■ AWS ECS

ELASTIC CONTAINER SERVICE

Complete Enterprise Guide — Interview | GUI | SOP | Architecture | POC

■ Version 1.0 | 2025 | Production-Ready Reference

■ TABLE OF CONTENTS

#	Section	Content
1	What is AWS ECS?	Core concept, interview explanation
2	ECS Architecture	Components deep dive
3	ECS vs Competitors	EKS, Fargate, Lambda comparison
4	GUI Step-by-Step	Console walkthrough with screenshots guide
5	How ECS Works (GUI Flow)	Request flow, internals
6	Benefits & Features	Key advantages
7	POC — Proof of Concept	Quick deployment code
8	SOP Documentation	Production standard procedures
9	GitHub README	Professional markdown format
10	Costing & Pricing	Real cost breakdown
11	Enterprise Architecture	Production diagram

1. ■ WHAT IS AWS ECS? — Interview Explanation

Simple One-Line Answer (For Interviews)

AWS ECS (Elastic Container Service) is a fully managed container orchestration service that lets you run, stop, and manage Docker containers on a cluster of EC2 instances or with AWS Fargate (serverless).

The Real-World Analogy

Think of ECS like a **smart restaurant manager**. Your Docker containers are the chefs. ECS is the manager who decides which chef works in which kitchen, scales up when it's busy, replaces a chef who gets sick, and ensures the food is always served. You just define the menu (task definition), and ECS handles the rest.

Core Concepts — Interview Ready

Term	Definition	Interview Tip
Cluster	Logical grouping of EC2 instances or Fargate capacity where tasks run	Always mention: can mix EC2 + Fargate in one cluster
Task Definition	Blueprint/template for your container — image, CPU, RAM, env vars, ports	Like a Dockerfile but for orchestration
Task	A running instance of a Task Definition (like a pod in Kubernetes)	Ephemeral — can stop/restart
Service	Ensures N number of tasks always run; handles LB integration & rolling updates	Key for production deployments
Container Agent	Daemon on each EC2 node that communicates with ECS control plane	Not needed with Fargate
Fargate	Serverless compute for containers — no EC2 management	Highlight cost vs EC2 tradeoffs
ECR	Elastic Container Registry — AWS's private Docker registry	Integrates natively with ECS

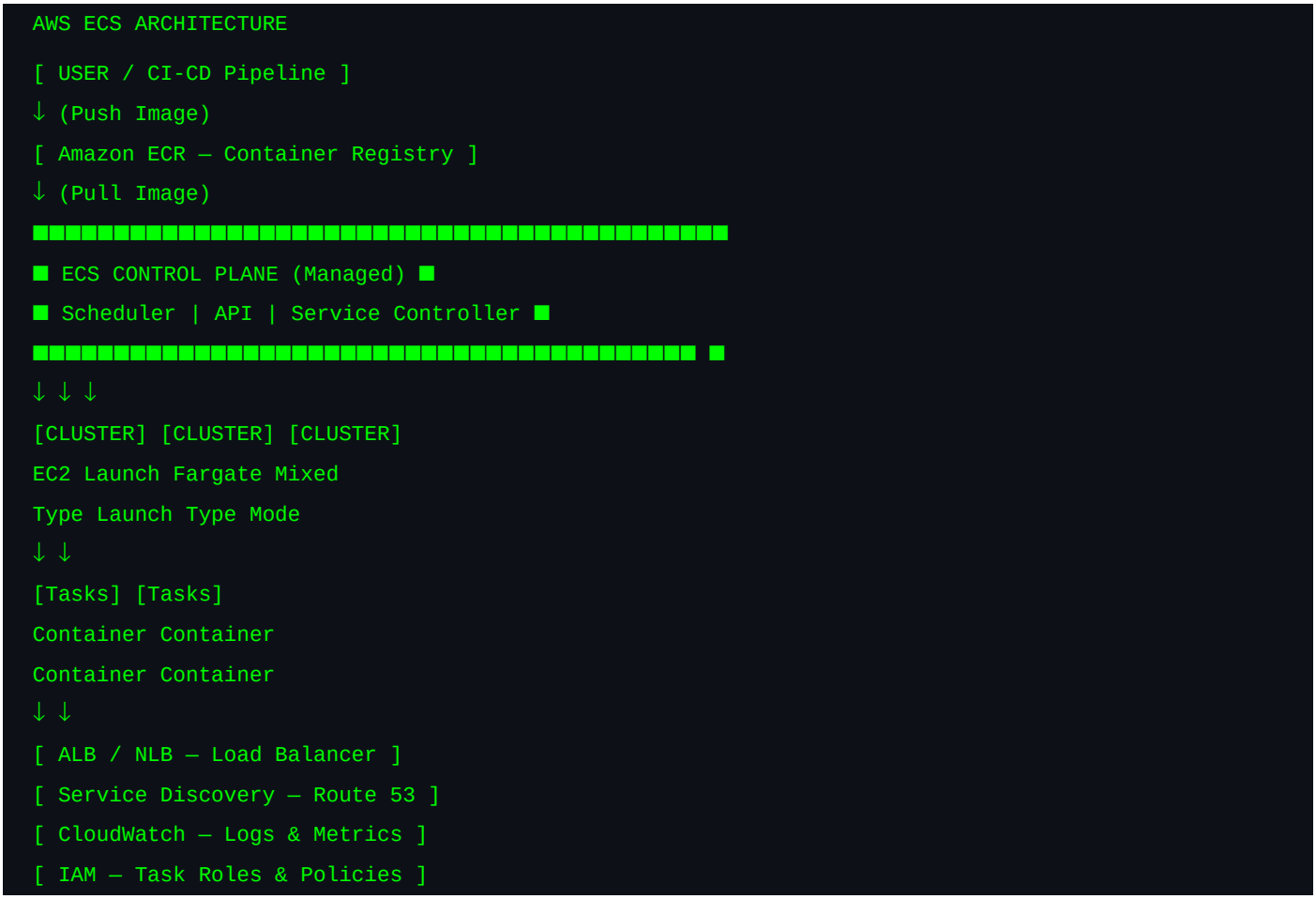
Interview Power Answer — ECS vs EKS

When asked 'Why ECS over EKS?', answer: **ECS is simpler, AWS-native, lower operational overhead, and faster to set up. EKS gives you full Kubernetes power but requires K8s expertise. For teams already in the AWS ecosystem wanting simplicity + reliability, ECS wins.**

2. ■ ECS ARCHITECTURE — Deep Dive

ECS Architecture Components

AWS ECS follows a control-plane / data-plane separation model. The **ECS Control Plane** is a fully managed AWS service — you never touch it. It schedules tasks, monitors health, handles service discovery, and integrates with other AWS services. The **Data Plane** is where your containers actually run — either on EC2 instances you manage, or on Fargate (AWS managed).



Launch Type Comparison

Feature	EC2 Launch Type	Fargate Launch Type
Server Management	You manage EC2 instances	AWS manages — serverless
Cost Model	Pay for EC2 always (idle waste)	Pay per task vCPU/memory used
Startup Time	Fast (instances pre-running)	30-90 sec cold start
Control	Full OS access, GPU support	No OS access
Scaling	Manual or auto-scale EC2 fleet	Auto-scales instantly
Best For	High throughput, GPU workloads	Variable, unpredictable loads
Spot Support	Yes — EC2 Spot Instances	Yes — Fargate Spot

3. ■ ECS vs COMPETITORS

Feature	AWS ECS	AWS EKS	AWS Lambda	Docker Swarm
Learning Curve	■ Low	■ High (K8s)	■ Very Low	■ Medium
Orchestration	AWS Native	Kubernetes	Event-Driven	Docker Native
Scaling Speed	Fast	Fast	Instant	Medium
Cost	$$	$$$	$	$$
Vendor Lock-in	High	Medium	Very High	Low
Community	AWS Docs	Huge K8s	AWS Docs	Declining
Service Mesh	App Mesh	Istio/Envoy	N/A	Limited
Multi-Cloud	No	Partial	No	Yes
Best For	AWS teams, microservices	Complex K8s, hybrid	Short tasks <15min	Small teams

4. ■ HOW TO CREATE ECS — GUI Step-by-Step

Prerequisites

■ AWS Account with Admin or ECS full-access IAM role

■ Docker image ready (or use nginx:latest for testing)

■ VPC with at least 2 public subnets

■ Security Group allowing port 80 / 443

■ IAM Role: ecsTaskExecutionRole (for Fargate)

STEP 1: Create ECR Repository (Push Your Image)

1. Go to AWS Console → Search 'ECR' → Click Elastic Container Registry
2. Click 'Create repository'
3. Visibility: Private | Repository name: my-app | Click 'Create repository'
4. Click on the repo → Click 'View push commands'
5. Run the 4 commands shown (authenticate, build, tag, push)

CLI Commands:

```
aws ecr get-login-password --region us-east-1 | docker login ...
```

```
docker build -t my-app .
```

```
docker tag my-app:latest .dkr.ecr.us-east-1.amazonaws.com/my-app:latest
```

```
docker push .dkr.ecr.us-east-1.amazonaws.com/my-app:latest
```

STEP 2: Create ECS Cluster

1. AWS Console → Search 'ECS' → Click Elastic Container Service
2. Left menu → 'Clusters' → Click 'Create Cluster'
3. Cluster name: my-production-cluster
4. Infrastructure: Select 'AWS Fargate (serverless)' ← recommended for beginners

OR select 'Amazon EC2 instances' for more control

5. Monitoring: Enable 'Container Insights' (CloudWatch) → Tick checkbox
6. Tags: Add Environment=production, Team=backend
7. Click 'Create' — Cluster takes ~30 seconds to become ACTIVE

STEP 3: Create Task Definition

1. Left menu → 'Task Definitions' → Click 'Create new task definition'
2. Task definition family name: my-app-task
3. Launch type: AWS Fargate
4. OS/Architecture: Linux/X86_64
5. Task size: CPU = 0.5 vCPU | Memory = 1 GB (start small, scale later)
6. Task role: ecsTaskExecutionRole (create if not exists)
7. Container — Click 'Add container':

• Name: my-app-container

• Image URI: :latest

• Port mappings: Container port 80, Protocol TCP

• Environment variables: KEY=VALUE (add secrets via Secrets Manager)

• Log configuration: Auto (CloudWatch Logs)

8. Click 'Create' — Note the Task Definition ARN

STEP 4: Create Load Balancer (ALB)

1. AWS Console → EC2 → Load Balancers → 'Create load balancer'
2. Choose: Application Load Balancer → Click 'Create'
3. Name: my-app-alb | Scheme: Internet-facing | IP type: IPv4
4. Network mapping: Select your VPC → Select 2+ public subnets
5. Security groups: Select/create SG allowing HTTP (80) from 0.0.0.0/0
6. Listener: Protocol HTTP, Port 80
7. Target group: Create new → Type: IP | Name: my-app-tg | Port 80
8. Health check path: /health or / | Healthy threshold: 2
9. Click 'Create load balancer' → Note the DNS name

STEP 5: Create ECS Service

1. Go back to ECS → Clusters → my-production-cluster
2. 'Services' tab → Click 'Create'
3. Compute options: Launch type → Fargate | Platform version: LATEST
4. Task definition: my-app-task | Revision: LATEST
5. Service name: my-app-service
6. Desired tasks: 2 (for HA — high availability)
7. Deployment type: Rolling update
8. Load balancing:

• Load balancer type: Application Load Balancer
• Load balancer: my-app-alb
• Container to load balance: my-app-container:80
• Target group: my-app-tg

9. Networking: Select VPC → Select private subnets → Assign public IP: ON
10. Auto Scaling: Min 1 | Desired 2 | Max 10

Scale-out policy: CPU > 70% for 2 minutes

Scale-in policy: CPU < 30% for 5 minutes

11. Click 'Create service'

■ Service will start deploying. Check 'Tasks' tab — wait for RUNNING status

5. ■ HOW ECS WORKS — Internal Flow

Request Flow — From User to Container

Step	Component	What Happens
1	User/Browser	HTTP request hits Route 53 DNS → resolves to ALB
2	ALB (Load Balancer)	Routes request to healthy target → ECS Task IP
3	ECS Task (Container)	Container receives request, processes, returns response
4	ECS Service Controller	Monitors task health, replaces unhealthy tasks
5	ECS Scheduler	On new deployment: starts new tasks, drains old ones
6	Auto Scaling	CloudWatch alarm triggers → ECS adds/removes tasks
7	ECR	On task start: pulls container image (if not cached)
8	CloudWatch	Logs streamed in real-time from containers
9	IAM	Task execution role authenticates to ECR, Secrets Manager
10	Secrets Manager	Injects env vars securely at container startup

Deployment Rolling Update Flow

When you push a new image and update the Task Definition, ECS's Service Controller performs a **rolling update**: It starts new tasks with the updated image. Once they pass health checks on the ALB, old tasks are drained and stopped. Minimum healthy percent (default 100%) ensures zero downtime. Maximum percent (default 200%) allows double tasks during transition.

6. ■ BENEFITS & FEATURES

■ Feature	■ Description	■ Business Value
Fully Managed	No control plane to manage — AWS handles HA, upgrades	Reduces DevOps overhead 60%
Fargate Serverless	No EC2 instances to provision or patch	Zero server management
Auto Scaling	Scale tasks based on CPU, memory, custom CloudWatch metrics	Handle traffic spikes automatically
Native AWS Integration	IAM, ALB, CloudWatch, Secrets Manager, ECR out of the box	Faster development cycles
Service Discovery	Built-in with AWS Cloud Map and Route 53	Microservices find each other
Rolling Deployments	Zero-downtime updates with health check integration	Production-safe deployments
Spot Support	Run tasks on Fargate Spot (up to 70% discount)	Massive cost savings
Multi-Region	Deploy clusters in any AWS region globally	Low latency worldwide
Blue/Green Deploy	Integrated with AWS CodeDeploy for B/G deployments	Instant rollback capability
Container Insights	Deep CloudWatch metrics for containers	Full observability
Task IAM Roles	Per-container AWS permissions — no shared credentials	Enhanced security
EFS Integration	Mount persistent EFS volumes into containers	Stateful workloads
GPU Support	EC2 GPU instances for ML/AI workloads	Run AI containers

7. ■ POC — PROOF OF CONCEPT

Quick POC: Deploy NGINX on ECS Fargate using Terraform

This POC deploys a production-ready NGINX container on ECS Fargate with ALB, CloudWatch logging, and Auto Scaling. Estimated deploy time: 10 minutes.

main.tf — Terraform ECS Fargate POC

```
# main.tf — AWS ECS Fargate POC
terraform {
  required_providers {
    aws = { source = "hashicorp/aws", version = "~> 5.0" }
  }
}

provider "aws" { region = "us-east-1" }

# Data sources
data "aws_vpc" "default" { default = true }
data "aws_subnets" "default" {
  filter { name = "vpc-id", values = [data.aws_vpc.default.id] }
}

# IAM Role for ECS Task Execution
resource "aws_iam_role" "ecs_task_execution_role" {
  name = "ecsTaskExecutionRole"
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action    = "sts:AssumeRole"
      Effect    = "Allow"
      Principal = { Service = "ecs-tasks.amazonaws.com" }
    }]
  })
}

resource "aws_iam_role_policy_attachment" "ecs_task_execution_policy" {
  role       = aws_iam_role.ecs_task_execution_role.name
  policy_arn = "arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy"
}

# ECS Cluster
resource "aws_ecs_cluster" "main" {
  name = "poc-cluster"
  setting { name = "containerInsights", value = "enabled" }
}

# CloudWatch Log Group
resource "aws_cloudwatch_log_group" "ecs" {
  name              = "/ecs/poc-app"
  retention_in_days = 7
}

# Task Definition
```

```
resource "aws_ecs_task_definition" "app" {
  family                   = "poc-app-task"
  network_mode             = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu                      = "256"
  memory                   = "512"
  execution_role_arn       = aws_iam_role.ecs_task_execution_role.arn

  container_definitions = jsonencode([{
    name      = "nginx"
    image     = "nginx:latest"
    cpu       = 256
    memory    = 512
    essential = true
    portMappings = [{ containerPort = 80, protocol = "tcp" }]
    logConfiguration = {
      logDriver = "awslogs"
      options = {
        "awslogs-group"         = "/ecs/poc-app"
        "awslogs-region"        = "us-east-1"
        "awslogs-stream-prefix" = "ecs"
      }
    }
  }])
}

# Security Group
resource "aws_security_group" "ecs_sg" {
  name   = "ecs-poc-sg"
  vpc_id = data.aws_vpc.default.id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# ECS Service
resource "aws_ecs_service" "app" {
  name            = "poc-app-service"
  cluster         = aws_ecs_cluster.main.id
  task_definition = aws_ecs_task_definition.app.arn
  desired_count   = 2
  launch_type     = "FARGATE"

  network_configuration {
    subnets          = data.aws_subnets.default.ids
```

```
    security_groups  = [aws_security_group.ecs_sg.id]
    assign_public_ip = true
  }
}

output "cluster_name" { value = aws_ecs_cluster.main.name }
```

Deploy Commands

```
terraform init # Initialize Terraform providers
```

```
terraform plan # Preview changes
```

```
terraform apply -auto-approve # Deploy infrastructure
```

```
terraform destroy # Tear down when done (avoid costs)
```

8. ■ SOP — STANDARD OPERATING PROCEDURES

SOP-001: New Service Deployment

Trigger:
New microservice ready for production deployment

Owner:
DevOps / Platform Engineering Team

Frequency:
Per release cycle

Pre-checks:

1. Docker image built and pushed to ECR successfully
2. Security scan passed (ECR image scanning enabled)
3. Task Definition reviewed and approved
4. ALB target group health check path validated
5. Required IAM task roles exist with minimum permissions

Steps:

1. Update Task Definition with new image URI → create new revision
2. Update ECS Service → select new Task Definition revision
3. Monitor deployment in ECS Console → Services → Deployments tab
4. Verify new tasks reach RUNNING state (max wait: 5 mins)
5. ALB health checks pass for all new tasks
6. Run smoke tests against ALB DNS endpoint
7. Monitor CloudWatch logs for errors (15 mins post-deploy)
8. Update deployment tracker in Jira/Confluence

Rollback:

1. ECS Console → Service → Update Service
2. Select previous Task Definition revision
3. Force new deployment → Rolling update restores previous version
4. Rollback time: ~3-5 minutes

SOP-002: Incident Response — Container Failure

Trigger:
ECS tasks entering STOPPED state unexpectedly or CloudWatch alarm

Owner:
On-Call SRE

Severity Levels:

P1: All tasks stopped — service completely down → 15 min SLA

P2: >50% tasks unhealthy → 30 min SLA

P3: Single task failure, service degraded → 2 hour SLA

Steps:

1. Check ECS Console → Cluster → Tasks → Stopped Tasks
2. Click on stopped task → 'Stopped reason' field for root cause

3. CloudWatch → Log groups → /ecs/ for container logs
4. Common causes checklist:
• OOMKilled: Task ran out of memory → increase Task memory
• Essential container exited: App crash → check application logs
• ResourceInitializationError: IAM role missing permissions
• CannotPullContainerError: ECR auth issue or image not found
5. Fix root cause → re-deploy or update Task Definition
6. Post-incident: update runbook with new failure mode

SOP-003: Auto Scaling Configuration

Trigger:

Service experiencing high CPU/memory or scheduled traffic spikes

Steps:

1. ECS Console → Service → Update → Service Auto Scaling
2. Configure Application Auto Scaling:
• Minimum tasks: 2 (never below for HA)
• Desired tasks: based on baseline load
• Maximum tasks: based on budget/capacity
3. Target Tracking Policy — Recommended:
• Metric: ECSServiceAverageCPUUtilization
• Target value: 70%
• Scale-out cooldown: 60 seconds
• Scale-in cooldown: 300 seconds
4. Test scaling by generating load (use Apache Bench or k6)
5. Verify CloudWatch shows scaling events

9. ■ GITHUB README — Professional Format

```
# ■ AWS ECS Microservices Platform

[![AWS ECS](https://img.shields.io/badge/AWS-ECS-orange)](https://aws.amazon.com/ecs/)
[![Terraform](https://img.shields.io/badge/IaC-Terraform-blueviolet)](https://terraform.io)
[![License: MIT](https://img.shields.io/badge/License-MIT-yellow.svg)](LICENSE)

## Overview
Production-ready AWS ECS Fargate infrastructure for containerized microservices
with auto-scaling, blue/green deployments, and full observability.

## Architecture
- **Compute**: AWS Fargate (serverless containers)
- **Registry**: Amazon ECR (private container registry)
- **Load Balancing**: Application Load Balancer (ALB)
- **Scaling**: ECS Service Auto Scaling + Application Auto Scaling
- **Monitoring**: CloudWatch Container Insights
- **Security**: IAM Task Roles, VPC private subnets, Secrets Manager
- **IaC**: Terraform modules

## Quick Start

### Prerequisites
- AWS CLI configured: `aws configure`
- Terraform >= 1.5.0
- Docker installed

### Deploy
```bash
git clone https://github.com/your-org/ecs-platform
cd ecs-platform
cp terraform.tfvars.example terraform.tfvars
terraform init && terraform apply
```

Push Your Image
```bash
aws ecr get-login-password --region us-east-1 | \
  docker login --username AWS --password-stdin
docker build -t my-app .
docker tag my-app:latest /my-app:latest
docker push /my-app:latest
```

Project Structure
```
.
■■■ terraform/
■   ■■■ main.tf          # Core ECS resources
■   ■■■ variables.tf     # Input variables
■   ■■■ outputs.tf       # Output values
■   ■■■ modules/
■       ■■■ ecs/         # ECS cluster + service
■       ■■■ alb/         # Load balancer
```
```

```
■       ■■■ ecr/         # Container registry
■■■ docker/
■   ■■■ Dockerfile
■■■ .github/workflows/
■   ■■■ deploy.yml       # CI/CD pipeline
■■■ docs/
    ■■■ architecture.md
```

Environments

Environment	Cluster	Tasks	Auto-Scale
dev	ecs-dev	1-2	Off
staging	ecs-stg	2-4	CPU > 80%
production	ecs-prd	4-20	CPU > 70%

Contributing

See CONTRIBUTING.md | Follow GitFlow branching strategy

10. ■ COSTING & PRICING — Real Breakdown

Fargate Pricing (us-east-1, 2025)

Resource	Unit	Price/Unit	Notes
vCPU	per vCPU-hour	$0.04048	1 task × 0.5 vCPU × 24h = $0.49/day
Memory	per GB-hour	$0.004445	1 task × 1 GB × 24h = $0.11/day
Fargate Spot vCPU	per vCPU-hour	$0.01173	~70% savings vs on-demand
Fargate Spot Memory	per GB-hour	$0.00129	~70% savings vs on-demand
ECR Storage	per GB/month	$0.10	First 500 MB free
ALB	per hour	$0.008	+ $0.008 per LCU-hour
CloudWatch Logs	per GB ingested	$0.50	First 5 GB free

Monthly Cost Scenarios

Scenario	Config	Monthly Cost (Est.)	Notes
Small App (Dev)	2 tasks × 0.25 vCPU × 0.5 GB, 730h	**~$15-20/mo**	Dev/Test workload
Medium App (Prod)	4 tasks × 0.5 vCPU × 1 GB + ALB	**~$60-80/mo**	Small production
Large App (Enterprise)	20 tasks × 1 vCPU × 2 GB + ALB + CloudWatch	**~$350-450/mo**	High traffic
With Fargate Spot	Same as Medium using Spot	**~$20-25/mo**	70% savings
EC2 Launch Type (t3.medium)	2 × t3.medium + ECS overhead	**~$60-70/mo**	More control

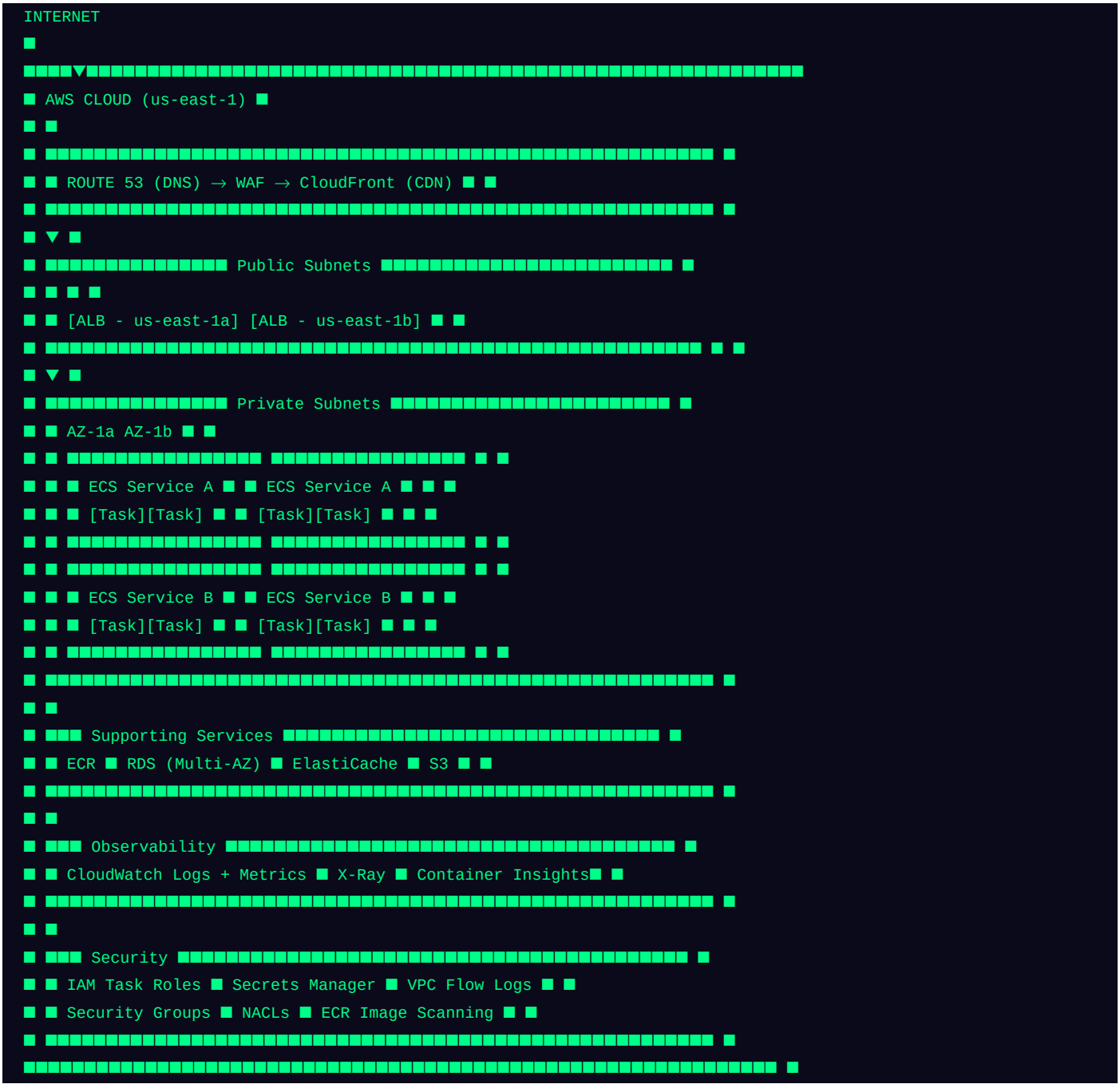
■ Cost Optimization Tips

- ■ Use Fargate Spot for batch/async workloads → save up to 70%
- ■ Right-size tasks: start small (0.25 vCPU, 0.5 GB) and scale up based on metrics
- ■ Enable CloudWatch Container Insights only in production (extra cost in dev)
- ■ Use ECR lifecycle policies to auto-delete old images and save storage costs
- ■ ALB: Consolidate multiple services behind one ALB using path-based routing
- ■ Reserved capacity: Fargate Compute Savings Plans save up to 52%
- ■ Use AWS Cost Explorer + ECS tags to track per-service spend

11. ■ ENTERPRISE ARCHITECTURE — Production Design

Multi-AZ Production Architecture

A production ECS deployment spans multiple Availability Zones (AZ) for high availability, uses private subnets for containers, public subnets for load balancers, and integrates with the full AWS security and observability stack.



CI/CD Pipeline Integration

Stage	Tool	Action
Source	GitHub / CodeCommit	Developer pushes code → triggers pipeline
Build	CodeBuild / GitHub Actions	docker build, test, docker push to ECR

Security Scan	ECR Image Scanning + Snyk	Scan image for CVEs before deploy
Deploy (Dev)	CodePipeline → ECS	Auto-deploy to dev cluster on every push
Deploy (Staging)	CodePipeline → ECS	Deploy on merge to main branch
Deploy (Prod)	CodeDeploy B/G → ECS	Manual approval → Blue/Green deployment
Monitor	CloudWatch + PagerDuty	Alerts on errors, latency, CPU anomalies
Rollback	ECS Service Update	One-click rollback to previous revision