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

It’s quite simple with four EC2 instances running a nodejs image containing some basic tests

Graphical user interface, application, Word

Description automatically generated

(Nodejs was later replaced with an nginx server)

To test this pipeline for yourself you have to make your own image builder template with these components, build and update the ImageId to your own AMIs. Its code should look like this:

Text

Description automatically generated

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, as it can take a solid bit of time, and make the GitHub actions time out. It should be mostly static too, so rebuilding the image every deployment would be a waste of time (and money).

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  
 }]  
 }  
 }  
 }  
}

GitHub Actions

Test environment deployment:

Text

Description automatically generated

Staging environment deployment:

Text

Description automatically generated

Production environment deployment:

Text

Description automatically generated

These actions deploy to their respective environments. Test deploy just deploys the code in an environment and runs the different tests. If, and only if, the test deploy deployment succeeds the staging environment is deployed. Here’s where I would do integration tests, having a “network” of all the services staging environments. However, as I only have the one this environment doesn’t do all that much except that it runs the latest code pushed that doesn’t immediately break. An example of its use could be if the service is supposed to run a simple website, you could go into the staging environment and see if everything looks right.

Lastly is the production deployment. This only happens when someone manually runs the workflow

Pipeline Runs

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 succeeded, if not it 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, 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

Next up, a failed 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

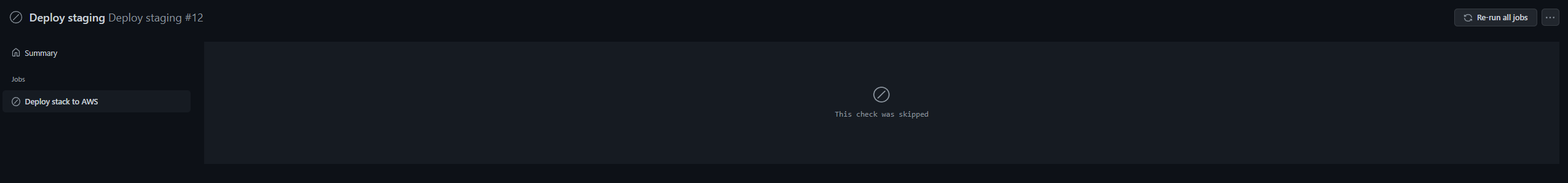
Description automatically generated

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 failing a 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 do this 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”

Adding tests in software is also straight forward, if a process exits with an error CloudFormation automatically interprets that as a failed deployment, so to test this with code would just require you to run some code and exit with an error. (Although getting the non-cloudformation code into the pipeline can be a bit difficult)

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 a 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

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

Cloudformaiton 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. An option could be running with Terraform which really is all about integration, including the integration of higher level functionality (upgrading a test deployment stack to a staging deployment 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).

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