NodeJS Application Deployment on AWS EC2 (Auto Scaled) with GitHub Actions & CodeDeploy

Table of Contents

Objective

Deploy a Node.js application on AWS EC2 instances with automated scaling and continuous deployment. The infrastructure uses Terraform for provisioning, AWS CodeDeploy for application deployment, and GitHub Actions for CI/CD automation with OIDC authentication.

Prerequisites

- AWS account with administrative access
- Terraform >= 1.0 installed
- AWS CLI configured
- GitHub repository with Actions enabled
- Node.js application listening on port 8081
- S3 bucket and DynamoDB table for Terraform backend

Project Structure

Terraform Infrastructure:

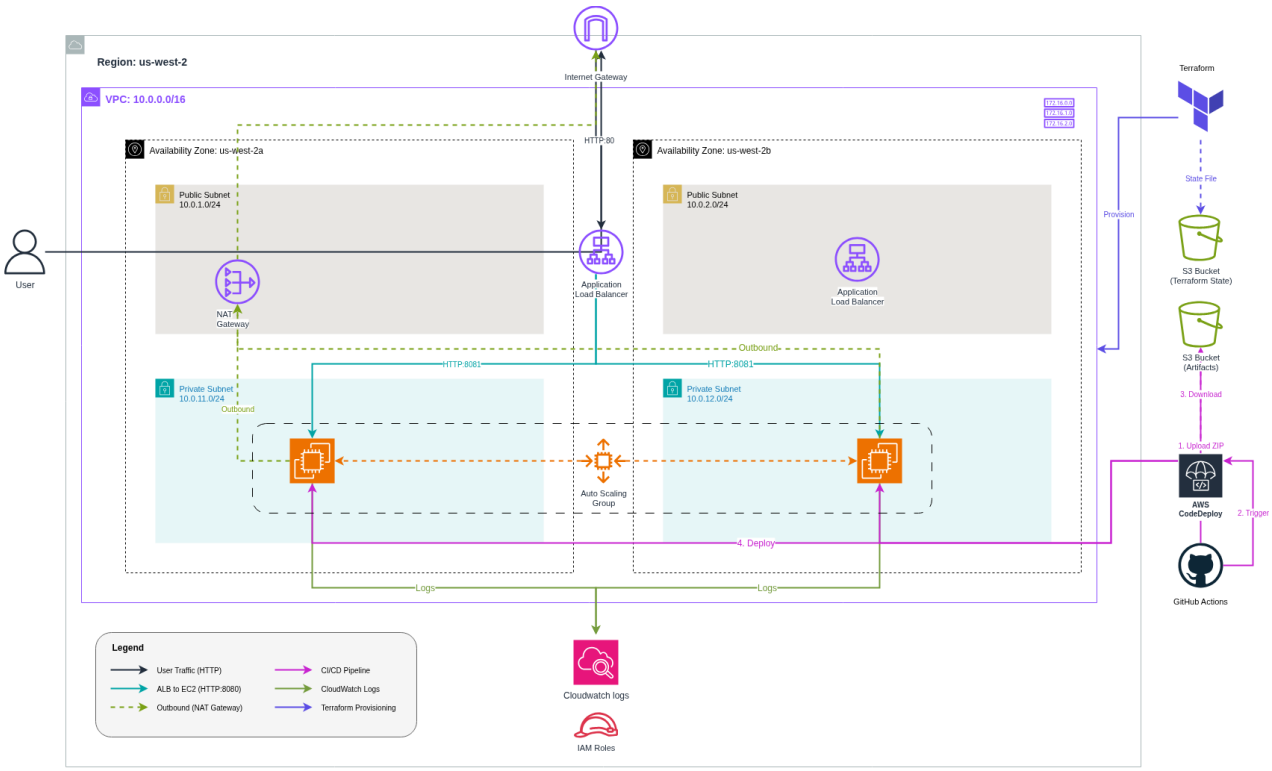
```
├── backend.tf
├── main.tf
├── modules
│   ├── alb
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf
│   ├── codedeploy
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf
│   ├── ec2
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf
```

```
│   ├── iam
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf
│   ├── security
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf
│   └── vpc
│       ├── main.tf
│       ├── outputs.tf
│       └── variables.tf
├── outputs.tf
├── terraform.tfvars
└── variables.tf
```

Application Repository:

index.js - Node.js application
package.json - Dependencies
appspec.yml - CodeDeploy configuration
codedeploy/scripts/ - Deployment lifecycle hooks
scripts/setup.sh - EC2 user data script
.github/workflows/deploy.yml - CI/CD pipeline

Architecture



Infrastructure Components

VPC Module

- VPC: 10.0.0.0/16 with DNS support enabled
- Availability Zones: us-west-2a, us-west-2b
- Public Subnets: 10.0.1.0/24, 10.0.2.0/24 (host ALB)
- Private Subnets: 10.0.11.0/24, 10.0.12.0/24 (host EC2)
- Internet Gateway: Public internet access
- NAT Gateway: Single NAT in first AZ for cost optimization
- Route Tables: Public routes to IGW, private routes to NAT

Security Module

ALB Security Group:

- Ingress: HTTP port 80 from 0.0.0.0/0
- Egress: All traffic

EC2 Security Group:

- Ingress: Port 8080 from ALB security group only
- Egress: All traffic (for package downloads, CodeDeploy)

ALB Module

- Load Balancer: Internet-facing, spans both public subnets
- Target Group: IP-based, port 8080, HTTP protocol
- Health Check: Path /, interval 30s, 2 healthy threshold
- Listener: HTTP port 80, forwards to target group

IAM Module

The IAM module provisions three distinct roles with specific permissions following the principle of least privilege.

EC2 Instance Role

Purpose:

Attached to EC2 instances, allows them to interact with AWS services during runtime.

Trust Policy:

Allows ec2.amazonaws.com to assume this role

Permissions:

- S3 GetObject and ListBucket on deployment bucket (CodeDeploy artifacts)
- CloudWatch Logs CreateLogGroup, CreateLogStream, PutLogEvents
- CloudWatch PutMetricData for custom metrics
- AWS Managed Policy: AmazonSSMManagedInstanceCore (for SSM access)

CodeDeploy Service Role

Purpose:

Used by CodeDeploy service to manage deployments on EC2 instances and Auto Scaling Groups.

Trust Policy:

Allows codedeploy.amazonaws.com to assume this role

Permissions:

- AWS Managed Policy: AWSCodeDeployRole (EC2, ALB, Auto Scaling operations)
- Auto Scaling: CompleteLifecycleAction, DescribeAutoScalingGroups
- Auto Scaling: PutLifecycleHook, DeleteLifecycleHook, RecordLifecycleActionHeartbeat
- EC2: DescribeInstances, DescribeInstanceStatus
- ELB: RegisterTargets, DeregisterTargets, DescribeTargetHealth

GitHub Actions Role

Purpose:

Enables GitHub Actions to authenticate with AWS using OIDC without storing long-lived credentials.

Trust Policy:

- Federated principal: GitHub OIDC provider
- Condition: StringEquals on audience (sts.amazonaws.com)
- Condition: StringLike on subject (specific GitHub repository)

Permissions:

- S3: PutObject, GetObject, ListBucket on deployment bucket
- CodeDeploy: CreateDeployment (scoped to all resources)
- CodeDeploy: GetDeployment, GetApplicationRevision, RegisterApplicationRevision
- CodeDeploy: GetDeploymentConfig

OpenID Connect (OIDC) Authentication

OIDC eliminates the need to store AWS access keys in GitHub. Instead, GitHub Actions requests temporary credentials directly from AWS STS.

How it works:

1. GitHub generates a signed JWT token for each workflow run
2. Token contains claims (repository, branch, commit SHA)
3. AWS validates token signature against GitHub's public keys
4. AWS checks claims against IAM role trust policy
5. AWS issues temporary credentials (valid 1 hour)
6. Workflow uses credentials for AWS API calls

Security benefits:

- No long-lived credentials stored in GitHub
- Credentials auto-expire after workflow completion
- Fine-grained access control via trust policy conditions
- Complete audit trail in CloudTrail

EC2 Module

- Launch Template: Amazon Linux 2023, t3.micro, user data for host preparation
- Auto Scaling Group: Min 1, Max 2, Desired 1
- Deployment: Private subnets across both AZs
- Health Check: EC2 type (ELB after first deployment)
- Target Tracking Policy: 50% average CPU utilization
- CloudWatch Log Group: /aws/ec2/wu-node-app, 7-day retention
- User Data: Installs Node.js, PM2, CodeDeploy agent, CloudWatch agent

CodeDeploy Module

- S3 Bucket: Stores deployment artifacts with versioning enabled
- CodeDeploy Application: Server compute platform (EC2/On-Premises)
- Deployment Group: Associated with ASG and ALB target group
- Deployment Config: OneAtATime (rolling deployment)
- Auto Rollback: Enabled on deployment failure
- Traffic Routing: ALB integration with automatic deregistration

GitHub Actions Pipeline

The CI/CD pipeline is defined in .github/workflows/deploy.yml and automates the deployment process.

Pipeline Configuration

Trigger:

Manual via workflow_dispatch (can be changed to trigger on push to main)

Permissions:

- id-token: write (required for OIDC)
- contents: read (checkout repository)

Environment Variables:

- AWS_REGION: us-west-2
- CODEDEPLOY_APP: Application name
- CODEDEPLOY_GROUP: Deployment group name
- S3_BUCKET: Deployment artifact bucket

Pipeline Steps

Step 1: Checkout Code

Clones the repository including application code, appspec.yml, and deployment scripts.

Step 2: Configure AWS Credentials

Uses aws-actions/configure-aws-credentials action with OIDC. Exchanges GitHub JWT token for temporary AWS credentials valid for the duration of the workflow.

Step 3: Create Deployment Package

Creates a ZIP archive containing:

- appspec.yml (root level)
- Application code (index.js, package.json)
- Deployment scripts (codedeploy/scripts/)

Excludes: .git, node_modules, terraform directory

Step 4: Upload to S3

Uploads deployment.zip to S3 with unique key:

deployments/deployment-<timestamp>-<commit-sha>.zip

Step 5: Trigger CodeDeploy

Creates a CodeDeploy deployment specifying:

- Application name
- Deployment group
- S3 location of artifact

Returns deployment ID for tracking

CodeDeploy Lifecycle

CodeDeploy orchestrates application deployment using lifecycle hooks defined in appspec.yml.

AppSpec Configuration

Files Section:

Copies all files from deployment archive to /opt/nodeapp on EC2 instances.

Hooks:

All hooks run as root to avoid permission conflicts.

Deployment Scripts

BeforeInstall (before_install.sh):

- Stops all PM2 processes
- Deletes all PM2 instances
- Cleans /opt/nodeapp directory
- Sets ownership to ec2-user

AfterInstall (install_deps.sh):

- Changes to /opt/nodeapp
- Ensures correct ownership
- Runs npm ci --omit=dev (production dependencies only)

ApplicationStart (start_server.sh):

- Deletes previous PM2 nodeapp instance if exists
- Starts application: PORT=8080 pm2 start index.js --name nodeapp
- Logs to /var/log/nodeapp/app.log
- Saves PM2 process list

ValidateService (validate_service.sh):

- Waits 15 seconds for application startup
- Checks PM2 process is running
- Curls http://localhost:8080/ to verify response
- Exits with error if validation fails (triggers rollback)

Complete Deployment Workflow

Initial Infrastructure Deployment

Step 1: Provision Infrastructure

Run terraform apply from the root directory. This creates:

- VPC, subnets, gateways, route tables
- Security groups
- Application Load Balancer
- IAM roles and policies
- Auto Scaling Group with 1 EC2 instance
- CodeDeploy application and deployment group
- S3 bucket for deployment artifacts

Step 2: EC2 Bootstrap

When EC2 instances launch, user data script executes:

- Installs Node.js, npm, PM2
- Installs and starts CodeDeploy agent
- Configures CloudWatch agent
- Creates /opt/nodeapp and /var/log/nodeapp directories
- Does NOT deploy application (directory remains empty)

Step 3: Instance Health

EC2 instances pass health checks (EC2 type, not ELB). ASG reaches desired capacity of 1. No application is running yet.

First Application Deployment

Step 1: Configure GitHub

Add repository secret:

- AWS_ROLE_ARN (from terraform output)

Add repository variables:

- AWS_REGION, CODEDEPLOY_APP, CODEDEPLOY_GROUP, S3_BUCKET

Step 2: Trigger Pipeline

Manually trigger workflow_dispatch or push to main branch. Pipeline executes all steps.

Step 3: CodeDeploy Execution

On each EC2 instance, CodeDeploy agent:

- Downloads deployment.zip from S3
- Executes BeforeInstall (cleanup)
- Copies files to /opt/nodeapp
- Executes AfterInstall (npm ci)
- Executes ApplicationStart (pm2 start)

- Executes ValidateService (health check)

Step 4: Application Running

Application now listens on port 8080. ALB health checks pass. Instance becomes healthy in target group. Application accessible via ALB DNS.

Subsequent Deployments

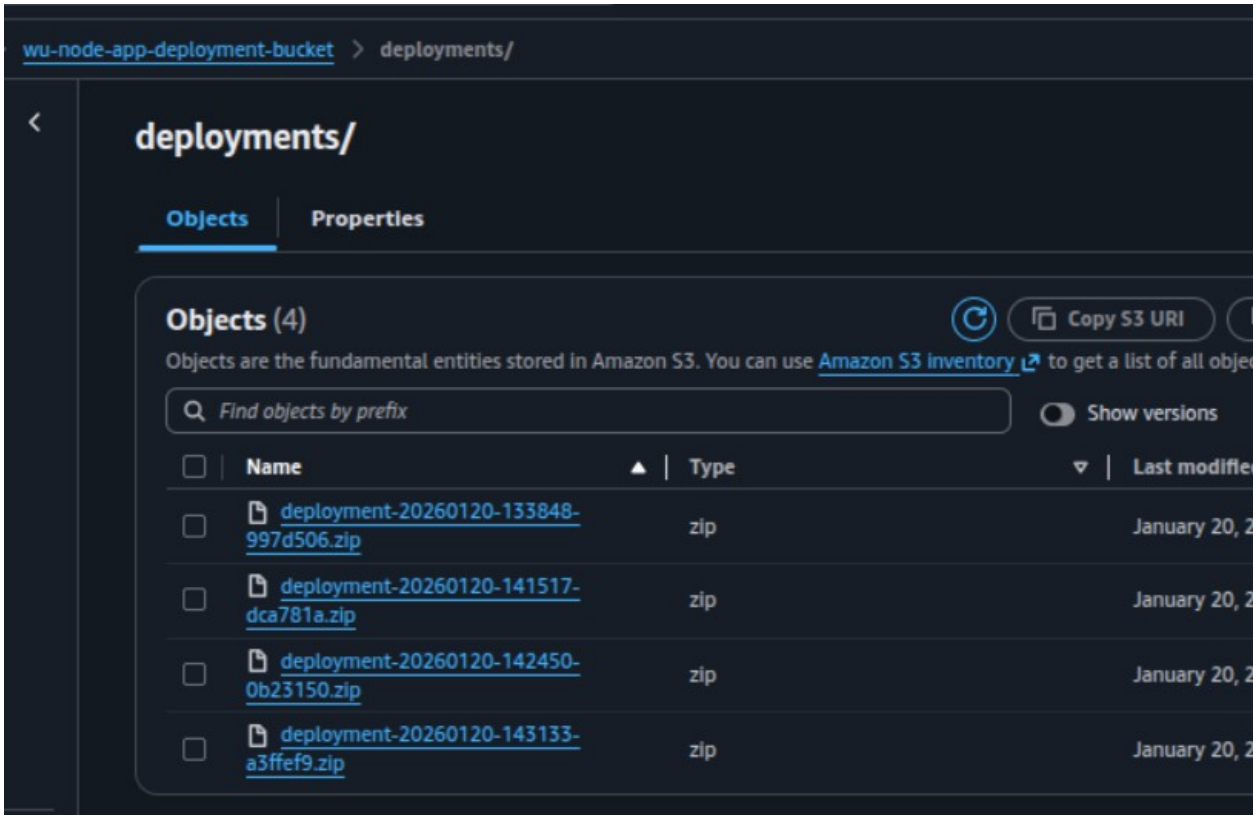
For updates, the same process repeats. CodeDeploy performs rolling deployment (OneAtATime), updating one instance while others serve traffic. If validation fails, CodeDeploy automatically rolls back to previous version.

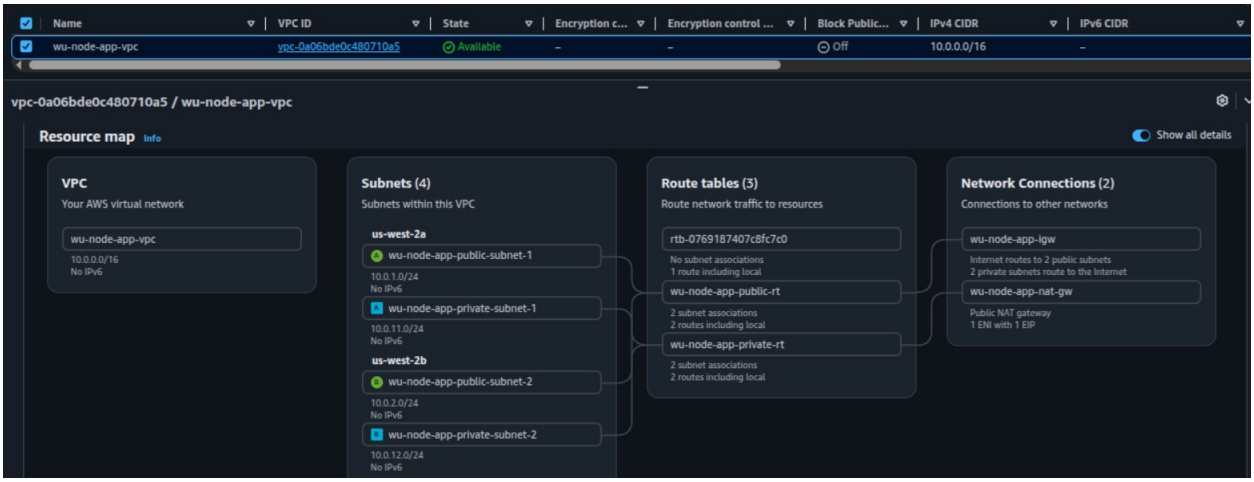
Testing and Validation

Post-Deployment Checks

1. Verify Infrastructure:

- terraform output - Check ALB DNS, role ARNs, bucket name.
- Check console if the artifacts are present in the s3 bucket.
- Verify the VPC and networking.



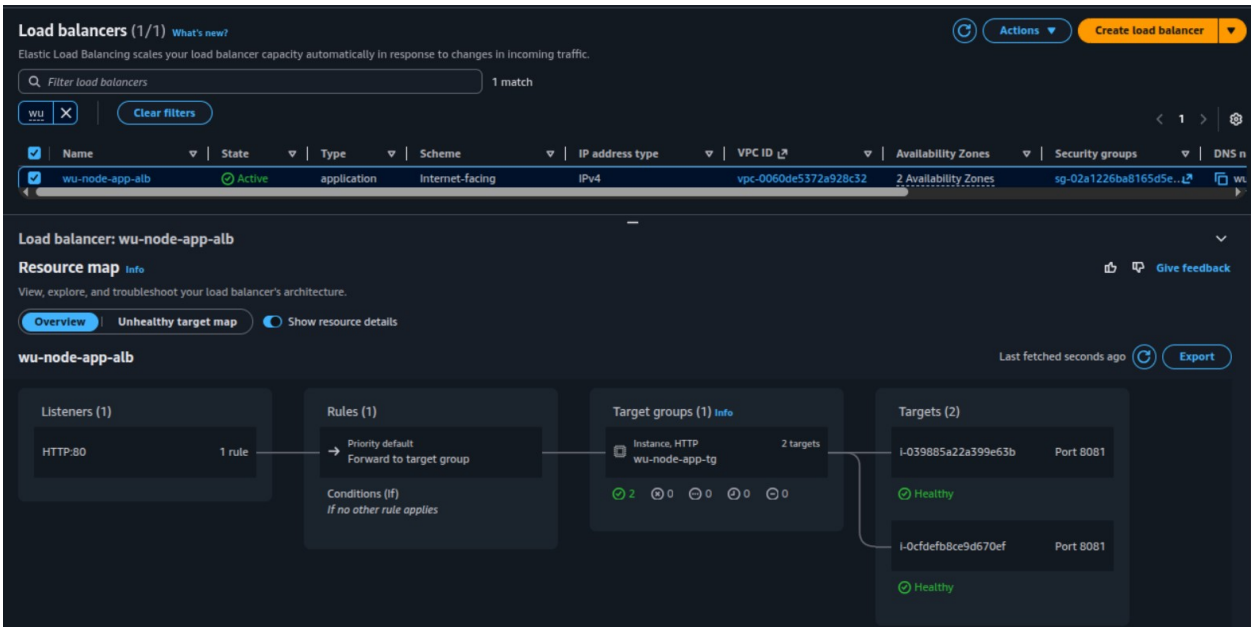


2. Check EC2 Instances:

- EC2 Console → Auto Scaling Groups → wu-node-app-asg
- Verify desired capacity reached and instances are InService

3. Verify Target Health:

- EC2 Console → Target Groups → wu-node-app-tg → Targets tab
- All instances should show 'healthy' status

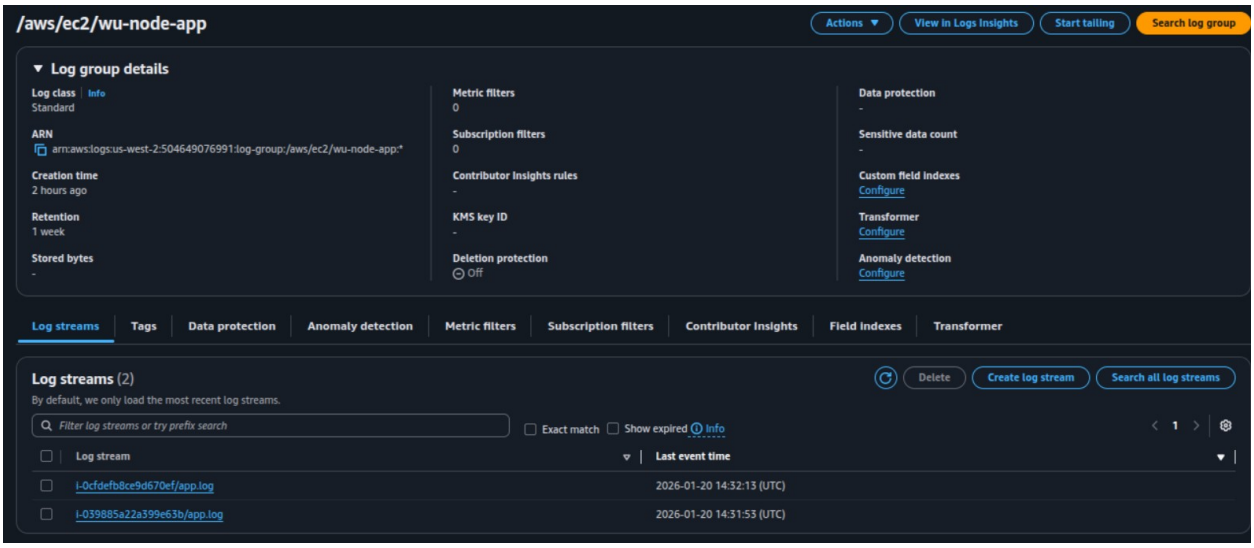


4. Test Application Access:

- Open browser to ALB DNS name
- Verify application responds correctly

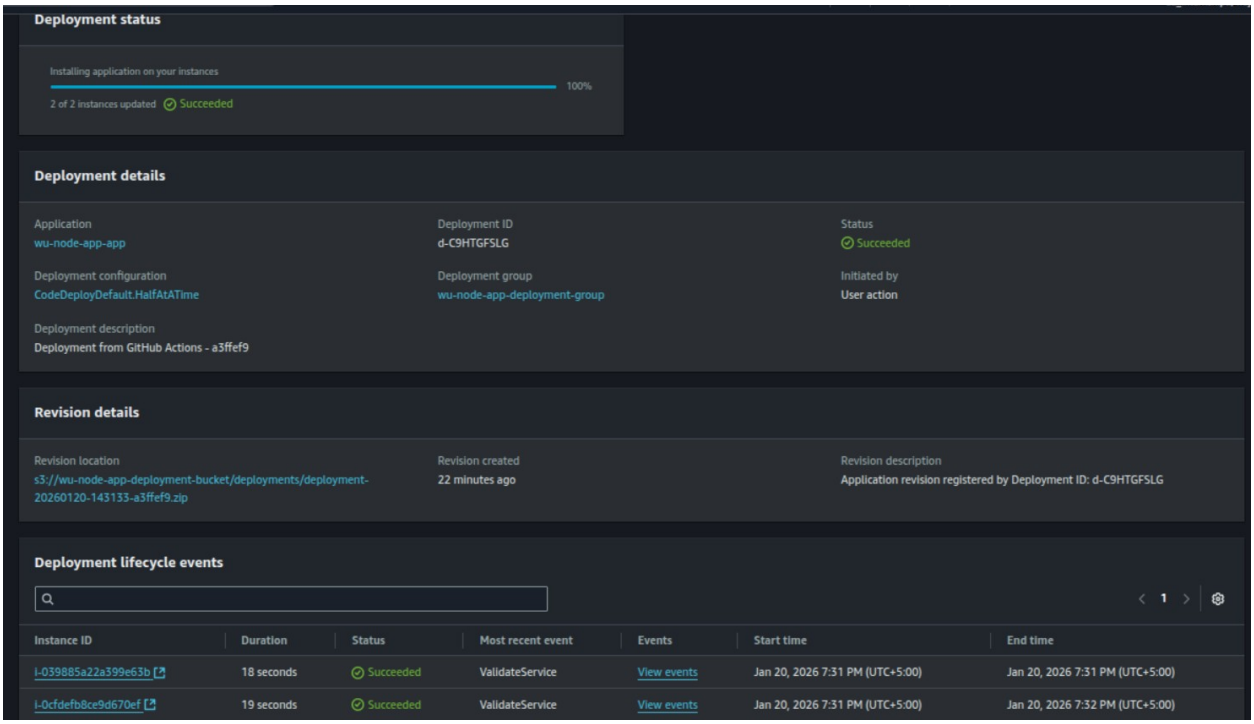
5. Check CloudWatch Logs:

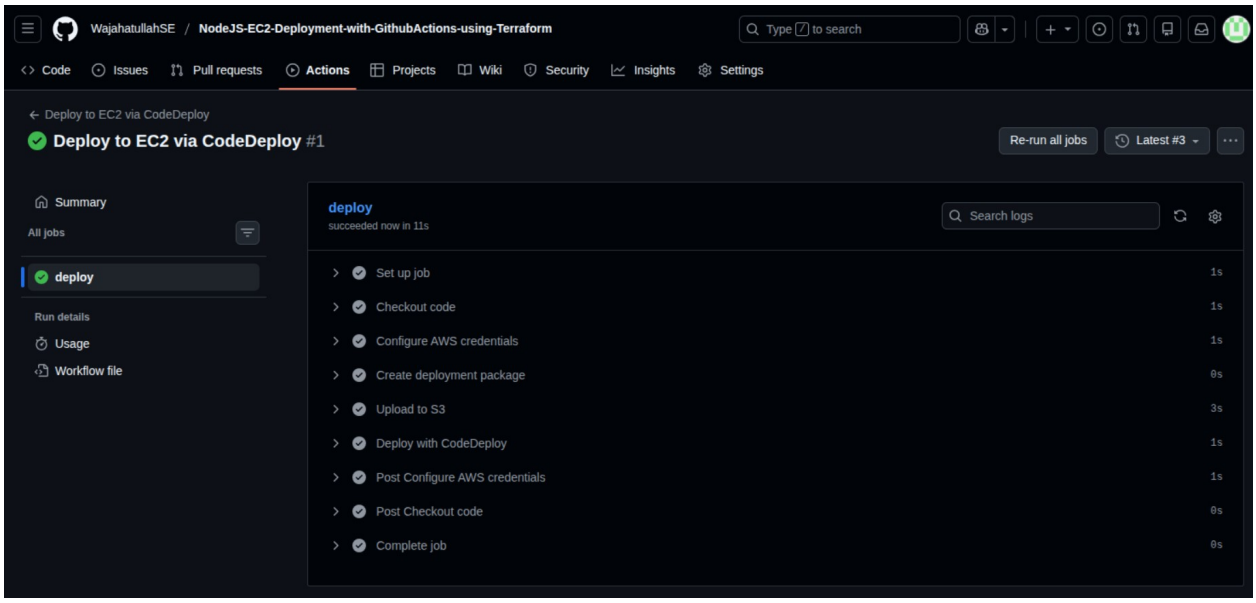
- CloudWatch Console → Log groups → /aws/ec2/wu-node-app
- Verify log streams exist and contain application logs



6. Validate CodeDeploy:

- CodeDeploy Console → Deployments
- Latest deployment status should be 'Succeeded'





7. Test Auto Scaling (Optional):

- Generate CPU load on instance
- Watch ASG scale up when CPU exceeds 50%

Troubleshooting Guide

Error: AppSpec File Not Found

Symptom:

CodeDeploy fails immediately with 'did not find an AppSpec file'

Cause:

appspec.yml is not at the root of the deployment ZIP. It's nested in codedeploy/ directory.

Fix:

Move appspec.yml to repository root:

mv codedeploy/appspec.yml ./appspec.yml

Update script paths in appspec.yml to: codedeploy/scripts/

Error: npm EACCES Permission Denied

Symptom:

AfterInstall hook fails with 'EACCES: permission denied' during npm install

Cause:

CodeDeploy extracts files as root but hook runs as ec2-user, causing permission mismatch.

Fix:

In install_deps.sh, ensure ownership before npm:

sudo chown -R ec2-user:ec2-user /opt/nodeapp

cd /opt/nodeapp

npm ci --omit=dev

Or run hook as root in appspec.yml: runas: root

Error: Too Many Failed Instances

Symptom:

CodeDeploy deployment fails with 'too many individual instances failed deployment'

Cause:

With only 1 instance and in-place deployment, there's downtime when instance is deregistered.

Fix:

Option 1: Increase ASG to min=1, desired=2 for overlap

Option 2: Use deployment config 'AllAtOnce' for single instance (testing only)

Error: IAM AccessDenied on CreateDeployment

Symptom:

GitHub Actions fails with 'not authorized to perform codedeploy:CreateDeployment'

Cause:

IAM policy scoped too tightly to application ARN instead of deployment group ARN.

Fix:

In modules/iam/main.tf, GitHub Actions policy:

Change Resource from specific ARN to '*' for CodeDeploy actions

Snippet

```
{
  "Effect": "Allow",
  "Action": ["codedeploy:CreateDeployment"],
  "Resource": "arn:aws:codedeploy:us-west-2:123456789012:deploymentgroup:app-name/deployment-group-name"
}
```

Or temp (broader)

```
{ "Effect":"Allow", "Action":"codedeploy:CreateDeployment", "Resource":"*" }
```

Error: User Data Deploying App (ASG Instability)

Symptom:

First instance works, subsequent deployments fail. ASG cannot stabilize at desired capacity.

Cause:

User data script clones repo and starts app, conflicting with CodeDeploy ownership. Creates race conditions and PM2 state conflicts.

Fix:

Remove ALL application deployment from user data:

• Delete git clone

• Delete npm install

• Delete pm2 start

User data should ONLY prepare host (install packages, create directories, start services)

Error: CodeDeploy Agent Not Running

Symptom:

Deployment never starts, instance shows no CodeDeploy activity

Cause:

CodeDeploy agent failed to install or start during user data execution

Fix:

SSH/SSM into instance and check:

sudo systemctl status codedeploy-agent

If not running: sudo systemctl start codedeploy-agent

Check logs: /var/log/aws/codedeploy-agent/codedeploy-agent.log

Error: Health Check Failing

Symptom:

Target group shows instances as unhealthy, ALB doesn't route traffic

Cause:

Application not listening on port 8080, crashed on startup, or health check path misconfigured

Fix:

1. Verify app listens on PORT=8081 (check environment variable)
2. Check PM2 status: pm2 list, pm2 logs nodeapp
3. Test locally: curl http://localhost:8081/
4. Verify security group allows ALB → EC2 on port 8080

Error: ASG Timeout During Terraform Apply

Symptom:

Terraform waits 10 minutes then fails with 'timeout waiting for capacity'

Cause:

Instances not passing health checks within grace period. Usually due to user data errors or application startup failures.

Fix:

1. Check user data logs: /var/log/cloud-init-output.log
2. Set health_check_type = 'EC2' temporarily during testing
3. Increase health_check_grace_period if needed
4. Start with desired_capacity = 0, scale manually after debugging