AWS ECS (Elastic Container Service): Complete Practical Guide

Table of Contents

What is AWS ECS?

Amazon Elastic Container Service (ECS) is AWS's fully managed container orchestration service that allows you to run, stop, and manage Docker containers on a cluster. ECS eliminates the need to install, operate, and scale your own cluster management infrastructure.

Key Capabilities

Capability	Description
Container Orchestration	Automatically place containers across your cluster based on resource needs
Fully Managed	AWS handles the control plane, no Kubernetes master nodes to manage
Highly Scalable	Scale from a single container to thousands of containers
AWS Integration	Native integration with VPC, IAM, CloudWatch, ALB, ECR, and more
Flexible Deployment	Run on EC2 instances, Fargate (serverless), or on-premises (ECS Anywhere)

Capability	Description
Cost Effective	Pay only for the resources you use (no additional ECS charges for EC2/Fargate)

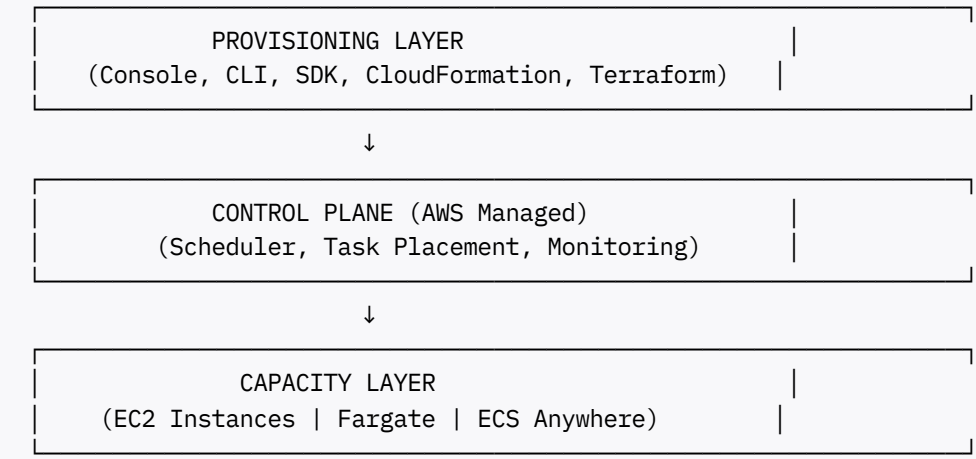
Why ECS Matters for Operations Engineers

- **Simpler than Kubernetes**: Easier learning curve, less complexity
- **AWS-Native**: Deep integration with AWS services out of the box
- **Self-Healing**: Automatically replaces failed containers
- **Blue-Green Deployments**: Built-in deployment strategies
- **Service Discovery**: Containers can find each other easily
- **No Control Plane Management**: AWS manages the orchestration layer

Core Concepts & Architecture

ECS Architecture Layers

ECS consists of three main layers:



Component Hierarchy

```
AWS Account
  └── ECS Cluster (logical grouping)
       ├── Task Definition (blueprint)
       │    └── Container Definition(s)
       │         ├── Image
       │         ├── CPU/Memory
       │         ├── Environment Variables
       │         └── Port Mappings
       │
       ├── Tasks (running instances)
       │    └── Container(s)
       │
```

```
        ├── Services (long-running applications)
        │    ├── Desired Count
        │    ├── Deployment Configuration
        │    └── Load Balancer Integration
        │
        └── Capacity Providers
             ├── EC2 (self-managed)
             ├── Fargate (serverless)
             └── ECS Managed Instances
```

Key Terminology

Term	Definition	Analogy
Cluster	Logical grouping of tasks and services	A data center or server farm
Task Definition	Blueprint/template for your application	A recipe or Dockerfile
Task	Running instance of a task definition	A running container group
Service	Maintains desired number of tasks	A deployment that self-heals
Container	Docker container running your application	Individual application process
Capacity Provider	Infrastructure to run containers on	The physical/virtual servers

ECS Clusters

What is a Cluster?

A **cluster** is a logical grouping of tasks or services. It provides the infrastructure capacity (EC2 instances or Fargate) where your containers run.

Cluster Types

1. ECS Managed Instances (Recommended)

Best for: Most workloads where AWS manages infrastructure.

Key Features:

- AWS handles provisioning, patching, scaling
- Optimal cost-performance balance
- Automatic instance optimization
- No infrastructure management overhead

When to Use:

- You want AWS to handle infrastructure
- Need cost-effective compute

- Want to focus on applications, not servers
- Predictable workloads

2. AWS Fargate (Serverless)

Best for: Variable workloads, getting started quickly.

Key Features:

- No servers to manage
- Pay only for resources tasks use
- Automatic scaling
- Minimal operational overhead

When to Use:

- Serverless operations preferred
- Unpredictable/variable workloads
- Want rapid deployment
- Don't want infrastructure management

3. Amazon EC2 Instances (Full Control)

Best for: Maximum control and customization.

Key Features:

- Full control over instances
- Custom AMIs supported
- Choose specific instance types
- Manage Auto Scaling groups yourself

When to Use:

- Need specific instance types
- Require custom AMIs
- Have existing EC2 infrastructure
- Need maximum control

Cluster Concepts

Important Points:

- Clusters are **region-specific** (cannot span regions)
- You can have **multiple clusters** per account
- Clusters can contain a **mix of EC2 and Fargate tasks**

- Clusters isolate resources for different environments (dev, prod, staging)
- No cost for the cluster itself (only pay for underlying resources)

Cluster States

State	Description
ACTIVE	Ready to accept tasks
PROVISIONING	Resources being created
DEPROVISIONING	Resources being deleted
FAILED	Resource creation failed
INACTIVE	Cluster has been deleted

Creating a Cluster

Method 1: AWS Console

Steps:

1. Navigate to **ECS Console → Clusters**
2. Click **Create Cluster**
3. Enter cluster name (e.g., `production-cluster`)
4. Choose infrastructure:
    - **AWS Fargate** (serverless)
    - **Amazon EC2 instances** (self-managed)
    - **ECS Managed Instances** (AWS-managed)
5. Configure networking (VPC, subnets)
6. (Optional) Enable CloudWatch Container Insights
7. Click **Create**

Method 2: AWS CLI

```
# Create a simple cluster
aws ecs create-cluster \
    --cluster-name production-cluster \
    --region us-east-1

# Create cluster with tags
aws ecs create-cluster \
    --cluster-name production-cluster \
    --tags key=Environment,value=Production key=Team,value=DevOps \
    --region us-east-1

# Create cluster with CloudWatch Container Insights
```

```
aws ecs create-cluster \
    --cluster-name production-cluster \
    --settings name=containerInsights,value=enabled \
    --region us-east-1
```

Method 3: Terraform

```
resource "aws_ecs_cluster" "production" {
  name = "production-cluster"

  setting {
    name  = "containerInsights"
    value = "enabled"
  }

  tags = {
    Environment = "Production"
    ManagedBy   = "Terraform"
  }
}

# Create cluster with capacity providers
resource "aws_ecs_cluster" "production_with_capacity" {
  name = "production-cluster"

  capacity_providers = ["FARGATE", "FARGATE_SPOT"]

  default_capacity_provider_strategy {
    capacity_provider = "FARGATE"
    weight            = 1
    base              = 1
  }
}
```

Viewing Cluster Information

```
# List all clusters
aws ecs list-clusters

# Describe a specific cluster
aws ecs describe-clusters \
    --cluster production-cluster \
    --include STATISTICS,SETTINGS,CONFIGURATIONS

# Get cluster metrics
aws cloudwatch get-metric-statistics \
    --namespace AWS/ECS \
    --metric-name CPUUtilization \
    --dimensions Name=ClusterName,Value=production-cluster \
    --start-time 2025-10-28T00:00:00Z \
    --end-time 2025-10-28T23:59:59Z \
```

```
    --period 3600 \
    --statistics Average
```

Task Definitions

What is a Task Definition?

A **task definition** is a JSON blueprint that describes one or more containers (up to 10) that form your application. It's similar to a Docker Compose file.

Task Definition vs Task

Aspect	Task Definition	Task
Type	Blueprint/Template	Running Instance
Lifecycle	Registered once, versioned	Created, runs, stops
Mutability	Immutable (new version created on update)	Ephemeral
Analogy	Class in OOP	Object/Instance
Example	Recipe for cookies	Actual baked cookies

Key Task Definition Parameters

1. Family and Revision

```
{
  "family": "web-app",
  "revision": 3
}
```

- **Family**: Logical name for grouping task definition versions
- **Revision**: Auto-incrementing version number
- Full identifier: `web-app:3`

2. Container Definitions

```
{
  "containerDefinitions": [
    {
      "name": "nginx-container",
      "image": "nginx:latest",
      "cpu": 256,
      "memory": 512,
      "essential": true,
      "portMappings": [
        {
```

```
          "containerPort": 80,
          "protocol": "tcp"
        }
      ],
      "environment": [
        {"name": "ENV", "value": "production"}
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/web-app",
          "awslogs-region": "us-east-1",
          "awslogs-stream-prefix": "nginx"
        }
      }
    }
  ]
}
```

Key Container Parameters:

Parameter	Description	Example
`name`	Container name (unique within task)	`"web-server"`
`image`	Docker image URI	`"nginx:1.21"` or ECR URI
`cpu`	CPU units (1024 = 1 vCPU)	`256`
`memory`	Hard memory limit (MB)	`512`
`memoryReservation`	Soft memory limit (MB)	`256`
`essential`	Task fails if this container stops	`true`
`portMappings`	Port bindings	`containerPort: 80`
`environment`	Environment variables	`[{"name": "DB_HOST", "value": "..."}]`
`secrets`	Secrets from Parameter Store/Secrets Manager	`[{"name": "DB_PASS", "valueFrom": "arn:..."}]`
`command`	Override CMD from Dockerfile	`["/bin/sh", "-c", "echo hello"]`
`entryPoint`	Override ENTRYPOINT	`["/app/entrypoint.sh"]`

3. Task-Level Settings

Launch Type Requirements:

```
{
  "requiresCompatibilities": ["FARGATE"],
  "networkMode": "awsvpc",
  "cpu": "512",
```

```
  "memory": "1024"
}
```

For Fargate:

- `networkMode` **must** be `awsvpc`
- Task-level `cpu` and `memory` are required
- Specific CPU/memory combinations allowed (see AWS documentation)

For EC2:

- `networkMode` can be `bridge`, `host`, `awsvpc`, or `none`
- Task-level CPU/memory optional (can be container-level only)

Valid Fargate CPU/Memory Combinations:

CPU	Memory Options
256 (.25 vCPU)	512 MB, 1 GB, 2 GB
512 (.5 vCPU)	1 GB, 2 GB, 3 GB, 4 GB
1024 (1 vCPU)	2 GB, 3 GB, 4 GB, 5 GB, 6 GB, 7 GB, 8 GB
2048 (2 vCPU)	4 GB to 16 GB (1 GB increments)
4096 (4 vCPU)	8 GB to 30 GB (1 GB increments)

4. IAM Roles

```
{
  "taskRoleArn": "arn:aws:iam::123456789012:role/ecsTaskRole",
  "executionRoleArn": "arn:aws:iam::123456789012:role/ecsTaskExecutionRole"
}
```

Task Role (`taskRoleArn`):

- Permissions for your **application code**
- Example: Access S3, DynamoDB, SQS from application

Execution Role (`executionRoleArn`):

- Permissions for **ECS agent**
- Example: Pull images from ECR, write logs to CloudWatch

5. Network Mode

Mode	Description	Use Case
`awsvpc`	Each task gets its own ENI and private IP	Fargate (required), security isolation
`bridge`	Docker's default bridge network	Simple EC2 deployments

Mode	Description	Use Case
`host`	Container uses host's network	High network performance needed
`none`	No network connectivity	Offline batch processing

6. Volumes

EFS Volume Example:

```
{
  "volumes": [
    {
      "name": "efs-storage",
      "efsVolumeConfiguration": {
        "fileSystemId": "fs-1234567",
        "transitEncryption": "ENABLED",
        "authorizationConfig": {
          "accessPointId": "fsap-1234567",
          "iam": "ENABLED"
        }
      }
    }
  ],
  "containerDefinitions": [
    {
      "name": "app",
      "mountPoints": [
        {
          "sourceVolume": "efs-storage",
          "containerPath": "/mnt/efs"
        }
      ]
    }
  ]
}
```

Creating Task Definitions

Method 1: AWS Console

Steps:

1. Navigate to **ECS** → **Task Definitions**
2. Click **Create new Task Definition**
3. Choose launch type compatibility (Fargate, EC2, or both)
4. Configure task definition:
   - **Family name**: `web-app`
   - **Task role**: IAM role for application

- **Network mode**: `awsvpc`
- **Task size**: CPU and memory

5. Add container:
   - **Name**: `nginx`
   - **Image**: `nginx:latest`
   - **Port mappings**: Container port 80
   - **Environment variables**
   - **Log configuration**
6. Click **Create**

Method 2: AWS CLI

Complete Example:

Save this as `task-definition.json`:

```
{
  "family": "web-app",
  "networkMode": "awsvpc",
  "requiresCompatibilities": ["FARGATE"],
  "cpu": "512",
  "memory": "1024",
  "taskRoleArn": "arn:aws:iam::123456789012:role/ecsTaskRole",
  "executionRoleArn": "arn:aws:iam::123456789012:role/ecsTaskExecutionRole",
  "containerDefinitions": [
    {
      "name": "nginx",
      "image": "123456789012.dkr.ecr.us-east-1.amazonaws.com/my-app:latest",
      "cpu": 256,
      "memory": 512,
      "essential": true,
      "portMappings": [
        {
          "containerPort": 80,
          "protocol": "tcp"
        }
      ],
      "environment": [
        {"name": "ENVIRONMENT", "value": "production"},
        {"name": "LOG_LEVEL", "value": "info"}
      ],
      "secrets": [
        {
          "name": "DB_PASSWORD",
          "valueFrom": "arn:aws:secretsmanager:us-east-1:123456789012:secret:db-password'
        }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
```

```
          "awslogs-group": "/ecs/web-app",
          "awslogs-region": "us-east-1",
          "awslogs-stream-prefix": "nginx"
        }
      },
      "healthCheck": {
        "command": ["CMD-SHELL", "curl -f http://localhost/ || exit 1"],
        "interval": 30,
        "timeout": 5,
        "retries": 3,
        "startPeriod": 60
      }
    }
  ]
}
```

Register it:

```
aws ecs register-task-definition \
    --cli-input-json file://task-definition.json
```

Method 3: Terraform

```
resource "aws_ecs_task_definition" "web_app" {
  family                   = "web-app"
  network_mode             = "awsvpc"
  requires_compatibilities = ["FARGATE"]
  cpu                      = "512"
  memory                   = "1024"
  task_role_arn           = aws_iam_role.ecs_task_role.arn
  execution_role_arn      = aws_iam_role.ecs_execution_role.arn

  container_definitions = jsonencode([
    {
      name      = "nginx"
      image     = "nginx:latest"
      cpu       = 256
      memory    = 512
      essential = true
      portMappings = [
        {
          containerPort = 80
          protocol      = "tcp"
        }
      ]
      environment = [
        {
          name  = "ENVIRONMENT"
          value = "production"
        }
      ]
      logConfiguration = {
        logDriver = "awslogs"
```

```
        options = {
          "awslogs-group"         = "/ecs/web-app"
          "awslogs-region"        = "us-east-1"
          "awslogs-stream-prefix" = "nginx"
        }
      }
    }
  ])
}
```

Updating Task Definitions

Task definitions are **immutable**. When you update:

1. A new **revision** is created
2. Old revisions remain (can be used or deregistered)
3. Services must be updated to use new revision

```
# Register updated task definition
aws ecs register-task-definition --cli-input-json file://task-definition-v2.json

# Update service to use new revision
aws ecs update-service \
    --cluster production-cluster \
    --service web-app-service \
    --task-definition web-app:2
```

Tasks vs Services

Understanding the Difference

Aspect	Task	Service
Purpose	Run once and exit	Run continuously
Use Case	Batch jobs, one-time tasks	Web servers, APIs, long-running apps
Self-Healing	No (stops when done)	Yes (replaces failed tasks)
Load Balancer	Not supported	Supported
Desired Count	N/A	Maintained by scheduler
Example	Data migration, ETL job	Web application, microservice

Tasks (One-Time Execution)

When to Use Tasks:

- Batch processing jobs
- Scheduled jobs (cron-like)
- Data migrations
- One-time administrative tasks
- CI/CD build jobs

Running a Task:

```
# Run a single task
aws ecs run-task \
    --cluster production-cluster \
    --task-definition batch-job:1 \
    --count 1 \
    --launch-type FARGATE \
    --network-configuration "awsvpcConfiguration={subnets=[subnet-12345],securityGroups=[

# Run task with environment overrides
aws ecs run-task \
    --cluster production-cluster \
    --task-definition batch-job:1 \
    --overrides '{
      "containerOverrides": [
        {
          "name": "batch-processor",
          "environment": [
            {"name": "BATCH_ID", "value": "2025-10-28"}
          ]
        }
      ]
    }'
```

Services (Long-Running Applications)

When to Use Services:

- Web applications
- APIs and microservices
- Background workers
- Streaming applications
- Any application that should always be running

Service Features:

- **Desired Count**: Maintains specified number of tasks
- **Load Balancing**: Integrates with ALB/NLB

- **Service Discovery**: Register in AWS Cloud Map
- **Auto Scaling**: Scale tasks based on metrics
- **Deployment Strategies**: Rolling, blue/green, canary
- **Health Checks**: Automatic replacement of unhealthy tasks

Creating a Service

Method 1: AWS Console

Steps:

1. Navigate to **Cluster** → **Services** tab
2. Click **Create**
3. Configure service:
   - **Launch type**: Fargate or EC2
   - **Task definition**: Select family and revision
   - **Service name**: `web-app-service`
   - **Desired tasks**: 3
4. Configure deployment:
   - **Deployment type**: Rolling update
   - **Min healthy**: 100%
   - **Max healthy**: 200%
5. (Optional) Configure load balancer
6. (Optional) Configure auto scaling
7. Click **Create**

Method 2: AWS CLI

```
# Create a basic service
aws ecs create-service \
    --cluster production-cluster \
    --service-name web-app-service \
    --task-definition web-app:1 \
    --desired-count 3 \
    --launch-type FARGATE \
    --network-configuration "awsvpcConfiguration={
        subnets=[subnet-12345,subnet-67890],
        securityGroups=[sg-12345],
        assignPublicIp=DISABLED
    }"

# Create service with load balancer
aws ecs create-service \
    --cluster production-cluster \
```

```
    --service-name web-app-service \
    --task-definition web-app:1 \
    --desired-count 3 \
    --launch-type FARGATE \
    --network-configuration "awsvpcConfiguration={
        subnets=[subnet-12345,subnet-67890],
        securityGroups=[sg-12345]
    }" \
    --load-balancers "targetGroupArn=arn:aws:elasticloadbalancing:us-east-1:123456789012:
    --health-check-grace-period-seconds 60
```

Method 3: Terraform

```
resource "aws_ecs_service" "web_app" {
  name            = "web-app-service"
  cluster         = aws_ecs_cluster.production.id
  task_definition = aws_ecs_task_definition.web_app.arn
  desired_count   = 3
  launch_type     = "FARGATE"

  network_configuration {
    subnets          = aws_subnet.private[*].id
    security_groups  = [aws_security_group.ecs_tasks.id]
    assign_public_ip = false
  }

  load_balancer {
    target_group_arn = aws_lb_target_group.app.arn
    container_name   = "nginx"
    container_port   = 80
  }

  deployment_configuration {
    maximum_percent         = 200
    minimum_healthy_percent = 100
  }

  depends_on = [aws_lb_listener.app]
}
```

Service Deployment Strategies

1. Rolling Update (Default)

How it Works:

1. Launch new tasks with updated task definition
2. Wait for new tasks to become healthy
3. Stop old tasks
4. Repeat until all tasks updated

Configuration:

```
{
  "deploymentConfiguration": {
    "maximumPercent": 200,
    "minimumHealthyPercent": 100
  }
}
```

- **maximumPercent**: Max tasks during deployment (200% = double capacity temporarily)
- **minimumHealthyPercent**: Min tasks that must stay healthy (100% = no downtime)

Example Scenarios:

Min%	Max%	Behavior
100	200	No downtime, double capacity temporarily
50	100	Allow 50% downtime, no extra capacity
0	100	Stop all old tasks, then start new (downtime!)

2. Blue/Green Deployment (with CodeDeploy)

How it Works:

1. Deploy new version (green) alongside old (blue)
2. Test green environment
3. Shift traffic from blue to green
4. Optionally rollback if issues detected

Benefits:

- Zero downtime
- Easy rollback
- Test before production traffic

Launch Types (EC2 vs Fargate)

Comparison

Aspect	EC2	Fargate
Infrastructure	You manage EC2 instances	AWS manages (serverless)
Cost Model	Pay for instances (always on)	Pay per task (pay-as-you-go)
Scaling	Manage Auto Scaling groups	Automatic, task-level
Flexibility	Full control, custom AMIs	Limited control

Aspect	EC2	Fargate
Overhead	Instance management overhead	Zero infrastructure management
Best For	Predictable workloads, cost optimization	Variable workloads, simplicity
Networking	All modes supported	`awsvpc` only

EC2 Launch Type

Architecture:

```
ECS Cluster
  ├── EC2 Instance 1 (running ECS agent)
  │    ├── Task 1 (containers)
  │    └── Task 2 (containers)
  ├── EC2 Instance 2
  │    ├── Task 3
  │    └── Task 4
  └── EC2 Instance 3
       └── Task 5
```

When to Use EC2:

- **Cost optimization**: Reserved instances for steady workloads
- **Custom requirements**: Specific instance types, GPUs, custom AMIs
- **High utilization**: Pack multiple tasks per instance
- **Windows containers**: Fargate only supports Linux
- **Existing infrastructure**: Already have EC2 capacity

Setting Up EC2 Launch Type:

1. **Create Launch Template or Auto Scaling Group**:

```
# User data script for EC2 instances
#!/bin/bash
echo "ECS_CLUSTER=production-cluster" &gt;&gt; /etc/ecs/ecs.config
echo "ECS_ENABLE_TASK_IAM_ROLE=true" &gt;&gt; /etc/ecs/ecs.config
```

2. **Launch EC2 Instances**:

```
# Use ECS-optimized AMI
aws ec2 run-instances \
    --image-id ami-0c55b159cbfafe1f0 \
    --instance-type t3.medium \
    --iam-instance-profile Name=ecsInstanceRole \
    --user-data file://ecs-user-data.sh \
    --subnet-id subnet-12345 \
    --security-group-ids sg-12345
```

3. **Instances Auto-Register to Cluster**

Fargate Launch Type

Architecture:

```
ECS Cluster (Fargate)
  ├── Task 1 (isolated compute)
  ├── Task 2 (isolated compute)
  ├── Task 3 (isolated compute)
  └── Task 4 (isolated compute)

(No visible EC2 instances)
```

When to Use Fargate:

- **Serverless**: No infrastructure management
- **Variable workloads**: Unpredictable traffic
- **Rapid deployment**: Get started quickly
- **Isolation**: Each task has dedicated compute
- **Simplicity**: Minimal operational overhead

Fargate Benefits:

- No instance patching or management
- Automatic scaling at task level
- Pay only for task runtime
- Enhanced security isolation

Fargate Pricing Example:

```
Task: 0.5 vCPU, 1 GB RAM
Running 24 hours

Cost = (vCPU price × vCPU × hours) + (memory price × GB × hours)
     = ($0.04048 × 0.5 × 24) + ($0.004445 × 1 × 24)
     = $0.486 + $0.107
     = $0.593 per day
```

Fargate Spot

What is it: Run Fargate tasks on spare AWS capacity at up to 70% discount.

Trade-off: Tasks can be interrupted with 2-minute warning.

When to Use:

- Fault-tolerant workloads

- Batch processing
- Cost optimization
- Non-critical development/test environments

Example:

```
aws ecs create-service \
    --cluster production-cluster \
    --service-name batch-processor \
    --task-definition batch-job:1 \
    --desired-count 5 \
    --capacity-provider-strategy \
        capacityProvider=FARGATE_SPOT,weight=70,base=0 \
        capacityProvider=FARGATE,weight=30,base=2
```

This runs:

- **2 tasks** on regular Fargate (base)
- **70%** of remaining tasks on Fargate Spot
- **30%** of remaining tasks on regular Fargate

Networking in ECS

Network Modes

1. awsvpc (Recommended)

How it Works:

- Each task gets its own **Elastic Network Interface (ENI)**
- Task has its own **private IP address**
- Task uses **security groups** directly
- Required for **Fargate**

Benefits:

- Enhanced security isolation
- Fine-grained network control
- VPC Flow Logs per task
- Simplified networking model

Configuration:

```
{
  "networkMode": "awsvpc",
  "containerDefinitions": [
```

```
    {
      "name": "app",
      "portMappings": [
        {
          "containerPort": 8080,
          "protocol": "tcp"
        }
      ]
    }
  ]
}
```

Service Network Configuration:

```
aws ecs create-service \
    ... \
    --network-configuration "awsvpcConfiguration={
        subnets=[subnet-abc123,subnet-def456],
        securityGroups=[sg-12345678],
        assignPublicIp=ENABLED
    }"
```

2. bridge (Default for EC2)

How it Works:

- Uses Docker's default bridge network
- Containers share host's network namespace
- Dynamic port mapping to host ports

Use Case: Simple EC2 deployments, legacy applications

3. host

How it Works:

- Container uses host's network directly
- No network isolation
- Container ports = host ports

Use Case: High network performance requirements

4. none

How it Works:

- No external network connectivity
- Loopback interface only

Use Case: Offline batch jobs, security isolation

Security Groups for ECS Tasks

With `awsvpc` mode, apply security groups directly to tasks:

```
resource "aws_security_group" "ecs_tasks" {
  name        = "ecs-tasks-sg"
  description = "Security group for ECS tasks"
  vpc_id      = aws_vpc.main.id

  # Allow inbound HTTP from ALB
  ingress {
    from_port       = 80
    to_port         = 80
    protocol        = "tcp"
    security_groups = [aws_security_group.alb.id]
  }

  # Allow outbound to internet (for API calls, etc.)
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

Load Balancing

ECS integrates with **Application Load Balancer (ALB)** and **Network Load Balancer (NLB)** for distributing traffic.

Application Load Balancer (ALB)

Best For: HTTP/HTTPS traffic, path-based routing, host-based routing

How it Works:

1. ALB receives incoming traffic
2. Routes to target group
3. ECS registers/deregisters tasks automatically
4. Traffic distributed across healthy tasks

Setup:

```
# Create target group
aws elbv2 create-target-group \
    --name ecs-web-app-tg \
    --protocol HTTP \
    --port 80 \
    --vpc-id vpc-12345 \
    --target-type ip \
```

```
    --health-check-path /health \
    --health-check-interval-seconds 30

# Create ECS service with load balancer
aws ecs create-service \
    --cluster production-cluster \
    --service-name web-app \
    --task-definition web-app:1 \
    --desired-count 3 \
    --launch-type FARGATE \
    --network-configuration "awsvpcConfiguration={...}" \
    --load-balancers "targetGroupArn=arn:aws:...,containerName=nginx,containerPort=80" \
    --health-check-grace-period-seconds 60
```

Health Check Grace Period:

- Time before ECS starts checking task health via ALB
- Allows tasks to warm up
- Default: 0 seconds
- Recommended: 60-120 seconds for most apps

Dynamic Port Mapping

With `awsvpc` mode and ALB, ECS automatically maps container ports to target group:

Task Definition:

```
{
  "portMappings": [
    {
      "containerPort": 80,
      "protocol": "tcp"
    }
  ]
}
```

No host port needed! ECS registers task's private IP directly with ALB.

Service Discovery

AWS Cloud Map integration allows containers to discover each other by name.

How it Works

```
Service A (web-frontend)
    ↓ DNS lookup: api.production.internal
Service B (api-backend) - 10.0.1.50
Service C (api-backend) - 10.0.1.51
```

Setting Up Service Discovery

1. **Create Namespace**:

```
aws servicediscovery create-private-dns-namespace \
    --name production.internal \
    --vpc vpc-12345
```

2. **Create Service with Service Discovery**:

```
aws ecs create-service \
    --cluster production-cluster \
    --service-name api-backend \
    --task-definition api:1 \
    --desired-count 2 \
    --launch-type FARGATE \
    --network-configuration "..." \
    --service-registries "registryArn=arn:aws:servicediscovery:..."
```

3. **Access from Another Service**:

```
# In your application code
import requests

response = requests.get("http://api.production.internal/users")
```

Auto Scaling

Service Auto Scaling

Automatically adjust desired task count based on metrics.

Scaling Policies:

1. **Target Tracking**: Maintain metric at target value
2. **Step Scaling**: Add/remove tasks in steps
3. **Scheduled Scaling**: Scale based on schedule

Example: Target Tracking (CPU):

```
# Register scalable target
aws application-autoscaling register-scalable-target \
    --service-namespace ecs \
    --scalable-dimension ecs:service:DesiredCount \
    --resource-id service/production-cluster/web-app \
    --min-capacity 2 \
    --max-capacity 10

# Create scaling policy
```

```
aws application-autoscaling put-scaling-policy \
    --service-namespace ecs \
    --scalable-dimension ecs:service:DesiredCount \
    --resource-id service/production-cluster/web-app \
    --policy-name cpu-target-tracking \
    --policy-type TargetTrackingScaling \
    --target-tracking-scaling-policy-configuration '{
      "TargetValue": 70.0,
      "PredefinedMetricSpecification": {
        "PredefinedMetricType": "ECSServiceAverageCPUUtilization"
      },
      "ScaleInCooldown": 300,
      "ScaleOutCooldown": 60
    }'
```

Cluster Auto Scaling (EC2 Only)

For EC2 launch type, scale the underlying instances using **Capacity Providers**.

Logging and Monitoring

CloudWatch Logs Integration

Configuration in Task Definition:

```
{
  "logConfiguration": {
    "logDriver": "awslogs",
    "options": {
      "awslogs-group": "/ecs/web-app",
      "awslogs-region": "us-east-1",
      "awslogs-stream-prefix": "nginx"
    }
  }
}
```

Create Log Group First:

```
aws logs create-log-group --log-group-name /ecs/web-app
aws logs put-retention-policy \
    --log-group-name /ecs/web-app \
    --retention-in-days 7
```

View Logs:

```
# List log streams
aws logs describe-log-streams \
    --log-group-name /ecs/web-app \
    --order-by LastEventTime \
    --descending
```

```
# Get log events
aws logs get-log-events \
    --log-group-name /ecs/web-app \
    --log-stream-name nginx/nginx/task-id
```

Container Insights

Enable comprehensive monitoring:

```
aws ecs create-cluster \
    --cluster-name production-cluster \
    --settings name=containerInsights,value=enabled
```

Metrics Available:

- CPU utilization
- Memory utilization
- Network performance
- Disk I/O
- Task-level and service-level metrics

Practical Hands-On Examples

Example 1: Deploy Simple Web Application on Fargate

Complete End-to-End Setup

Step 1: Create Cluster

```
aws ecs create-cluster \
    --cluster-name web-app-cluster \
    --settings name=containerInsights,value=enabled
```

Step 2: Create CloudWatch Log Group

```
aws logs create-log-group --log-group-name /ecs/web-app
aws logs put-retention-policy \
    --log-group-name /ecs/web-app \
    --retention-in-days 7
```

Step 3: Create Task Execution IAM Role

```
# Create trust policy
cat &gt; trust-policy.json &lt;&lt;EOF
{
```

```
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ecs-tasks.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
EOF

# Create role
aws iam create-role \
    --role-name ecsTaskExecutionRole \
    --assume-role-policy-document file://trust-policy.json

# Attach policy
aws iam attach-role-policy \
    --role-name ecsTaskExecutionRole \
    --policy-arn arn:aws:iam::aws:policy/service-role/AmazonECSTaskExecutionRolePolicy
```

Step 4: Register Task Definition

```
cat &gt; task-definition.json &lt;&lt;EOF
{
  "family": "web-app",
  "networkMode": "awsvpc",
  "requiresCompatibilities": ["FARGATE"],
  "cpu": "256",
  "memory": "512",
  "executionRoleArn": "arn:aws:iam::123456789012:role/ecsTaskExecutionRole",
  "containerDefinitions": [
    {
      "name": "nginx",
      "image": "nginx:alpine",
      "cpu": 256,
      "memory": 512,
      "essential": true,
      "portMappings": [
        {
          "containerPort": 80,
          "protocol": "tcp"
        }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/web-app",
          "awslogs-region": "us-east-1",
          "awslogs-stream-prefix": "nginx"
        }
      }
    }
```

```
  ]
}
EOF

aws ecs register-task-definition \
    --cli-input-json file://task-definition.json
```

Step 5: Create Service

```
aws ecs create-service \
    --cluster web-app-cluster \
    --service-name web-app-service \
    --task-definition web-app:1 \
    --desired-count 2 \
    --launch-type FARGATE \
    --network-configuration "awsvpcConfiguration={
        subnets=[subnet-abc123,subnet-def456],
        securityGroups=[sg-12345678],
        assignPublicIp=ENABLED
    }"
```

Step 6: Verify Deployment

```
# Check service status
aws ecs describe-services \
    --cluster web-app-cluster \
    --services web-app-service

# List running tasks
aws ecs list-tasks \
    --cluster web-app-cluster \
    --service-name web-app-service

# Get task details
aws ecs describe-tasks \
    --cluster web-app-cluster \
    --tasks task-id
```

Example 2: Multi-Container Application (Sidecar Pattern)

Scenario: Web app + logging sidecar

```
{
  "family": "app-with-sidecar",
  "networkMode": "awsvpc",
  "requiresCompatibilities": ["FARGATE"],
  "cpu": "512",
  "memory": "1024",
  "executionRoleArn": "arn:aws:iam::123456789012:role/ecsTaskExecutionRole",
  "containerDefinitions": [
    {
      "name": "web-app",
```

```
      "image": "my-app:latest",
      "cpu": 256,
      "memory": 512,
      "essential": true,
      "portMappings": [
        {
          "containerPort": 8080
        }
      ],
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/app",
          "awslogs-region": "us-east-1",
          "awslogs-stream-prefix": "app"
        }
      }
    },
    {
      "name": "log-router",
      "image": "fluent/fluentd:latest",
      "cpu": 256,
      "memory": 512,
      "essential": false,
      "logConfiguration": {
        "logDriver": "awslogs",
        "options": {
          "awslogs-group": "/ecs/app",
          "awslogs-region": "us-east-1",
          "awslogs-stream-prefix": "fluentd"
        }
      }
    }
  ]
}
```

Best Practices

1. Task Definition Best Practices

✓ **Use specific image tags** - Avoid `latest`, use version tags (`nginx:1.21.0`)
✓ **Set resource limits** - Always define CPU and memory limits
✓ **Use health checks** - Define container health checks
✓ **Enable logging** - Always configure CloudWatch Logs
✓ **Use secrets** - Store passwords in Secrets Manager, not environment variables
✓ **Minimize container size** - Use alpine images when possible
✓ **One process per container** - Follow container best practices

2. Service Best Practices

✅ **Use multiple AZs** - Deploy across availability zones
✅ **Set appropriate desired count** - Minimum 2 for high availability
✅ **Configure deployment parameters** - Set min/max healthy percentages
✅ **Enable service auto scaling** - Adjust capacity based on demand
✅ **Use health check grace period** - Allow tasks to warm up
✅ **Implement circuit breakers** - Prevent cascading failures

3. Security Best Practices

✅ **Use IAM roles** - Task role for app, execution role for ECS agent
✅ **Network isolation** - Use private subnets for tasks
✅ **Security groups** - Restrict inbound/outbound traffic
✅ **Encrypt data** - Use TLS for communication, encrypt EFS volumes
✅ **Regular updates** - Keep images and dependencies updated
✅ **Scan images** - Use ECR image scanning

4. Cost Optimization

✅ **Use Fargate Spot** - For fault-tolerant workloads
✅ **Right-size tasks** - Don't over-provision CPU/memory
✅ **Use reserved instances** - For EC2 launch type with predictable workloads
✅ **Clean up unused resources** - Deregister old task definitions
✅ **Set log retention** - Don't keep logs indefinitely

Common Use Cases

1. Microservices Architecture

Deploy multiple services that communicate via service discovery:

```
ALB → Web Frontend Service
          ↓ (Service Discovery)
      API Service
          ↓
      Database (RDS)
```

2. Batch Processing

Run tasks on-demand for data processing:

```
aws ecs run-task \
    --cluster batch-cluster \
    --task-definition data-processor:1 \
    --count 10 \
    --launch-type FARGATE
```

3. Scheduled Jobs

Use EventBridge to run tasks on schedule:

```
aws events put-rule \
    --name nightly-backup \
    --schedule-expression "cron(0 2 * * ? *)"

aws events put-targets \
    --rule nightly-backup \
    --targets "Id=1,Arn=arn:aws:ecs:...:cluster/...,RoleArn=...,EcsParameters={TaskDefini
```

Troubleshooting

Common Issues

1. Tasks Not Starting

Check:

- Task execution role has ECR pull permissions
- Subnets have route to internet (for pulling images)
- Security groups allow necessary traffic
- Sufficient capacity in cluster

2. Service Fails to Reach Steady State

Check:

- Health check configuration
- Container crashes (check logs)
- Resource limits too low
- Application startup time

3. Tasks Fail Health Checks

Check:

- Health check path is correct
- Application is listening on correct port
- Health check interval/timeout settings
- Grace period is sufficient

Debugging Commands:

```
# View service events
aws ecs describe-services \
```

```
    --cluster production-cluster \
    --services web-app \
    --query 'services[0].events'

# Get task stopped reason
aws ecs describe-tasks \
    --cluster production-cluster \
    --tasks task-id \
    --query 'tasks[0].stoppedReason'

# View logs
aws logs tail /ecs/web-app --follow
```

Summary

Amazon ECS is a powerful, fully managed container orchestration service that simplifies running containerized applications on AWS.

For Junior Operations Engineers

Must-Know Concepts:

1. **Cluster** - Logical grouping of capacity
2. **Task Definition** - Blueprint for your application
3. **Task** - Running instance for one-time jobs
4. **Service** - Maintains desired count of long-running tasks
5. **Launch Types** - Fargate (serverless) vs EC2 (self-managed)

Key Skills:

- Create and manage clusters
- Write task definitions
- Deploy services with load balancers
- Configure logging and monitoring
- Troubleshoot deployment issues
- Implement auto scaling

Interview Preparation:

- Understand task vs service difference
- Know when to use Fargate vs EC2
- Explain networking modes (especially `awsvpc`)
- Describe deployment strategies
- Be familiar with IAM roles (task role vs execution role)

Quick Reference

Essential Commands

```
# Clusters
aws ecs create-cluster --cluster-name my-cluster
aws ecs list-clusters
aws ecs describe-clusters --cluster my-cluster

# Task Definitions
aws ecs register-task-definition --cli-input-json file://task-def.json
aws ecs list-task-definitions
aws ecs describe-task-definition --task-definition family:revision

# Services
aws ecs create-service --cluster ... --service-name ... --task-definition ...
aws ecs list-services --cluster my-cluster
aws ecs describe-services --cluster my-cluster --services my-service
aws ecs update-service --cluster ... --service ... --desired-count 5

# Tasks
aws ecs run-task --cluster ... --task-definition ...
aws ecs list-tasks --cluster my-cluster
aws ecs describe-tasks --cluster my-cluster --tasks task-id
aws ecs stop-task --cluster my-cluster --task task-id

# Logs
aws logs tail /ecs/app-name --follow
```

This comprehensive guide covers everything you need to know about AWS ECS as a junior operations engineer! Practice these concepts hands-on to solidify your understanding.