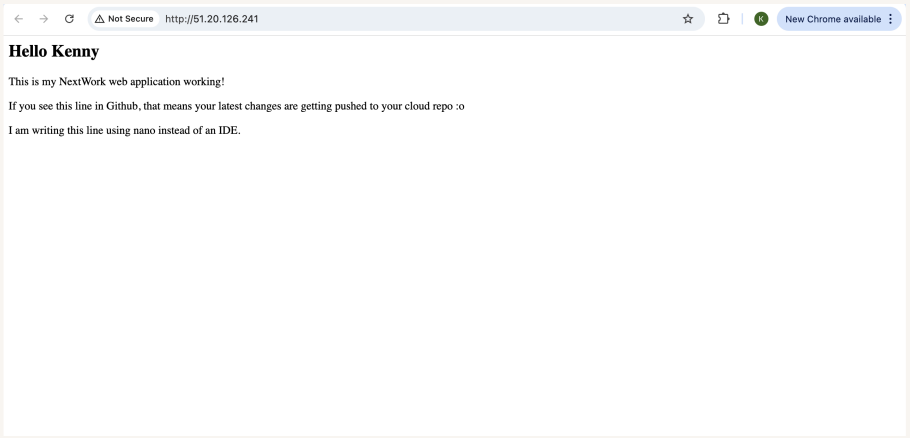
nextwork.org

Deploy a Web App with CodeDeploy

Kehinde Abiuwa



K
Kehinde Abiuwa
NextWork Student

Introducing Today's Project!

In this project, I will demonstrate how to deploy deploy a web app with AWS CodeDeploy. I'm doing this project to learn how to design and implement a fully functional CI/CD pipeline.

Key tools and concepts

Services I used were AWS CloudFormation (to define infra as code), EC2 (to host the app), VPC (for networking and isolation), IAM (to manage permissions), S3 (for storing artifacts), CodeBuild (to build and package), and CodeDeploy (to automate deployments). Key concepts I learnt include Infrastructure as Code (repeatable, version-controlled infra), CI/CD pipelines (build → package → deploy automatically), deployment groups vs. applications in CodeDeploy, lifecycle hooks with scripts (stop, install, start, validate), using tags to target EC2 instances, handling rollbacks, and monitoring deployments with logs/alarms.

Project reflection

This project took me approximately 140 minutes. The most challenging part was troubleshooting failed deployments (like fixing broken scripts and handling file conflicts) and understanding how CodeDeploy uses the current revision's scripts during the stop phase. It was most rewarding to see the full CI/CD pipeline come together—pushing code to GitHub, having CodeBuild package it, and then watching CodeDeploy automatically deliver and run my web app successfully on EC2.

Kehinde Abiuwa
NextWork Student

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 right away

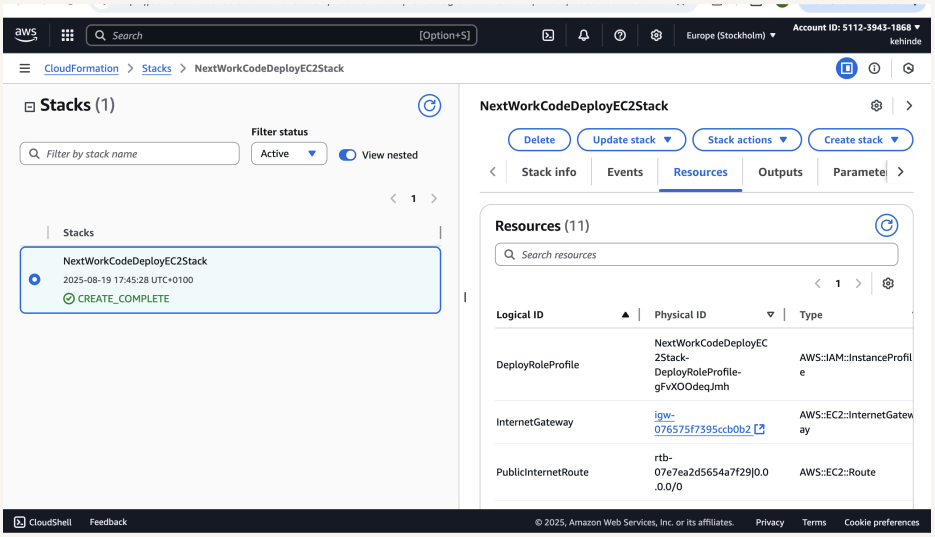
Deployment Environment

To set up for CodeDeploy, I launched an EC2 instance and VPC because the EC2 host runs the CodeDeploy agent and my application, while the VPC gives me an isolated, production-grade network (subnets, routing, security groups) to expose only what's needed, separate prod from my dev box, integrate with load balancers/Auto Scaling for rolling or blue-green releases, control access to S3/CodeDeploy/CloudWatch via NAT or VPC endpoints, and define everything as repeatable Infrastructure as Code with CloudFormation.

Instead of launching these resources manually, I used AWS CloudFormation to provision them from a version-controlled template. When I need to delete these resources, I simply delete the CloudFormation stack, and it tears everything down in the correct order.

Other resources created in this template include the Internet Gateway, route table, default route, and the gateway attachment/associations that connect the VPC and subnet to the internet. They're also in the template because the web server needs outbound access (to fetch packages/artifacts) and an inbound path for HTTP, and defining them in CloudFormation keeps the whole stack reproducible and auditable as Infrastructure as Code

Kehinde Abiuwa
NextWork Student



Deployment Scripts

Scripts are small, executable text files (e.g., Bash on Linux) that run a series of terminal commands for you. They make repetitive steps consistent, repeatable, and safe to run multiple times (no forgotten commands or typos). To set up CodeDeploy, I also wrote scripts to: stop any running version of the app before updating, install required packages and set file permissions, start the new version as a background service, and validate the deployment with a quick health check (failing fast if something's wrong).

The 'install_dependencies will install tomcat and apache and then create settings that let these programs to work together, making my website accessible to visitors on the internet.

The start_server.sh will start my app stack now and make it persist across reboots. Specifically, it starts Tomcat and Apache HTTPD immediately and enables both services so they auto-start on boot (via systemctl start/enable tomcat.service and httpd.service).

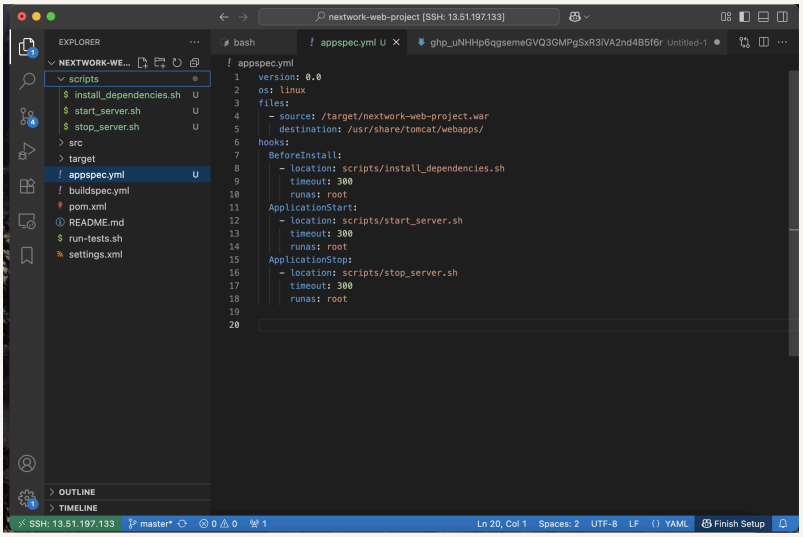
The stop_server.sh will safely shut down the web stack if it's running, first checking for Apache HTTPD and Tomcat processes with pgrep, then issuing systemctl stop httpd.service and systemctl stop tomcat.service as needed.

appspec.yml

Then, I wrote an appspec.yml file to tell CodeDeploy exactly what to copy from my build and which scripts to run during deployment. In my case, it copies the built WAR into Tomcat's webapps/ folder and runs my install/stop/start scripts so the new version comes online cleanly. The key sections in appspec.yml are: version / os – declares the schema and target platform for an EC2/on-prem deployment (version: 0.0, os: linux). files – maps files from the artifact to the instance. Here it takes target/nextwork-web-project.war → /usr/share/tomcat/webapps/ so Tomcat can auto-deploy the WAR. hooks – ties lifecycle events to your scripts: ApplicationStop → scripts/stop_server.sh (gracefully stop existing services). BeforeInstall → scripts/install_dependencies.sh (packages/config/permissions). ApplicationStart → scripts/start_server.sh (start and enable services).

I also updated buildspec.yml because CodeDeploy only gets what's inside the build artifact—so the WAR, the appspec.yml, and the deployment scripts all need to be bundled together. I added the artifacts.files entries for: target/nextwork-web-project.war (the app to deploy) appspec.yml (deployment manifest at the artifact root) scripts/**/* (hook scripts referenced by appspec.yml) and set discard-paths: no so the scripts/ folder structure is preserved (letting hooks like scripts/start_server.sh resolve correctly).

Kehinde Abiuwa
NextWork Student

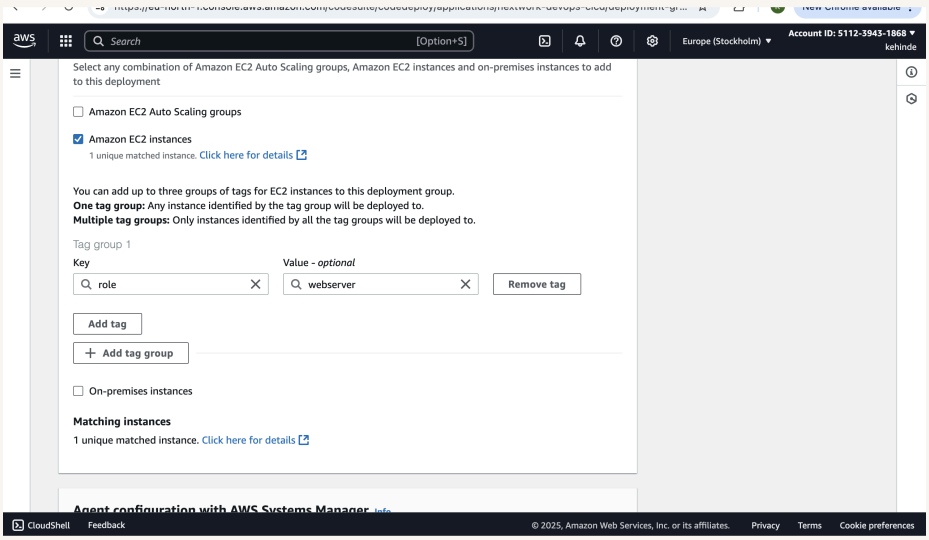


Setting Up CodeDeploy

A deployment group is the specific targets and rules for a release—Which servers to update (e.g., prod or staging) and how to do it. A CodeDeploy application is the overall app container in CodeDeploy—the thing you deploy. One application can have multiple deployment groups.

To set up a deployment group, you also need to create an IAM role to let CodeDeploy act on your behalf—read the deployment artifact (S3/CodeArtifact), target the right EC2 instances (tags/ASG), coordinate with the CodeDeploy agent, update/load-balance traffic for health checks and rollbacks, and write logs/metrics to CloudWatch. Without this role, CodeDeploy can't access or manage the resources needed to perform the deployment.

Tags are helpful for targeting servers by purpose, so deployments keep hitting the right machines even if you scale, replace instances, or add environments. I used the tag role=webserver to let CodeDeploy automatically find the correct EC2 instance(s) for this app and deploy to them.

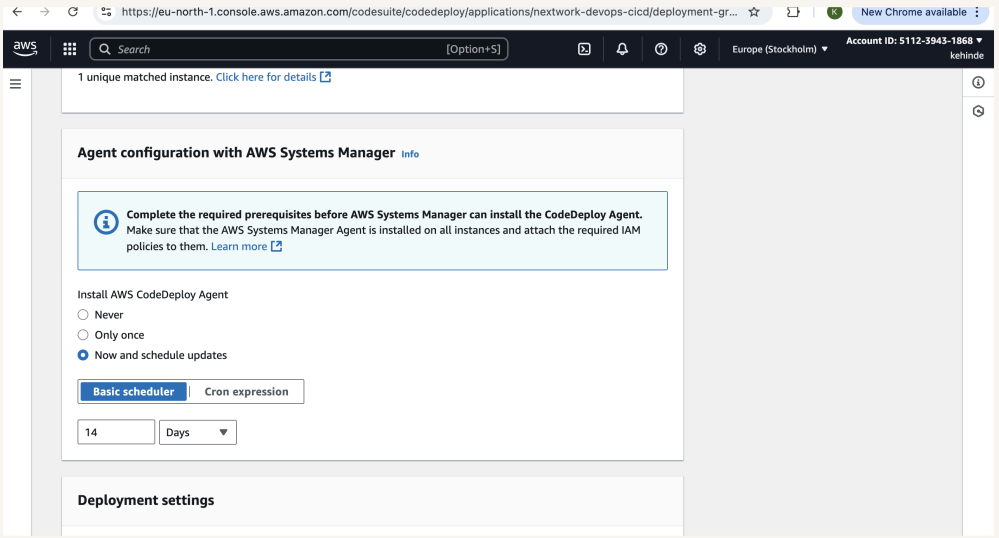

Search
[Option+S]
Europe (Stockholm)
Account ID: 5112-3943-1868
kehinde
Select any combination of Amazon EC2 Auto Scaling groups, Amazon EC2 instances and on-premises instances to add to this deployment
Amazon EC2 Auto Scaling groups
Amazon EC2 instances
1 unique matched instance. Click here for details
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

Deployment configurations

Another key setting is the deployment configuration, which affects how many instances update at once and how much healthy capacity must stay online during the rollout. For EC2/In-Place deployments, the built-in options are: CodeDeployDefault.AllAtOnce – deploy to all instances at once. Fastest, but can cause brief downtime while services restart; if any instance fails, the deployment fails. CodeDeployDefault.HalfAtATime – update about 50% at a time, keeping roughly half serving traffic. CodeDeployDefault.OneAtATime – update one instance at a time. Slowest, but highest availability. I used CodeDeployDefault.AllAtOnce, so the new version is pushed to all tagged web servers simultaneously—it's the quickest path, acceptable for a single-instance setup or a short maintenance window, with the trade-off of possible brief downtime during Tomcat/HTTPD restarts.

In order to connect the CodeDeploy service to my EC2 instance, a CodeDeploy Agent is also set up to run on the instance, poll CodeDeploy for deployments, download the artifact and appspec.yml, execute my lifecycle scripts (stop/install/start/validate), and report status/logs back to CodeDeploy. I configure it to auto-update about every 14 days so the agent stays current.

Kehinde Abiuwa
NextWork Student

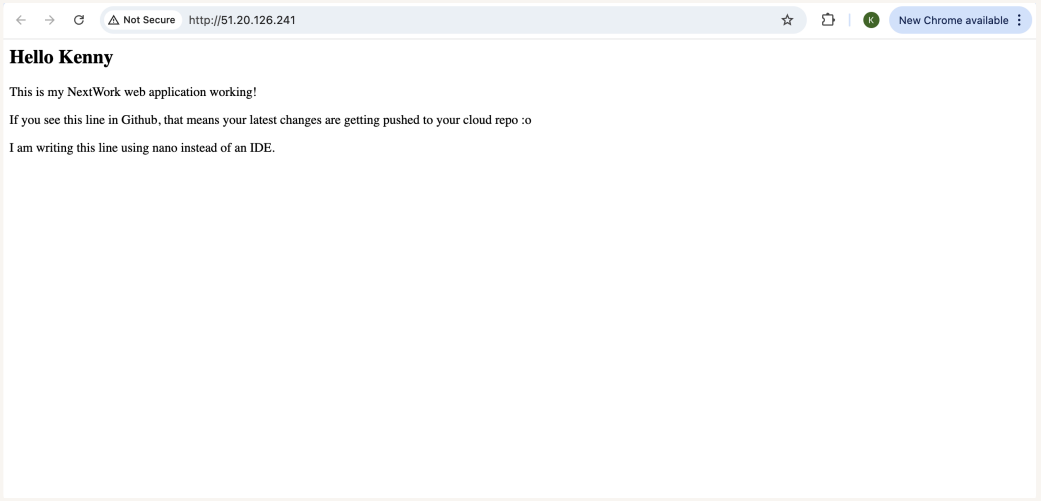


Success!

A CodeDeploy deployment is a single rollout of a specific app version to my targets. CodeDeploy uses it to run the lifecycle steps (stop → copy → install → start → validate) and records its own ID, status, and logs. The difference to a deployment group is that the deployment group defines where and how to deploy (target instances, strategy, health checks), while a deployment is one execution that uses those settings to deliver a particular version.

I had to configure a revision location, which means telling CodeDeploy where to fetch the exact build artifact and appspec.yml for this release, so it can download that package and run the deployment. My revision location was an S3 object produced by aws codebuild—a single ZIP that includes the WAR (target/nextwork-web-project.war), appspec.yml, and the scripts/ folder (for the lifecycle hooks).

To check that the deployment was a success, I opened the IPv4 public address of my EC2 instance in a web browser. I saw the HTML content of my web app load correctly, which confirmed that CodeDeploy had copied the WAR, started Tomcat/HTTPD, and served the application as expected.


Not Secure
http://51.20.126.241
New Chrome available
Hello Kenny
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
I am writing this line using nano instead of an IDE.

Kehinde Abiuwa
NextWork Student

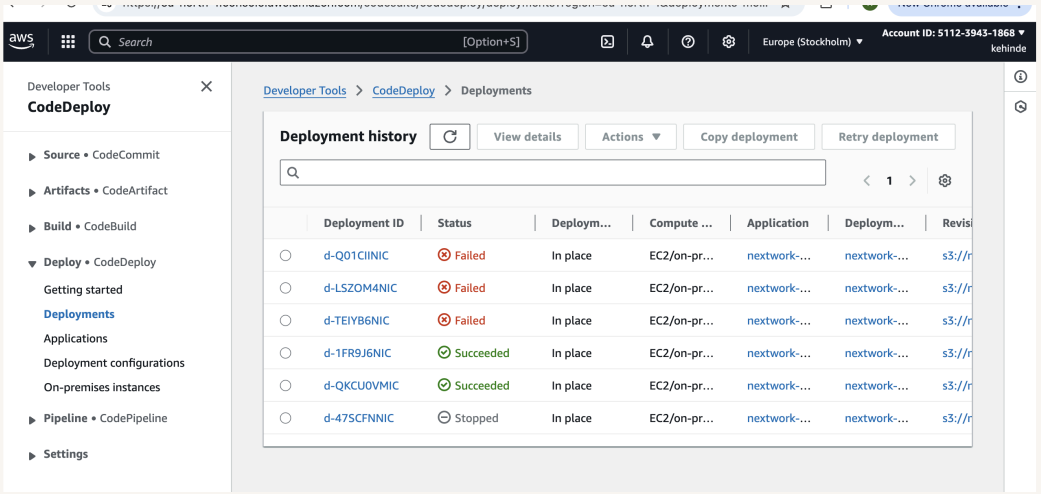
Disaster Recovery

In a project extension, I decided to simulate a disaster recovery scenario by breaking one of my deployment scripts on purpose. The intentional error I created was misspelling the systemctl command as systemctll inside stop_server.sh and also adding an explicit exit 1. This will cause the deployment to fail because the script exits with a non-zero status, making CodeDeploy mark the ApplicationStop hook as failed and rolling back the deployment.

I also enabled rollbacks with this deployment, which means that if the deployment fails at any lifecycle step (like my broken stop_server.sh), CodeDeploy will automatically revert to the last known good revision on the instance. This helps restore service quickly without manual intervention and reduces downtime caused by bad releases.

When my deployment failed, the automatic rollback restored the last good version because I enabled rollback on deployment failures. To actually recover from the bad release, I'd have to keep the previous artifact available, fail fast with a health check, and let CodeDeploy redeploy the last successful revision. In production environments I keep it simple: turn on auto-rollback (on failure + alarms), use blue/green behind an ALB, add a ValidateService health check, and keep artifacts versioned and changes backward-compatible so rolling back is safe.

Kehinde Abiuwa
NextWork Student



The place to learn & showcase your skills

Check out nextwork.org for more projects