Node.js Application Deployment on AWS Elastic Beanstalk with CodePipeline using Terraform

Table of Contents

3

1. Overview 4

2. Objective 4

3. Prerequisites 4

4. Architecture Diagram 5

5. Modular Project Structure 6

6. Module Architecture and Configuration 6

6.1 VPC Module 6

Core Components: 7

6.2 Security Module 7

ALB Security Group: 7

Instance Security Group: 7

6.3 Elastic Beanstalk Module 8

Application Configuration: 8

Load Balancer Configuration: 8

Health Check Configuration: 8

Auto Scaling Configuration: 8

Deployment Strategy: 8

Logging Configuration: 8

6.4 CodeBuild Module 9

Build Environment: 9

Build Specification (Buildspec): 9
6.5 CodePipeline Module 9
Pipeline Stages: 9
Artifact Storage: 10
7. IAM Roles and Required Policies 10
7.1 Elastic Beanstalk Service Role 10
7.2 EC2 Instance Profile 11
7.3 CodeBuild Service Role 11
7.4 CodePipeline Service Role 12
7.5 IAM Security Best Practices 12
8. Deployment Steps 13
8.1 Pre-Deployment Verification 13
8.2 Terraform Initialization 13
8.3 Infrastructure Planning 13
8.4 Infrastructure Provisioning 13
8.5 Output Verification 14
9. Testing and Validation 14
9.1 Initial Deployment Validation 14
9.2 Elastic Beanstalk Health Verification 15
9.3 CodePipeline Functionality Test 16
9.4 Post-Deployment Application Verification 16
9.5 Infrastructure Health Monitoring 17
9.6 Load Balancer Validation 18
9.7 Auto Scaling Validation 18
10. Troubleshooting Guide 19
10.1 Application Files Not Found 19

10.2 CodeBuild Artifact Contains Malformed Procfile 19

10.3 Version Label Conflict 19

1. Overview

This document provides comprehensive technical specifications for deploying a Node.js application on AWS Elastic Beanstalk with automated continuous integration and continuous deployment (CI/CD) using AWS CodePipeline. The infrastructure provisioning is fully automated through Terraform using a modular architecture approach, ensuring scalability, maintainability, and infrastructure-as-code best practices.

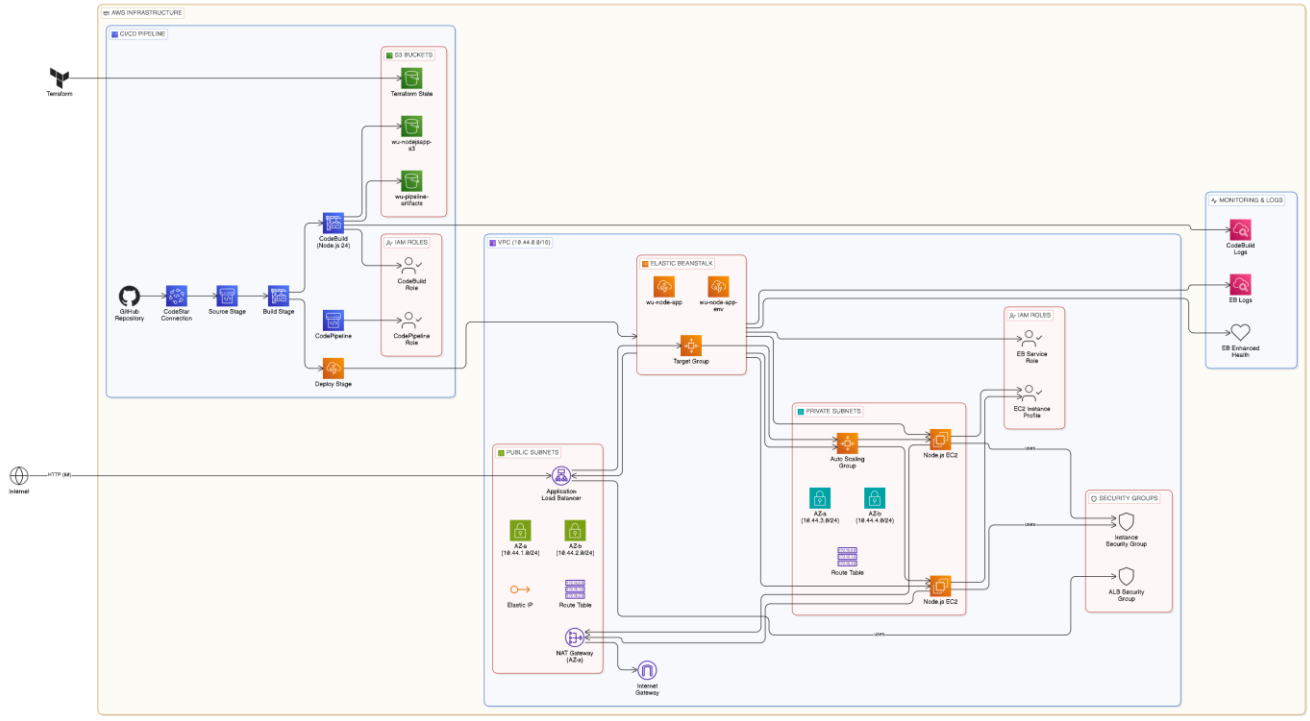
2. Objective

Establish a production-ready, automated deployment pipeline for Node.js applications leveraging AWS managed services. The solution implements a load-balanced Elastic Beanstalk environment within a custom VPC architecture, with automatic deployment triggers from GitHub repository commits. The infrastructure supports high availability across multiple availability zones with auto-scaling capabilities and rolling update deployment strategies.

3. Prerequisites

- NodeJS application with a GitHub Repository
- AWS account with necessary permissions
- AWS CLI configured with appropriate credentials
- S3 bucket containing initial application version (v1.zip)
- CodeStar Connections established with GitHub

4. Architecture Diagram


AWS INFRASTRUCTURE
CI/CD PIPELINE
S3 BUCKETS
Terraform
Terraform State
wu-nodejsapp-s3
wu-pipeline-artifacts
CodeBuild (Node.js 24)
IAM ROLES
CodeBuild Role
GitHub Repository
CodeStar Connection
Source Stage
Build Stage
CodePipeline
CodePipeline Role
Deploy Stage
MONITORING & LOGS
CodeBuild Logs
EB Logs
EB Enhanced Health
VPC (10.44.8.0/16)
ELASTIC BEANSTALK
wu-node-app
wu-node-app-env
Target Group
IAM ROLES
EB Service Role
EC2 Instance Profile
PRIVATE SUBNETS
Auto Scaling Group
Node.js EC2
AZ-a (10.44.3.0/24)
AZ-b (10.44.4.0/24)
Route Table
Node.js EC2
SECURITY GROUPS
Instance Security Group
ALB Security Group
PUBLIC SUBNETS
Internet
HTTP (80)
Application Load Balancer
AZ-a (10.44.1.0/24)
AZ-b (10.44.2.0/24)
Elastic IP
Route Table
NAT Gateway (AZ-a)
Internet Gateway

5. Modular Project Structure

The Terraform infrastructure follows a modular architecture pattern, separating concerns into distinct, reusable components. This approach enhances maintainability, enables component reusability, and simplifies infrastructure management.

Project directory structure:

```
terraform-eb-pipeline/
├── main.tf                    # Root module orchestration
├── variables.tf               # Variable definitions
├── terraform.tfvars           # Actual values and configuration
├── outputs.tf                 # Infrastructure outputs
├── provider.tf                # AWS provider configuration
├── backend.tf                 # Terraform state backend
└── modules/
    ├── vpc/                   # VPC and networking
    │   ├── main.tf
    │   ├── variables.tf
    │   └── outputs.tf
    ├── security/              # Security groups
    │   ├── main.tf
    │   ├── variables.tf
    │   └── outputs.tf
    ├── elb/                   # Elastic Beanstalk
    │   ├── main.tf
    │   ├── variables.tf
    │   └── outputs.tf
    ├── codebuild/             # CodeBuild project
    │   ├── main.tf
    │   ├── variables.tf
    │   └── outputs.tf
    └── codepipeline/          # CodePipeline and artifacts
        ├── main.tf
        ├── variables.tf
        └── outputs.tf
```

6. Module Architecture and Configuration

6.1 VPC Module

The VPC module establishes the foundational networking infrastructure, implementing a multi-availability zone architecture for high availability and fault tolerance.

Core Components:

- VPC with custom CIDR block (10.44.0.0/16) enabling DNS hostnames and DNS support
- Two availability zones (us-west-2a, us-west-2b) for redundancy
- Public subnets (10.44.1.0/24, 10.44.2.0/24) with automatic public IP assignment
- Private subnets (10.44.3.0/24, 10.44.4.0/24) for application tier isolation
- Internet Gateway attached to VPC for public internet connectivity
- Single NAT Gateway deployed in first public subnet with Elastic IP
- Public route table directing internet-bound traffic (0.0.0.0/0) to Internet Gateway
- Private route table directing internet-bound traffic to NAT Gateway for outbound connectivity

Network Flow Logic: Public subnets host the Application Load Balancer for incoming HTTP traffic. Private subnets contain Elastic Beanstalk EC2 instances, which access the internet through the NAT Gateway for package downloads and external API calls while remaining inaccessible from the public internet.

6.2 Security Module

Implements defense-in-depth security architecture through security groups, controlling traffic flow between infrastructure components based on the principle of least privilege.

ALB Security Group:

- Ingress: HTTP port 80 from 0.0.0.0/0 (public internet access)
- Egress: All traffic to 0.0.0.0/0 (enables ALB to communicate with backend instances)
- Purpose: Serves as the entry point for all external HTTP traffic

Instance Security Group:

- Ingress: All TCP ports from ALB Security Group (accepts traffic only from load balancer)
- Egress: All traffic to 0.0.0.0/0 (enables NAT Gateway routing for outbound internet)
- Purpose: Protects application instances while allowing necessary connectivity

Security Architecture: The security group configuration implements a zero-trust network model where instances accept connections exclusively from the ALB, preventing direct internet exposure. Outbound internet access through the NAT Gateway enables npm package installation and external service communication during deployment and runtime.

6.3 Elastic Beanstalk Module

Manages the application runtime environment with comprehensive configuration covering compute resources, load balancing, auto-scaling, health monitoring, and deployment strategies.

Application Configuration:

- Platform: 64bit Amazon Linux 2023 v6.7.0 running Node.js 24
- Initial deployment from S3 bucket (wu-nodejsapp-s3) using v1.zip
- Application port: 8081 injected via PORT environment variable
- Enhanced health reporting for detailed instance and environment metrics

Load Balancer Configuration:

- Type: Application Load Balancer (Layer 7)
- Placement: Public subnets across both availability zones
- Protocol: HTTP with default listener on port 80
- Target routing: HTTP traffic to instances on port 8081
- Idle timeout: 600 seconds for long-running requests
- Deregistration delay: 20 seconds for graceful connection draining

Health Check Configuration:

- Path: / (root endpoint)
- Interval: 30 seconds between checks
- Timeout: 5 seconds per check
- Healthy threshold: 3 consecutive successful checks
- Unhealthy threshold: 5 consecutive failed checks
- Success criteria: HTTP 200 status code

Auto Scaling Configuration:

- Instance type: t3.small (2 vCPU, 2 GB memory)
- Minimum instances: 2 (ensures high availability)
- Maximum instances: 4 (controls cost while maintaining performance)

- Placement: Private subnets for enhanced security
- IAM instance profile: aws-elasticbeanstalk-ec2-role-wu

Deployment Strategy:

- Policy: Rolling deployment with health-based updates
- Batch size: 50% of instances updated simultaneously
- Minimum instances in service: 1 during updates
- Update type: Health-based rolling ensures zero-downtime deployments

Logging Configuration:

- CloudWatch Logs streaming: Enabled for real-time log access
- Retention period: 7 days
- Deletion on termination: Disabled (logs persist after environment deletion)

Environment Architecture: The load-balanced environment maintains high availability through multi-AZ deployment. The ALB distributes incoming traffic across healthy instances while health checks continuously monitor application responsiveness. Rolling deployments ensure zero-downtime updates by maintaining minimum instance count during version transitions.

6.4 CodeBuild Module

Automates the application build process, compiling source code, installing dependencies, and packaging artifacts for deployment to Elastic Beanstalk.

Build Environment:

- Compute type: BUILD_GENERAL1_SMALL (3 GB memory, 2 vCPUs)
- Container image: aws/codebuild/standard:7.0 (Ubuntu-based)
- Runtime: Node.js 24 for consistency with Elastic Beanstalk platform
- Build timeout: 15 minutes maximum execution time

Build Specification (Buildspec):

- Install phase: Configures Node.js 24 runtime version
- Pre-build phase: Executes npm install --production for production dependencies
- Build phase: Logs build completion timestamp
- Post-build phase: Prepares deployment artifact
- Artifacts: Packages all files (**/*) maintaining directory structure

- Output: Uploads build artifact to pipeline S3 bucket for deployment stage

Build Process Flow: CodePipeline triggers the build upon detecting GitHub repository changes. CodeBuild provisions a fresh container, clones the source code, installs Node.js dependencies, and packages the complete application including node_modules directory. The artifact maintains the original directory structure, preserving .ebextensions configurations and application code hierarchy. CloudWatch Logs capture all build output for debugging and audit purposes.

6.5 CodePipeline Module

Orchestrates the complete CI/CD workflow with three stages: source retrieval, build execution, and deployment, providing automated delivery from code commit to production.

Pipeline Stages:

Stage 1: Source

- Provider: AWS CodeStar Source Connection
- Repository: WajahatullahSE/NodeJS-Application-Deployment-on-AWS-Beanstalk-with-CodePipeline-using-Terraform
- Branch: main
- Trigger: Automatic on push events (webhook-based)
- Output: source_output artifact containing repository snapshot

Stage 2: Build

- Provider: AWS CodeBuild
- Input: source_output from previous stage
- Process: Executes buildspec phases (install, pre_build, build, post_build)
- Output: build_output artifact with compiled application and dependencies

Stage 3: Deploy

- Provider: AWS Elastic Beanstalk
- Input: build_output from build stage
- Target: wu-node-app-env environment
- Process: Creates new application version and performs rolling deployment
- Result: Updated application running on Elastic Beanstalk instances

Artifact Storage:

- S3 bucket: wu-pipeline-artifacts
- Versioning: Enabled for artifact history and rollback capability
- Purpose: Stores intermediate artifacts between pipeline stages
- Lifecycle: Managed by CodePipeline with automatic cleanup

Pipeline Execution Flow: Git push to main branch triggers CodeStar connection webhook. Pipeline pulls latest code from GitHub, passes it to CodeBuild for dependency installation and packaging, then deploys the artifact to Elastic Beanstalk. The rolling deployment strategy updates instances in batches, maintaining application availability throughout the update process. Each stage transition stores artifacts in S3, enabling stage retries and deployment auditing.

7. IAM Roles and Required Policies

The infrastructure requires four pre-configured IAM roles with specific managed and custom policies for service authorization and cross-service communication.

7.1 Elastic Beanstalk Service Role

Role Name: aws-elasticbeanstalk-service-role-wu

Purpose: Enables Elastic Beanstalk to manage AWS resources on behalf of the environment (load balancers, auto-scaling groups, CloudWatch logs)

Required Managed Policies:

- AWSElasticBeanstalkEnhancedHealth - Enhanced health reporting and monitoring
- AWSElasticBeanstalkManagedUpdatesCustomerRolePolicy - Managed platform updates

Trust Relationship: elasticbeanstalk.amazonaws.com

7.2 EC2 Instance Profile

Profile Name: aws-elasticbeanstalk-ec2-role-wu

Purpose: Grants EC2 instances permissions to interact with S3, CloudWatch, and other AWS services during application runtime

Required Managed Policies:

- AWSElasticBeanstalkWebTier - Web server tier permissions
- AWSElasticBeanstalkWorkerTier - Background processing capabilities
- AWSElasticBeanstalkMulticontainerDocker - Container management (optional)

Custom Policy Requirements:

- S3 read access to application version bucket (wu-nodejsapp-s3)
- CloudWatch Logs write permissions for log streaming
- SSM Parameter Store read access if using environment secrets

Trust Relationship: ec2.amazonaws.com

7.3 CodeBuild Service Role

Role ARN: arn:aws:iam::504649076991:role/CodeBuild-ECSEC2-ServiceRole-wu

Purpose: Authorizes CodeBuild to pull source code, execute builds, and upload artifacts to S3

Required Permissions:

- S3 full access to pipeline artifacts bucket (wu-pipeline-artifacts)
- CloudWatch Logs create and write permissions for build logs
- CodeBuild project management permissions
- VPC network interface creation if using VPC-connected builds

Custom Policy Example:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": ["s3:GetObject", "s3:PutObject"],
    "Resource": "arn:aws:s3:::wu-pipeline-artifacts/*"
  }, {
    "Effect": "Allow",
    "Action": ["logs:CreateLogGroup", "logs:CreateLogStream", "logs:PutLogEvents"],
    "Resource": "arn:aws:logs:*:*:*"
  }]
}
```

Trust Relationship: codebuild.amazonaws.com

7.4 CodePipeline Service Role

Role ARN: arn:aws:iam::504649076991:role/service-role/AWSCodePipelineServiceRole-us-west-2-wu-codepipeline

Purpose: Orchestrates pipeline execution across source, build, and deploy stages with appropriate service permissions

Required Permissions:

- S3 full access to artifacts bucket for stage transitions
- CodeStar Connections UseConnection for GitHub integration
- CodeBuild StartBuild and BatchGetBuilds for build stage execution
- Elastic Beanstalk CreateApplicationVersion and UpdateEnvironment for deployment
- IAM PassRole for passing execution roles to services

Custom Policy Example:

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": "elasticbeanstalk:*",
    "Resource": "*"
  }, {
    "Effect": "Allow",
    "Action": "codebuild:*",
    "Resource": "arn:aws:codebuild:us-west-2:504649076991:project/wu-nodejs-build"
  }, {
    "Effect": "Allow",
    "Action": "codestar-connections:UseConnection",
    "Resource": "arn:aws:codeconnections:us-west-2:504649076991:connection/*"
  }]
}
```

Trust Relationship: codepipeline.amazonaws.com

7.5 IAM Security Best Practices

- Apply least privilege principle - grant only necessary permissions
- Use managed policies where available for AWS service integrations
- Implement resource-level permissions using ARN constraints
- Enable CloudTrail logging for IAM role assumption auditing
- Regularly review and rotate IAM credentials
- Use IAM policy simulator to validate permissions before deployment

8. Deployment Steps

8.1 Pre-Deployment Verification

- Verify AWS CLI configuration: aws sts get-caller-identity
- Confirm S3 bucket exists with v1.zip: aws s3 ls s3://wu-nodejsapp-s3/
- Validate GitHub connection status in AWS CodeStar Connections console
- Ensure all IAM roles exist with correct policies attached
- Review terraform.tfvars for accurate values (region, ARNs, repository ID)

8.2 Terraform Initialization

Execute in project root directory:

```
terraform init
```

This command initializes the working directory, downloads provider plugins (AWS), and configures the backend for state storage. Verify successful initialization with "Terraform has been successfully initialized!" message.

8.3 Infrastructure Planning

Generate and review execution plan:

```
terraform plan -out=tfplan
```

Review the plan output carefully. Verify:

- Resource count matches expectations (~40-50 resources)
- VPC CIDR and subnet allocations are correct
- IAM role ARNs reference existing roles

- Elastic Beanstalk platform version is accurate
- No unexpected resource deletions or replacements

8.4 Infrastructure Provisioning

Apply the Terraform configuration:

```
terraform apply tfplan
```

Deployment timeline:

- VPC and networking resources: 2-3 minutes
- Security groups: 1 minute
- Elastic Beanstalk environment: 10-15 minutes (longest operation)
- CodeBuild project: 1 minute
- CodePipeline: 2-3 minutes
- Total estimated time: 15-20 minutes

Note: Elastic Beanstalk environment creation includes EC2 instance provisioning, load balancer configuration, and initial application deployment from S3, accounting for the extended creation time.

8.5 Output Verification

Retrieve deployment outputs:

```
terraform output
```

Record the following outputs:

- eb_environment_url: Application endpoint URL
- codepipeline_name: Pipeline identifier for AWS console navigation
- artifacts_bucket: S3 bucket name for artifact storage
- vpc_id: VPC identifier for network troubleshooting

9. Testing and Validation

9.1 Initial Deployment Validation

Verify the initial application deployment:

```
curl http://<eb_environment_url>
```

Expected response:

```
<h1>Node.js Sample Application, Version #01</h1>
<p>Deployed on Elastic Beanstalk with CodePipeline</p>
<p>Welcome to Application!</p>
```

HTTP status code should be 200. If environment variables are not displayed, .ebextensions configuration has not yet been applied (will appear after first pipeline deployment).

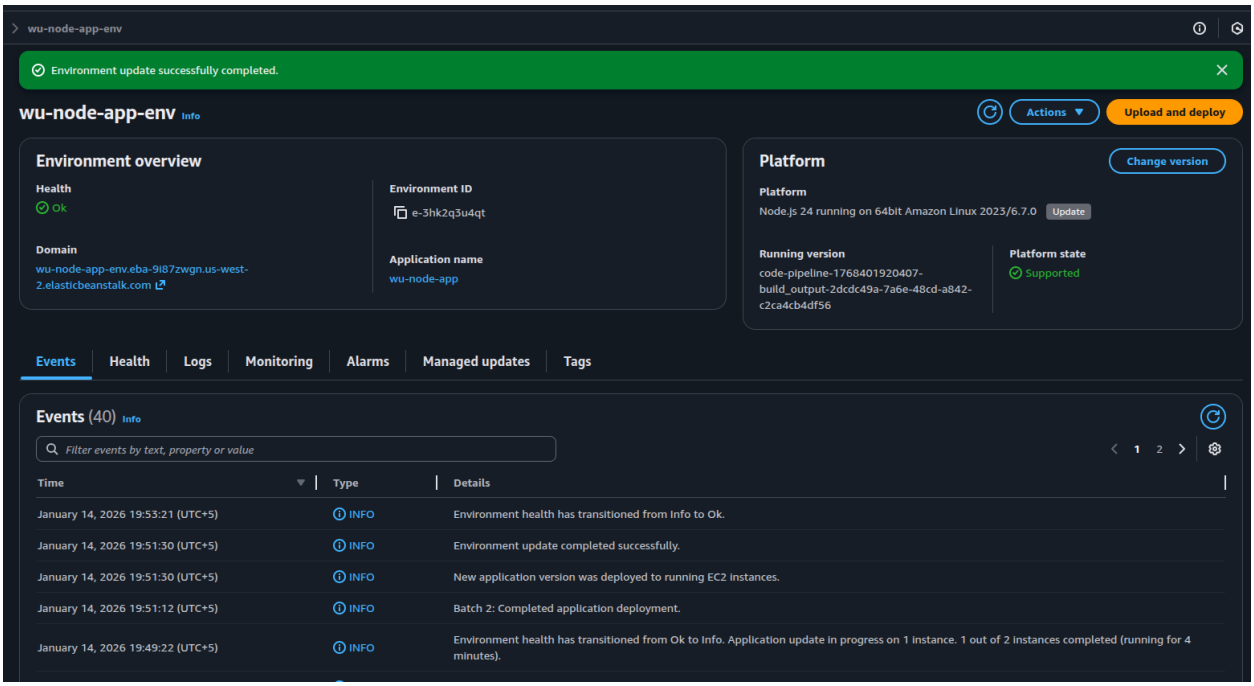
9.2 Elastic Beanstalk Health Verification

Access AWS Console → Elastic Beanstalk → wu-node-app-env

Verify environment health indicators:

- Overall health status: Green (Ok)
- Running instances: 2/2 showing as healthy
- Recent events: No error messages or warnings

- Enhanced health overview: All metrics within normal ranges
- Load balancer status: Active with healthy target instances



9.3 CodePipeline Functionality Test

Trigger automated deployment by pushing code changes:

Step 1: Modify application code (e.g., update version number in index.js to #02)

Step 2: Commit and push to GitHub:

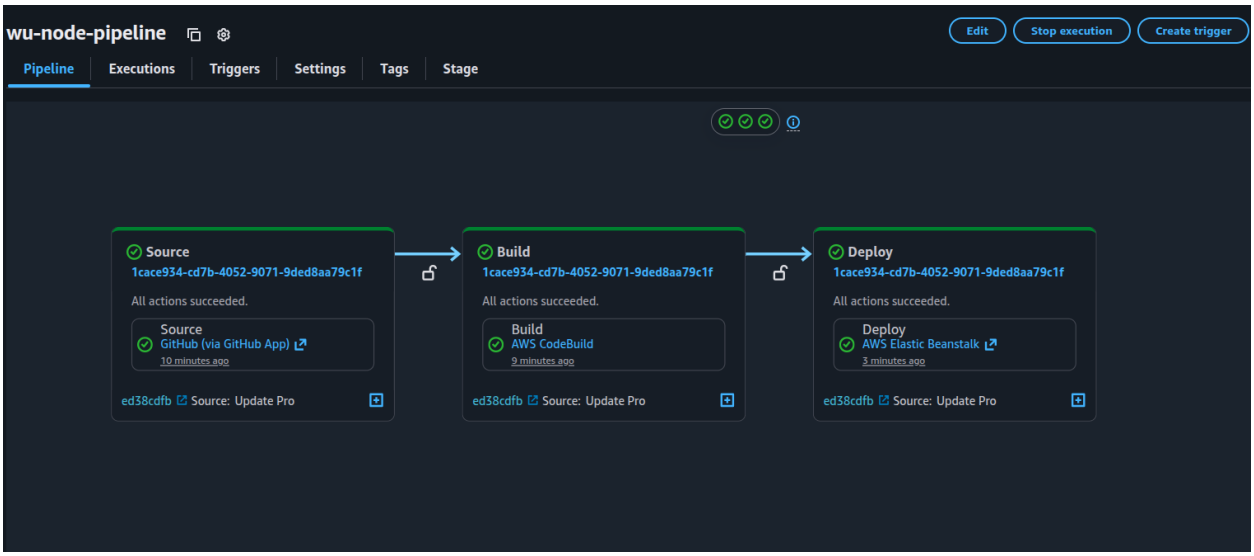
```
git add .
git commit -m "Update application version to #02"
git push origin main
```

Step 3: Monitor pipeline execution in AWS Console → CodePipeline → wu-node-pipeline

Observe stage progression:

- Source stage: Completes within 10-15 seconds after push
- Build stage: Executes npm install, typically 2-3 minutes
- Deploy stage: Performs rolling deployment, 5-7 minutes

- Overall pipeline execution: 7-10 minutes total



9.4 Post-Deployment Application Verification

Validate the updated application:

```
curl http://<eb_environment_url>
```

Expected changes:

- Version number updated to #02
- Environment variables (NODE_ENV, CUSTOM_VAR) populated from .ebextensions
- PORT environment variable shows 8081

Node.js Sample Application, Version 11

Deployed on Elastic Beanstalk with CodePipeline

Welcome to Application!

Environment Variables:

NODE_ENV: production

CUSTOM_VAR: hello_world

PORT: 8081

9.5 Infrastructure Health Monitoring

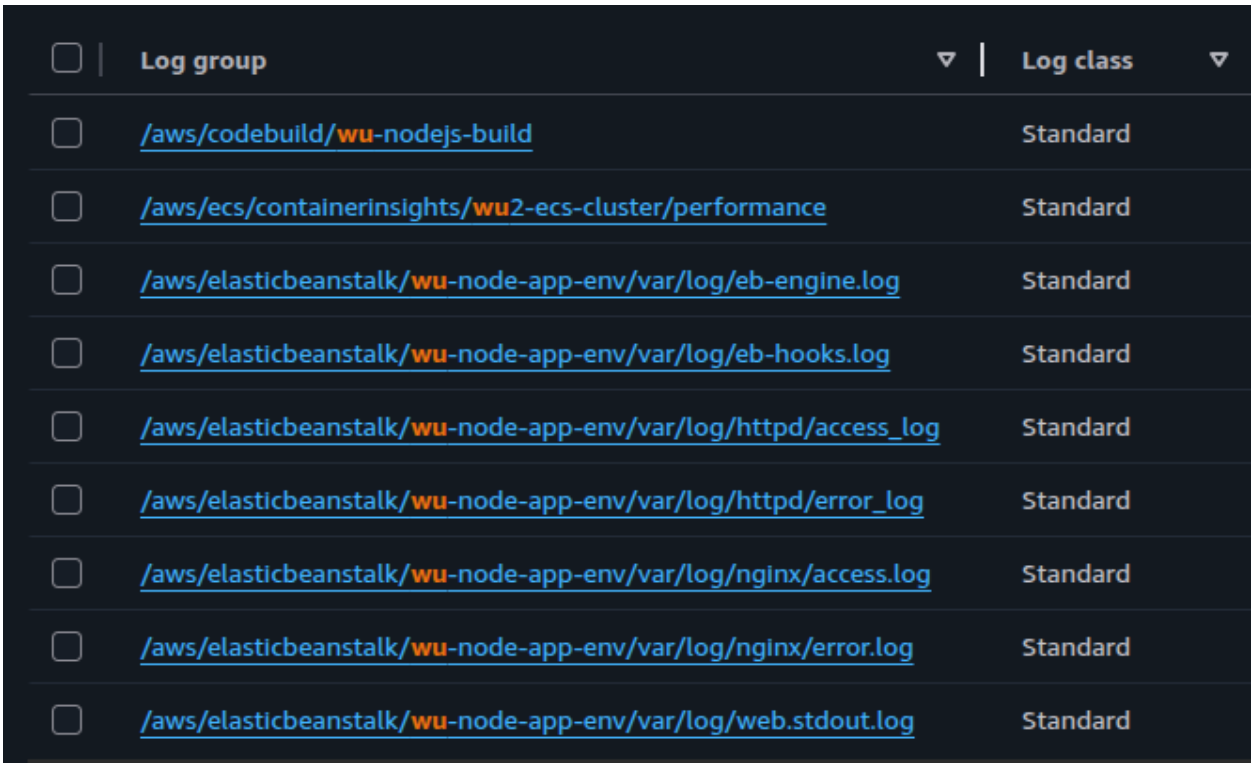
Verify CloudWatch metrics and logs:

Elastic Beanstalk Metrics (Console → CloudWatch → Metrics → Elastic Beanstalk):

- EnvironmentHealth: Monitoring overall environment status
- InstanceHealth: Per-instance health scores
- ApplicationRequests2xx: Successful HTTP response count
- ApplicationLatencyP99: 99th percentile response times

Log Groups (Console → CloudWatch → Logs):

- /aws/elasticbeanstalk/wu-node-app-env: Application logs
- /aws/codebuild/wu-nodejs-build: Build execution logs
- Retention period: 7 days as configured



Log group	Log class
/aws/codebuild/wu-nodejs-build	Standard
/aws/ecs/containerinsights/wu2-ecs-cluster/performance	Standard
/aws/elasticbeanstalk/wu-node-app-env/var/log/eb-engine.log	Standard
/aws/elasticbeanstalk/wu-node-app-env/var/log/eb-hooks.log	Standard
/aws/elasticbeanstalk/wu-node-app-env/var/log/httpd/access_log	Standard
/aws/elasticbeanstalk/wu-node-app-env/var/log/httpd/error_log	Standard
/aws/elasticbeanstalk/wu-node-app-env/var/log/nginx/access.log	Standard
/aws/elasticbeanstalk/wu-node-app-env/var/log/nginx/error.log	Standard
/aws/elasticbeanstalk/wu-node-app-env/var/log/web.stdout.log	Standard

9.6 Load Balancer Validation

Verify ALB configuration and target health:

Navigate to AWS Console → EC2 → Load Balancers

- Identify ALB created by Elastic Beanstalk (prefixed with awseb-)
- Target group health: All registered targets showing healthy status
- Health check configuration: Path /, port 8081, HTTP protocol
- Connection draining: 20 second deregistration delay configured
- Access logs: Optional - can be enabled for traffic analysis

9.7 Auto Scaling Validation

Test auto-scaling behavior under load:

Optional: Generate load using Apache Bench or similar tool:

```
ab -n 10000 -c 100 http://<eb_environment_url>/
```

Monitor AWS Console → EC2 → Auto Scaling Groups:

- Current capacity should remain at 2 instances under normal load
- Scaling policies trigger based on CPU or request count thresholds
- Maximum capacity capped at 4 instances as configured
- Cooldown periods prevent excessive scaling oscillations

10. Troubleshooting Guide

Common deployment issues and their resolutions for Node.js application on AWS Elastic Beanstalk with CodePipeline.

10.1 Application Files Not Found

Issue: Elastic Beanstalk error: "Instance deployment failed to generate a Procfile for Node.js. Provide one of these files: package.json, server.js, or app.js"

Cause: Application zip file contains an extra parent directory layer. When extracting, files are nested under a folder instead of being at the root level.

Fix: Create zip from within the application directory, not from outside:

```
cd my-app
zip -r ../v1.zip .
```

Verification: Execute unzip -l v1.zip to confirm package.json and index.js appear at root level without parent folder.

10.2 CodeBuild Artifact Contains Malformed Procfile

Issue: Procfile is correct in repository but deployment from CodePipeline fails with parse error. Inspecting build artifact reveals trailing whitespace in Procfile.

Cause: Text editors or Git operations introduce trailing spaces or line endings during build process.

Fix: Add Procfile sanitization step to buildspec.yml in build phase:

```
build:
  commands:
    - echo "Build completed on $(date)"
    - echo "Trimming Procfile..."
    - sed -i 's/[[:space:]]*$//' Procfile
```

This removes all trailing whitespace and blank lines from Procfile before artifact creation and the pipeline does not fail.

10.3 Version Label Conflict

Issue: Deployment error: "Incorrect application version code-pipeline-xxx. Expected version wu-node-app-v1"

Cause: Elastic Beanstalk environment is pinned to initial version label in Terraform configuration, preventing CodePipeline from deploying new versions.

Fix: Remove version_label attribute from aws_elastic_beanstalk_environment resource in Terraform, allowing dynamic version updates from pipeline.

```
# Elastic Beanstalk Environment
resource "aws_elastic_beanstalk_environment" "this" {
  name                = var.environment_name
  application         = aws_elastic_beanstalk_application.this.name
  solution_stack_name = var.solution_stack
  #version_label       = aws_elastic_beanstalk_application_version.initial.name
  version_label = ""
```