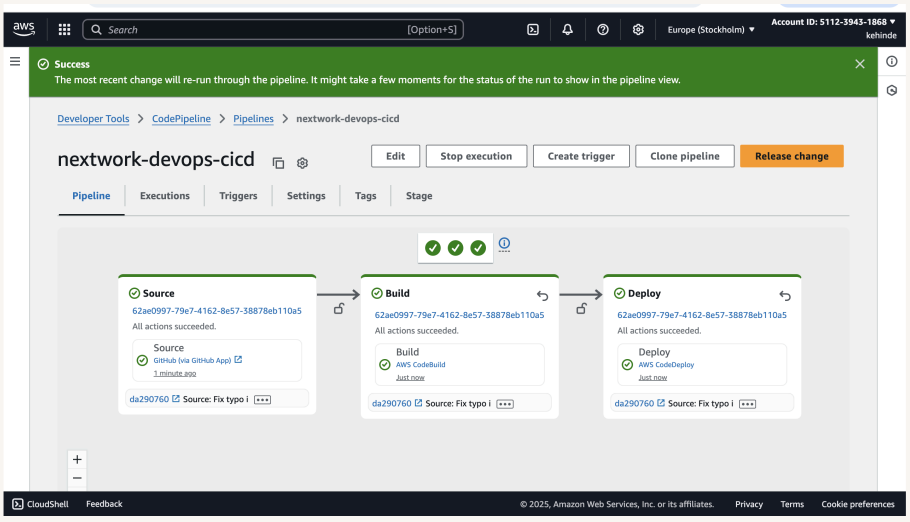
nextwork.org

Build a CI/CD Pipeline with AWS

Kehinde Abiuwa



Kehinde Abiuwa
NextWork Student

Introducing Today's Project!

In this project, I will demonstrate how to build a complete CI/CD pipeline in AWS CodePipeline that connects a GitHub source stage to CodeBuild for builds/tests and CodeDeploy for automated deployments—end-to-end. I'll wire up the source, build, and deploy stages, then create and run the pipeline to watch changes flow automatically, including testing an automatic deployment trigger from a commit. I'm doing this project to learn how to automate builds and releases with CodePipeline, gain visibility across the workflow, and practice safe delivery patterns—like automatically rolling back a deployment if something goes wrong

Key tools and concepts

Services I used were: AWS CodePipeline (orchestrator), GitHub (source), AWS CodeBuild (build/test), Amazon S3 (artifact store), AWS CodeDeploy (deploy), Amazon EC2 (targets), AWS IAM (roles/policies), AWS CloudFormation (infrastructure as code). Key concepts I learnt include: End-to-end CI/CD flow (Source → Build → Deploy), GitHub webhooks & default branch triggers, pipeline execution modes (Superseded/Queued/Parallel), artifacts (SourceArtifact → BuildArtifact), buildspec/appspec files, CodeDeploy lifecycle hooks & automatic rollback, IAM service roles/permissions, and using CloudFormation to provision the environment reproducibly.

Project reflection

This project took me approximately 210 minutes. The most challenging part was diagnosing the Deploy stage failures—realizing that a stage retry reuses the same artifact, that CodeDeploy was initially running an old on-instance hook (stop_server.sh), and that my JSP app needed Tomcat (not just Apache) on Amazon Linux 2023, plus fixing an IAM permission gap (codedeploy:GetApplicationRevision).

Sorting out appspec paths, executable bits, and mapping traffic (80 → 8080) also took careful tweaking. It was most rewarding to see a fully automated Source → Build → Deploy run kick off from a GitHub commit, watch the app go live after wiring Tomcat correctly, and then prove resilience by performing a rollback. Along the way I solidified buildspec/appspec usage, Superseded execution mode behavior, artifact flow, and the role of IAM permissions in cross-service orchestration.

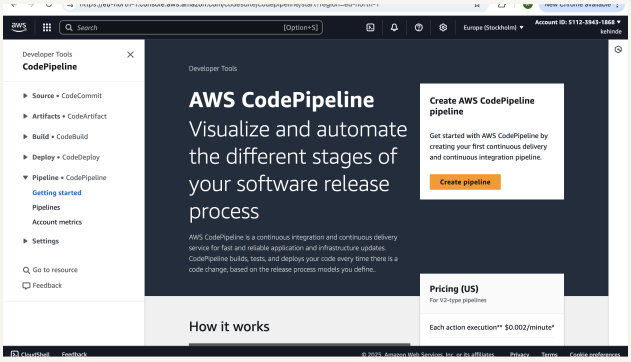
Starting a CI/CD Pipeline

AWS CodePipeline is a fully managed continuous integration and continuous delivery (CI/CD) service that automates the steps required to release your software changes. It connects your source (like GitHub), build (like CodeBuild), and deployment (like CodeDeploy) stages into a single pipeline, so every time you push code changes, they are automatically built, tested, and deployed. It helps developers release updates faster, safer, and with less manual work by providing automation, visibility, and built-in rollback support when something fails.

CodePipeline offers different execution modes based on how you want multiple pipeline runs to be handled when new code changes arrive. I chose Superseded mode because it ensures that only the most recent code changes are built and deployed, canceling any older in-progress executions—this keeps my pipeline focused on the latest version of the application. Other options include Queued mode, where executions wait their turn and run one after another, and Parallel mode, where multiple executions can run independently at the same time, useful if you need to process different branches or builds concurrently.

A service role gets created automatically during setup so that CodePipeline has permission to interact with other AWS services on your behalf. For example, the role allows CodePipeline to pull source code from GitHub or CodeCommit, trigger builds in CodeBuild, store and retrieve artifacts in S3, and start deployments in CodeDeploy. Without this role, CodePipeline wouldn't be able to orchestrate the CI/CD workflow across services securely.

Kehinde Abiuwa
NextWork Student

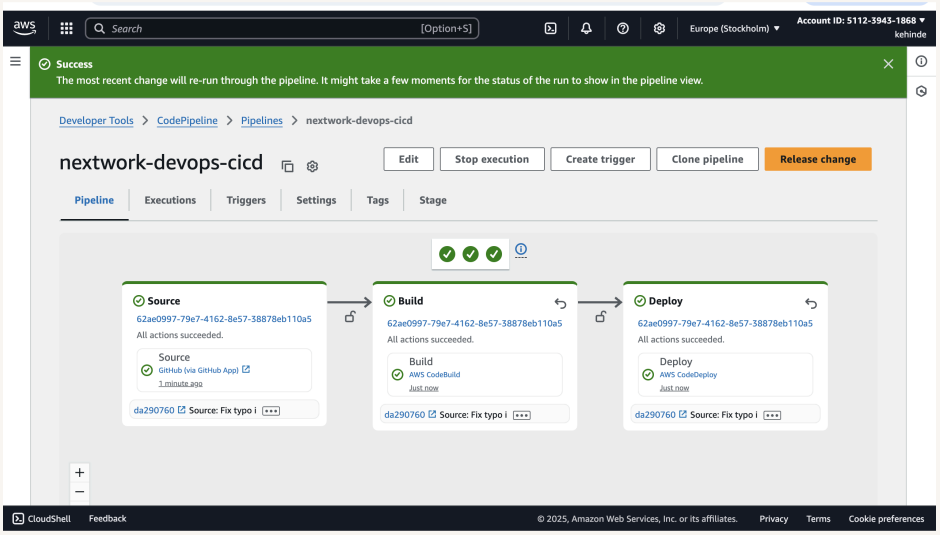


CI/CD Stages

The three stages I've set up in my CI/CD pipeline are Source → Build → Deploy. Source (GitHub): Watches the default branch for changes via webhooks and outputs SourceArtifact. I learned how branch selection and webhook events drive automatic executions, and why the CodePipeline service role is needed to access GitHub/S3. Build (CodeBuild): Consumes SourceArtifact, runs compile/test steps, and produces BuildArtifact. I learned how buildspecs package artifacts for downstream stages and why the build stage's input artifact must be the source output. Deploy (CodeDeploy): Uses BuildArtifact to update the target deployment group with automatic rollback on failure. I learned how deployments reference an application + deployment group, and how execution modes (I chose Superseded) affect concurrent runs.

CodePipeline organizes the three stages into a left-to-right flow of cards — Source → Build → Deploy — so you can see at a glance which stage is running or has succeeded/failed. In each stage, you can see more details on: Status & timing: green check/failed icon and "last run" timestamp. Action provider: e.g., GitHub, AWS CodeBuild, AWS CodeDeploy. Revision used: the commit SHA and message (clickable link to the commit). Links to runs/logs: open the specific CodeBuild build (CloudWatch logs) or CodeDeploy deployment (events, lifecycle hooks, instance-level logs). Artifacts flow: which artifact is input and output (e.g., SourceArtifact → BuildArtifact). Controls/history: retry stage/release change, and per-stage execution history with duration when you drill in.

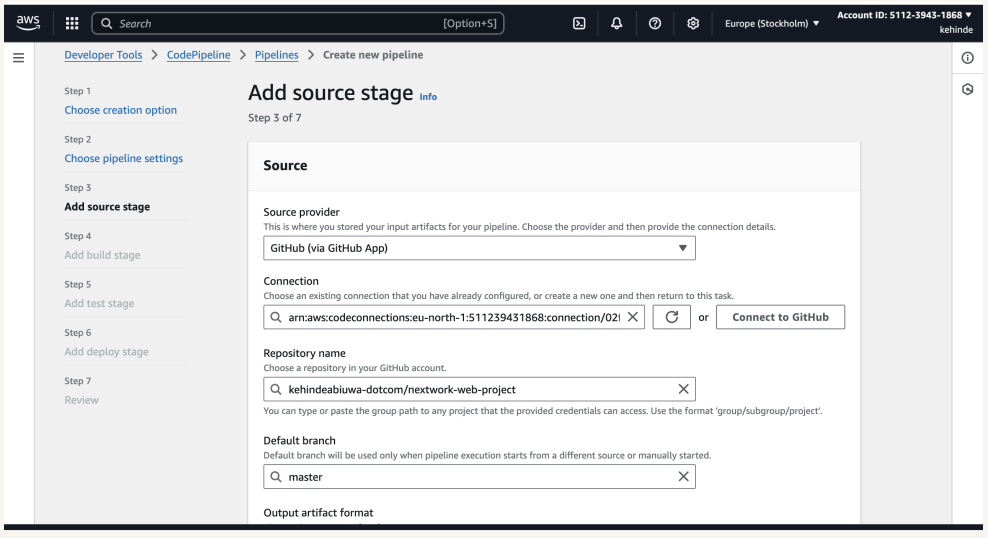
Kehinde Abiuwa
NextWork Student



Source Stage

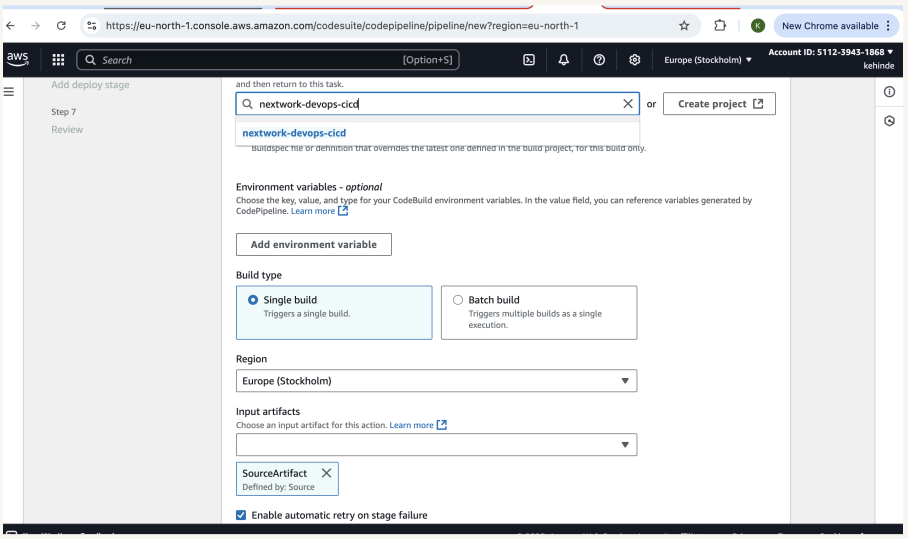
In the Source stage, the default branch tells CodePipeline which branch of your repository it should monitor for changes and use as the starting point for the pipeline. This ensures that only commits pushed to that branch (for example, main or master) will automatically trigger a new pipeline execution.

The source stage is also where you enable webhook events, which are important because they let CodePipeline automatically detect and react to changes in your source repository. Instead of manually starting the pipeline, webhook events trigger a new execution the moment code is pushed—ensuring your CI/CD pipeline stays fast, automated, and always in sync with the latest changes.



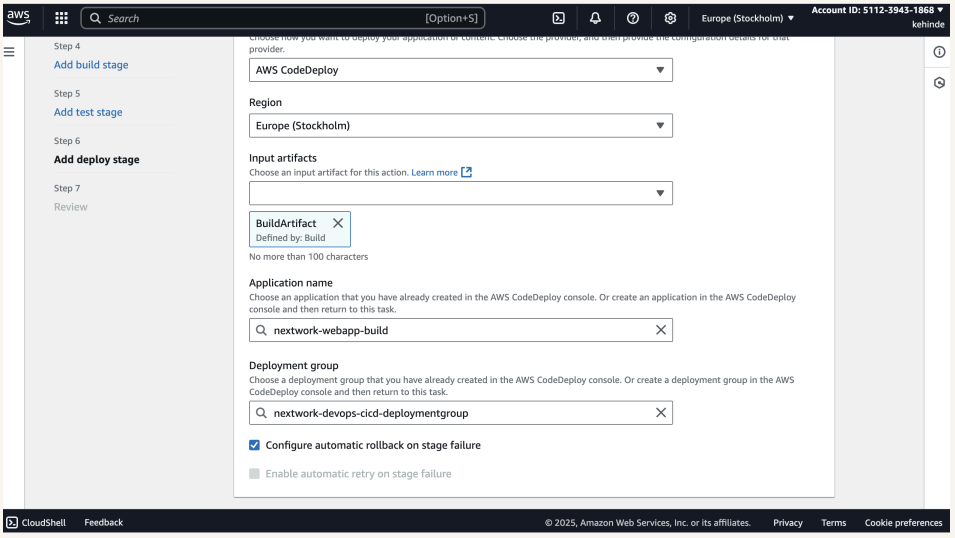
Build Stage

The Build stage sets up the environment to compile and test my application code. I configured this stage to use the output from the Source stage as its starting point. The input artifact for the build stage is SourceArtifact because it contains the code pulled from my GitHub repository, and that's what CodeBuild needs in order to run the build and generate new build artifacts for deployment.



Deploy Stage

The Deploy stage is where the built artifacts from CodeBuild are deployed to the target environment using AWS CodeDeploy. I configured the stage to use BuildArtifact (output from the Build stage) as the input artifact, selected my CodeDeploy application (nextwork-webapp-build) and its deployment group (nextwork-devops-cicd-deploymentgroup), and enabled automatic rollback on stage failure so that the system will revert to the last working version if something goes wrong during deployment.



Success!

Since my CI/CD pipeline gets triggered by commits to the master branch via the GitHub webhook, I tested my pipeline by editing src/main/webapp/index.jsp and adding a new line inside the <body>, I then committed and pushed the change. That push triggered the pipeline (Source → Build → Deploy), and I verified the new text appeared after deployment.

The moment I pushed the code change to the master branch, the GitHub webhook fired and CodePipeline started a new execution (in Superseded mode, canceling any older run). Source immediately pulled the latest commit, Build kicked off with that revision and produced a new BuildArtifact, and Deploy used that artifact to update the environment. The commit message under each stage reflects the same revision, showing exactly which commit was fetched, built, and deployed.

Once my pipeline executed successfully, I checked my web app from a web browser by visit the public ipv4 address, and I saw the changes I made in my index.jsp

Hello NextWork!

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 :D

If you see this line, that means your latest changes are automatically deployed into production by CodePipeline!

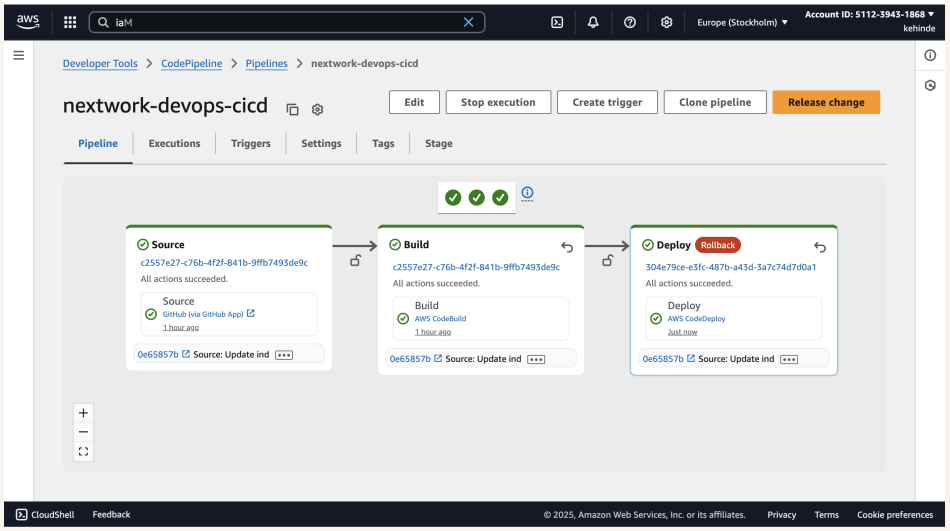
Testing the Pipeline

I triggered a rollback in the Deploy stage to redeploy the last successful build to the same deployment group. Automatic rollback is important because it quickly restores a working version, reduces downtime, and gives you a safety net when a bad change or failing hook slips through.

During the rollback, the Source and Build stages are unaffected because rollback happens only in the Deploy stage: CodeDeploy simply redeploys the last successful artifact (N-1) that's already in S3/its revision history—no new commit is fetched and no new build is created. I could verify this by comparing: the commit SHA shown under Source/Build (still the latest commit), with the revision used by the rollback deployment in CodeDeploy (it points to the previous artifact/S3 key). You'll also see no new CodeBuild run at the rollback time in the build history—another confirmation that Source/Build didn't re-run.

After rollback, the live web app reverted to the last successful deployment (N-1)—the version just before my change. I confirmed it by refreshing the site (the new line in index.jsp was gone) and by checking CodeDeploy, which showed the deployment using the previous artifact/S3 key labeled "last successful revision."

Kehinde Abiuwa
NextWork Student



The place to learn & showcase your skills

Check out nextwork.org for more projects