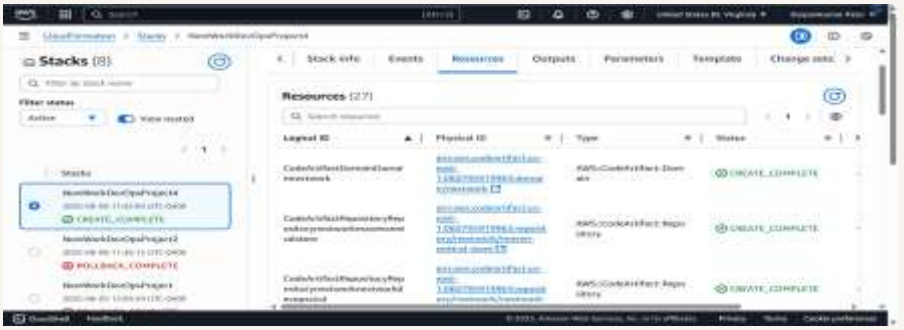
Darpan Patel

Infrastructure as Code with CloudFormation



Darpan Patel

Introducing Today's Project!

In this project, I will demonstrate how to convert my entire CI/CD pipeline infrastructure into code using AWS CloudFormation. I'm doing this project to learn how to automate the provisioning of AWS resources like EC2 instances, CodeBuild projects, CodeDeploy deployment groups, and S3 buckets using Infrastructure as Code (IaC) principles. This approach not only helps eliminate manual setup errors but also ensures consistent, repeatable environments that can be deployed across multiple stages or accounts. It’s a crucial step toward mastering DevOps automation and working more efficiently in cloud-based production environments.

Key tools and concepts

Services I used were **AWS CloudFormation** for infrastructure as code, **AWS CodeBuild** for continuous integration builds, **AWS CodeDeploy** for automated deployment, **Amazon S3** for storing build artifacts, and **CloudWatch Logs** for monitoring build logs. Key concepts I learnt include how to define and parameterize CloudFormation templates to create reusable and flexible infrastructure, how to manually add critical CI/CD resources like CodeBuild projects and CodeDeploy deployment groups, how to manage dependencies between resources with `DependsOn`, and how to validate and deploy infrastructure changes using CloudFormation stacks. This project also reinforced best practices for automating and streamlining the software delivery pipeline using AWS managed services.

Darpan Patel

Project reflection

This project took me approximately **2 to 3 hours** to complete. The most challenging part was carefully editing the CloudFormation template to correctly add the CodeBuild and CodeDeploy resources while ensuring all resource references and dependencies were accurate.

It was most rewarding to see the entire CI/CD pipeline infrastructure come together seamlessly in AWS, knowing that the template I enhanced can now be reused and shared for consistent deployments.

This project is part six of a series of DevOps projects where I'm building a CI/CD pipeline! I'll be working on the next project soon—probably within the next few days— to continue enhancing my skills and complete the full pipeline setup. I’m excited to keep progressing and apply what I’ve learned so far to even more advanced DevOps tasks.

Generating a CloudFormation Template

The IaC Generator is a tool within AWS CloudFormation that helps you automatically create Infrastructure as Code (IaC) templates by analyzing your existing AWS resources. It works in a three-step process where: 1. **You scan your AWS account** – The generator discovers all the current resources in your account (like EC2 instances, IAM roles, S3 buckets, etc.). 2. **You**

Darpan Patel

select and group resources into a template – You bundle the scanned resources you want to manage together into a reusable CloudFormation template. 3. **You import the template into CloudFormation** – This lets you deploy and manage those resources as code, enabling consistent and repeatable infrastructure deployments. This tool is especially useful because it eliminates the need to manually write complex CloudFormation code from scratch, speeds up the development process, and reduces human errors in infrastructure setup.

A CloudFormation template is a YAML or JSON file that defines AWS infrastructure as code. It enables automated, consistent, and repeatable provisioning of cloud resources, reducing manual setup and minimizing errors. Templates consist of resources like compute, storage, networking, and IAM components. They are crucial for managing infrastructure lifecycles and supporting DevOps practices such as

CI/CD. The resources I could add to my template include:

AWS::CodeArtifact::Domain and **Repository** for managing packages.

AWS::S3::Bucket for storing build artifacts.

AWS::CodeStarConnections::Connection to link GitHub.

AWS::CodeDeploy::Application to handle deployments.

AWS::IAM::InstanceProfile, Policy, and **Role** to manage permissions and access for services like EC2 and CodeBuild. These resources reflect the core infrastructure behind a CI/CD pipeline, making deployments fast, consistent, and scalable.

Darpan Patel

The resources I couldn’t add to my template were **AWS::CodeBuild::Project** and **AWS::CodeDeploy::DeploymentGroup**, because the IaC Generator cannot automatically detect or generate these resources due to their complex configurations and dependencies. For example, a CodeBuild project requires detailed environment settings like compute type, runtime versions, buildspec files, and encryption keys. Similarly, a CodeDeploy deployment group needs explicit configurations for deployment types, target EC2 instances or Auto Scaling groups, and alarm settings. These require additional security permissions and custom parameters that aren't easily inferred through a scan. While the IaC Generator simplifies template creation for many standard resources, these advanced, highly configurable components must be added **manually** to ensure accuracy and completeness in deployment automation.

Darpan Patel

Template Testing

Before testing my template, I deleted all the existing CI/CD-related resources— including the CodeDeploy application, CodeBuild project, CodeArtifact repository and domain, S3 bucket, EC2 instance, IAM roles, and policies—because CloudFormation cannot create resources that already exist with the same names. Removing them ensured that my stack would deploy cleanly without conflicts or errors. It also helped me maintain a tidy AWS environment and prevent duplicate charges.

I tested my template by deploying it through the AWS CloudFormation console after cleaning up my existing resources. The result of my first test was successful because the template was well-structured, the dependencies were resolved, and only supported, valid resources were included. This confirmed that the infrastructure-ascode approach worked as expected and could reliably recreate my CI/CD setup from scratch.

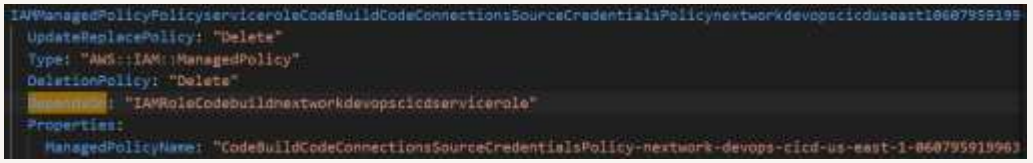
Darpan Patel

DependsOn

To resolve the error, I added the `DependsOn` attribute to each IAM policy in my CloudFormation template. The `DependsOn` attribute means CloudFormation will create the specified IAM role before attempting to create and attach the IAM managed policies to it. This fixed the issue because the policies were previously being deployed before the IAM role existed, which caused the deployment to fail. By explicitly defining this dependency, I ensured the resources were created in the correct order, allowing the stack to deploy successfully.

Darpan Patel

The `DependsOn` line was added to four different parts of my CloudFormation template: 1. **IAMManagedPolicy00policyservicerole00** – for the CodeBuild service role.
2. **IAMManagedPolicy01policyservicerole00** – for the CloudWatch Logs policy. 3. **IAMManagedPolicy02policyservicerole00** – for the CodeConnections source credentials policy. 4. **IAMManagedPolicy03policyservicerole00** – for the CodeBuild base policy. For the CodeArtifact policy (IAMManagedPolicy04policyec2instance00), I added a `DependsOn` line that references the IAM role attached to my EC2 instance, ensuring that role is created before the policy is attached. This sequence avoids dependency errors during stack creation.



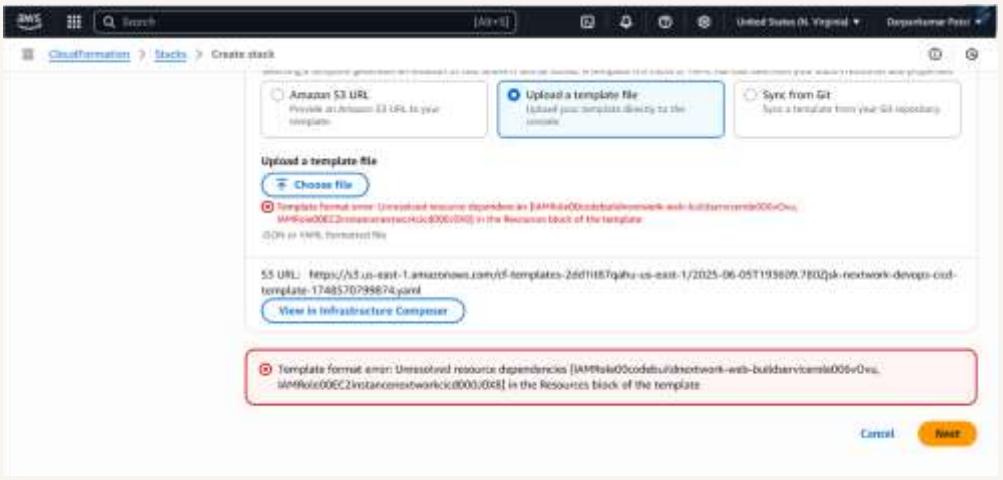
Darpan Patel

Circular Dependencies

I gave my CloudFormation template another test! But this time, I ran into a **circular dependency error**. This happened because my IAM roles and IAM policies were referencing each other in a loop — the policies depended on the roles, and the roles also referenced the policies. CloudFormation couldn't figure out which one to create first, leading to a deadlock in the stack creation process.

To fix this error, I opened my CloudFormation template and removed the `ManagedPolicyArns` lines from the IAM role configurations. These lines were referencing IAM policies that already depended on the IAM role, which created a circular dependency. By removing those references, I broke the loop and allowed CloudFormation to create the IAM role and policies in the correct order. This resolved the error and enabled the template to proceed with stack creation.

Darpan Patel



Manual Additions

In a project extension, I manually defined two more resources: 1. **AWS::CodeBuild::Project** – This resource sets up a CodeBuild project that pulls code from a GitHub repository, uses a `buildspec.yml` file to define the build steps, and stores build artifacts in an S3 bucket. It also configures CloudWatch for log streaming and uses an Amazon Linux environment with Corretto 8 for Java builds. 2**. AWS::CodeDeploy::DeploymentGroup** –

Darpan Patel

This resource creates a deployment group linked to a CodeDeploy application. It specifies deployment settings such as the "AllAtOnce" deployment strategy and targets EC2 instances with a specific g (`role: webserver`). It also uses an IAM service role to execute the deployment actions securely. These additions enhanced the CI/CD pipeline by enabling automated builds and controlled deployments directly from the CloudFormation template.

I also had to make sure the references were consistent in this template, so I edited the CodeBuild project definition by replacing the placeholder values with actual logical resource IDs used elsewhere in the template. Specifically: Replaced `<YOUR_CODEBUILD_SERVICE_ROLE_ID>` with the correct logical ID of the IAM role assigned to CodeBuild (e.g., `CodeBuildServiceRole`). Replaced `<YOUR_S3_BUCKET_ID>` with the actual logical ID of the S3 bucket used for storing build artifacts (e.g., `BuildArtifactBucket`). Updated the `ServiceRole` property to use `!GetAtt CodeBuildServiceRole.Arn` for correct permission reference. Updated the `Location` under `Artifacts` to use `!Ref BuildArtifactBucket` so the output artifacts are correctly stored. These edits ensure that the CodeBuild project integrates properly with other resources defined in the template and that CloudFormation can resolve all references without errors.

I also introduced **Parameters**, which are user-defined inputs that make a CloudFormation template more **flexible and reusable**. Instead of hardcoding values like GitHub usernames or repository names, parameters allow you to pass in these values at deployment time. This means the same template can be used across different environments

or projects simply by changing the input values—without modifying the actual code. Parameters are a best practice in infrastructure as code because they promote **modularity, maintainability, and scalability**.

Success!

I could verify all the deployed resources by visiting the Resources tab in the CloudFormation console. This tab provides a complete list of all the AWS resources created by the stack, along with direct hyperlinks to each resource where available.

Darpan Patel