**Architecture Components**

**Services Used:**

* **API Gateway (HTTP API)**: Entry point for all requests
* **AWS Lambda (2 functions)**: Startup orchestration & cleanup automation
* **ECS Fargate**: Serverless container platform (Task, Service, Cluster)
* **ECR (Container Registry)**: Docker image storage
* **DynamoDB**: User session tracking & scaling decisions
* **EventBridge**: Automated cleanup scheduling
* **IAM**: Security roles & policies

---# AWS Serverless Container Auto-Scaling Setup Guide

**Project Overview**

This architecture creates a **cost-effective, serverless containerized application** that automatically scales from 0 to N containers based on real-time demand. Perfect for applications with unpredictable traffic patterns, development environments, or microservices that don't need to run 24/7.

**What You'll Achieve:**

* **Zero-cost idle time**: Pay nothing when no users are active
* **Instant scale-up**: Containers start automatically on first request
* **Intelligent scale-out**: Additional containers spawn based on CPU/memory load
* **Automatic cleanup**: Scales back to zero after certain time of inactivity
* **Session persistence**: User sessions tracked across scaling events

**Architecture Flow:** API Gateway → Lambda (Startup) → ECS Fargate → Task (Container) → Auto-scaling based on load EventBridge → Lambda (Cleanup) → Monitor DynamoDB → Scale down when idle

**Step 1: Prepare Your Container Application**

**1.1 Containerize Your Application**

* Create a Dockerfile for your application
* Ensure your app listens on port 80 (or configure CONTAINER\_PORT)
* Test locally with Docker

**Step 2: Create ECR Repository**

**2.1 Create ECR Repository**

* Go to ECR Console → Create Repository
* Repository name: my-scaling-app
* Note down the repository URI: <account-id>.dkr.ecr.us-east-1.amazonaws.com/my-scaling-app

**2.2 Build and Push Docker Image**

* Use ECR push commands provided in console
* Tag your image and push to ECR
* Save the complete image URI for later use

**Step 3: Create DynamoDB Table**

**3.1 Create Session Table**

* Go to DynamoDB Console → Create Table
* Table name: session-table
* Partition key: uuid (String)
* Use on-demand billing
* Enable TTL on expires attribute (optional)

**Step 4: Create IAM Roles and Policies**

**4.1 ECS Task Execution Role**

**Role Name:** ECSTaskExecutionRole  
**Trust Policy:** ECS Tasks service  
**Managed Policy:** AmazonECSTaskExecutionRolePolicy

**4.2 ECS Task Role (for container)**

**Role Name:** ECSTaskRole **Trust Policy:** ECS Tasks service **Custom Policy:**

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"logs:CreateLogGroup",

"logs:CreateLogStream",

"logs:PutLogEvents"

],

"Resource": "\*"

}

]

}

**4.3 Lambda Startup Function Role**

**Role Name:** LambdaStartupRole **Trust Policy:** Lambda service **Custom Policy:**

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"logs:CreateLogGroup",

"logs:CreateLogStream",

"logs:PutLogEvents"

],

"Resource": "arn:aws:logs:\*:\*:\*"

},

{

"Effect": "Allow",

"Action": [

"ecs:UpdateService",

"ecs:DescribeServices",

"ecs:ListTasks",

"ecs:DescribeTasks"

],

"Resource": "\*"

},

{

"Effect": "Allow",

"Action": [

"ec2:DescribeNetworkInterfaces"

],

"Resource": "\*"

},

{

"Effect": "Allow",

"Action": [

"dynamodb:GetItem",

"dynamodb:PutItem",

"dynamodb:UpdateItem"

],

"Resource": "arn:aws:dynamodb:\*:\*:table/session-table"

}

]

}

**4.4 Lambda Cleanup Function Role**

**Role Name:** LambdaCleanupRole **Trust Policy:** Lambda service **Custom Policy:**

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Action": [

"logs:CreateLogGroup",

"logs:CreateLogStream",

"logs:PutLogEvents"

],

"Resource": "arn:aws:logs:\*:\*:\*"

},

{

"Effect": "Allow",

"Action": [

"ecs:UpdateService",

"ecs:DescribeServices"

],

"Resource": "\*"

},

{

"Effect": "Allow",

"Action": [

"dynamodb:GetItem"

],

"Resource": "arn:aws:dynamodb:\*:\*:table/session-table"

}

]

}

**Step 5: Create ECS Infrastructure**

**5.1 Create ECS Cluster**

* Go to ECS Console → Create Cluster
* Cluster name: my-scaling-cluster
* Infrastructure: AWS Fargate (serverless)

**5.2 Create Security Group**

* Go to EC2 Console → Security Groups → Create
* Name: ecs-container-sg
* VPC: Default or your VPC
* Inbound rules:
  + Type: HTTP, Port: 80, Source: 0.0.0.0/0

**5.3 Create Task Definition**

* Go to ECS Console → Task Definitions → Create
* Launch type: Fargate
* Task definition name: my-scaling-task
* Task role: ECSTaskRole
* Task execution role: ECSTaskExecutionRole
* CPU: 0.25 vCPU (256)
* Memory: 0.5 GB (512)

**Container Definition:**

* Container name: my-app-container
* Image URI: Your ECR image URI
* Port mappings: Container port 80, Protocol TCP
* Log configuration: CloudWatch logs
* Log group: /ecs/my-scaling-task

**5.4 Create ECS Service**

* Go to ECS Console → Clusters → your cluster → Services → Create
* Launch type: Fargate
* Task definition: my-scaling-task
* Service name: my-scaling-service
* Number of tasks: 0 (important!)
* VPC: Default or your VPC
* Subnets: Select available subnets
* Security group: ecs-container-sg
* Auto-assign public IP: Enabled

**Auto Scaling Configuration:**

* Enable service auto scaling: Yes
* Minimum number of tasks: 0
* Maximum number of tasks: 3 (or as per your requirement)
* Scaling policy: Target tracking
* ECS service metric: Average CPU utilization (or Average memory utilization)
* Target value: 70%
* Scale-out cooldown period: 300 seconds
* Scale-in cooldown period: 300 seconds

**Step 6: Create Lambda Functions**

**6.1 Create Startup Lambda Function**

* Go to Lambda Console → Create Function
* Function name: container-startup
* Runtime: Python 3.9
* Execution role: LambdaStartupRole

**Configuration:**

* Timeout: 3 minutes
* Memory: 256 MB

**Environment Variables:**

* CLUSTER: my-scaling-cluster
* SERVICE\_NAME: my-scaling-service
* CONTAINER\_PORT: 80
* DYNAMODB\_TABLE: session-table
* SCALE\_UP\_TIMEOUT: 180 (maximum time the lambda waits to get the container running and display the content/website/static page)

**Code:** Upload the startup lambda code provided

**6.2 Create Cleanup Lambda Function**

* Go to Lambda Console → Create Function
* Function name: container-cleanup
* Runtime: Python 3.9
* Execution role: LambdaCleanupRole

**Configuration:**

* Timeout: 60 seconds
* Memory: 128 MB

**Environment Variables:**

* CLUSTER: my-scaling-cluster
* SERVICE\_NAME: my-scaling-service
* DYNAMODB\_TABLE: session-table
* IDLE\_TIMEOUT: 300 (5 minutes - this determines when to scale down the service)

**Code:** Upload the cleanup lambda code provided

**Note:** This lambda is invoked by EventBridge every minute to check the last request time. If the last request from user is at 10:00 PM, when the cleanup lambda runs at 10:06 PM and finds no recent requests, it will scale the ECS service to 0 tasks.

**Step 7: Create EventBridge Rule**

**7.1 Create EventBridge Rule**

* Go to EventBridge Console → Rules → Create Rule
* Name: container-cleanup-schedule
* Rule type: Schedule
* Schedule pattern: Recurring schedule
* Schedule type: Rate-based schedule
* Rate expression: 1 minute (adjust according to how often you want to trigger cleanup)

**Note:** This rule triggers the cleanup lambda every minute to monitor user activity and scale down containers when idle.

**7.2 Add Target**

* Target type: AWS service
* Service: Lambda function
* Function: container-cleanup

**7.3 Lambda Permission**

* EventBridge automatically adds the required permission to invoke the Lambda function

**Step 8: Create API Gateway**

**8.1 Create HTTP API**

* Go to API Gateway Console → Create API → HTTP API
* API name: container-scaling-api
* Description: API for auto-scaling containers

**8.2 Create Integration**

* Integration type: Lambda function
* Lambda function: container-startup
* Payload format version: 2.0

**8.3 Create Routes**

* Route: ANY /{proxy+} (this is a wildcard route that handles all stage and route requests)
* Integration: Your Lambda integration
* Add another route: ANY / (for root path)

**8.4 Create Stage**

* Stage name: prod
* Auto-deploy: Enabled

**8.5 Lambda Permission**

* API Gateway automatically adds permission for Lambda invocation

**Your API URL:** https://<api-id>.execute-api.us-east-1.amazonaws.com/prod

**Step 9: Testing and Validation**

**9.1 Test Flow**

1. **First Request:** Hit your API URL - container should scale from 0→1 (takes 30-60 seconds)
2. **Subsequent Requests:** Should get immediate responses
3. **Wait 5+ Minutes:** Container should scale back to 0 due to inactivity
4. **Next Request:** Container scales up again

**Example Timeline:**

* 10:00 PM: User makes request → Container scales up
* 10:01-10:04 PM: Active requests served immediately
* 10:05 PM: Last user request
* 10:10 PM: Cleanup lambda detects 5+ minutes of inactivity → Scales down to 0
* 10:15 PM: New user request → Container scales up again

**9.2 Monitoring Locations**

* **Lambda Logs:** CloudWatch → Log Groups → /aws/lambda/container-startup and /aws/lambda/container-cleanup
* **ECS Service:** ECS Console → Clusters → Services (check desired/running count)
* **DynamoDB:** DynamoDB Console → Tables → session-table → View items
* **API Gateway:** API Gateway Console → your API → Monitoring

**Configuration Summary**

**IAM Roles Summary**

| **Service** | **Role Name** | **Key Permissions** |
| --- | --- | --- |
| ECS Task Execution | ECSTaskExecutionRole | ECR, CloudWatch Logs |
| ECS Task | ECSTaskRole | CloudWatch Logs |
| Startup Lambda | LambdaStartupRole | ECS, EC2, DynamoDB, Logs |
| Cleanup Lambda | LambdaCleanupRole | ECS, DynamoDB, Logs |

**Environment Variables**

* CLUSTER: ECS cluster name
* SERVICE\_NAME: ECS service name
* CONTAINER\_PORT: Application port (80)
* DYNAMODB\_TABLE: Session table name
* SCALE\_UP\_TIMEOUT: Container start timeout (180s)
* IDLE\_TIMEOUT: Idle time before scale-down (300s - 5 minutes)

**Expected Behavior**

1. **Scale Up:** Container scales from 0→1 on first request
2. **Active State:** Serves requests immediately while active
3. **Scale Down:** After 5 minutes of inactivity, scales to 0
4. **Auto Scale Out:** Scale up even further if load increases (1→3, 5, 8, even 10 containers as per your configuration)
5. **Cost Optimization:** Pay only when containers are running

**Common Issues**

* **Container won't start:** Check task definition, security groups, subnet configuration
* **Lambda timeout:** Verify IAM permissions, network connectivity, lambda runtime configuration
* **Scaling issues:** Check ECS service auto-scaling settings, EventBridge rule is enabled, review cleanup lambda logs
* **API Gateway errors:** Verify Lambda integration and permissions

This serverless architecture provides automatic scaling of your container from 0-1 containers based on demand and further scale out based on demand and workload, optimizing costs by only running containers when needed.