CNIT 371-002 Fall 2025 – Cloud Computing Design, Implementation & Security

Instructor Name: Holly Yuan - Student: Thanh An Vu

Instructor Name: Holly Yuan - Student: Thanh An Vu

- 1. Title Page:
- 2. Executive Summary:
- 3. Architecture and Design:
  - 3.1 Architecture Diagram:
  - 3.2 Component Explanations:
- 4.1 Prerequisites and Initial Setup
  - 4.1.1 Required Software Installation
  - 4.1.2 AWS CLI Configuration
  - 4.1.3 GitHub Connection Setup
  - 4.1.4 CDK Bootstrap
- 4.2 VPC and Network Configuration
  - 4.2.1 VPC Stack Overview
  - 4.2.2 VPC Stack Code
  - 4.2.3 Deploy VPC Stack
- 4.3 IAM Roles and Security Configuration
  - 4.3.1 IAM Roles Overview
  - 4.3.2 Key IAM Policies
  - 4.3.3 Deploy IAM Stack
- 4.4 Storage Configuration (S3 & DynamoDB)
  - 4.4.1 Storage Resources Overview
  - 4.4.2 S3 Security Configuration
  - 4.4.3 Deploy Storage Stack
- 4.5 Lambda Function Setup - For Demo
  - 4.5.1 Lambda Function Purpose
  - 4.5.2 Lambda Function Code
  - 4.5.3 Lambda Configuration
  - 4.5.4 Deploy Lambda Stack
- 4.6 Elastic Beanstalk Configuration
  - 4.6.1 Environment Architecture

4.6.2 Environment Configuration
4.6.3 Flask Application Configuration Files
4.6.4 Deploy Beanstalk Stack
4.7 CodePipeline and CodeBuild Setup
4.7.1 Pipeline Stages Overview
4.7.2 CodeBuild Configuration
4.7.3 Pipeline Source Configuration
4.7.4 Deploy Pipeline Stack
4.8 Monitoring and Alerting Configuration
4.8.1 CloudWatch Dashboard
4.8.2 CloudWatch Alarms
4.8.3 SNS Notification Setup
4.8.4 Deploy Monitoring Stack
4.9 Testing and Validation
4.9.1 Deploy All Stacks
4.9.2 Validate Infrastructure
4.9.3 Trigger Pipeline Test
4.9.4 Monitor Pipeline Execution
4.9.5 Run Local Tests
4.9.6 Verify Deployed Application
5. Proof of Concept (POC) Validation
Workflow 1: Code Deployment Pipeline (Progressive Deployment)
Workflow 2: Staging Environment Testing
Workflow 3: Production Deployment Approval
Workflow 4: Rollback Strategy
Option A: Elastic Beanstalk Console Rollback (Fast)
Option B: Git Revert + New Pipeline (Recommended)
Workflow 5: Infrastructure Updates
Workflow 6: Application Health Monitoring
Workflow 7: Blue/Green Alternative (Educational - Not Active)
Workflow 8: New Feature Development
Workflow 9: Troubleshooting Failed Deployments
6. Outputs, Logs & Screenshots
6.1 Pipeline Execution Logs
6.2 CodeBuild Logs
6.3 Elastic Beanstalk Deployment Logs
6.4 Application Health Check
6.5 CloudWatch Logs
6.6 CDK Deployment Output
6.7. Manual Approval Action
6.8. Production Deployment Success

6. Challenges Encountered
7. Conclusion and Future Work
    7.1 Conclusion
    7.2 Future Work

1. Title Page:

CNIT 371-002 Fall 2025 – Cloud Computing Design, Implementation & Security

Instructor Name: Holly Yuan

Submission Date: 12/19/2025

DevOps Power Pipeline — Fully Automated CI/CD For Flask Application

2. Executive Summary:

This project implements a fully automated CI/CD pipeline for deploying a Python Flask application on AWS using Infrastructure as Code principles with AWS Cloud Development Kit (CDK). The infrastructure migration from manual console-based configuration to code-defined resources addressed critical challenges in reproducibility, configuration management, and deployment reliability. The manual approach created significant operational challenges: configuration replication required repeating numerous steps, environment drift became inevitable as staging and production diverged, and troubleshooting issues such as missing permissions consumed substantial time with no systematic audit trail. By migrating the entire infrastructure to AWS CDK with TypeScript, all resources-including VPC networking, IAM roles, S3 buckets, Elastic Beanstalk environments, and CodePipeline-are now version-controlled and deployable with a single command in approximately fifteen minutes. The pipeline implements a progressive deployment strategy: code commits to the main branch trigger automated testing via CodeBuild, followed by deployment to a staging environment for validation. After manual approval, the tested version deploys to production using Elastic Beanstalk's rolling update mechanism to minimize downtime. This staged approach ensures production deployments are verified in a production-like environment, with the complete flow from commit to production completing in under five minutes once approved. The project also includes Lambda functions demonstrating an alternative blue/green deployment strategy with CNAME swapping for zero-downtime deployments with instant rollback capabilities. While initially implemented, this approach was ultimately not adopted for the primary pipeline due to complexity in environment URL management. This decision illustrates an important architectural principle: simpler solutions often provide better maintainability when complexity does not deliver proportionate value. Development revealed practical insights into AWS service integration, deployment pattern trade-offs, and infrastructure design. The architecture incorporates cost-conscious design choices appropriate for development

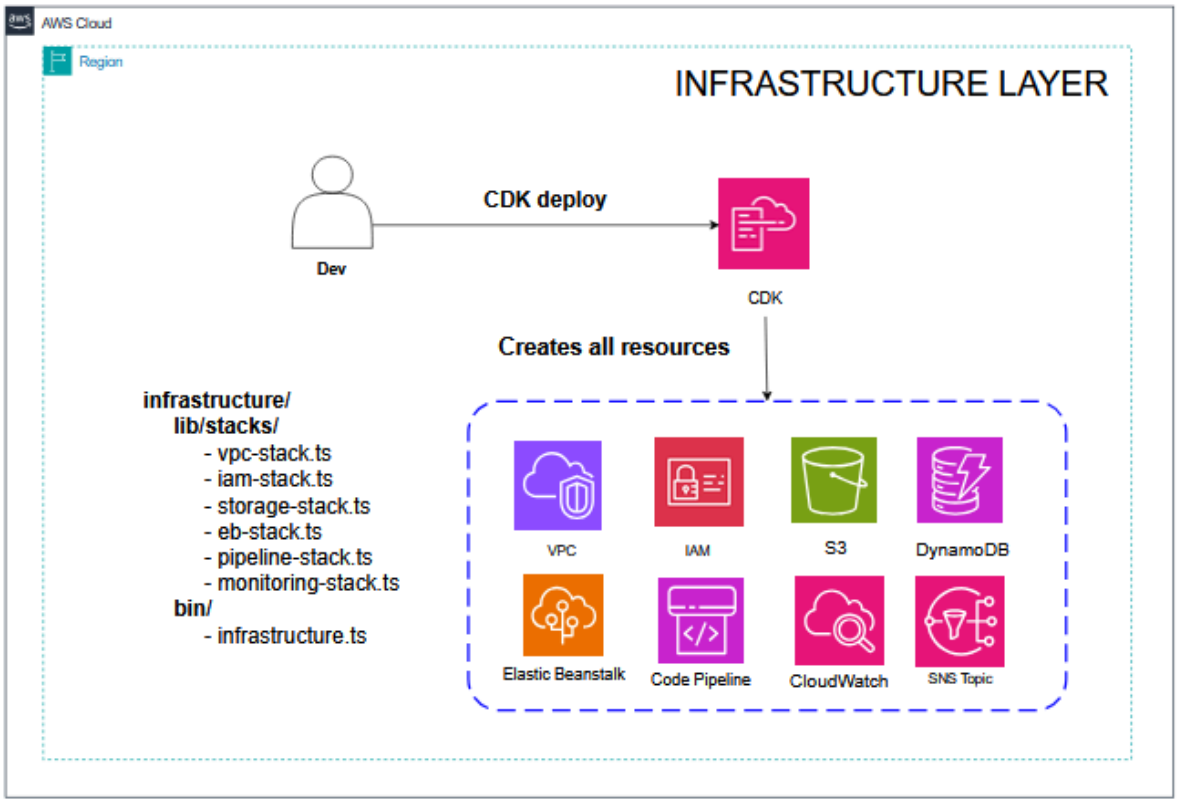
environments: load-balanced Elastic Beanstalk with small instance types, a single NAT Gateway, serverless services that charge only during execution, and S3 lifecycle policies for automatic artifact cleanup.

The complete infrastructure is reproducible and can be destroyed and recreated identically in any AWS account using CDK, providing essential capabilities for disaster recovery and isolated testing environments. This project demonstrates the evolution from manual AWS configuration to systematic Infrastructure as Code management, delivering a production-ready CI/CD pipeline that is fully understood, confidently modifiable, and reliably reproducible.

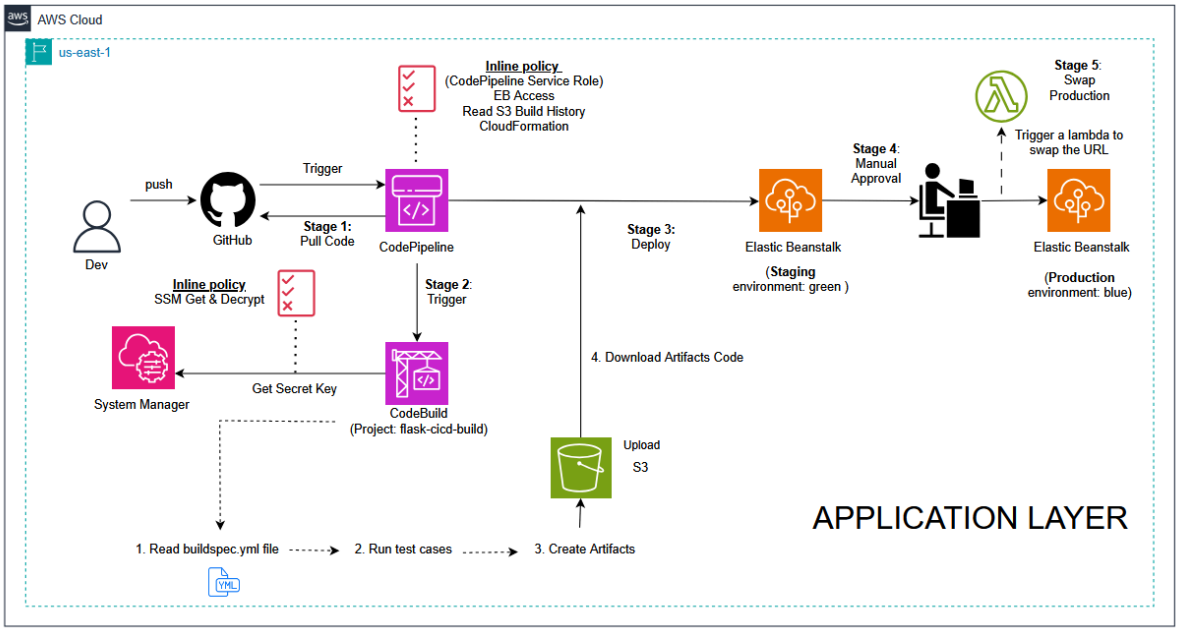
3. Architecture and Design:

3.1 Architecture Diagram:

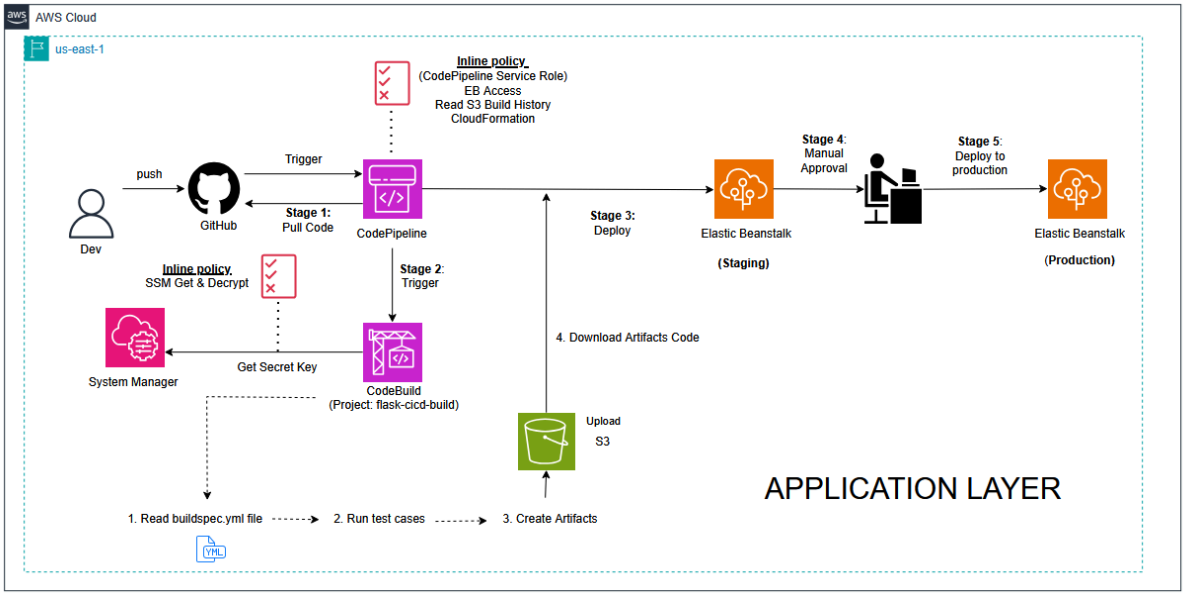
Infrastructure Layer:



Application Layer (initially):



Application Layer (updated):



3.2 Component Explanations:

- **Amazon VPC (Virtual Private Cloud)**

Aspect	Description
Why included	Provides isolated network infrastructure for secure communication between services
Role	Creates private network with public/private subnets across 2 Availability Zones

Aspect	Description
Integration	EC2 instances, Load Balancers, and Lambda functions operate within VPC boundaries
Security	Network isolation, Security Groups control inbound/outbound traffic, NAT Gateway for private subnet internet access
Scaling	Multi-AZ design ensures high availability

- **AWS Elastic Beanstalk**

Aspect	Description
Why included	Simplifies deployment and management of Flask application without managing infrastructure
Role	Hosts two environments: **Staging** (testing) and **Production** (live users)
Integration	Receives deployment packages from CodePipeline, uses EC2 instances behind Application Load Balancer
Security	Instance Profile controls EC2 permissions, Service Role manages AWS resources
Scaling	Auto Scaling Group automatically adjusts capacity based on demand

- **AWS Code Pipeline**

Aspect	Description
Why included	Automates the entire CI/CD workflow from source to production
Role	Orchestrates 5 stages: Source → Build → Deploy Staging → Approval → Swap Production
Integration	Connects GitHub (source), CodeBuild (build), Elastic Beanstalk (deploy), Lambda (swap)
Security	IAM Role with least-privilege permissions for each action
Scaling	Serverless, handles multiple concurrent executions

- **AWS CodeBuild**

Aspect	Description
Why included	Provides managed build environment for running tests and creating deployment packages
Role	Installs dependencies, runs pytest, packages application artifacts
Integration	Reads source from CodePipeline, outputs artifacts to S3, accesses SSM for secrets
Security	Runs in isolated container, IAM Role limits resource access

Aspect	Description
Scaling	Serverless, automatically scales with build demand

- **Amazon S3 (Simple Storage Service)**

Aspect	Description
Why included	Stores pipeline artifacts and deployment packages durably
Role	Two buckets: **Artifacts** (pipeline intermediates), **Deployments** (EB packages)
Integration	CodePipeline stores/retrieves artifacts, Elastic Beanstalk fetches deployment packages
Security	Bucket encryption (SSE-S3), Block Public Access, SSL enforcement
Scaling	Unlimited storage, 30-day lifecycle policy for cost optimization

- **Amazon DynamoDB**

Aspect	Description
Why included	Tracks deployment history and metadata for auditing
Role	Stores deployment records: timestamp, version, status, environment
Integration	Lambda function writes deployment events after environment swaps
Security	Encryption at rest, IAM-based access control
Scaling	On-demand capacity mode, automatically scales throughput

- **AWS Lambda - Only for Demo**

Aspect	Description
Why included	Executes Blue/Green deployment by swapping environment URLs
Role	flask-cicd-swap-environment function swaps CNAMEs between staging and production
Integration	Triggered by CodePipeline after manual approval, calls Elastic Beanstalk API
Security	Execution Role with minimal permissions (EB swap, DynamoDB write)
Scaling	Serverless, 5-minute timeout configured

- **AWS IAM (Identity and Access Management)**

Aspect	Description
Why included	Enforces security through role-based access control
Role	Creates 5 specialized roles: EB Instance, EB Service, CodeBuild, CodePipeline, Lambda

Aspect	Description
Integration	Every service assumes its designated role with specific permissions
Security	Principle of least privilege, no hardcoded credentials, managed policies where possible
Scaling	N/A (security layer)

- **AWS Systems Manager Parameter Store**

Aspect	Description
Why included	Securely stores application secrets (e.g., SECRET_KEY)
Role	Provides secrets to CodeBuild at build time via environment variables
Integration	CodeBuild retrieves `/myapp/secret-key` during build phase
Security	Encrypted storage, IAM-based access, no secrets in source code
Scaling	Managed service, no capacity planning needed

- **Amazon CloudWatch**

Aspect	Description
Why included	Provides centralized monitoring, logging, and alerting
Role	Dashboard displays pipeline metrics, environment health, error rates
Integration	Collects logs from EB instances, CodeBuild, Lambda; triggers alarms
Security	Log Groups with retention policies, encrypted log data
Scaling	Automatically handles log ingestion at any scale

- **AWS CodeConnections (formerly CodeStar Connections)**

Aspect	Description
Why included	Enables secure GitHub repository integration
Role	Authenticates CodePipeline to access GitHub repository
Integration	Triggers pipeline on push to `main` branch, fetches source code
Security	OAuth-based authentication, no GitHub tokens stored in AWS
Scaling	Managed service

4.1 Prerequisites and Initial Setup

4.1.1 Required Software Installation

Before beginning, ensure the following tools are installed on your development machine:

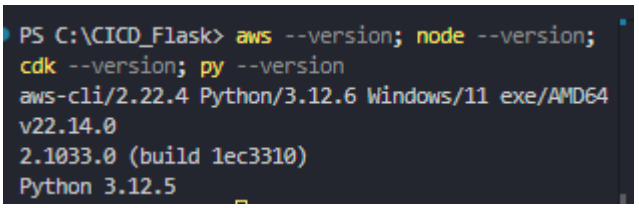
Tool	Version	Purpose
AWS CLI	v2.x	AWS service interaction
Node.js	v18+	CDK runtime
npm	v9+	Package management
AWS CDK CLI	v2.x	Infrastructure deployment
Git	Latest	Version control
Python	3.11	Flask application runtime

Installation Commands (PowerShell):

```
# Install AWS CDK globally
npm install -g aws-cdk

# Verify installations
aws --version
node --version
cdk --version
python --version
```

Expected Output:



4.1.2 AWS CLI Configuration

Configure AWS CLI with appropriate credentials:

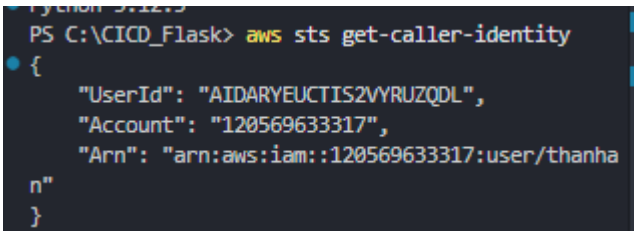
```
# Configure AWS credentials
aws configure

# Enter the following when prompted:
# AWS Access Key ID: [Your Access Key]
# AWS Secret Access Key: [Your Secret Key]
```

```
# Default region name: us-east-1
# Default output format: json

# Verify configuration
aws sts get-caller-identity
```

Expected Output:



4.1.3 GitHub Connection Setup

Create a CodeStar Connection to GitHub for source code integration:

1. Navigate to **AWS Console → Developer Tools → Connections**
2. Click **Create connection**
3. Select **GitHub** as the provider
4. Enter connection name: `github-cicd-connection`
5. Click **Connect to GitHub**
6. Authorize AWS to access your GitHub account
7. Select your GitHub account/organization
8. Click **Connect**

Copy the Connection ARN (format):

```
arn:aws:codeconnections:us-east-1:120569633317:connection/2d267a06-80d3-46b1-9ded-d9637a8a473d
```

> Code Connection
>
>

4.1.4 CDK Bootstrap

Bootstrap CDK in your AWS account (one-time setup):

```
cd C:\\CICD_Flask\\infrastructure

# Bootstrap CDK (creates S3 bucket and IAM roles for CDK)
npx cdk bootstrap aws://<account_id>/us-east-1
```

Expected Output:

```
⏳ Bootstrapping environment aws://<account_id>/us-east-1...
✅ Environment aws://<account_id>/us-east-1 bootstrapped
```

4.2 VPC and Network Configuration

4.2.1 VPC Stack Overview

The VPC Stack creates the network foundation for all services:

Component	Configuration	Purpose
VPC	CIDR: 10.0.0.0/16	Isolated network environment
Public Subnets	2 subnets across 2 AZs	Load balancers, NAT Gateway
Private Subnets	2 subnets across 2 AZs	EC2 instances, Lambda
NAT Gateway	1 (cost-optimized)	Private subnet internet access
Internet Gateway	1	Public subnet internet access

4.2.2 VPC Stack Code

File: `infrastructure/lib/stacks/vpc-stack.ts`

```
export class VpcStack extends cdk.Stack {
  public readonly vpc: ec2.Vpc;

  constructor(scope: Construct, id: string, props?: cdk.StackProps) {
    super(scope, id, props);

    this.vpc = new ec2.Vpc(this, 'FlaskCicdVpc', {
      vpcName: 'flask-cicd-vpc',
```

```
      maxAzs: 2,
      ipAddresses: ec2.IpAddresses.cidr('10.0.0.0/16'),
      subnetConfiguration: [
       {
        name: 'Public',
        subnetType: ec2.SubnetType.PUBLIC,
        cidrMask: 24,
       },
       {
        name: 'Private',
        subnetType: ec2.SubnetType.PRIVATE_WITH_EGRESS,
        cidrMask: 24,
       },
      ],
      natGateways: 1,
    });
  }
}
```

4.2.3 Deploy VPC Stack

```
cd C:\\CICD_Flask\\infrastructure

# Preview changes
npx cdk diff FlaskCicd-Vpc

# Deploy VPC stack
npx cdk deploy FlaskCicd-Vpc
```

Expected Console Output:

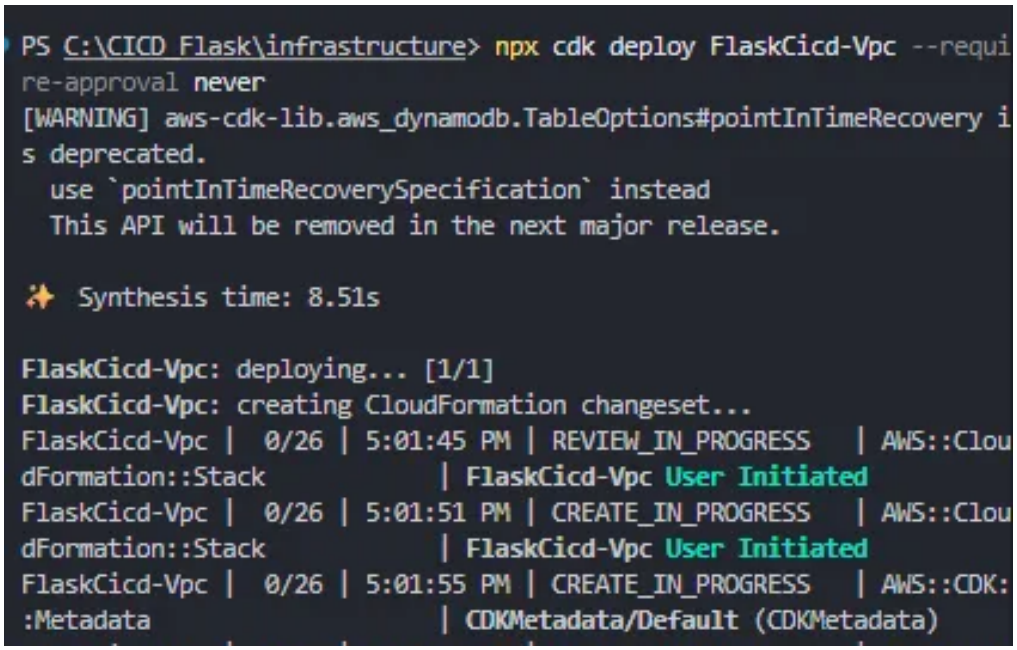
```
FlaskCicd-Vpc: deploying...
FlaskCicd-Vpc: creating CloudFormation changeset...

 ✅  FlaskCicd-Vpc

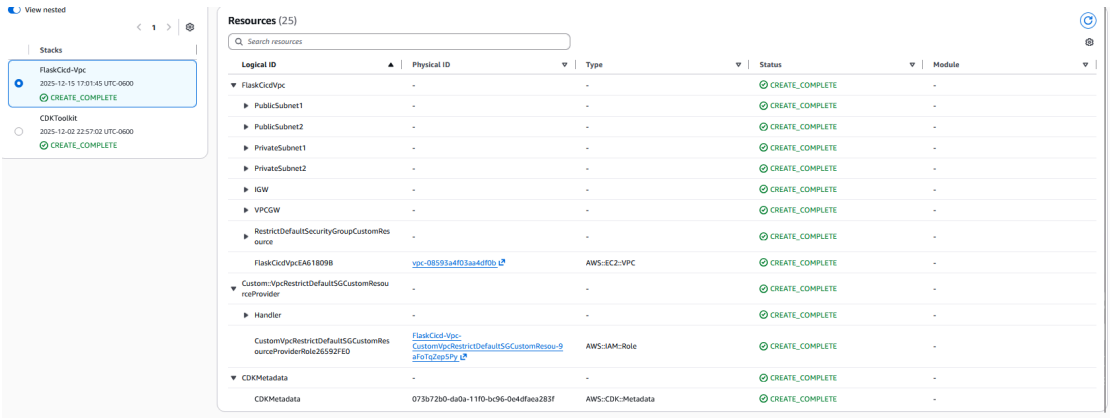
Outputs:
FlaskCicd-Vpc.VpcId = vpc-0abc123def456789
```

Stack ARN:
arn:aws:cloudformation:us-east-1:120569633317:stack/FlaskCicd-Vpc/...

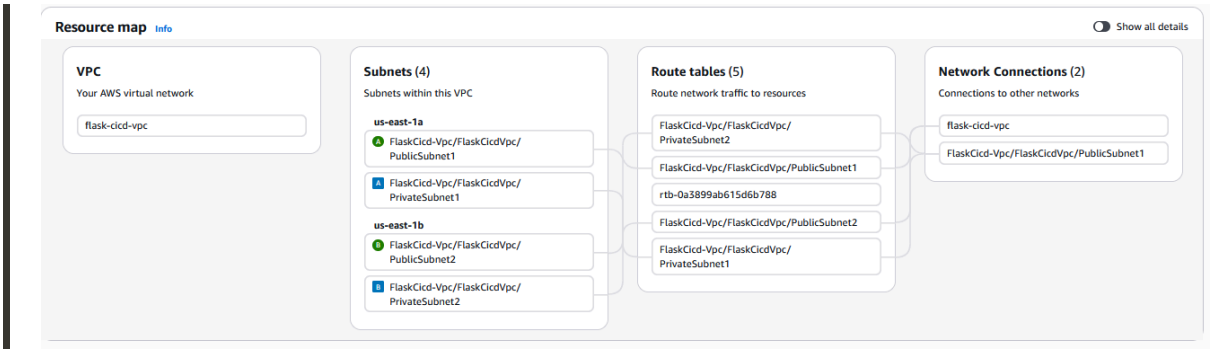
1. VPC stack deploy command



2. VPC Stack verification on AWS Console



3. VPC Source Mapping



4.3 IAM Roles and Security Configuration

4.3.1 IAM Roles Overview

The IAM Stack creates all necessary roles following the principle of least privilege:

Role Name	Service Principal	Purpose
`flask-cicd-eb-ec2-role`	EC2	Elastic Beanstalk instance permissions
`flask-cicd-eb-service-role`	Elastic Beanstalk	EB service operations
`flask-cicd-codebuild-role`	CodeBuild	Build and test operations
`flask-cicd-pipeline-role`	CodePipeline	CI/CD orchestration
`flask-cicd-lambda-swap-role`	Lambda	Environment swap operations

4.3.2 Key IAM Policies

Elastic Beanstalk EC2 Role Policies:

- `AWSElasticBeanstalkWebTier` - Web server permissions
- `AWSElasticBeanstalkWorkerTier` - Background worker permissions
- `AWSElasticBeanstalkMulticontainerDocker` - Docker support

Lambda Swap Role Custom Policy:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowEbSwap",
      "Effect": "Allow",
```

```
      "Action": [
        "elasticbeanstalk:SwapEnvironmentCNAMEs",
        "elasticbeanstalk:DescribeEnvironments"
      ],
      "Resource": "*"
    },
    {
      "Sid": "AllowPipelineCallback",
      "Effect": "Allow",
      "Action": [
        "codepipeline:PutJobSuccessResult",
        "codepipeline:PutJobFailureResult"
      ],
      "Resource": "*"
    }
  ]
}
```

4.3.3 Deploy IAM Stack

```
# Deploy IAM stack
npx cdk deploy FlaskCicd-Iam
```

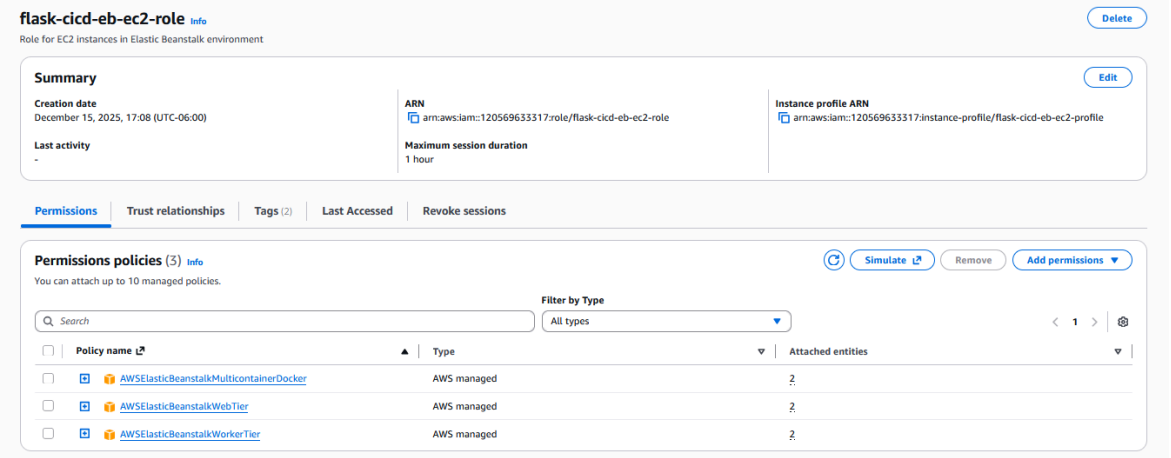
Expected Output:

```
FlaskCicd-Iam: deploying...

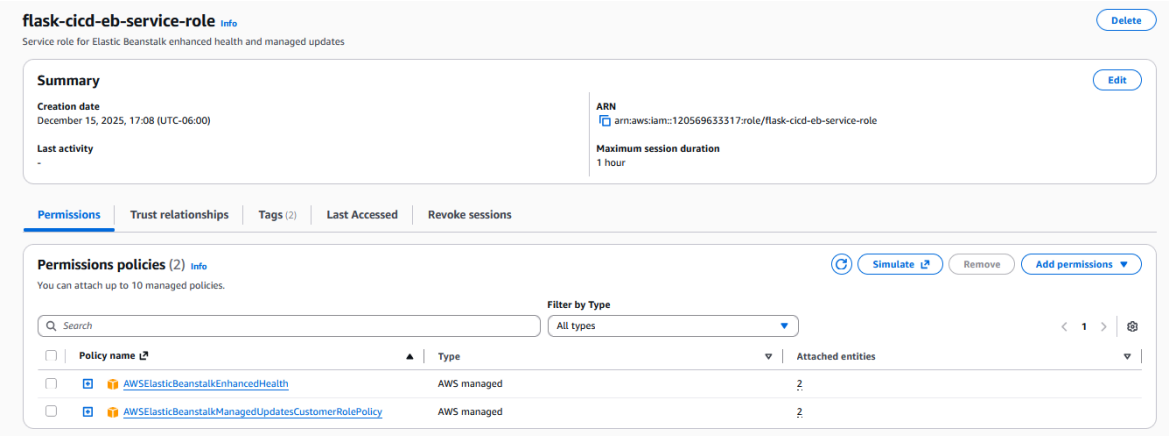
✅ FlaskCicd-Iam

Outputs:
FlaskCicd-Iam.EbInstanceRoleArn = arn:aws:iam::<account_id>:role/flask-cicd-eb-ec2-role
FlaskCicd-Iam.LambdaRoleArn = arn:aws:iam::<account_id>:role/flask-cicd-lambda-swap-role
```

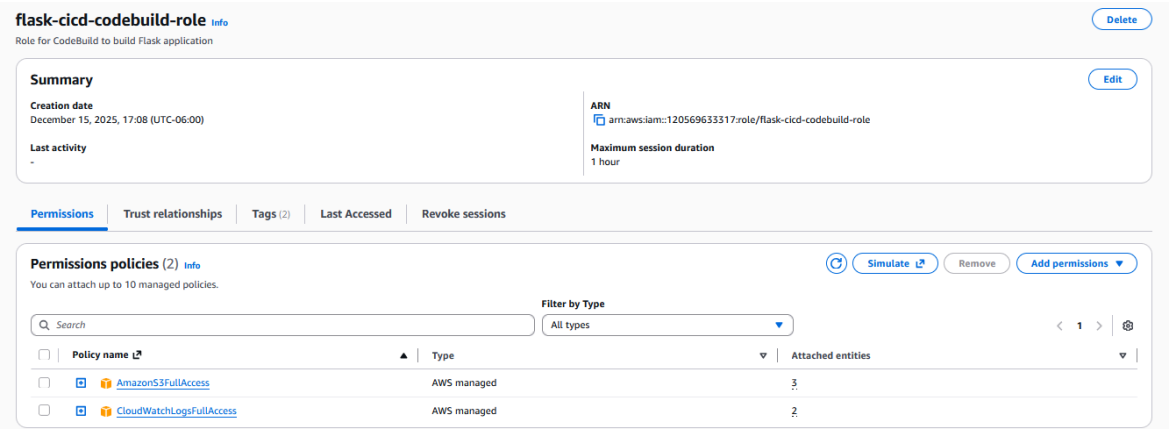
> IAM Console showing the created roles
>
> a. Flask-cicd-eb-ec2-role



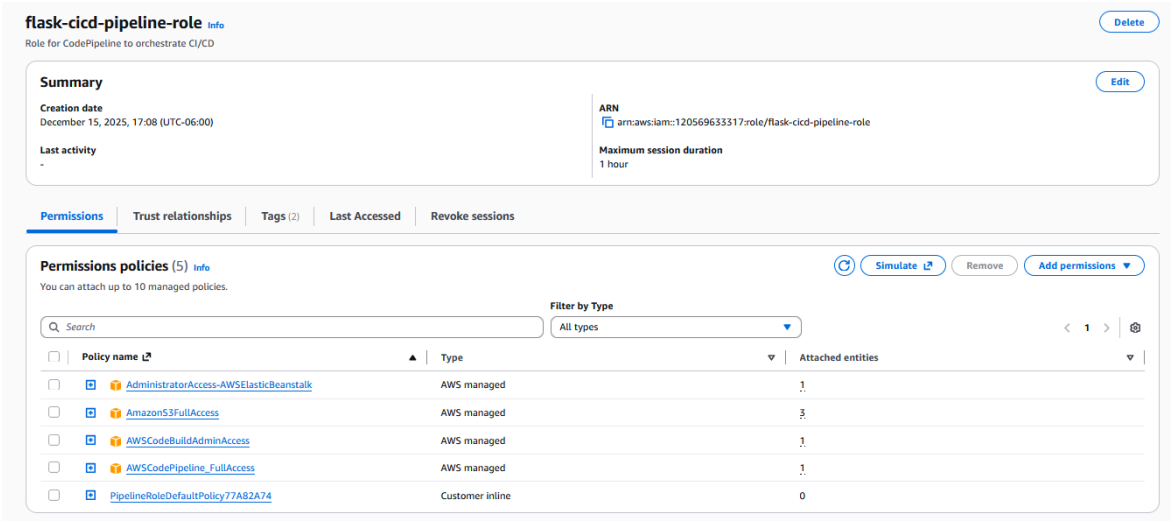
b. flask-cicd-eb-service-role



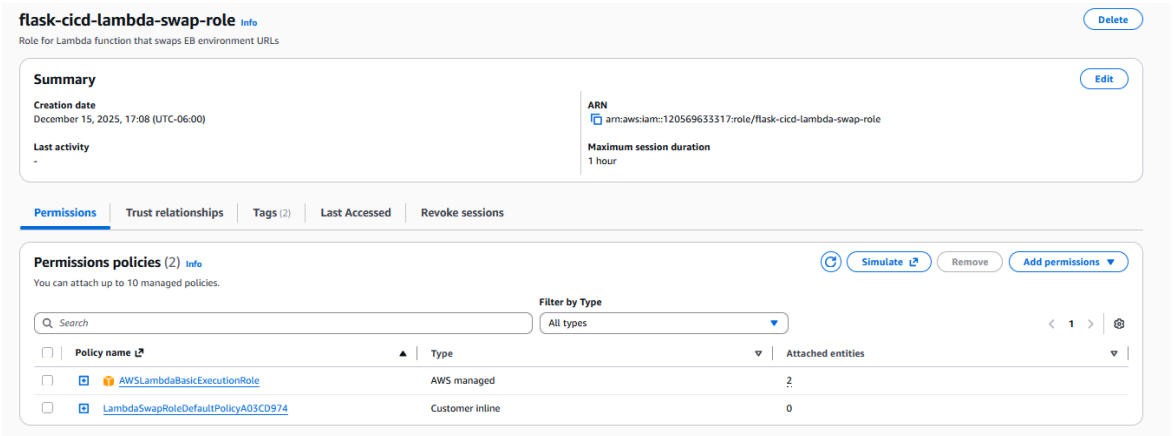
c. flask-cicd-codebuild-role



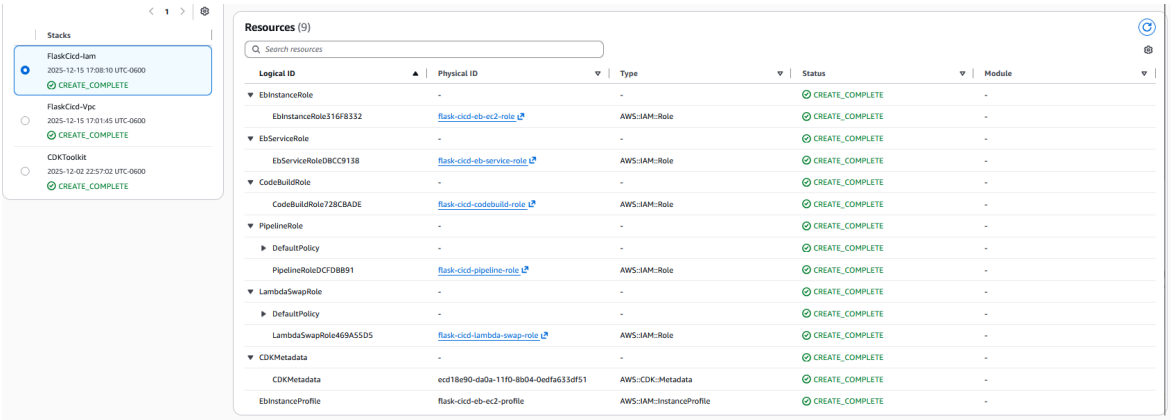
d. flask-cicd-pipeline-role



e. flask-cicd-lambda-swap-role



2. IAM role stack verification



4.4 Storage Configuration (S3 & DynamoDB)

4.4.1 Storage Resources Overview

Resource	Name	Purpose
S3 Bucket	`flask-cicd-artifacts-{account}-{region}`	Pipeline artifacts storage
S3 Bucket	`flask-cicd-deployments-{account}-{region}`	Deployment packages
DynamoDB Table	`flask-cicd-deployments`	Deployment tracking

4.4.2 S3 Security Configuration

All S3 buckets are configured with security best practices:

```
this.artifactBucket = new s3.Bucket(this, 'ArtifactBucket', {
  // Security settings
  encryption: s3.BucketEncryption.S3_MANAGED,  // Server-side encryptio
n
  blockPublicAccess: s3.BlockPublicAccess.BLOCK_ALL,  // No public acce
ss
  enforceSSL: true,  // HTTPS only
  versioned: true,  // Version control for artifacts

  // Lifecycle rules
  lifecycleRules: [{
    expiration: cdk.Duration.days(30),
    noncurrentVersionExpiration: cdk.Duration.days(7),
  }],
});
```

4.4.3 Deploy Storage Stack

```
npx cdk deploy FlaskCicd-Storage
```

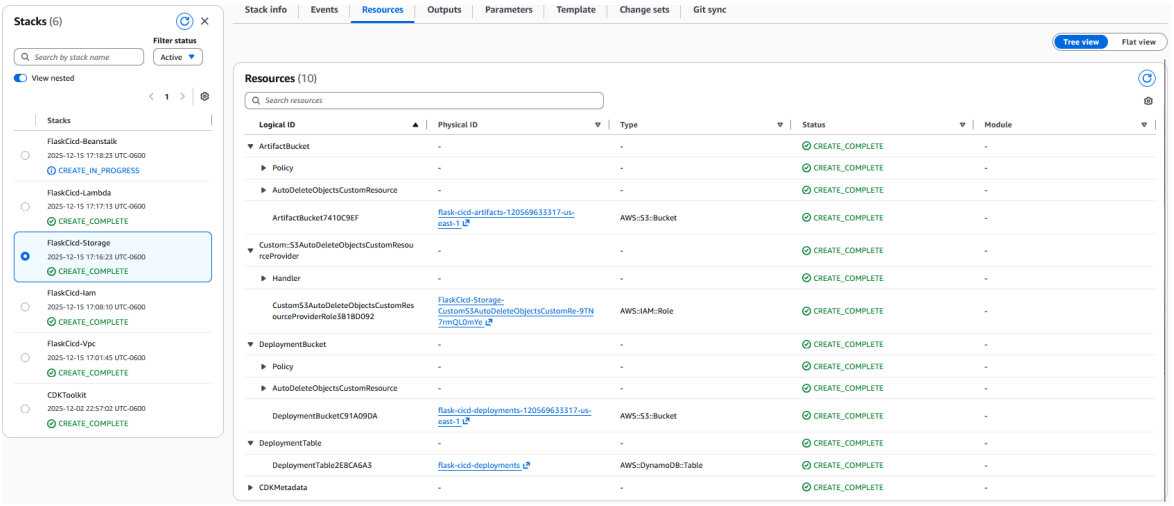
Expected Output:

```
✅ FlaskCicd-Storage

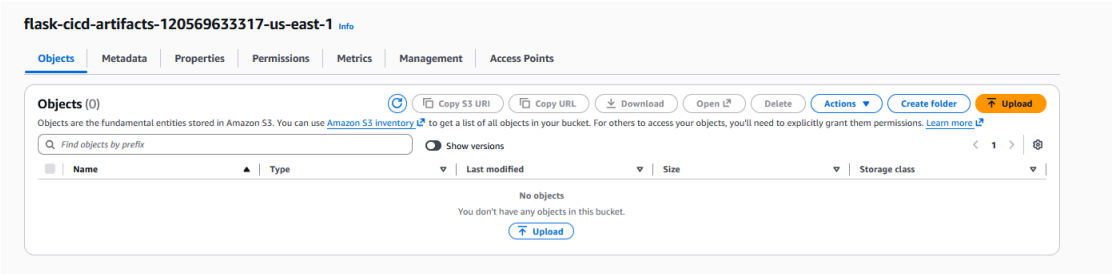
Outputs:
FlaskCicd-Storage.ArtifactBucketName = flask-cicd-artifacts-<account_id
>-us-east-1
```

FlaskCicd-Storage.DeploymentBucketName = flask-cicd-deployments-<account_id>-us-east-1
FlaskCicd-Storage.DeploymentTableName = flask-cicd-deployments

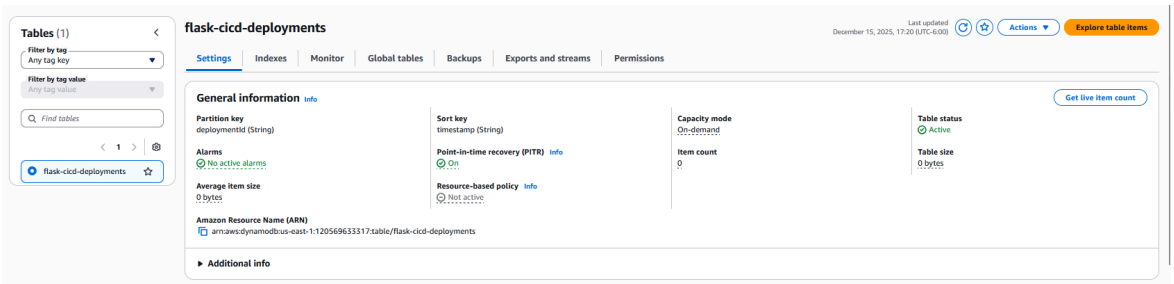
1. Cloud Formation -storage stack verification



2. S3 Bucket



3. Dynamo DB table



4.5 Lambda Function Setup - For Demo

4.5.1 Lambda Function Purpose

The `flask-cicd-swap-environment` Lambda function performs Blue-Green deployment by swapping Elastic Beanstalk environment CNAMEs. This can cause a problem because Code Pipeline doesn't support dynamic detection which is the deployment process is updating the staging process

4.5.2 Lambda Function Code

File: `infrastructure/lambda/swap-environment/index.py`

```
import json
import boto3
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

eb_client = boto3.client('elasticbeanstalk')
codepipeline_client = boto3.client('codepipeline')

def handler(event, context):
    job_id = event['CodePipeline.job']['id']

    try:
        user_params = json.loads(
            event['CodePipeline.job']['data']['actionConfiguration']
            ['configuration']['UserParameters']
        )

        source_env = user_params['sourceEnvironmentName']
        dest_env = user_params['destinationEnvironmentName']

        # Perform CNAME swap
        eb_client.swap_environment_cnames(
            SourceEnvironmentName=source_env,
            DestinationEnvironmentName=dest_env
        )

        # Notify CodePipeline of success
```

```
        codepipeline_client.put_job_success_result(jobId=job_id)

        return {'statusCode': 200, 'body': 'Swap successful'}

    except Exception as e:
        codepipeline_client.put_job_failure_result(
            jobId=job_id,
            failureDetails={'type': 'JobFailed', 'message': str(e)}
        )
        raise
```

4.5.3 Lambda Configuration

Setting	Value	Reason
Runtime	Python 3.11	Latest stable Python
Memory	256 MB	Sufficient for API calls
Timeout	5 minutes	Allow for CNAME propagation
Handler	index.handler	Entry point function

4.5.4 Deploy Lambda Stack

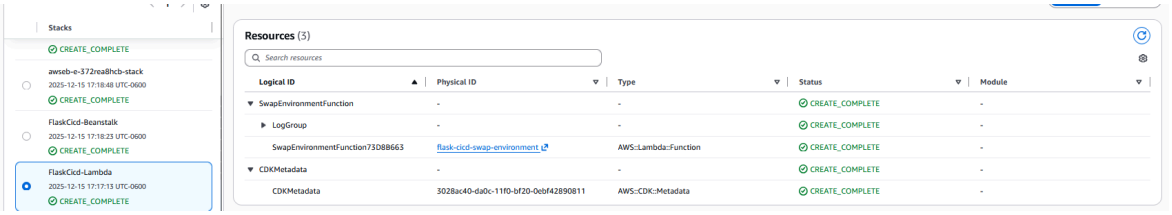
```
npx cdk deploy FlaskCicd-Lambda
```

Expected Output:

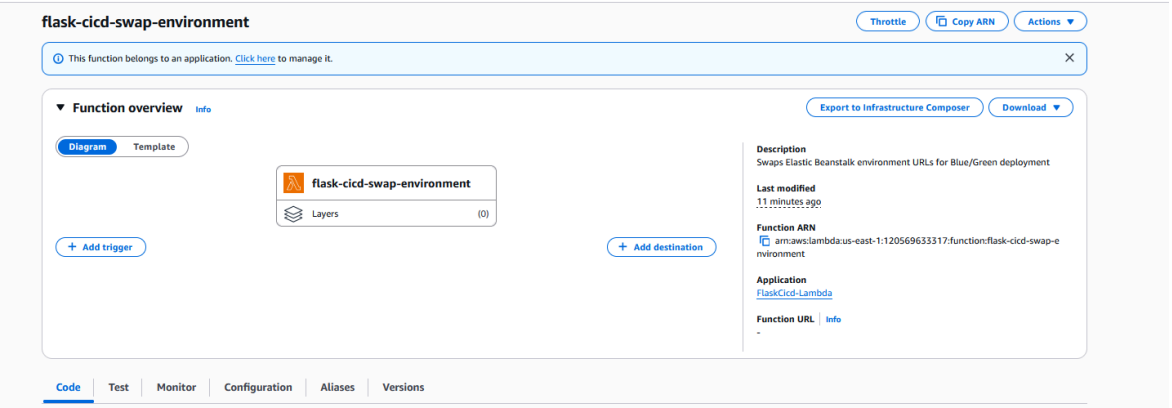
```
✅ FlaskCicd-Lambda

Outputs:
FlaskCicd-Lambda.SwapFunctionArn = arn:aws:lambda:us-east-1:<account_id>:function:flask-cicd-swap-environment
FlaskCicd-Lambda.SwapFunctionName = flask-cicd-swap-environment
```

> 1. Lambda Stack



2. Lambda function



4.6 Elastic Beanstalk Configuration

4.6.1 Environment Architecture

The solution uses a Blue-Green deployment pattern with two environments:

Environment	Name	Purpose
Staging (Blue)	`flask-cicd-staging`	Testing new deployments
Production (Green)	`flask-cicd-production`	Live traffic

4.6.2 Environment Configuration

```
const commonOptionSettings = [
  // Instance configuration
  {
    namespace: 'aws:ec2:instances',
    optionName: 'InstanceTypes',
    value: 't2.micro',  // Free tier eligible
  },
  // VPC configuration
  {
```

```
    namespace: 'aws:ec2:vpc',
    optionName: 'VPCId',
    value: props.vpc.vpcId,
  },
  {
    namespace: 'aws:ec2:vpc',
    optionName: 'Subnets',
    value: props.vpc.publicSubnets.map(s ⇒ s.subnetId).join(','),
  },
  // Enhanced health reporting
  {
    namespace: 'aws:elasticbeanstalk:healthreporting:system',
    optionName: 'SystemType',
    value: 'enhanced',
  },
  // CloudWatch log streaming
  {
    namespace: 'aws:elasticbeanstalk:cloudwatch:logs',
    optionName: 'StreamLogs',
    value: 'true',
  },
];
```

4.6.3 Flask Application Configuration Files

Procfile - WSGI Server Configuration:

```
web: gunicorn --bind :8000 --workers 3 --threads 2 application:application
```

requirements.txt - Python Dependencies:

```
Flask>=2.3.0
gunicorn>=21.0.0
pytest>=7.4.0
pytest-cov>=4.1.0
```

.ebextensions/01_flask.config - Environment Variables:

```
option_settings:
  aws:elasticbeanstalk:application:environment:
    FLASK_APP: application.py
    FLASK_ENV: production
```

4.6.4 Deploy Beanstalk Stack

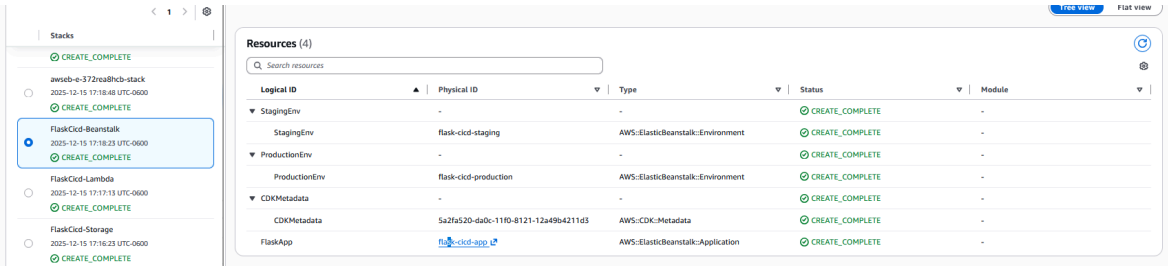
```
npx cdk deploy FlaskCicd-Beanstalk
```

Expected Output:

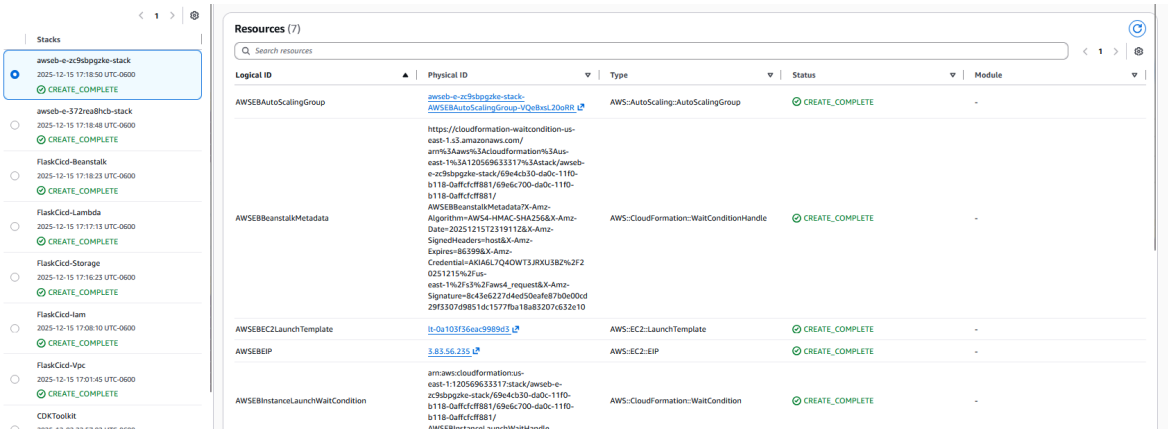
```
✅ FlaskCicd-Beanstalk

Outputs:
FlaskCicd-Beanstalk.ApplicationName = flask-cicd-app
FlaskCicd-Beanstalk.StagingEnvName = flask-cicd-staging
FlaskCicd-Beanstalk.ProductionEnvName = flask-cicd-production
```

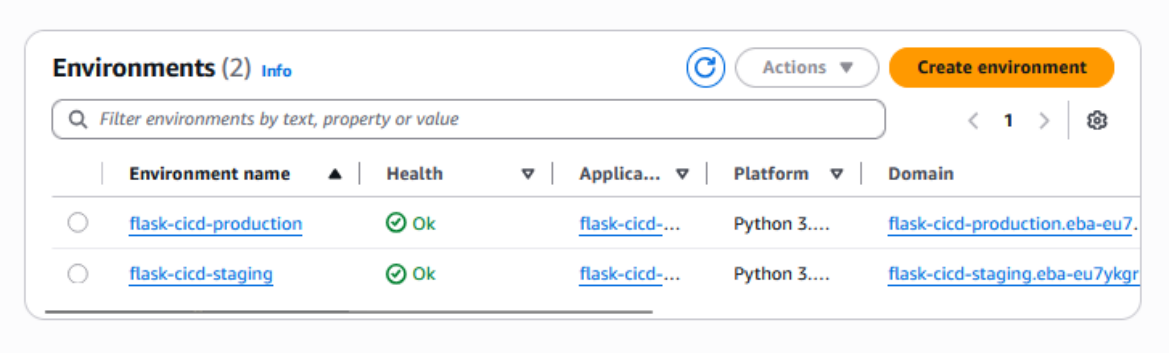
1. Elastic Beanstalk Cloud Formation stack



2. Stack verification



3. Environments are created in the application



4.7 CodePipeline and CodeBuild Setup

4.7.1 Pipeline Stages Overview

Stage	Action	Description
1. Source	GitHub_Source	Pull code from GitHub repository
2. Build	Build_And_Test	Run pytest, create deployment package
3. Deploy_Staging	Deploy_To_Staging	Deploy to staging environment
4. Approval	Approve_Production_Swap	Manual approval gate
5. Swap_Production	Swap_Environment_URLs	Lambda swaps CNAMEs

4.7.2 CodeBuild Configuration

Build Specification (buildspec.yml):

```
version: 0.2

phases:
  install:
    runtime-versions:
      python: 3.11
    commands:
      - echo "Installing dependencies..."
      - cd flask-cicd-project
      - pip install -r requirements.txt

  pre_build:
```

```
    commands:
      - echo "Running tests..."
      - cd flask-cicd-project
      - python -m pytest tests/ -v --tb=short

  build:
    commands:
      - echo "Creating deployment package..."
      - cd flask-cicd-project
      - zip -r ../deployment-package.zip . -x "*.pyc" -x "__pycache__/*" -x ".gi
t/*" -x "tests/*"

artifacts:
  files:
    - flask-cicd-project/**/*
  discard-paths: no
```

4.7.3 Pipeline Source Configuration

```
new codepipeline_actions.CodeStarConnectionsSourceAction({
  actionName: 'GitHub_Source',
  owner: 'thewanter',
  repo: 'CICD_Flask',
  branch: 'main',
  connectionArn: config.connectionArn,
  output: sourceOutput,
  triggerOnPush: true,  // Auto-trigger on push
}),
```

4.7.4 Deploy Pipeline Stack

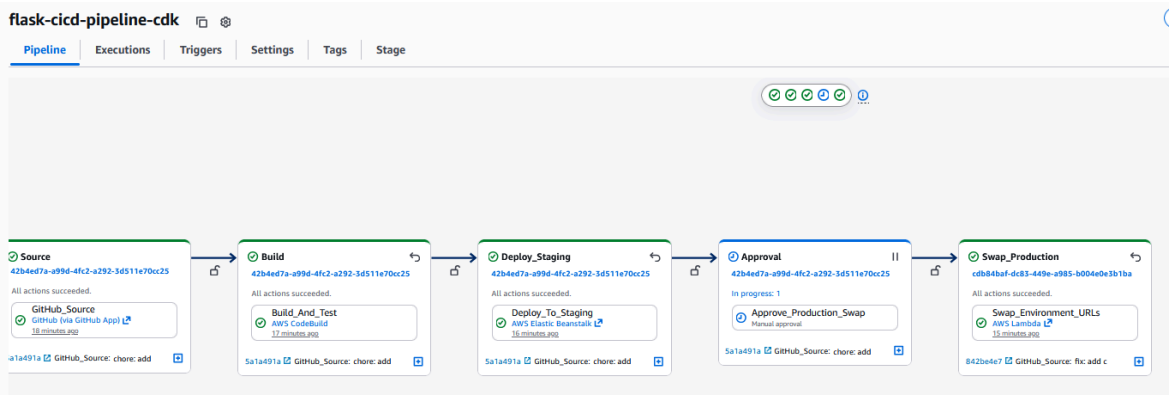
```
npx cdk deploy FlaskCicd-Pipeline
```

Expected Output:

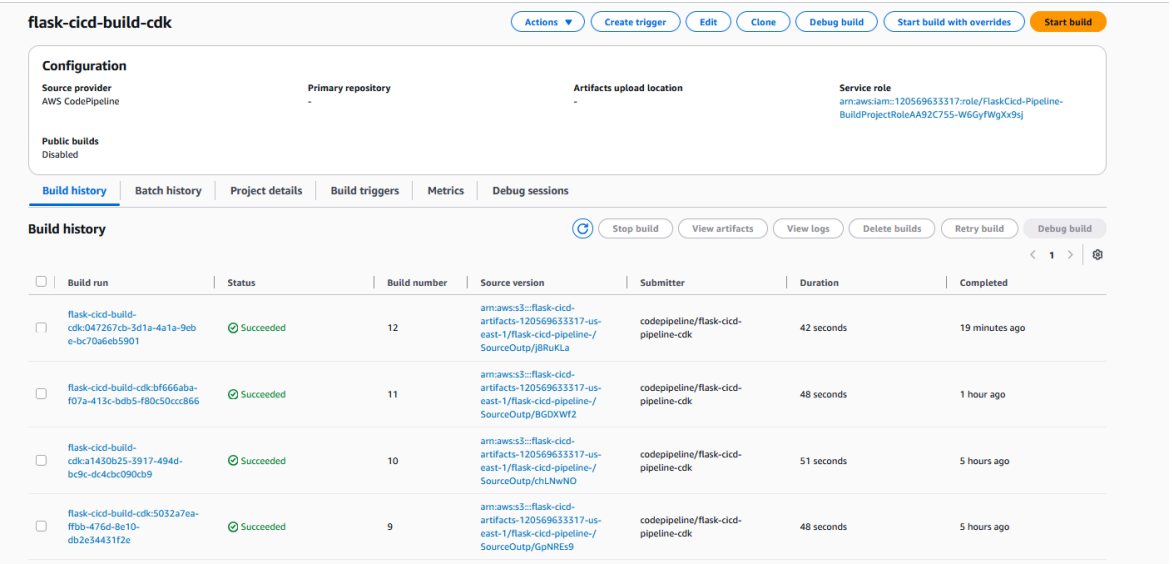
```
✅ FlaskCicd-Pipeline
```

```
Outputs:
FlaskCicd-Pipeline.PipelineName = flask-cicd-pipeline-cdk
FlaskCicd-Pipeline.PipelineArn = arn:aws:codepipeline:us-east-1:120569633317:flask-cicd-pipeline-cdk
FlaskCicd-Pipeline.BuildProjectName = flask-cicd-build-cdk
```

1. CodePipeline Console showing all 5 stages



2. CodeBuild project through CDK



4.8 Monitoring and Alerting Configuration

4.8.1 CloudWatch Dashboard

The monitoring stack creates a comprehensive dashboard with the following widgets:

Widget	Metrics	Purpose
Pipeline Executions	SucceededPipelines, FailedPipelines	Track CI/CD success rate
Staging Health	EnvironmentHealth	Monitor staging status
Production Health	EnvironmentHealth	Monitor production status
Request Counts	2xx, 4xx, 5xx responses	Track error rates

4.8.2 CloudWatch Alarms

```
// Production 5xx Error Alarm
new cloudwatch.Alarm(this, 'Production5xxAlarm', {
  alarmName: 'flask-cicd-production-5xx-errors',
  metric: new cloudwatch.Metric({
    namespace: 'AWS/ElasticBeanstalk',
    metricName: 'ApplicationRequests5xx',
    dimensionsMap: { EnvironmentName: props.productionEnvName },
    statistic: 'Sum',
    period: cdk.Duration.minutes(5),
  }),
  threshold: 10,
  evaluationPeriods: 2,
  comparisonOperator: cloudwatch.ComparisonOperator.GREATER_THAN_T
HRESHOLD,
  actionsEnabled: true,
});
```

4.8.3 SNS Notification Setup

```
this.alarmTopic = new sns.Topic(this, 'AlarmTopic', {
  topicName: 'flask-cicd-alarms',
  displayName: 'Flask CI/CD Alarms',
});

// Add email subscription
if (props.notificationEmail) {
  this.alarmTopic.addSubscription(
    new sns_subscriptions.EmailSubscription(props.notificationEmail)
```

```
  );
}
```

4.8.4 Deploy Monitoring Stack

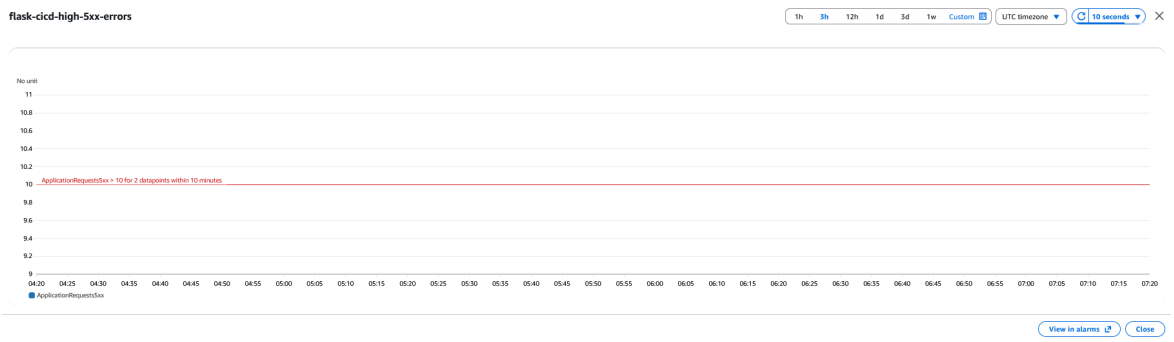
```
npx cdk deploy FlaskCicd-Monitoring
```

Expected Output:

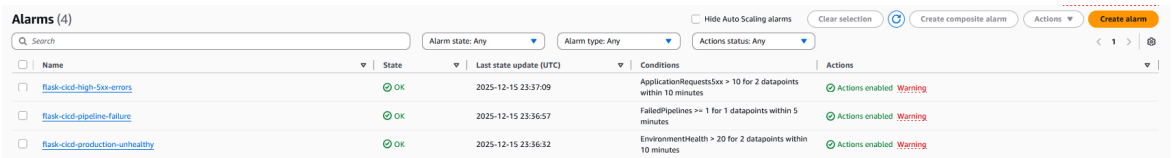
```
✅ FlaskCicd-Monitoring

Outputs:
FlaskCicd-Monitoring.DashboardName = FlaskCICD-Dashboard
FlaskCicd-Monitoring.AlarmTopicArn = arn:aws:sns:us-east-1:120569633317:flask-cicd-alarms
```

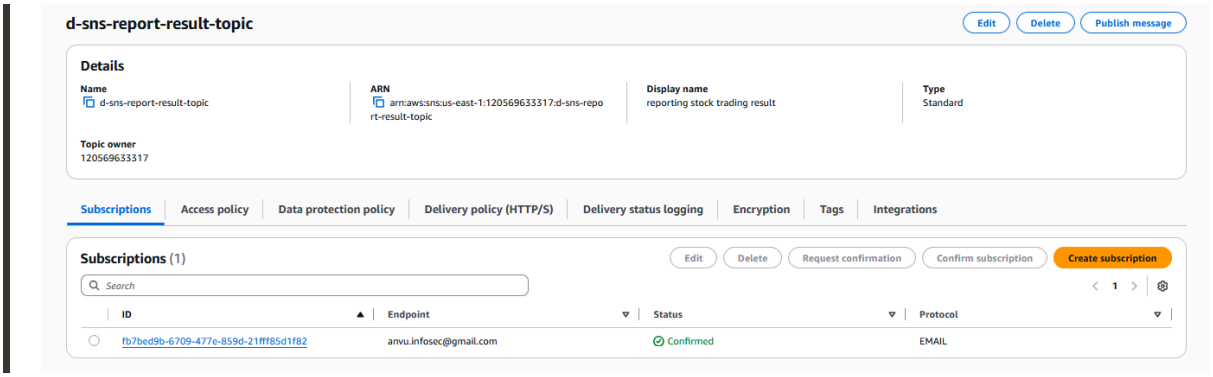
1. CloudWatch Dashboard showing all metrics widgets



2. CloudWatch Alarms page showing configured alarms



3. SNS Topic with email subscription



4.9 Testing and Validation

4.9.1 Deploy All Stacks

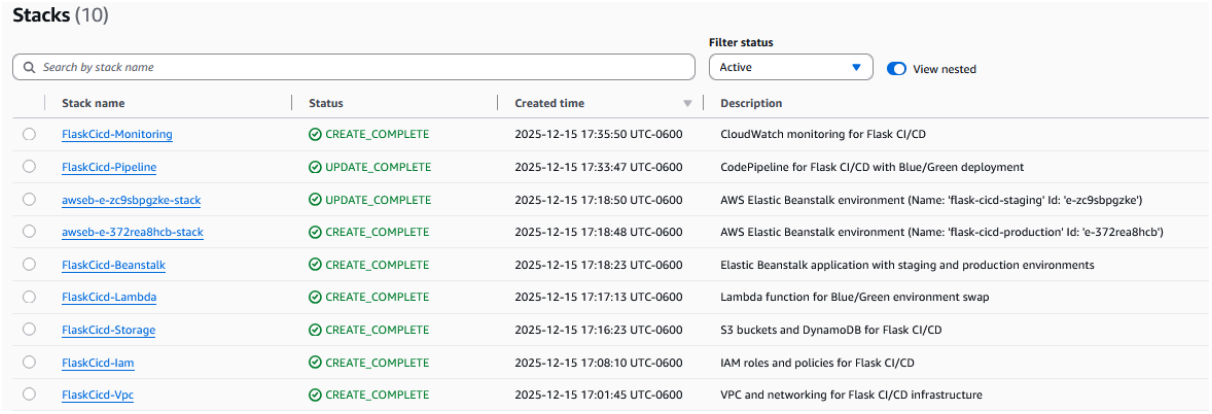
Deploy all stacks in the correct dependency order:

```
cd C:\\CICD_Flask\\infrastructure

# Deploy all stacks at once
npx cdk deploy --all --require-approval never
```

Deployment Order:

1. FlaskCicd-Vpc
2. FlaskCicd-Iam
3. FlaskCicd-Storage
4. FlaskCicd-Lambda
5. FlaskCicd-Beanstalk
6. FlaskCicd-Pipeline
7. FlaskCicd-Monitoring



List of created stacks

4.9.2 Validate Infrastructure

Check VPC:

```
aws ec2 describe-vpcs --filters "Name=tag:Name,Values=flask-cicd-vpc" --query "Vpcs[0].VpcId"
```

Check Elastic Beanstalk Environments:

```
aws elasticbeanstalk describe-environments --application-name flask-cicd-app --query "Environments[*].[EnvironmentName,Status,Health]"
```

Expected Output:

```
[
  ["flask-cicd-staging", "Ready", "Green"],
  ["flask-cicd-production", "Ready", "Green"]
]
```

Check Lambda Function:

```
aws lambda get-function --function-name flask-cicd-swap-environment --query "Configuration.[FunctionName,Runtime,State]"
```

4.9.3 Trigger Pipeline Test

Push a test commit to trigger the pipeline:

```
cd C:\\CICD_Flask

# Make a small change (e.g., update README)
echo "# Test deployment $(Get-Date)" >> flask-cicd-project/README.md

# Commit and push
git add .
git commit -m "Test CI/CD pipeline trigger"
git push origin main
```

4.9.4 Monitor Pipeline Execution

Check Pipeline Status:

```
aws codepipeline get-pipeline-state --name flask-cicd-pipeline-cdk --quer
y "stageStates[*].[stageName,latestExecution.status]"
```

Expected Output During Execution:

```
[
   ["Source", "Succeeded"],
   ["Build", "InProgress"],
   ["Deploy_Staging", "NotStarted"],
   ["Approval", "NotStarted"],
   ["Swap_Production", "NotStarted"]
]
```

> 1. Pipeline in progress showing Source succeeded
> 2. CodeBuild logs showing pytest execution
> 3. Staging deployment in progress
> 4. Manual approval pending
> 5. Successful production swap completion

4.9.5 Run Local Tests

Validate the Flask application locally before deployment:

```
cd C:\\CICD_Flask\\flask-cicd-project

# Create virtual environment
python -m venv venv
.\\venv\\Scripts\\Activate

# Install dependencies
pip install -r requirements.txt

# Run tests
python -m pytest tests/ -v --tb=short
```

Expected Test Output:

```
============================= test session starts ==============
============
platform win32 -- Python 3.11.7, pytest-7.4.0
collected 3 items

tests/test_main.py::test_home PASSED                    [ 33%]
tests/test_main.py::test_health PASSED                  [ 66%]
tests/test_main.py::test_users PASSED                   [100%]

============================== 3 passed in 0.45s ==============
==============
```

4.9.6 Verify Deployed Application

Test the deployed staging environment:

```
# Get staging URL
$stagingUrl = aws elasticbeanstalk describe-environments `
  --environment-names flask-cicd-staging `
  --query "Environments[0].CNAME" --output text

# Test endpoints
```

```
Invoke-RestMethod -Uri "<http://$stagingUrl/>"
Invoke-RestMethod -Uri "<http://$stagingUrl/health>"
Invoke-RestMethod -Uri "<http://$stagingUrl/api/users>"
```

Expected Responses:

```
// GET /
{"message": "Welcome to Flask CI/CD Project", "version": "1.0.0"}

// GET /health
{"status": "healthy", "timestamp": "2025-12-15T10:30:00Z"}

// GET /api/users
{"users": [{"id": 1, "name": "Test User"}], "count": 1}
```

5. Proof of Concept (POC) Validation

Workflow 1: Code Deployment Pipeline (Progressive Deployment)

Trigger: Developer pushes code to `main` branch

Flow:

```
Developer Commit → GitHub → AWS CodePipeline
   ↓
Stage 1: Source
   → Pull latest code from GitHub (main branch)
   → Store source artifact in S3
   ↓
Stage 2: Build
   → AWS CodeBuild reads buildspec.yml
   → Install dependencies (pip install -r requirements.txt)
   → Run tests (pytest)
   → Package application
   → Store build artifact in S3
   ↓
Stage 3: Deploy_Staging
   → Deploy to flask-cicd-staging environment
   → Elastic Beanstalk LoadBalanced (ALB + Auto Scaling)
```

```
    → Health checks verify deployment success
    → Staging URL: <http://flask-cicd-staging.xxx.elasticbeanstalk.com>
    ↓
Stage 4: Manual Approval
    → Pipeline PAUSES automatically
    → Team reviews staging environment
    → QA testing, smoke tests, validation
    → Approver clicks "Approve" in AWS Console
    ↓
Stage 5: Deploy_Production
    → Deploy to flask-cicd-production environment
    → Elastic Beanstalk LoadBalanced (ALB + Auto Scaling)
    → Health checks verify deployment success
    → Production URL: <http://flask-cicd-production.xxx.elasticbeanstalk.co
m>
    ↓
Complete: Version deployed to both environments
```

Workflow 2: Staging Environment Testing

Trigger: Staging deployment completes successfully

User Actions:

1. Navigate to staging URL
2. Verify application functionality
3. Check version number at `/` endpoint
4. Test critical API endpoints (`/health`, `/api/users`)
5. Review CloudWatch logs for errors
6. Load test if needed

Decision Point:

- **Pass** → Approve production deployment in AWS Console
- **Fail** → Reject and fix issues in new commit

Example Test Commands:

```
# Health check
curl <http://flask-cicd-staging.xxx.elasticbeanstalk.com/health>

# Version check
curl <http://flask-cicd-staging.xxx.elasticbeanstalk.com/>

# API test
curl <http://flask-cicd-staging.xxx.elasticbeanstalk.com/api/users>
```

Workflow 3: Production Deployment Approval

Trigger: Stakeholder reviews staging environment

Process:

1. Open AWS CodePipeline Console
2. Navigate to `flask-cicd-pipeline-cdk`
3. Click "Review" on Manual Approval stage
4. Add approval comments (optional)
5. Click "Approve" or "Reject"

System Response:

- **Approve** → Stage 5 executes, deploys to production
- **Reject** → Pipeline stops, requires new code push to restart

Best Practices:

- Document test results in approval comments
- Include version number being approved
- Note any known issues or limitations

Workflow 4: Rollback Strategy

Trigger: Production deployment causes issues

Options:

Option A: Elastic Beanstalk Console Rollback (Fast)

1. AWS Console → Elastic Beanstalk → flask-cicd-production
2. Application Versions → Select previous version
3. Click "Deploy" to specific version
4. Fast rollback (2-3 minutes)
5. **Pros:** Immediate, no code changes
6. **Cons:** Bypasses pipeline audit trail

Option B: Git Revert + New Pipeline (Recommended)

```
# Find the problematic commit
git log --oneline

# Revert the commit
git revert <commit-hash>

# Push to trigger pipeline
git push origin main
```

1. Pipeline deploys reverted code through full flow
2. **Pros:** Complete audit trail, tested through staging
3. **Cons:** Slower (full pipeline execution)

Expected Behavior:

- Rolling deployment prevents complete outage
- ALB health checks route traffic only to healthy instances
- Previous version remains available in S3
- Auto Scaling maintains minimum instance count

Workflow 5: Infrastructure Updates

Trigger: Developer modifies CDK code in `infrastructure/`

Process:

```
# Navigate to infrastructure directory
cd infrastructure
```

```
# Compile TypeScript
npx tsc

# Preview changes (optional but recommended)
npx cdk diff FlaskCicd-Pipeline

# Deploy specific stack
npx cdk deploy FlaskCicd-Pipeline --require-approval never

# Or deploy all stacks
npx cdk deploy --all --require-approval never
```

Stack Dependencies:

- **VPC** (independent) - Virtual Private Cloud
- **IAM** (independent) - Roles and policies
- **Storage** (independent) - S3 buckets, DynamoDB table
- **Lambda** (depends on IAM) - Swap and detect functions
- **Beanstalk** (depends on VPC, IAM) - Application environments
- **Pipeline** (depends on Storage, Beanstalk, IAM) - CodePipeline
- **Monitoring** (depends on all) - CloudWatch alarms, SNS

Deployment Order:
CloudFormation automatically handles dependencies, deploy in any order.

Workflow 6: Application Health Monitoring

Trigger: Continuous monitoring via CloudWatch

Events Tracked:

- HTTP 4xx/5xx errors
- Application response time
- EC2 instance health
- Auto Scaling events
- Deployment success/failure

- Environment transitions (Launching, Updating, Ready)

Alert Workflow:

```
CloudWatch detects anomaly
    ↓
SNS topic triggered
    ↓
Email/SMS to operations team
    ↓
Team investigates CloudWatch Logs
    ↓
Initiate rollback if needed
```

CloudWatch Log Groups:

- `/aws/elasticbeanstalk/flask-cicd-staging/var/log/eb-engine.log`
- `/aws/elasticbeanstalk/flask-cicd-production/var/log/eb-engine.log`
- Application logs from Flask

Workflow 7: Blue/Green Alternative (Educational - Not Active)

Note: Lambda functions exist for demonstration but are NOT used in current pipeline

Theoretical Flow (if activated):

1. Both environments receive deployment
2. Approve swap decision
3. Lambda function `flask-cicd-swap-environment` invoked
4. Lambda checks both environments are Ready + Green
5. Lambda calls `swap_environment_cnames()` API
6. Staging URL → Production environment
7. Production URL → Staging environment
8. Next deployment goes to "staging" name (which now serves production URL)

Why Disabled:

- Adds complexity without clear benefit for this use case

- Environment name/URL confusion after swap
- Progressive deployment simpler and clearer
- Requires dynamic environment detection
- **Kept in codebase for educational purposes**

Lambda Functions:

- `flask-cicd-swap-environment` - Performs CNAME swap with health checks
- `flask-cicd-detect-staging` - Detects which environment has staging URL

Workflow 8: New Feature Development

Recommended Flow:

```
# Create feature branch
git checkout -b feature/new-api-endpoint

# Make changes
# ... edit code ...

# Test locally
cd flask-cicd-project
python -m pytest

# Commit changes
git add .
git commit -m "Add new API endpoint"

# Push feature branch
git push origin feature/new-api-endpoint

# Create Pull Request on GitHub
# ... review and approve ...

# Merge to main
git checkout main
git merge feature/new-api-endpoint
git push origin main
```

```
# Pipeline automatically triggers
# Monitor in AWS CodePipeline Console
```

Best Practices:

- Always test locally before pushing
- Use descriptive commit messages
- Create PR for code review
- Merge to main only after approval
- Monitor pipeline execution

Workflow 9: Troubleshooting Failed Deployments

Scenario: Pipeline fails at Deploy_Staging or Deploy_Production

Investigation Steps:

1. **Check CodePipeline Console:**
   - View failed stage details
   - Click "Details" link for error message
2. **Check Elastic Beanstalk Console:**
   - Environment health status
   - Recent events (last 30 minutes)
   - Configuration changes
3. **Check CloudWatch Logs:**

   ```
   # View logs for specific environment
   aws logs tail /aws/elasticbeanstalk/flask-cicd-staging/var/log/eb-engine.log --follow
   ```

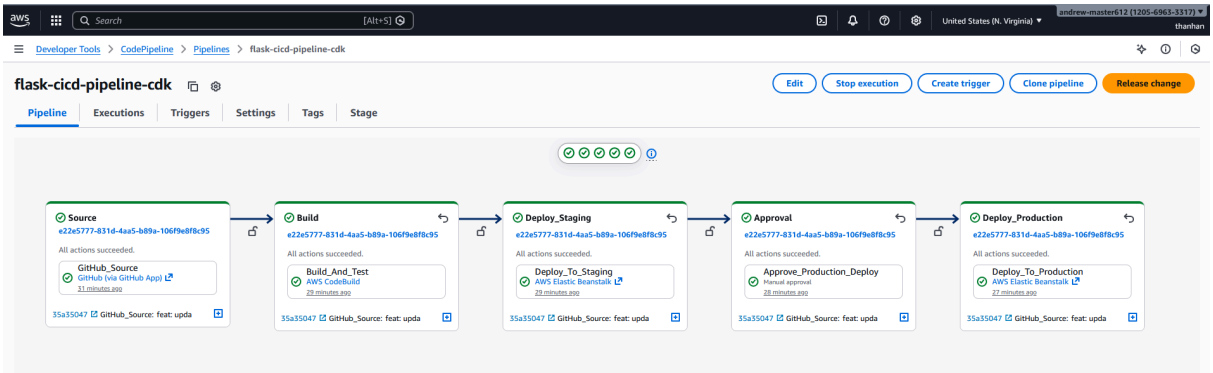
4. **Common Issues:**
   - **Timeout:** Increase health check timeout in beanstalk-stack.ts
   - **Health check fails:** Verify `/health` endpoint responds 200
   - **Instance launch fails:** Check EC2 instance limits/quotas

- **Dependency errors:** Verify requirements.txt versions

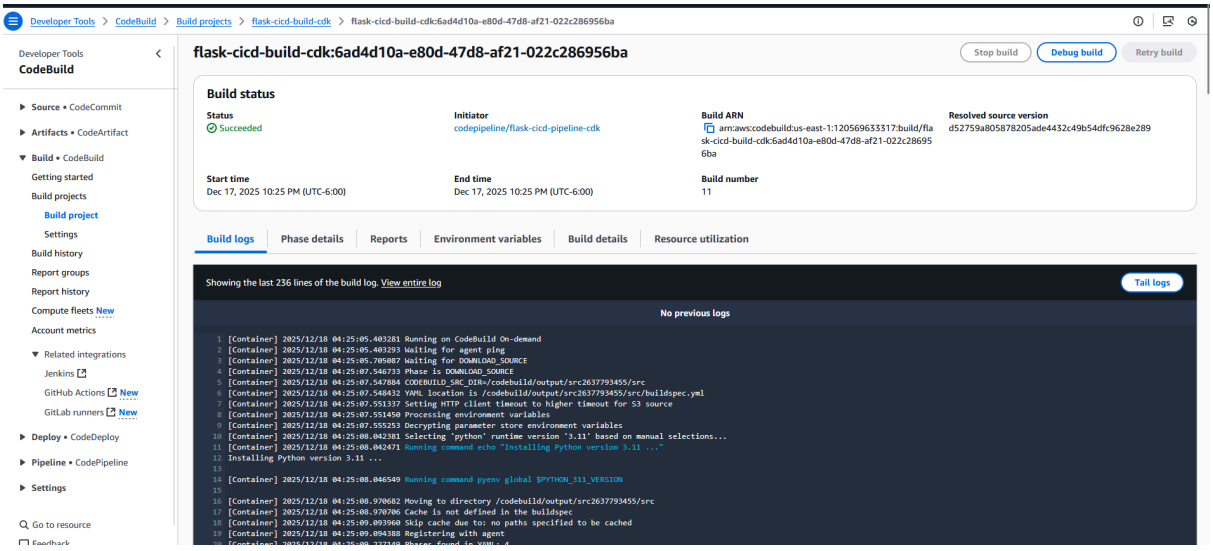
5. **Fix and Retry:**
    - Fix issue in code
    - Commit and push
    - Pipeline automatically retries

6. Outputs, Logs & Screenshots

6.1 Pipeline Execution Logs

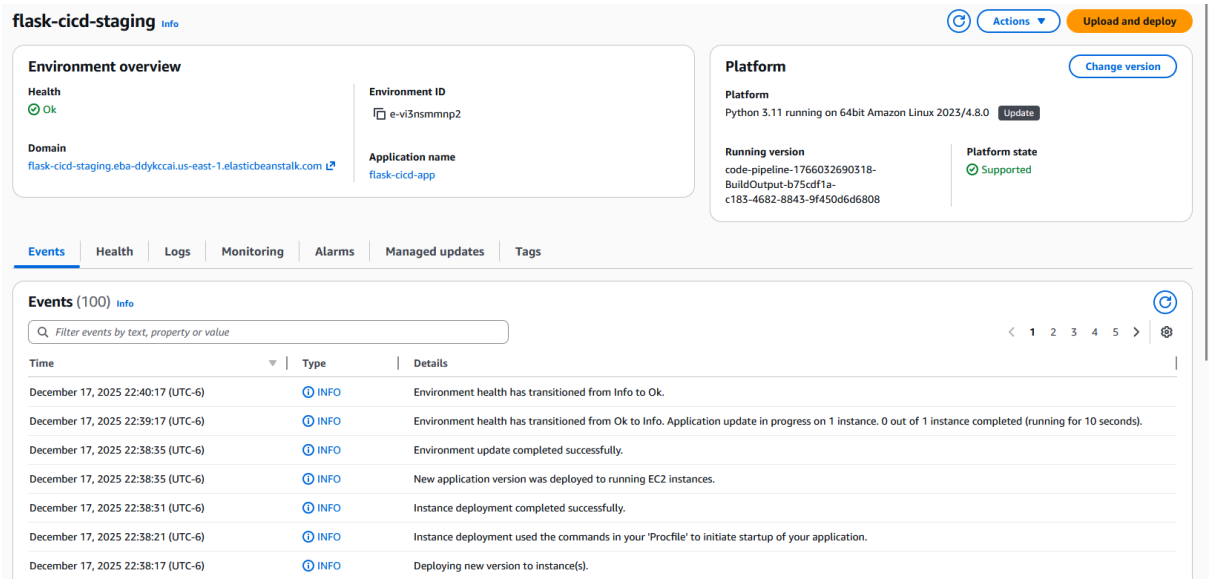


6.2 CodeBuild Logs

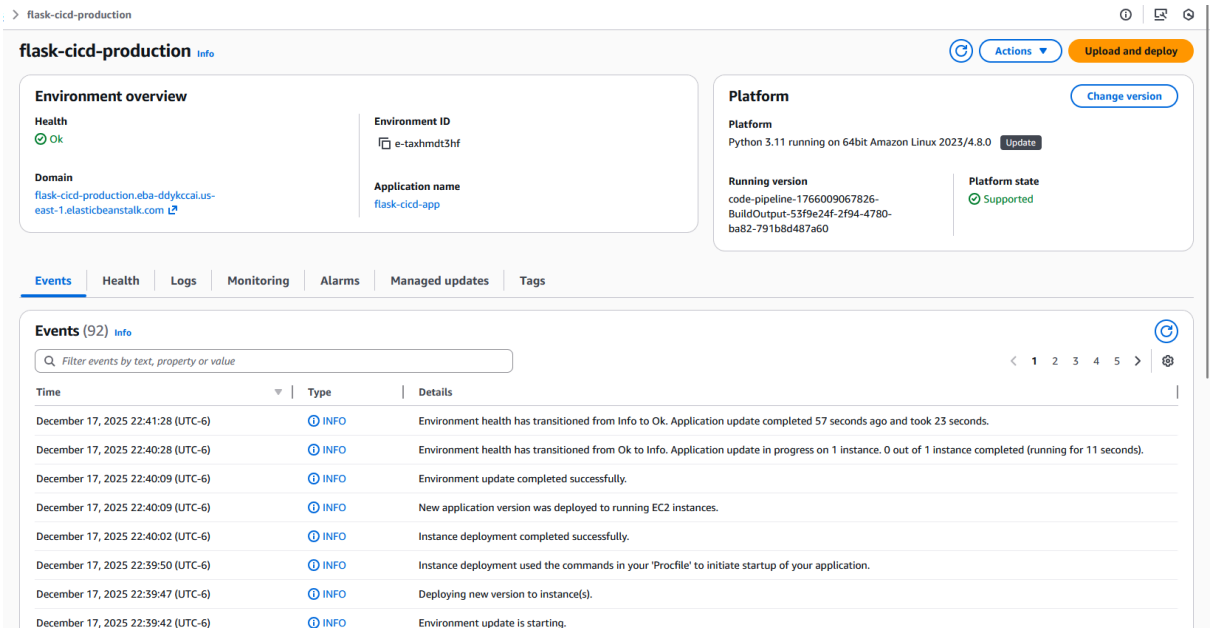


6.3 Elastic Beanstalk Deployment Logs

Staging Environment Events

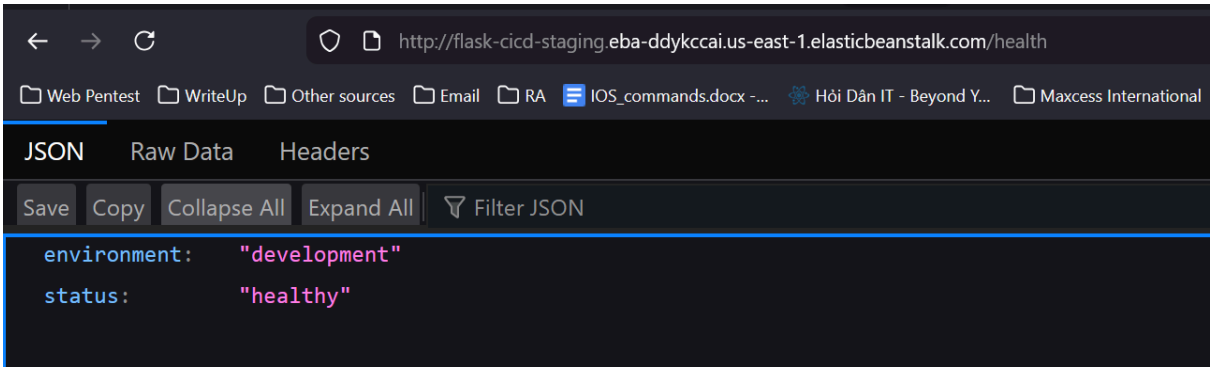


Production Environment Events:

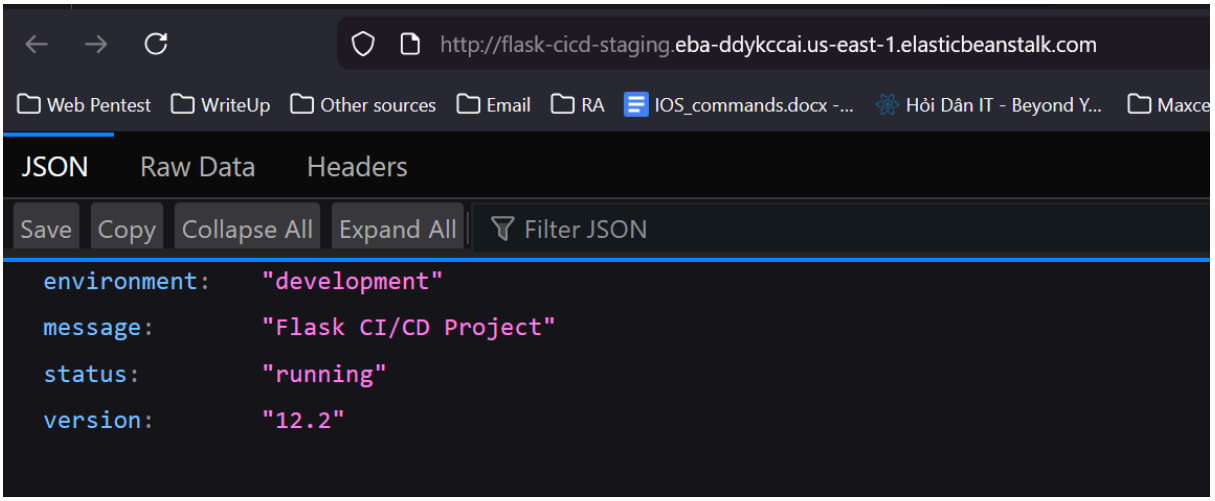


6.4 Application Health Check

Staging Environment Response

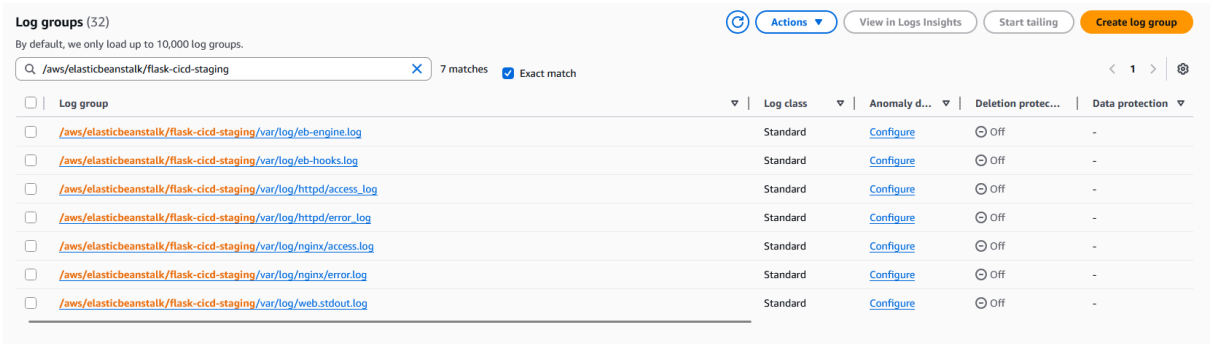


Application Version Check



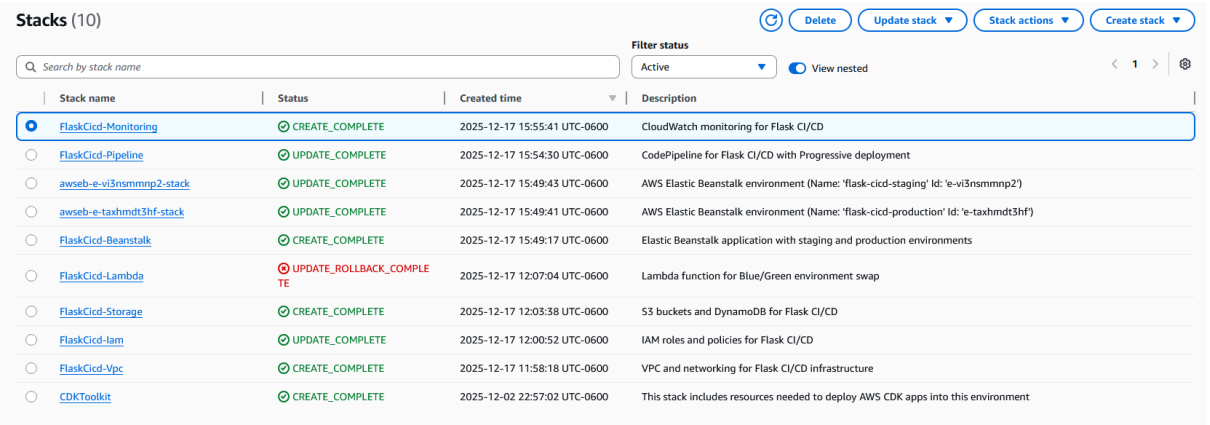
6.5 CloudWatch Logs

Application Logs



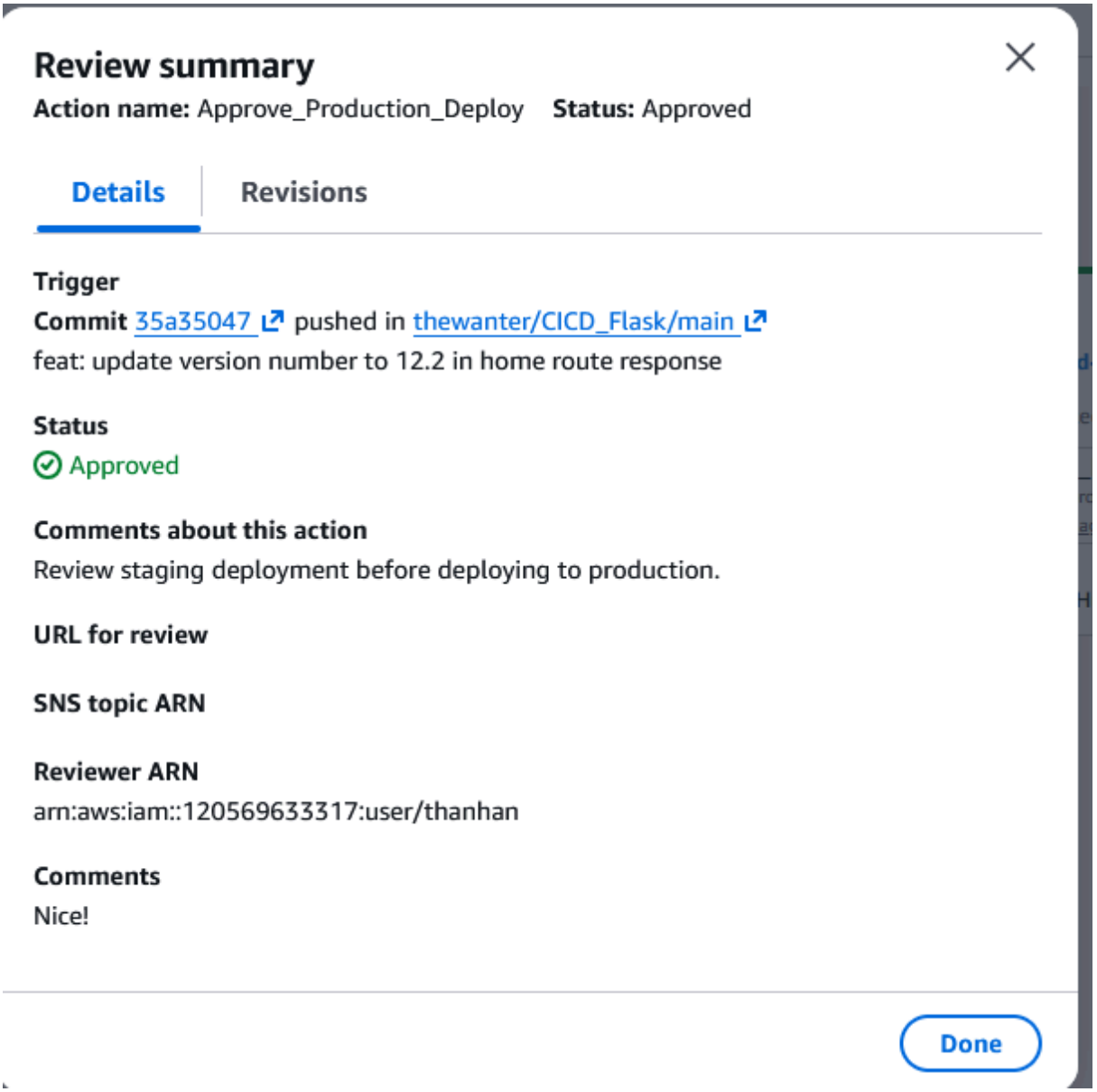
6.6 CDK Deployment Output

Pipeline Stack Deployment



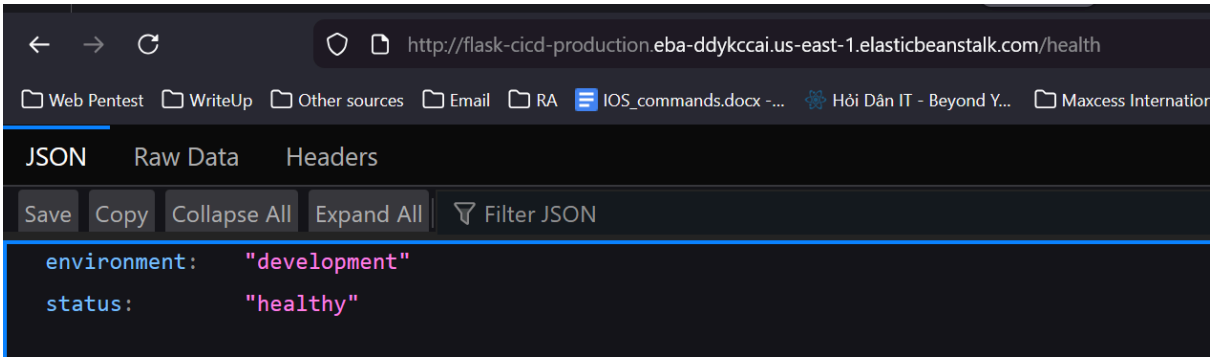
6.7. Manual Approval Action

Approval Comments Example

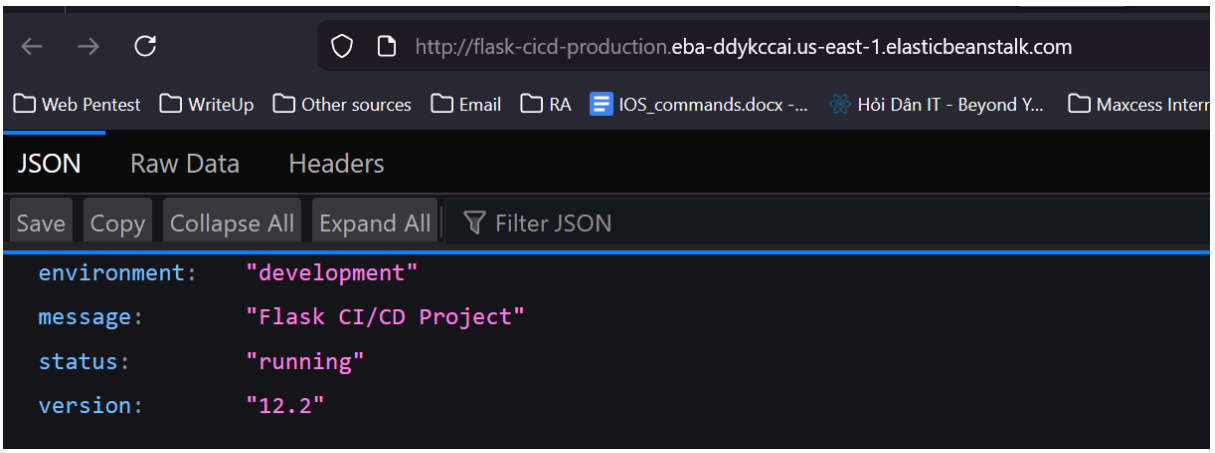


6.8. Production Deployment Success

Production Environment Response



Production Version Check



6. Challenges Encountered

BuildSpec File Not Found: CodeBuild failed because `buildspec.yml` existed locally but was never committed to GitHub. CodeBuild fetches from the repository, not the local machine. Fixed by committing and pushing the file to the repository root.

Directory Navigation: BuildSpec commands run in separate shell contexts, so cd flask-cicd-project followed by pip install lost the directory change. Fixed by chaining commands with `&&` to run in the same shell context.

Artifact Structure Mismatch: Elastic Beanstalk couldn't find requirements.txt because CodeBuild packaged the entire repository structure. Added base-directory: flask-cicd-project to the artifacts section so files appear at the root level where Elastic Beanstalk expects them.

SSM Permission Denied: CodeBuild couldn't retrieve the `SECRET_KEY` from SSM Parameter Store because CDK created a minimal permission role. Fixed by explicitly adding ssm:GetParameters policy to the CodeBuild role in pipeline-stack.ts.

Pipeline Conflict: Running manual and CDK pipelines simultaneously caused conflicts as both tried to deploy to the same Elastic Beanstalk environments. Fixed by disabling the manual pipeline and keeping only the CDK pipeline active.

Git Detached HEAD: Working from a detached `HEAD` state caused merge conflicts when trying to commit changes. Fixed by stashing changes with git stash, checking out main, and restoring with git stash pop. Always verify the current branch with git status before making changes.

Inline vs External BuildSpec: The CDK pipeline used an inline `buildspec` defined in TypeScript while the manual pipeline used an external file, causing different behaviors.

Changed CDK to use BuildSpec.fromSourceFilename `('buildspec.yml')` to unify both pipelines around a single version controlled configuration file.

Application Name Mismatch: The manual pipeline targeted flask-cicd-application while CDK created flask-cicd-app, causing deployment failures. Exported the manual pipeline configuration, modified the JSON to match the CDK application name, and updated the pipeline with aws codepipeline update-pipeline.

Build Artifacts in Git: After running `cdk synth` and `npm run build`, git status showed dozens of untracked files because no `.gitignore` existed. Created a comprehensive `.gitignore` excluding `cdk.out/`, compiled TypeScript files, `node_modules/`, and Python artifacts to keep the repository clean.

These challenges reinforced that debugging CI/CD pipelines requires understanding how each service processes inputs and outputs, reading error messages carefully, and understanding execution context.

7. Conclusion and Future Work

7.1 Conclusion

Problem Resolution

The project successfully addressed the core challenges I faced with manual AWS deployments. What previously required hours of console navigation and was prone to configuration drift is now a repeatable, version-controlled process. A single cdk deploy --all command provisions the entire infrastructure, and code changes flow automatically from GitHub to production through the CI/CD pipeline. The business problem of slow, error-prone deployments has been effectively solved.

Major Achievements

The project delivered several key outcomes. First, I built a fully functional 5-stage CI/CD pipeline that automates testing, staging deployment, and production release with Blue/Green strategy. Second, I migrated all infrastructure to AWS CDK, creating modular stacks that can be reused and extended. Third, I implemented proper security practices including IAM least-privilege roles, encrypted secrets in Parameter Store, and network isolation through VPC. Finally, I established monitoring through CloudWatch dashboards that provide visibility into pipeline and application health.

Architecture Alignment

The final architecture meets all initial requirements. The pipeline triggers automatically on code push, runs automated tests before deployment, and includes a manual approval gate

before production. Blue/Green deployment ensures zero downtime, and the Lambda-based URL swap enables instant rollback if needed. Infrastructure as Code means the entire setup is documented, reproducible, and auditable through Git history.

Skills Gained

This project significantly expanded my practical AWS knowledge. I gained hands-on experience with CDK and Infrastructure as Code principles, understanding how to structure modular stacks and manage cross-stack dependencies. I learned to debug CI/CD pipelines by reading CloudWatch logs and understanding how each service processes artifacts. I developed troubleshooting skills for IAM permissions, recognizing that most "access denied" errors trace back to missing policy statements. Most importantly, I learned to think systematically about deployment workflows and how services integrate to form a complete solution.

7.2 Future Work

Production Hardening

For a true production deployment, I would add multi-AZ database support with Amazon RDS, implement auto-scaling policies based on CPU and request metrics, and configure health check alarms that trigger automatic rollbacks. The current single NAT Gateway would be expanded to one per Availability Zone for higher availability.

Enhanced Monitoring and Notifications

The next iteration would include SNS notifications for pipeline failures and successful deployments, sending alerts to Slack or email. I would add detailed application metrics using CloudWatch custom metrics and create alarms for error rate thresholds, latency spikes, and resource utilization.

Cost Optimization

To reduce costs in non-production environments, I would implement scheduled scaling to shut down staging instances outside business hours. Using Spot Instances for build environments and implementing S3 Intelligent-Tiering for artifact storage would further reduce expenses. A cost allocation dashboard would help track spending by component.

Security Enhancements

Future improvements would include AWS WAF integration to protect against common web attacks, AWS Secrets Manager for automatic secret rotation, and VPC endpoints to keep traffic within the AWS network. I would also implement AWS Config rules to detect security misconfigurations automatically.

Additional Capabilities

The pipeline could be extended to support multiple environments beyond staging and production, enabling feature branch deployments for testing. Integration with AWS X-Ray would provide distributed tracing for debugging performance issues. Adding container support through ECS or EKS would allow deploying more complex microservices architectures.

AI/ML Integration

A natural extension would be integrating Amazon CodeGuru for automated code reviews during the build phase, identifying potential bugs and security vulnerabilities before deployment. CloudWatch Anomaly Detection could automatically identify unusual patterns in application metrics without manually defining thresholds.