Infrastructure Delivery Pipeline

By David Rise Knotten

Contents

[The infrastructure 3](#_Toc88060512)

[Custom nGinx server component 5](#_Toc88060513)

[CloudFormation code 6](#_Toc88060514)

[GitHub Actions / The pipeline 9](#_Toc88060515)

[General 9](#_Toc88060516)

[Test environment deployment: 10](#_Toc88060517)

[Staging environment deployment: 11](#_Toc88060518)

[Production environment deployment: 12](#_Toc88060519)

[Successful pipeline run 13](#_Toc88060520)

[Failed pipeline run 17](#_Toc88060521)

[Some things to note 20](#_Toc88060522)

[Shortcomings 21](#_Toc88060523)

[Conclusion 22](#_Toc88060524)

[“Bonus” 23](#_Toc88060525)

[Code 25](#_Toc88060526)

# The infrastructure

For the IsaC I will just use the CloudFormation template from the previous assignment.

It’s quite simple with four EC2 instances running an image containing some basic tests, a nodejs component and an nGinx component

Graphical user interface, application, Word

Description automatically generated

To test this pipeline for yourself you have to make your own image builder template with these components. The nGinx is a custom component which installs an nginx server and does a basic setup, creating an index.html file that just says hello world, creating the component is described on the next page. Run the image pipeline, after adding all the needed components, take the new images imageid and replace the imageids in the cloudformaiton template. The readme file in the project has a bit more in-depth instructions to do this. This is what the imagebuilder recipe should look like using the aws CLI

Text

Description automatically generated

To find the image run “aws imagebuilder list-image-recipes”, find the correct recipe, copy its ARN and run “aws imagebuilder get-image-recipe –image-recipe-arn <ARN>”

## Custom nGinx server component

To get nginx into the VM for testing I had to create a custom image component:

{

"component": {

"arn": "arn:aws:imagebuilder:eu-west-1:001397529725:component/nginx/1.0.1/1",

"name": "nginx",

"version": "1.0.1",

"type": "BUILD",

"platform": "Linux",

"owner": "001397529725",

"data": "name: nGinxInstall\ndescription: Installs nGinx\nschemaVersion: 1.0\n\nphases:\n - name: build\n steps:\n - name: InstallnGinx\n action: ExecuteBash\n inputs:\n commands:\n - sudo apt-get install nginx -y\n - name: ReplaceIndex\n action: ExecuteBash\n inputs: \n commands:\n - sudo chmod a+w /var/www/html -R\n - sudo rm /var/www/html/index.nginx-debian.html\n - echo \"Hello world\" > /var/www/html/index.html\n\n",

"encrypted": true,

"dateCreated": "2021-11-12T21:44:47.353Z",

"tags": {}

}

}

I created and built the image in the aws console, instead of defining it with code mostly because it’s quite a hassle defining it that way, but building an image takes a lot of time, which will make the cloudformation deploy action time out on github. I don’t think that is an issue, though as the image recipe should remain fairly static.

## CloudFormation code

{  
 "AWSTemplateFormatVersion": "2010-09-09",  
 "Metadata": {  
 "AWS::CloudFormation::Designer": {  
 "ae95d48d-a3ac-47a3-aea1-506d4e3e8365": {  
 "size": {  
 "width": 60,  
 "height": 60  
 },  
 "position": {  
 "x": 480,  
 "y": 370  
 },  
 "z": 0,  
 "embeds": []  
 },  
 "c6ee5b9a-f394-4f46-8f2c-626a44b86bce": {  
 "size": {  
 "width": 60,  
 "height": 60  
 },  
 "position": {  
 "x": 570,  
 "y": 370  
 },  
 "z": 0,  
 "embeds": []  
 },  
 "df433d24-5ad6-4a94-b88a-f79e6f9696fa": {  
 "size": {  
 "width": 60,  
 "height": 60  
 },  
 "position": {  
 "x": 660,  
 "y": 370  
 },  
 "z": 0,  
 "embeds": []  
 },  
 "ea1d2848-7bc8-408c-823a-c298d732224f": {  
 "size": {  
 "width": 60,  
 "height": 60  
 },  
 "position": {  
 "x": 400,  
 "y": 370  
 },  
 "z": 0  
 }  
 }  
 },  
 "Resources": {  
 "EC2I3C31M": {  
 "Type": "AWS::EC2::Instance",  
 "Properties": {  
 "ImageId": "ami-0f9407f68dc039539",  
 "InstanceType": "t2.micro",  
 "KeyName": "aws-eb",  
 "SecurityGroupIds": [  
 {  
 "Ref": "SecGroup"  
 }]  
 },  
 "Metadata": {  
 "AWS::CloudFormation::Designer": {  
 "id": "ae95d48d-a3ac-47a3-aea1-506d4e3e8365"  
 }  
 }  
 },  
 "EC2IW00S": {  
 "Type": "AWS::EC2::Instance",  
 "Properties": {  
 "ImageId": "ami-0f9407f68dc039539",  
 "InstanceType": "t2.micro",  
 "KeyName": "aws-eb",  
 "SecurityGroupIds": [  
 {  
 "Ref": "SecGroup"  
 }]  
 },  
 "Metadata": {  
 "AWS::CloudFormation::Designer": {  
 "id": "c6ee5b9a-f394-4f46-8f2c-626a44b86bce"  
 }  
 }  
 },  
 "EC2I3UWUI": {  
 "Type": "AWS::EC2::Instance",  
 "Properties": {  
 "ImageId": "ami-0f9407f68dc039539",  
 "InstanceType": "t2.micro",  
 "KeyName": "aws-eb",  
 "SecurityGroupIds": [  
 {  
 "Ref": "SecGroup"  
 }]  
 },  
 "Metadata": {  
 "AWS::CloudFormation::Designer": {  
 "id": "df433d24-5ad6-4a94-b88a-f79e6f9696fa"  
 }  
 }  
 },  
  
 "EC2I57G": {  
 "Type": "AWS::EC2::Instance",  
 "Properties": {  
 "ImageId": "ami-0f9407f68dc039539",  
 "InstanceType": "t2.micro",  
 "KeyName": "aws-eb",  
 "SecurityGroupIds": [  
 {  
 "Ref": "SecGroup"  
 }]  
 },  
 "Metadata": {  
 "AWS::CloudFormation::Designer": {  
 "id": "ea1d2848-7bc8-408c-823a-c298d732224f"  
 }  
 }  
 },  
 "ElasticIP": {  
 "Type": "AWS::EC2::EIPAssociation",  
 "Properties": {  
 "AllocationId": "eipalloc-0a8775d56eb20d32c",  
 "InstanceId": {  
 "Ref": "EC2I57G"  
 }  
 }  
 },  
 "SecGroup": {  
 "Type": "AWS::EC2::SecurityGroup",  
 "Properties": {  
 "GroupDescription": "Webservers with SSH connectivity",  
 "SecurityGroupIngress": [{  
 "CidrIp": "0.0.0.0/0",  
 "IpProtocol": "tcp",  
 "FromPort": 80,  
 "ToPort": 80  
 }, {  
 "CidrIp": "0.0.0.0/0",  
 "IpProtocol": "tcp",  
 "FromPort": 22,  
 "ToPort": 22  
 }]  
 }  
 }  
 }  
}

To set up SSH look at the project readme

# GitHub Actions / The pipeline

## General

These actions deploy to their respective environments.

The test deployment workflow runs unit tests with jest for the node program, as well as testing if the deployed infrastructure and software is working as intended, by checking if the nGinx server installed in the nginx component responds. If, and only if the test deployment is successful the workflow continues onto the staging deployment phase

The staging deployment phase just deploys to the staging environment, without running any real tests, but it’s here I would implement any tests involving other services/infrastructure. With the timeframe of this project, creating loads of deployment environments would take too much time. One thing this staging environment would be good for is checking that the website runs as intended (In this case it’s a quite simple website)

The last stage, production deployment, has to be manually run, and this updates the production environment to the latest code

## Test environment deployment:

Text

Description automatically generated

## Staging environment deployment:

Text

Description automatically generated

## Production environment deployment:

Text

Description automatically generated

## Successful pipeline run

First I’ll show a successful run of the pipeline

Graphical user interface, application, Teams

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Test deployment:

Graphical user interface, text

Description automatically generated

Here we see that the workflow completed without any errors, which means that the template can be built and run, as well as the test components of our machine image succeeded, if not the deployment would have failed. The nGinx endpoint is also tested, and a 200 OK is returned, so we know the nGinx server is up and running as expected

This step is triggered by a push on the master branch

Staging deployment:

Graphical user interface, text

Description automatically generated

Again we know at this point that the template is valid and works as intended, so this should also succeed, straight forward. Remember, this happens automatically, depending on the status of the test deployment. It is at this point you would do manual and integration-testing.

Prod deployment

Graphical user interface, text

Description automatically generated

On to production deployment. This stage only happens if you manually run the deployment to production workflow, which should be after both a successful test deployment as well as some testing in the staging environment, depending on your situation.

This has been an example of how how a successful pipeline run would work

## Failed pipeline run

Graphical user interface, text, application

Description automatically generated

To fail a run, in this example, I will just re-deploy the exact same code I did the last time. Thanks to CloudFormations own testing, a stack update is changed if the new template lacks any changes. If the nGinx server failed to return a 200OK response the outcome would be the same

Test deployment:

Text

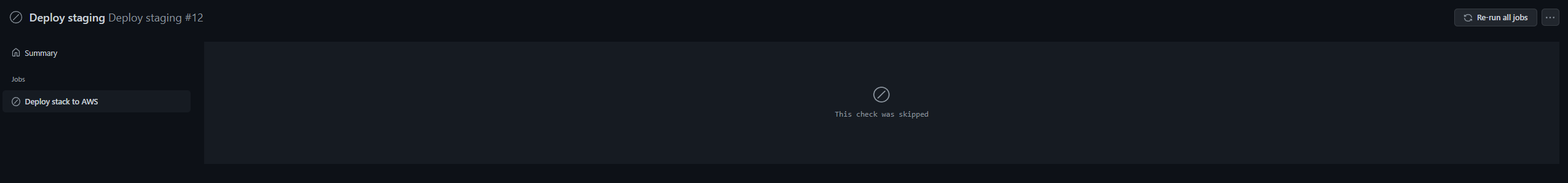
Description automatically generated

As you can see in the error from the deployment:

“**Error:** Failed to create Change Set: The submitted information didn't contain changes. Submit different information to create a change set.”

The deployment failed due to CloudFormation failing to create a change set, as we just pushed the same code as last time.

Staging deployment:



Since our test deployment failed, we skipped the staging deployment, so our last working staging deployment works as before.

This example of a failed deployment works for any other failure, such as if the test components of the machine image fails, CloudFormation will detect it and propagate the error and fail itself

# Some things to note

The CloudFormation deploy github action makes deployment straight forward, but it is somewhat lacking in customizability. One thing that this pipeline struggles with is a failing deployment when there is no previous state to rollback to. In CloudFormation, a stack which is stuck in the rollback complete state cannot be updated/pushed to. So you might need to go into AWS, and manually delete the stacks every now and again. An alternative would be to use the “--on-failure delete” option which is found in the AWS cli, but there seems to be no support for that using GitHub actions.

AWS authentication

You need a couple parameters for the AWS credentials task used in the pipeline. You add these by going to the repository -> settings -> secrets and add the following:

Graphical user interface, application

Description automatically generated

The credentials have to be tied to a user on AWS with the following policies:

Graphical user interface, text, application

Description automatically generated

(You could, and in the “real world” should, go with more limited policies, but to keep things simple…)

The AWS credential type you need is “Access key – Programmatic access”

# Shortcomings

The most major shortcoming is the lack of testing. As mentioned earlier, I did setup an nGinx component to be able to test endpoints. But it doesn’t seem like cloudformationdeploy supports that.. And neither did I find a way to do it using TaskCat (Which took way too long to set up), which is the semi-official testing framework for cloudformation. I did however find a way around it, using elastic IPs and a http-response-check task. It’s quite simple, just checking if the elastic IP responds with 200, but at least it lets us know the nginx server is running and you can access it

On the plus side, using this method you can check pretty much any API with the premise it has an endpoint which returns 200 (Other status codes can also be tested for). But testing for a specific response using GitHub actions is somewhat harder

One shortcoming I didn’t manage to solve was to find a way to “promote” an environment, which I know other tools like terraform can. But to do this I would need to store the machine state and launch the new environment using it, which is way outside my skill level on AWS services. I’d imagine that AWS codepipeline has better support for cloudformation, but re-doing everything is something I just don’t have time to for a detail (although in a professional project it would be a must)

# Conclusion

CloudFormation alone works really well, and I definitely see the use for it, but I don’t think I would use it myself, due to a lack of outward integration. Instead I’d use Terraform which really is all about integration, including the integration of higher level functionality (upgrading a test deployment stack to a staging deployment stack). I’m not saying CloudFormation is bad, but it does seem like it’s a bit too much of a hassle to work with, and that’s probably why services such as Terraform are so big, as they add a little layer on top of CloudFormation, making setups easier (Especially features such as promoting a stack, instead of having to deploy an all new stack)

While most, if not all, things can be done with code, there are some things that are way easier to do using a console. The biggest example in this project is the console use to create an image builder pipeline and a component.

Am I happy with the outcome? Not really… While I did manage to do a lot, but also having to work a lot to do it, there are some details that just fell through and things that just wouldn’t work. Such as the stack upgrading and deleting the test stack, deleting the test stack requires you to manually go into the console and delete the stack, there’s no way to delete it after building (Note: I did actually find a way to do this, but it required a lot of work and seemed extremely cumbersome, also requiring me to learn multiple different AWS services).

# “Bonus”

A bit last minute I also added a node backend to the project, which shows how you could integrate unit testing to the pipeline. The server is quite simple with only a “hello world” endpoint at “/”. The test is only run in the deploy\_test step

To test that the node backend works as expected I’ve used jest. This simply tests that the response code is 200, and that the server returns the correct JSON response. But in a bigger project, the tests would of course be quite a lot larger. The server is located in the src folder, and the test is in the test folder in the project

Server.js

A screenshot of a computer

Description automatically generated with medium confidence

Server.test.js

A screenshot of a computer

Description automatically generated with medium confidence

Here’s the code added to the workflow

Text

Description automatically generated

And here’s what the workflow looks like with the additional test

Text

Description automatically generated

All the code is available in the repository on the last page

Code

<https://github.com/davrikn/cloudFormationDeploy>

The README contains information about what you need to run this yourself. I think I have everything covered in there, but with how delicate cloud services are, any issue wouldn’t be a surprise. So just ask if there’s an issue, or alternatively create an issue