PRJ-SAP-018: CI/CD Pipeline for Containerized Microservices

Certification: AWS Certified Solutions Architect — Professional (SAP-C02) **Domain:** Continuous Integration & Continuous Delivery (CI/CD) **Author:** Mo | CloudGuard Portfolio

---

1. Project Overview

In modern cloud-native environments, the ability to ship software reliably and frequently is a competitive differentiator. Manual deployments are slow, error-prone, and introduce unacceptable risk to production systems. A well-designed CI/CD pipeline eliminates human error from the deployment process, enforces quality gates, and enables teams to release with confidence at any time.

This project implements a complete, production-grade CI/CD pipeline for deploying containerized microservices on AWS. The pipeline automates the entire path from code commit to production deployment, incorporating infrastructure as code, automated Docker image builds, and blue/green deployments to achieve zero-downtime releases. It aligns with the AWS Well-Architected Framework's Operational Excellence pillar and directly maps to the deployment automation patterns tested in the AWS Solutions Architect Professional exam.

The guide incorporates all real-world lessons learned from a live deployment, so every command and configuration detail reflects what actually works in practice.

---

2. Business Context and GRC Mapping

Standardized GRC Mapping

GRC Category	Control / Objective	Implementation Details
Governance	Deployment Auditability	AWS CodePipeline maintains a full audit trail of every pipeline execution, stage transition, and approval action. All pipeline events are logged to AWS CloudTrail, providing a tamper-proof record of who deployed what and when.
Risk Management	Zero-Downtime Release Risk	AWS CodeDeploy's blue/green strategy eliminates deployment downtime. The old task set (blue) remains live until the new task set (green) passes health checks, enabling instant rollback if issues are detected post-deployment.
Compliance	Change Control Enforcement	The manual approval gate between staging and production enforces a formal change control process. No code can reach production without an explicit human approval action, satisfying SOC 2 CC8.1 and ISO 27001 A.12.1.2 change management controls.
Security	Least-Privilege IAM	Each pipeline stage (CodeBuild, CodeDeploy, ECS task execution) uses a dedicated IAM role with the minimum permissions required. Container images are scanned for vulnerabilities in Amazon ECR before deployment.

Framework Alignment

Framework	Relevant Control	How This Project Satisfies It
AWS Well-Architected	OPS 5 — How do you deploy your workload?	Automated pipeline with blue/green and rollback capability
SOC 2 Type II	CC8.1 — Change Management	Manual approval gate + CodePipeline audit trail
ISO 27001	A.12.1.2 — System Change Control	Staged pipeline with enforced approval before production
NIST CSF	PR.IP-3 — Configuration Change Control	IaC-driven pipeline; all changes tracked in CodeCommit

3. Architecture

The architecture is a classic, event-driven CI/CD workflow that automates the path from code commit to production, with a human approval gate enforcing change control between environments.

Pipeline Stages

The pipeline consists of five sequential stages:

1. **Source:** Monitors the main branch of the AWS CodeCommit repository (`ecs-blue-green-app`). Any push to main automatically triggers the pipeline via Amazon EventBridge.
2. **Build:** Runs the CodeBuild project (`build-ecs-app`) that logs into ECR, builds the Docker image tagged with the Git commit hash, pushes the image to ECR, and produces three artifacts: `imageDetail.json` , `appspec.yaml` , and `taskdef.json` .
3. **Deploy to Staging:** Uses AWS CodeDeploy to perform a blue/green deployment against the staging ECS service (`app-service-staging`). CodeDeploy provisions a new green task set alongside the existing blue task set, waits for health checks to pass on port 3000, then shifts traffic.
4. **Manual Approval:** Pauses the pipeline and sends an SNS notification to the approver. A QA engineer validates the staging environment and approves or rejects the release in the CodePipeline console.
5. **Deploy to Production:** Repeats the blue/green process against the production ECS service (`app-service-prod`), with instant rollback capability.

Key AWS Resources

Resource	Name	Purpose
CodeCommit Repository	`ecs-blue-green-app`	Source of truth for application code
CodeBuild Project	`build-ecs-app`	Builds and pushes Docker images to ECR
ECR Repository	`ecs-blue-green-app`	Stores versioned Docker images
ECS Cluster	`app-cluster`	Fargate compute cluster
ECS Service (Staging)	`app-service-staging`	Staging deployment target
ECS Service (Prod)	`app-service-prod`	Production deployment target
ECS Task Definition	`app-task-def`	Container spec (container: `app-container` , port: 3000)
ALB (Staging)	`staging-alb`	Listener 80 (blue), Listener 8080 (green)
ALB (Production)	`prod-alb`	Listener 80 (blue), Listener 8080 (green)
Target Groups	`staging-tg-blue/green` , `prod-tg-blue/green`	HTTP, port 3000, IP type
CodeDeploy App (Staging)	`AppECS-app-cluster-app-service-staging`	Blue/green deployment controller
CodeDeploy App (Prod)	`AppECS-app-cluster-app-service-prod`	Blue/green deployment controller
CodePipeline	`cicd-pipeline`	Pipeline orchestrator

4. Prerequisites and IAM Setup

Before creating any resources, you must configure the required IAM roles. Skipping this step causes failures later in the deployment.

4.1 Create the ECS Service-Linked Role

If you have never used ECS in your AWS account, the ECS service-linked role does not exist and cluster creation will fail. Run this command in CloudShell:

```
aws iam create-service-linked-role --aws-service-name ecs.amazonaws.com
```

> ***Note:*** *If the role already exists, you will receive an error saying it already exists. That is fine — it means you are already covered.*

4.2 Configure the CodePipeline Service Role

The CodePipeline service role requires three specific policy attachments. These are not added by default and will cause authorization failures during the Deploy stages if missing. Run these commands **after** creating the pipeline in Step 5.7 (once the role has been created by the pipeline wizard).

```
# Get the CodePipeline service role name
PIPELINE_ROLE=$(aws iam list-roles \
  --query "Roles[?contains(RoleName, 'AWSCodePipelineServiceRole-us-east-1-cicd-pipeline')].RoleName" \
  --output text)

echo "Pipeline Role: $PIPELINE_ROLE"

# 1. Attach CodeDeploy full access (required for Deploy stages)
aws iam attach-role-policy \
  --role-name $PIPELINE_ROLE \
  --policy-arn arn:aws:iam::aws:policy/AWSCodeDeployFullAccess

# 2. Attach ECS full access (required for ECS Blue/Green action provider)
aws iam attach-role-policy \
  --role-name $PIPELINE_ROLE \
  --policy-arn arn:aws:iam::aws:policy/AmazonECS_FullAccess

# 3. Attach inline PassRole policy (required for CodePipeline to pass roles to ECS tasks)
aws iam put-role-policy \
  --role-name $PIPELINE_ROLE \
  --policy-name PassRoleForECS \
  --policy-document '{
    "Version": "2012-10-17",
    "Statement": [{
      "Effect": "Allow",
      "Action": "iam:PassRole",
      "Resource": "*",
      "Condition": {
        "StringLike": {
          "iam:PassedToService": "ecs-tasks.amazonaws.com"
        }
      }
    }]
  }'
```

---

5. Step-by-Step Deployment Guide

Step 5.1: Create the ECR Repository, ECS Cluster, and CodeCommit Repo

Navigate to the AWS Console and create the following resources in **us-east-1**:

1. **Amazon ECR:** Create a private repository named `ecs-blue-green-app`. Enable **Scan on push** to automatically check images for CVEs. Note the repository URI.
2. **Amazon ECS:** Create a cluster named `app-cluster` using the AWS Fargate (Serverless) launch type.

3. **AWS CodeCommit:** Create a repository named `ecs-blue-green-app`.

Step 5.2: Create Application Source Files via CloudShell

Open **AWS CloudShell** and run the following commands to create all application files using heredocs.

1. Create the working directory:

```
mkdir -p ~/ecs-blue-green-app && cd ~/ecs-blue-green-app
```

2. Create `app.js`:

This is a simple Node.js Express server. It exposes a root route `/` returning a version string, and a `/health` route returning HTTP 200. The health route is required by both the Docker `HEALTHCHECK` and the Application Load Balancer to verify the container is ready to receive traffic.

```
cat > app.js << 'EOF'
const express = require("express");
const app = express();

app.get("/", (req, res) => res.send("Hello from CloudGuard! Version 1"));
app.get("/health", (req, res) => res.status(200).json({ status: "healthy" }));

app.listen(3000, () => console.log("App listening on port 3000"));
EOF
```

3. Create `package.json`:

```
cat > package.json << 'EOF'
{
  "name": "ecs-blue-green-app",
  "version": "1.0.0",
  "dependencies": {
    "express": "^4.18.0"
  }
}
EOF
```

4. Create `Dockerfile`:

Two critical real-world fixes are baked into this Dockerfile:

- **Fix #7 — Docker Hub Rate Limit:** Use `public.ecr.aws/docker/library/node:18-alpine` instead of `node:18-alpine`. Docker Hub enforces a 100-pull-per-6-hours rate limit on unauthenticated requests. CodeBuild will fail with a `429 Too Many Requests` error if you use the Docker Hub image directly.
- **Fix #17 — HEALTHCHECK Reliability:** Use a Node.js HTTP check instead of `wget`. The `wget` binary is not reliably available in all Alpine variants and can cause the HEALTHCHECK to fail even when the application is healthy.

```
cat > Dockerfile << 'EOF'
FROM public.ecr.aws/docker/library/node:18-alpine
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install --production
COPY . .
EXPOSE 3000
HEALTHCHECK --interval=30s --timeout=5s --start-period=10s --retries=3 \
  CMD node -e "require('http').get('http://localhost:3000/health', (r) => process.exit(r.statusCode === 200 ? 0 :
1)).on('error', () => process.exit(1))"
CMD ["node", "app.js"]
EOF
```

5. Create `buildspec.yml`:

Three critical real-world fixes are baked into this buildspec:

- **Fix #8 — YAML_FILE_ERROR:** The file must be named exactly `buildspec.yml` and committed to the repository root. Verify with `git ls-files` after committing.
- **Fix #12 — imageDetail.json format:** For ECS Blue/Green deployments, CodeDeploy requires a file named `imageDetail.json` (not `imagedefinitions.json`) with the format `{"ImageURI":"..."}`. Using `imagedefinitions.json` causes a "file not found" error in the Deploy stage.
- **Fix #13 — imageDetail.json empty:** Use `echo` (not `printf`) to generate `imageDetail.json`. The `printf` command can cause variable expansion issues that result in an empty or malformed file.

```
cat > buildspec.yml << 'EOF'
version: 0.2
phases:
  pre_build:
    commands:
      - echo Logging in to Amazon ECR...
      - aws ecr get-login-password --region $AWS_DEFAULT_REGION | docker login --username AWS --password-stdin
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com
      - COMMIT_HASH=$(echo $CODEBUILD_RESOLVED_SOURCE_VERSION | cut -c 1-7)
      - IMAGE_TAG=${COMMIT_HASH:=latest}
  build:
    commands:
      - echo Build started on `date`
      - docker build -t $IMAGE_REPO_NAME:$IMAGE_TAG .
      - docker tag $IMAGE_REPO_NAME:$IMAGE_TAG
$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$IMAGE_REPO_NAME:$IMAGE_TAG
  post_build:
    commands:
      - echo Pushing the Docker image...
      - docker push $AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$IMAGE_REPO_NAME:$IMAGE_TAG
      - echo "
{\"ImageURI\":\"$AWS_ACCOUNT_ID.dkr.ecr.$AWS_DEFAULT_REGION.amazonaws.com/$IMAGE_REPO_NAME:$IMAGE_TAG\"}" >
imageDetail.json
artifacts:
  files:
    - imageDetail.json
    - appspec.yaml
    - taskdef.json
EOF
```

Step 5.3: Configure Git and Push Initial Files to CodeCommit

We commit only the application source files at this point. The `appspec.yaml` and `taskdef.json` files depend on the ECS task definition ARN, which does not exist yet. They will be created and committed in Step 5.6.

```
git config --global user.name "CloudGuard Admin"
git config --global user.email "admin@cloudguard.local"
git config --global credential.helper '!aws codecommit credential-helper $@'
git config --global credential.UseHttpPath true

git init
git add app.js package.json Dockerfile buildspec.yml
git commit -m "Initial commit: CloudGuard ECS blue/green app source files"
git remote add origin https://git-codecommit.us-east-1.amazonaws.com/v1/repos/ecs-blue-green-app
git checkout -b main
git push -u origin main
```

Verify the files were committed correctly:

```
git ls-files
```

You should see: `Dockerfile`, `app.js`, `buildspec.yml`, `package.json`.

Step 5.4: Create the CodeBuild Project

Navigate to **CodeBuild → Create build project** and configure as follows:

Setting	Value	Rationale
Project name	`build-ecs-app`	Standard naming
Source provider	AWS CodeCommit	Connects to our repo
Repository	`ecs-blue-green-app`	Our source repo
Branch	`main`	Trigger on main branch
Environment image	Managed image	AWS-managed build environment
Operating system	Amazon Linux	Standard Linux build
Runtime	Standard	General-purpose runtime
Image	`aws/codebuild/amazonlinux-x86_64-standard:6.0`	Latest standard image
Privileged mode	**Enabled**	**Fix #2: Required for Docker**
Service role	New service role	Auto-created by CodeBuild

Fix #2 — Privileged Mode: Under **Additional configuration**, you must check the **Enable this flag if you want to build Docker images or want your builds to get elevated privileges** checkbox. Without this, Docker cannot run inside the build container and your build will fail with a "Cannot connect to the Docker daemon" error.

Environment Variables: Add the following environment variables:

Name	Value	Type
`AWS_ACCOUNT_ID`	`094869897684`	Plaintext
`IMAGE_REPO_NAME`	`ecs-blue-green-app`	Plaintext

IAM Policy for CodeBuild: After creating the project, navigate to the CodeBuild service role in IAM and attach an inline policy allowing ECR access:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ecr:GetAuthorizationToken",
        "ecr:BatchCheckLayerAvailability",
        "ecr:CompleteLayerUpload",
        "ecr:InitiateLayerUpload",
        "ecr:PutImage",
        "ecr:UploadLayerPart"
      ],
      "Resource": "*"
    }
  ]
}
```

Step 5.5: Create ALBs and Target Groups

Fix #3 — Target Group Protocol and Port: Target groups must use **HTTP protocol on port 3000** (not TCP, not port 80). Using TCP or port 80 causes CodeDeploy to fail health checks because the container listens on port 3000.

Navigate to **EC2 → Target Groups** and create four target groups:

Name	Target Type	Protocol	Port	Health Check Path
`staging-tg-blue`	IP addresses	**HTTP**	**3000**	`/health`
`staging-tg-green`	IP addresses	**HTTP**	**3000**	`/health`
`prod-tg-blue`	IP addresses	**HTTP**	**3000**	`/health`
`prod-tg-green`	IP addresses	**HTTP**	**3000**	`/health`

Navigate to **EC2 → Load Balancers** and create two Application Load Balancers:

ALB Name	Listener 80	Listener 8080
`staging-alb`	Forward to `staging-tg-blue`	Forward to `staging-tg-green`
`prod-alb`	Forward to `prod-tg-blue`	Forward to `prod-tg-green`

Fix #16 — Security Group for ECS Tasks: The Fargate task security group must allow inbound TCP port 3000 from the ALB security group. Without this, the “Install” phase of the CodeDeploy deployment will get stuck for 35+ minutes and eventually time out.

Run this in CloudShell to add the required inbound rule:

```
# Get the ALB security group ID
ALB_SG=$(aws elbv2 describe-load-balancers \
  --names staging-alb \
  --query 'LoadBalancers[0].SecurityGroups[0]' \
  --output text)

# Get the default security group (used by ECS tasks)
TASK_SG=$(aws ec2 describe-security-groups \
  --filters Name=group-name,Values=default \
  --query 'SecurityGroups[0].GroupId' \
  --output text)

echo "ALB SG: $ALB_SG"
echo "Task SG: $TASK_SG"

# Allow inbound TCP 3000 from ALB to ECS tasks
aws ec2 authorize-security-group-ingress \
  --group-id $TASK_SG \
  --protocol tcp \
  --port 3000 \
  --source-group $ALB_SG
```

Step 5.6: Create the ECS Task Definition and Services

1. Create the ECS Task Definition:

Navigate to **ECS → Task Definitions → Create new task definition** and configure:

Setting	Value
Task definition family	`app-task-def`
Launch type	AWS Fargate
CPU	0.25 vCPU
Memory	0.5 GB
Container name	`app-container`
Image URI	`<your-account-id>.dkr.ecr.us-east-1.amazonaws.com/ecs-blue-green-app:latest`
Container port	`3000`
Protocol	TCP

2. Create ECS Services via CLI:

Fix #4 — ECS Console Bug: The ECS console has a known bug where the green target group dropdown is not populated during blue/green service creation. Create the services using the AWS CLI with the `--deployment-controller type=CODE_DEPLOY` flag to bypass this.

```
# Get subnet IDs (comma-separated)
SUBNETS=$(aws ec2 describe-subnets \
  --filters Name=defaultForAz,Values=true \
  --query 'Subnets[*].SubnetId' \
  --output text | tr '\t' ',')

# Get default security group ID
SG=$(aws ec2 describe-security-groups \
  --filters Name=group-name,Values=default \
  --query 'SecurityGroups[0].GroupId' \
  --output text)

# Get staging blue target group ARN
STAGING_TG_BLUE=$(aws elbv2 describe-target-groups \
  --names staging-tg-blue \
  --query 'TargetGroups[0].TargetGroupArn' \
  --output text)

# Get prod blue target group ARN
PROD_TG_BLUE=$(aws elbv2 describe-target-groups \
  --names prod-tg-blue \
  --query 'TargetGroups[0].TargetGroupArn' \
  --output text)

# Create Staging Service
aws ecs create-service \
  --cluster app-cluster \
  --service-name app-service-staging \
  --task-definition app-task-def \
  --desired-count 2 \
  --launch-type FARGATE \
  --deployment-controller type=CODE_DEPLOY \
  --network-configuration "awsvpcConfiguration={subnets=[$SUBNETS],securityGroups=[$SG],assignPublicIp=ENABLED}" \
  --load-balancers "targetGroupArn=$STAGING_TG_BLUE,containerName=app-container,containerPort=3000"

# Create Production Service
aws ecs create-service \
  --cluster app-cluster \
  --service-name app-service-prod \
  --task-definition app-task-def \
  --desired-count 2 \
  --launch-type FARGATE \
  --deployment-controller type=CODE_DEPLOY \
  --network-configuration "awsvpcConfiguration={subnets=[$SUBNETS],securityGroups=[$SG],assignPublicIp=ENABLED}" \
  --load-balancers "targetGroupArn=$PROD_TG_BLUE,containerName=app-container,containerPort=3000"
```

3. Generate appspec.yaml and taskdef.json:

Fix #15 — AppSpec Placeholders: The `appspec.yaml` must use the `<TASK_DEFINITION>` placeholder (not a real ARN). CodeDeploy replaces this at deploy time. The `ContainerName` must match exactly (`app-container`) and `ContainerPort` must be `3000`.

Fix #11 — taskdef.json Placeholder: The `taskdef.json` must use the `<IMAGE>` placeholder (not a real image URI). CodePipeline replaces this with the URI from `imageDetail.json` at deploy time.

```
cd ~/ecs-blue-green-app

# Create appspec.yaml with required placeholders
cat > appspec.yaml << 'EOF'
version: 0.0
Resources:
  - TargetService:
      Type: AWS::ECS::Service
      Properties:
        TaskDefinition: <TASK_DEFINITION>
        LoadBalancerInfo:
          ContainerName: "app-container"
          ContainerPort: 3000
EOF

# Export current task definition to taskdef.json
aws ecs describe-task-definition \
  --task-definition app-task-def \
  --query 'taskDefinition' > taskdef.json

# Replace the hardcoded image URI with the <IMAGE> placeholder
sed -i 's|"image": ".*ecs-blue-green-app.*"|"image": "<IMAGE>"|' taskdef.json

# Verify the placeholder was applied
grep '"image"' taskdef.json

# Commit and push
git add appspec.yaml taskdef.json
git commit -m "Add CodeDeploy configuration: appspec.yaml and taskdef.json"
git push
```

Step 5.7: Create CodeDeploy Applications and CodePipeline

Fix #6 — CodeDeploy Apps Not Auto-Created: When ECS services are created via the CLI (as we did in Step 5.6), AWS does not automatically create the corresponding CodeDeploy applications and deployment groups. You must create them manually before configuring the pipeline deploy stages.

Fix #5 — InvalidDeploymentStyleException: The deployment group must be created with `--deployment-style deploymentType=BLUE_GREEN,deploymentOption=WITH_TRAFFIC_CONTROL`. Without this, CodeDeploy throws an `InvalidDeploymentStyleException`.

```
# Get the CodeDeploy service role ARN
CODEDEPLOY_ROLE=$(aws iam get-role \
  --role-name AWSCodeDeployRoleForECS \
  --query 'Role.Arn' \
  --output text)

# Get staging listener ARN (port 80)
STAGING_ALB_ARN=$(aws elbv2 describe-load-balancers \
  --names staging-alb \
  --query 'LoadBalancers[0].LoadBalancerArn' \
  --output text)

STAGING_LISTENER=$(aws elbv2 describe-listeners \
  --load-balancer-arn $STAGING_ALB_ARN \
  --query 'Listeners[?Port==`80`].ListenerArn' \
  --output text)

# Get prod listener ARN (port 80)
PROD_ALB_ARN=$(aws elbv2 describe-load-balancers \
  --names prod-alb \
  --query 'LoadBalancers[0].LoadBalancerArn' \
  --output text)

PROD_LISTENER=$(aws elbv2 describe-listeners \
  --load-balancer-arn $PROD_ALB_ARN \
  --query 'Listeners[?Port==`80`].ListenerArn' \
  --output text)

# --- Create Staging CodeDeploy App and Deployment Group ---
aws deploy create-application \
  --application-name AppECS-app-cluster-app-service-staging \
  --compute-platform ECS

aws deploy create-deployment-group \
  --application-name AppECS-app-cluster-app-service-staging \
  --deployment-group-name DgpECS-app-cluster-app-service-staging \
  --deployment-config-name CodeDeployDefault.ECSAllAtOnce \
  --deployment-style deploymentType=BLUE_GREEN,deploymentOption=WITH_TRAFFIC_CONTROL \
  --service-role-arn $CODEDEPLOY_ROLE \
  --ecs-services clusterName=app-cluster,serviceName=app-service-staging \
  --load-balancer-info "targetGroupPairInfoList=[{targetGroups=[{name=staging-tg-blue},{name=staging-tg-
green}],prodTrafficRoute={listenerArns=[$STAGING_LISTENER]}}]" \
  --blue-green-deployment-configuration "terminateBlueInstancesOnDeploymentSuccess=
{action=TERMINATE,terminationWaitTimeInMinutes=5},deploymentReadyOption={actionOnTimeout=CONTINUE_DEPLOYMENT}"

# --- Create Production CodeDeploy App and Deployment Group ---
aws deploy create-application \
  --application-name AppECS-app-cluster-app-service-prod \
  --compute-platform ECS

aws deploy create-deployment-group \
  --application-name AppECS-app-cluster-app-service-prod \
  --deployment-group-name DgpECS-app-cluster-app-service-prod \
  --deployment-config-name CodeDeployDefault.ECSAllAtOnce \
  --deployment-style deploymentType=BLUE_GREEN,deploymentOption=WITH_TRAFFIC_CONTROL \
  --service-role-arn $CODEDEPLOY_ROLE \
  --ecs-services clusterName=app-cluster,serviceName=app-service-prod \
  --load-balancer-info "targetGroupPairInfoList=[{targetGroups=[{name=prod-tg-blue},{name=prod-tg-
green}],prodTrafficRoute={listenerArns=[$PROD_LISTENER]}}]" \
  --blue-green-deployment-configuration "terminateBlueInstancesOnDeploymentSuccess=
{action=TERMINATE,terminationWaitTimeInMinutes=5},deploymentReadyOption={actionOnTimeout=CONTINUE_DEPLOYMENT}"
```

Create the Pipeline:

1. Navigate to **CodePipeline → Create pipeline** and name it `cicd-pipeline`.

2. In the wizard, add the **Source** stage (CodeCommit, `ecs-blue-green-app`, main branch) and the **Build** stage (CodeBuild, `build-ecs-app`).
3. Click **Skip deploy stage** in the wizard.
4. After the pipeline is created, apply the IAM role policies from Section 4.2.
5. Click **Edit** on the pipeline and add the following stages manually:

Stage: Deploy-Staging

- Action name: `Deploy-Staging`
- Action provider: **Amazon ECS (Blue/Green)** — Fix #9: must NOT be CodeBuild or standard ECS
- AWS CodeDeploy application: `AppECS-app-cluster-app-service-staging`
- AWS CodeDeploy deployment group: `DgpECS-app-cluster-app-service-staging`
- Amazon ECS task definition: `BuildArtifact` → `taskdef.json`
- AWS CodeDeploy AppSpec file: `BuildArtifact` → `appspec.yaml`
- Dynamically update task definition image: `BuildArtifact` → `imageDetail.json`
- Input artifact: `BuildArtifact`

Stage: Manual-Approval

- Action name: `Manual-Approval`
- Action provider: **Manual approval**
- SNS topic ARN: (optional) your notification topic

Stage: Deploy-Prod

- Action name: `Deploy-Prod`
- Action provider: **Amazon ECS (Blue/Green)**
- AWS CodeDeploy application: `AppECS-app-cluster-app-service-prod`
- AWS CodeDeploy deployment group: `DgpECS-app-cluster-app-service-prod`
- (Same artifact configuration as Deploy-Staging)

---

6. Testing the Pipeline

Trigger a deployment by updating the application code:

```
cd ~/ecs-blue-green-app
sed -i 's/Version 1/Version 2/' app.js
git add app.js
git commit -m "Deploy Version 2: test blue/green pipeline"
git push
```

Watch the pipeline in the CodePipeline console. It will:

1. Detect the source change and trigger automatically.
2. Build the Docker image and push it to ECR.
3. Deploy to staging using blue/green, shifting traffic after health checks pass.
4. Pause at the Manual Approval stage.
5. After you click **Approve**, deploy to production with zero downtime.

---

7. Troubleshooting Reference

The following table documents all real-world issues encountered during the live deployment of this project, along with their root causes and verified fixes.

#	Symptom / Error	Root Cause	Fix
1	ECS cluster creation fails with IAM error	ECS service-linked role does not exist	`aws iam create-service-linked-role --aws-service-name ecs.amazonaws.com`
2	CodeBuild fails: “Cannot connect to Docker daemon”	Privileged mode not enabled	Enable **Privileged mode** under Additional Configuration in CodeBuild
3	CodeDeploy health checks fail; target group shows unhealthy	Target groups use TCP or port 80 instead of HTTP/3000	Recreate target groups with **Protocol: HTTP, Port: 3000**
4	ECS console blocks green target group selection	Known ECS console UI bug	Create ECS services via CLI with `--deployment-controller type=CODE_DEPLOY`
5	`InvalidDeploymentStyleException`	Deployment group created without Blue/Green style flag	Add `--deployment-style deploymentType=BLUE_GREEN,deploymentOption=WITH_TRAFFIC_CONTI` to `create-deployment-group`
6	Pipeline Deploy stage shows “No options” for CodeDeploy app	CLI-created ECS services do not auto-generate CodeDeploy apps	Manually run `aws deploy create-application` and `aws deploy create-deployment-group`
7	CodeBuild fails: “429 Too Many Requests”	Docker Hub rate limit exceeded	Change Dockerfile base image to `FROM public.ecr.aws/docker/library/node:18-alpine`
8	CodeBuild fails: `YAML_FILE_ERROR`	`buildspec.yml` not committed or misnamed	Verify with `git ls-files` ; ensure file is named exactly `buildspec.ym`
9	Deploy-Staging stage fails or does nothing	Action provider set to CodeBuild instead of ECS Blue/Green	Edit pipeline action and change provider to **Amazon ECS (Blue/Green**
10	`codedeploy:GetApplication` not authorized	CodePipeline service role missing CodeDeploy permissions	Attach `AWSCodeDeployFullAccess` to the CodePipeline service role
11	`<IMAGE> does not match any missing containers`	`taskdef.json` contains a real image URI instead of `<IMAGE>` placeholder	Run `sed -i ’s
12	Deploy stage fails: `imageDetail.json` not found	`buildspec.yml` generates `imagedefinitions.json` instead of `imageDetail.json`	Change buildspec output to `imageDetail.json` with format `{"ImageURI":"..."}`
13	`imageDetail.json` is empty or malformed	`printf` does not expand variables correctly	Use `echo` instead of `printf` to generate `imageDetail.json`
14	`ecs:RegisterTaskDefinition` not authorized	CodePipeline service role missing ECS permissions	Attach `AmazonECS_FullAccess` + inline `iam:PassRole` policy to CodePipeline role
15	AppSpec validation error	`appspec.yaml` uses real ARN or wrong container name/port	Use `<TASK_DEFINITION>` placeholder; `ContainerName: app-contain` `ContainerPort: 3000`
16	Install phase stuck for 35+ minutes	Fargate task security group blocks inbound TCP 3000 from ALB	Add inbound TCP 3000 rule from ALB security group to ECS task securi group
17	Container HEALTHCHECK fails in Alpine	`wget` not reliably available in Alpine Linux	Replace with Node.js HTTP check: `CMD node -e "require('http').get('http://localhost:3000/health', (r) =>`

#	Symptom / Error	Root Cause	Fix
			`process.exit(r.statusCode === 200 ? 0 : 1)).on('error', () => process.exit(1))"`

---

8. Cleanup Order

To avoid dependency errors, delete resources in this exact order:

1. **CodePipeline** — `cicd-pipeline`
2. **CodeDeploy Applications** — `AppECS-app-cluster-app-service-staging` and `AppECS-app-cluster-app-service-prod`
3. **ECS Services** — `app-service-staging` and `app-service-prod` (wait for tasks to drain to 0)
4. **ECS Cluster** — `app-cluster`
5. **Application Load Balancers** — `staging-alb` and `prod-alb`
6. **Target Groups** — all four target groups
7. **ECR Repository** — delete all images first, then delete the repository `ecs-blue-green-app`
8. **CodeBuild Project** — `build-ecs-app`
9. **CodeCommit Repository** — `ecs-blue-green-app`
10. **IAM Roles** — CodeBuild service role, CodePipeline service role (if no longer needed)

Part of the CloudGuard 100-Project Portfolio — PRJ-SAP-018 of 100