Audio          Video
          |             |              |
   ECS Transcode   ECS Transcribe   (Parked)
          |             |
     Update Status   Update Status
```

The key architectural change:

> **Lambdas decide.**
> **ECS does the heavy work.**

Step Functions now launches **Fargate tasks** directly using:

```
arn:aws:states:::ecs:runTask.sync
```

This allows:

- Real compute
- Long-running jobs
- Large media files
- Horizontal scaling

What problems solved

This phase required solving real production-grade problems:

◇ ECR & IAM

- Workers initially failed to pull images
- Fixed by attaching:

```
AmazonECSTaskExecutionRolePolicy
```

to ECS task roles

◇ CloudWatch Logs

- ECS tasks crashed because log groups didn't exist
- Fixed by provisioning:

```
/ecs/smmu-dev-transcode
/ecs/smmu-dev-transcribe
```

◇ S3 Permissions

- Workers failed with 403 / 404 on S3
- Fixed by adding **bucket-scoped IAM policies**:
  - Read from `smmu-dev-raw-media`
  - Write to `smmu-dev-processed-media`

◇ Step Functions Data Passing

- ECS environment variables needed `Value.$` JSONPath mapping
- Step input now cleanly passes:
  - jobId
  - inputKey
  - outputKey
  - table name

Result

At the end of Phase 4, the system became:

> **A real distributed compute pipeline**

You can now:

1. Send a job to SQS
2. Dispatcher triggers Step Functions
3. Step Functions chooses the worker
4. ECS runs a container
5. Worker downloads media from S3
6. Processes it

7. Uploads result
8. Updates DynamoDB
9. Step Functions completes

This is the same architecture used by:

- Netflix
- AWS MediaConvert
- AI inference pipelines
- SaaS background job platforms

Phase 5 – Production API Layer

This phase turned the system from “AWS pipeline” into a **real backend service**.

🎯 Goal

Expose the media pipeline through a proper FastAPI backend so clients no longer need to touch AWS directly.

What was built

We created a **production-grade API** running on ECS:

```
Client → API → S3 → SQS → Step Functions → ECS → DynamoDB → S3
```

Key components

1. FastAPI backend

The API runs in ECS behind an ALB and exposes:

Endpoint	Purpose
POST /media/upload/init	Generates a presigned S3 URL
POST /media/upload/complete	Creates job & triggers pipeline
GET /jobs/{jobId}	Fetches job status
GET /health	Health check

2. Presigned S3 upload

Uploads are now done securely:

```
Client → API → presigned S3 URL
Client → S3 (direct upload)
```

This prevents API servers from handling large files and reduces cost.

3. Job creation

When upload completes:

- API creates a job record in DynamoDB
- API pushes job to SQS
- Dispatcher Lambda starts Step Functions

The entire AWS pipeline is now **API-driven**.

Phase 6 – Secure Multi-User SaaS

This phase converted the system into a **true SaaS platform** with user isolation.

🎯 Goal

Ensure every job belongs to a user and cannot be accessed by anyone else.

🔐 JWT Authentication

All protected endpoints now require:

```
Authorization: Bearer <JWT>
```

Tokens must contain:

```
sub = userId
exp = expiry
```

The API validates and attaches the user to every request.

👤 User ownership enforced

All jobs now contain:

```
jobId
userId   ← owner
```

```
mediaId
status
inputKey
progress
```

When querying a job:

```
GET /jobs/{jobId}
```

The API checks:

```
job.userId == token.sub
```

If not → returns 404 (even if the job exists).

This prevents:

- Data leaks
- ID guessing
- Cross-user access

🔗 End-to-end user-aware pipeline

User ID now flows through:

```
API → DynamoDB → SQS → Step Functions → ECS Workers → DynamoDB
```

Every job is traceable to a real user.

This is what makes the system **commercially viable**.

What we achieved by Phase 6

You now have:

- A production API
- Secure authentication
- Per-user job isolation
- Fully automated AWS media pipeline
- End-to-end SaaS architecture

It is a **cloud-native media processing platform**.
