Jacob Flores Gomez
08/20/2025
Documentation for my End-to-End CI/CD Pipeline on AWS

Introduction

This project represents the completion of my end-to-end Continuous Integration and Continuous Deployment (CI/CD) pipeline on Amazon Web Services (AWS). The primary objective was to automate the entire software delivery lifecycle, beginning with code commits and extending through build, testing, packaging, and deployment to a live environment. By documenting each step of the process, the project demonstrates not only the technical skills required to build such a system but also the troubleshooting and attention to detail that go into real-world DevOps work.

The motivation came when I wanted to deepen my practical understanding of AWS services and the DevOps methodology by building an app from scratch. On the other hand, I aimed to create a portfolio-ready artifact that showcases my ability to design and implement automation pipelines using industry-standard tools. DevOps, as a discipline, emphasizes collaboration between development and operations teams, continuous monitoring, and automated deployments that increase reliability and reduce human error. CI/CD embodies these values by creating a process where each code update flows seamlessly from version control to production without manual intervention.

By the end of this project, I had created a fully functioning pipeline that integrated GitHub for version control, AWS CodeArtifact for dependency management, AWS CodeBuild for automated compilation, AWS CodeDeploy for deployment automation, and AWS CodePipeline for orchestration. Each of these stages presented challenges and required iterative problem-solving, but they also provided valuable insights into the complexities and rewards of automating software delivery at scale.

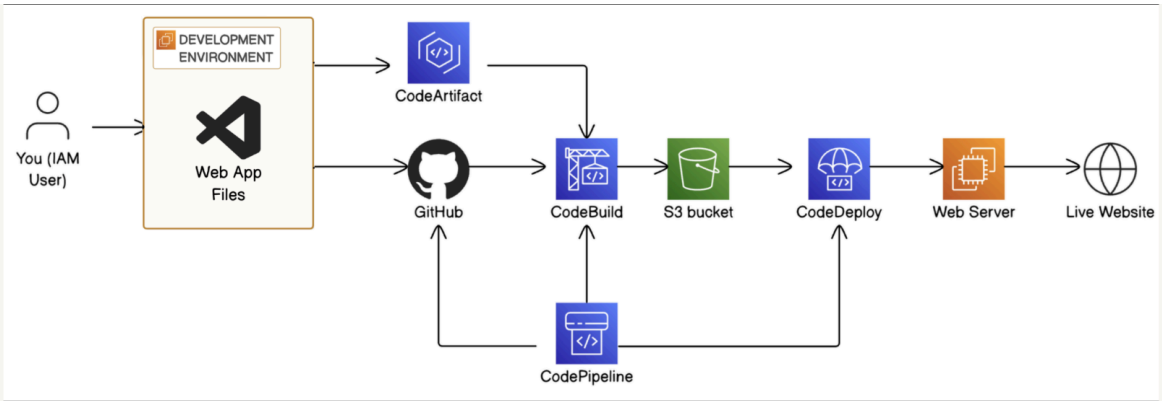

Figure 1: *CI/CD Pipeline AWS diagram*

Methodology

Instead of attempting to build the entire CI/CD pipeline at once, I divided the work into six smaller projects, each with specific learning outcomes and deliverables. This iterative approach mirrors real-world DevOps practices, where large initiatives are broken down into smaller, manageable tasks that can be tested, refined, and integrated over time.

The first step was to establish a foundation by launching a web application on an Amazon EC2 instance and configuring remote development access through Visual Studio Code (VS Code). This created a controlled environment where I could deploy, test, and modify code in a manner similar to production systems. Once the application was running, I introduced version control by connecting the project to a GitHub repository.

From there, I focused on package and dependency management using AWS CodeArtifact. By centralizing dependencies in a secure artifact repository, I ensured consistent builds and reduced reliance on external sources. After this, I moved on to continuous integration with AWS CodeBuild, where I defined build instructions through a buildspec.yml file and automated compilation, testing, and packaging of the application. The output of this stage was a production-ready artifact stored in an Amazon S3 bucket.

Then, using AWS CodeDeploy, I created scripts and lifecycle event definitions to automate the installation, configuration, and startup of the application on a target EC2 instance. This stage required significant troubleshooting, as I had to configure networking, permissions, and server behavior to ensure successful deployments.

Finally, I integrated all components into a single automated workflow using AWS CodePipeline. This service orchestrated the source, build, and deploy stages, enabling true CI/CD. A code commit to GitHub would now trigger the entire sequence: build with CodeBuild, package and upload to S3, and deploy with CodeDeploy to the EC2 instance. The methodology was validated by introducing small code changes and observing them propagate automatically through the pipeline until they were reflected in the live web application.

Throughout the process, I relied heavily on hands-on experimentation, AWS documentation, and continuous troubleshooting. I carefully documented every step, including the commands run, configuration files created, and issues encountered, because these details reflect not only the technical implementation but also the problem-solving required to complete a project of this scope. Screenshots and logs were captured at each milestone, providing visual evidence of progress and verification of success.

Set Up a Web App Using AWS and VS Code

The first stage of the project focused on launching a web application on AWS by provisioning an Elastic Compute Cloud (EC2) instance, connecting to it through Secure Shell (SSH), and configuring remote development access with Visual Studio Code (VS Code). This step was crucial in establishing a foundation for the larger CI/CD pipeline, as it provided a real-world server environment to host and manage the application. By successfully completing this stage, I ensured that subsequent steps like version control, package management, and continuous integration would build on a working deployment environment.

I began by provisioning an EC2 instance, which served as a virtual server in the cloud. This instance provided the infrastructure necessary to run and test my application remotely. After launching the instance, I generated and downloaded a key pair in the form of a .pem file. This key pair was essential for secure authentication, with the public key stored on the server and the private key stored locally on my machine. To secure the key, I adjusted its permissions by running the command chmod 400 keypair.pem, ensuring that only I could read the file and preventing unauthorized access.

Once the instance was running, I connected to it using SSH directly from the terminal in VS Code. The connection command included the key pair and the public IPv4 DNS of the instance, which acted as its unique network address. Establishing this connection allowed me to access the remote machine securely and interact with it as if it were running locally. The successful login validated the key pair setup and demonstrated that I could now manage files, install packages, and run processes on the EC2 instance. Screenshot captured during this stage illustrate the EC2 configuration, key pair setup, and the first successful SSH login (Figure 2).

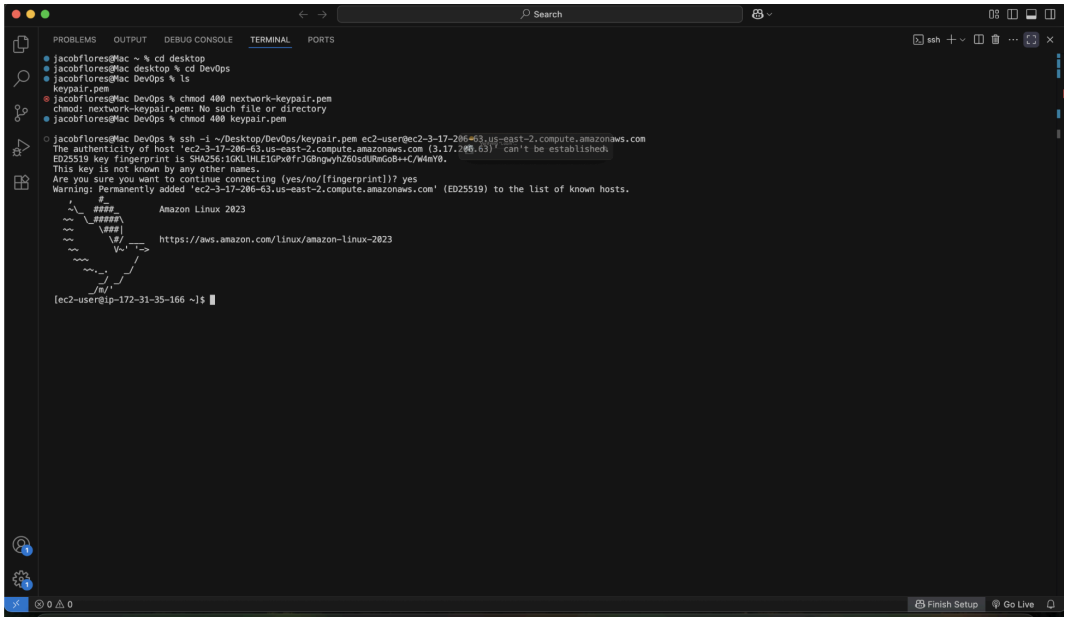

Figure 2: *Terminal configuration*

After connecting to the instance, the next task was to prepare the development environment. I installed Java (Amazon Corretto 8) to provide the runtime required for building and executing Java-based applications. In addition, I installed Apache Maven, a build automation and project management tool widely used in Java projects. Maven simplified the process of generating a structured project, managing dependencies, and compiling code. To confirm successful installation, I ran version checks for both Java and Maven, verifying that the EC2 instance was properly configured for application development.

With the environment ready, I generated my first web application using Maven's archetype feature. Running the Maven command created a structured Java project, complete with a pom.xml file for dependency management and a src directory that contained all source files. Inside the src folder, the webapp subdirectory included key resources such as HTML, CSS, and JavaScript files, while the main application logic was implemented through Java files.

To make remote development seamless, I installed the "Remote – SSH" extension in VS Code. This extension enabled me to connect directly to the EC2 instance, browse and edit files, and manage the project as if it were hosted locally. With this integration, I could open and modify files directly within the VS Code editor. One of my first changes was editing the index.jsp file—a JavaServer Page file that combines HTML with Java code to create dynamic web content. I replaced the default placeholder code with a custom snippet that displayed my name (Figure 3).

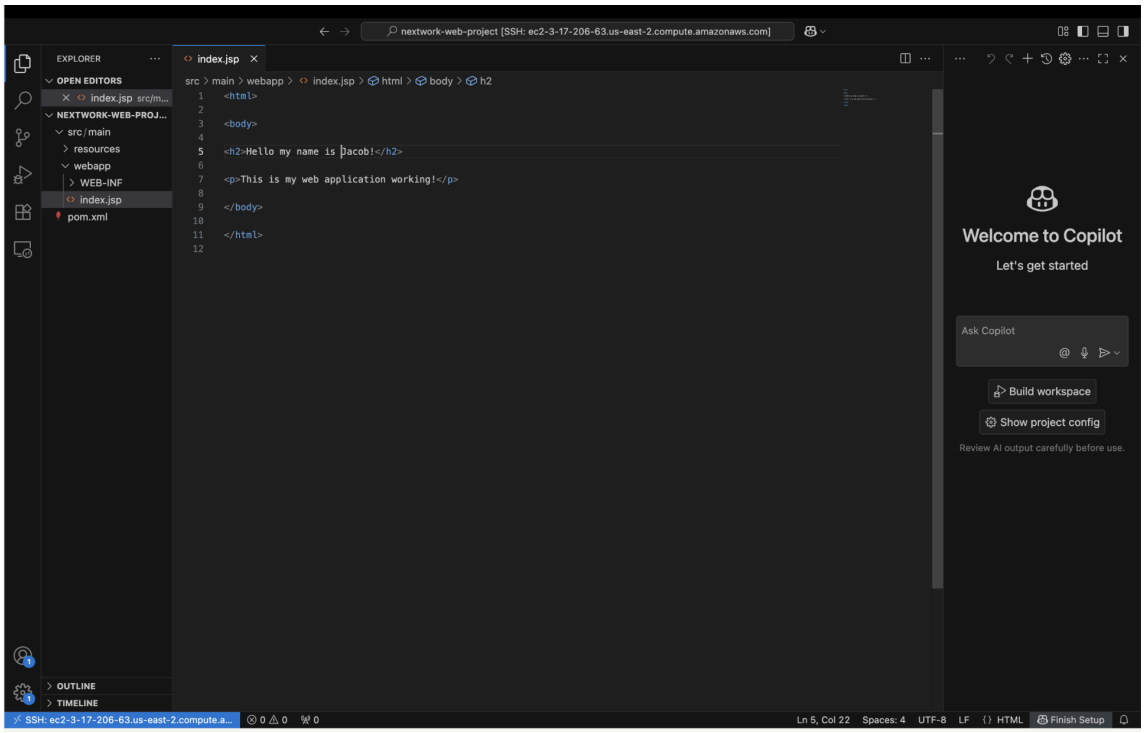

Figure 3: *Project directory structure*

Completing this stage provided me with a functional cloud-based development environment. The EC2 instance was configured with the necessary tools, connected to VS Code for remote editing, and running a Java web application that I could customize. Beyond the technical steps, this stage taught me the value of secure access management through key pairs, the convenience of remote development with VS Code, and the power of Maven in quickly setting up a structured Java application.

Connect a GitHub Repo with AWS

Once the web application was successfully running on AWS, the next step was to connect the project to GitHub in order to implement version control. This stage was essential because it established the foundation for tracking code changes, managing versions, and integrating GitHub commits directly into the CI/CD pipeline. By storing the project in a GitHub repository, I ensured that my code would be securely backed up in the cloud, easily shareable, and ready to trigger automated builds and deployments in later stages.

The process began by installing Git on the EC2 instance, which allowed me to manage version history and communicate with GitHub from within the remote server. To do this, I first updated the system's software packages using the command sudo dnf update -y and then installed Git with sudo dnf install git -y. After confirming the installation with git --version, I configured my Git identity by assigning my username and email address to commits. This ensured that all changes would be properly attributed in the project history. Screenshot taken at this point demonstrate the successful installation of Git and the initial repository setup (Figure 4).

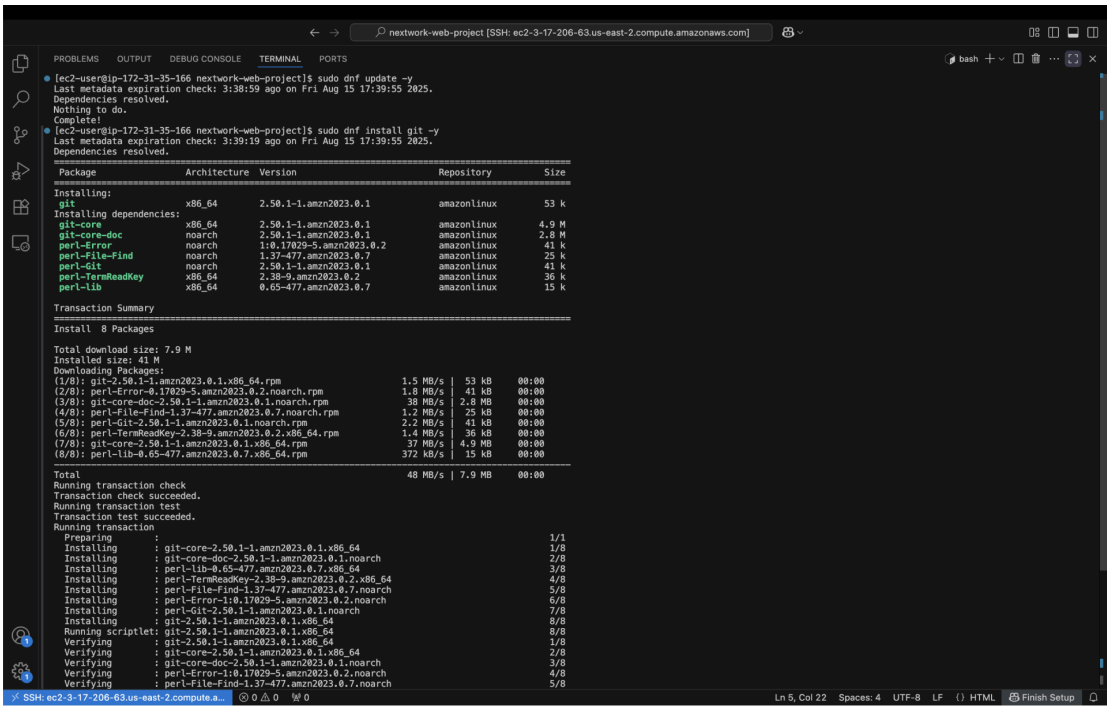

Figure 4: *Git and the initial repository setup*

Next, I initialized a Git repository inside my web application directory using the git init command. This created the necessary structure for version control and enabled Git to begin tracking changes in my project. I then linked the local repository on the EC2 instance to a remote repository on GitHub using git remote add origin <repository URL>. This connection created a direct pathway between my cloud-based project and GitHub, allowing me to push and pull updates as needed.

Once the repository was connected, I staged changes using git add ., committed them with git commit -m "Initial commit with web application setup", and pushed them to the remote repository using git push -u origin master (Figure 5).

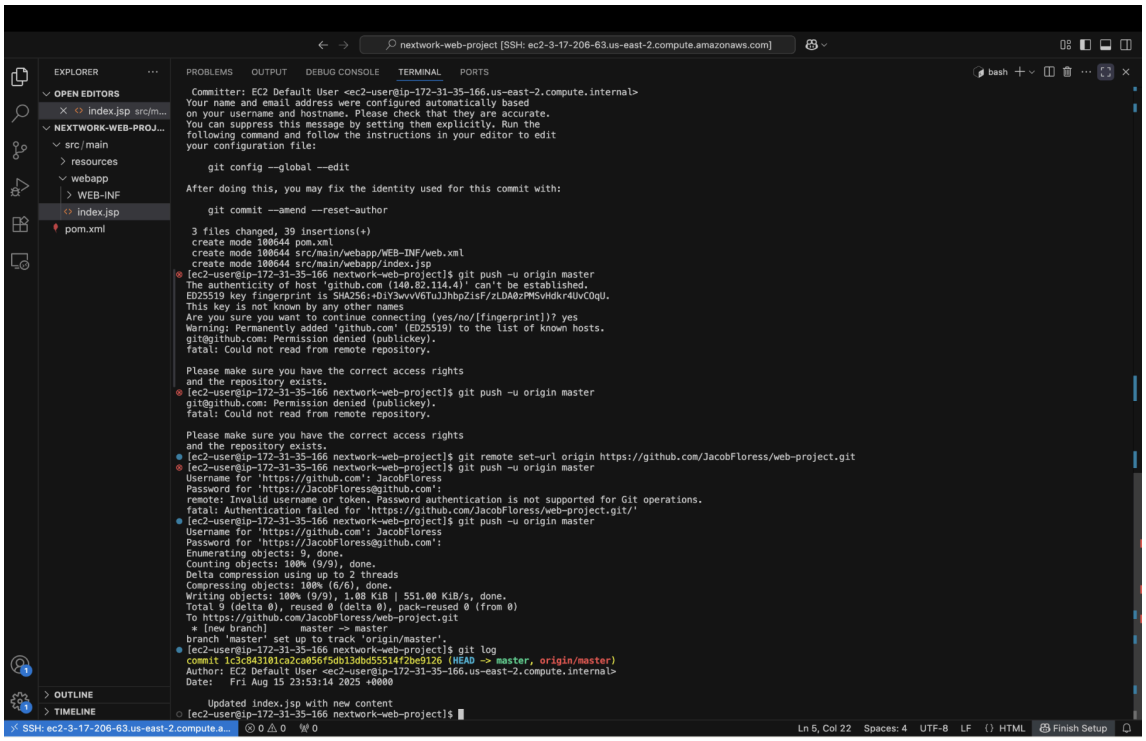

Figure 5: *Terminal commands*

At this point, GitHub requested authentication. Since account passwords are no longer supported for HTTPS connections, I created a personal access token by navigating to my GitHub account's Developer Settings. There, I generated a token with the appropriate scope for repository access, set an expiration period, and used it in place of a password during authentication. With the token in place, I was able to successfully push my code to GitHub, where the initial commit appeared in the repository (Figure 6).

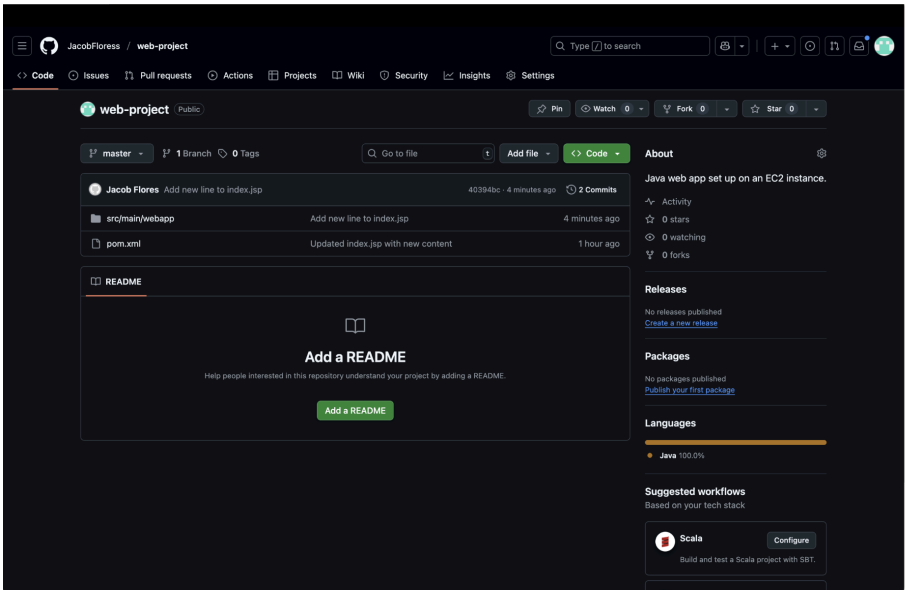

Figure 6: *Github repository*

After the initial setup, I tested the workflow by making changes to the index.jsp file on the EC2 instance. I staged the changes with git add ., committed them with a descriptive message, and pushed the update to GitHub. Reviewing the repository confirmed that the changes were reflected immediately.

Using the git log command, I was able to view the history of commits, including their unique IDs, timestamps, and associated author information. I also experimented with creating branches, which provided a way to work on isolated features without impacting the stability of the main codebase. These explorations gave me a stronger understanding of how collaborative teams can work efficiently with GitHub. With this setup, any future commits would become the trigger point for downstream automation, making this step an indispensable bridge between development and deployment.

Secure Packages with CodeArtifact

With the application version-controlled and stored in GitHub, the next goal was to implement secure package management using AWS CodeArtifact. This service provides a centralized repository for dependencies, ensuring that applications pull libraries from a controlled, secure, and reliable source instead of directly from the internet. Integrating CodeArtifact into the workflow not only enhanced security but also improved consistency and efficiency, as dependencies could be cached and reused across builds.

I began by creating a CodeArtifact domain and repository. The domain served as the umbrella under which repositories are managed, allowing me to define security and access policies at a higher level. Within this domain, I configured a repository linked to Maven Central, which acted

as the upstream source. This setup ensured that if my project required a package not yet stored in CodeArtifact, it would fetch it from Maven Central, cache it locally, and make it available for future use. A screenshot of the repository setup illustrates how CodeArtifact was configured with both local storage and upstream connections (Figure 7).

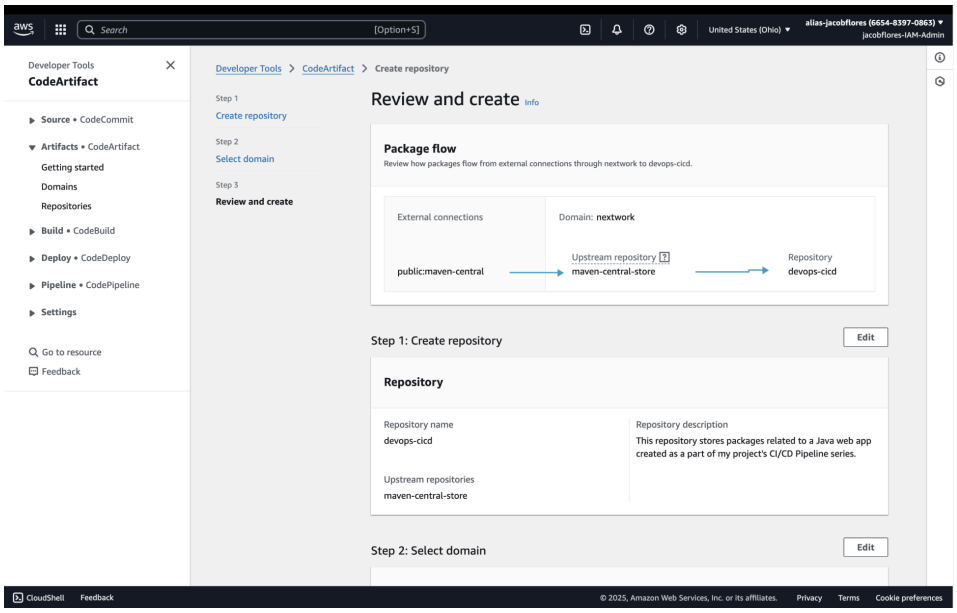

Figure 7: *AWS CodeArtifact Repository Setup*

To grant my EC2 instance permission to interact with CodeArtifact, I attached an IAM role with a specific JSON policy. This policy included permissions to retrieve authorization tokens, discover repository endpoints, read package versions, and request service bearer tokens.

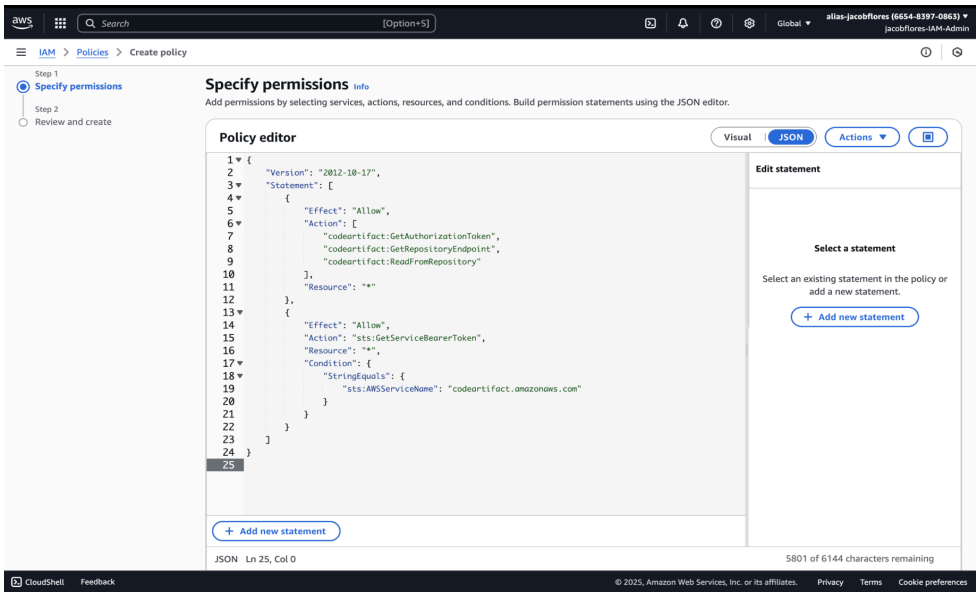

Figure 8: *JSON policy implementation*

Initially, I encountered an error when attempting to retrieve the CodeArtifact authorization token, as the EC2 instance did not yet have the correct IAM role attached. The issue was resolved by ensuring the IAM role was properly assigned. Once the role was recognized, I successfully retrieved the token using the AWS CLI. This experience highlighted the importance of IAM roles as a best practice, since they provide temporary and least-privilege credentials without embedding long-term access keys directly into the system.

After securing access, I integrated CodeArtifact with Maven. This required updating the settings.xml file on the EC2 instance to include the CodeArtifact repository URL and authentication details. The settings.xml file instructed Maven to request a fresh authorization token when connecting, ensuring that all dependencies would be fetched through CodeArtifact instead of directly from Maven Central. With the configuration complete, I compiled my project using Maven. The build process successfully downloaded required dependencies from the repository, verified them, and stored them locally on the EC2 instance.

To validate the integration, I revisited the CodeArtifact console and confirmed that the required Maven packages had been cached in the repository. This confirmed that the system was working as intended—dependencies were fetched through CodeArtifact, cached securely, and made available for subsequent builds. Screenshots taken at this stage show both the updated settings.xml file and the populated CodeArtifact repository containing the cached packages.

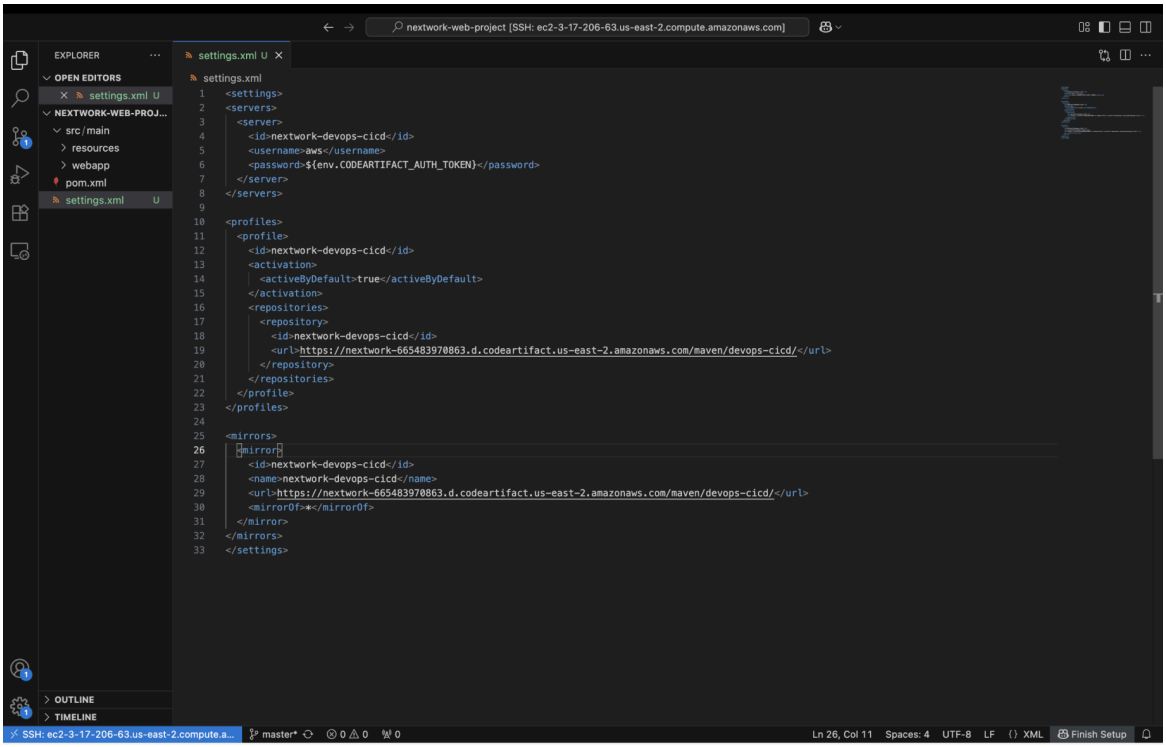

Figure 9: *Updated settings.xml file*

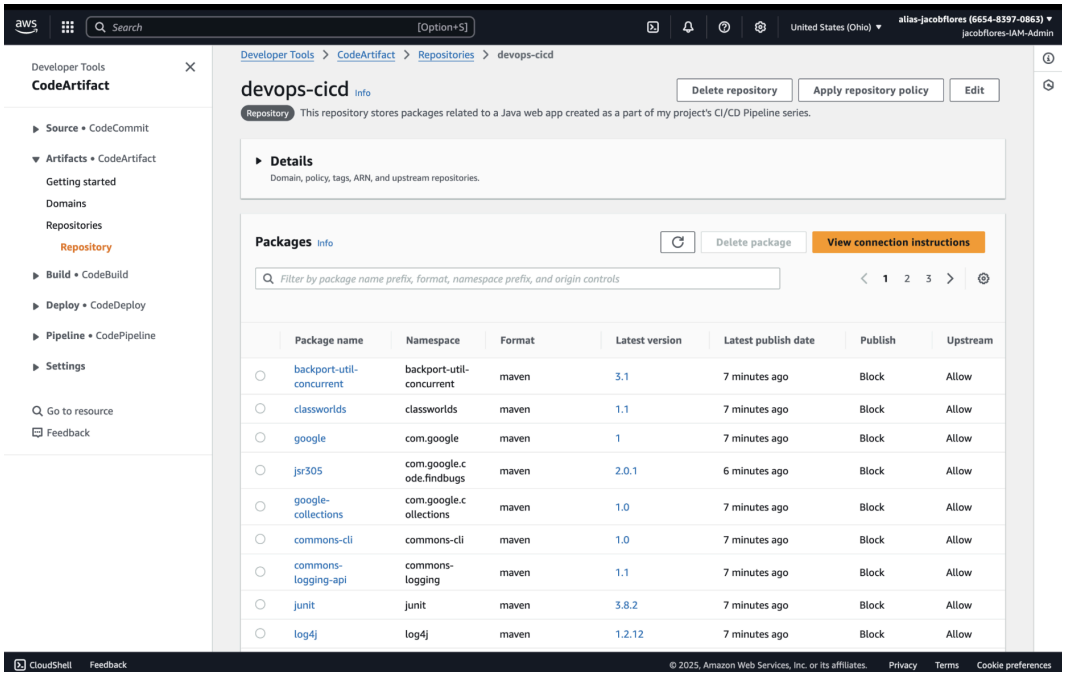

Figure 10: *CodeArtifact repository containing the cached packages*

The successful integration of CodeArtifact provided a strong security layer for dependency management and set the stage for automating builds with AWS CodeBuild. Beyond the technical configurations, I gained valuable insight into the importance of centralized package management in enterprise DevOps workflows. By ensuring that all dependencies flow through a secure and controlled channel, teams can reduce vulnerabilities, standardize builds, and guarantee consistency across development, testing, and production environments.

Continuous Integration with CodeBuild

With dependency management secured through CodeArtifact, the next step was to automate the build process using AWS CodeBuild. Continuous Integration (CI) is a cornerstone of DevOps, and CodeBuild provided the mechanism to automatically compile, test, and package my web application every time a new change was pushed to GitHub. This automation reduced the need for manual builds, ensured that code was consistently validated, and created deployable artifacts ready for the next stage of the pipeline.

The first task was to create a new CodeBuild project in the AWS Management Console. I configured GitHub as the source provider and connected my repository using a GitHub App integration. This approach was more secure than personal access tokens, since AWS managed the authentication, eliminating the need for me to manually handle sensitive credentials. To facilitate this connection, I also set up CodeConnections, which acted as the secure bridge between CodeBuild and GitHub. Screenshot from this step show the project configuration and the linked repository (Figure 11).

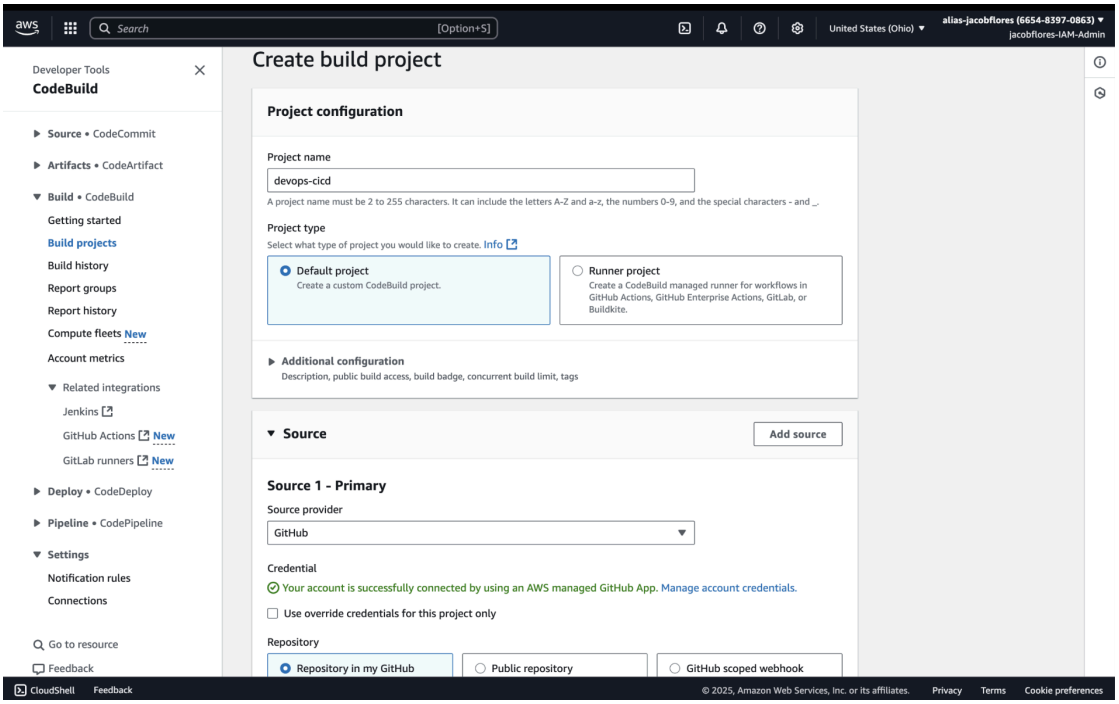

Figure 11: *CodeBuild Project configuration*

Next, I defined the build environment. I selected on-demand provisioning with an Amazon Linux managed image, using Corretto 8 as the runtime since it matched the requirements of my Java web application. I chose EC2 as the compute type and created a dedicated service role for CodeBuild. This role ensured that the build process had the necessary permissions to interact with other AWS services, such as S3 for artifact storage and CodeArtifact for dependency retrieval.

The artifacts were configured to be stored in an S3 bucket. Specifically, CodeBuild was instructed to generate a Web Application Archive (WAR) file containing the compiled code, libraries, and resources of my project. This file would later be deployed by CodeDeploy to an EC2 instance. To optimize artifact storage, I configured the output as a compressed ZIP package, which made uploads faster, reduced storage costs, and kept files organized.

One of the most critical parts of CodeBuild configuration was the buildspec.yml file. This YAML file defined the different phases of the build:

- **Install phase:** Installed Java 8.
- **Pre-build phase:** Retrieved a new CodeArtifact authentication token to ensure dependencies could be pulled securely.
- **Build phase:** Compiled the code with Maven, resolving dependencies through CodeArtifact.
- **Post-build phase:** Packaged the compiled application into a deployable WAR file and prepared it as an artifact.

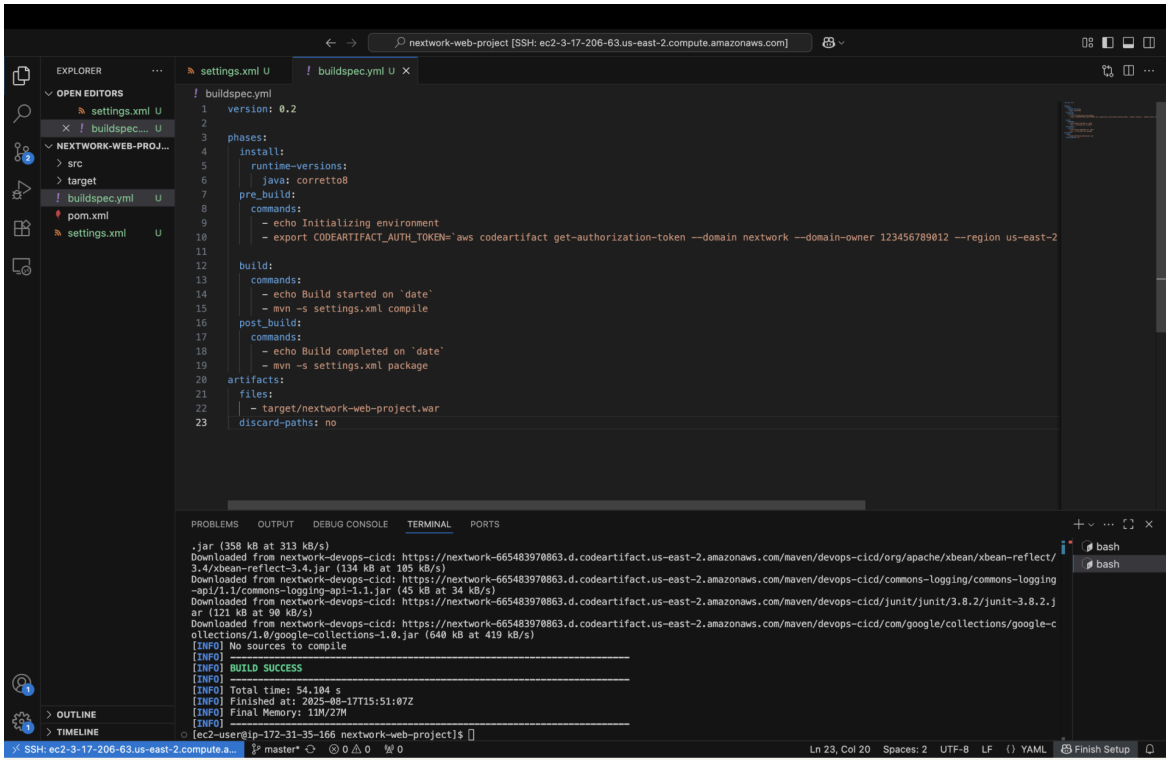

Figure 12: *buildspec.yml file code configuration*

Initially, the build failed because CodeBuild could not locate the buildspec.yml file. Without it, the service did not know which commands to execute. To fix this, I created and uploaded the file to the root of my GitHub repository, then triggered the build again. This time, the error message changed, indicating progress but revealing a new issue: CodeBuild was unable to access CodeArtifact to fetch dependencies.

The dependency issue was caused by insufficient IAM permissions in the CodeBuild service role. To resolve this, I attached the **codeartifact-consumer-policy**, which granted CodeBuild explicit permissions to authenticate and download dependencies from CodeArtifact. After updating the role, I re-ran the build. This time, the process completed successfully. Reviewing the CloudWatch logs confirmed that Maven was able to download dependencies, compile the code, and package the WAR file.

Verifying the results in the S3 bucket confirmed that the build artifact, named devopscicd-artifact, had been generated and uploaded correctly. This outcome proved that the entire build pipeline—from GitHub commit to compiled artifact—was working as intended. Screenshot of the successful CodeBuild execution, and the CloudWatch log output illustrate the process end to end (Figure 13).

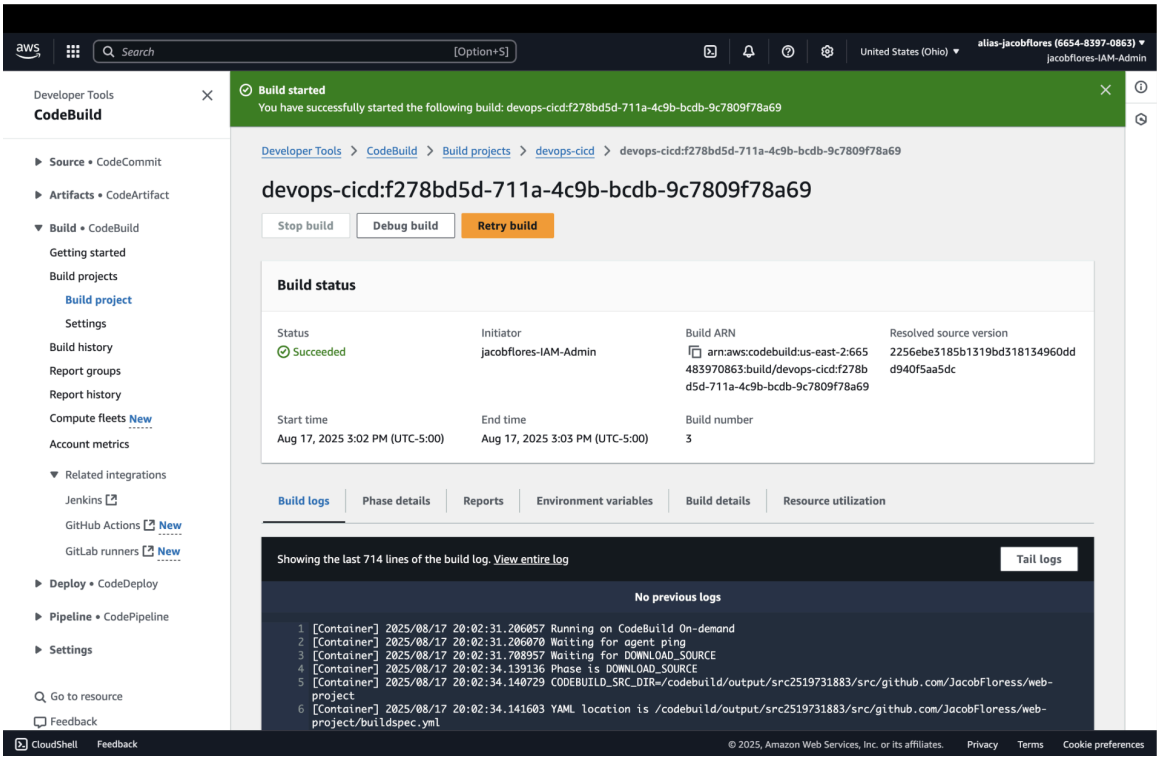

Figure 13: *Successful CodeBuild execution*

CodeBuild ensured that every new code change would be validated and packaged without manual intervention, creating a reliable flow of artifacts for deployment. This experience deepened my understanding of build automation, IAM permissions, and artifact management, while reinforcing the importance of logs and error tracking for debugging in real-world DevOps workflows.

Deploy a Web App with CodeDeploy

With the build process automated and artifacts reliably stored in S3, the next step was to automate deployment using AWS CodeDeploy. This service allowed me to release new versions of the application to an EC2 instance in a consistent, repeatable, and automated manner, reducing downtime and minimizing human error. CodeDeploy was a key milestone because it bridged the gap between automated builds and a live production-like environment.

To prepare for deployment, I provisioned an EC2 instance and a dedicated Virtual Private Cloud (VPC) to act as the production environment. Instead of configuring these resources manually, I used an AWS CloudFormation template. By describing the infrastructure as code, I ensured consistency across deployments and laid the groundwork for a scalable, repeatable environment. The CloudFormation stack created both compute and networking resources, including the EC2 instance, subnet, internet gateway, route tables, and a security group. Screenshot of the CloudFormation template and stack outputs illustrate this provisioning process (Figure 14).

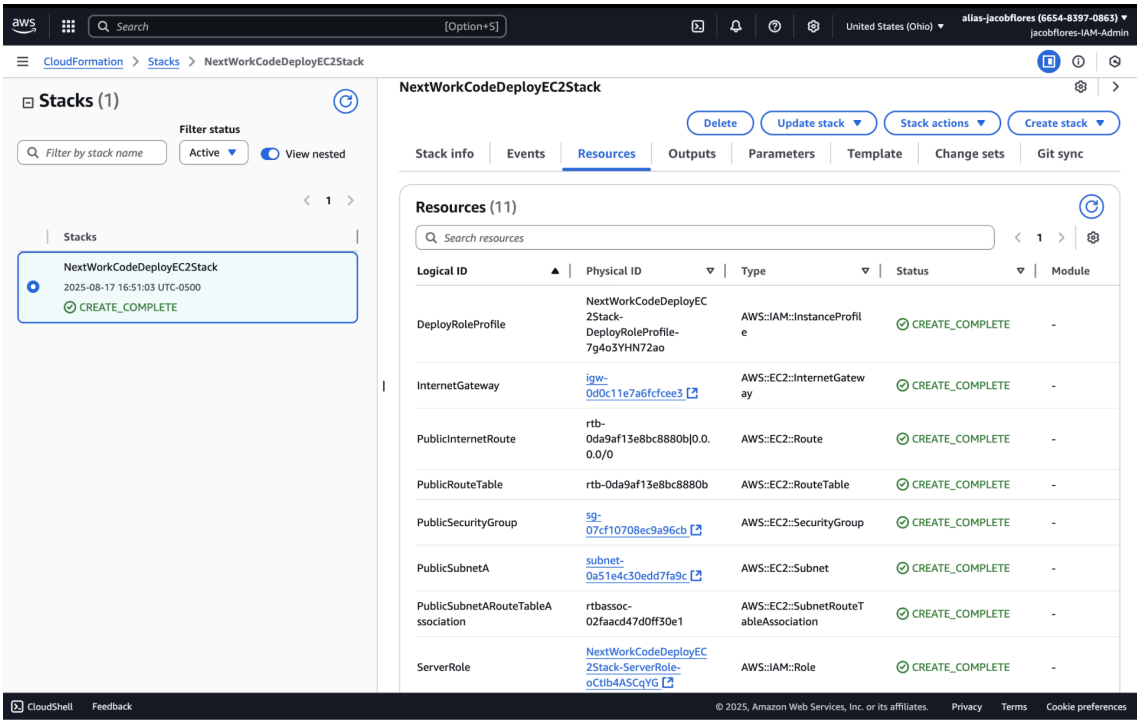

Figure 14: *CloudFormation stack outputs*

Deployment with CodeDeploy required automation scripts to manage application installation and lifecycle events. I wrote three main scripts:

1. **install_dependencies.sh** – Installed Apache Tomcat and the Apache HTTP server, then configured Apache as a reverse proxy to forward traffic from port 80 to Tomcat. This setup ensured the Java web application was accessible publicly while running on the EC2 instance.
2. **start_server.sh** – Started the Tomcat and Apache services and configured them to restart automatically after a system reboot, ensuring high availability.
3. **stop_server.sh** – Safely stopped Tomcat and Apache services, checking first if they were running before shutting them down, which reduced the risk of deployment errors.

These scripts were referenced in the **appspec.yml** file, which acted as the deployment blueprint. The appspec.yml defined which files should be copied to the EC2 instance and in which order the lifecycle scripts should run. To ensure CodeDeploy had access to these files, I updated the buildspec.yml file in the build stage to package the appspec.yml file and the scripts folder into the artifact uploaded to S3.

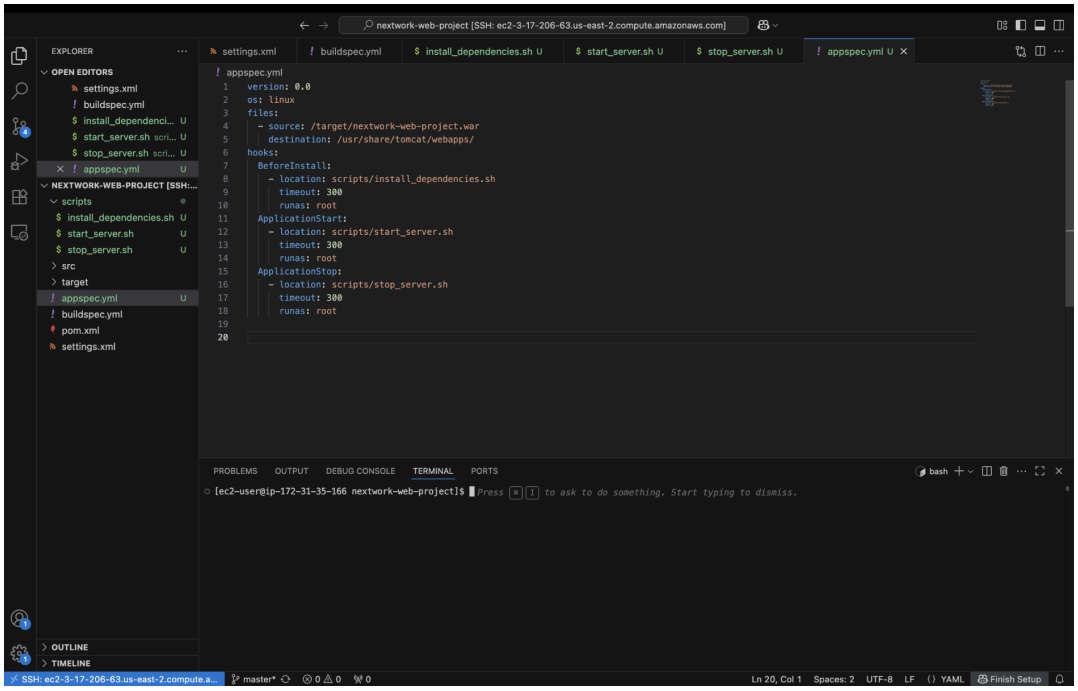

Figure 15: *appspec.yml file configuration*

Once the deployment package was ready, I created a new CodeDeploy application and a deployment group. The deployment group targeted the EC2 instance using tags defined in the CloudFormation template, allowing CodeDeploy to identify and deploy to the correct environment automatically. I also created a dedicated IAM role for CodeDeploy so it could securely interact with S3, EC2, and other AWS services on my behalf.

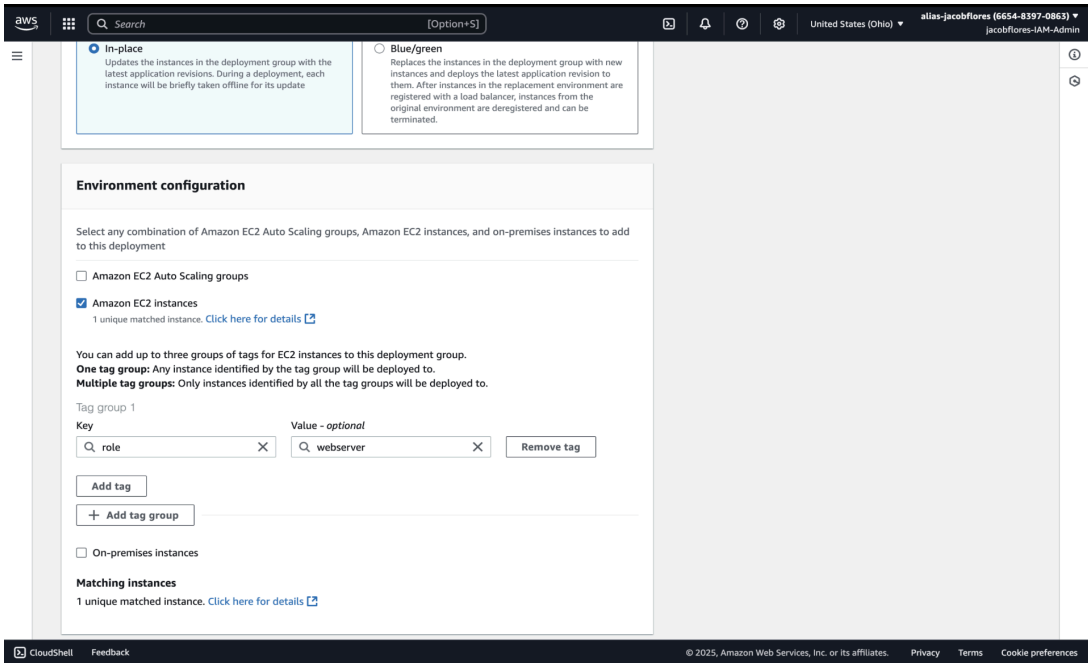

Figure 16: *Environment configuration*

CodeDeploy supports different deployment strategies such as **AllAtOnce** (fastest but riskiest), **OneAtATime** (safest but slowest), and **HalfAtATime** (balanced). For this project, I experimented with the AllAtOnce configuration, which updated the application on the EC2 instance in a single step. I also ensured the CodeDeploy agent was installed and running on the instance so it could receive instructions, execute lifecycle events, and report status back to the AWS console.

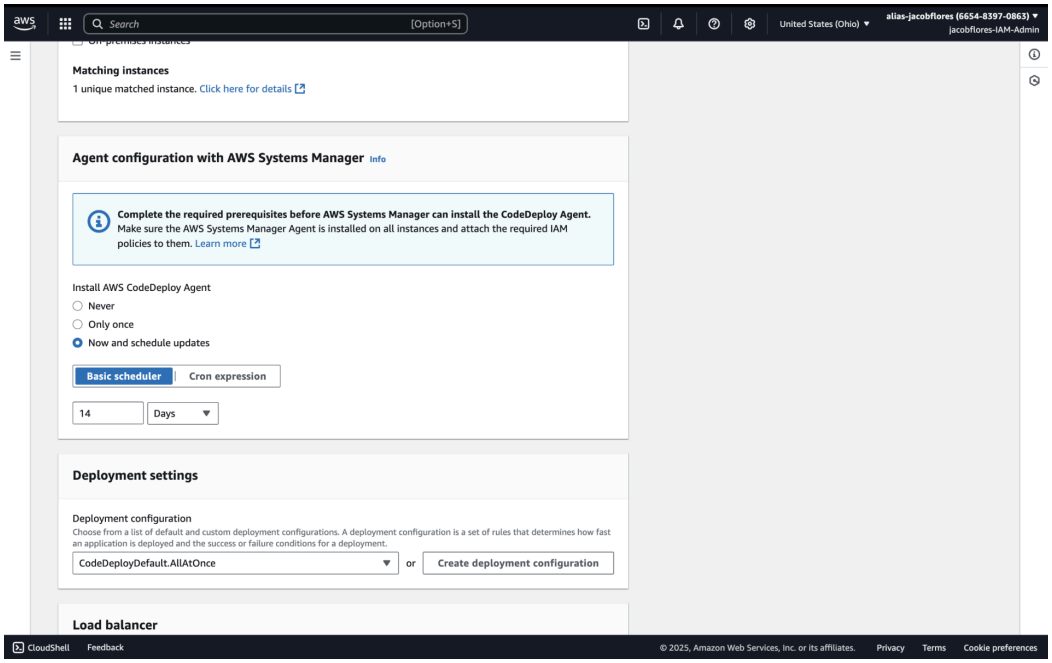

Figure 17: *Agent configuration and Deployment settings*

During the first deployment attempt, I encountered errors related to missing files in the revision package. This was resolved by confirming that the appspec.yml and all scripts were properly included in the artifact uploaded by CodeBuild. After correcting this oversight, the deployment succeeded.

CodeDeploy logs showed each lifecycle event—**ApplicationStop**, **BeforeInstall**, **AfterInstall**, **ApplicationStart**, and **ValidateService**—executed successfully in sequence. To confirm the deployment, I accessed the EC2 instance's Public IPv4 DNS over HTTP and verified that the web application loaded correctly. Screenshot of the running application in a browser serve as final validation of this stage (Figure 18).

Figure 18: *My application running successfully in a browser serve*

By combining CloudFormation, lifecycle scripts, and CodeDeploy, I experienced firsthand how DevOps practices bring consistency, resilience, and automation to real-world deployment processes.

Build an End-to-End CI/CD Pipeline with AWS CodePipeline

After setting up automated builds with CodeBuild and automated deployments with CodeDeploy, the final step was to integrate all components into a single continuous integration and continuous delivery (CI/CD) pipeline using AWS CodePipeline. This service allowed me to fully automate the release process, ensuring that every code change pushed to GitHub flowed seamlessly through build, test, and deployment stages without manual intervention.

The pipeline was designed with three main stages: **Source, Build, and Deploy**.

In the **Source stage**, CodePipeline was connected directly to my GitHub repository through a webhook. This configuration ensured that any new commit pushed to the repository automatically triggered a pipeline execution. I set the pipeline to monitor the master branch so only production-ready code was deployed. Screenshot of the configuration show the GitHub connection and the initial source stage setup (Figure 19).

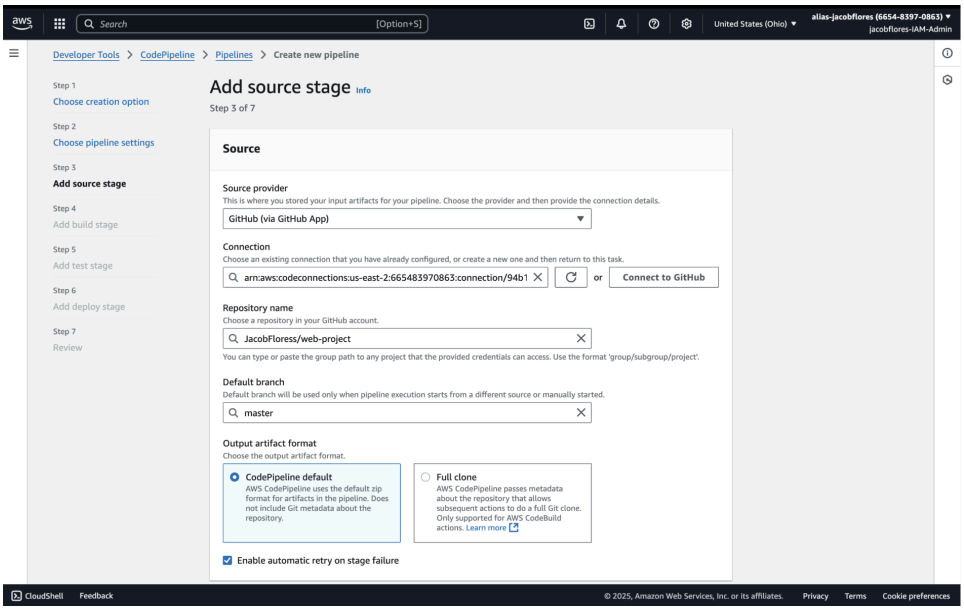

Figure 19: *Source stage configuration*

The **Build stage** used CodeBuild, which was already configured to compile the Java application, pull dependencies from CodeArtifact, and package the output into a Web Application Archive (WAR) file. CodePipeline passed the source artifact from the GitHub stage into CodeBuild as input, and once the build succeeded, the resulting WAR file was uploaded to S3 as the build

artifact. CloudWatch logs captured the build progress, confirming that each step in the buildspec.yml executed correctly (Figure 20).

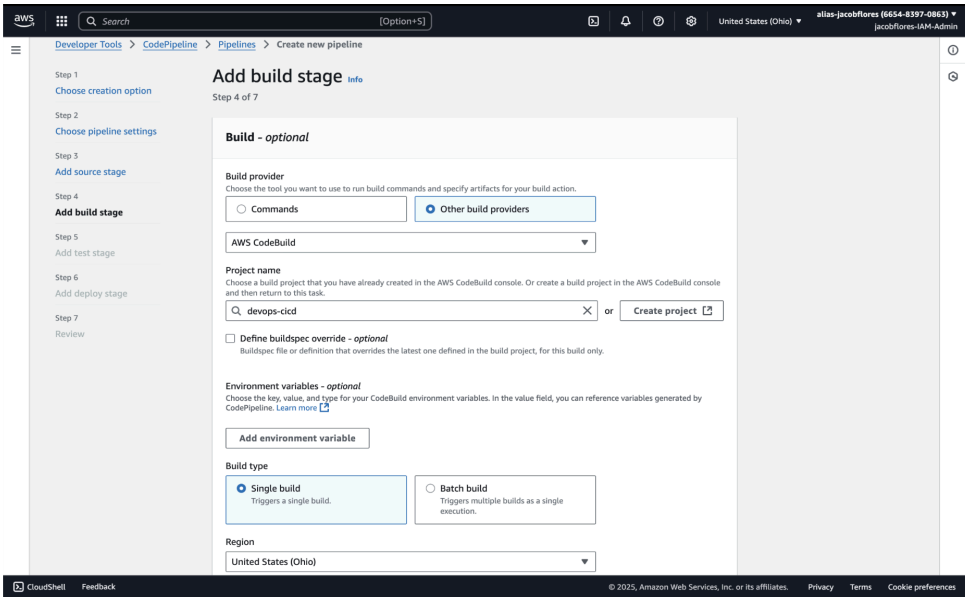

Figure 20: *Build stage configuration*

In the **Deploy stage**, CodePipeline handed the build artifact to CodeDeploy, which used the appspec.yml file and lifecycle scripts to release the new application version to the EC2 instance. The deployment group targeted the instance created through CloudFormation, and the CodeDeploy agent executed the stop, install, and start scripts to complete the release. CodePipeline monitored the deployment in real time and displayed success or failure at each step (Figure 21).

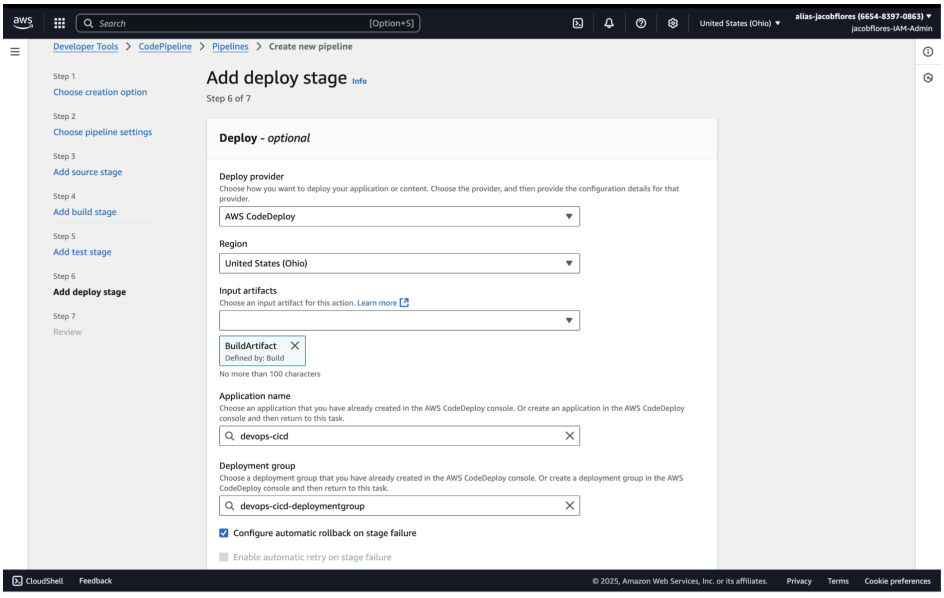

Figure 21: *Deploy stage configuration*

To improve efficiency, I configured the pipeline execution mode to **Superseded**, which ensured that if multiple commits were pushed in rapid succession, only the latest commit would be deployed. This avoided wasted resources on outdated builds and guaranteed that the most recent code always reached production. A dedicated IAM service role for CodePipeline granted secure permissions to orchestrate GitHub, CodeBuild, CodeDeploy, and S3 interactions.

Once the pipeline was active, I tested it by making a small change in my application. I modified the index.jsp file to include an additional line of text. As soon as I committed and pushed the change to the GitHub repository, the pipeline was triggered automatically. In the AWS Console, I could see the new execution appear, with each stage—Source, Build, and Deploy—progressing in sequence. The commit message was displayed alongside the execution, linking the pipeline activity back to the version control history.

After the deployment completed successfully, I verified the change by accessing the EC2 instance's Public IPv4 DNS in a browser. The updated text appeared in the application, confirming that the pipeline had correctly built and deployed the new version. Screenshots of the successful execution and the running application serve as final proof of the end-to-end automation (Figure 16).

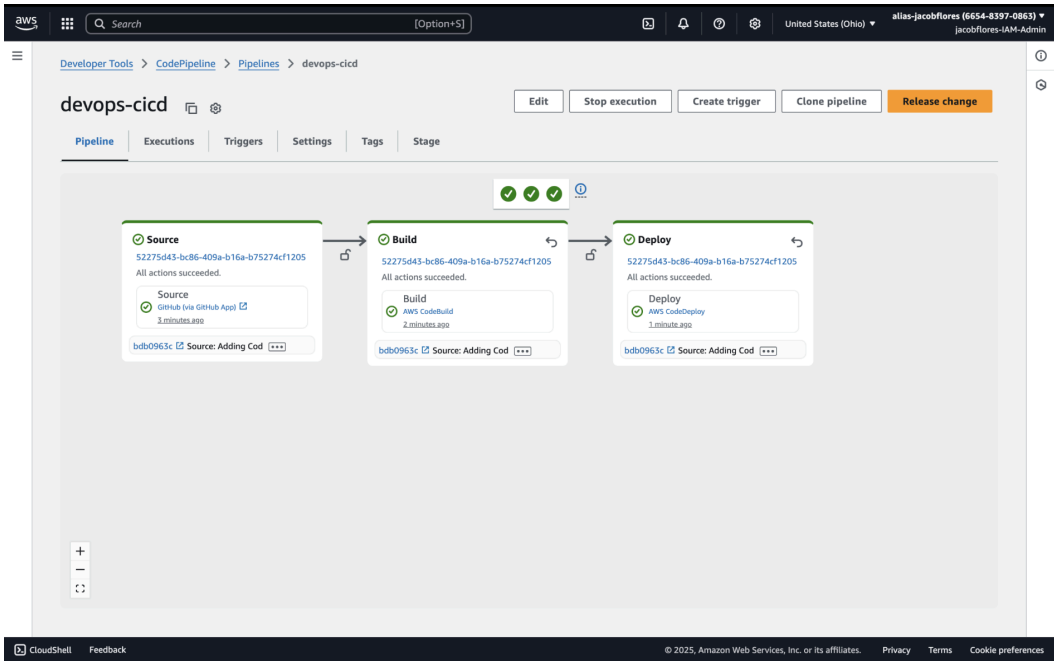

Figure 22: *CI/CD pipeline successful execution*

This stage took approximately six to seven hours to complete, largely because of the careful configuration of roles, permissions, and artifact handoffs between services. The most challenging part was ensuring that CodePipeline had the correct IAM permissions to interact with all the components, while the most rewarding moment was seeing a single GitHub commit cascade through the pipeline and automatically deploy to a live environment.

By completing this stage, I built a fully operational CI/CD pipeline that integrated GitHub, CodeBuild, CodeDeploy, and CodePipeline. This experience reinforced the importance of automation in modern software delivery, demonstrated the interplay between multiple AWS services, and gave me practical, portfolio-ready experience in building production-style pipelines from scratch.

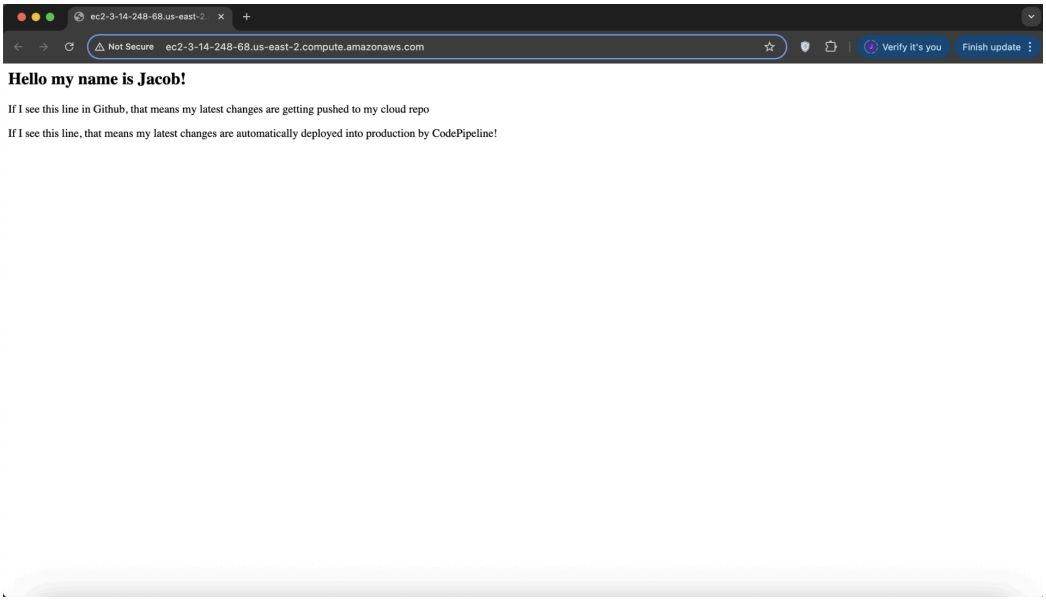

Figure 23: *Application running successfully*