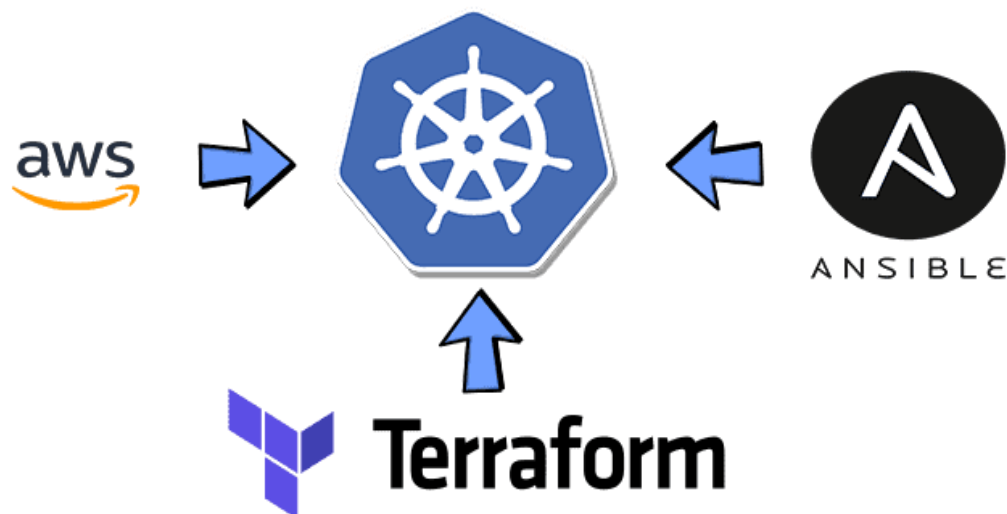


Lokeshkumar

Deploy a **Kubernetes** cluster using **Terraform** and **Ansible** on **AWS**: Use **EC2** or **EKS**

27



Lokeshkumar

While I was in the initial stages of learning Kubernetes, I always struggled to get a quick Kubernetes cluster to work on my practice labs. Though there are many online alternatives available which can be used in such scenarios but somehow I always felt if I could easily spin up a lab environment of my own, it will be good for my learning. That's when I came up with this process to spin up a basic Kubernetes cluster easily and quickly. The process described here can be a good learning to use Terraform and Ansible in such scenarios too. With modifications to the steps, this can also be used to launch actual clusters in projects.

For this post, I will go through the process of spinning up a basic Kubernetes cluster using Terraform on AWS. I will cover two ways to launch the cluster:

- Using Terraform to launch EC2 instances on AWS and use Ansible to bootstrap and start a Kubernetes cluster
- Using Terraform launch an AWS EKS cluster

The GitHub repo for this post can be found [Here](#)

Pre Requisites

Before I start the walkthrough, there are some pre-requisites which need to be met if you want to follow along or want to install your own cluster:

- Some Kubernetes, Terraform and Ansible knowledge. This will help understand the workings of the process

- Jenkins server to run Jenkins jobs
- An AWS account. The EKS cluster may incur some charges so make sure to monitor that
- Terraform installed
- Jenkins Knowledge

Apart from this I will explain all of the steps so it can be followed easily. I will cover each of the steps in detail so if you have some basic conceptual understanding of the above mentioned topics, you should be able to follow along.

Functional Flow

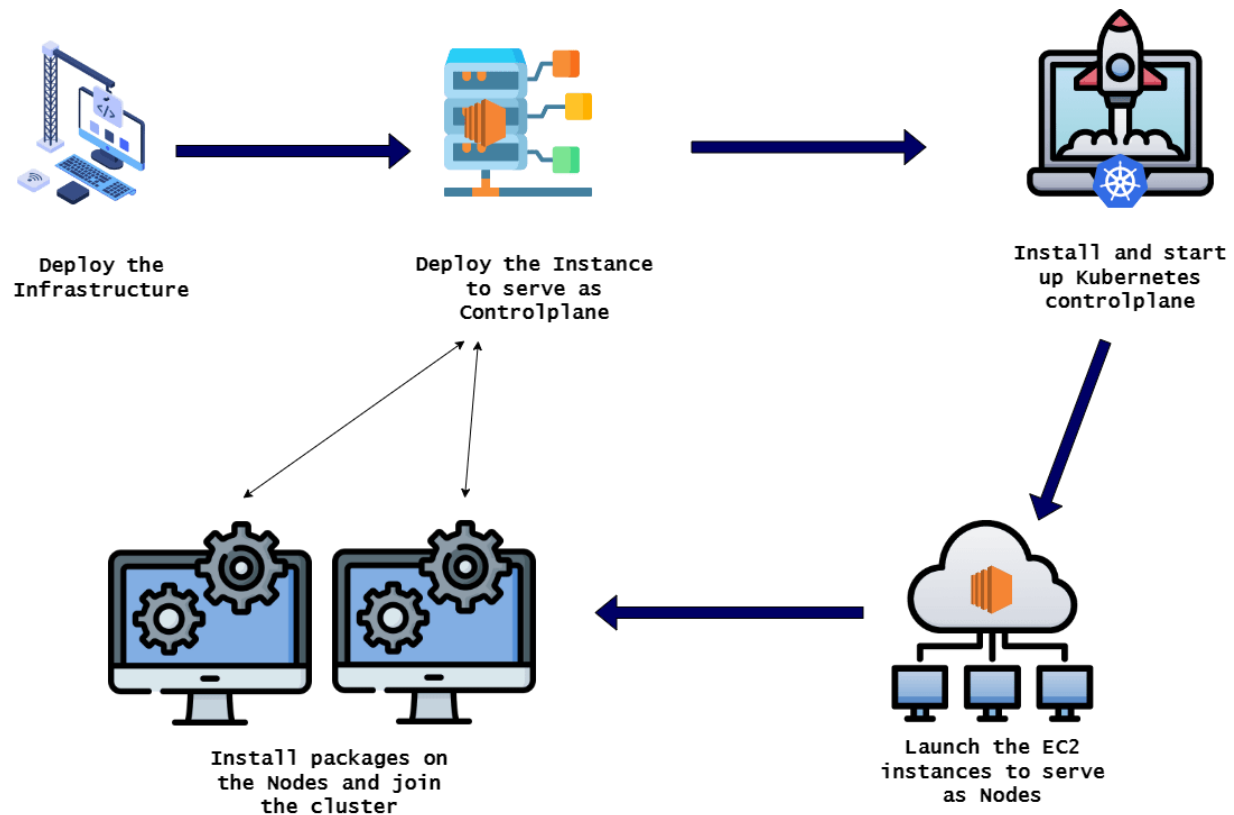
Let me first explain what's the process I will be following to deploy the cluster. I will go through two methods of deploying the cluster:

- *Using EC2:* In this method the cluster will be deployed using just EC2 instances on AWS. Both control plane and the nodes will be EC2 instances which you can directly access and control. The controlplane and nodes EC2 instances will be configured accordingly.
- *Using AWS EKS:* In this method the cluster will be deployed on the managed platform called EKS, which is provided by AWS. Since EKS is a managed cluster by AWS, we will be doing a declarative config here and deploy the cluster. To have a serverless approach, for the nodes I will be using Fargate instances. To know more about what is Fargate, click [Here](#).

Lets go through each of the methods and see how this will work.

Using EC2 Instances

First method we will go through is deploying the cluster using EC2 instances. In this we will deploy both the controlplane and some nodes on various EC2 instances. Below flow shows the overall process.



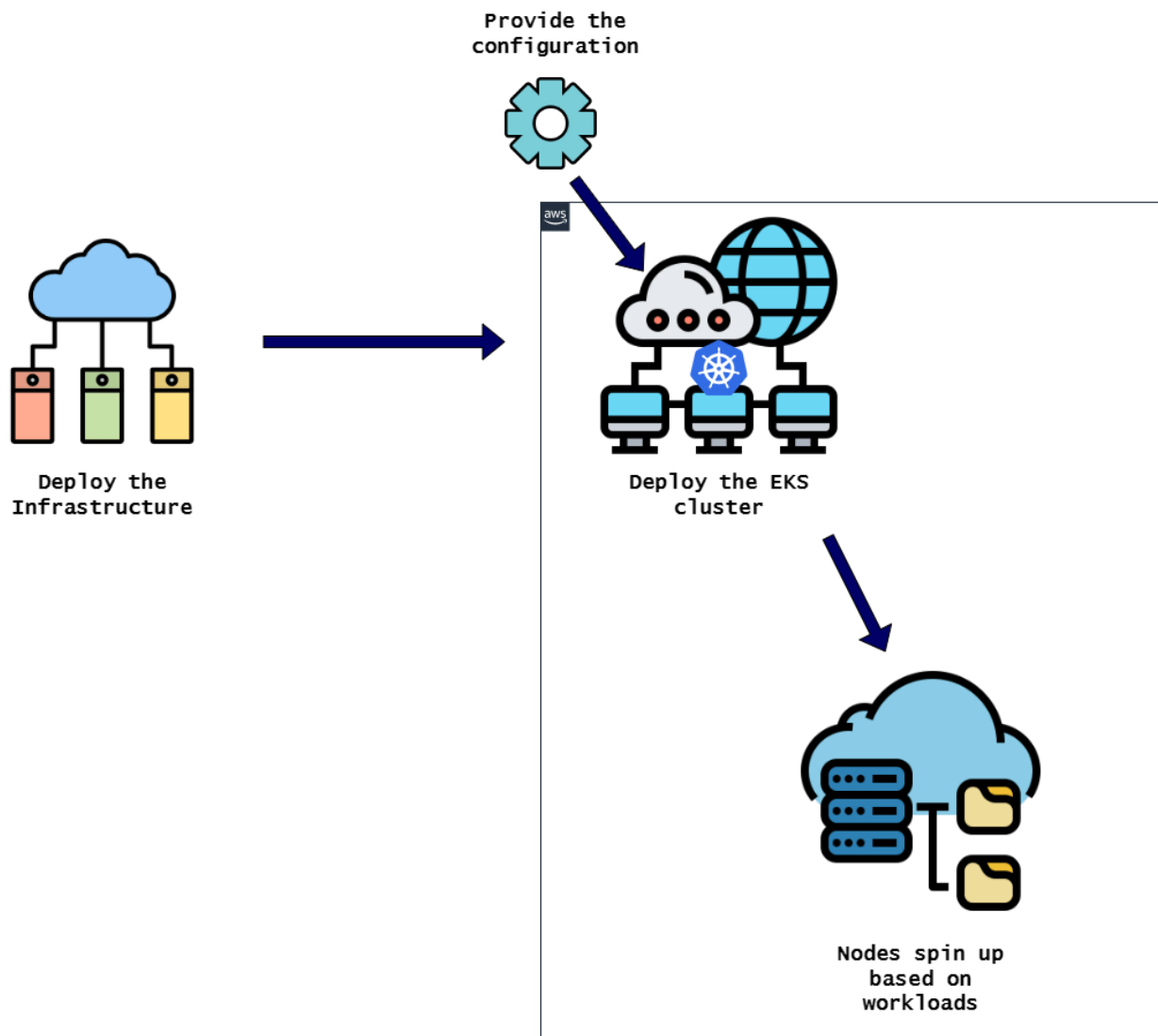
- **Deploy the Infrastructure:** This is the first step where the infrastructure to support the cluster will be deployed. The infrastructure will include different parts of the whole architecture which will be supporting the cluster running on the EC2 instances. Some of the items which will be deployed as part of the infrastructure are:
 - Networking components like VPC, Subnets etc.

- S3 buckets to store various temporary files needed during the deployment flow. Also to store the infrastructure state during the deployment
- **Deploy the Controlplane Instance:** In this stage the EC2 instance is deployed on the above infrastructure. In this process I am deploying one EC2 instance as the controlplane of the cluster. Normally Production grade Kubernetes cluster can have multiple controlplane nodes and add a scalability to it. But for this example, I am deploying one EC2 instance as the controlplane.
- **Bootstrap the controlplane:** The EC2 instance deployed above is needed to be prepared so Kubernetes cluster can start on it. In this stage multiple packages will be installed on the EC2 instance and the Kubernetes cluster will be started on the instance. At this point the cluster only comprises of the controlplane node.
- **Launch the EC2 instances for Nodes:** To make the Kubernetes cluster scalable, we will need more nodes added to it. In this step I am deploying some more EC2 instances which will become the child nodes in the Kubernetes cluster with the controlplane EC2 instance as the controller.
- **Prepare the Nodes and join cluster:** For the nodes to be part of the cluster, they need to be prepared. In this step the needed packages are installed on the Nodes and the Kubernetes cluster join command is executed on each of the node EC2 instance, so they join the cluster.

That completes the high level flow of deploying the cluster on EC2 instances.

Using AWS EKS

In this method, the cluster will be deployed on the AWS managed platform called EKS (Elastic Kubernetes Service). This platform takes away many of the steps to prepare the cluster and takes a more declarative approach. Below flow shows what I will be following in this process:



- **Deploy the Infrastructure:** In this step, the supporting infrastructure is deployed. Even though the cluster is AWS managed, the supporting infrastructure has to be deployed and can be controlled by us. As part of this step, these are the items which are deployed:
 - Networking components like VPC, subnets etc.
 - S3 buckets to store any temporary files needed during the deployment
 - Any IAM roles needed by the components
- **Deploy the EKS Cluster:** Once we have the infrastructure ready, we can go ahead and deploy the EKS cluster. Since the cluster is managed by AWS, we don't need to do any installations manually. We can pass the configuration needed for the cluster (like VPC, subnets etc.) and AWS handles spinning up the cluster
- **Spin up Nodes for the cluster:** With the EKS cluster, AWS spins up the controlplane for the cluster. For this example I am using Fargate instances as Nodes. So those nodes spin up automatically as workloads get deployed on the cluster. Based on the workloads more Fargate nodes spin up to handle the scalability. Apart from Fargate instances, even normal EC2 instances can be used as nodes and attach to the cluster. Even if using EC2, no need of manual installations. AWS EKS will handle spinning up the EC2 nodes and manage as part of the cluster.

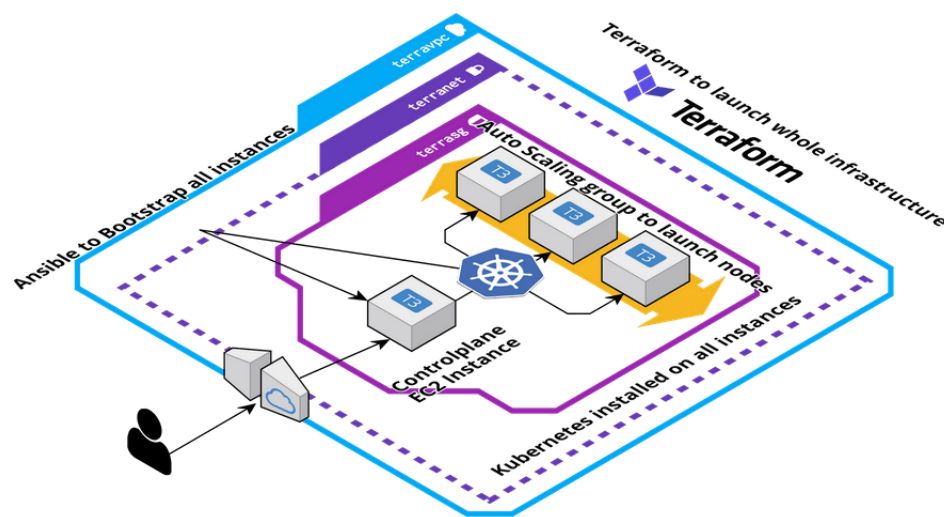
Now that we have some basic understanding of what I will be deploying, let's dive into some technical details of the parts.

Tech Details

Let me explain how each of the cluster will be composed using different AWS components. For both of the methods, I will be using Jenkins to automate the whole deployment. Here I am describing each of the cluster.

Cluster on EC2

This cluster comprises of EC2 instances which are deployed separately. Below image will show the overall cluster which will be built in this process:



- **Controlplane EC2 Instance:** One EC2 instance is deployed which will be serving as the controlplane node for this cluster. Since its the controlplane node, the size of the EC2 should be larger like t3. For the EC2 instance to act as the

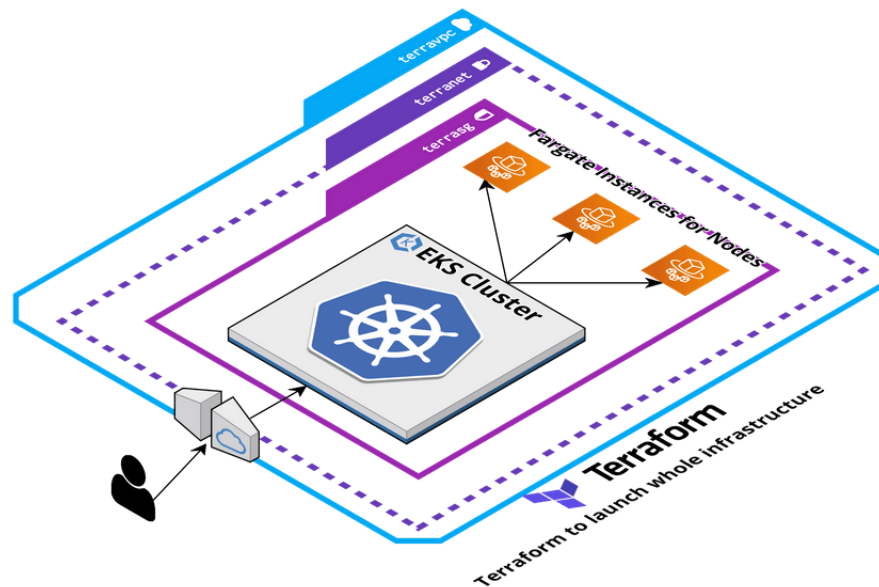
part of the process. These are the components which gets deployed as part of the networking for the whole cluster:

- VPC for the whole cluster
- Subnets for different instances across multiple AZs
- Security groups to control the traffic to these instances
- Internet Gateway for outside access to these instances
- **Use of Terraform:** All of the infrastructure which is mentioned here, is being deployed using Terraform. Different Terraform modules have been created to handle each of the components of the cluster. I will be going through the Terraform scripts in detail below.
- **Use of Ansible:** Ansible is used to handle of the package installations and steps needed on all of the instances. Since the installations and the cluster commands are automated, they are all ran by different Ansible playbooks. These playbooks are executed on the instances via SSH from the Jenkins pipeline. I will go through these playbooks in detail in below sections.

That should explain about the whole cluster architecture and it is deployed across multiple EC2 instances.

Cluster on EKS

In the next method, I am describing the cluster architecture which is deployed using AWS EKS. In this there are no direct involvement with launching EC2 instances and bootstrapping them. Below image describes the cluster architecture:



- **AWS EKS Cluster:** This is the central component of the cluster. An EKS cluster is deployed using a declarative configuration. Using Terraform the EKS cluster is deployed and AWS manages spinning up the cluster and operational. I will cover in detail how this is deployed.
- **Networking:** The networking part is the same as the above method with EC2 instances. Even for this EKS cluster we will need the networking components deployed separately, so the cluster can run in it. Same networking components are also deployed in this process
- **Fargate Instances:** For the nodes in the cluster, I am using Fargate instances. These are AWS managed instances which work as nodes in the cluster. To work with the autoscaling, these instances spin up automatically as workloads get deployed to the cluster. This is also a serverless node

added to the cluster. Will cover in detail later how the workloads get deployed to the Fargate instances.

- **Use of Terraform:** All of the infrastructure described above is deployed using Terraform. The declarative configuration for the cluster is specified in the Terraform script which deploys the whole cluster.

That explanation should give a good idea of both of the processes and what we will be setting up in this post. Lets walk through setting up each of the clusters.

Setup Walkthrough

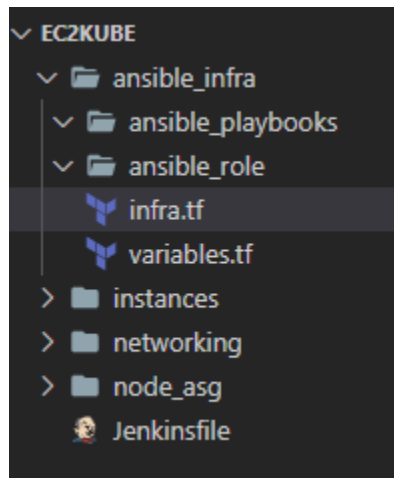
In this setup I will walk you through the setting up of the whole cluster via both of the processes. You can use any of these to spin up your own cluster. At high level this is the general flow which I will be setting up on Jenkins to launch each of the cluster.



Folder Structure

<https://www.linkedin.com/in/lokeshkumar-%E2%98%81%E2%8F%4a2860244>

- **Using EC2 Instances:** Below is the folder structure for this



ansible infra: This folder holds everything related to ansible used to bootstrap the instances in the cluster. It holds all the playbooks and the role to install and start Kubernetes on the controlplane and other nodes in the cluster. It also holds the Terraform script to launch the infrastructure needed to hold the the ansible playbooks and inventory files for the nodes to pull from. more on that later.

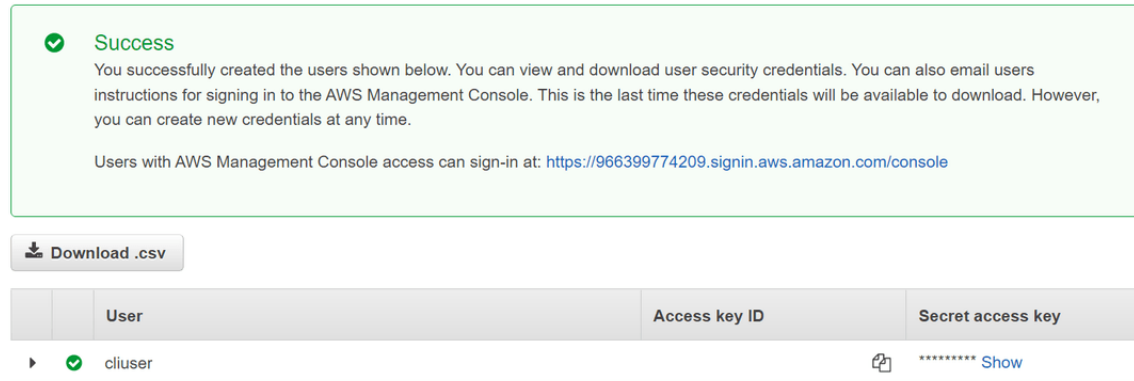
instances: This folder holds the Terraform scripts to launch the EC2 instances for the controlplane node.

networking: This folder holds the Terraform scripts to launch all the networking components of the cluster like VPC, subnets etc.

node asg: This folder contains all the Terraform scripts to launch the Auto scaling group that will launch all of the other nodes in the cluster.

Add user

1 2 3 4 5



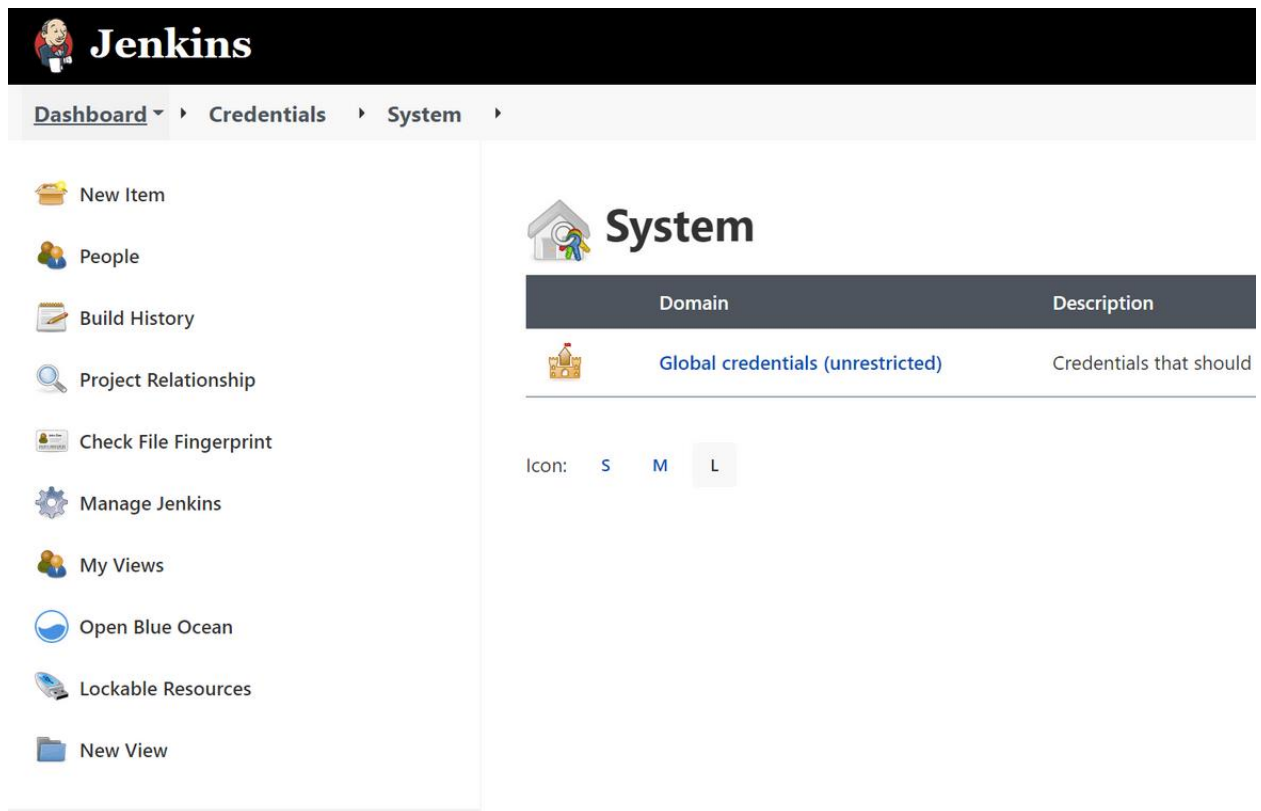
Once we have the IAM user lets move to setting up jenkins.

- **Install Jenkins:**

Of course the first step is to install Jenkins on a server or a system as suitable. I wont go through the steps to install Jenkins but you can follow the steps [Here](#).

- **Setup Credentials:**

For Jenkins to connect to AWS, it needs the AWS credentials. In this step we will setup the credentials in Jenkins. Login to Jenkins and Navigate to the Manage credentials page:



The screenshot shows the Jenkins web interface. The top navigation bar includes 'Dashboard', 'Credentials', and 'System'. The left sidebar contains various options like 'New Item', 'People', 'Build History', 'Project Relationship', 'Check File Fingerprint', 'Manage Jenkins', 'My Views', 'Open Blue Ocean', 'Lockable Resources', and 'New View'. The main content area is titled 'System' and displays a table of credentials. The table has two columns: 'Domain' and 'Description'. One entry is visible: 'Global credentials (unrestricted)' with the description 'Credentials that should'. Below the table, there are tabs for 'Icon', 'S', 'M', and 'L'.

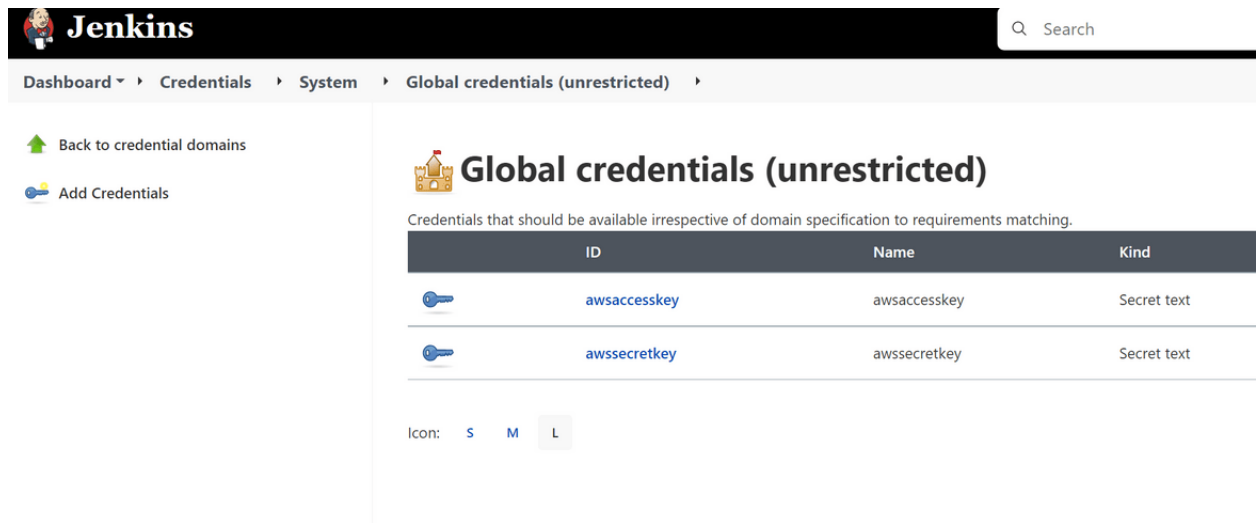
Domain	Description
Global credentials (unrestricted)	Credentials that should

On the credentials page, add a new Credential of kind Secret Text. Put the AWS key value in the Secret content field and provide a name for the credential on the ID field. This is the name which will be referred in the Jenkinsfile to access the credential.

Lokeshkumar

Add two such credentials for each of the AWS key from the IAM user we created earlier. There should be two credentials now. If you added on a different name, update accordingly on the Jenkinsfile too. This is what I have used for my Jenkins pipeline:

Lokeshkumar



Jenkins Search



Dashboard ▾ ▸ Credentials ▸ System ▸ Global credentials (unrestricted) ▸

🏠 Back to credential domains

🔑 Add Credentials

Global credentials (unrestricted)

Credentials that should be available irrespective of domain specification to requirements matching.

ID	Name	Kind
 awsaccesskey	awsaccesskey	Secret text
 awssecretkey	awssecretkey	Secret text

Icon: S M L


These two credentials will be used by the pipeline to communicate with AWS.

- **Setup Pipeline:**


Now we will setup the actual pipeline which will be running and performing the deployment. As I stated above, in this example I have kept the EKS related and EC2 cluster related files in two separate branches. So I will be creating a multi branch pipeline to scan both branches and run accordingly. Login to Jenkins and create a new Multibranch pipeline from the create option

Dashboard

All




This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.




Multi-configuration project

Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.




MultiJob Project

MultiJob Project, suitable for running other jobs




Folder

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.



Multibranch Pipeline

Creates a set of Pipeline projects according to detected branches in one SCM repository.



Organization Folder

Creates a set of multibranch project subfolders by scanning for repositories.

If you want to create a new item from other existing, you can use this option:

On the options page, select the GIT repo source which will be the source for the scripts and the two branches. To be able to connect the repo, the credentials will also have to be specified on the same option. If its a public repo no credentials needed. I would suggest clone my repo and use your own repo for the pipeline.

Dashboard ▾

TerraKube_Pipeline

General

Branch Sources

Build Configuration

Scan Multibranch Pipeline Triggers

Orphaned Item Strategy

Appearance

Health metrics

Properties

Pipeline Libraries

Branch Sources

Git

Project Repository ?

Credentials ?

- none - ▾

Add ▾

Behaviors

Discover branches ?

Add ▾

Property strategy

All branches get the same properties ▾

Add property ▾

Dashboard › TerraKube_Pipeline ›

Save the pipeline. Once you save, Jenkins will start scanning for branches in the repo. At this point it will fail since you won't have any content in the pipeline. If you have created the branches already, you should have the multiple branches

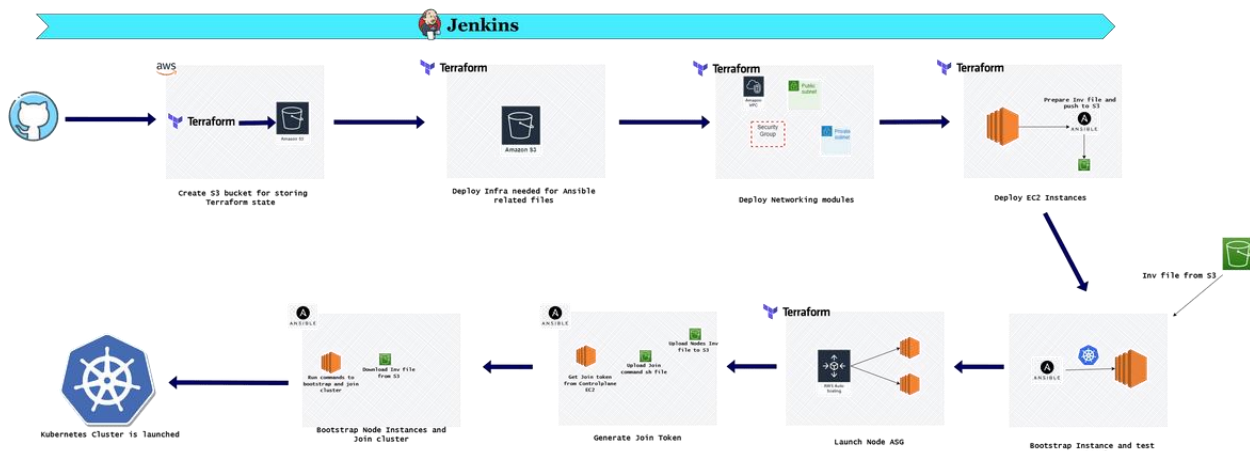
showing up as separate runnable pipeline on Jenkins

Jenkins						
TerraKube_Pipeline ☆ ⚙			Pipelines Administration ↗ Logout			
			Activity Branches Pull Requests			
HEALTH	STATUS	BRANCH	COMMIT	LATEST MESSAGE	COMPLETED	
🌟	✅	ekskube	—	a	2 commits	a month ago ☆
☁️	❌	ec2kube	—	Branch indexing		a month ago ☆

That should complete the initial Jenkins setup and you are ready to run the pipelines for deployment. Next lets move on to understanding the two pipelines and run them.

Deploy EC2 Cluster

Lets first go through deploying the Kubernetes cluster using EC2 instances. Below pipeline shows the whole deployment steps which are carried out by Jenkins:



Lets understand each of the stages.

- **Environment Variables:** First step is to set the needed environment variables which are used by different steps in

the pipeline. Here are the environment variables which are being set:

```
environment{
    AWS_ACCESS_KEY_ID=<Jenkins_creds_for_access_key>
    AWS_SECRET_ACCESS_KEY=<Jenkins_creds_for_secret_key>
    AWS_DEFAULT_REGION="us-east-1"
    SKIP="N" # control which stage to skip if needed
    TERRADESTROY="Y" # if the destroy stage need to run
    FIRST_DEPLOY="Y"
    STATE_BUCKET="<bucket_name>" #bucket name for Terraform state
    ANSIBLE_BUCKET_NAME="<bucket_name>" # bucket name for ansible files
}
```

- **Create S3 Buckets for State:** In this stage, the pipeline creates the S3 buckets which are needed for storing the Terraform state. The bucket name is mentioned on the env variables and same name gets specified in the Terraform scripts so the state can be managed in this bucket.


```
stage("Create Terraform State Buckets"){
    when{
        environment name:'FIRST_DEPLOY',value:'Y'
        environment name:'TERRADESTROY',value:'N'
        environment name:'SKIP',value:'N'
    }
    steps{
        sh'''
        aws s3 mb s3://<bucket_name>'''
    }
}
```

- **Deploy Infra for Ansible:** In this stage, S3 buckets are created which will be used by Ansible to store the updated Inventory files and read the Inventory files to run commands. Terraform is used to create the buckets. The bucket name is passed via the variable file.

```
terraform {
  required_providers {
    aws = {
      source = "hashicorp/aws"
      version = "~> 3.74.1"
    }
  }
  backend "s3" {
    bucket = "<bucket_name>"
    key    = "<file_name>"
    region = "us-east-1"
  }
}

provider "aws" {
  region = "us-east-1"
}

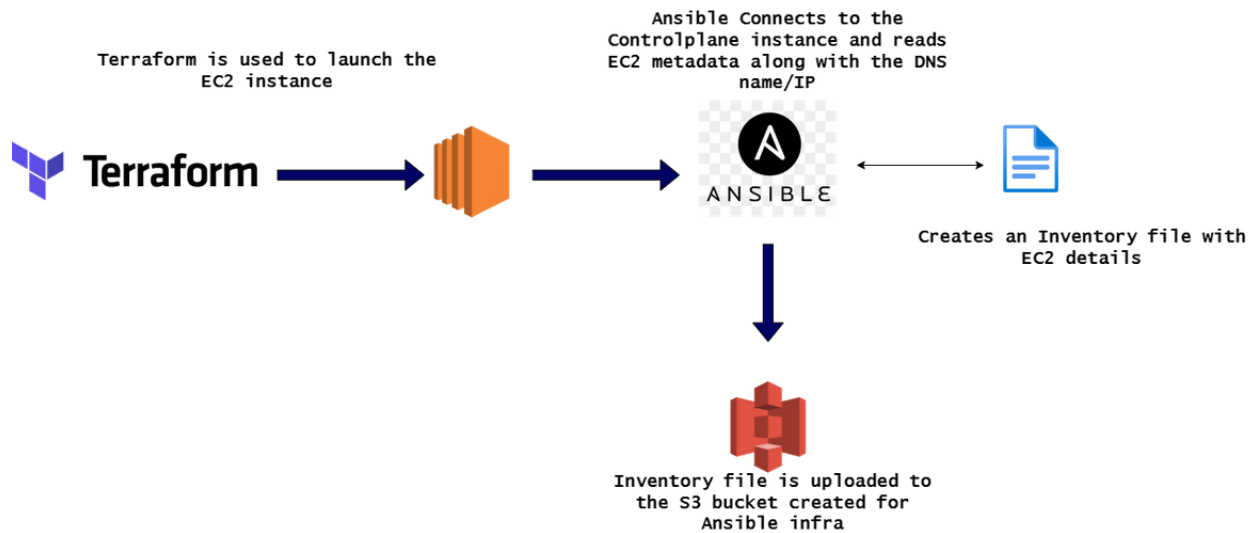
module "s3-bucket" {
  source      = "terraform-aws-modules/s3-bucket/aws"
  version     = "2.6.0"
  bucket     = var.ansible_bucket

  block_public_acls      = true
  block_public_policy    = true
  ignore_public_acls     = true
  restrict_public_buckets = true
  force_destroy          = true
}
```



```
stage("Deploy Networking"){
    when{
        environment name:'TERRADESTROY',value:'N'
        environment name:'SKIP',value:'N'
    }
    stages{
        stage('Validate n/w Infra'){
            steps{
                sh '''
                cd networking
                terraform init
                terraform validate'''
            }
        }
        stage('Deploy n/w Infra'){
            steps{
                sh '''
                cd networking
                terraform plan -out outfile
                terraform apply outfile'''
            }
        }
    }
}
```

- **Deploy EC2 Instances:** In this stage the EC2 instance for controlplane get created. The EC2 instance is defined in a Terraform module which gets deployed in this stage. But its not just deploying the instance, in this stage, the instance details also gets updated to an inventory file. This is what happens in this stage.



An Ansible playbook is created to perform the Inventory file update and upload to S3. The playbook can be found in the repo I shared.

```
- hosts: localhost
gather_facts: yes
become: true
connection: local
tasks:
  - name: get ec2
    ec2_instance_facts:
      aws_access_key: "{{ lookup('env', 'AWS_ACCESS_KEY_ID') }}"
      aws_secret_key: "{{ lookup('env', 'AWS_SECRET_ACCESS_KEY') }}"
      ec2_region: "us-east-1"
      filters:
        tag:Name: KubeCtrlPlane
    register: ec2_facts
  - debug:
      var: ec2_facts.instances[0].public_dns_name
      # ec2_facts.instances[0].instance_id
  - name: add ec2 ip to inv file
    lineinfile:
      path: ./inv
      line: "{{ item.public_dns_name }}"
      loop: "{{ ec2_facts.instances }}"
```

- **Bootstrap Controlplane Instance:** In this stage the controlplane EC2 which was launched earlier, gets bootstrapped where all the needed packages and Kubernetes gets installed on the instance. An ansible role has been created to perform the bootstrap steps. These are the steps which happen in this stage:
 - The inventory file which was uploaded earlier, is pulled from the S3 bucket in the local folder within pipeline
 - The ansible role is executed which installs all the necessary packages and starts the Kubernetes service on the controlplane node

```
---
# tasks file for kubecontrolplane
- name: Run the equivalent of "apt-get update" as a separate step
  apt:
    update_cache: yes
- name: Install utils
  apt:
    name: "{{ item }}"
    state: present
  loop:
    - apt-transport-https
    - software-properties-common
    - curl
- include_tasks: install_docker.yml
- include_tasks: install_kubernetes.yml
```

```
✓ cd ansible_infra cd ansible_role aws s3 cp s3://ansibleinfra1/inv inv ls -la pwd ansible-playbook main.yml -i inv -- Shell Script
55 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : Run the equivalent of "apt-get update" as a separate step] ***
56 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
57
58 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : Install kube and kubectl] ***
59 changed: [ec2-54-234-159-234.compute-1.amazonaws.com] => (item=kubelet)
60 changed: [ec2-54-234-159-234.compute-1.amazonaws.com] => (item=kubeade)
61 ok: [ec2-54-234-159-234.compute-1.amazonaws.com] => (item=kubectl)
62
63 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : swapoff for kubernetess] ***
64 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
65
66 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : update docker file system] ***
67 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
68
69 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : restart docker] ***
70 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
71
72 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : remove containerd config] ***
73 [WARNING]: consider using the file module with state=absent rather than running
74 'rm'. If you need to use command because file is insufficient you can add
75 'warn: false' to this command task or set 'command_warnings=False' in
76 ansible.cfg to get rid of this message.
77 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
78
79 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : restart containerd] ***
80 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
81
82 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : run kubeadm init] ***
83 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
84
85 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : save kubeconfig file] ***
86 [WARNING]: consider using the file module with state=directory rather than
87 running 'mkdir'. If you need to use command because file is insufficient you
88 can add 'warn: false' to this command task or set 'command_warnings=False' in
89 ansible.cfg to get rid of this message.
90 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
91
92 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : copy kubeconfig file] ***
93 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
94
95 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : change permission for kubeconfig file] ***
96 [WARNING]: The value 0 (type int) in a string field was converted to '0' (type
97 string). If this does not look like what you expect, quote the entire value to
98 ensure it does not change.
99 ok: [ec2-54-234-159-234.compute-1.amazonaws.com]
100
101 TASK [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : install network plugin] ***
102 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
103
104 RUNNING HANDLER [/var/lib/jenkins/workspace/TerraKubeDeploy_ec2kube/ansible_infra/ansible_role/kubecontrolplane : docker service] ***
105 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
106
107 PLAY RECAP *****
108 ec2-54-234-159-234.compute-1.amazonaws.com : ok=23 changed=19 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

After the Kubernetes service starts, there is also a test step which runs in this stage to ensure the Kubernetes starts properly on the controlplane EC2. The kubectl test step is also performed using Ansible. During the test step, same Inventory file is also downloaded from the S3 bucket, and used by Ansible to connect to the EC2.

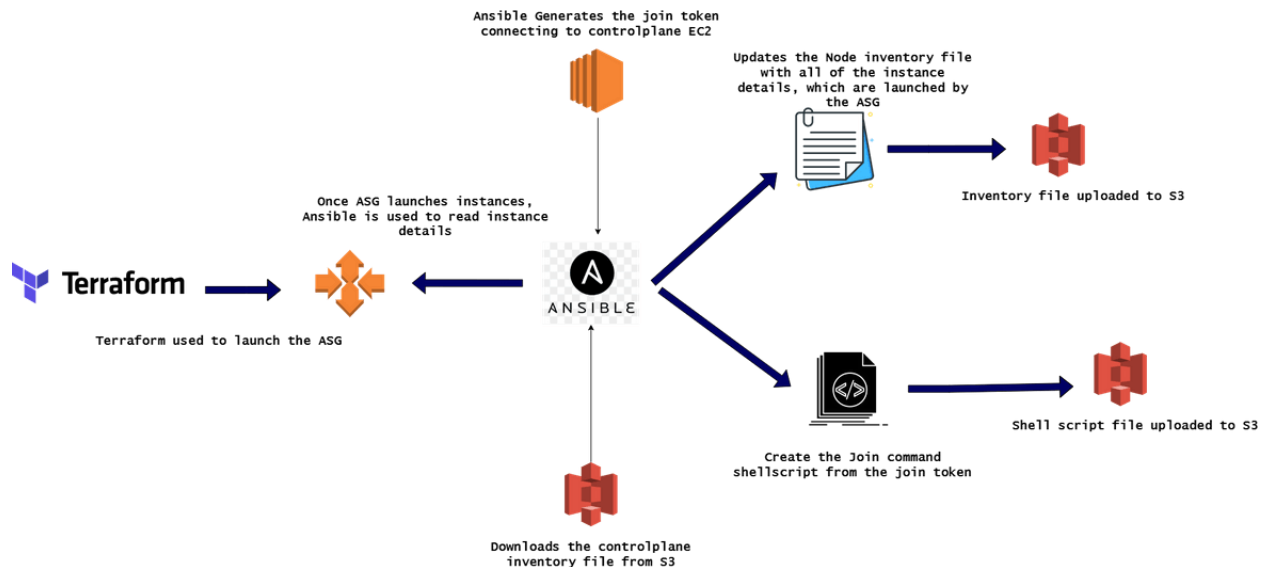
Lokeshkumar



test kubectl - 4s

```
✓ cd ansible_infra cd ansible_playbooks aws s3 cp s3://ansibleinfra1/inv inv ansible-playbook testkubectl.yml -i inv -- Shell Script
1 + cd ansible_infra
2 + cd ansible_playbooks
3 + aws s3 cp s3://ansibleinfra1/inv inv
4 Completed 1.2 KiB/1.2 KiB (19.7 KiB/s) with 1 file(s) remainingdownload: s3://ansibleinfra1/inv to ./inv
5 + ansible-playbook testkubectl.yml -i inv
6
7 PLAY [kubectrl] *****
8
9 TASK [Gathering Facts] *****
10 ok: [ec2-54-234-159-234.compute-1.amazonaws.com]
11
12 TASK [test kubectl from controlplane] *****
13 changed: [ec2-54-234-159-234.compute-1.amazonaws.com]
14
15 TASK [debug] *****
16 ok: [ec2-54-234-159-234.compute-1.amazonaws.com] => {
17   "kubectl_output.stdout_lines": [
18     "NAME          STATUS    ROLES    AGE   VERSION",
19     "ip-10-0-1-166  NotReady control-plane 11s   v1.24.3"
20   ]
21 }
22
23 PLAY RECAP *****
24 ec2-54-234-159-234.compute-1.amazonaws.com : ok=3  changed=1  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

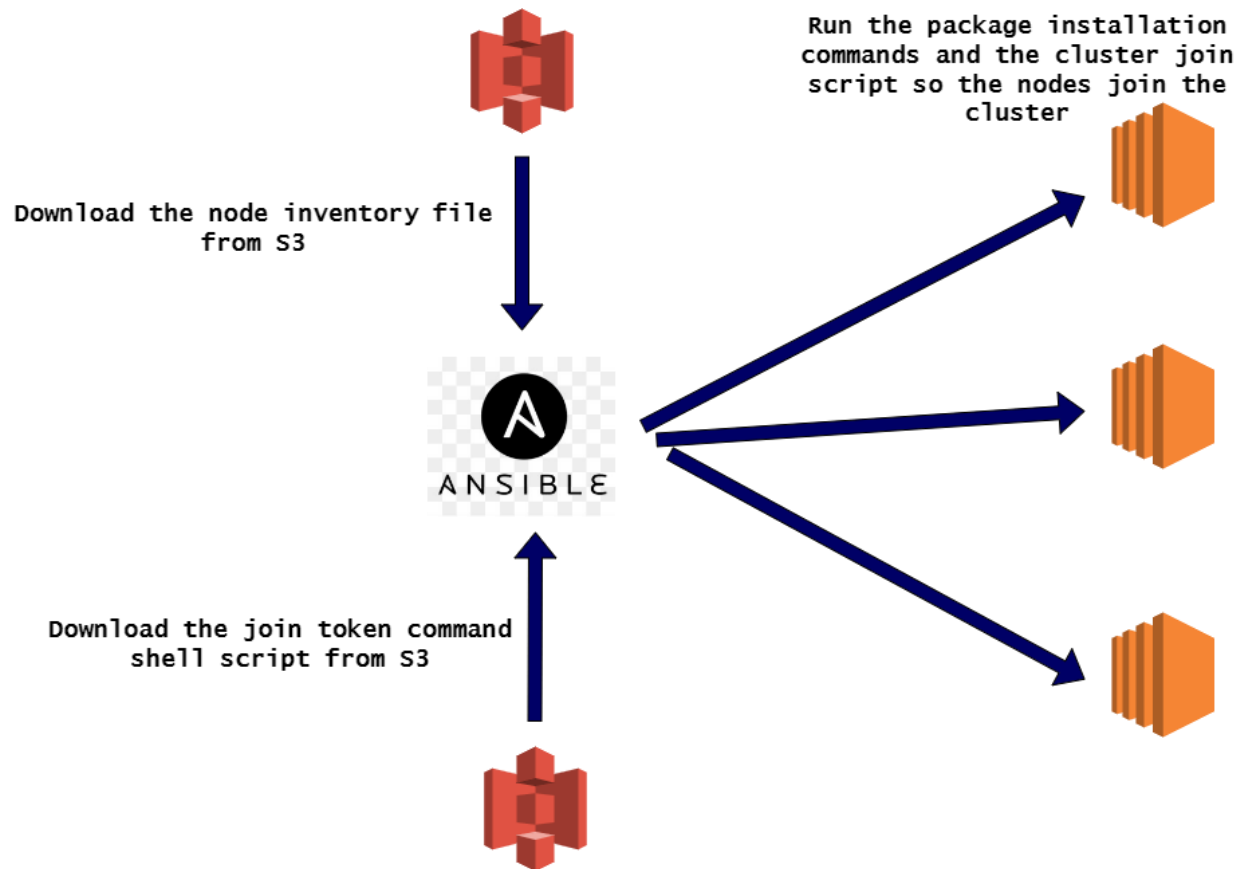
- **Launch Auto Scaling Group for Nodes:** Now that we have the controlplane up and running, in this stage we will launch the instances which will join as nodes to the cluster. There are few things happening in this stage apart from launching the ASG. This should explain the steps:



An Ansible playbook has been created to get the launched instance details and update in an inventory file. That inventory file gets uploaded to a S3 bucket for further steps.

```
- hosts: localhost
gather_facts: yes
become: true
connection: local
tasks:
  - name: get ec2 node
    ec2_instance_facts:
      aws_access_key: "{{ lookup('env', 'AWS_ACCESS_KEY_ID') }}"
      aws_secret_key: "{{ lookup('env', 'AWS_SECRET_ACCESS_KEY') }}"
      ec2_region: "us-east-1"
      filters:
        tag:instanceid: node
    register: ec2_facts
  - debug:
      var: ec2_facts.instances[0].public_dns_name
  - name: add ec2 ip to inv file
    lineinfile:
      path: ./nodeinv
      line: "{{ item.public_dns_name }}"
      loop: "{{ ec2_facts.instances }}"
```

- **Bootstrap Node Instances:** In this stage, the node instances which were launched on last step, get bootstrapped. All the needed packages are installed on these nodes and the Kubeadm join commands are executed on these nodes so these join the cluster. This is the flow which happens in this step:



To bootstrap the nodes, another Ansible playbook has been created which runs the commands. Once the nodes successfully join the cluster, finally a test `kubectl` command is executed using an Ansible playbook. This `kubectl` will test that the nodes have successfully joined the cluster. You can check all the nodes and the controlplane running on the console

That completes the deployment of the cluster using EC2 instances. Once these steps complete, you should have a working cluster using the EC2 instances. Once all the changes are done push the changes to the Git repo which is the source for the pipeline.

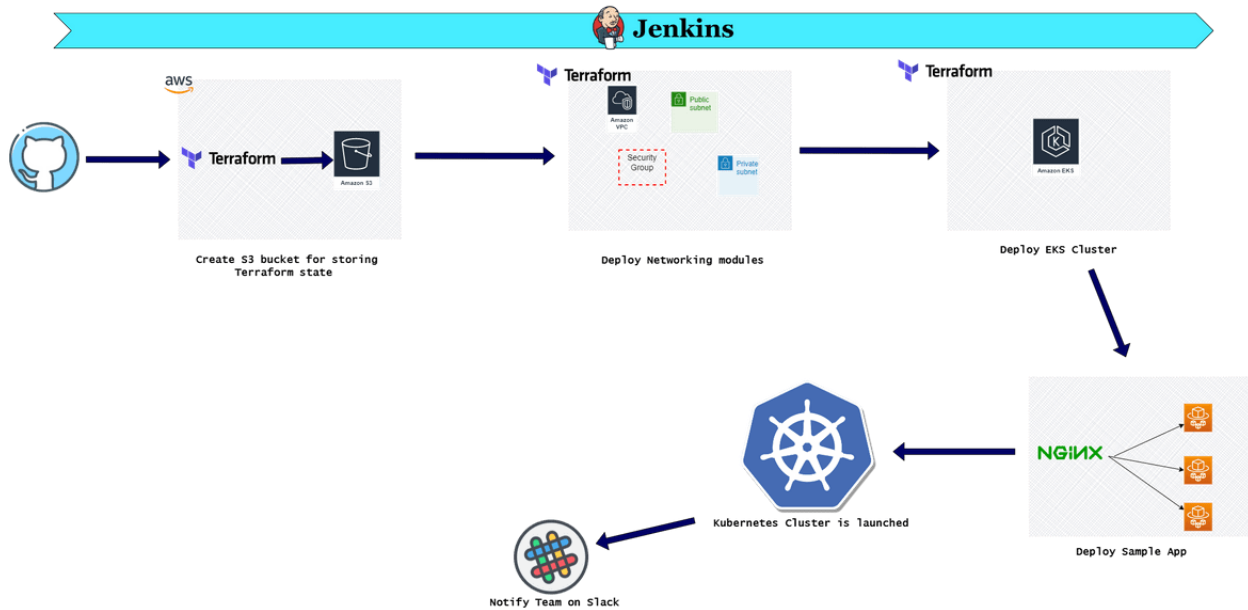
To execute the steps, run the pipeline which was created. Once the pipeline finishes, the cluster will be up and running.

Lokeshkumar

Note: If you are running through spinning up a cluster just for learning, make sure to destroy the resources which were created or you will incur charges on AWS. Just run the destroy phase of the pipeline in my repo by setting the environment variable. It will destroy all of the resources which were created by the pipeline.

Now lets move focus to the other type of process. In this process the cluster will be deployed on AWS EKS. This deployment is also orchestrated using Jenkins. Below is the overall pipeline:

Lokeshkumar



Lets go through each of the stages:

- **Environment Variables:** First step is to set the needed environment variables which are used by different steps in the pipeline. Here are the environment variables which are being set:


```
environment{
  AWS_ACCESS_KEY_ID=<jenkins_creds_for_key>
  AWS_SECRET_ACCESS_KEY=<jenkins_creds_for_access_key>
  AWS_DEFAULT_REGION="us-east-1"
  SKIP="N" # Control if a step need to be skipped
  TERRADESTROY="Y" # control if destroy stage need to run
  FIRST_DEPLOY="Y"
  STATE_BUCKET="<busket_name>" # S3 bucket for Terraform state
  CLUSTER_NAME="<cluster_name>" # EKS cluster name
}
```

- **Create S3 bucket for Terraform state:** This stage is identical as the one from the EC2 process. In this stage the S3 bucket is created which will house the Terraform state file.
- **Deploy Networking modules:** This stage is also identical to the EC2 process. All of the networking components are deployed in this stage. Terraform is used for this stage and it follows the same steps from the EC2 process.
- **Deploy EKS Cluster:** In this stage the EKS cluster is launched. Since its a cluster managed by AWS, we just need to declaratively launch the cluster and AWS will handle the rest. A Terraform module is created to deploy the cluster. In the Terraform script all of the parameters for the cluster is specified. As part of the cluster deployed, the node groups are also deployed where the workloads will run. I am using Fargate instance node groups which will spin up as and when workloads starts running on the cluster. In the Terraform module, a Fargate profile is defined which gets deployed


```
fargate_profiles = {
  default = {
    name = "default"
    selectors = [
      {
        namespace = "apps"
        labels = {
          Application = "app"
        }
      },
      {
        namespace = "default"
        labels = {
          WorkerType = "fargate"
        }
      },
      {
        namespace = "monitoring"
        labels = {
          WorkerType = "fargate"
        }
      },
      {
        namespace = "kube-system"
        labels = {
          k8s-app = "kube-dns"
        }
      }
    ]
  }

  tags = {
    Owner = "default"
  }

  timeouts = {
    create = "20m"
    delete = "20m"
  }
}

secondary = {
  name = "secondary"
  selectors = [
    {
      namespace = "default"
      labels = {
        Environment = "dev"
      }
    }
  ]

  subnet_ids = [data.terraform_remote_state.network_state.outputs.cluster_private_subnets[1]]

  tags = {
    Owner = "secondary"
  }
}
```

Once the cluster deploys successfully, you can view the cluster on console

The screenshot shows the AWS Management Console interface for the Amazon Elastic Kubernetes Service (EKS). The left sidebar displays the 'Amazon Elastic Kubernetes Service' logo and a list of related services including Amazon ECR, Documentation, and Submit feedback. The main content area shows the 'terra-kube-cluster' page with a navigation bar at the top containing 'Overview', 'Resources', 'Compute', 'Networking', 'Add-ons', 'Authentication', 'Logging', 'Update history', and 'Tags'. The 'Compute' tab is selected, displaying a table of nodes and node groups. A notification at the top indicates that a new Kubernetes version is available. The 'Nodes (1)' section shows a single node with the name 'ip-10-0-2-175.ec2.internal', instance type 't2.small', and node group 'grp1-20220723185159776900000015'. The 'Node groups (1)' section shows a single node group with the name 'grp1-20220723185159776900000015', desired size of 1, AMI release version '1.21.12-20220629', and launch template 'grp1-20220723185159461700000013 (1)'. A notification at the bottom indicates that AWS Fargate is migrating service quotas from the current Amazon EKS pod count-based quotas to vCPU-based quotas.

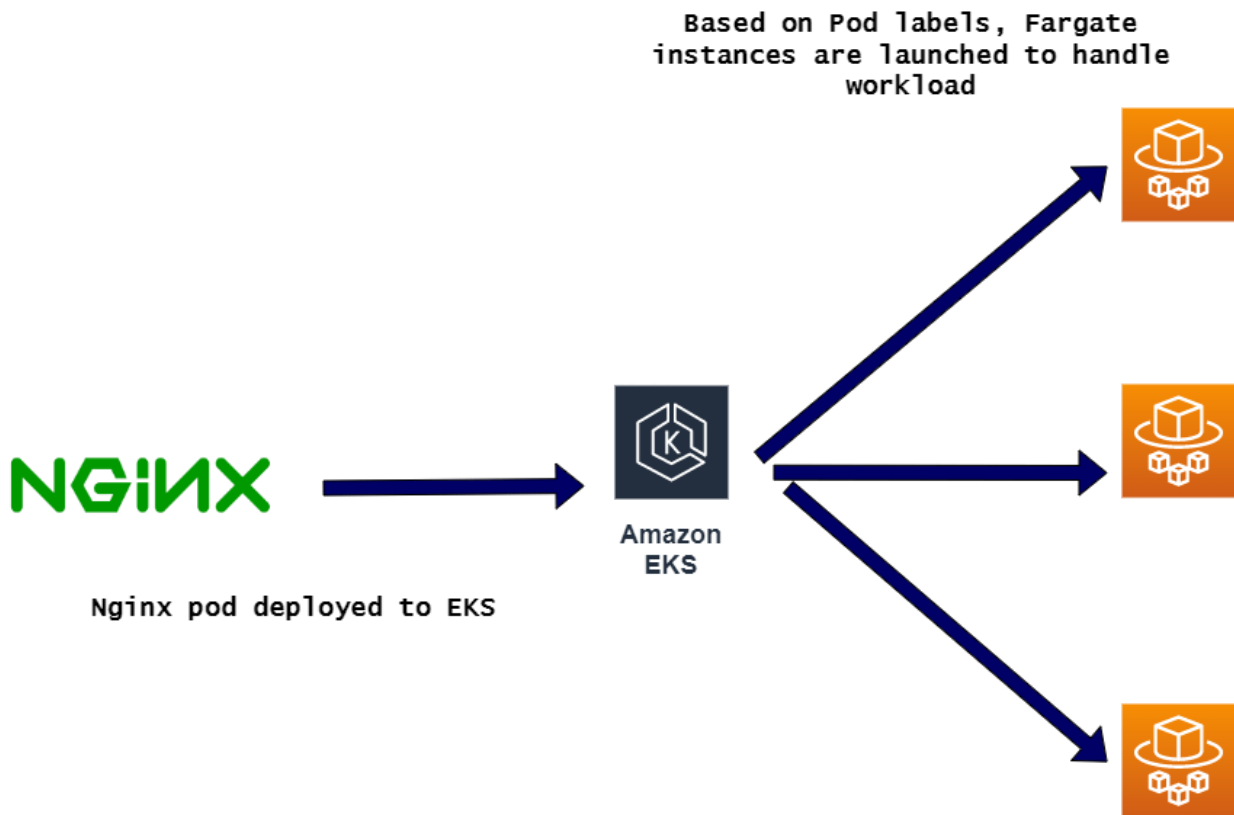
Node name	Instance type	Node group	Created
ip-10-0-2-175.ec2.internal	t2.small	grp1-20220723185159776900000015	Created 22 minutes ago

Group name	Desired size	AMI release version	Launch template
grp1-20220723185159776900000015	1	1.21.12-20220629	grp1-20220723185159461700000013 (1)

One thing to note here is that I am also launching a non-Fargate node group so that the coredns pods can be scheduled. To run coredns pods on Fargate instances is possible but I will not cover that in this post.

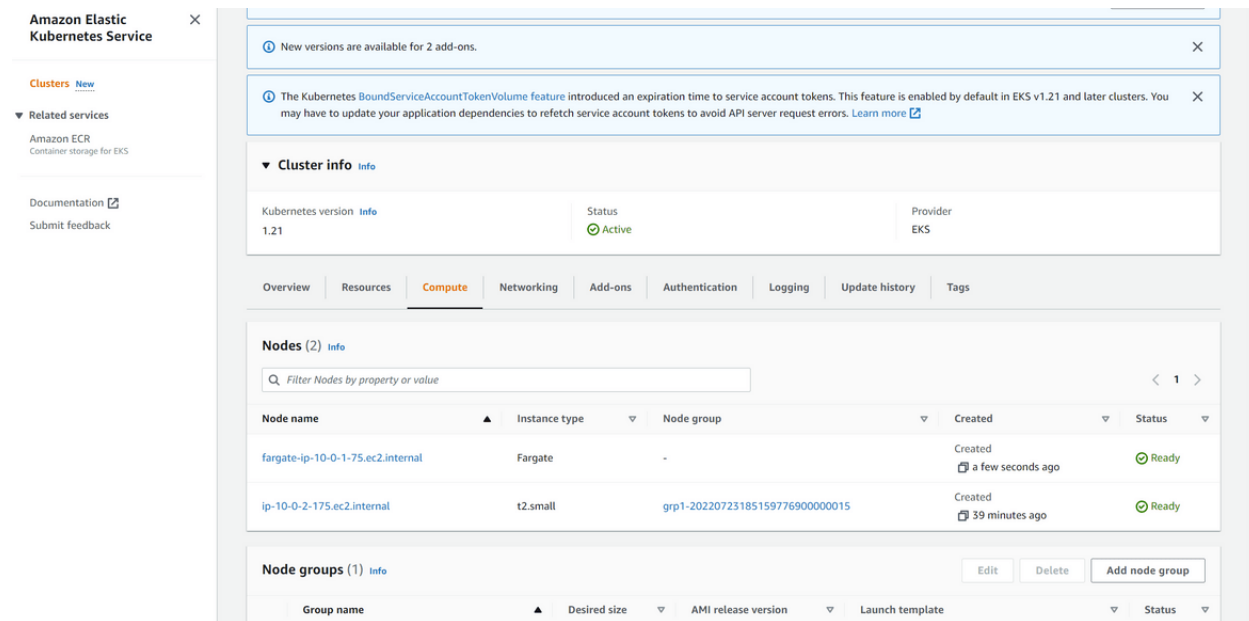
- **Deploy Sample App:** Since we now have the cluster up and running, to see how the Fargate instances spin up, we will deploy a sample app in this stage. It will be a simple pod with the nginx image which will be deployed to the cluster. The pod will have the specific labels that will trigger creation of the Fargate instance nodes.

Lokeshkumar



Once the pods start running, the Fargate instances can be seen on the AWS console for EKS:

Lokeshkumar



To test the nodes, run this command from your local terminal.
Make sure to configure AWS CLI so it authenticates to correct AWS instance

```
aws eks update-kubeconfig --name cluster_name
kubectl get nodes
λ kubectl get nodes
NAME                                STATUS    ROLES    AGE    VERSION
fargate-ip-10-0-1-75.ec2.internal    Ready    <none>    2m42s  v1.21.9-eks-14c7a48
ip-10-0-2-175.ec2.internal           Ready    <none>    41m    v1.21.12-eks-5308cf7
```

The sample pod should also be running now on the Fargate instance node:

```
λ kubectl get pods
NAME    READY    STATUS    RESTARTS    AGE
mypod   1/1      Running   0            6m21s
```

As last step of this stage, a test kubectl command is executed to ensure the cluster is functioning properly and the nodes are up and running too.

