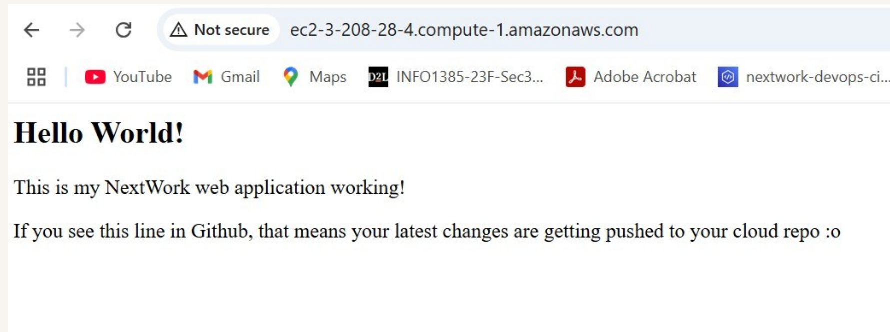
Darpan Patel

Deploy a Web App with CodeDeploy



Darpan Patel

Introducing Today's Project!

In this project, I will demonstrate how to deploy a Java-based web application to an EC2 instance using AWS CodeDeploy. I'm doing this project to learn how to automate deployments as part of a CI/CD pipeline, minimize downtime during releases, and handle deployment failures using rollback strategies. This hands-on experience will strengthen my understanding of deployment automation and help me build production-ready DevOps skills using services like EC2, CodeDeploy, CodeBuild, CodeArtifact, and CloudFormation.

Key tools and concepts

Services I used were AWS CodeCommit, CodeBuild, CodeDeploy, and S3. These services helped manage version control, build automation, deployment automation, and artifact storage. Key concepts I learnt include continuous integration and deployment (CI/CD), how to write and use AppSpec files for lifecycle hooks, the importance of rollback and recovery strategies in real-world deployments, and how failures in lifecycle scripts can impact an entire deployment. I also learned how CodeDeploy behaves during rollbacks and that it doesn’t restore previous artifacts unless integrated with a more advanced setup like CodePipeline. This project reinforced the importance of automation, versioned artifacts, and clear error handling in DevOps workflows.

Project reflection

This project took me approximately 1.5 to 2 hours to complete. The most challenging part was troubleshooting the intentional error and understanding why the automatic rollback didn’t recover the last working state—especially realizing that CodeDeploy reuses the deployment configuration, not the previous artifact.

Darpan Patel

It was most rewarding to manually recover from a failed deployment by fixing the code, rebuilding the artifact, and successfully redeploying it, which gave me handson experience with real-world DevOps problem-solving.

This project is part five of a series of DevOps projects where I'm building a CI/CD pipeline! I'll be working on the next project within the next day or two to keep up the momentum and continue developing hands-on expertise with AWS DevOps tools. Each project builds on the last, so I'm excited to dive into the next phase, which will likely focus on enhancing automation, testing, or deployment strategies. Staying consistent with this series is helping me master real-world DevOps workflows step by step.

Deployment Environment

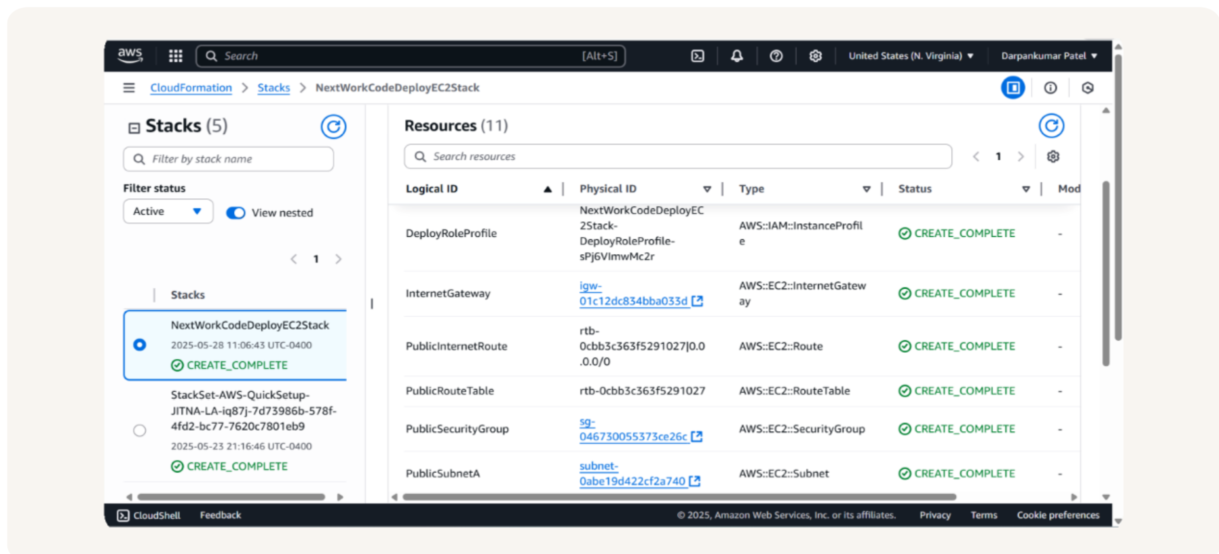
To set up for CodeDeploy, I launched an EC2 instance and VPC because we need a secure environment to run our application in production. The EC2 instance is where the app will be deployed, and the VPC provides the network infrastructure to control traffic and secure access. Using CloudFormation ensures the setup is automated, consistent, and easy to manage with Infrastructure as Code.

Instead of launching these resources manually, I used AWS CloudFormation to deploy my EC2 instance and VPC. When I need to delete these resources, I can simply delete the CloudFormation stack, and it will automatically remove everything it created—making management much easier and more efficient.

Other resources created in this template include a Virtual Private Cloud (VPC), subnet, route tables, internet gateway, and security group. These are essential components for building a secure and functional environment for the EC2 instance. The VPC acts as an isolated network for the instance. The subnet is a smaller segment within the VPC that allows better organization and routing. The route tables manage how traffic is directed within the VPC. The internet gateway connects the VPC to the internet, enabling external access. The security group acts as a virtual firewall to control traffic

Darpan Patel

to and from the EC2 instance. They're also in the template because having these defined as code ensures that every environment you create is consistent, secure, and reproducible. This level of control and automation is especially important for deploying web applications that may need strict access rules or public connectivity.



appspec.yml

Then, I wrote an **appspec.yml** file to serve as the deployment blueprint for AWS CodeDeploy, guiding how the application should be installed and managed during deployment. The key sections in **appspec.yml** are: **version:** Specifies the format version that CodeDeploy expects. **os:** Indicates the target operating system, in this case, Linux. **files:** Defines which files to copy from the build artifact to the target destination on the server. **hooks:** Lists scripts to run at specific deployment lifecycle events (like BeforeInstall, ApplicationStop, ApplicationStart) to automate setup, starting, and stopping of services. This file orchestrates the entire deployment process, ensuring that the right files are placed correctly and the necessary scripts run in the proper sequence.

I also updated buildspec.yml because after the build process, I needed to ensure that the deployment package included not just the compiled application (the WAR file) but

Darpan Patel

also the appspec.yml file and all the deployment scripts inside the scripts folder. This way, AWS CodeDeploy would have all the necessary files to correctly deploy and configure the application on the target servers. Specifically, I modified the artifacts section to add appspec.yml and scripts/*/ so these resources are bundled with the build output.

```
phases:
  pre_build:

    build:
    commands:
      - echo "====== BEGINNING TEST PHASE ======"
      - echo "Running tests on `date`"
      - chmod +x run-tests.sh
      - ./run-tests.sh
      - echo "====== TEST PHASE COMPLETE ======"
      - echo "====== BEGINNING BUILD PHASE ======"
      - echo "Build started on `date`"
      - mvn -s settings.xml compile
      - echo "====== BUILD PHASE COMPLETE ======"

  post_build:
    commands:
      - echo "Build completed on `date`"
      - mvn -s settings.xml package

artifacts:
  files:
    - target/nextwork-web-project.war
    - appspec.yml
    - scripts/**/*
  discard-paths: no
```

Setting Up CodeDeploy

A deployment group is a collection of EC2 instances or on-premises servers that are targeted together for deploying an application. It defines where, how, and with what settings the deployment should happen, such as deployment strategy, load balancing, and rollback options. A CodeDeploy application is a higher-level container or namespace that organizes all the deployment configurations, deployment groups, and revisions related to a specific software application. It acts as the main project folder under which one or more deployment groups operate.

Darpan Patel

To set up a deployment group, you also need to create an IAM role to grant CodeDeploy the necessary permissions to interact with your AWS resources securely. This IAM role allows CodeDeploy to: Access and deploy your application onto EC2 instances. Read from S3 buckets to retrieve the application bundle. Interact with Auto Scaling groups during blue/green deployments. Temporarily remove instances from load balancers during deployments. Send logs and metrics to CloudWatch for monitoring deployment status. Without this IAM role, CodeDeploy wouldn't have the authorization it needs to perform these tasks. The role ensures operations are executed under the principle of least privilege, meaning CodeDeploy only gets access to the specific resources it needs—nothing more, nothing less.

Tags are helpful for dynamically and accurately identifying the target EC2 instances during deployment.

I used the tag role-webserver to tell CodeDeploy exactly which instance (s) should receive the application deployment. This approach offers several advantages: Flexibility: If I launch new instances with the same tag in the future, CodeDeploy will automatically include them in deployments. Clarity: Tags like role=webserver clearly communicate the instance's purpose, making it easier to manage and understand resources. Integration: Since my CloudFormation template already assigned this tag, everything connects seamlessly without extra manual setup. Using tags ensures that my deployments are targeted, scalable, and maintainable.

Developer Tools
CodeDeploy
Source • CodeCommit
Artifacts • CodeArtifact
Build • CodeBuild
Deploy • CodeDeploy
Getting started
Deployments
Applications
Application
Settings
Deployment configurations
On-premises instances
Pipeline • CodePipeline
Settings

Select any combination of Amazon EC2 Auto Scaling groups, Amazon EC2 instances, and on-premises instances to add to this deployment
Amazon EC2 Auto Scaling groups
Amazon EC2 instances
1 unique matched instance. Click here for details
You can add up to three groups of tags for EC2 instances to this deployment group.
One tag group: Any instance identified by the tag group will be deployed to.
Multiple tag groups: Only instances identified by all the tag groups will be deployed to.
Tag group 1
Key | Value - optional
role | webserver | Remove tag
Add tag
Add tag group
On-premises instances
Matching instances
1 unique matched instance. Click here for details

CloudShell Feedback © 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences

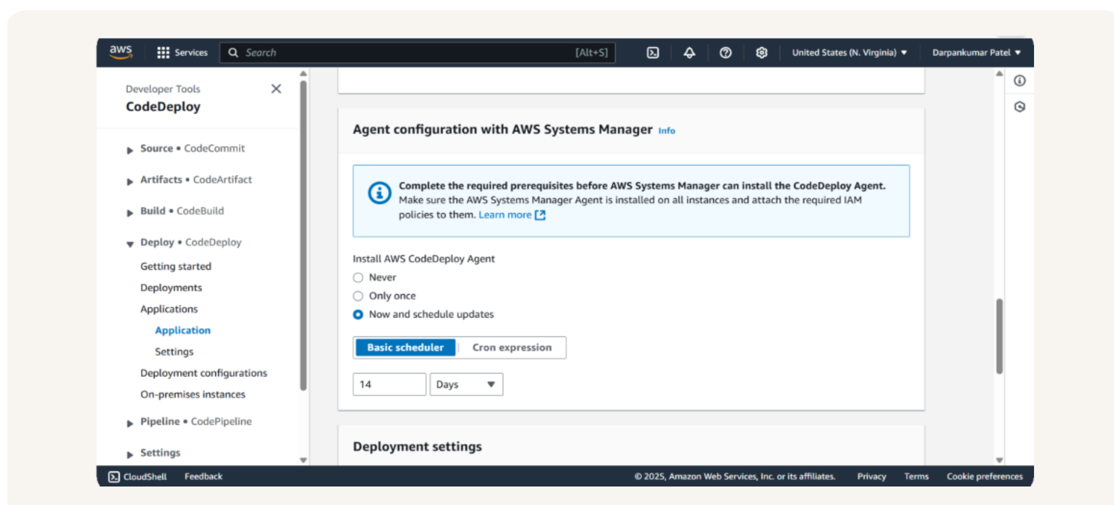
Darpan Patel

Deployment configurations

Another key setting is the deployment configuration, which affects how CodeDeploy rolls out new application versions across EC2 instances.

I used CodeDeployDefault.AllAtOnce, so all instances get updated at the same time. It's fast but risky—if something fails, every instance is affected. Other options include: CodeDeployDefault.OneAtATime: updates one instance at a time, ensuring safety but slower deployment. CodeDeployDefault.HalfAtATime: deploys to 50% of instances first, then the rest if successful. You can also create custom configs with specific instance percentages or success thresholds. These options are vital in production to minimize downtime and errors. But for a single-instance learning project like ours, AllAtOnce is the simplest and fastest approach.

In order to connect your EC2 instances with the CodeDeploy service, a CodeDeploy Agent is also set up to run on each instance. This agent acts as the bridge between AWS CodeDeploy and your actual server. It’s responsible for: Receiving deployment instructions from CodeDeploy (like scripts or application packages). Executing those instructions to install, update, or remove application code. Sending back status updates and logs to help you monitor the deployment process. Without the CodeDeploy Agent running on your instances, deployments won’t work—because CodeDeploy would have no way to interact with the server. Keeping this agent updated ensures compatibility with AWS and avoids deployment errors.



Darpan Patel

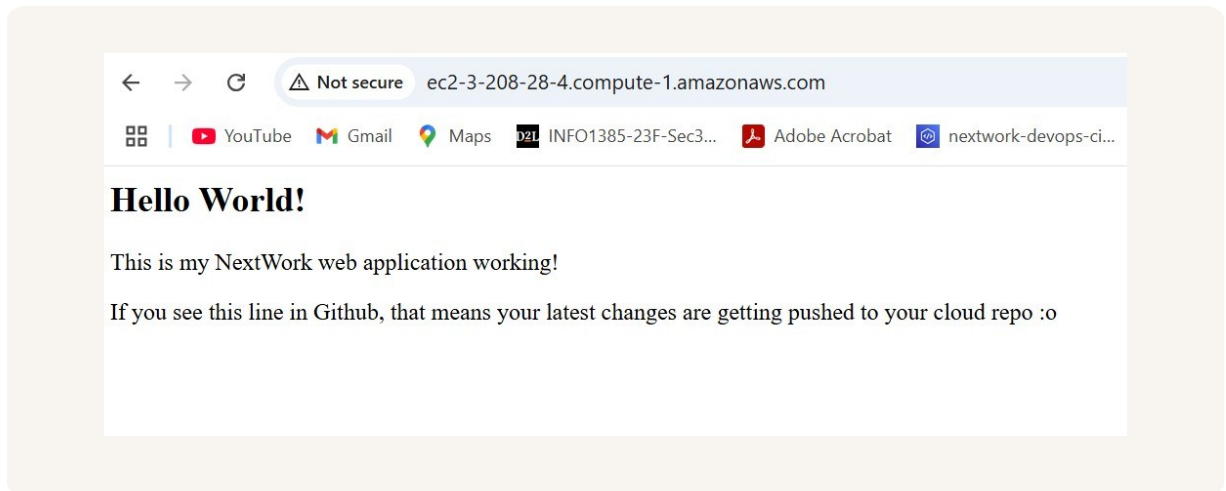
Success!

A CodeDeploy deployment is a specific instance of updating your application with a chosen version (revision) to the target environment. It includes instructions on what to deploy, where to deploy it (the deployment group), and how to carry out the deployment according to the configured settings. The difference to a deployment group is that a deployment group is a logical set of instances or targets where deployments are performed, while a deployment is the actual execution of delivering and installing the application revision on those targets.

I had to configure a revision location, which means specifying where CodeDeploy should retrieve the application files for deployment. My revision location was an Amazon S3 URI pointing to a .zip file that contained the packaged build artifact — including the WAR file and deployment scripts. This setup ensures that CodeDeploy can access the correct version of my application and execute the deployment steps on the EC2 instance.

To check that the deployment was a success, I visited the Public IPv4 DNS of my EC2 instance in a web browser. I saw my web application running as expected, confirming that CodeDeploy successfully deployed the latest build artifact from S3, executed the lifecycle events defined in appspec.yml, and placed the WAR file into the correct Tomcat directory. Additionally, in the CodeDeploy console, the deployment status showed "Succeeded", and each lifecycle event (like BeforeInstall, ApplicationStart) was marked as Completed without errors. This confirmed that the deployment process executed smoothly end to end.

Darpan Patel



When my deployment failed, the automatic rollback didn't work because CodeDeploy reused the latest broken build artifact instead of reverting to a previous working version. To actually recover from the failure, I had to manually fix the script, push the changes to GitHub, trigger a new build in CodeBuild, and redeploy the fixed version. In production environments, automatic rollbacks can be implemented using AWS CodePipeline. CodePipeline allows rollback to a previously successful artifact by integrating with CloudWatch Alarms and setting up failure actions.

This enables automatic detection of failures in deployments and triggers a rollback to a known-good version stored in S3. You can also include approval steps, automated tests, and health checks to prevent broken code from reaching production. Unlike CodeDeploy alone, CodePipeline's versioned artifact support ensures a reliable recovery path, minimizing downtime and manual intervention during deployment failures.