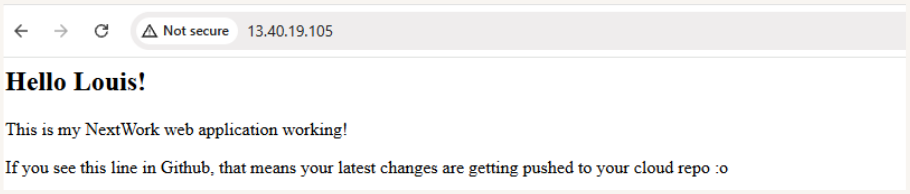
nextwork.org

Deploy a Web App with CodeDeploy

Louis Moyo



Louis Moyo
NextWork Student

Introducing Today's Project!

In this project, I will demonstrate how to use AWS CodeDeploy to automate the deployment of an application. I'm doing this project to learn how continuous deployment works in practice, how to reduce manual steps, and how AWS services help ensure consistent, reliable, and error-free rollouts.

Key tools and concepts

Services I used were AWS CodeDeploy, CodeBuild, CodeArtifact, CloudFormation, EC2, S3, IAM, and GitHub. Key concepts I learnt include deployment automation, lifecycle event hooks in appspec.yml, rollback and recovery strategies, Infrastructure as Code with CloudFormation, tagging for targeting instances, and how the CodeDeploy agent orchestrates deployments. I also deepened my understanding of how CI/CD services connect together into a full pipeline.

Project reflection

This project took me approximately 4 hours. The most challenging part was debugging deployment failures at the ApplicationStop stage and making sure my scripts were packaged correctly into the S3 artifact. The most rewarding part was seeing my web application successfully deployed to the production EC2 instance via CodeDeploy, proving that my CI/CD pipeline works end-to-end.

L

Louis Moyo
NextWork Student

nextwork.org

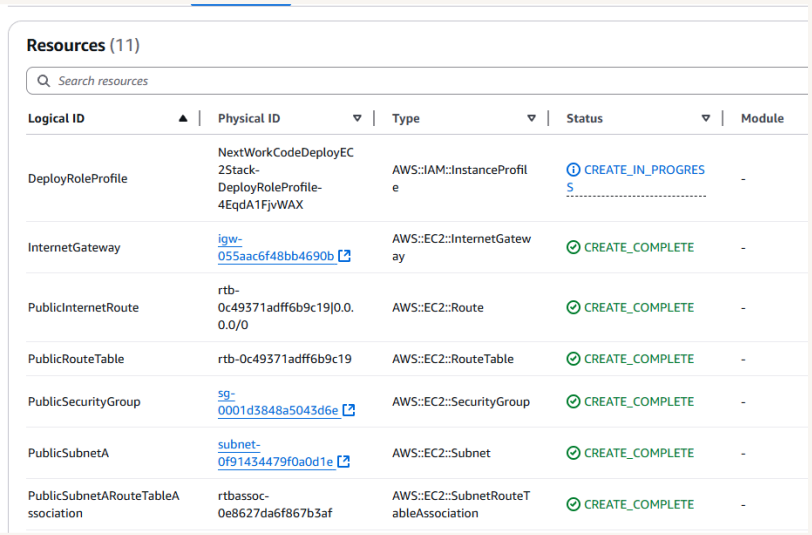
This project is part five of a 7-day DevOps challenge where I'm building a CI/CD pipeline step by step. I'll be working on the next project tomorrow, which continues the series and adds further automation and best practices to the pipeline.

Deployment Environment

To set up for CodeDeploy, I launched an EC2 instance and VPC because my application needs a secure, isolated environment to run in production. The EC2 instance acts as the server where my app will be deployed, and the VPC provides the networking, firewall, and routing rules to control access. Having a dedicated instance and VPC ensures security, stability, and separation from my development environment.

Instead of launching these resources manually, I used AWS CloudFormation. CloudFormation lets me define my infrastructure as code, so the EC2 instance, security groups, and networking setup are created consistently every time. When I need to delete these resources, I can simply delete the CloudFormation stack, and AWS will automatically remove all related resources in one action.

Other resources created in this template include a VPC, subnet, route tables, internet gateway, security group, and an EC2 instance. They're also in the template because an EC2 instance on its own cannot function in isolation - it needs networking resources to define how traffic flows, a security group to control access, and a VPC with routing to connect safely to the internet. By bundling these together, we ensure our deployment is secure, reproducible, and production-ready.



Resources (11)

Search resources

Logical ID	Physical ID	Type	Status	Module
DeployRoleProfile	NextWorkCodeDeployEC2Stack-DeployRoleProfile-4EqdA1FjvWAX	AWS::IAM::InstanceProfile	CREATE_IN_PROGRESS	-
InternetGateway	igw-055aac6f48bb4690b	AWS::EC2::InternetGateway	CREATE_COMPLETE	-
PublicInternetRoute	rtb-0c49371adff6b9c19|0.0.0.0/0	AWS::EC2::Route	CREATE_COMPLETE	-
PublicRouteTable	rtb-0c49371adff6b9c19	AWS::EC2::RouteTable	CREATE_COMPLETE	-
PublicSecurityGroup	sg-0001d3848a5043d6e	AWS::EC2::SecurityGroup	CREATE_COMPLETE	-
PublicSubnetA	subnet-0f91434479f0a0d1e	AWS::EC2::Subnet	CREATE_COMPLETE	-
PublicSubnetARouteTableAssociation	rtbassoc-0e8627da6f867b3af	AWS::EC2::SubnetRouteTableAssociation	CREATE_COMPLETE	-

Deployment Scripts

Scripts are small programs written in text files that contain a list of commands for the computer to run automatically. Instead of typing commands one by one, scripts let us bundle them together and execute them in sequence. To set up CodeDeploy, I also wrote scripts to install dependencies, start and stop the server, and configure services so that CodeDeploy knows exactly how to deploy and manage the application.

The install_dependencies.sh script prepares the environment for our application by installing and configuring the required software. It installs Tomcat (the Java application server) and Apache (the HTTP web server), then sets Apache up as a reverse proxy that forwards incoming requests to Tomcat. This ensures visitors can access our application through Apache while Tomcat handles the actual app logic behind the scenes.

The start_server.sh script ensures that both Tomcat (the Java application server) and Apache (the web server) are started and kept running on the EC2 instance. It uses systemctl to start each service immediately and also enables them to restart automatically whenever the server reboots. This guarantees that our application stays accessible to users without requiring manual intervention after restarts or crashes.

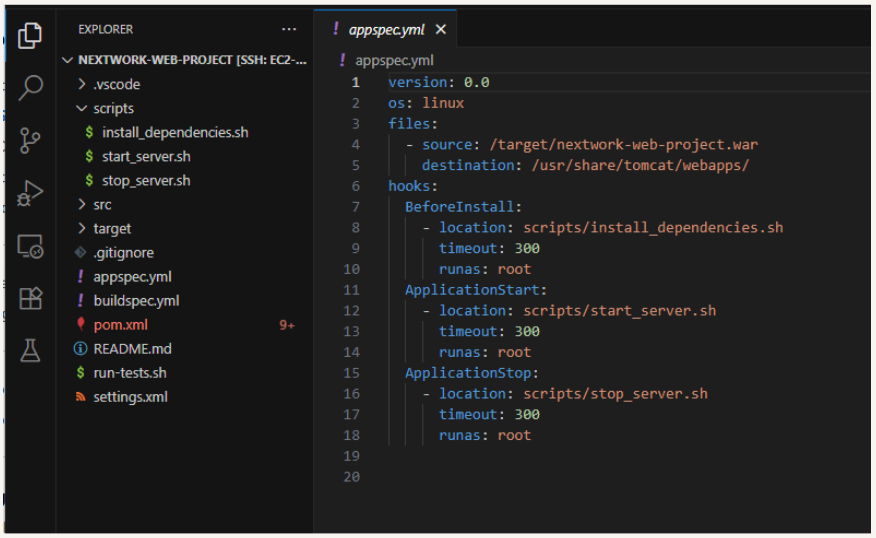
The stop_server.sh script safely shuts down the web server services running on our EC2 instance. It first checks if Apache (httpd) and Tomcat are active using pgrep, and only issues stop commands if they are running. This prevents unnecessary errors from trying to stop inactive services.

By handling services gracefully, the script makes our deployment process more reliable and avoids failures caused by unexpected service states.

appspec.yml

Then, I wrote an appspec.yml file to tell CodeDeploy how to handle deployments. It defines the app version and OS, specifies which files to copy, and maps lifecycle events (BeforeInstall, AfterInstall, ApplicationStart, ApplicationStop) to our scripts. This ensures deployments run in the correct order and stay consistent every time.

I also updated buildspec.yml because CodeDeploy needs more than just the compiled Java application to deploy correctly. In the artifacts section, I added the appspec.yml file and the scripts folder so they are included in the build output. This is important because appspec.yml defines the deployment instructions and the scripts handle installation, starting, and stopping the server. Without these, CodeDeploy wouldn't know how to complete the deployment.



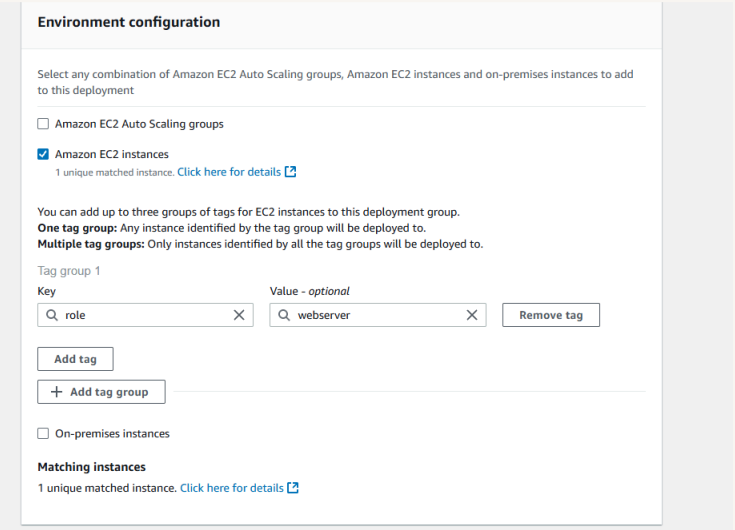
L
Louis Moyo
NextWork Student

Setting Up CodeDeploy

A CodeDeploy application is like a container or namespace that represents the overall software project you want to deploy. It organises all the deployment configurations, groups, and revisions for that app. A deployment group is a subset within that application that defines where and how the app gets deployed - for example, which EC2 instances to target, which IAM role to use, and which deployment strategy (all-at-once, rolling, blue/green). The application is the big picture, while deployment groups are the specific environments where deployments actually happen.

To set up a deployment group, you also need to create an IAM role so that CodeDeploy has permission to interact with your AWS resources on your behalf. This role allows CodeDeploy to safely perform tasks like deploying code to EC2 instances, accessing artifacts stored in S3, adjusting Auto Scaling groups, managing load balancer connections, and sending logs to CloudWatch. Without an IAM role, CodeDeploy wouldn't have the authority to coordinate these services securely.

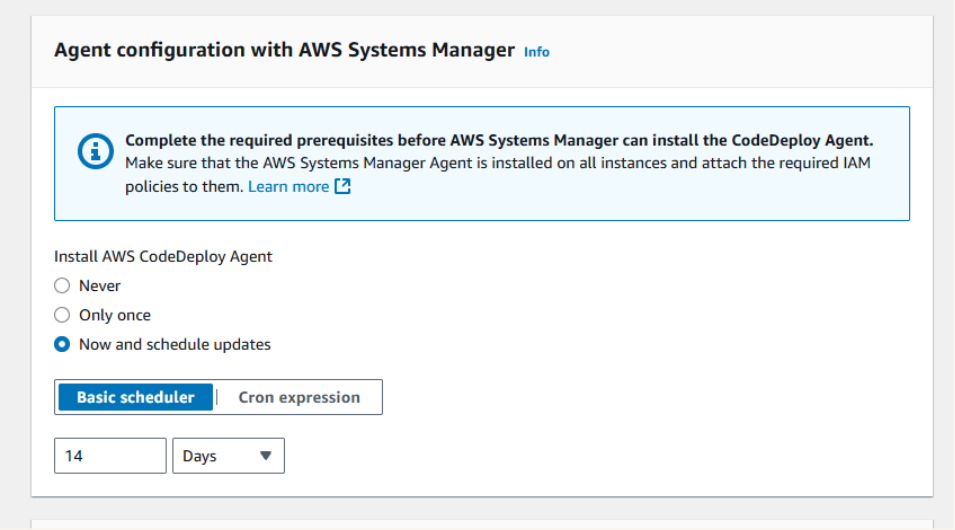
Tags are helpful for identifying and organising resources in AWS, and CodeDeploy uses them to know exactly which EC2 instances should receive a deployment. Instead of manually selecting servers, we apply a tag to mark their role. I used the tag role:webserver to target my deployment instance. This makes deployments flexible (new instances with the same tag are included automatically), clear (tags document resource purpose), and integrated with our CloudFormation setup.


Environment configuration
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

Deployment configurations

Another key setting is the deployment configuration, which affects how updates roll out across instances in a deployment group. Options include AllAtOnce (deploy to all instances simultaneously), OneAtATime (deploy to one instance, verify, then continue), and HalfAtATime (deploy to 50% first, then the rest). I used CodeDeployDefault.AllAtOnce, since I only have one instance in this project and speed matters more than caution.

In order to connect EC2 instances with AWS CodeDeploy, a CodeDeploy Agent is also set up on the server. The agent is a small piece of software that runs in the background and listens for deployment instructions from the CodeDeploy service. When a new deployment is triggered, the agent downloads the application files and the appspec.yml, runs the associated lifecycle event scripts (like install, start, and stop), and reports status back to CodeDeploy. Without the agent, CodeDeploy would have no way to communicate with or control what happens on the instance.

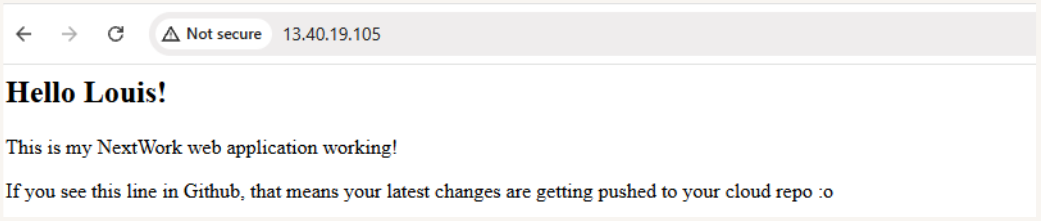


Success!

A CodeDeploy deployment is a single release event where a chosen application revision is pushed to the target environment. It includes instructions on which version to deploy, which deployment group to use, and which configuration strategy to follow. The difference to a deployment group is that the deployment group defines the environment and rules (which servers, how to deploy), while the deployment itself is the actual execution of those rules for a specific version of the app.

I had to configure a revision location, which means telling CodeDeploy where to find the application build artifacts for this deployment. The revision location points to the storage location of the packaged app version that CodeDeploy will install on the target servers. My revision location was the S3 bucket (nextwork-devops-cicd) containing the .zip/WAR file, so CodeDeploy could retrieve and deploy the latest build of my web app.

To check that the deployment was a success, I visited the Public IPv4 DNS of my EC2 instance in a web browser. Since the deployment ran all the lifecycle events successfully, I saw my web application running through Apache and Tomcat, which confirmed that CodeDeploy had correctly copied the artifact from S3, executed the scripts, and started the server. Reaching the app in the browser was the final proof that the automated deployment worked end-to-end.

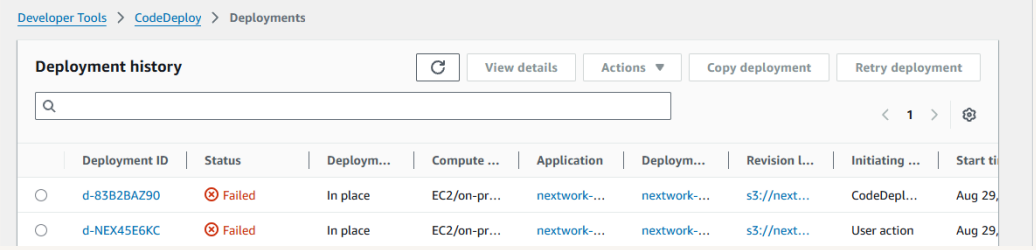

Not secure
13.40.19.105
Hello Louis!
This is my NextWork web application working!
If you see this line in Github, that means your latest changes are getting pushed to your cloud repo :o

Disaster Recovery

In a project extension, I decided to intentionally break my deployment by modifying the stop_server.sh script. The intentional error I created was misspelling the systemctl command as systemctll. This will cause the deployment to fail because the script tries to call a non-existent command, resulting in a non-zero exit code. By doing this, I can safely simulate a failed deployment and then practise performing a rollback to recover.

I also enabled rollbacks with this deployment, which means if the deployment fails, CodeDeploy will automatically restore the last known good version of my application. This is important because deployments can introduce bugs or errors (like a broken script), and rollbacks minimise downtime by quickly reverting to the previous working version. It acts like an “undo button,” keeping the application available to users while I fix issues in the background.

When my deployment failed, the automatic rollback would revert my application back to the last known good version because CodeDeploy (or CodePipeline) would be configured to detect errors during the deployment lifecycle. To actually recover from the failure, I’d have to redeploy a fixed version of the code. In production environments, I would implement automated rollbacks by enabling the rollback on failure setting in CodeDeploy or integrating CodePipeline with CloudWatch alarms. This way, if a deployment fails or a health check is missed, the system automatically triggers a rollback without manual intervention, minimising downtime and ensuring users always have access to a working version of the application.


Developer Tools > CodeDeploy > Deployments
Deployment history
View details
Actions
Copy deployment
Retry deployment
1
Deployment ID
Status
Deploym...
Compute ...
Application
Deploym...
Revision l...
Initiating ...
Start ti
d-83B2BAZ90
Failed
In place
EC2/on-pr...
nextwork-...
nextwork-...
s3://next...
CodeDepl...
Aug 29,
d-NEX45E6KC
Failed
In place
EC2/on-pr...
nextwork-...
nextwork-...
s3://next...
User action
Aug 29,

The place to learn & showcase your skills

Check out nextwork.org for more projects