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

1 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

---

2 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.

5️⃣ 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-Pipeline Expansion (Upcoming)

Phase 4 will extend the platform into a **full media processing engine**.

What will be added

Feature	**What it enables**
Video pipeline	Video transcoding
Audio pipeline	Speech-to-text
Media type routing	Smart pipeline selection
Progress tracking	Real job lifecycle
Webhooks	Notify external systems

Feature	What it enables
Retry + DLQ logic	Production reliability

The platform will evolve from:

> “An image processor”

to

> “A distributed, fault-tolerant media processing backend”
