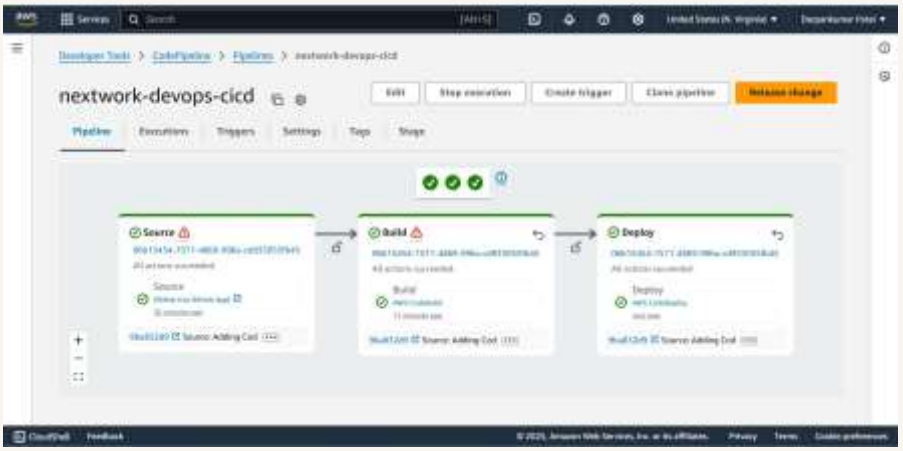
Darpan Patel

Build a CI/CD Pipeline with AWS



Darpan Patel

Introducing Today's Project!

In this project, I will demonstrate how to build a complete CI/CD pipeline using AWS CodePipeline that automates the process of building, testing, and deploying a web application. I'm doing this project to learn how to integrate AWS services like CodeBuild, CodeDeploy, and GitHub into an automated workflow that reduces manual effort, speeds up releases, and ensures reliable deployments through continuous integration and continuous delivery practices.

Key tools and concepts

Services I used were AWS CodePipeline, CodeBuild, CodeDeploy, EC2, S3, IAM, and GitHub. These services collectively helped me implement a fully automated CI/CD pipeline. Key concepts I learnt include how pipelines are structured into stages (Source, Build, Deploy), how artifacts are passed between stages, and how webhooks trigger executions automatically. I also gained hands-on experience with rollback mechanisms, IAM service roles, artifact storage, and monitoring pipeline executions. Overall, this project gave me a complete view of modern DevOps practices and how to manage code changes from commit to production deployment seamlessly.

Project reflection

This project took me approximately 2–3 hours to complete. The most challenging part was troubleshooting deployment failures during the CodeDeploy stage, especially when instances became unhealthy or failed lifecycle events. Diagnosing and resolving these issues helped me better understand deployment hooks and EC2 configurations. It was most rewarding to see the entire pipeline respond automatically to a GitHub commit and deploy the updated web app without any manual steps. That moment truly demonstrated the power of automation and gave me confidence in setting up production-ready CI/CD pipelines.

Darpan Patel

Starting a CI/CD Pipeline

AWS CodePipeline is a fully managed continuous integration and continuous delivery (CI/CD) service that automates the build, test, and deployment phases of your release process. It enables you to model and visualize your software release workflow as a series of stages, such as source, build, and deploy. By integrating with services like GitHub (for source control), AWS CodeBuild (for building), and AWS CodeDeploy (for deployment), CodePipeline helps deliver updates rapidly and reliably. Every time you commit code, CodePipeline can automatically trigger the workflow, ensuring consistent, error-reduced deployments across your environments.

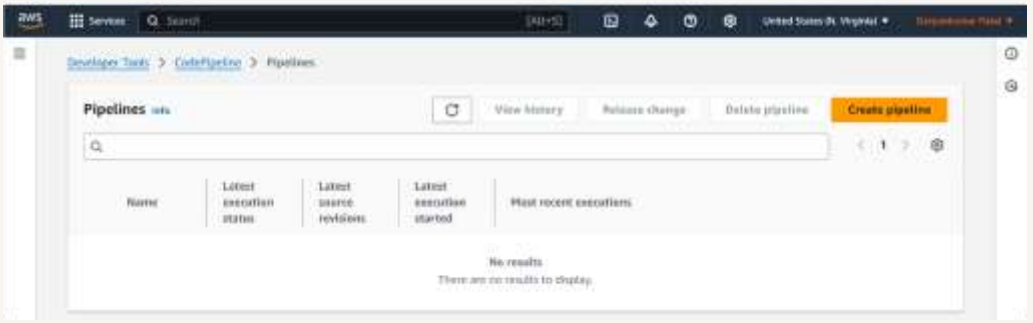
CodePipeline offers different execution modes based on how you want pipeline runs to be handled when multiple executions are triggered. I chose Superseded mode, where the newest pipeline run cancels any currently running execution. This is ideal for fast-moving development workflows because it ensures only the latest changes are deployed. Other options include: Queued mode – New executions wait in line until the current one finishes. This ensures every change is deployed in order but may slow down deployments. Parallel mode – Multiple executions run simultaneously and independently. This is useful when working with multiple branches or when builds don’t interfere with one another. Each mode supports a different CI/CD strategy depending on your team's needs and how critical sequential consistency is.

A service role gets created automatically during setup so that AWS CodePipeline can securely perform actions on your behalf across other AWS services.

This role grants CodePipeline the necessary permissions to: Pull source code from repositories (e.g., GitHub or S3) Start and manage builds in AWS CodeBuild
Deploy your application using AWS CodeDeploy Store and retrieve artifacts from Amazon S3 Log activity to CloudWatch Logs Without this service role, CodePipeline wouldn't be able to orchestrate the CI/CD workflow, as it wouldn't have the authority to interact with

Darpan Patel

those services. The role ensures that all operations are securely managed and scoped to only the resources and actions required.



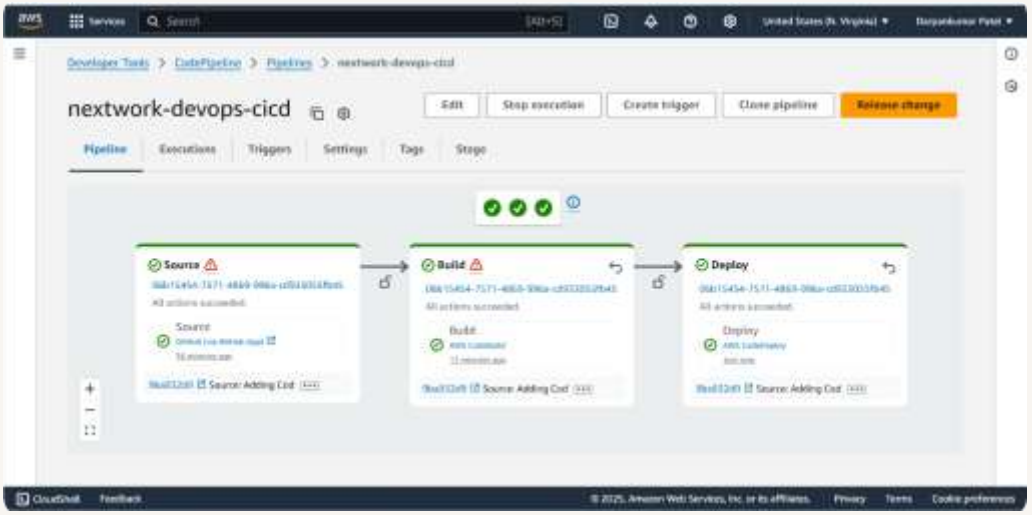
CI/CD Stages

The three stages I've set up in my CI/CD pipeline are Source, Build, and Deploy. While setting up each part, I learned about the key roles they play in automation. The Source stage connects to my GitHub repo and fetches the latest code using webhook events, enabling automatic triggers. The Build stage uses AWS CodeBuild to compile and package the code. I learned how input artifacts from the source are used to create deployable outputs. Finally, the Deploy stage uses AWS CodeDeploy to push the build to EC2 instances. Enabling rollback ensures stability in case of errors. Setting up this pipeline taught me how each service connects, and how to automate code delivery from commit to production using CodePipeline.

CodePipeline organizes the three stages—Source, Build, and Deploy—into a clear visual diagram, helping you monitor and manage the flow of code from GitHub to deployment on EC2. In each stage, you can view detailed status updates like whether it's pending (grey), running (blue), successful (green), or failed (red). You can also access logs and artifacts from each stage to see exactly what was executed. Each pipeline run (or execution) gets a unique ID, letting you track the history and see what changed between runs. Clicking into any stage

Darpan Patel

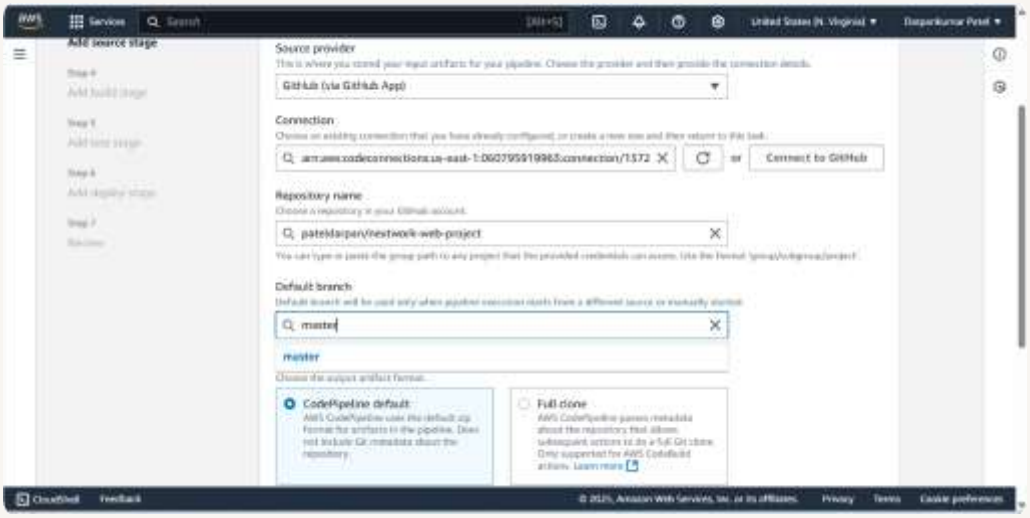
provides information such as input/output artifacts, environment variables, and real-time logs (especially useful in the Build and Deploy stages). You can also enable features like automatic rollback in Deploy or automatic retries in Build. This visibility allows for fast troubleshooting, consistent monitoring, and assurance that your CI/CD process is running as expected — automating everything from code commit to deployment.



Source Stage

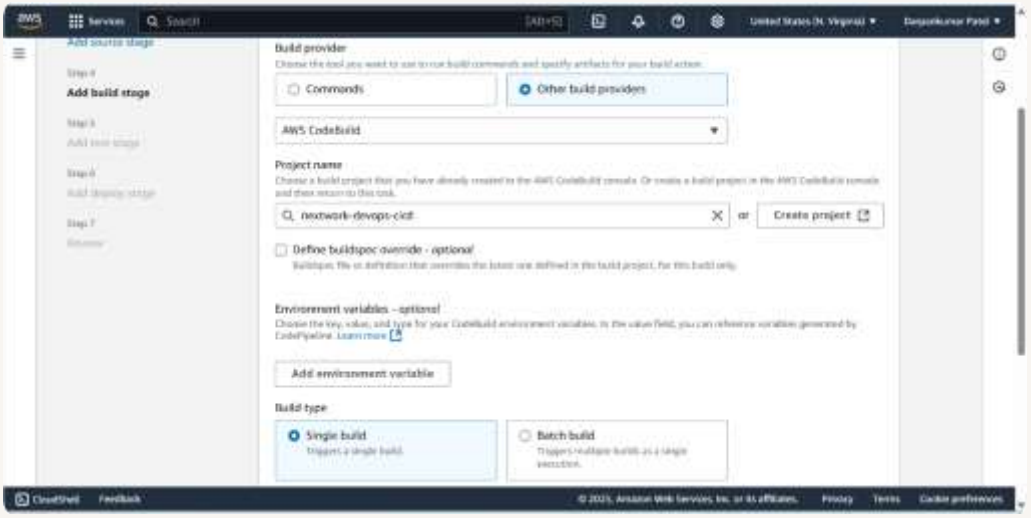
In the Source stage, the default branch tells CodePipeline which version of your codebase to monitor for changes. By specifying a default branch—typically master — you ensure that any new commits to that branch automatically trigger the pipeline. This helps maintain a consistent and automated CI/CD workflow, as only changes to the designated stable or production-ready branch initiate the build and deployment processes.

The source stage is also where you enable webhook events, which are important because they allow the CI/CD pipeline to automatically detect and respond to code changes in real time. When a developer pushes new code to the specified branch in GitHub, the webhook sends a notification to CodePipeline, instantly triggering the pipeline to start the build and deployment process without any manual intervention. This automation ensures faster feedback loops, reduces the risk of human error, and keeps the development and deployment process seamless and continuous.



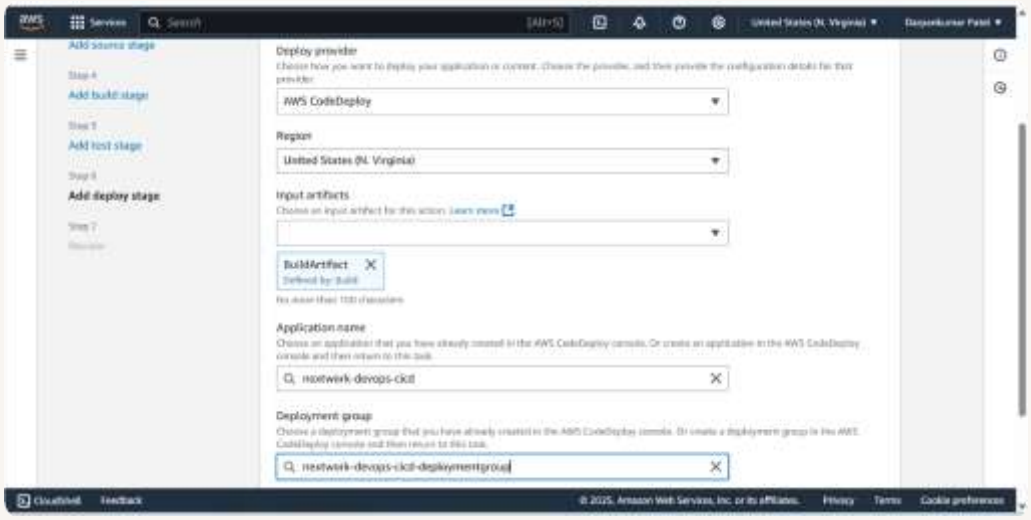
Build Stage

The Build stage sets up the process where the source code is compiled and packaged into deployable artifacts. I configured CodePipeline to use the output from the Source stage as the input for this step. The input artifact for the build stage is SourceArtifact, which is the ZIP file containing the latest version of the source code fetched from the GitHub repository. This ensures that the build process works on the most recent code changes, maintaining the flow of the CI/CD pipeline.



Deploy Stage

The Deploy stage is where the built application is delivered to the target environment for users to access. I configured this stage to use AWS CodeDeploy by selecting the existing CodeDeploy application and deployment group. I set the input artifact to BuildArtifact, which contains the output from the Build stage, and enabled automatic rollback to ensure that if the deployment fails, the pipeline will revert to the last successful deployment to maintain application stability.



Success!

Since my CI/CD pipeline gets triggered by changes to the master branch in my GitHub repository, I tested my pipeline by editing the index.jsp file located in src/main/webapp/. Specifically, I added the following line inside the <body> section: If you see this line, that means your latest changes are automatically deployed into production by CodePipeline! This simple change served as a clear visual indicator to confirm that the updated code was automatically built and deployed through the pipeline after pushing the commit.

The moment I pushed the code change to the `master` branch of my GitHub repository, the pipeline was automatically triggered — thanks to the webhook integration set up in the Source stage. The commit was detected instantly, and a new pipeline execution began. The Source stage fetched the latest version of the code, and under its details, the commit message I used — "Update index.jsp with a new line to test CodePipeline" — appeared clearly. This confirmed that the webhook and source integration were working as expected. Next, the Build stage picked up the output from the Source stage (the `SourceArtifact`), ran the build commands from my CodeBuild project, and produced the build artifact (`BuildArtifact`). Finally, the Deploy stage took that build artifact and deployed it to the EC2 instance using AWS CodeDeploy. Once all three stages turned green, it confirmed that my pipeline had responded correctly and completed the deployment successfully — all without manual intervention.

Once my pipeline executed successfully, I checked the deployed web application using the Public IPv4 DNS of the EC2 instance.

I opened the application in a browser and verified that the new line I added in the `index.jsp` file appeared exactly as expected: "If you see this line, that means your latest changes are automatically deployed into production by CodePipeline!" This confirmed that the pipeline detected the GitHub commit, built the project through CodeBuild, and deployed it using CodeDeploy — without any manual intervention. The successful display of my update on the live site validated that the CI/CD pipeline is fully automated and functioning correctly from source to production.

Testing the Pipeline

In a project extension, I initiated a rollback on the Deploy stage of the pipeline. Automatic rollback is important for maintaining application availability and stability in production environments. It ensures that if a deployment fails — due to issues like misconfigurations, bugs, or instance health check failures — the pipeline can quickly revert to the most recent successful version without requiring manual intervention. This minimizes downtime, reduces risk, and helps maintain user trust by ensuring that only tested, stable builds remain live.