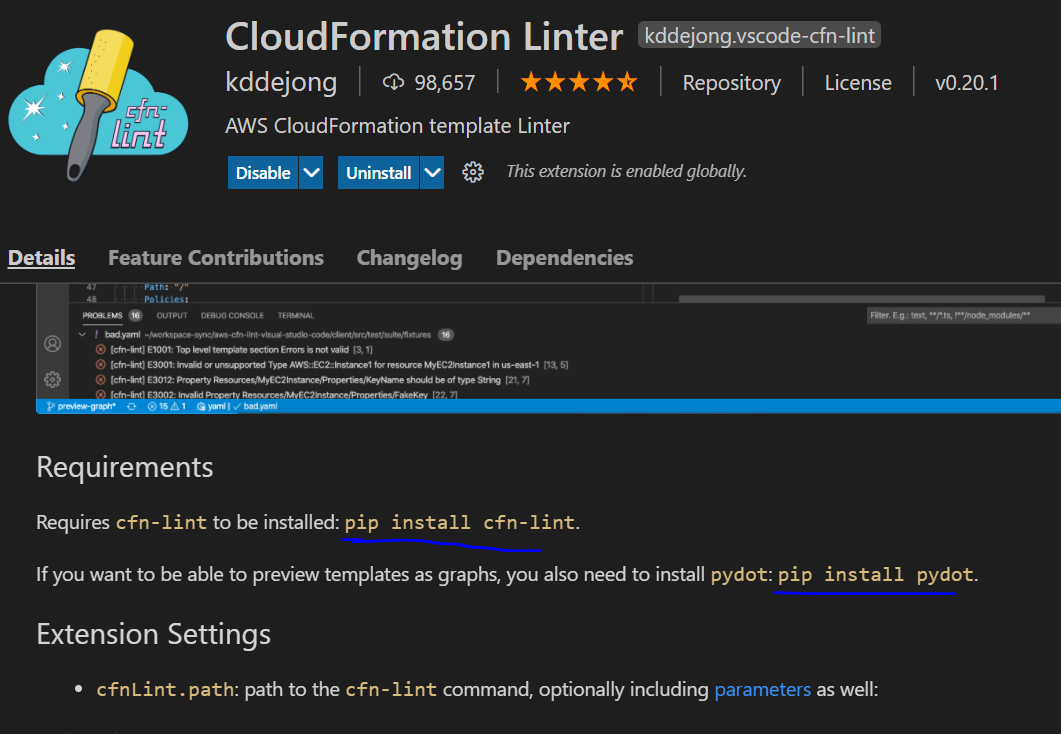
**Prerequisites:**

Install CloudFormation linter extension

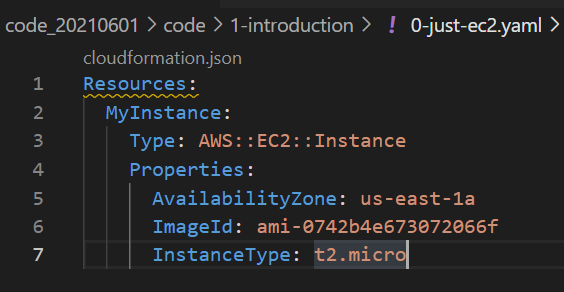


This extension needs these two requirements so install them too

**Tasks**

**Task- Creating an EC2 by just uploading the template**

Open CF and upload the template



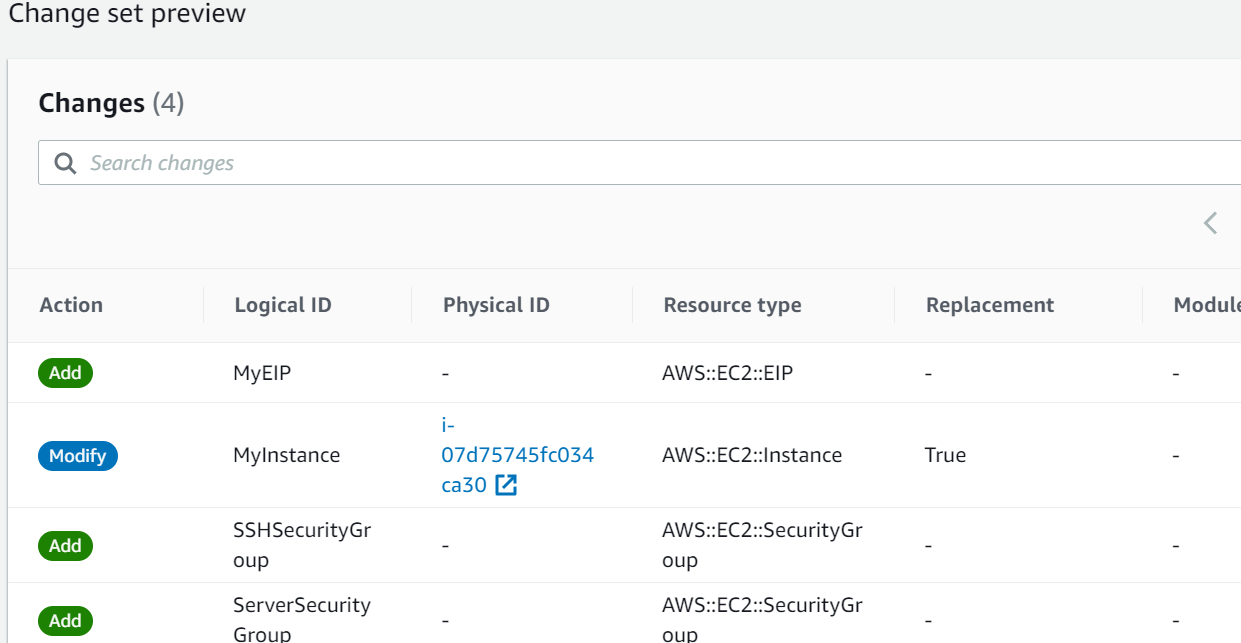
It will create the EC2.

**Task- Updating a stack**

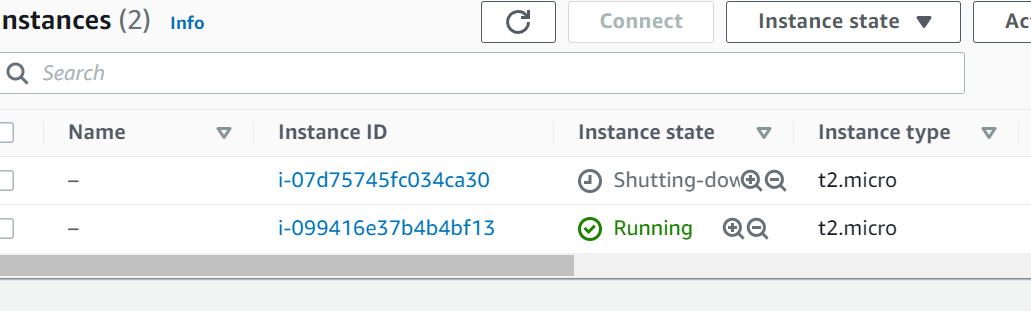
Go to your stack and click update

Select replace current template and upload your new template

We can see which resoutce going to be replaced and which one added/removed under change set preview



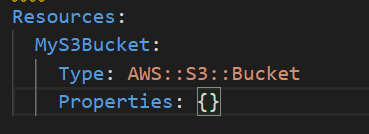
We can see during update AWS automatically cleanup the old resources which are going to be replaced by new one. Make sure you have taken backup for those if necessary, before updating the stack.



Note: To delete all those resources we just have to delete the stack and AWS will cleanup everything. Simple.

**Task- Creating a S3 bucket**

Creating S3 bucket without any custom specification is super simply

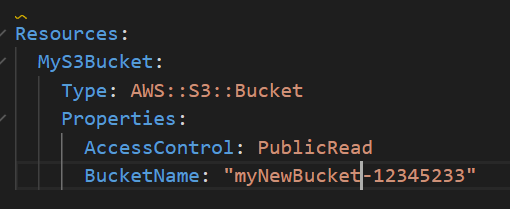


Changes to template can be three types

1. Update with no interruptions
   1. Without disrupting the resource operations that is without changing physical id.
   2. Ex: Updating IAM instance type of an EC2
2. Update with some interruptions
   1. Updating an instance type from micro to medium
3. Replacements
   1. Recreating the resource with new physical ids

The replacement one will create a new resource and then delete the old one.

Ex: - Simple renaming a S3 bucket will create a new S3 bucket with new name and then will delete the old one.





Note: Before Deleting S3 bucket related Stack make sure the bucket is empty otherwise stack deletion will fail.

CloudFormation template options

1. Tags
2. Permissions
3. Notification options = Add sns to cloudformations
4. Timeout = set to wait for time period before declaring stack creation failed
5. Rollback on failure = preserve the old resources if new resource creation failed
6. Rollback configurations = Monitoring the events while rollbacking
7. Stack Policy
8. Termination Protection = Preventing accidental templet deletion
9. Quick start link

**CloudFormation Template Designer**

It’s a cool tool to visualize the stack. We can use it in two ways

1. Creating a template visually – Drag and Drop
2. Copy paster the template code on designer to see it visually

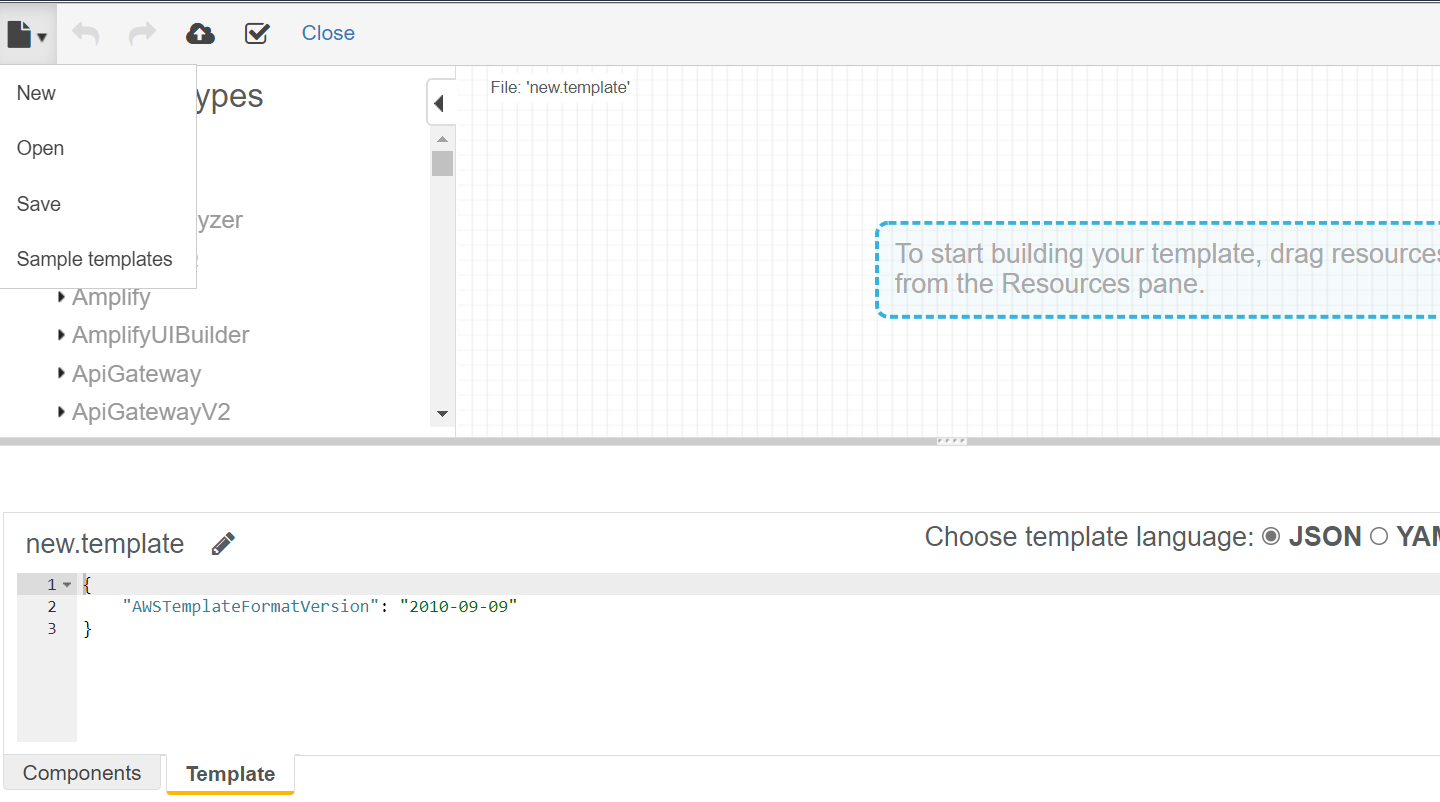
Option-1

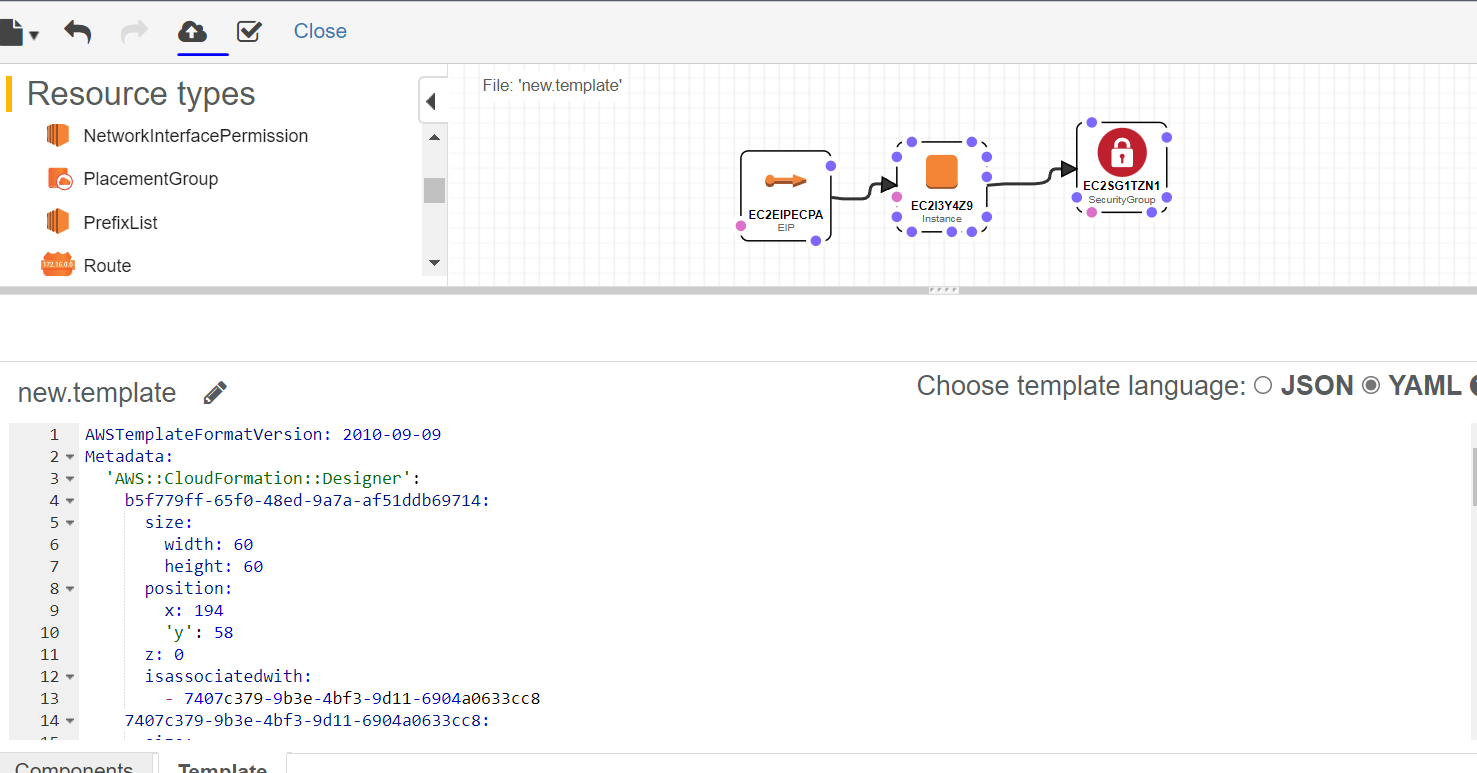
Lets create an EC2 instance which attached to to a security group and assign an elastic ip

We have to choose the option create template designer while creating the stack on console

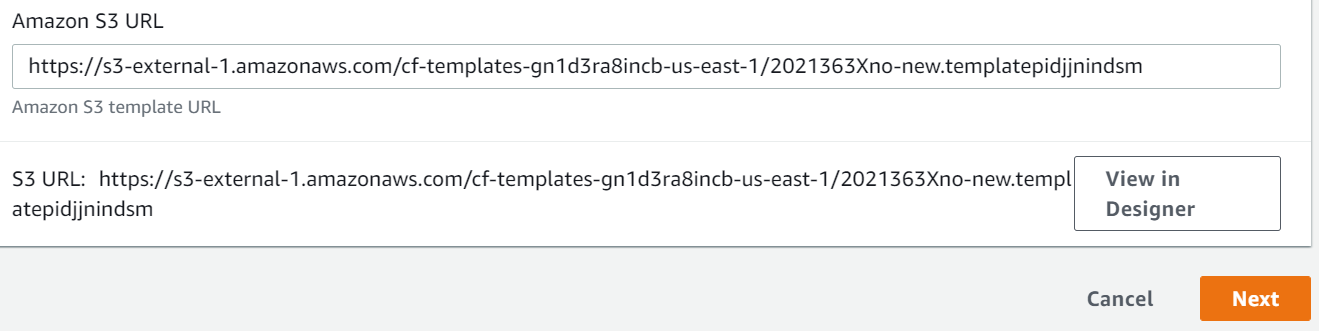
Everything is empty. Drag the resources from left Resource column.

Connecting one resource to another is tricky. You have to find the correct dot to connect.





To create the stack, click on the cloud option on top left corner. It will convert it into yml file and put it into S3 bucket.



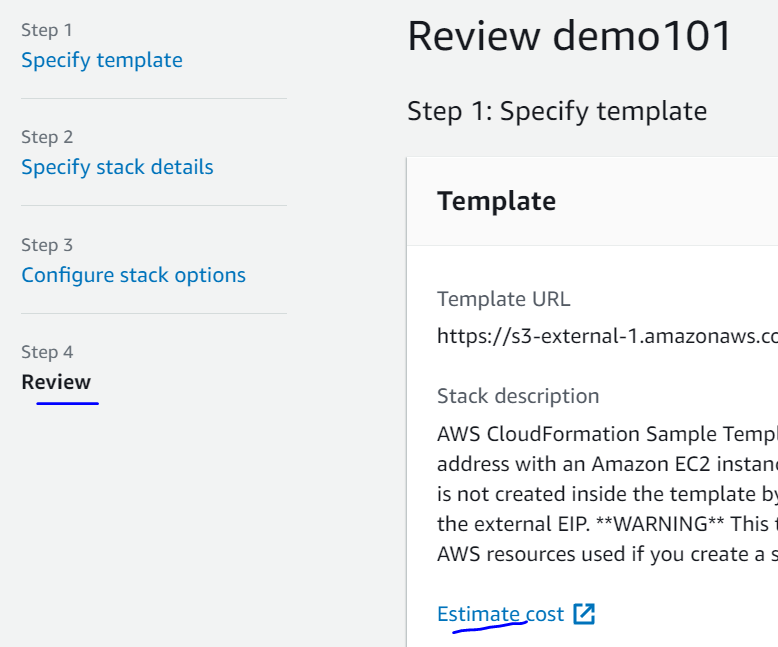
Other option is straight forward just copying your code to designer hit refresh and it will show you the stack.

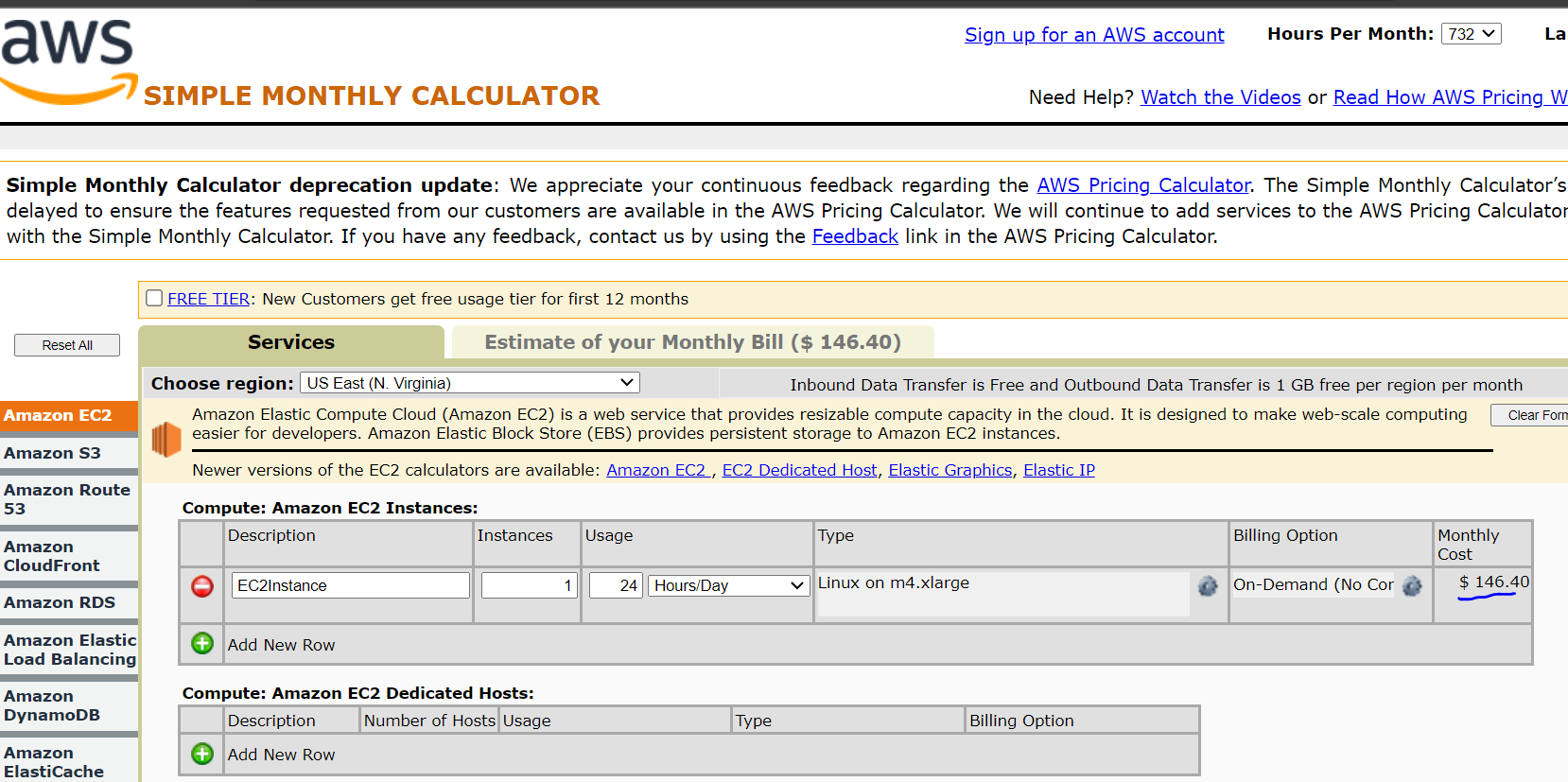
**CloudFormation Building Blocks**

1. AWSTemplateFormatVersion : Identifies the capability of the template
2. Description: Comments about the template
3. Transform: Specifies one or more Macros that used to run the template
4. Metadata
5. Resources: Your resource declaration
6. Parameters: The dynamic input for the template
7. Mappings: The static variable for the template
8. Outputs: Reference to what has been created
9. Conditionals: List of conditions to perform resource creation
10. Rules: Validate a parameter during stack creation/update

**Cost Estimation of Template**

While creating the stack at final step there is an option to review the cose if we click on that it will open a page which shows the monthly cost of the stack we are going to create.





**Parameters:**

Parameters are the way to provide the inputs to template.

When should we use the parameters?

* If a particular resource configuration is likely going to change in future? If so, then make it a parameter.
* We don’t have to re upload the template just to change its content.

**Parameter Settings:**

1. Types:
   1. String
   2. Number
   3. CommaDelimetedList
   4. List<Number>
   5. AWS Specific parameters (to catch invalid values-Match against existing values in the AWS account)
   6. SSM Parameters (Get parameters from SSM store)
2. Description
3. Constraint Description (String)
4. Min/Max Length
5. Default
6. AllowedValues
7. AllowedPattren (Regex)
8. NoEcho (Boolean)

**How to reference a parameter?**

1. We can reference a parameter anywhere inside the template using the function **Fn::Ref**
2. Parameters can be used anywhere except
   1. AWSTemplateFormatVersion
   2. Description
   3. Transform
   4. Mappings
3. The shorthand for this YAML is !**Ref**
4. This function can also reference other elements in template

**DBSubnet1:**

**Type: AWS::EC2::Subnet**

**Properties:**

**VpcId: !Ref MyVPC**

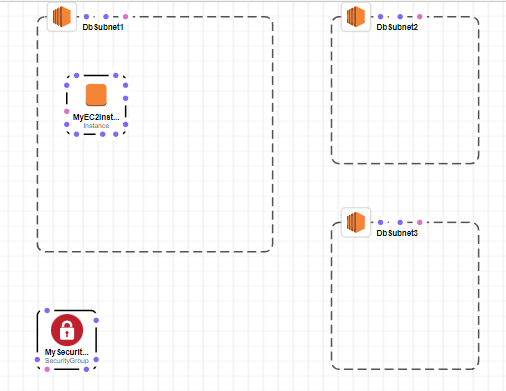
MyVPC is the parameter we referenced here in DBSubnet1 resource declaration.

**Task- Hands on Parameters**

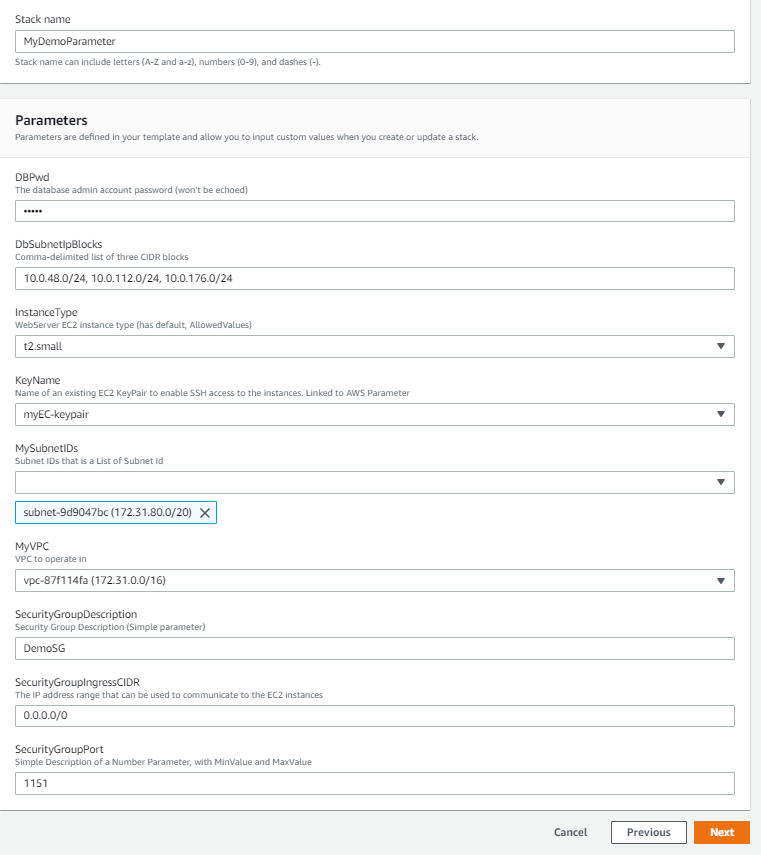
Will create a template with using all above parameter options

If we upload the template the designer view will looks like something below



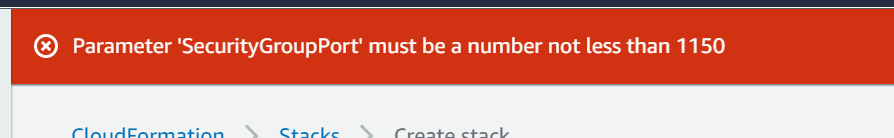


This template not making any sense so we will not create the stack but will just see the parameter section. The next page will ask you enter the parameters based on its declarations inside template



If we give any arbitrary value and that does not match the constraint, then it would throw error if constraint declared for that specific parameter. Example for SecurityGroupPort Min and Max value is declared inside template. Min = 1150 and if I pass 1149 then it will fail while validation.

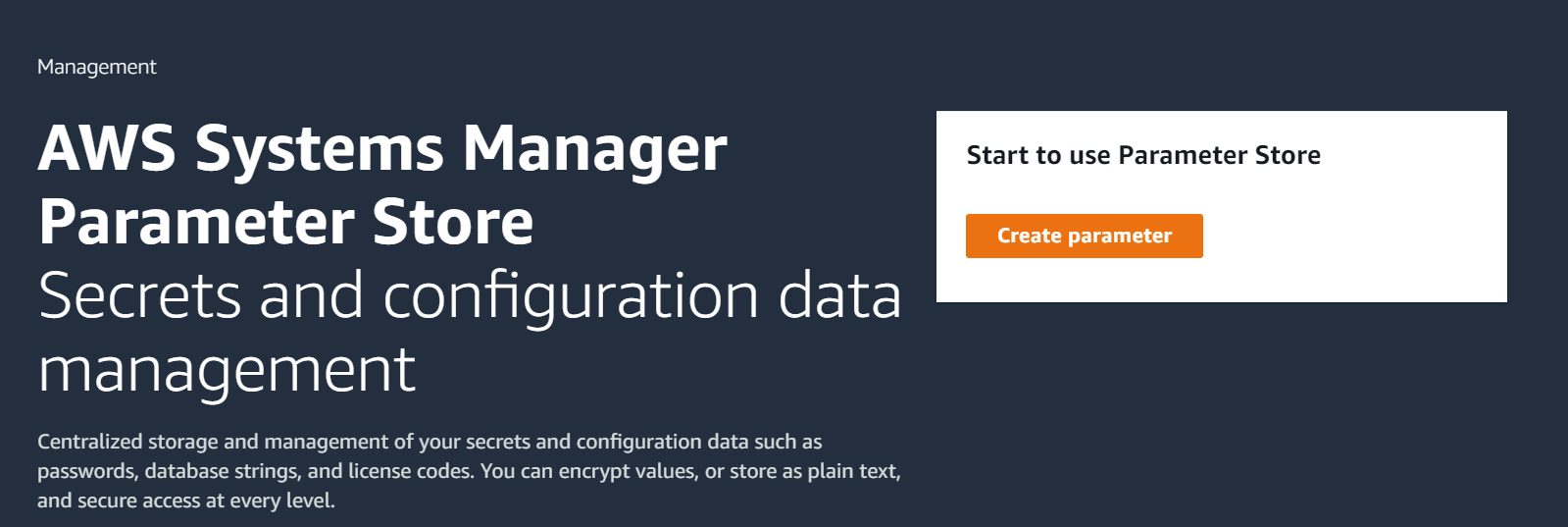
DBPwd is coming as dots because we have declared it NoEcho properties as True.



**SSM Parameter Type**

This is a service which allowed us to store the template parameters outside the template in a service called System Manager Parameter Store.

Use Case for SSM Parameter : The Image ids are not same for all regions and its keep on changing time to time. To update the image ids we don’t have to change all out templates instead we can declare the image id as an SSM type parameter because SSM parameter always pull the latest available ids so in that way our template never go outdated.



**Resources:**

Resources declared and referenced by each other.

We don’t have to define any order. AWS create them systematically based on the references we made in resource declarations.

Resources Type: **AWS::aws-product name::data-type-name**

**Optional Attributes of resources:**

**DependsOn** : If we are not referencing the resources we can use this option to create link.

Ex: Create ECS cluster only after ASG

**DeletionPolicy**: Keeping resources even after the stack deleted. Ex: - RDS databases we don’t want to lose. It can reuse.

**UpdateReplacePloicy**: Similar to deletion policy to keep the resources getting updated while doing stack update.

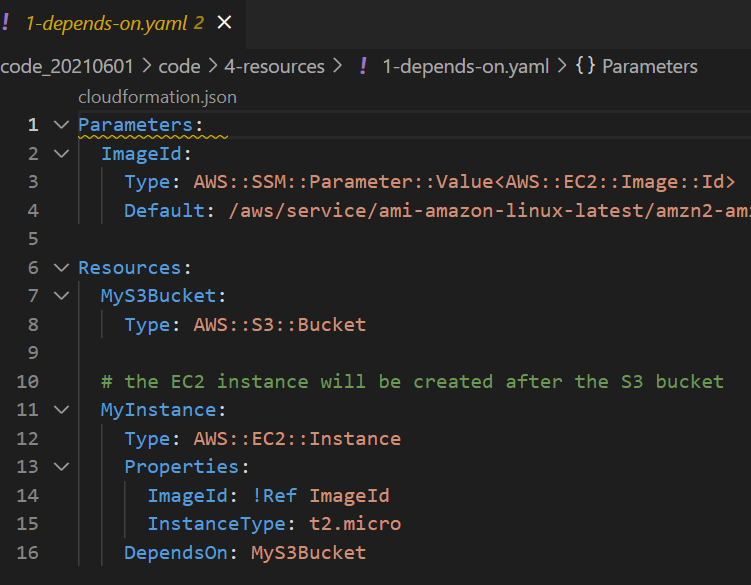
**CreationPolicy**

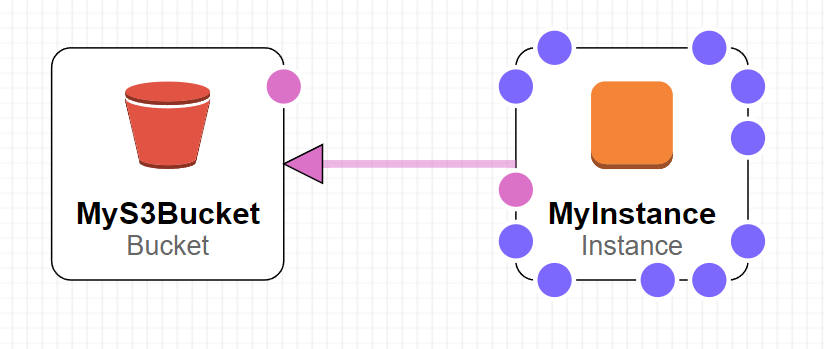
**UpdatePolicy**

**MetaData**

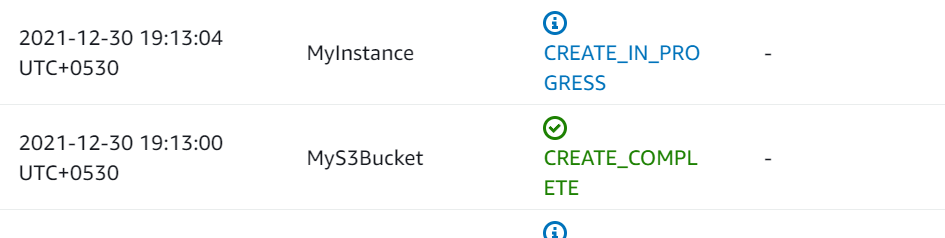
**Task- Creating an EC2 with depend on S3 bucket**

Here we have the template that will create an EC2 instance using type fetched from SSM. Before it creates the EC2 first it will create the S3. Same can also be seen by the pink line in designer session

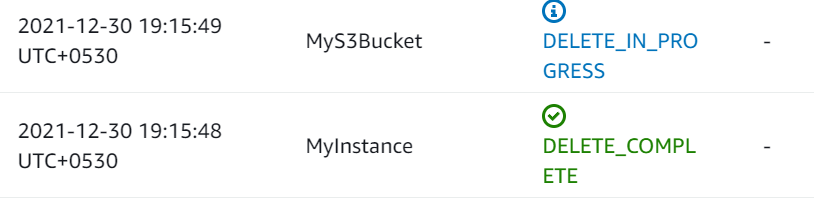




We can see resource creation order from events. It creates EC2 only after S3 bucket created.



We can also see the reverse order while deleting the stack. It will first delete the EC2 and then go for S3



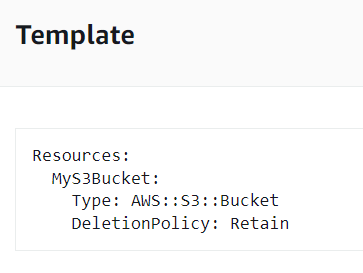
**Deletion Policy:**

1. Deletion Policy = Retain (Works on any resource)
2. DeletionPolicy = Snapshot (Takes a snapshot before deletion)
   1. EBS Volumes, Elastic Cache,
3. DeletionPolicy = Default (Default is deletion)
   1. For AWS::RDS::DBCluster default is Snapshot

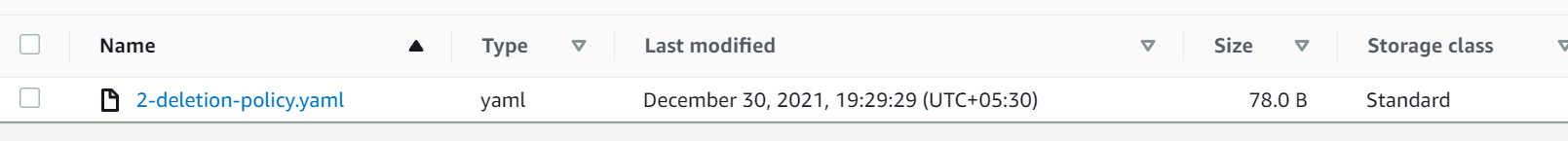
**Task- Creating a stack with deletion policy**

We will create a stack with a S3 bucket with deletion policy retain that means when we delete the stack then it will be deleted but S3 will be retained. And, it will not fail during deletion as S3 has objects inside it.

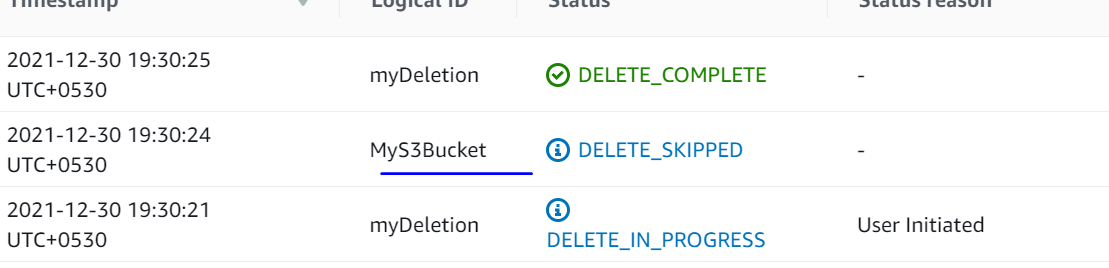
Note: Nonempty S3s can’t be deleted.



We uploaded an object to S3



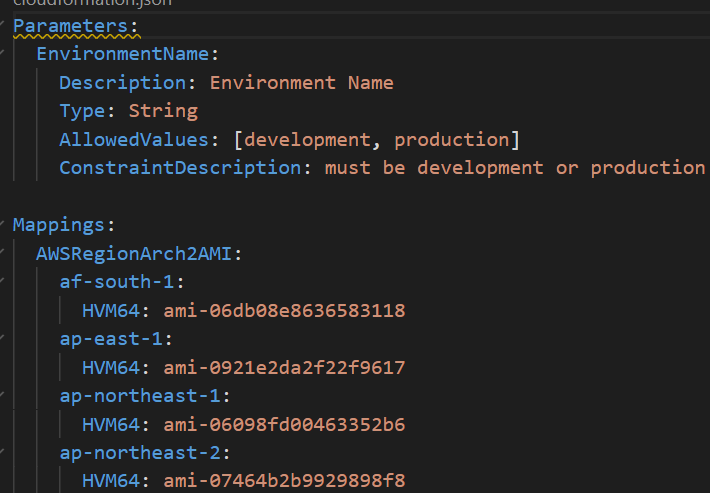
Now we will go and delete the stack. Stack got deleted while skipping the S3



**UpdateReplacePolicy worked on same way as DeletionPolicy.**

**Mappings:**

Mappings are fixed variables declared in template. They are very handy to differentiate between environments, regions etc. All the values are hardcoded in template.

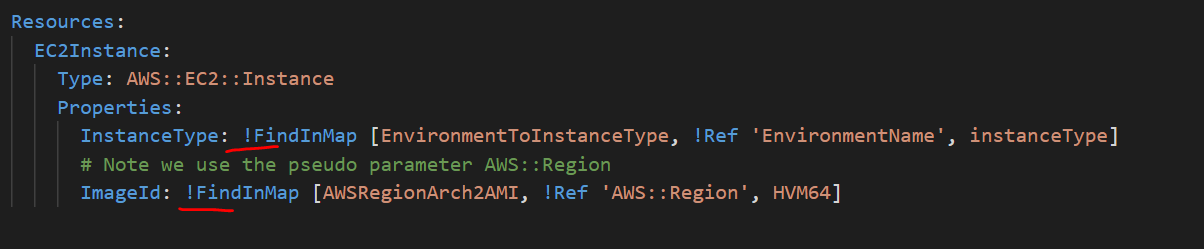


**When would you use mappings vs parameters?**

* Mappings are useful when we already know the values. Ex- Region, Availability zones etc.
* They allow safer control over template
* Use parameter when values are really user specific.

**Accessing Mapping Values:**

* We use **Fn::FindInMap** to return a named value from a specific key.
* !FindInMap [MapName, TopLevelkey, SecondLevelKey]



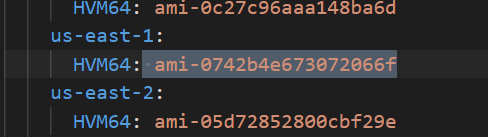
**Task- Creating a EC2 using mappings**

The stack will simply create an EC2 but it will choose the Region and AMI based on our account.

Deployed the 0-mapping-ec2.yml template and it created the EC2



It created with t2.micor which is correct as we choose the dev environment as parameter while creating the stack. If we search the AMI in template we can tally its belongs to our region which is us-east-1



**Concept of Pseudo Parameters**

* These can be used at any time and they are enabled by default.
* These parameters have some default value

Ex: AWS::Region by default have value = us-east-1

[Pseudo parameters reference - AWS CloudFormation (amazon.com)](https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/pseudo-parameter-reference.html)

**CloudFormation Outputs:**

* Its optional thing but we used to export any variable out of the current stack so that any othere stack can use it.
* We can view the outputs in AWS console or CLI

**Task- Exporting one stack variable and use it in another stack using Output**

We will create two stacks.

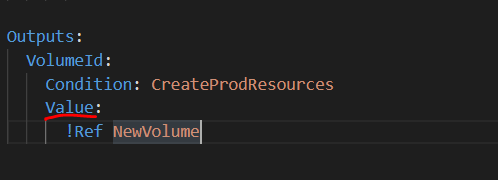
Stack1 = DemoSGExport – It will create SGs and export the group as a variable outside of the stack

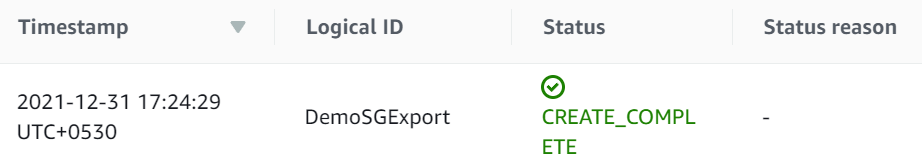
Stack2 = DemoSGImport – This stack will import the SG exported by Stack1 and use it in creation of an Ec2

DemoSGExport



Create the stack. Remember the value it exported is a Global unique variable that means it can be used by anyone in AWS. We are exporting the variable using the keyword Export, if we sue some other value then it can not be exported and used by other stacks.

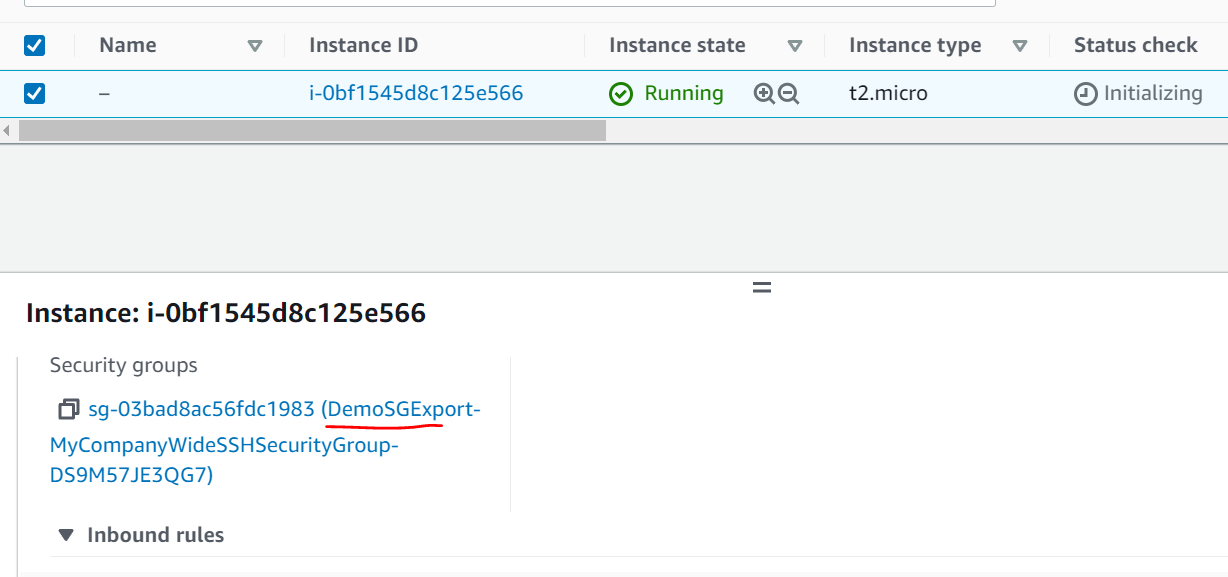




Now we will create Stack 2 DemoSGImport



Under SecurityGroup we are doing ImportValue. We can see the EC2 created with our exported SecurityGroup.



Now if we try to delete the stack we have to delete the Stck2 first before deleting Stack1 as there is a dependency.

**Conditions:**

* Conditions are used to create any resource or outputs conditionally.
* It’s an option.
* Each condition can refer another condition, parameter value or mapping.

How to define a condition?

The syntax is like this:

Conditions:

  CreateProdResources: !Equals [ !Ref EnvType, prod ]

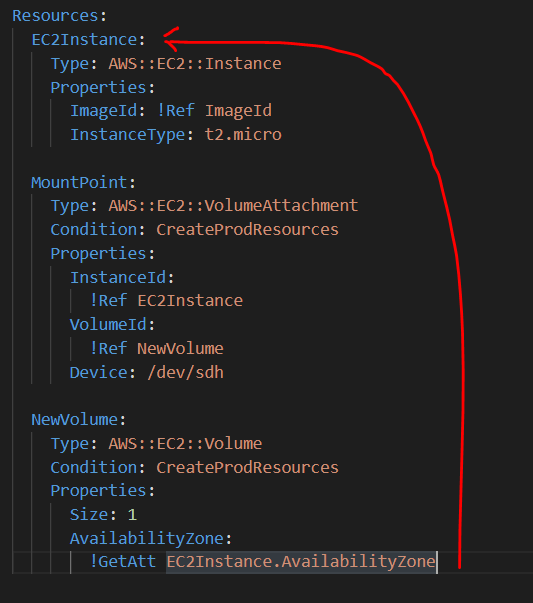
* The condition name could be anything you choose. Logical name.
* Here we are saying apply condition for env type = prod
* The reference EnvType could be a parameter

**Fn::GetAtt**

Attributes are attached to any resource we create. We may not exclusively define these while creating the Resource but they are attached to it which we can extract using Fn::GetAtt and use it based on requirement.

Note: Not every resources has GetAtt functionality.

In below example we are extracting Avilabilityzone information using attribute from an EC2 whereas EC2 created with only explicit information like AMI and Instance type data.

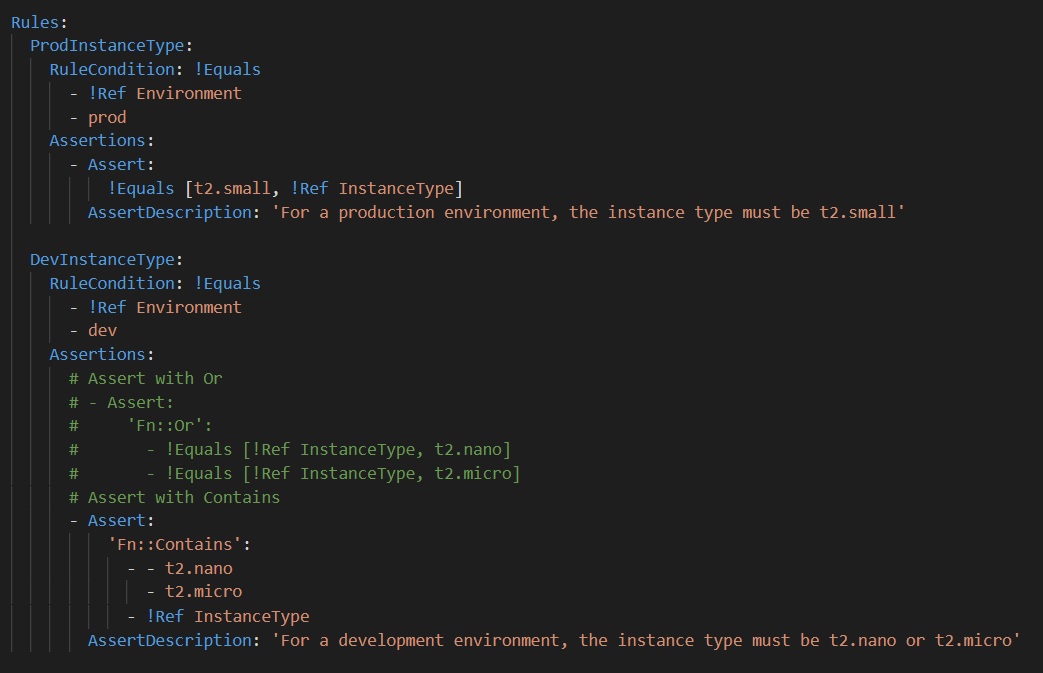


**CloudFormation Rules:**

* Parameter provides the constraints for the checks like Type, AllowedValues etc.
* But to validate the parameters based on other template parameter (cross validation) we can use Rules.

**How to define a Rule?**

Its little bit complicated.



Rule Specific intrinsic functions

* Use to define a Rule condition and assertion.
* Can only be used in Rules section.
* Function can be nested, but the result of a rule condition or assertion must be either true or false.
* Supported Function
  + Fn::And
  + Fn::Contains
  + Fn::EachMemberEquals
  + Fn::Equals ..etc

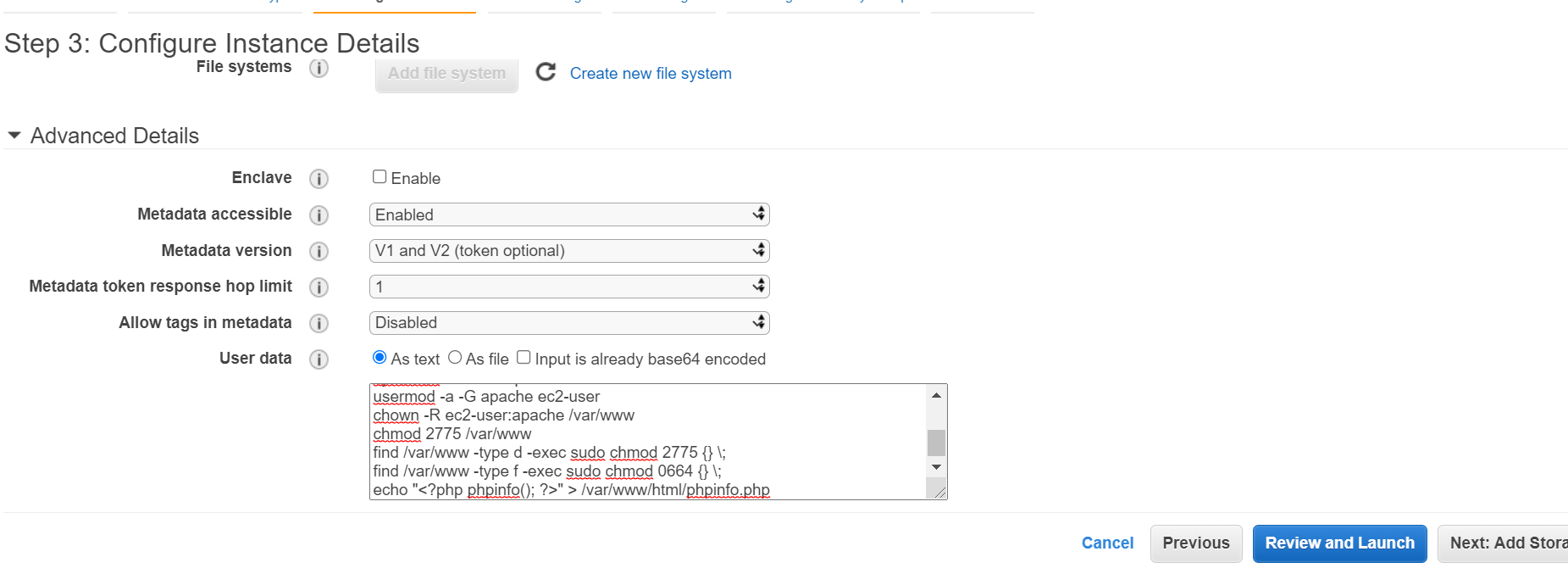
**EC2 User data:**

Ec2 user data is set of commands we mentioned while provisioning the EC2 instance so that while instance get crated this script will auto run so that what we want to configure it get configured automatically right after the EC2 up and running.

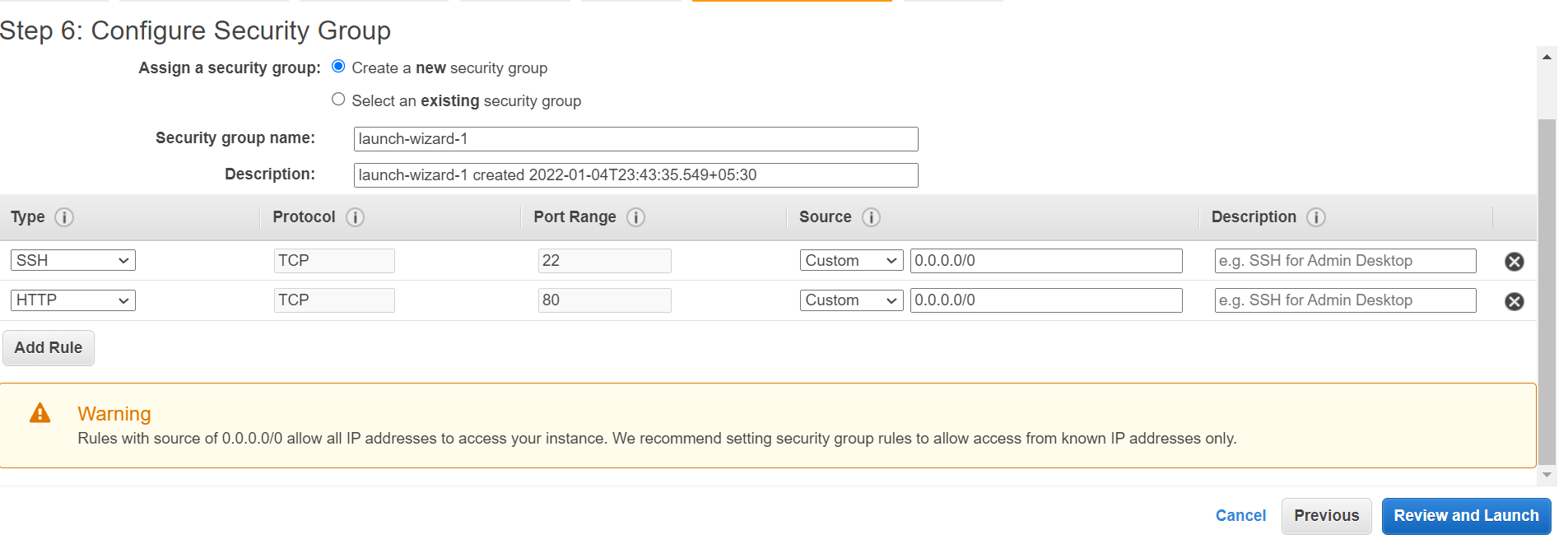
Task : Create a EC2 and Setup a Web server (Using User data)

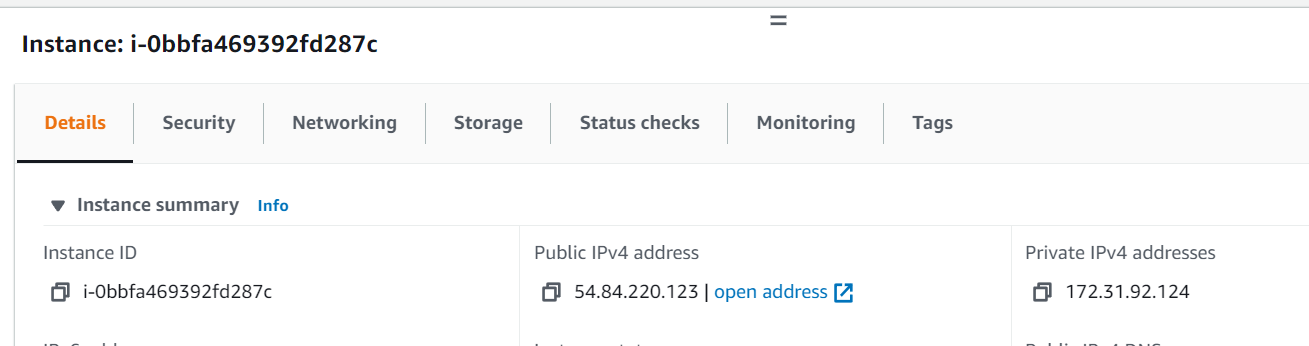
Here we are going to create an EC2 and configure it as a web server using EC2 user data approach manually

Create an EC2 from Console use 0-ec2-user-data.sh userdata

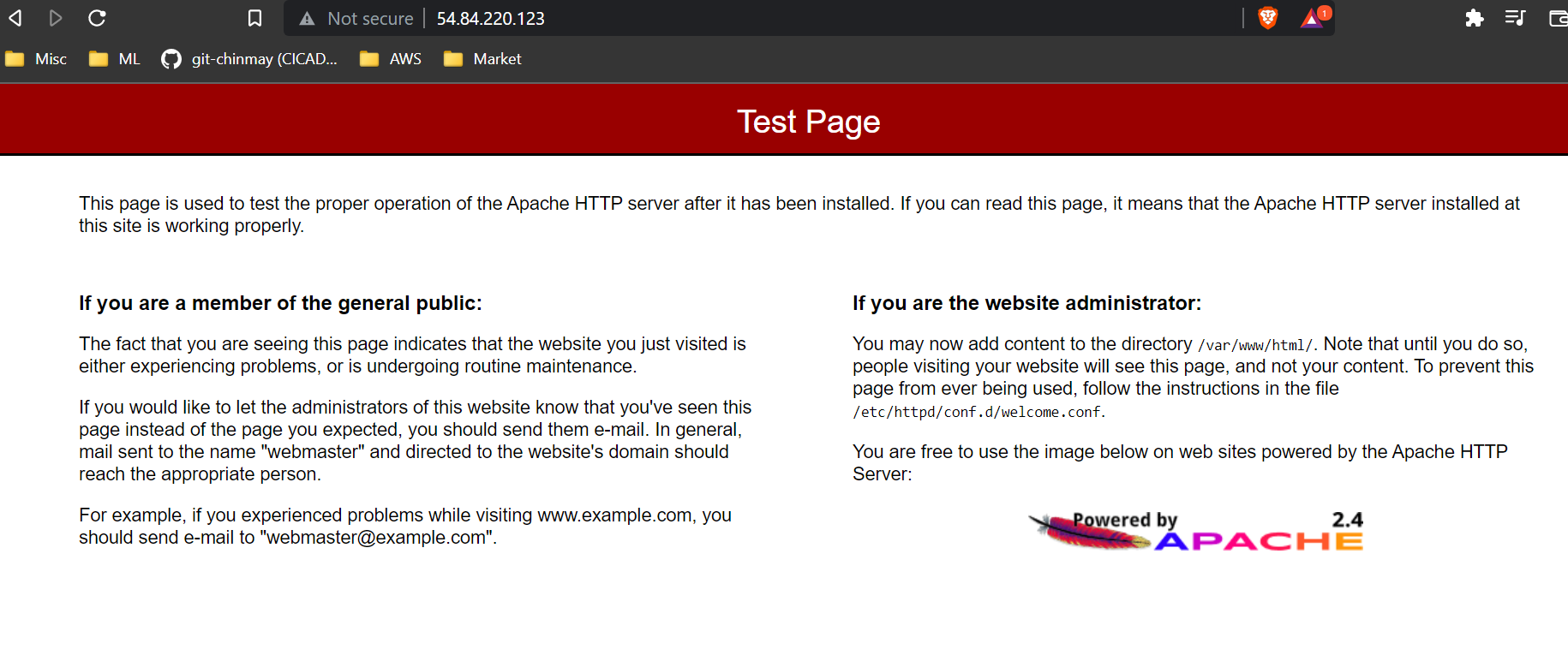


Configure security

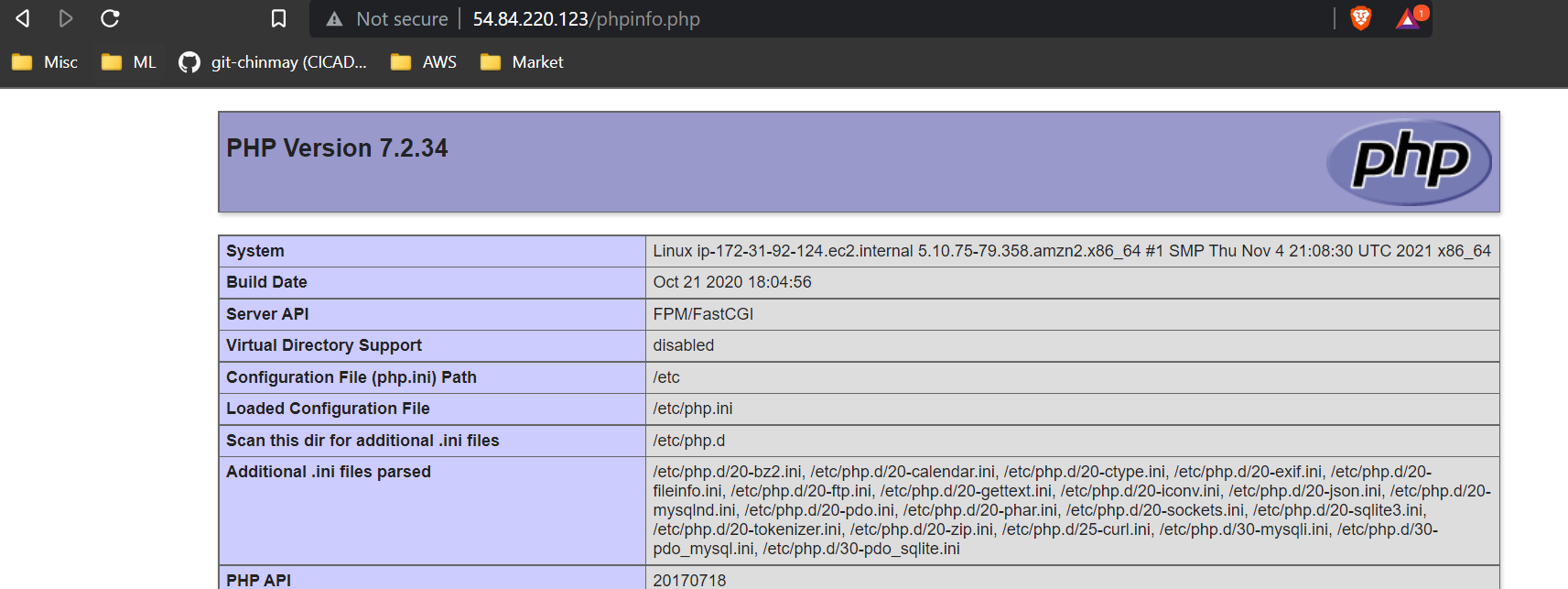
Open the public IP in browser. It will take time to get configured so have some patients (Copy the IP, don’t use the open address option)



Webpage is ready.

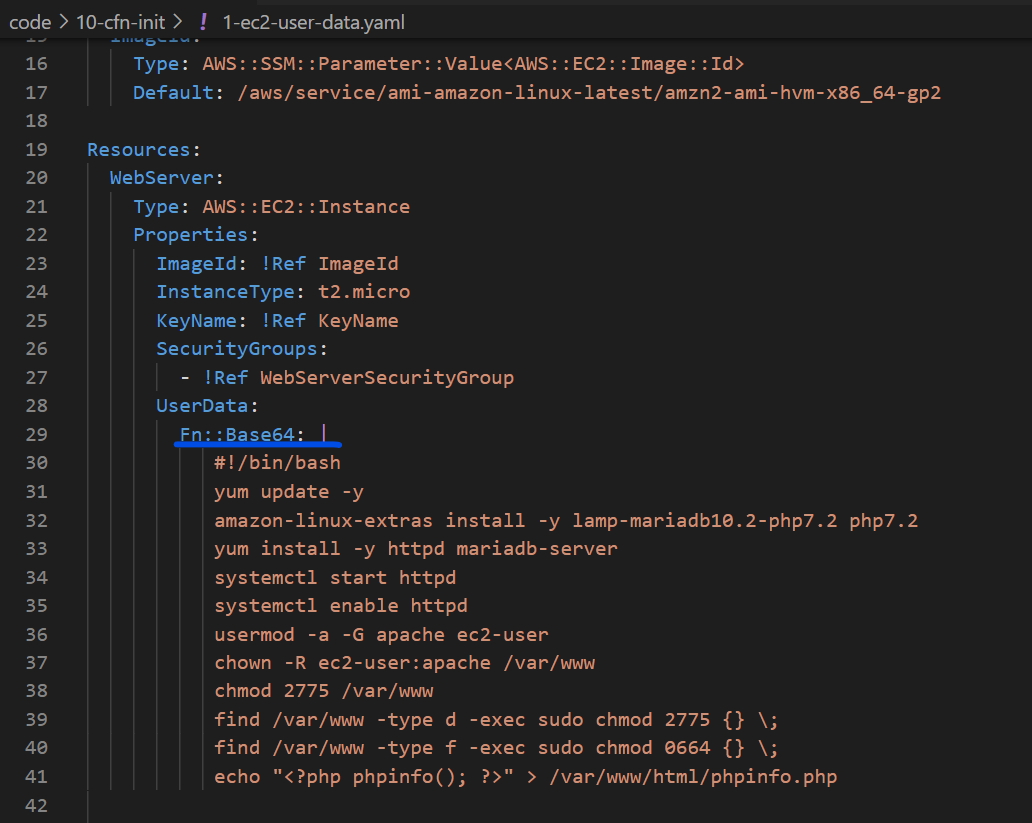


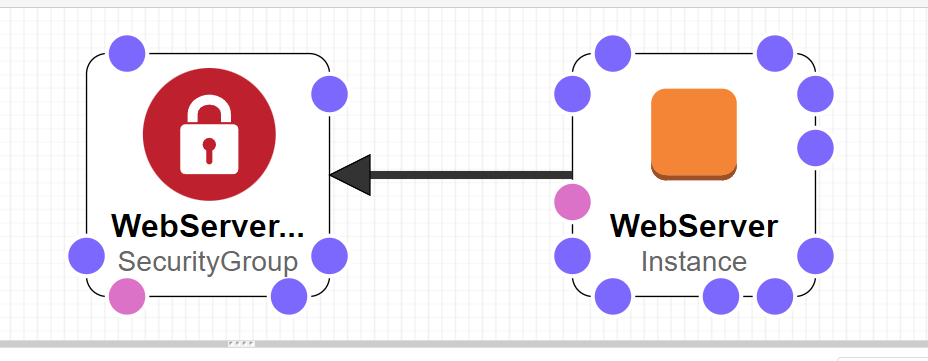
Php also running



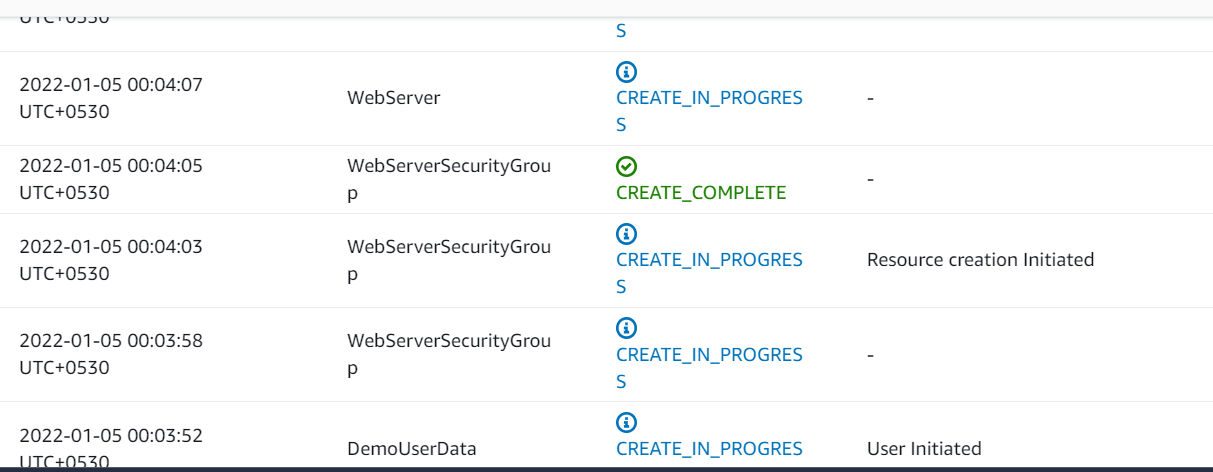
We can perform all above steps using CloudFormation Template

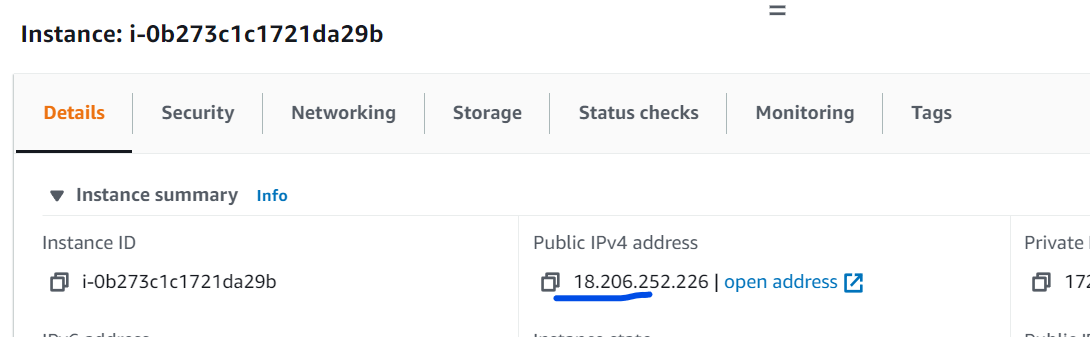
Important function is to notice use of Fn::Base64 to declare the userdata script

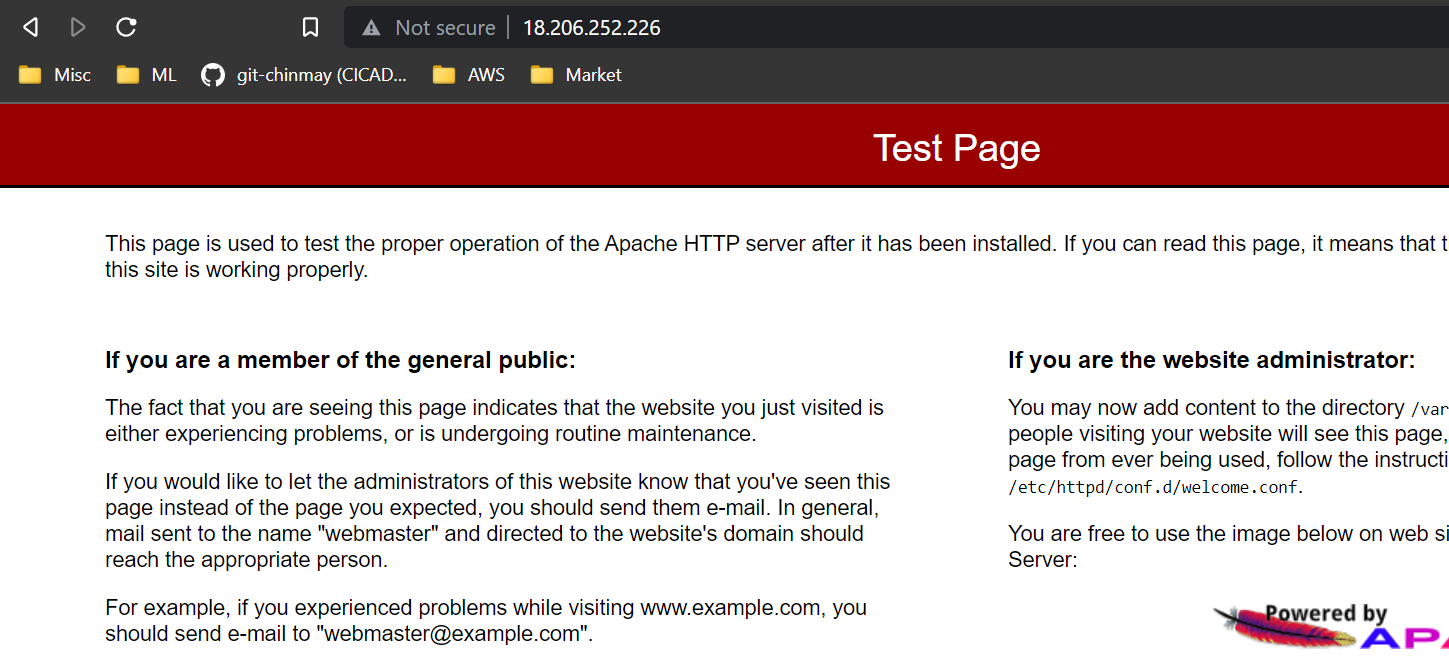




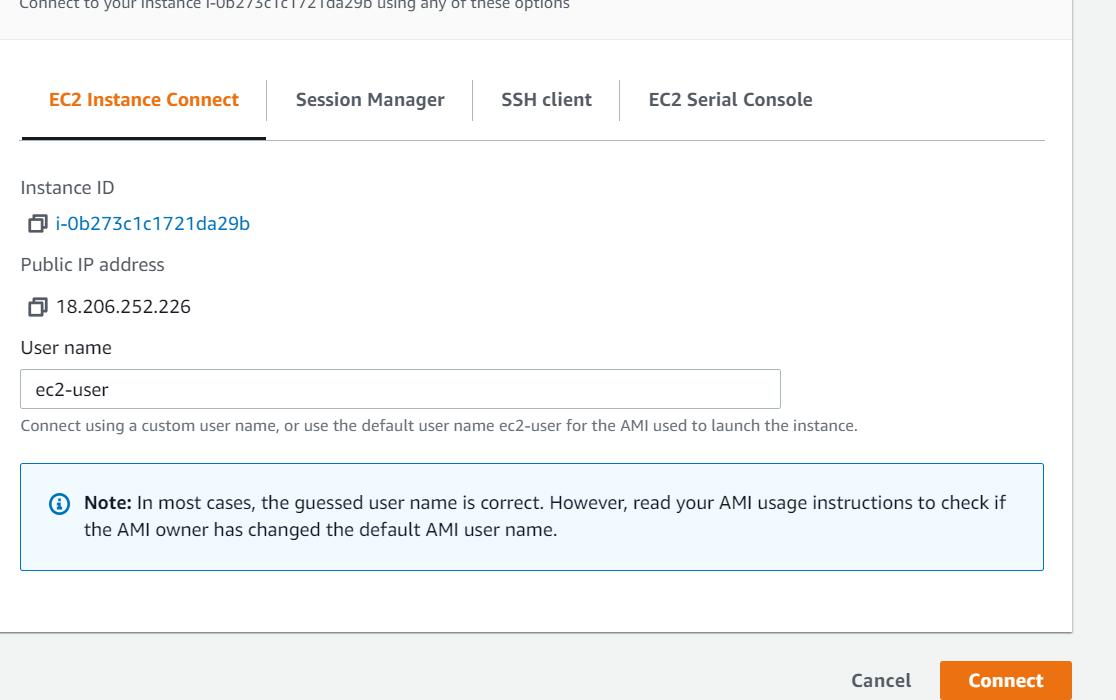
Create the stack

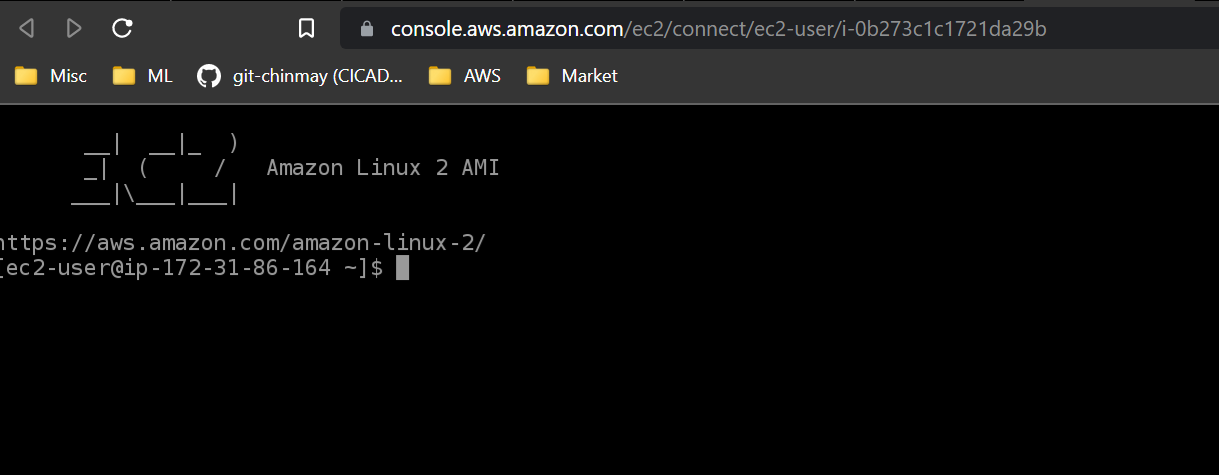




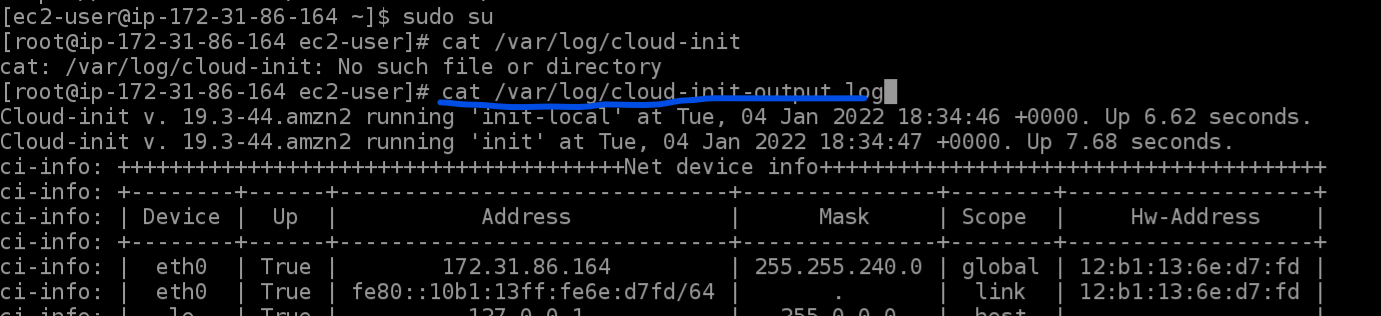


If we connect to the EC2





If we want to see what are the commands runs we can see inside the EC2 logs



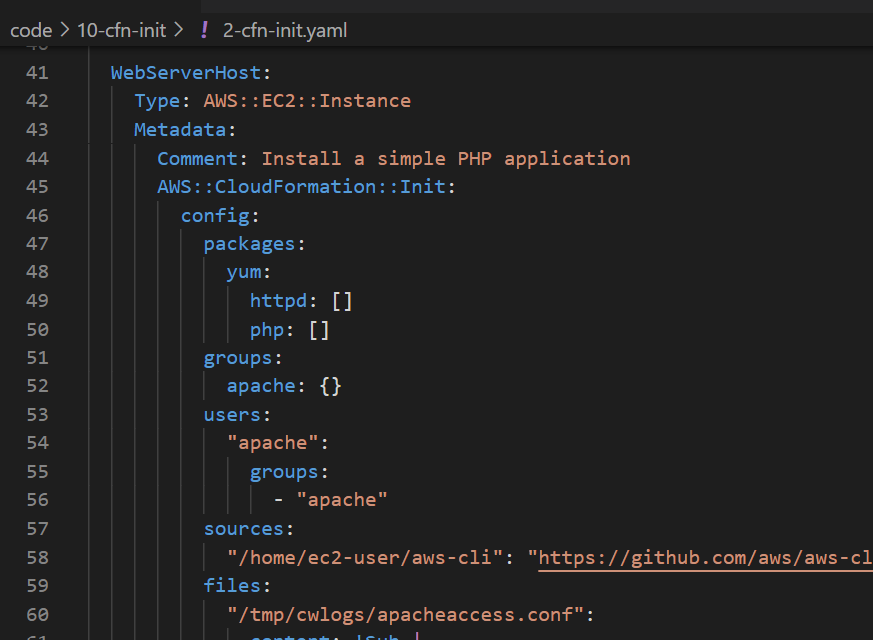
**CloudFormation helper Script:**

There are 4 python scripts come default in Amazon AMI Linux 2, we can also install them using yum on non Amazon Linux2 machines.

* cfn-init : Used to retrieve and interpret the resource metadata, installing packages, creating files and starting services.
* cfn-signal : wrapper script used by resources to talk back to CloudFormation for example to say resource creating was successful or not
* Cfn-get-metadata : A wrapper script to retrieve all metadata of a resource or path
* Cfn-hup : A daemon to check the updates to metadata and run custom hooks if foun any update.

**Cfn-init :**

Init structure is like below



Packages: used to download and install packages

Groups: Define user groups

Users: Define users and which group they belong to

Sources : Downlaod or archive files inside the Ec2

Files: Created files inside the EC2 using inline or can be pulled from a URL

Commands : run a seies of commands

Services : Run series of services using sysvinit

**AWS::CloudFormation::Authentication:**

Used to specify authentication credentials for files and sources in AWS::CloudFormation::Init

Two types:

* Basic : When the source is a URL
* S3 : When the source is an S3 bucket

**Function Fn::Sub : (!Sub)**

* Substitute function. Substitute a variable from a text.
* We can use it combing with Reference and Pseudo variables.
* String must contain ${<variable name>}

**Services:**

* Launch a bunch of services at EC2 instance launch.
* 
* It ensures services are started when file changed or package are updated.

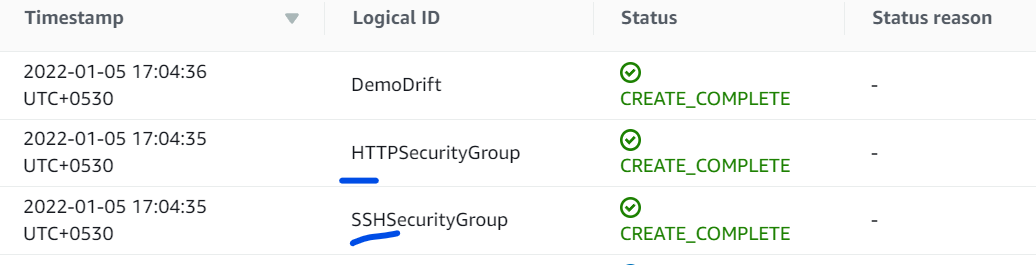
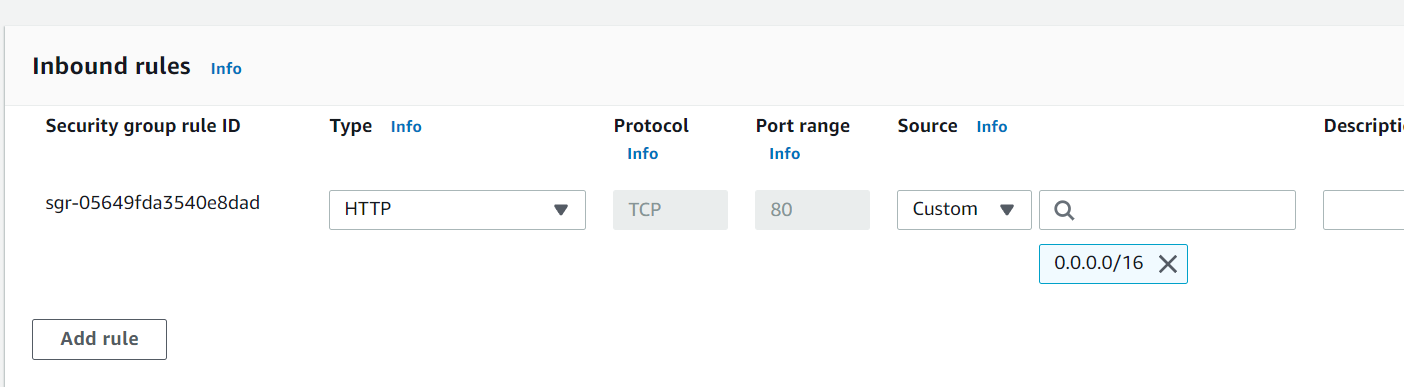
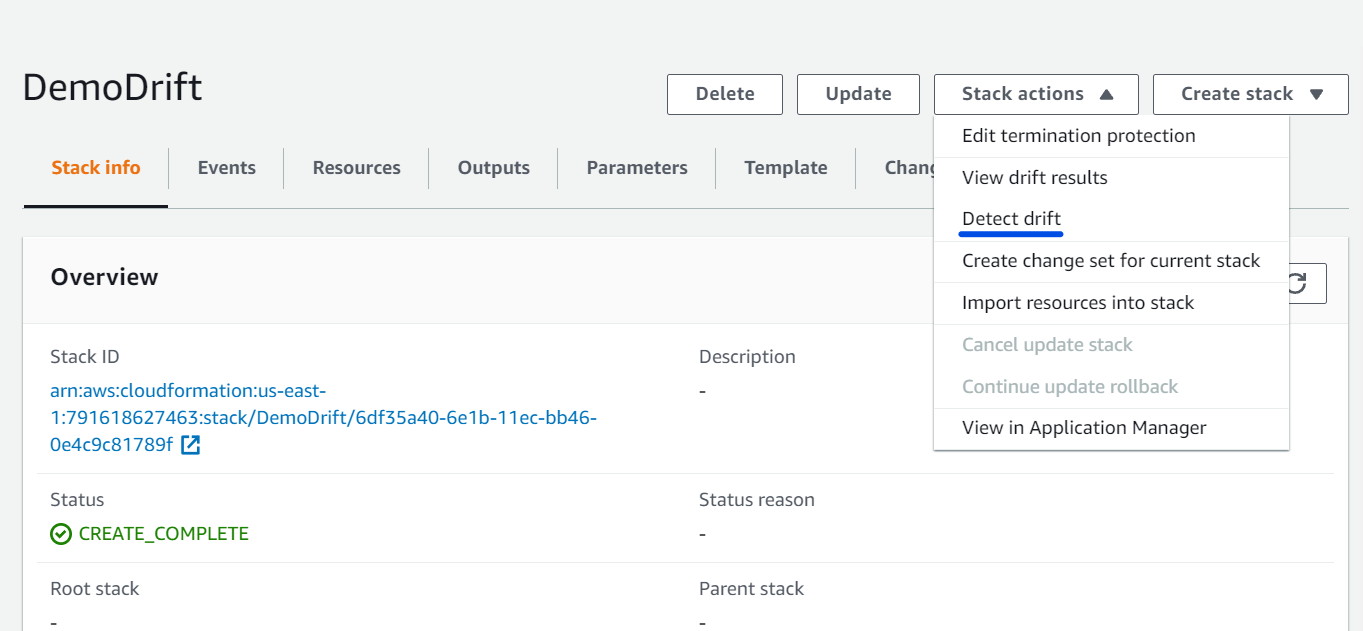
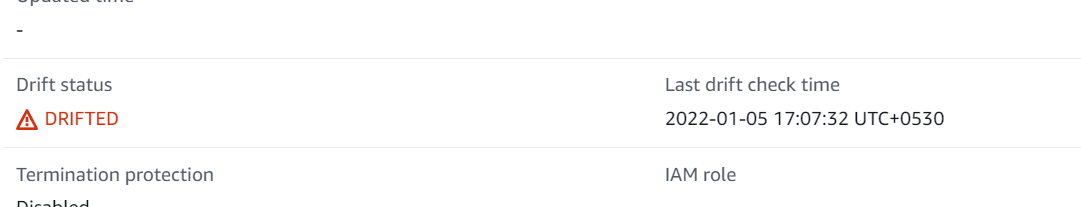
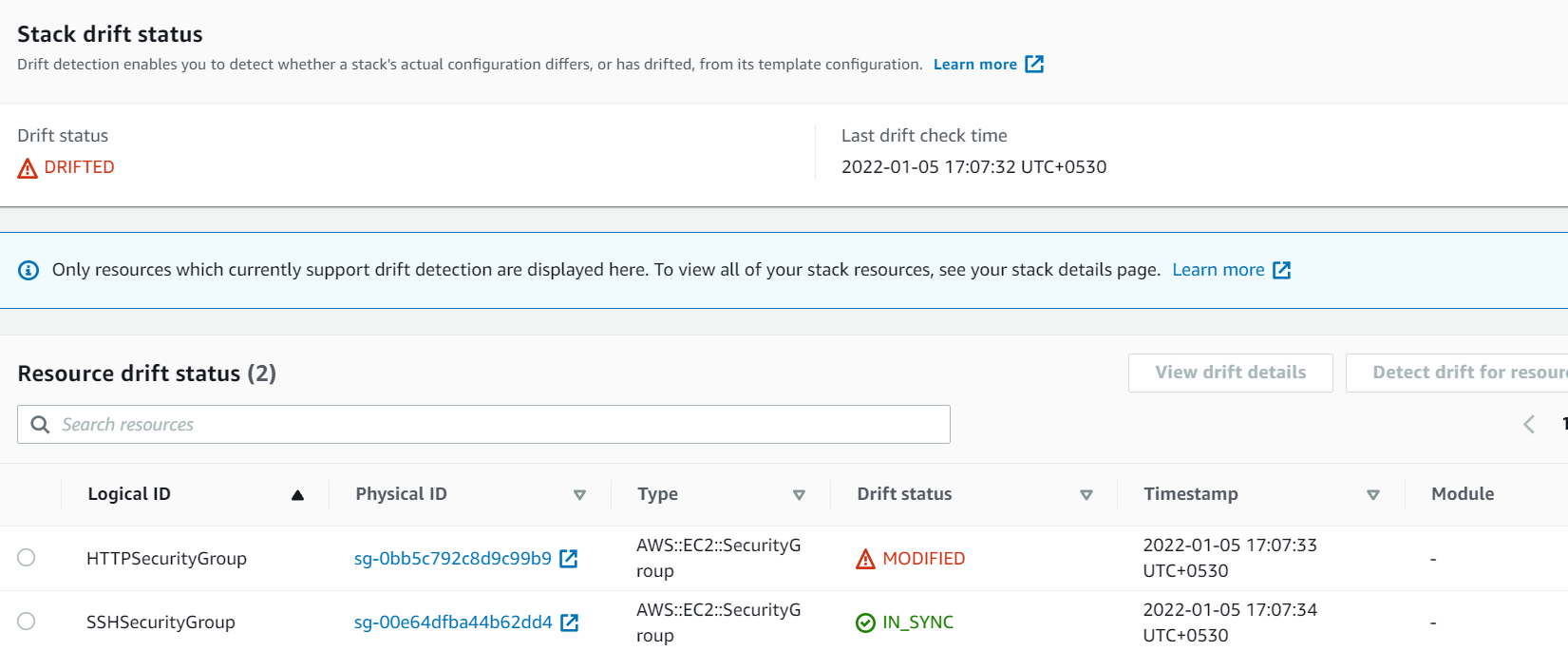
**User Data vs CloudFormation::Init vs Helper Scripts**

* EC2 User data is an imperative way to provision/bootstrap the EC2 instance using Shell syntax
* AWS::CloudFormation::Init is a declarative way to provision/bootstrap the EC2 instance using YAML or JSON syntax
* AWS::CloudFormation::Init is useless if it’s NOT triggered by a script within the EC2 User Data
* Triggering AWS::CloudFormation::Init inside EC2 User Data is done by using cfn-init or cfn-hup

**CloudFormation Drift:**

* By using template, we are able to build the infract structure but nothing prevents individual users to modify the configurations.
* CloudFormation Drift compares the entire stack or individual resources in it and shows if any difference is there with respect to original template.

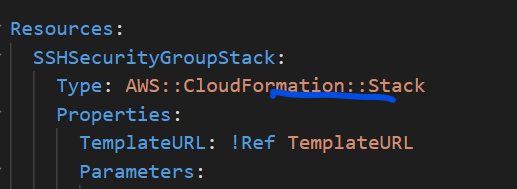
**Task: Create a stack and modify some configuration and detect the changes using Drift**

* Create the Stack using drift-security-group.yml template
* 
* Now modify one of the Security Group
* Modified the inbound rule
* 
* Now run the Drift
* 
* 
* 
* 
* 
* 

**Nested Stacks:**

Stacks that are part of another Stack. Nested stacks are not sharable.

As soon as ou saw below type for a resource understood that that it’s a nested stack



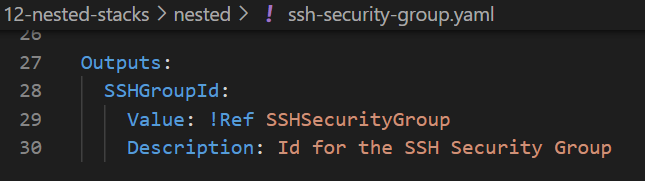
**Task: Nested stack demo**

We will have two stacks

* Stack-1 = Will create a Security group
* Stack-2 = Will create an EC2 instance

But we will use stack-1 for assigning the Security group to Stack-2 EC2 creations. We will basically add Output to expose the SG ids and will use it in Stack-2.

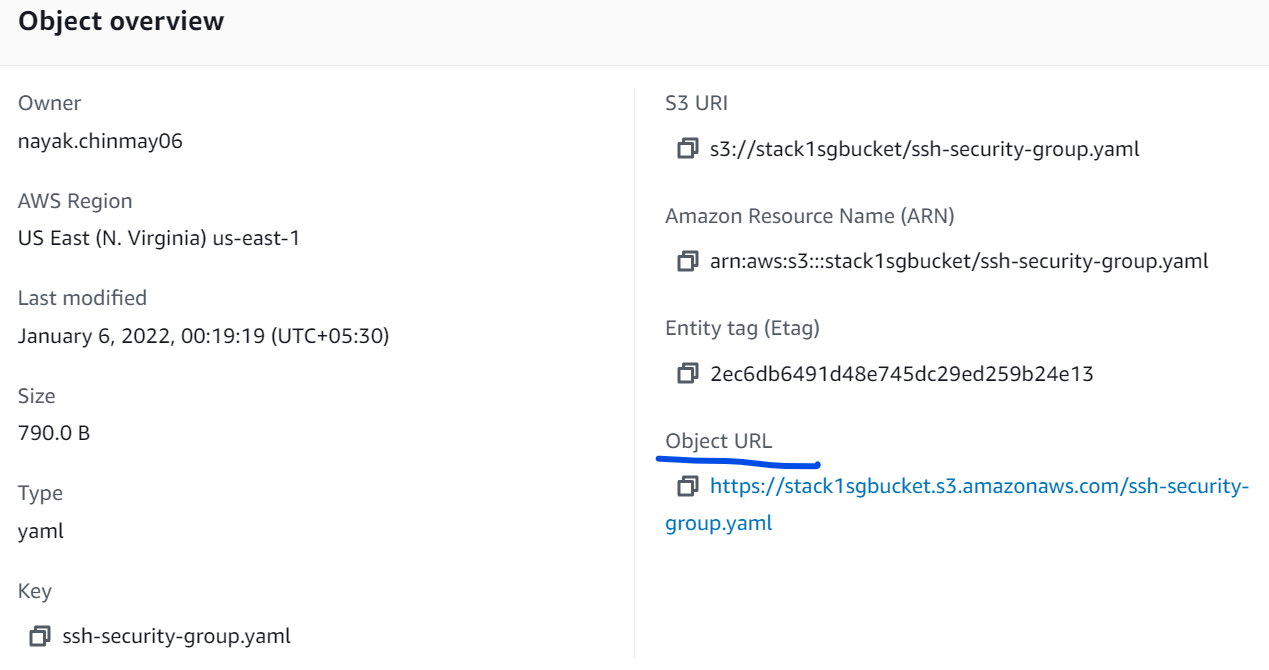
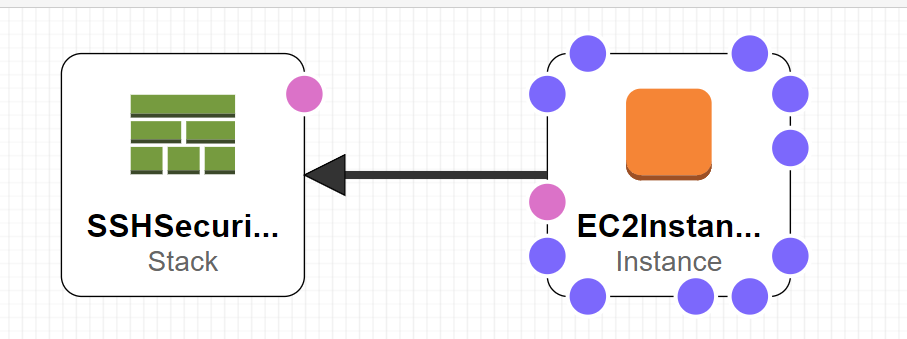
Stack-1



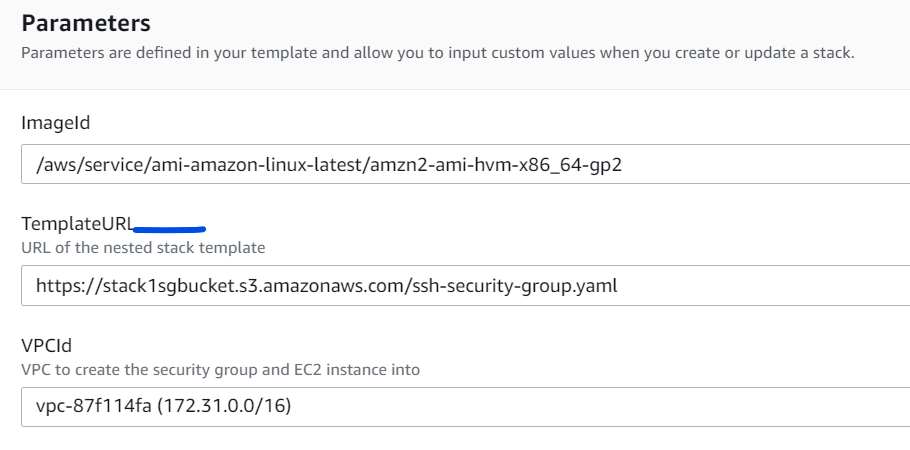
Stack-2

* Template URL we will get from Stack-1 properties
* Securitygroupids we are fetching using the GetAtr function.

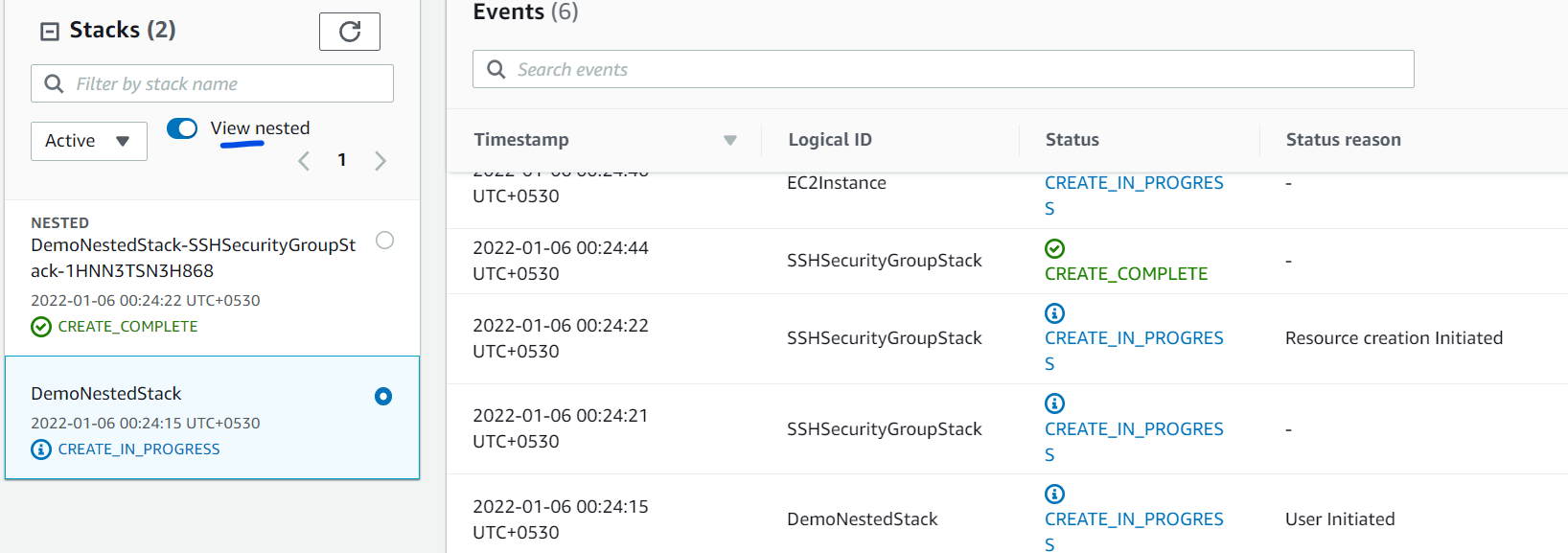


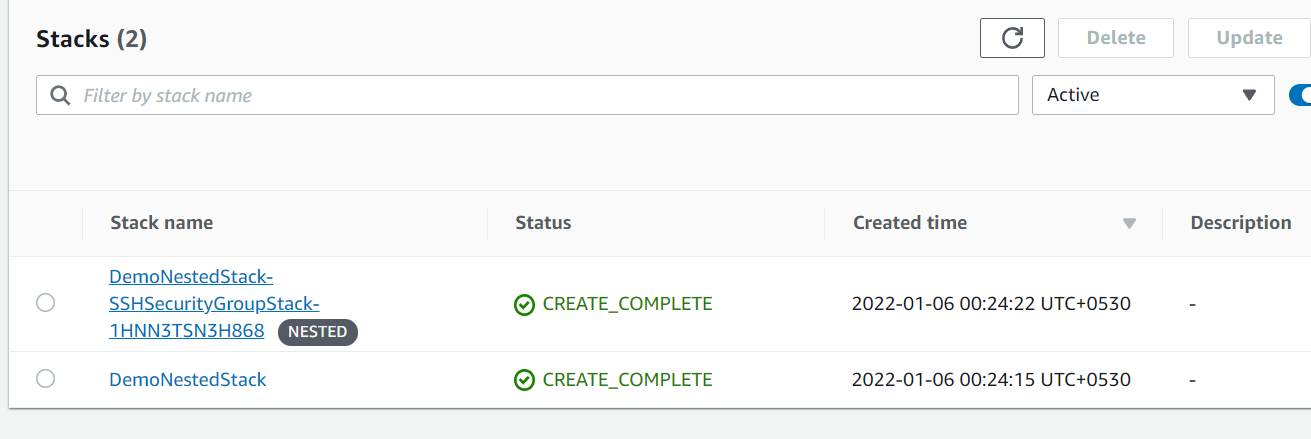
* First put the Stack-1 template into a S3 bucket so tat we get a template URL
* Copy the object URI
* 
* <https://stack1sgbucket.s3.amazonaws.com/ssh-security-group.yaml>
* Now create the Stck-2 on CloudFormation
* 

Fill the template url parameter



We will observe a change in left side

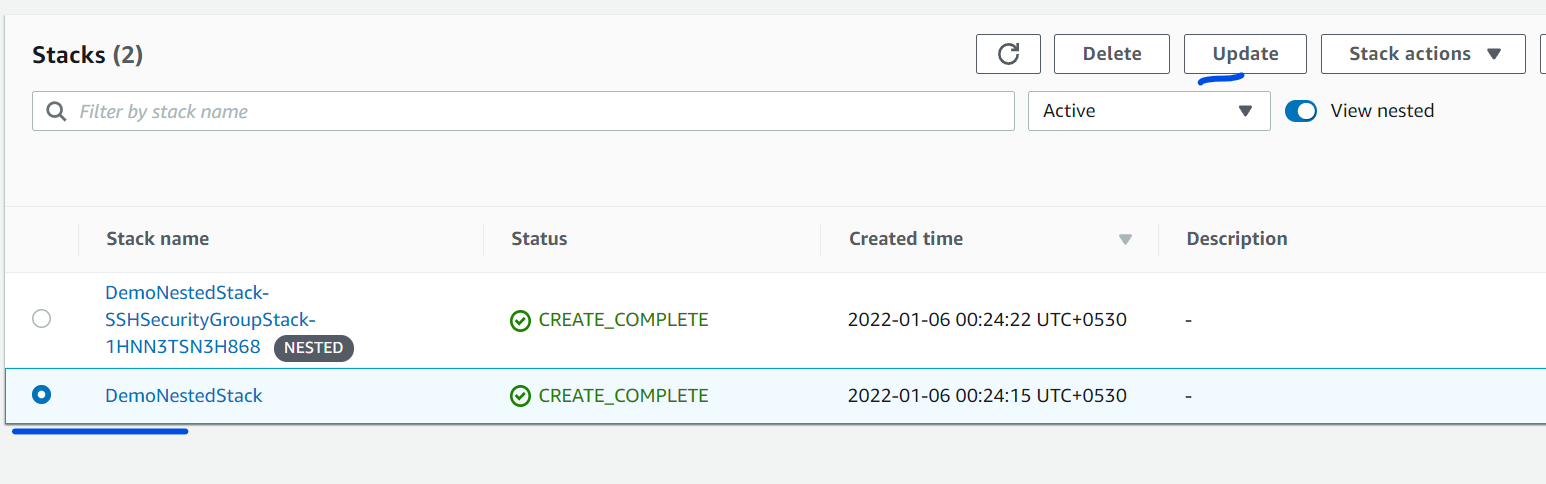




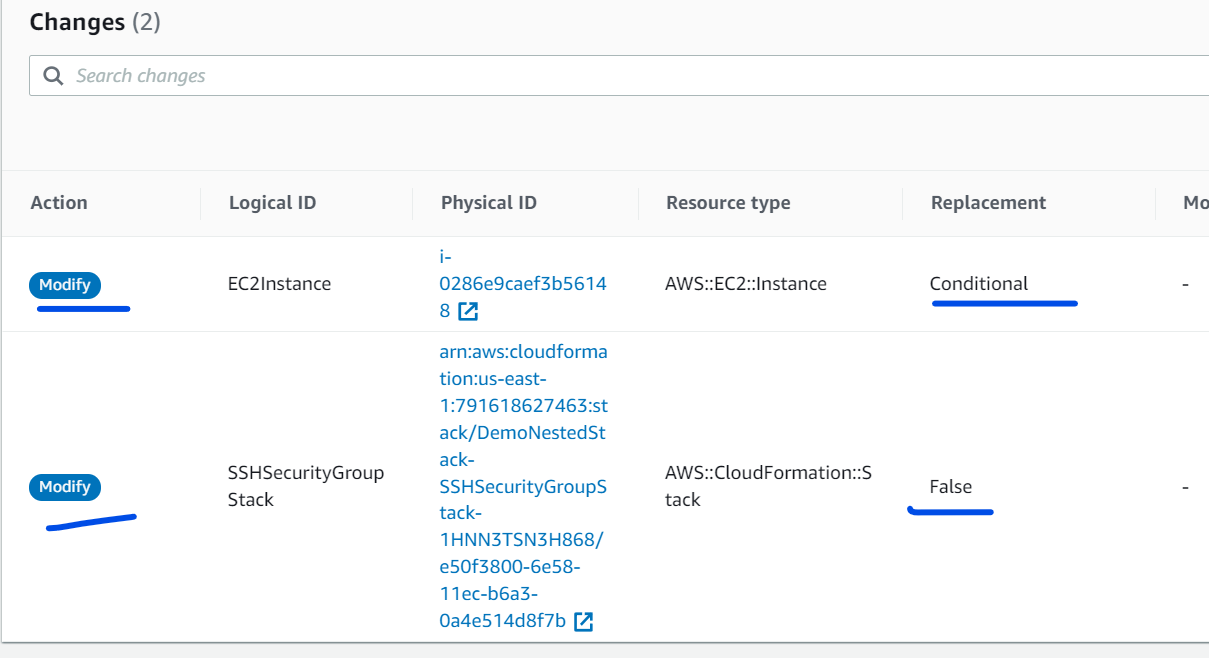
If you update the child template then we have to redo the same steps

Upload the updated template to S3

Come to the CloudFormation and update only the Root/Parent Stack

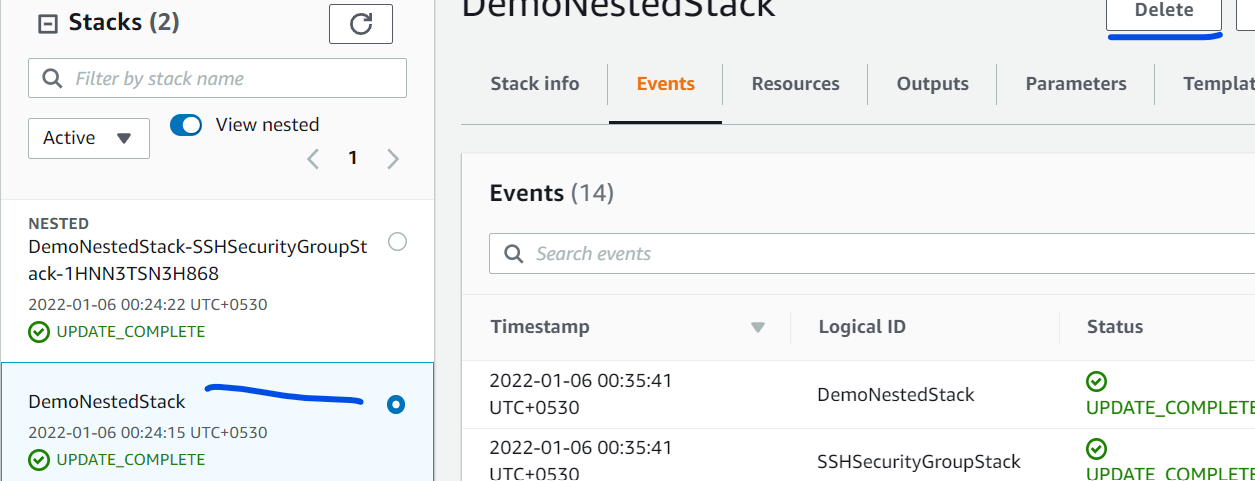


Although we are not changing anything on root template still its showing modified bcz it may be affected due to change in child template.



Deleting a Nested Stack:

Never ever delete the child/nested stack. Always delete the Root/Parent stack.



Exported Stack Output Values vs. Using Nested Stacks:

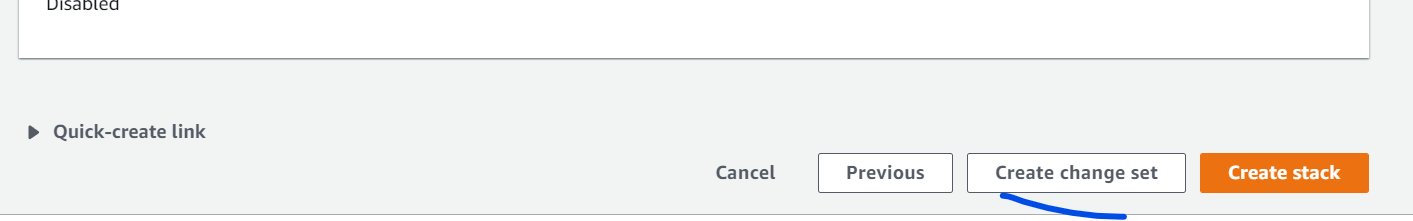
* If you have a central resource that is shared between many different other stacks, use Exported Stack Output Values
* If you need other stacks to be updated right away if a central resource is updated, use Exported Stack Output Values
* If the resources can be dedicated to one stack only and must be re-usable pieces of code, use Nested Stacks
* Note that you will need to update each Root stack manually in case of Nested stack updated

**CloudFormation StackSets:**

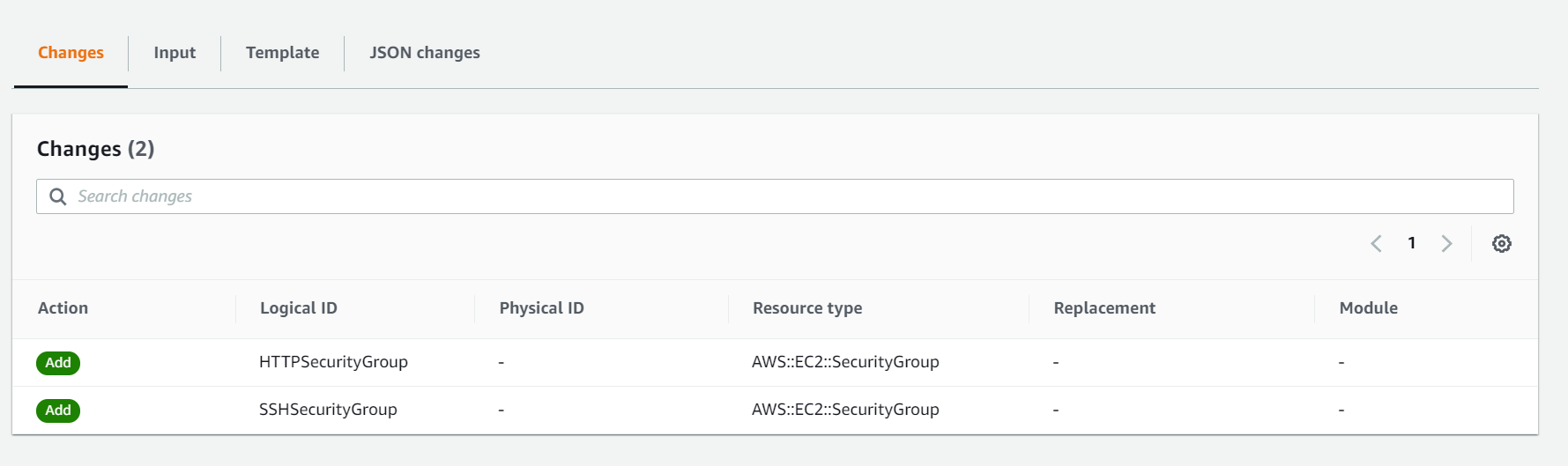
* StackSets allows create, update, deploy and delete of a stack on multiple regions/accounts.
* The change in stackset automatically made changes to all concerned region’s/account’s stacks.
* Administrator account should create the StackSet.
* Updates always affects all the stacks. Selective update not possible.
* We can delete a stack and resources for particular region/ account
* Delete a Stack from StackSet
* Delete all Stacks from StackSet (Preparing the StackSet Deletion)
* To delete the StackSet first we have to detach the stacks inside it and then delete the set. To clean it completely we have to delete the Stacks individually as we so normally.

**Changeset:**

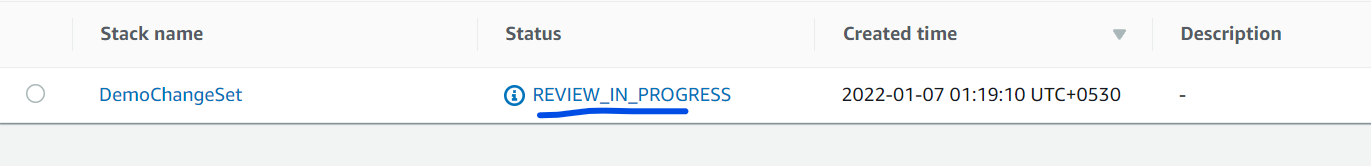
Its gives us a preview of the Stack before the actual creation of the stacks. We can examine the various configuration before the actual creation.



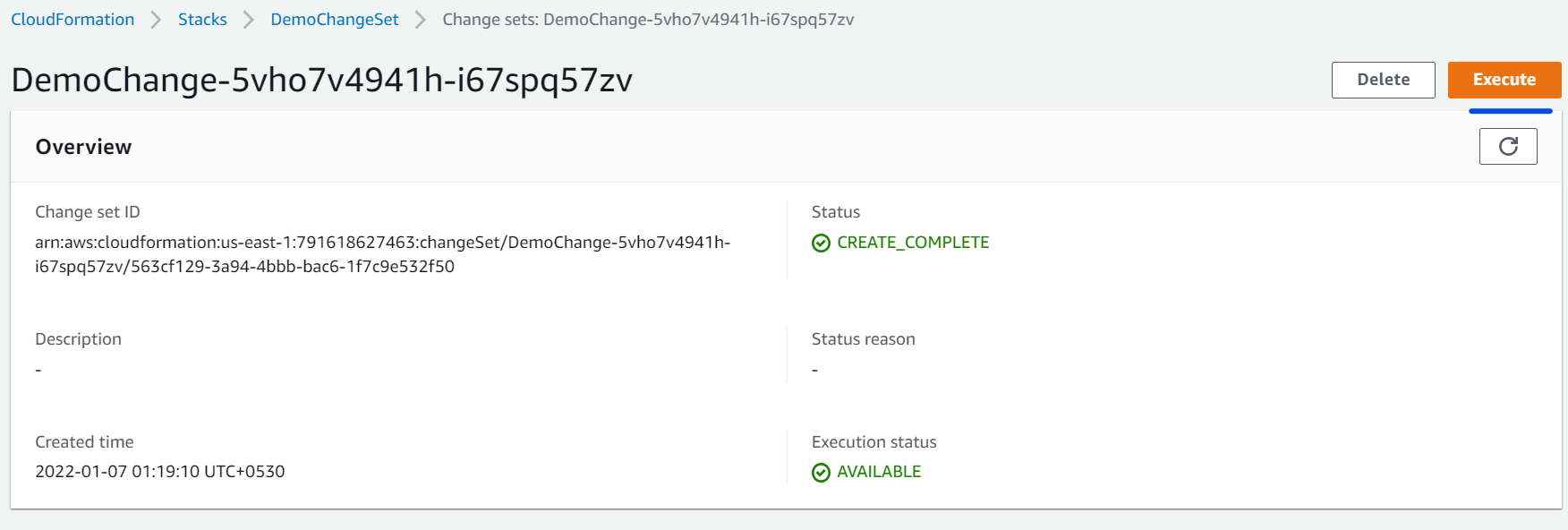
Click on Create Change Set

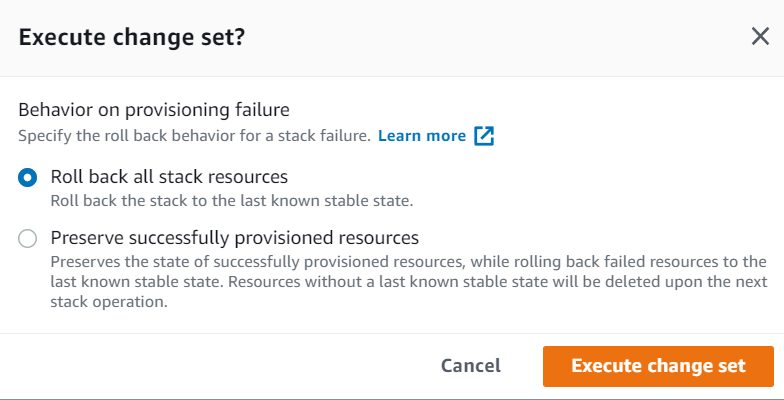


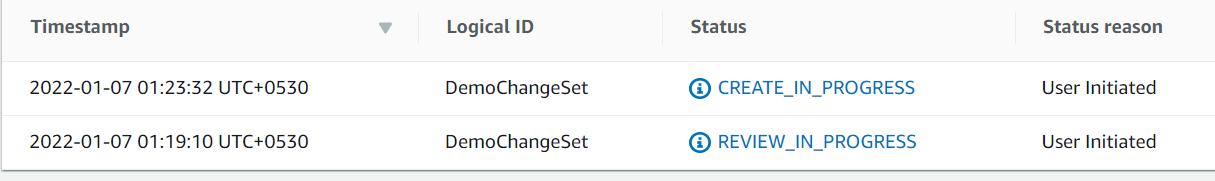
We can see all the sections as previews. If we go back to the Stack Page it will be in Review in progress state. Select the stack



If you are happy with the review we can then got for Execution







This is a great option in organisation to do the review of the stack by senior member before deploying it.

**Rollback Triggers:**

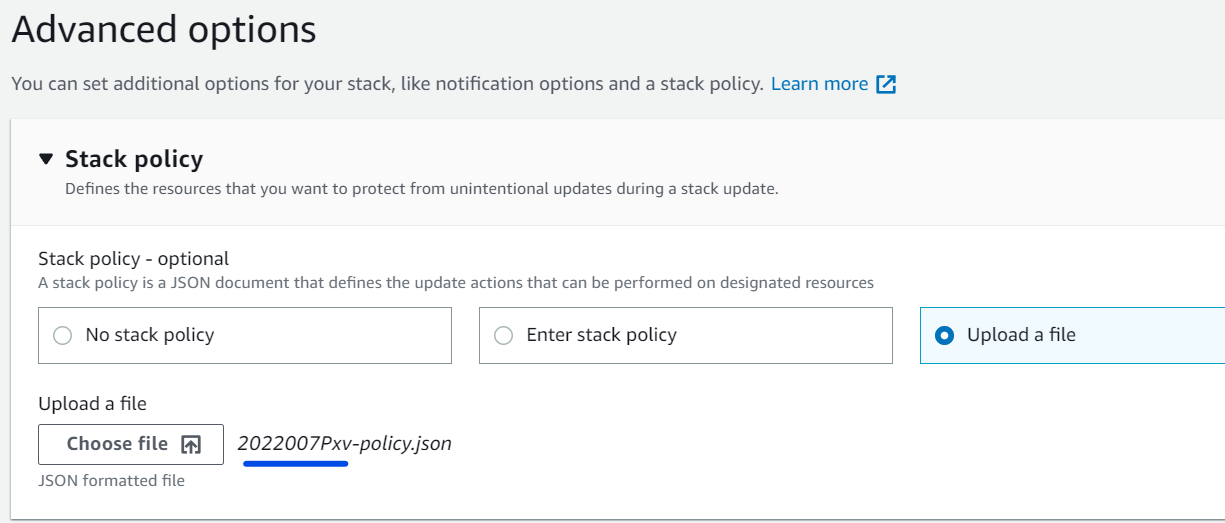
* When a stack creation/update is ging on we can monitor the CloudWatch alarm and if it status at alarm state then it can trigger the rollback of the stack operations.
* CloudFormations monitors the specified CloudWatch alarms.
* We can have upto 5 CloudWatch alarams.
* In Nested stacks rollback of the parent stack will also rollback all the child stacks.

**Stack Policy:**

StackPolicy provides an extra layer of protection to Stack resources while getting updating or deleting.

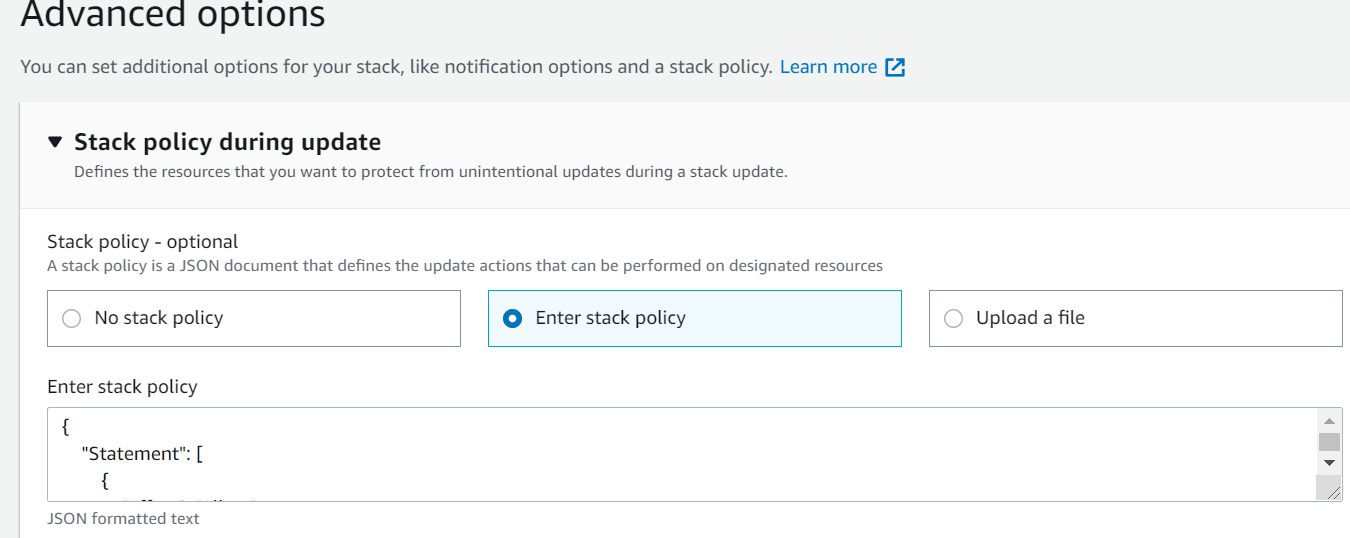
Task: Create a stack with a policy and then try to upload the stack which didn’t follow the policy and update get failed.

We can add the poliy under Advance section of the Stack creation page

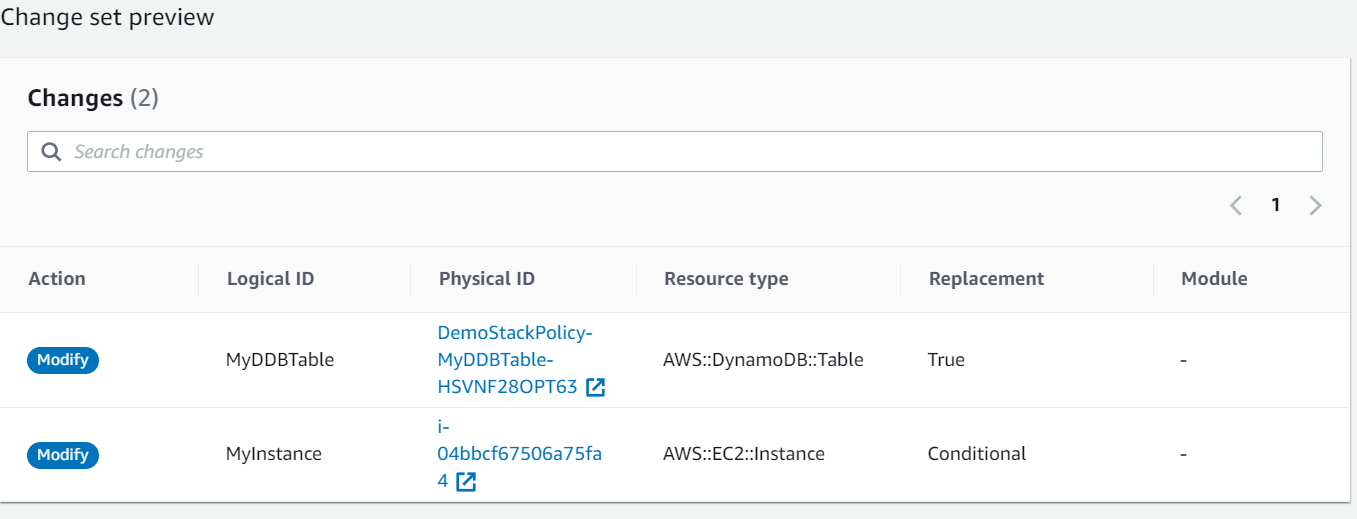


The above stack basically created an EC2 and a Dynamo DB and in Policy we defined that the update to stack is possible but anything change to DynamoDB will not be allowed. Hence when we will update the Stack mentioning to change in DynamoDB then policy will breech and cause the rollback.

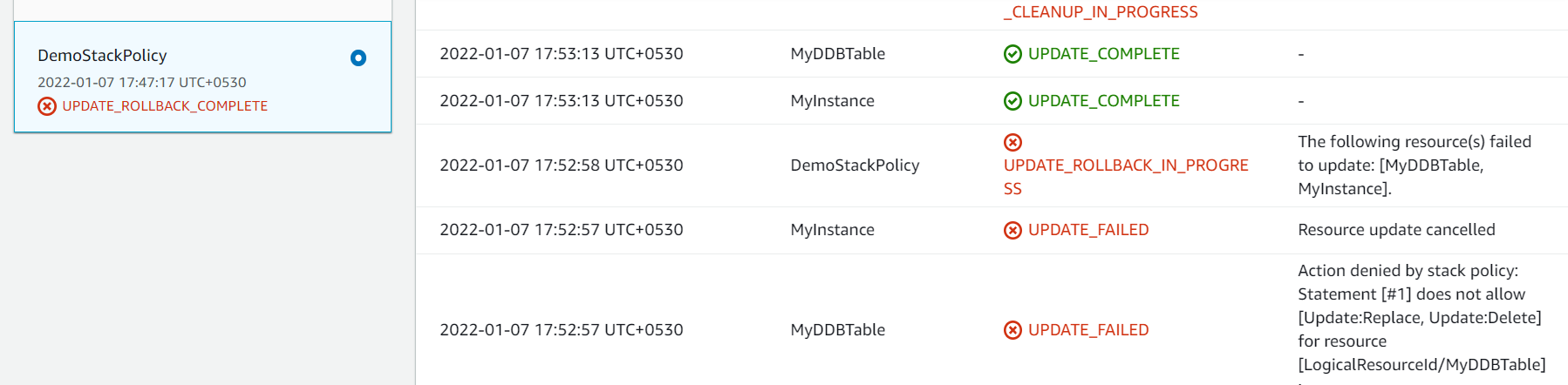
While updating the Stack also we can update the policy.



But here we will go with old policy



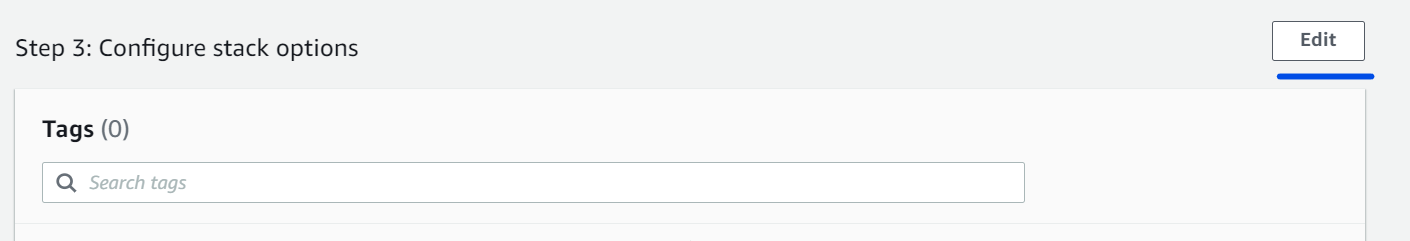
Rollback



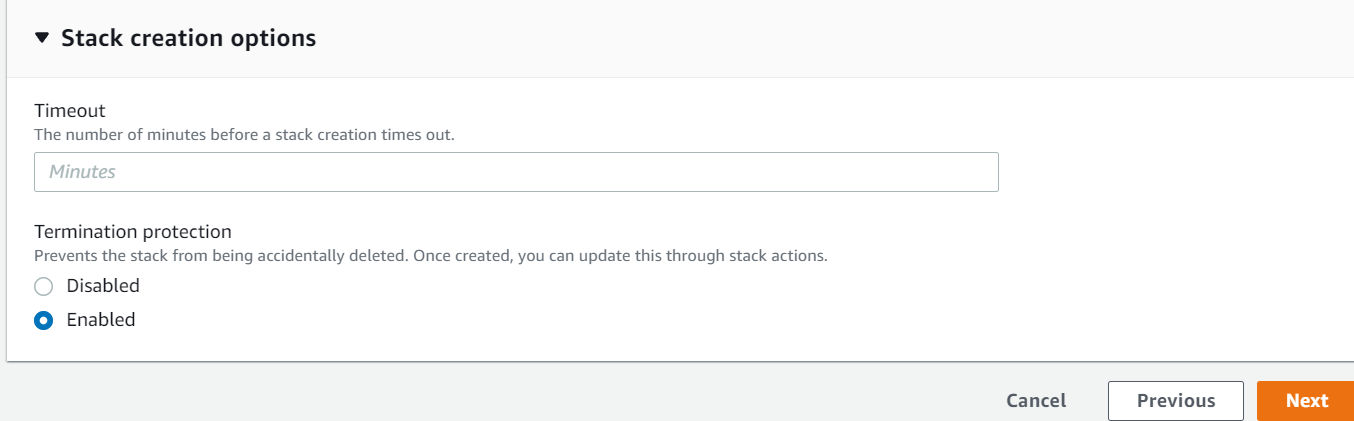
**Termination Protection:**

* Like EC2 we can also enable termination protection to prevent any accidental delete of the Stack resources.
* Applied to any Nested Stacks
* We can tighten the IAM policies (Only specific users can delete)

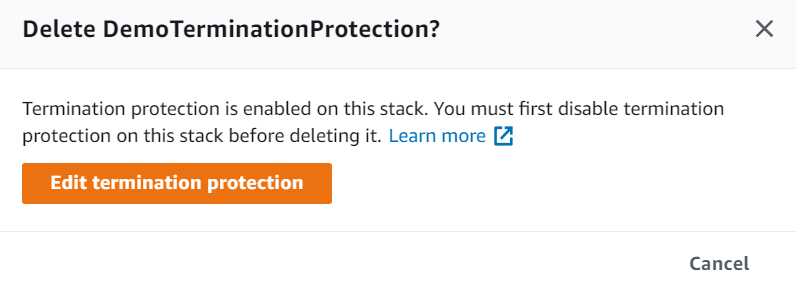
While creating the Stack



Then enable the termination protection



Now when you delete the Stack the extra message will pop up to disable the Termination



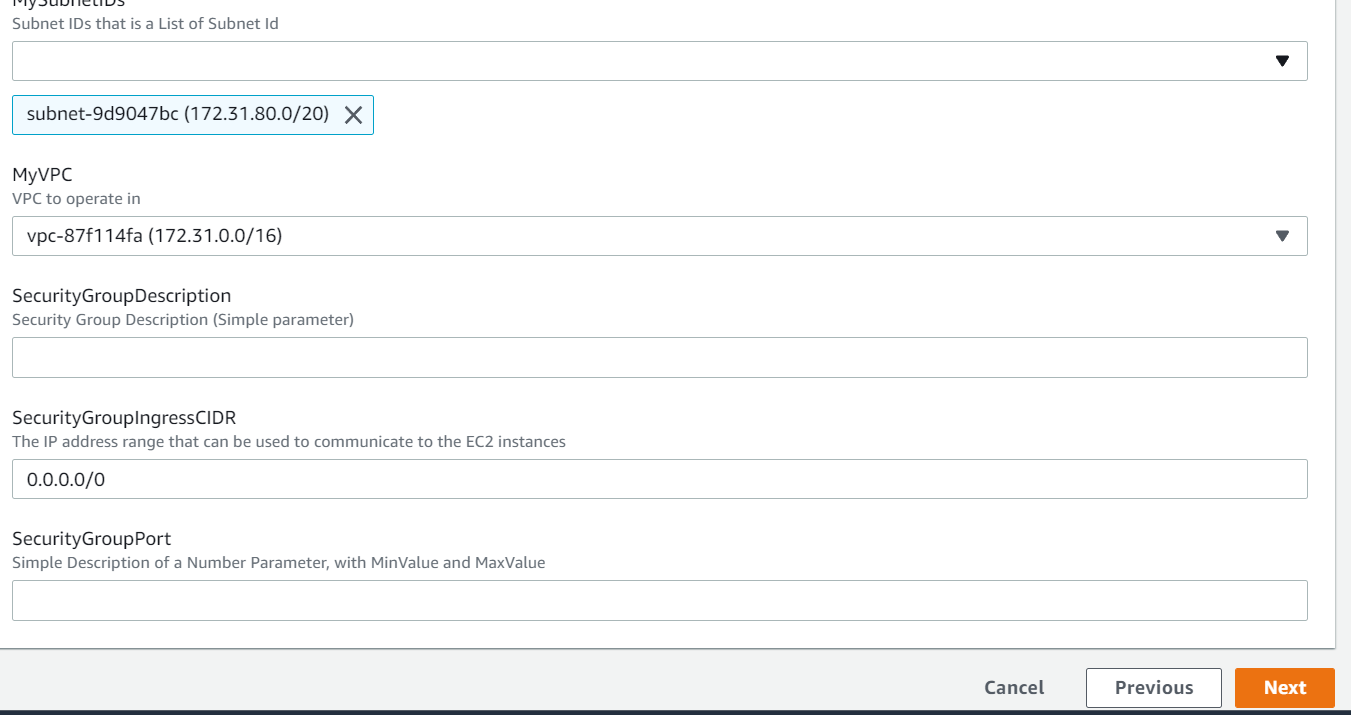
**Service Role:**

* IAM role that allows CloudFormation to create/update/delete stack resources on your behalf
* By default, CloudFormation uses a temporary session that it generates from your credentials.

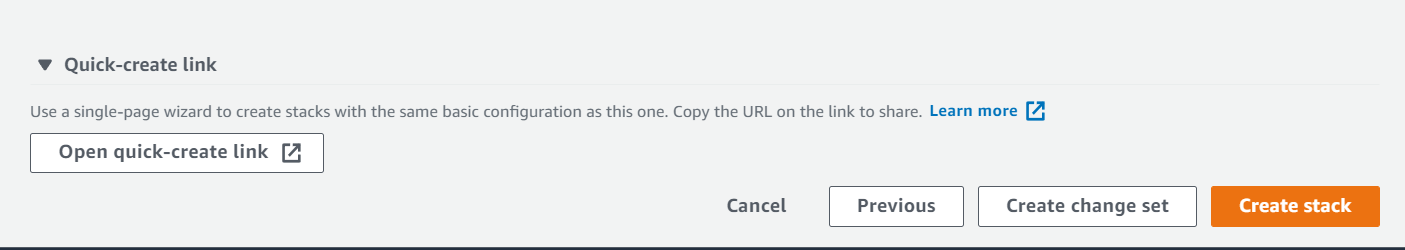
**Quick-Create Link for Stacks:**

A CloudFormation feature that allows you to create CloudFormation stacks quickly from AWS Console without pass through the wizard pages.

Upload the template and fill the parameters



UnderReview section in just before giving the create task



<https://console.aws.amazon.com/cloudformation/home?region=us-east-1#/stacks/quickcreate?templateUrl=https%3A%2F%2Fs3-external-1.amazonaws.com%2Fcf-templates-gn1d3ra8incb-us-east-1%2F2022007DM6-0-parameters-hands-on.yaml&stackName=Demo-Quick-Link&param_DbSubnetIpBlocks=10.0.48.0%2F24%2C%2010.0.112.0%2F24%2C%2010.0.176.0%2F24&param_InstanceType=t2.small&param_KeyName=myEC-keypair&param_MySubnetIDs%5B%5D=subnet-9d9047bc&param_MyVPC=vpc-87f114fa&param_SecurityGroupDescription=&param_SecurityGroupIngressCIDR=0.0.0.0%2F0&param_SecurityGroupPort=>

Give this link whoever you want to create the task.