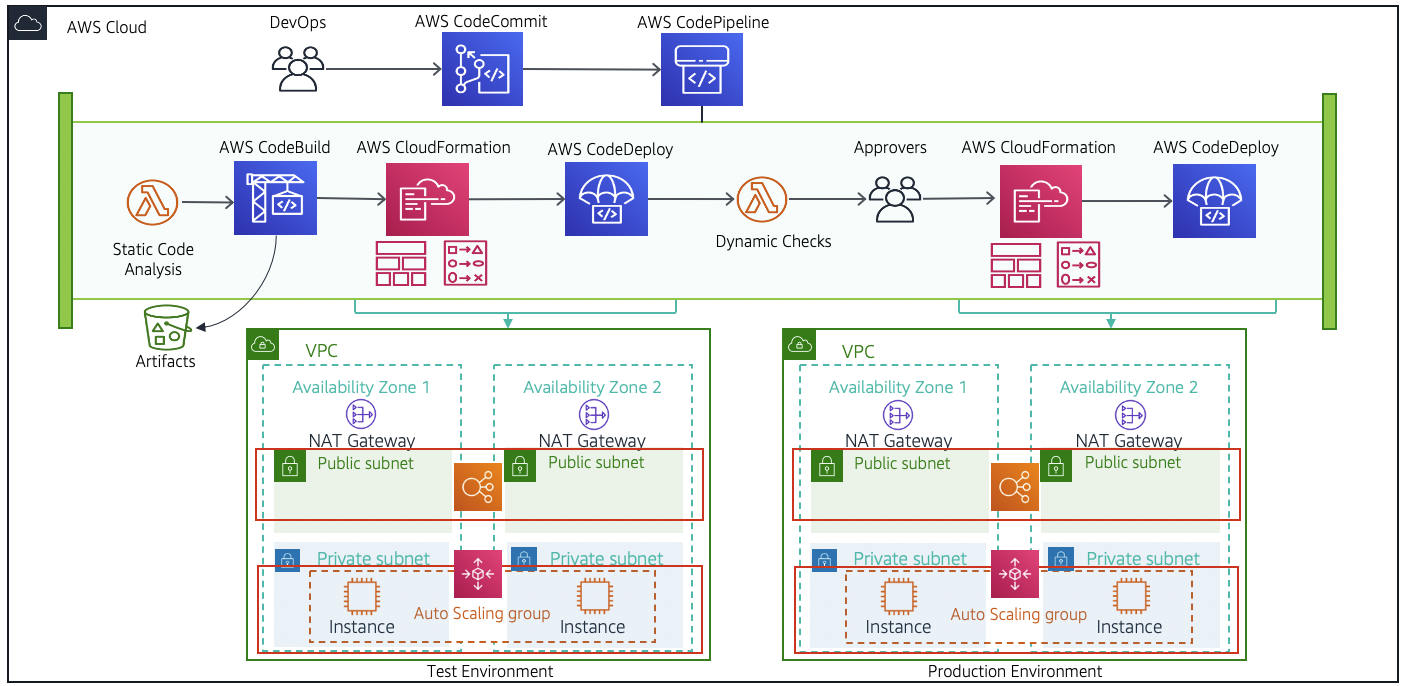
# Lab 6: Using AWS DevOps tools for CI/CD pipeline automations

We work on a release pipeline that has failed in the build stage

Objectives of this lab:

1. Understand the architecture of a release pipeline.
2. Perform basic troubleshooting of failed stages in AWS CodePipeline by locating, analyzing logs, and applying fixes as needed.
3. Adjust the configurations of an AWS infrastructure based on test results from the pipeline.
4. Validate and manually approve a change between stage transitions of a pipeline.
5. Add new actions to an existing AWS CodePipeline stage.

**NOTE -** X-Ray provides an end-to-end view of requests as they travel through your application, making it easier to identify performance bottlenecks and errors.



An **AWS CodeCommit** repository has already been created, and the code for the demo calculator web application has already been uploaded into the AWS CodeCommit repository.

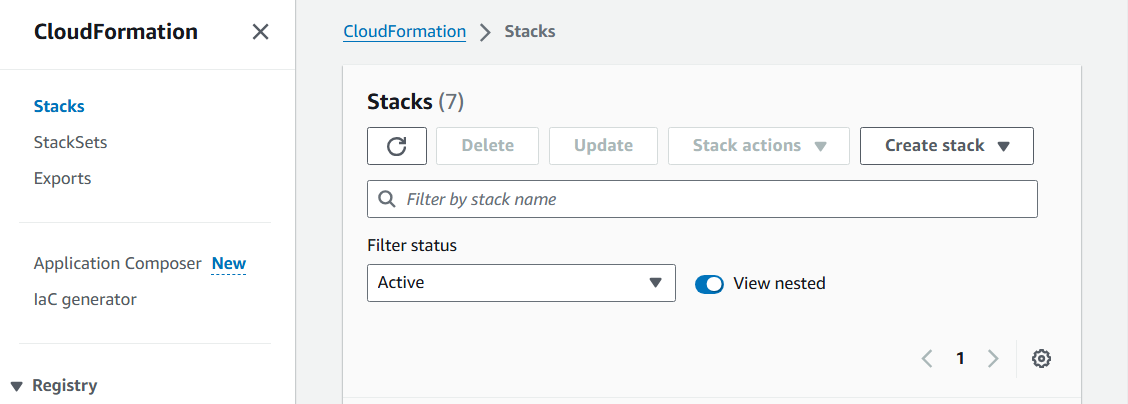
In the **Static\_Check stage**, a static analysis of the code is performed using an AWS Lambda function named CFNValidateLambda. The Lambda function uses regular expression language to find patterns and identify security group policy violations, then Lambda fails the pipeline.

**Build specification files**, commonly referred to as build spec files, are YAML formatted files used by AWS CodeBuild to define the commands and settings for build projects.

**Task 1 - Building the Pipeline**

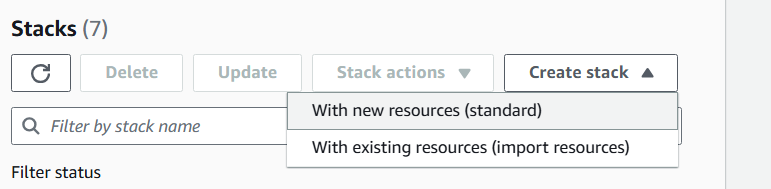
1.1 Search and open CloudFormation in AWS

AWS CloudFormation is a tool provided by Amazon Web Services (AWS) that helps you set up and manage AWS resources using code. Instead of manually configuring resources, you define what you need in a text file, and CloudFormation takes care of the rest.

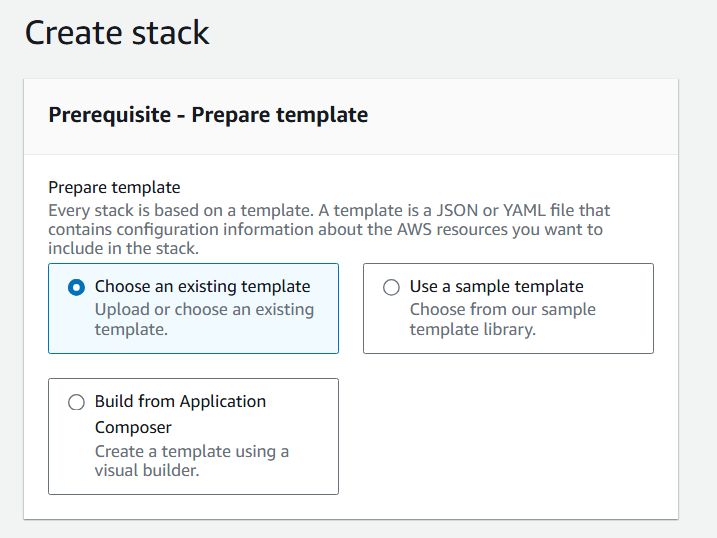


1.2 Click on create standard stack

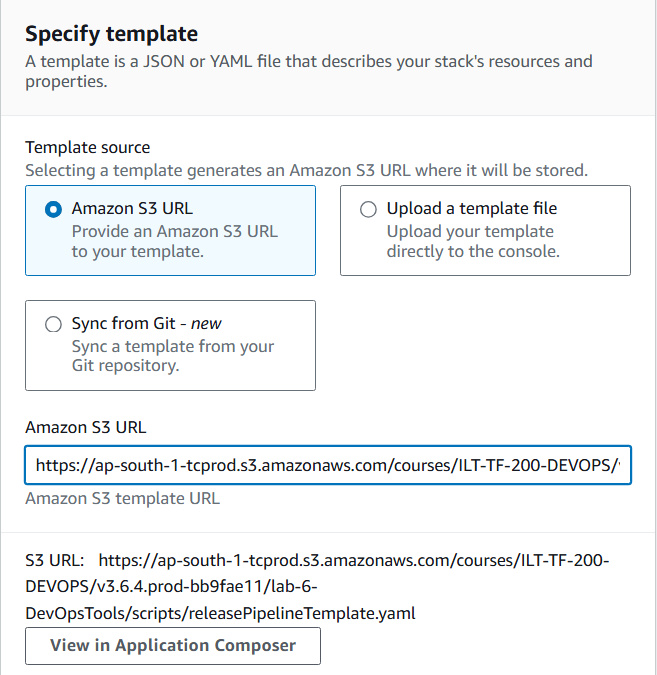
A stack is a collection of AWS resources that you can manage as a single unit. Stacks enable you to create, update, and delete related resources together, ensuring they are managed consistently.



1.3 For the template, choose existing stack

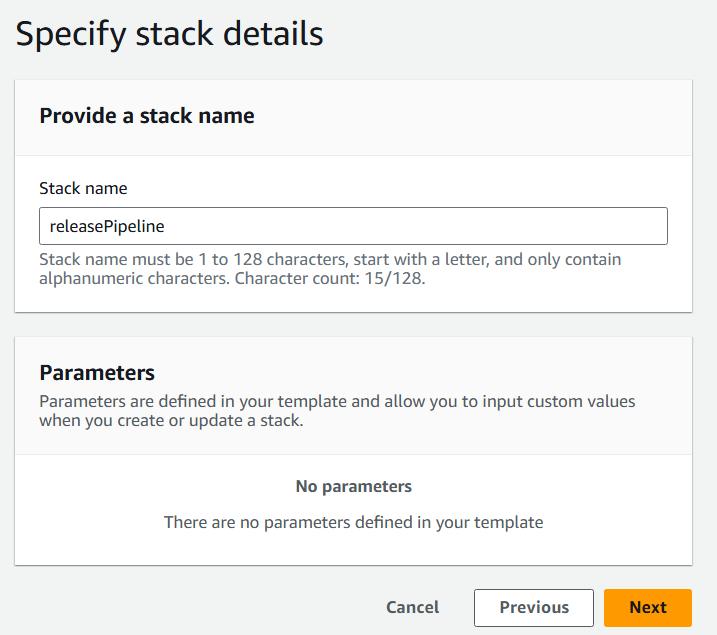


1.4 Specify the template from S3 bucket, whose location is in lab manual

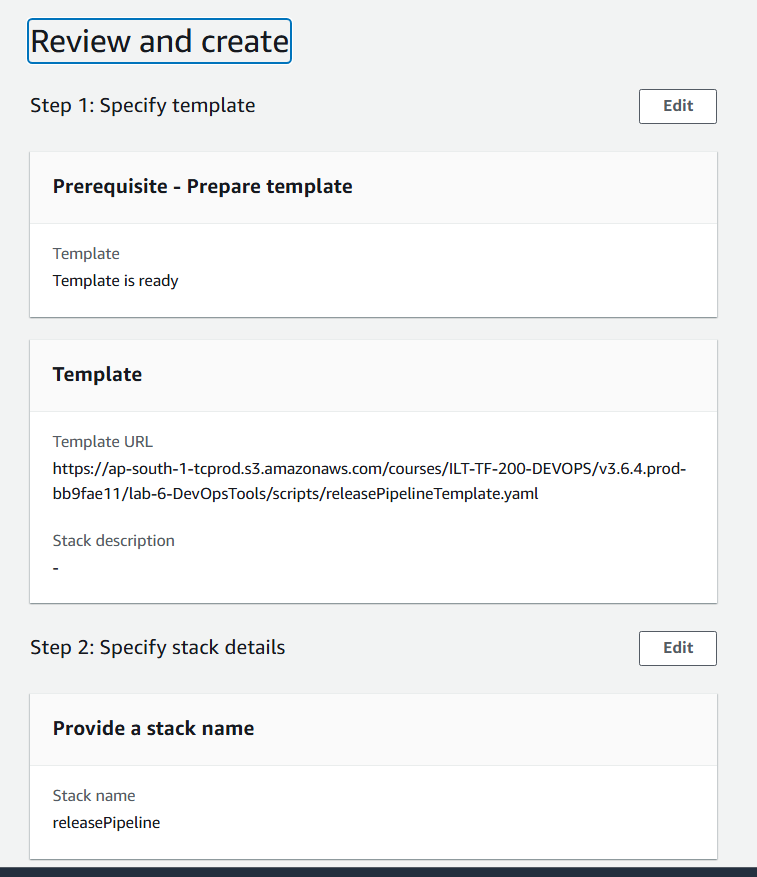


1.5 Click next

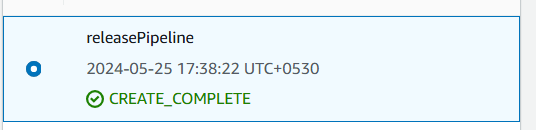
1.6 Give the stack name, and next



1.7 Review the inputs, and proceed

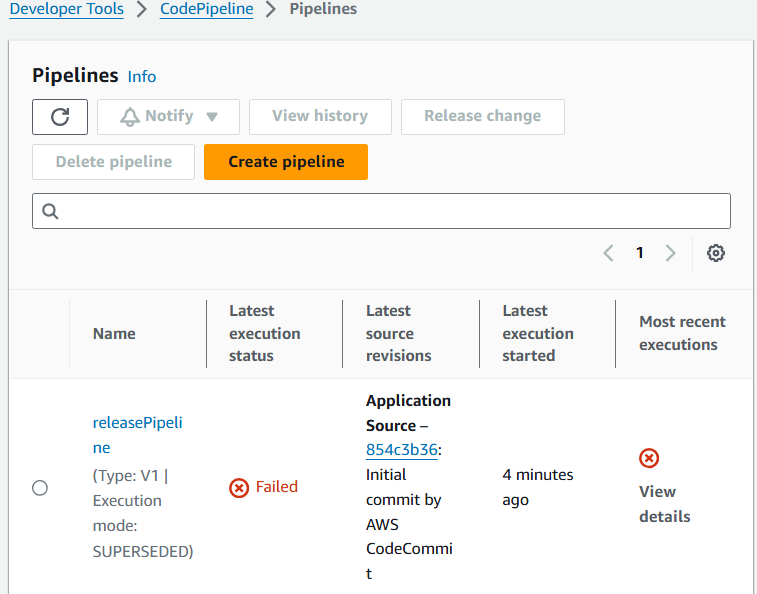


1.8 CloudFormation has finished and the pipeline is ready to use.

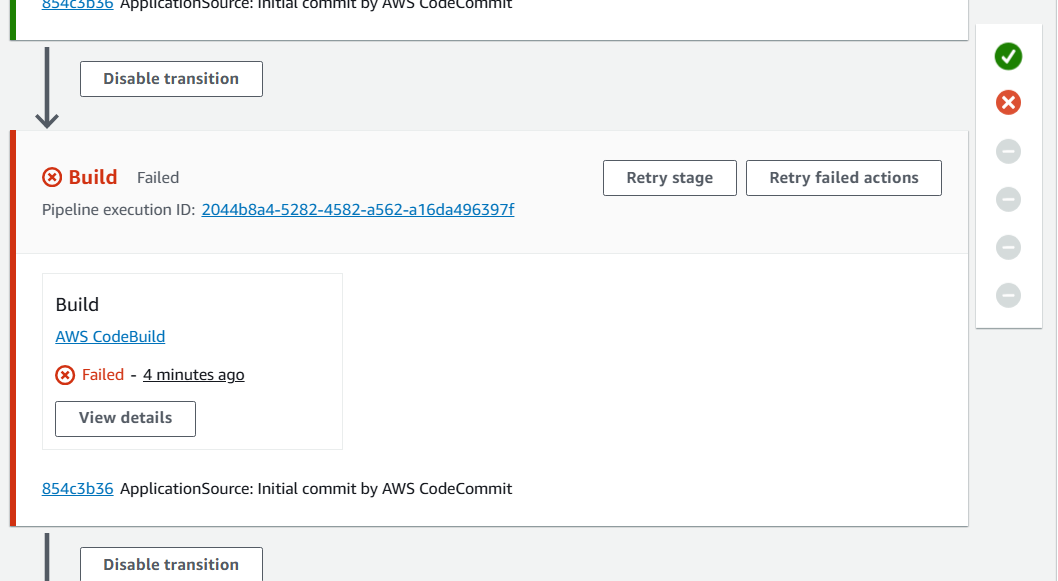


**Task 2 - Fixing the build stage**

2.1 Open CodePipeline, and choose release pipeline link

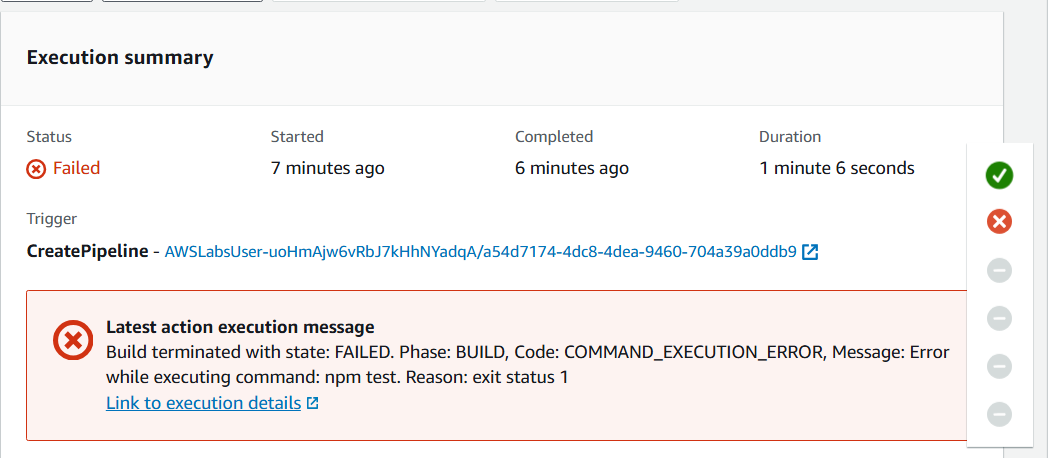


2.2 We see that the build phase for the pipeline has failed.

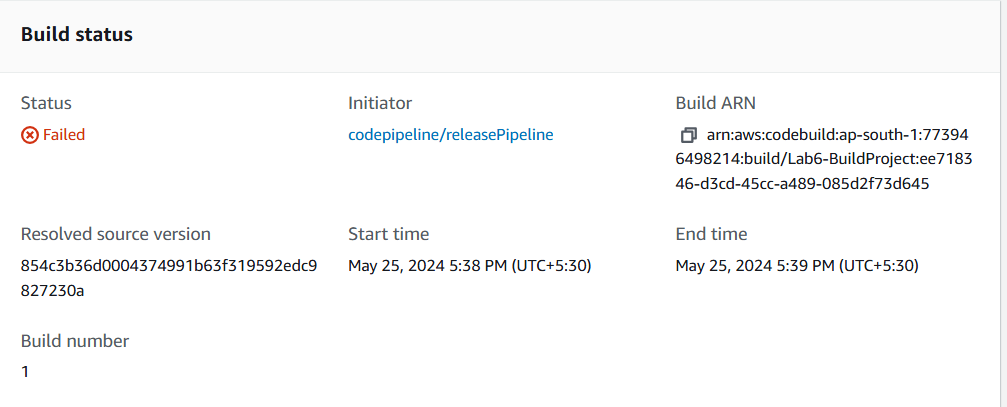


2.3 Open pipeline execution id, last snapshot

2.4 Select link to exec. Details

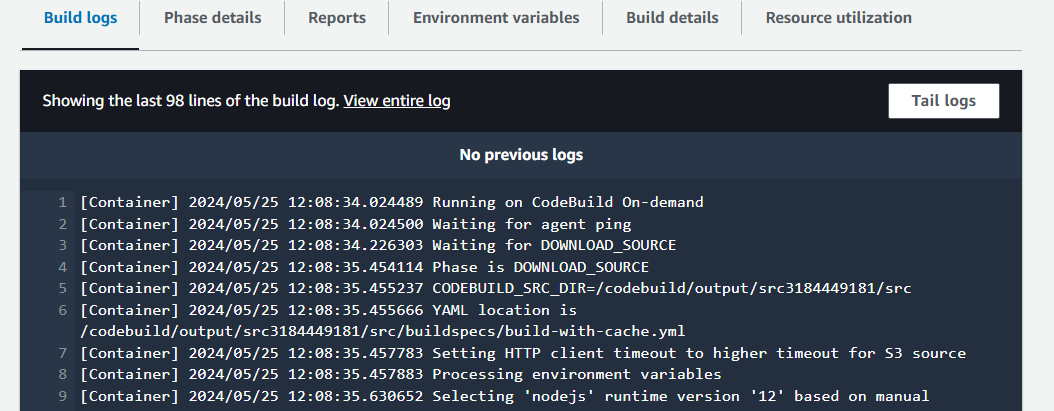


CodeBuild page opens and the CodeBuild page shows what happened during the stage’s action.

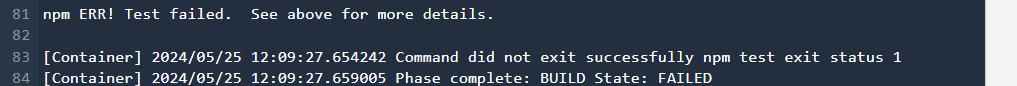


2.5 Scroll down to see the build logs

Entries are timestamped, indicate what ran, and where it ran.

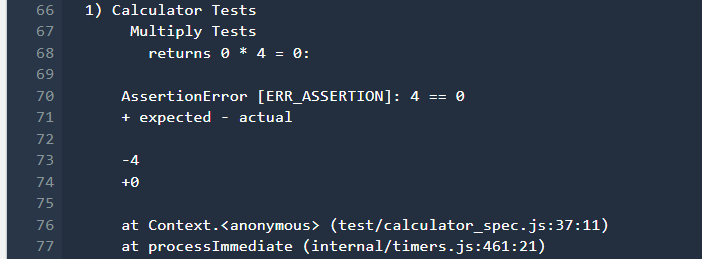


2.6 Scroll down to the very bottom, it shows which state failed



2.7 Scroll just up, and see the cause of the error

Here, it is an assertion error (4==0)



Another way to phrase what the Assertion Error in the log is saying is, The expected output of the test and the actual output from the test when it ran, are not equivalent to each other.

2.8 Note down the following:

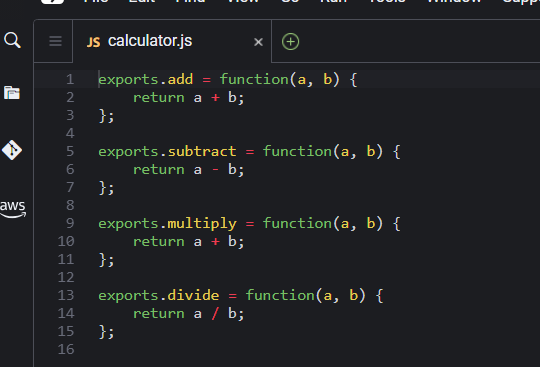
1. Who caused the error: Calculator multiplication tests returned an error during the build process.
2. What the type of error was: Assertion Error in Node.js.
3. When in the build process the error occurred: During the build phase (timestamps are available, if needed).
4. Where the error occurred in the test code: The file path and file name of the test definitions is test/calculator\_spec.js. The line number and character location is 37:11.
5. Why this error occurred: The root cause of the error was a mismatch between the value returned from the test and the expected value defined for the test.
6. How this error occurred: Likely a human error, such as a typo, occurred when creating the test code.

**Task 3 - Fixing error in the test code**

3.1 Open cloud9, and clone the repo in IDE

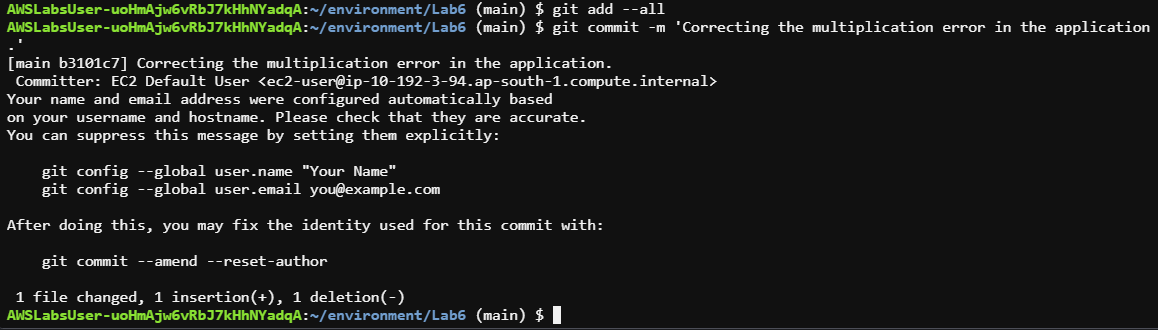


3.2 Open the calculator file code using the file explorer

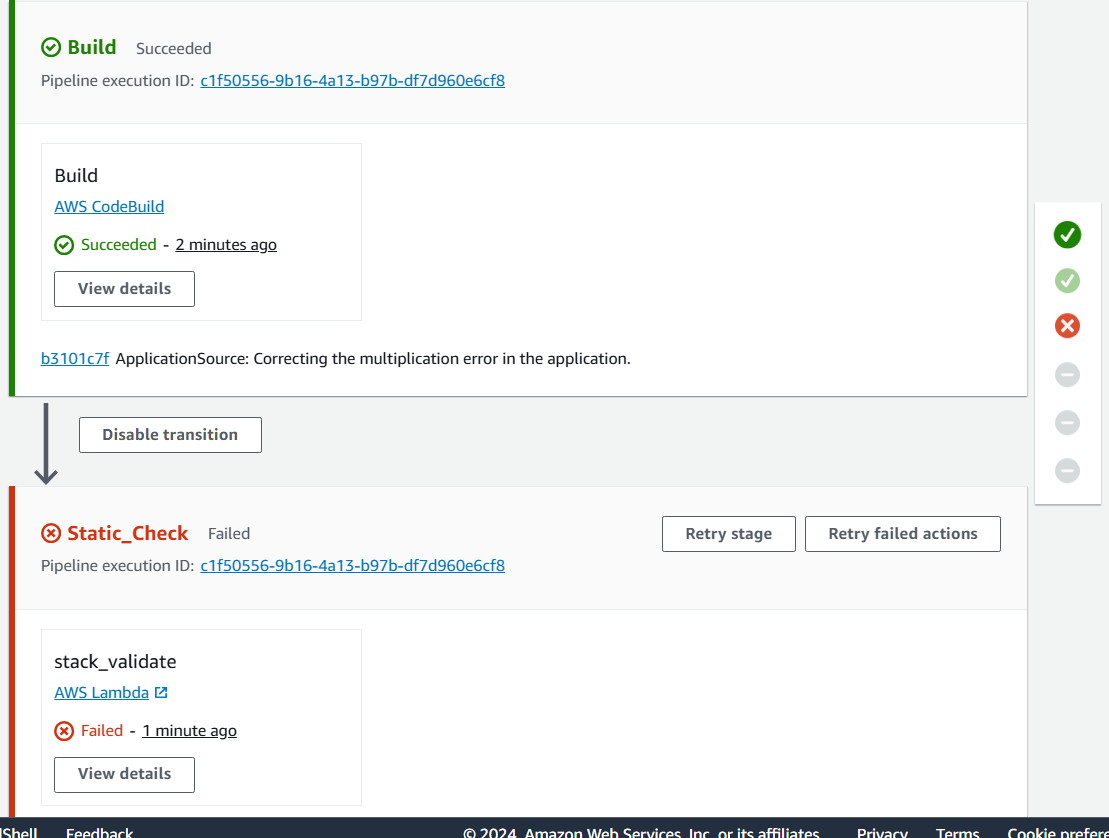


3.3 Change line 10 to a\*b

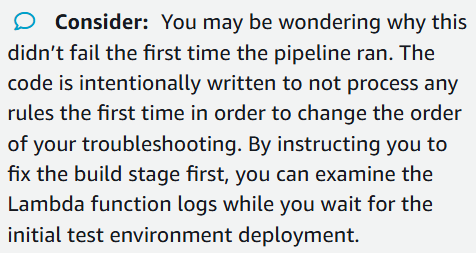
3.4 Commit the changes



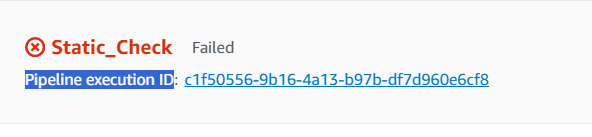
3.5 OPen codepipeline again, we see build stage has passed, but static check has failed



**Task 4: Fixing the Lambda Static check**

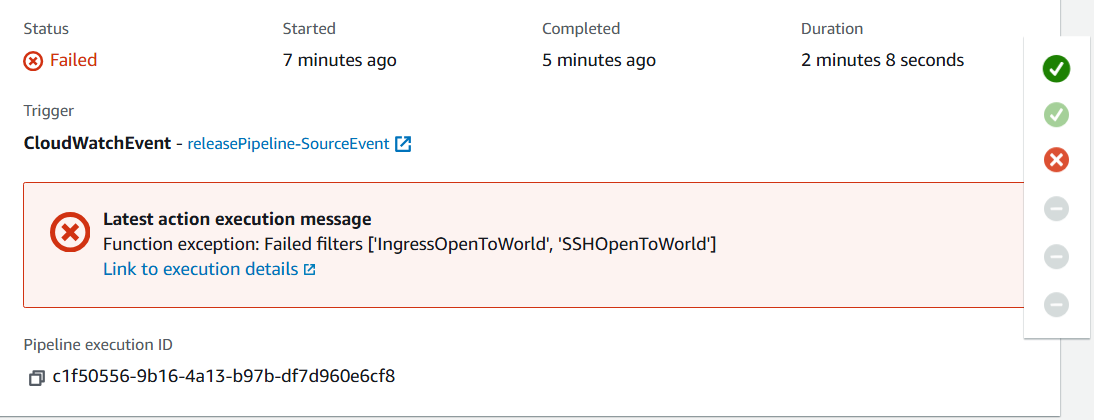
****

4.1 Click on the Pipeline execution ID



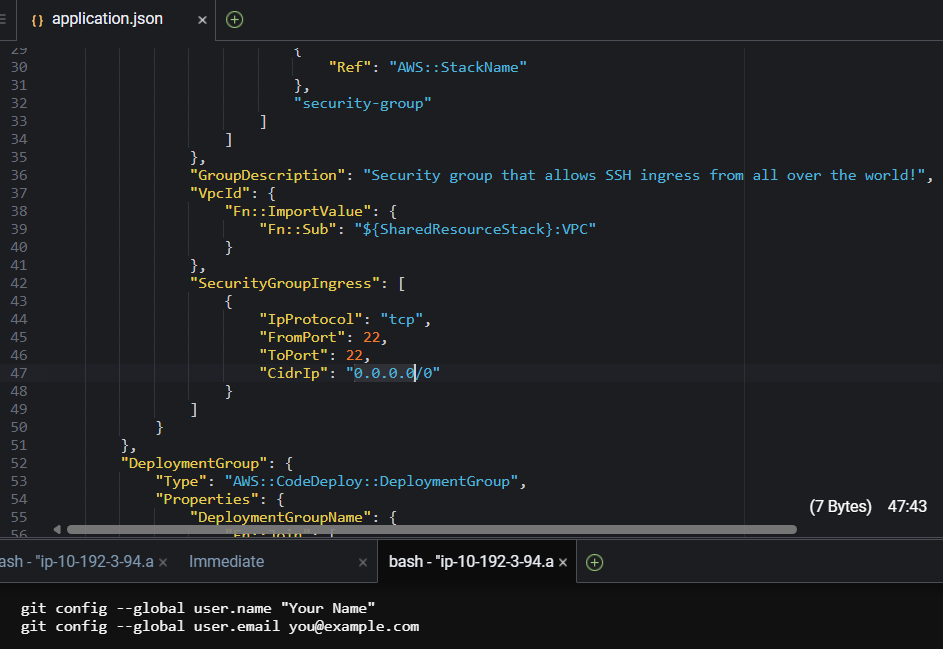
4.2 error:

The Action execution failed message dialog box displays the error returned from the Lambda CFNValidateLambda function. The reason for the failure is that the function found global access to the SSH network ingress ports.

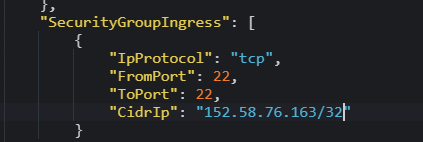


4.3 Return to Cloud9

4.4 Open application.json file from the left pane explorer



4.5 On Line 47, change “CidrIp”: “0.0.0.0/0” so that the value is your own private IP address instead, to allow access only from your IP address



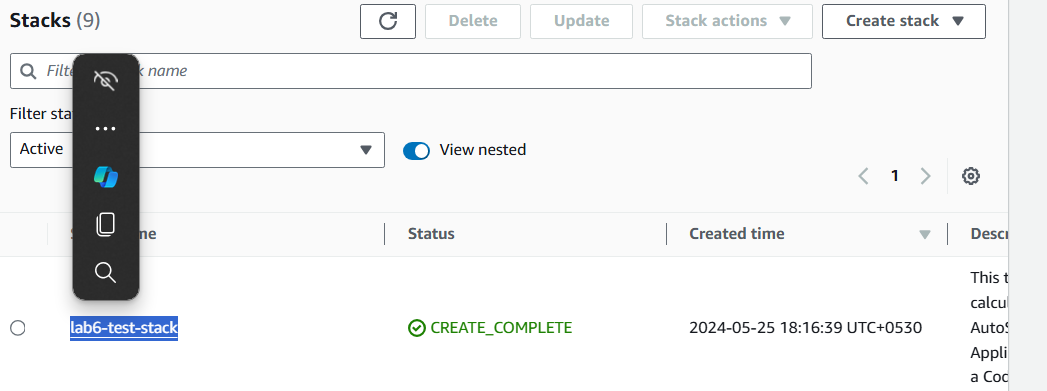
4.6 Save and push the changes

**Task 5: Validating in the Test Environment**

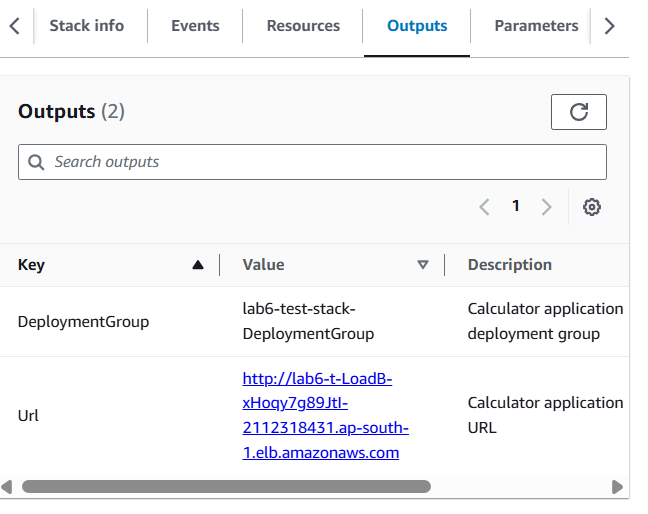
**Your pipeline uses these deployment action providers -**

1. **AWS CloudFormation** for deploying immutable infrastructure
2. **AWS CodeDeploy** for deploying applications.

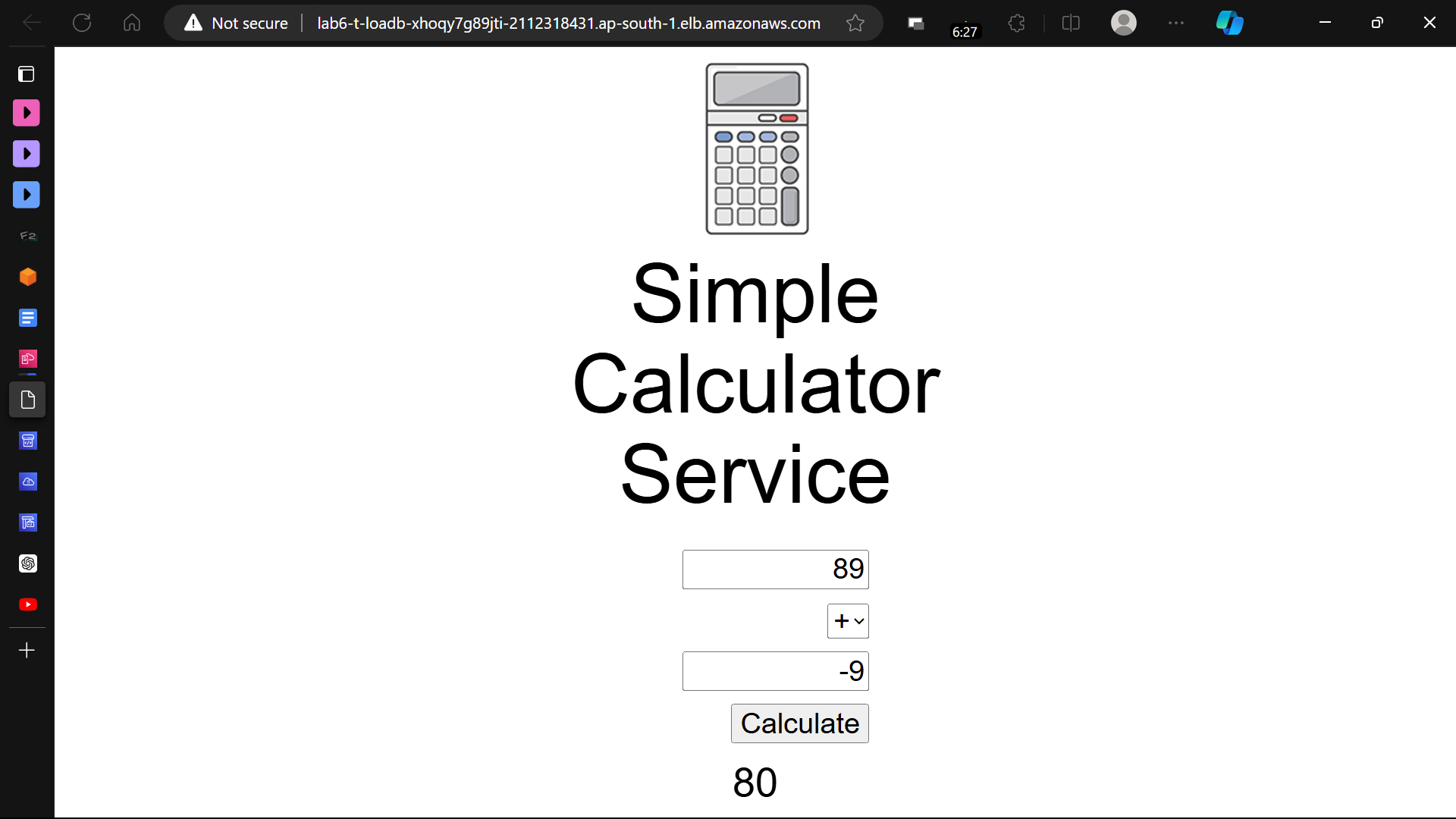
5.1 Open CloudFormation, and select test-stack



5.2 Open Outputs tab, and open the URL in new tab



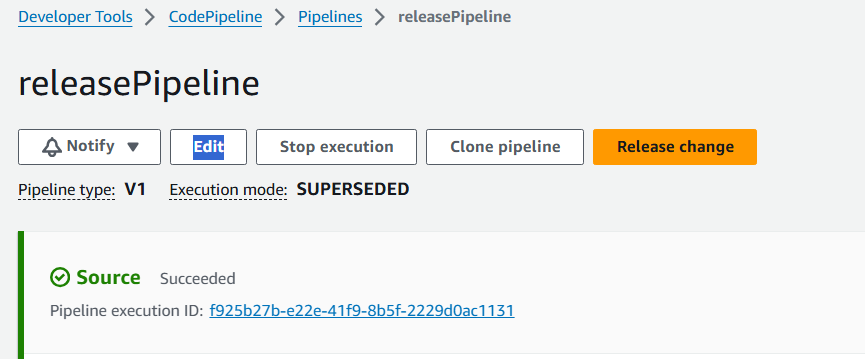
5.3 Our app opens, and it works!!!



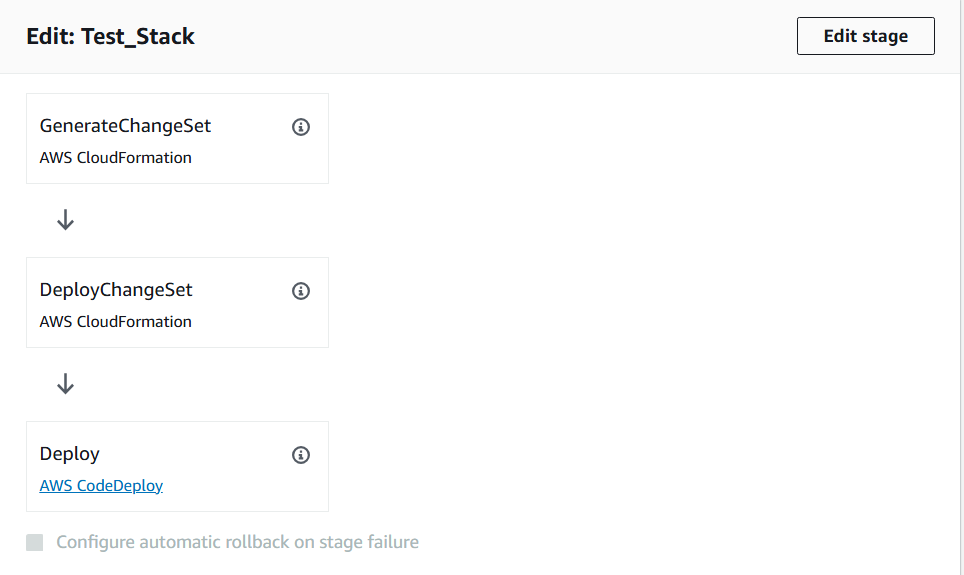
**Task 6: Automating the testing of the test environment**

In a previous task, wemanually validated and tested that the web application site was responsive. In this task, we use an AWS Lambda function in the release pipeline to automate the validation step for the test environment.

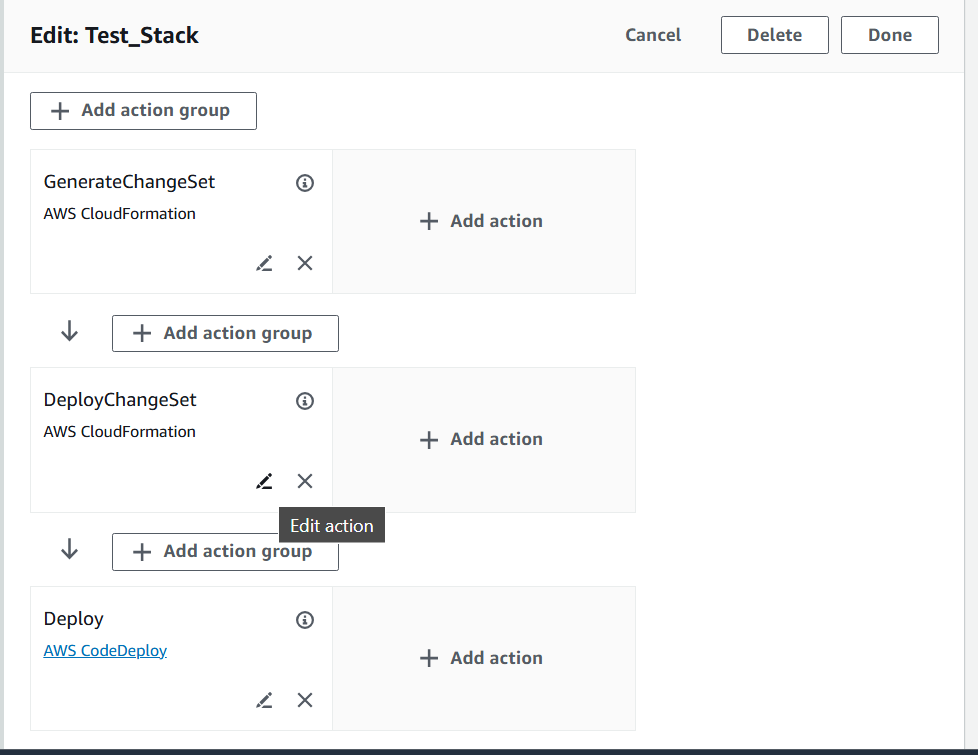
6.1 Go to CodePipeline and choose edit



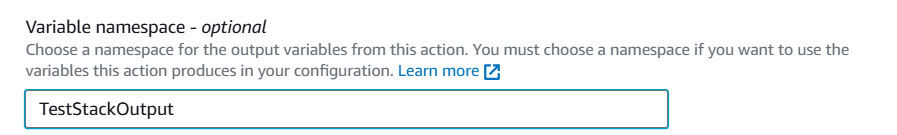
6.2 Choose edit stage in test\_stack



6.3 Click on edit in DeployChangeSet

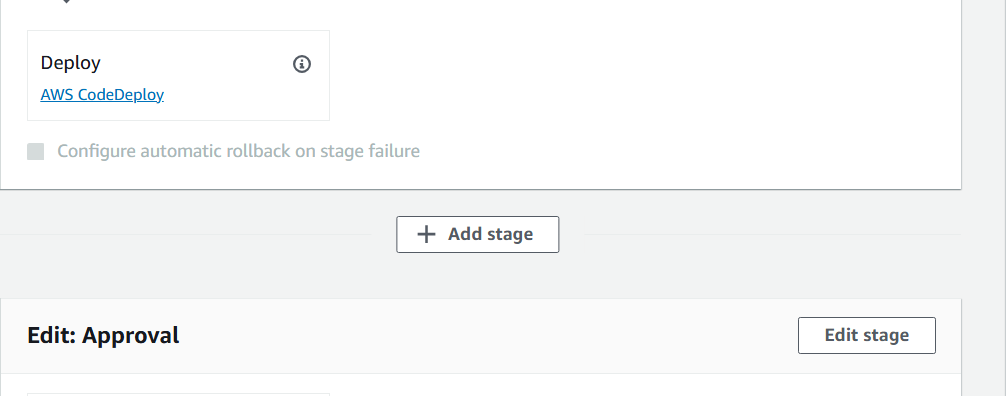


6.4 Fill this value

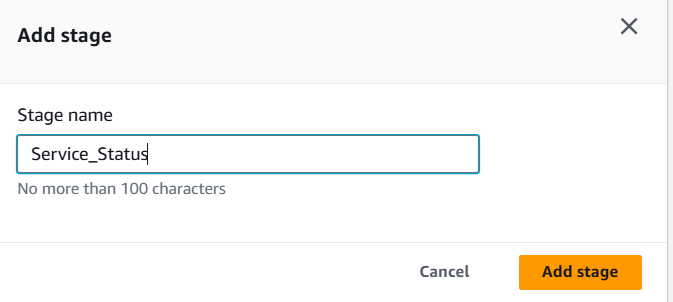


6.5 choose next

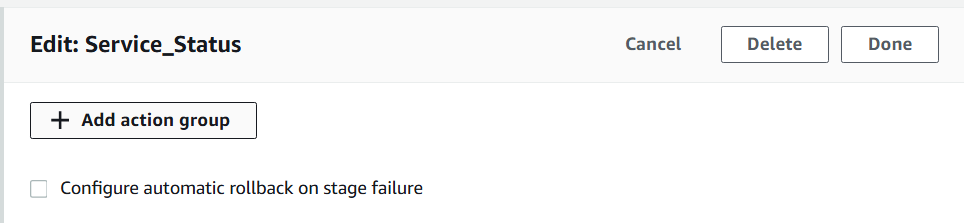
6.6 Click on add stage option after



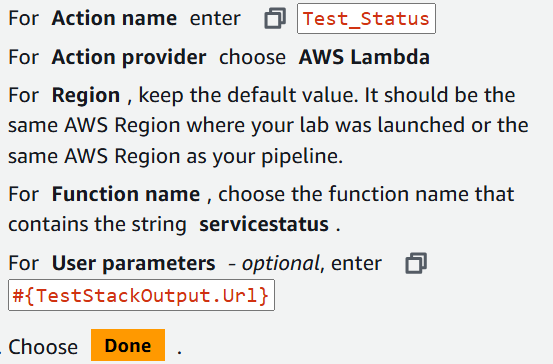
6.7



6.8 Choose add action grp

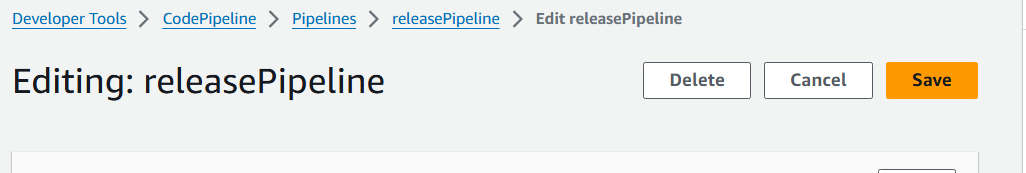


6.9 Following values



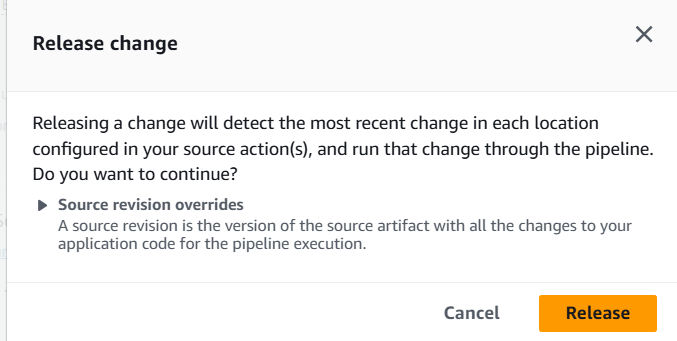
Click on done

6.10 Save the edits in pipeline

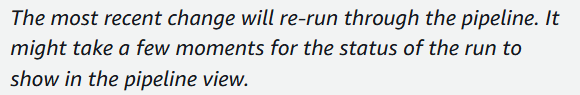


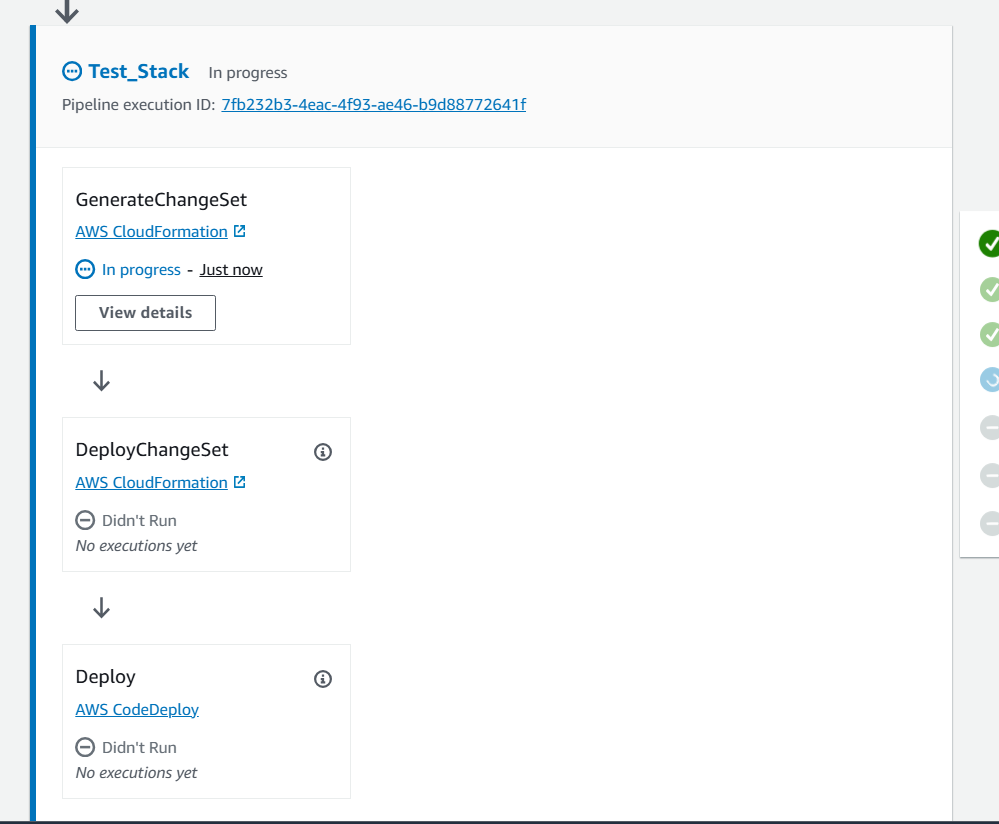
To test the new pipeline stage with the Lambda function, release the most recent change through the pipeline.

6.11 Click on release Pipeline



6.12

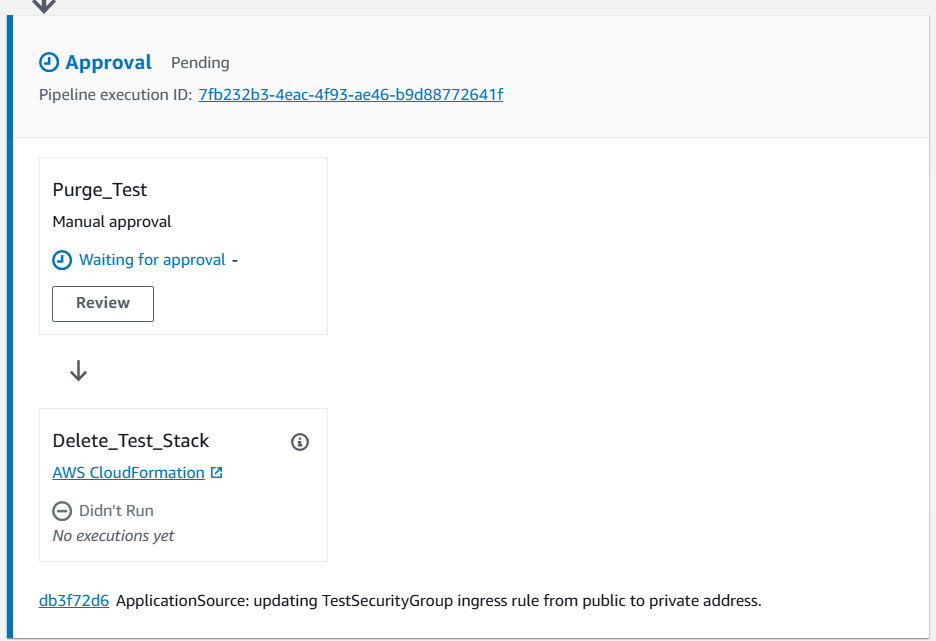




**NOTE -** We used the CodePipeline console to configure the DeployChangeSet action in the Test\_Stack stage to export it’s CloudFormation outputs as variables. You then added a new stage in the release pipeline and added an action in that stage and tested the new stage.

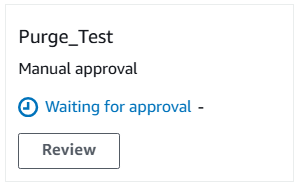
**Task 7: Approving the deletion of the Test environment and build the Production environment**

7.1 On the CodePipeline console, locate the Approval stage of the pipeline.

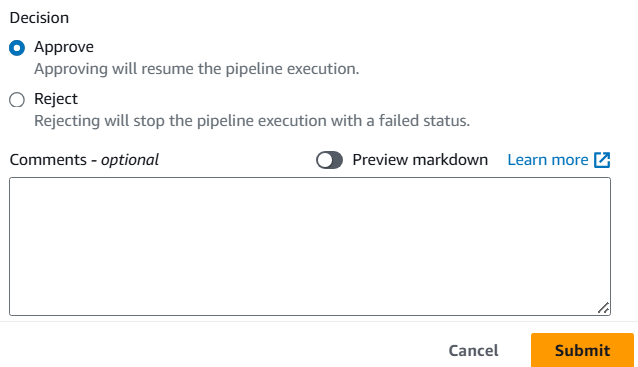


It is in pending state

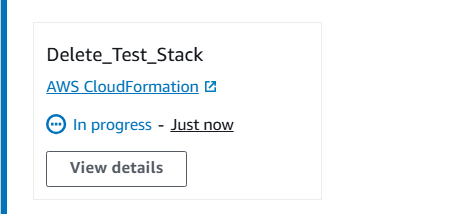
7.2 In the Purge\_Test action, choose Review .



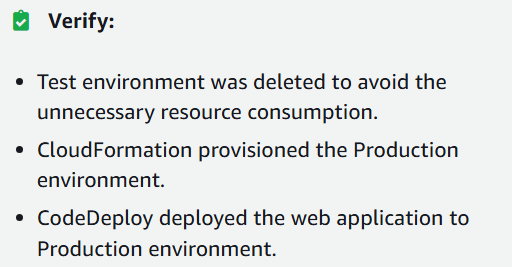
7.3 choose approve and submit



7.4 The pipeline performs actions with CloudFormation to delete the test environment stack.



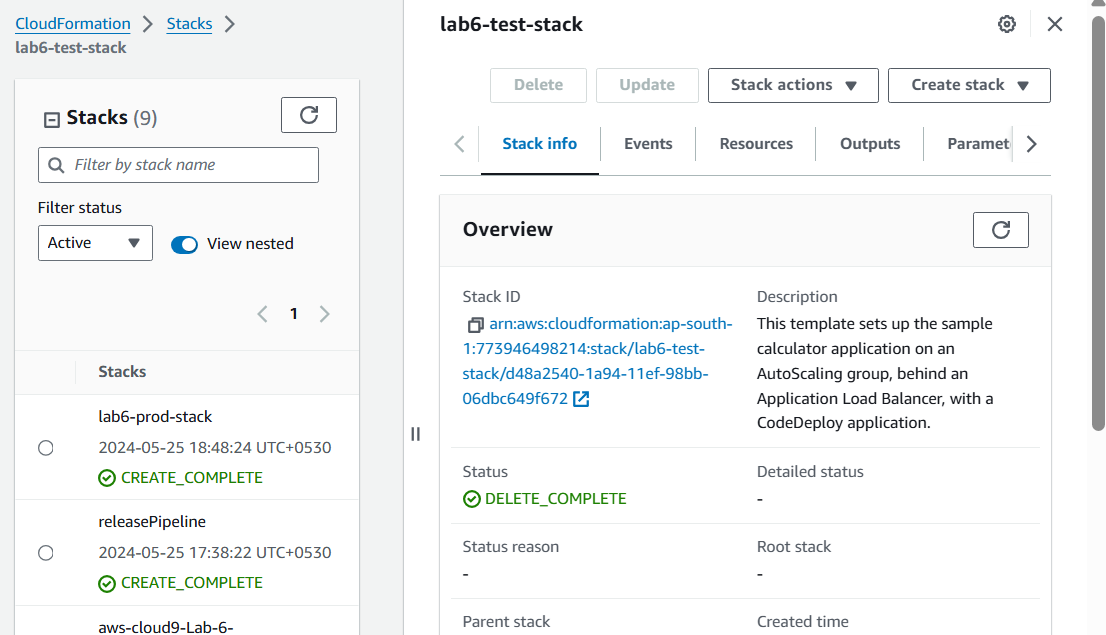
**Task 9: Verify the application deployment to Production environment**

We have to verify: 

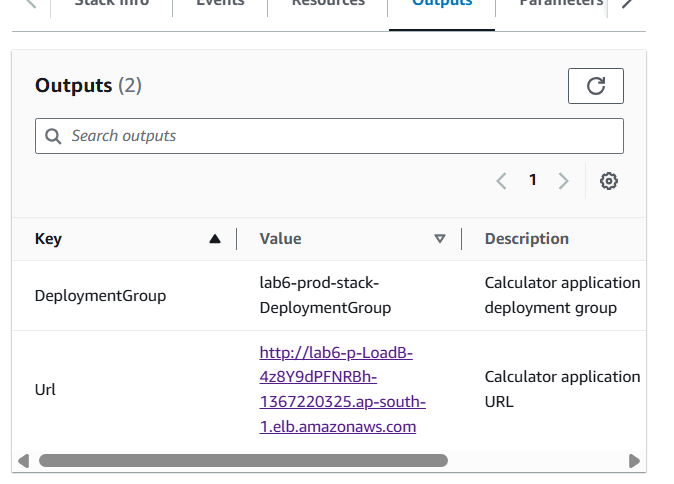
9.1 Open CloudFormation again, and we no longer see a stack that contains the string test-stack.

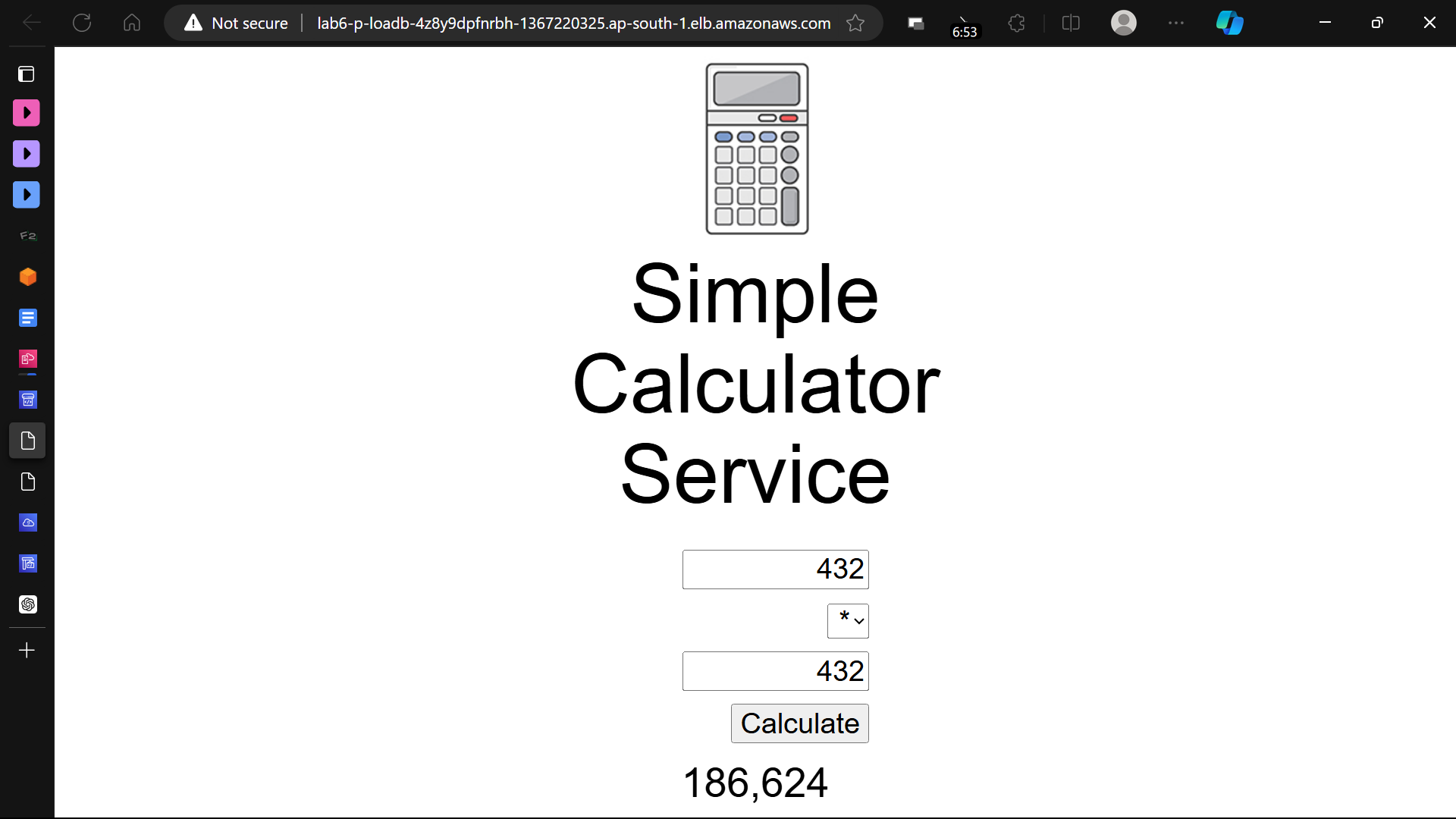
Test-stack shows as deleted;

Only prod-stack is available



9.2 Open the url from prod-stack, to see the calculator once again





Lab complete