Czech University of Life Sciences Prague

Faculty of Economics and Management

Department of information technology

Bachelor Thesis

Linux server system setup and configuration using terraform and ansible

by

Tinashe Bvukumbwe

CZECH UNIVERSITY OF LIFE SCIENCES PRAGUE

Faculty of Economics and Management

BACHELOR THESIS TOPIC

Author of thesis:	akad. Tinashe Bvukumbwe, prom. geog.
Study programme:	Informatics
Thesis supervisor:	Ing. Tomáš Vokoun
Supervising department:	Department of Information Technologies
Language of a thesis:	English
Thesis title:	**Linux server system setup and configuration using terraform and ansible**

Objectives of thesis: Main objective

The main objective of this thesis is to Investigate how a small company can leverage open-source tools such as Ansible and Terraform to run efficient infrastructure using gitops principles.

Partial objectives

-Explain and evaluate the existing literature on cloud computing, google cloud platform, Linux, ansible, terraform, bitbucket and GitOps.

-Explain and evaluate trends in IT infrastructure from physical hardware to virtual machines.

-Explain how IT infrastructure can be managed using code, from setting up a Linux server using Terraform to configuring it using Ansible.

Methodology: The study will analyze information sources, including books and online journals. The student will also use scientific and expert literature as well as data from academic journals that cover the topics of Linux infrastructure and configuration. The author will also create a virtual machine and configure it.

The proposed extent of the thesis: 40-50

Keywords: Linux, cloud computing, google cloud platform, Linux, ansible, terraform, bitbucket, git ops

Recommended information sources:

BRESNAHAN, Christine and Richard BLUM. CompTIA Linux+ Study Guide: Exam XK0-004. Newark: John Wiley & Sons, Incorporated, 2019. ISBN 9781119556039;1119556031

FLICKENGER, Rob. Linux server hacks. Sebastopol: O'Reilly, 2003. ISBN 9780596004613;0596004613

HOCHSTEIN, Lorin and René MOSER. Ansible: up and running: automating configuration management and deployment the easy way. Second. ed. Beijing;Farnham;Sebastopol;Tokyo;Boston: O'Reilly, 2017. ISBN 1491979801;9781491979808

Linux System Administration for the 2020s: The Modern Sysadmin Leaving Behind the Culture of Build and Maintain by Kenneth Hitchcock Released February 2022 Publisher(s): Apress ISBN: 9781484279847

MALLETT, Andrew and SpringerLink (ONLINE SLUŽBA). Red Hat certified engineer (RHCE) study guide: ansible automation for the Red Hat Enterprise Linux 8 Exam (EX294). New York: Apress, 2021. ISBN 148426861X;9781484268612

Red Hat Certified System Administrator (RHCSA) RHEL 9 by Sander van Vugt Released March 2023 Publisher(s): Pearson IT Certification ISBN: 0137931522

Expected date of thesis defence: 2023/24 WS - PEF

Electronically approved: 4. 7. 2023

doc. Ing. Jiří Vaněk, Ph.D.

Head of department

Declaration

I declare that I have worked on my bachelor thesis titled " Linux server system setup and configuration using terraform and ansible" by myself and I have used only the sources mentioned at the end of the thesis. As the author of the bachelor thesis, I declare that the thesis does not break any copyrights.

In Prague on 8/6/2023 _____

Acknowledgement

I would like to thank my supervisor for his guidance and my family and friends for their support

Summary

This thesis is about how a small firm, and entrepreneurs can take advantage of open-source technologies.

It shows how   the process of setting up a linux server is done using infrastructure as code and git ops principles.

southern

Tato práce pojednává o tom, jak mohou malá firma a podnikatelé využít výhod open-source technologií.

Ukazuje, jak se proces nastavení linuxového serveru provádí pomocí infrastruktury jako kódu a principů git op

Table of content

Introduction

Small businesses and entrepreneurs can simply get started thanks to open-source technologies. They allow small businesses to cut costs and to modify them for their needs without many restrictions because they are free. Setting up servers can be a very expensive and challenging task. Both money and a lot of expertise

are needed. The configuration management process can be automated with the use of open-source tools like ansible. Automation of the procedure eliminates the need for repetitive configuration effort. The provisioning of the servers is aided by Terraform, which speeds up the process if the number of servers needed needs to be increased. Utilizing cloud platforms like AWS and GCP enables small businesses to launch without having to make significant financial investments. The business can modify these tools to meet its needs by utilizing open-source technologies.

There is practically no cost involved. The tools can be replaced by additional options. For Ansible, for instance, such options might be Chef or Puppet. Windows and Mac are potential Linux substitutes. However, there are numerous Linux distributions, to name a few: openSUSE, Fedora, Debian, Ubuntu, and Fedora.

Using the Ubuntu distribution will be used in this study report. There are also a number of additional cloud computing options, including Linde, IBM Cloud,

Microsoft Azure, and Oracle Cloud. Bitbucket equivalents include things like GitHub, GitLab, and Azure Repositories, among others.

Objectives and Methodology

Main objective

The main objective of this thesis is to Investigate how a small company can leverage open source tools such as Ansible and Terraform to run efficient infrastructure using GitOps principles.

Partial objectives

-Explain and evaluate the existing literature on cloud computing, google cloud platform, Linux, ansible, terraform, bitbucket and GitOps.

-Explain and evaluate trends in IT infrastructure from physical hardware to virtual machines.

-Explain how IT infrastructure can be managed using code, from setting up a Linux server using Terraform to configuring it using Ansible.

Methodologies

The study will analyze information sources, including books and online journals. The student will also use scientific and expert literature as well as data from academic journals that cover the topics of Linux infrastructure and configuration. The author will also create a virtual machine and configure it.

Literature Review

GitOps

Introduction

GitOps is an operational framework that uses continuous integration/continuous delivery (CI/CD), version control, and DevOps methods to automate infrastructure and manage software deployment. It enables programmers to automate operational processes by saving the ideal infrastructure condition. From the beginning of the development workflow all the way through deployment, a set of best

...

practices known as GitOps are used(gitlab 2023). GitOps is a developer-focused approach that uses tools that developers are already familiar with. Git is currently used by developers to store the source code for applications; GitOps extends this concept to encompass infrastructure, operational procedures, and

application configuration. All of a project's infrastructure, including that which takes the form of code, configuration, and application code files, is maintained by GitOps in Git repositories. Every change to the apps and infrastructure is immediately synchronized with the live environment.(2023 codefresh)

GitOps can be used to manage the deployments of any infrastructure. Platform engineers and software developers that use Kubernetes and want to switch to continuous operating models will find it especially useful. GitOps makes continuous deployment for cloud-native applications straightforward. This is

achieved by ensuring that cloud infrastructure can be quickly replicated based on a Git repository's current state.

GitOps vs DevOps

It is important to stress that GitOps is not a natural extension of DevOps or even DevOps 2.0. GitOps is a set of deployment techniques, whereas DevOps is a philosophy, or better yet, an attitude. Due to the GitOps process and

contemporary DevOps approaches' shared concepts, teams find it easier to apply them.

The benefits of GitOps

A GitOps framework enables infrastructure automation, and while automation is valuable in and of itself, GitOps also has other advantages. Companies that use GitOps gain additional benefits with long-lasting effects.

Cooperating on infrastructure upgrades will allow senior engineers to focus on issues other than infrastructure management because each update will go through the same change/merge request/review/approval process.

Time to market is shortened since manual point-and-click operations take longer to execute in code. Stable environments can be provided fast due to the repeated and automated nature of test cases.

Making audits simpler. When infrastructure upgrades are made manually across numerous different interfaces, auditing can become challenging and time-

consuming. Data must be gathered and harmonized from numerous sources in order to execute the audit. GitOps makes audits easy because all changes to environments are documented in the git log.

reduced danger. All infrastructure changes are tracked via merge requests, which also enable rolling back modifications to an earlier state**.**

less liable to mistakes. Infrastructure descriptions are recorded and replicable, which lowers the possibility of human error. Through the use of collaborative merge requests and code reviews, errors can be detected and rectified before being released into production.

reduced costs and downtime. Automation of infrastructure definition and testing reduces downtime, boosts productivity, and gets rid of human tasks thanks to

built-in revert/rollback functionality. Additionally, automation makes it possible for IT staff to better manage cloud resources, which reduces cloud costs.

improved access control. Since changes are automatically made, just CI/CD needs access, so all infrastructure components don't need login information.

Typical GitOps tools

GitOps stands out since it consists of numerous platforms, plugins, and products.

Teams may manage IT infrastructure using the same processes they use to develop applications, thanks to a solution called GitOps. Although Ansible,

Terraform, and Kubernetes are popular technologies, the GitOps process is largely technology-independent (with the obvious exception of Git).

GitOps is appropriate in a variety of circumstances. For instance, Kubernetes and GitOps complement each other quite nicely. Kubernetes, which is supported by all of the main cloud platforms, makes use of stateless and immutable containers.

Because Kubernetes-based containerized apps are self-contained, you don't need to provision and build up servers for every program. Kubernetes clusters and

other essential infrastructure, such as networking and database systems, are provisioned using Terraform.

When deploying stateful apps, external services like Redis caches and Amazon Aurora database instances need to be taken into account. Even though Kubernetes is a solid foundation for a GitOps architecture,

GitOps execution is not required. It can also be utilized with virtual machines and other traditional cloud infrastructure. In this case, new VMs would be provisioned using Terraform, and they would be set up using a configuration management tool like Ansible (gitlab 2023).

	Tools	
Git repository	git	
Git management tool	GitHub	bitbucket
Continuous integration tool	Jenkins	circleci
Continuous delivery tool	spinnaker	flux
Configuration manager	ansible	Puppet
Infrastructure provisioning	terraform	Aws CloudFormation
Container orchestration	Kubernetes	

Introduction to cloud computing

A small business can use cloud computing to obtain all the IT infrastructure it requires. To provide quicker innovation, adaptable resources, and scale economies, cloud computing is the supply of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the internet.

According to Azure 2023, users often only pay for the cloud services they really use, which lowers operational expenses, improves infrastructure management, and enables them to scale as their business requirements evolve.

A small business may decide to use a cloud platform instead of its own on- premises infrastructure for a variety of reasons. The following are a few potential motives:

- By reducing the costs related to maintaining IT infrastructure, using the cloud platform will significantly reduce the costs of IT infrastructure. To put it another way, using the cloud enables users to swap fixed costs for variable costs. Only what customers actually utilize will be charged. They use the pay-as-you-go method of payment.

The cloud is more secure than on-premise since it offers a broad set of policies, technologies, and controls that strengthen security, helping protect data, apps, and infrastructure from potential threats.

- The cloud enables the instantaneous delivery of computing services. This makes it possible for the business to get going right away rather than having to wait for the hardware to be set up, which could take weeks or even months. - The cloud is more trustworthy. They do this by making data

backup, disaster recovery, and business continuity simpler and less

expensive because of the ability to mirror data across numerous redundant sites on the network of the cloud provider.

The cloud also makes it possible for businesses to quickly extend into other countries and regions. Faster server and service deployments are made possible by it...

Types of cloud computing

Public cloud

These are owned and run by a separate cloud service provider that offers online access to computer resources like servers and storage. In a public cloud, the cloud provider owns and manages all of the hardware, software, and other supporting infrastructure.

Private cloud

Cloud computing resources used solely by a single company or organization are referred to as private clouds. Physically, a private cloud may be situated on the business's on-site datacenter.

Hybrid cloud

Public and private clouds are combined in hybrid clouds, which are connected by a system that enables data and applications to be transferred between them. A hybrid cloud allows your company more flexibility and deployment options by enabling data and apps to migrate between private and public clouds. It also helps to optimize your current infrastructure, security, and compliance.

(IaaS) Infrastructure as a Service

IaaS contains the basic building blocks of cloud computing. It is common practice to provide access to networking capabilities, computers (virtual or on dedicated hardware), and data storage capacity. IaaS gives you the most flexibility and administrative control over your IT resources. It is most similar to the present IT resources that many IT businesses and developers are used to (AWS 2023).

PaaS (Platform as a Service)

By relieving you of the burden of managing the underlying infrastructure (typically hardware and operating systems), PaaS allows you to focus on the deployment and administration of your apps. Because you won't have to worry about things

like resource acquisition, capacity planning, software maintenance, patching, or any other undifferentiated heavy lifting (AWS 2023), you can run your application more effectively.

SaaS (software as a service)

Through SaaS, you can have a complete product that is run and managed by the service provider. When someone talks about SaaS, they typically mean end-user apps, such as web-based email. When choosing a SaaS provider, you don't have to worry about how the service is managed or the underlying infrastructure is

maintained. You need just think about how you intend to use that particular piece of software (AWS 2023).

Terraform

Terraform is a declarative coding tool that enables developers to specify the intended "end-state" cloud or on-premises infrastructure for operating an application using the high-level configuration language known as HCL (HashiCorp Configuration Language). After that, it creates a plan to get there and carries it out to furnish the infrastructure.

Terraform is now one of the most well-liked infrastructure automation tools

because it has a clear syntax, can provision infrastructure across different cloud and on-premises data centers, and can safely and effectively re-provision

infrastructure in response to configuration changes. You'll probably want to learn about Terraform if your company intends to create a hybrid cloud or multi-cloud

infrastructure(IBM 2023).

It helps to first grasp the advantages of Infrastructure as Code (IaC) to better understand the benefits of Terraform. IaC enables developers to codify infrastructure in a way that automates, accelerates, and repeats provisioning. It's an essential part of Agile and DevOps techniques like continuous integration, continuous deployment, and version control.

Infrastructure provisioning using terraform.

Is an infrastructure as code tool that is used to build, change, and version cloud and on-prem resources safely and efficiently. It also allows the user to define both cloud and on-prem resources in human-readable configuration files that can be reused and shared. Terraform can manage low-level components like computer, storage, and networking resources, as well as high-level components like DNS entries (hashicorp 2023). That can provision resources from the cloud from simple declarative code.

Infrastructure as code can help with the following:

Boost speed: When it comes to deploying and/or connecting resources, automation is quicker than manually browsing an interface.

Increased dependability is important since it is simple to setup resources incorrectly or to provision services in the wrong sequence when your

infrastructure is huge. The resources are always provided and set up exactly as declared when using IaC.

Prevent configuration drift: When the configuration that was used to provision your environment no longer corresponds to the real environment, configuration drift occurs. (Read more about 'Immutable infrastructure' below.)

Support for experimentation, testing, and optimization: Infrastructure as Code makes provisioning new infrastructure so much faster and simpler that you can

make and test experimental changes without devoting a lot of time and resources. If the results are positive, you can quickly scale up the new infrastructure for production.

Why Terraform?

Developers favor Terraform over competing Infrastructure as Code technologies for a few main reasons:

Open source: Terraform is supported by sizable contributor communities that create platform extensions. No matter which cloud provider you choose, it's simple to locate plugins, extensions, and expert support. This also means that terraform develops quickly, adding new features and benefits on a regular basis.

Platform agnostic: It is platform-agnostic, so you can use it with any supplier of cloud services. The majority of other IaC solutions are made to function with a

single cloud provider.

Immutable infrastructure: Infrastructure as Code technologies typically produce malleable infrastructure, which can alter to accommodate modifications like a middleware upgrade or new storage server. Configuration drift is the risk with mutable infrastructure; when the modifications multiply, the provisioning of various servers or other infrastructure components 'drifts' further from the initial configuration, making problems or performance issues challenging to identify and fix. Terraform creates immutable infrastructure, which implies that if the

environment changes, the infrastructure is reprovisioned with a new configuration that takes the new environment into account. Better still, earlier settings can be kept around as versions, allowing rollbacks if required or desired.

controlling any infrastructure

In the Terraform Registry, you may find providers for many of the platforms and services you already use. You can create your own as well. The complexity of updating or changing your services and infrastructure is reduced by Terraform's immutable approach to infrastructure.

Following your infrastructure

Before changing your infrastructure, terraform produces a plan and requests your consent. Additionally, it maintains a status file of your actual infrastructure, which serves as the environment's single source of truth. Terraform analyzes the state

file to decide what modifications need to be made to your infrastructure in order for it to comply with your setup.

Automate your changes.

Declarative configuration files for Terraform explain the final state of your

infrastructure. Terraform handles the underlying logic, so creating resources does not require you to write detailed instructions. Terraform simultaneously develops or alters non-dependent resources while constructing a resource graph to identify resource dependencies. Terraform can provision resources effectively because of this.

Establish uniform configurations.

In order to save time and promote best practices, terraform enables reusable configuration components known as modules that define configurable groups of infrastructure. Use the Terraform Registry's publicly accessible modules or create your own.

Collaborate

As your configuration is stored in a file, you may utilize Terraform Cloud to effectively manage Terraform processes between teams and commit your

configuration to a Version Control System (VCS). In addition to offering safe access to shared state and secret data, role-based access controls, a private registry for sharing both modules and providers, and other features, Terraform Cloud

executes Terraform in a dependable, consistent environment.

How does Terraform function?

Terraform uses application programming interfaces (APIs) to construct and

manage resources on cloud platforms and other services. Terraform can integrate with practically any platform or service that has an accessible API, thanks to

providers.

To handle a wide range of resources and services, HashiCorp and the Terraform community have already created thousands of providers. All publicly accessible providers, such as Amazon Web Services (AWS), Azure, Google Cloud Platform (GCP) and many others, may be found on the Terraform Registry.

The primary stages of the Terraform workflow are as follows:

You specify the resources, which might be spread across various cloud service providers. As an example, you could build up a network in a Virtual Private Cloud (VPC) with security groups and a load balancer to deploy an application on virtual machines.

Plan: Terraform creates an execution plan that details the infrastructure it will create, update, or remove based on the already-existing infrastructure and your configuration.

Apply: After accepting, Terraform accurately and with care for any resource requirements carries out the specified activities. For instance, if you alter the

features of a VPC and alter the number of virtual machines in that VPC. Terraform will regenerate the VPC before scaling the virtual machines.

Modules for Terraform

Terraform modules are condensed, reusable Terraform configurations for a

collection of infrastructure resources. Because they use reusable, customizable components, Terraform modules allow for the automated automation of complex resources. A module is created even while writing a very simple Terraform file. The configuration-assembly procedure can be sped up and made easier by a module's capacity to call other modules, also referred to as child modules. Modules may also be invoked several times, either within the same configuration or in different configurations (IBM 2023).

Terraform providers.

Resource types are implemented via plugins called terraform providers. On behalf of the user, providers have all the necessary code to authenticate and connect to a service—typically from a public cloud provider. You can locate providers for the cloud services and platforms you employ, include them in your configuration, and then use their resources to provision infrastructure. Nearly all the main cloud

providers, SaaS offerings, and other services have providers that were created and/or are sponsored by the Terraform community or specific businesses

(HASHICORP 2023).

What is Ansible?

Ansible is an infrastructure automation, tool that automates provisioning, configuration management, application deployment, orchestration, and many

other manual IT processes. It is an agentless automation tool that can be installed on a single host (referred to as the control node). From the control node, Ansible can manage an entire fleet of machines and other devices (referred to as

managed nodes) remotely with SSH all from a simple command-line interface. There is no need for extra servers, daemons, or databases (Redhat 2021).

How Ansible works

Ansible works by establishing a connection to the task you wish to automate and then pushing scripts that carry out instructions that would have been manually carried out. These applications make use of Ansible modules that were created

with a focus on the connection, user interface, and command expectations of the endpoint. The modules are then executed by Ansible (by default over regular SSH) and, if necessary, removed at the end (Redhat 2021).

What are Ansible playbooks?

Ansible playbooks are blueprints for automation tasks, which are advanced IT operations carried out without the need for human intervention. Ansible playbooks are written in human-readable YAML format and executed on a set,

group, or classification of hosts, which together make up an Ansible inventory. We will discuss more of how the ansible playbooks work in the practical part of this thesis paper.

Uses of Ansible

Infrastructure provisioning with Ansible

The infrastructure (such as a server or cloud endpoint, for example), must first be setup before you can install and configure an application. Companies aiming to

scale IT rapidly and reliably use Ansible playbooks since manually provisioning hundreds or thousands of servers is not practical. The practice of supplying hundreds or thousands is no longer practical. You can create a single instance with

Ansible, then use it or any number of other servers immediately using the same infrastructure parameters or information. The next step after provisioning the environment is configuration, which Ansible excels at handling as part of the IT operational life cycle(Redhat 2021).

Configuration management with Ansible

The simplest method for automating common IT operations is Ansible. It is intended to have a very little learning curve for administrators, developers, and IT managers and to be minimal in nature, consistent, secure, and highly

reliable. Ansible uses straightforward data descriptions of your infrastructure that are both machine-passable and human-readable, ensuring that any member of your team can comprehend the purpose of each configuration task.

Application deployment with Ansible

With Ansible, you can reliably and consistently deploy multi-tier applications using a single framework. From a single shared system, you can push application

artifacts and configure necessary services. Your team no longer creates specialized code to automate your processes; instead, they merely provide straightforward task descriptions that even the newest team member can immediately comprehend. This reduces upfront expenses and makes it simpler to respond to ongoing change.

In this research paper we are going to use ansible mainly for configuration management.

Practical part

At first, we are going to creation a linux server by using terraform. For this we will need the following and we will need to install terraform.

Prerequisites for creating a virtual machine using terraform.

Github account- this is where we will keep our files. We can easily create a free account.

AWS account – we will use Amazon web service as our cloud platform. We can a free account and use it for a year before we start to get charge

AWS access key id – we will need this to identify which account we will create the server on

AWS secret access key – this is to grant us the permission to create the virtual machine.

Terraform – it's required for us to use it to create the virtual machine.

Linux based machine – We are going use a linux machine as a preference not a requirement. We can use a code editor like virtual studio code and install terraform and use a terminal to run commands.

Installing terraform on amazon Linux

We will be using an amazon Linux machine and we will need to install terraform using the following commands and process

Install yum-config-manager to manage your repositories

```
sudo yum install -y yum-utils
```

We will use the following command to add the official Hashicorp Linux repository.

```
sudo yum-config-manager --add-repo
https://rpm.releases.hashicorp.com/AmazonLinux/hashicorp.repo
```

3 Final we install terraform

```
sudo yum -y install terraform
```

After we will check if terraform is installed and also that we can the current version

```
terraform --version
```

After the above steps, we will then move on to create a GitHub repository. We will keep our project files

First, we will need to create an SSH key by running the below command

```
[root@ip-172-31-17-0 ~]# ssh-keygen
```

The above command will create a ssh key. We will copy this key and paste it in our github account under the ssh and gpg keys.

After this, we will create two repositories, one for our ansible and another for Terraform

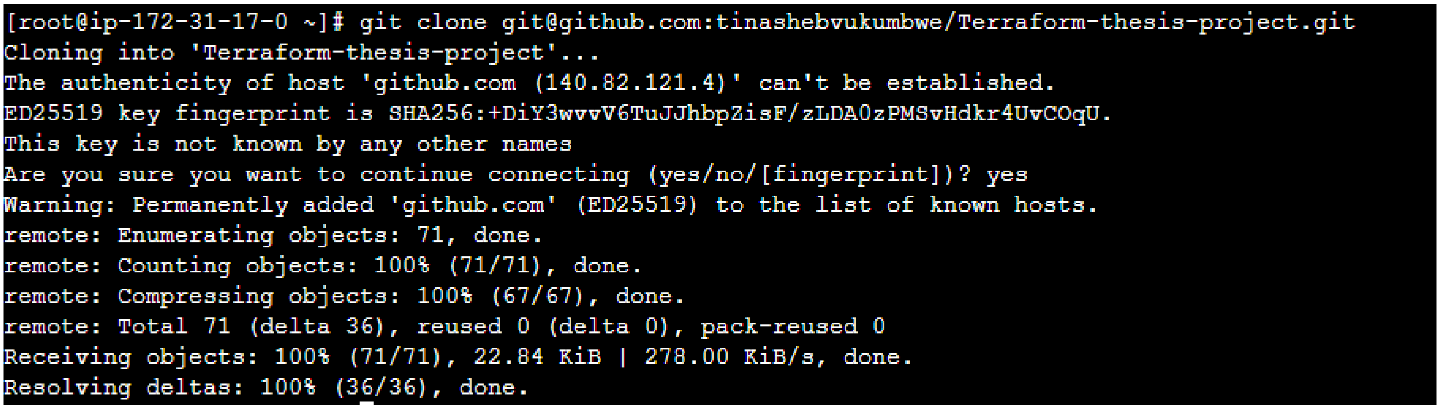
We will then get into one of these repositories and copy their URL and run the below command to download the folder in our linux virtual machine. At first we will start with terraform

At first we will run the below command to see if we have git installed

```
[root@ip-172-31-17-0 ~]# which git

/usr/bin/git
```

The below command will download our git repository

```
[root@ip-172-31-17-0 ~]# git clone git@github.com:tinashebvukumbwe/Terraform-thesis-project.git
```



```
[root@ip-172-31-17-0 ~]# git clone git@github.com:tinashebvukumbwe/Terraform-thesis-project.git
Cloning into 'Terraform-thesis-project'...
The authenticity of host 'github.com (140.82.121.4)' can't be established.
ED25519 key fingerprint is SHA256:+DiY3wvvV6TuJJhbpZisF/zLDA0zPMSvHdkr4UvCOqU.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added 'github.com' (ED25519) to the list of known hosts.
remote: Enumerating objects: 71, done.
remote: Counting objects: 100% (71/71), done.
remote: Compressing objects: 100% (67/67), done.
remote: Total 71 (delta 36), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (71/71), 22.84 KiB | 278.00 KiB/s, done.
Resolving deltas: 100% (36/36), done.
```

The below credentials will tell git who we are.

```
[root@ip-172-31-17-0 Terraform-thesis-project]# git config --global user.name "tinashebvukumbwe"

[root@ip-172-31-17-0 Terraform-thesis-project]# git config --global user.email "trbvukumbwe1@gmail.com"
```

The below commands contain our aws access key id and our aws secret access key. These will enable us to create resources on our aws account

```
[root@ip-172-31-17-0 Terraform-thesis-project]# export AWS_ACCESS_KEY_ID="AKIAXYKJWZRVTRZGVHWW"

[root@ip-172-31-17-0 Terraform-thesis-project]# export AWS_SECRET_ACCESS_KEY="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"
```

Creating a linux instance and a virtual private network

After the above steps are completed. We will start to write our terraform code. This is the code that will be used to create all the resources we will need to set up a Linux server.

Main.tf

```
resource "aws_instance" "thesis_server" {

  ami           = "ami-03484a09b43a06725"

  instance_type = "t2.micro"
  tags = {
    Name = "exampleserver"
  }
}


resource "aws_vpc" "this" {

  for_each             = var.vpc_parameters

  cidr_block           = each.value.cidr_block

  enable_dns_support   = each.value.enable_dns_support

  enable_dns_hostnames = each.value.enable_dns_hostnames

  tags = merge(each.value.tags, {

    Name : each.key

  })

}



resource "aws_subnet" "this" {

  for_each   = var.subnet_parameters

  vpc_id     = aws_vpc.this[each.value.vpc_name].id
  cidr_block = each.value.cidr_block
  tags = merge(each.value.tags, {
    Name : each.key
  })
}


resource "aws_internet_gateway" "this" {
  for_each = var.internet_gatway_parameters
  vpc_id   = aws_vpc.this[each.value.vpc_name].id
  tags = merge(each.value.tags, {
    Name : each.key
  })
}


resource "aws_route_table" "this" {
  for_each = var.route_table_parameters
  vpc_id   = aws_vpc.this[each.value.vpc_name].id
  tags = merge(each.value.tags, {
    Name : each.key
  })

  dynamic "route" {
    for_each = each.value.routes
    content {
      cidr_block = route.value.cidr_block
      gateway_id = route.value.use_igw ?
aws_internet_gateway.this[route.value.gateway_id].id : route.value.gateway_id
    }
  }
}

resource "aws_route_table_association" "this" {
  for_each       = var.route_table_association_parameters

  subnet_id      = aws_subnet.this[each.value.subnet_name].id

  route_table_id = aws_route_table.this[each.value.rt_name].id

}
provider "aws" {
  region = var.region
}
```

outputs.tf

```
output "vpcs" {

  description = "VPC Outputs"

  value       = { for vpc in aws_vpc.this : vpc.tags.Name => { "cidr_block" : vpc.cidr_block, "id" : vpc.id } }

}
```

providers.tf

```
terraform {

  required_version = "~> 1.7.4"

  required_providers {

    aws = {

      source  = "hashicorp/aws"
      version = ">= 5.0.0"
    }
  }
}
```

variables

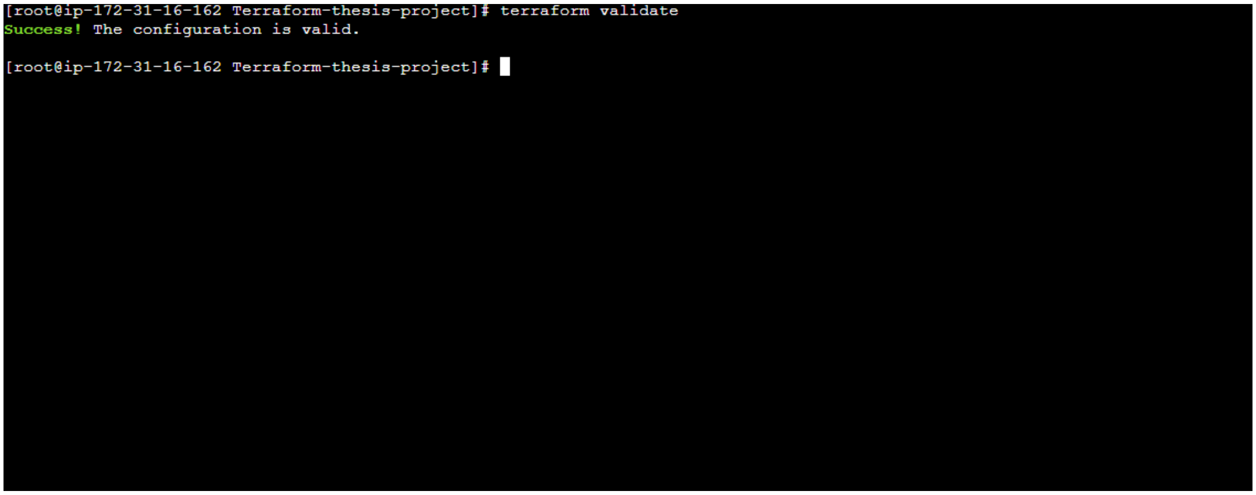
```
variable "vpc_parameters" {
  description = "VPC parameters"
  type = map(object({
    cidr_block           = string
    enable_dns_support   = optional(bool, true)
    enable_dns_hostnames = optional(bool, true)
    tags                 = optional(map(string), {})
  }))
  default = {}
}
```

```
variable "subnet_parameters" {
  description = "Subnet parameters"
  type = map(object({
    cidr_block = string
    vpc_name   = string
    tags       = optional(map(string), {})
  }))
  default = {}
}

variable "internet_gatway_parameters" {
  description = "Internet gateway parameters"
  type = map(object({
    vpc_name = string
    tags     = optional(map(string), {})
  }))
  default = {}
}


variable "route_table_parameters" {
  description = "Route table parameters"
  type = map(object({
    vpc_name = string
    tags     = optional(map(string), {})
    routes = optional(list(object({
      cidr_block = string
      use_igw    = optional(bool, true)
      gateway_id = string
    })), [])
  }))

  default = {}

}

variable "route_table_association_parameters" {

  description = "Route Table association parameters"

  type = map(object({

    subnet_name = string

    rt_name     = string

  }))

  default = {}

}
variable "region" {
  description = "AWS region"
  type        = string
  default     = "eu-central-1"
}
```

After creating the above files, we will then run some terraform commands to create all the resources we mentioned above

Firstly, we will need to the validity of our syntax by running the following command

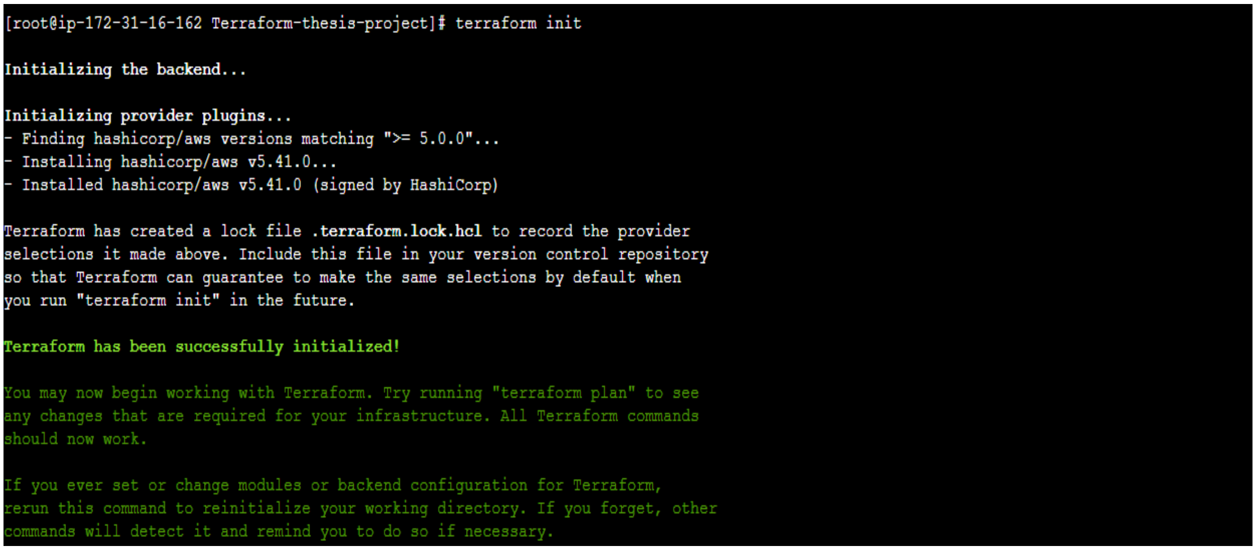
```
terraform validate
```



```
[root@ip-172-31-16-162 Terraform-thesis-project]# terraform validate
Success! The configuration is valid.

[root@ip-172-31-16-162 Terraform-thesis-project]# 
```

If everything is okay, we will move on to running the next command

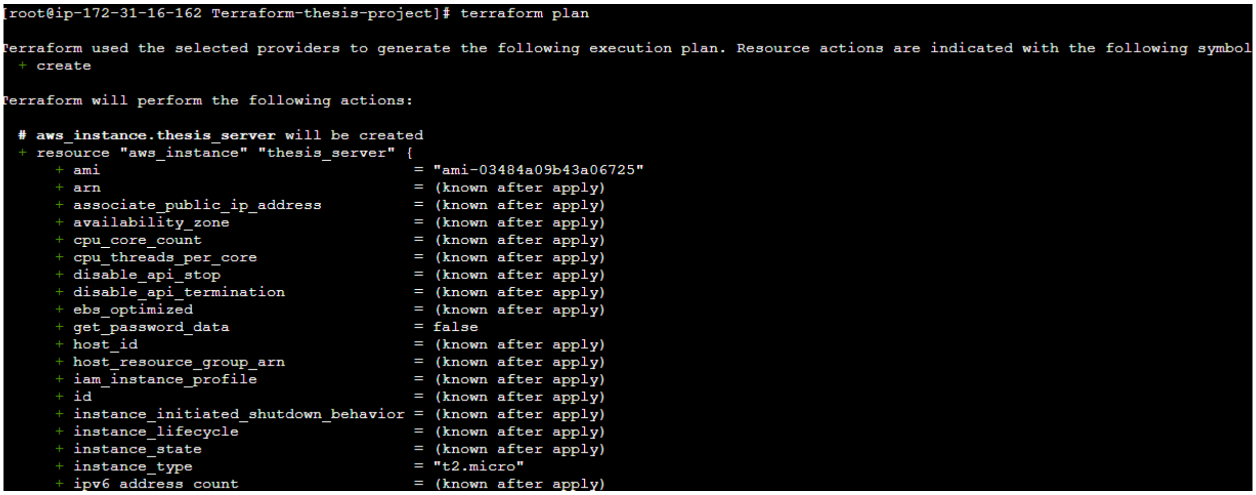
```
[root@ip-172-31-16-162 Terraform-thesis-project]# terraform init
```



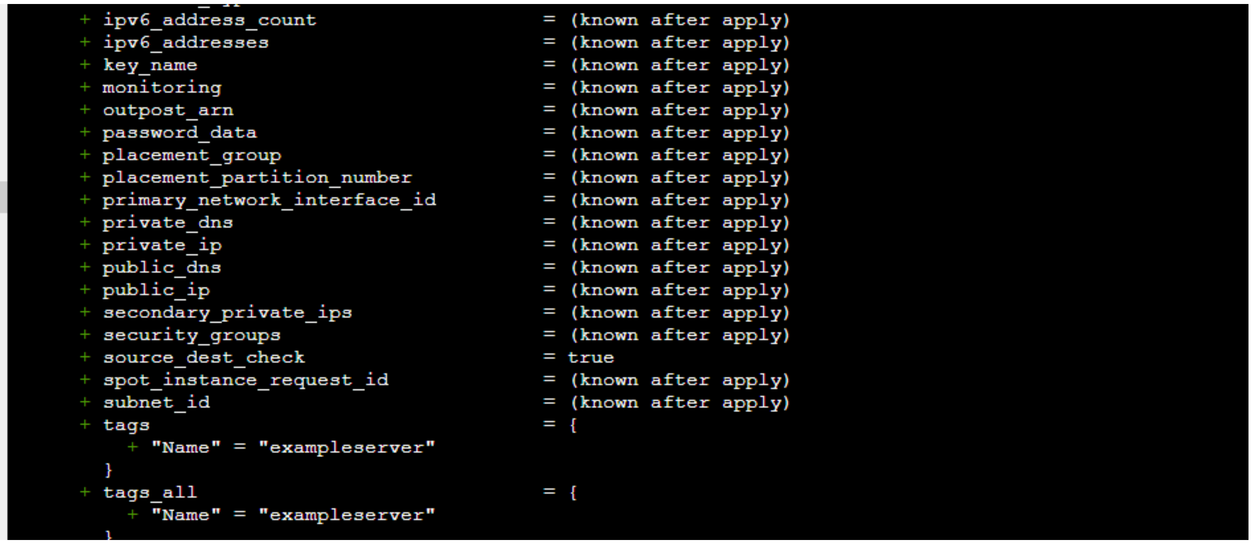
```
[root@ip-172-31-16-162 Terraform-thesis-project]# terraform init

Initializing the backend...

Initializing provider plugins...
- Finding hashicorp/aws versions matching ">= 5.0.0"...
- Installing hashicorp/aws v5.41.0...
- Installed hashicorp/aws v5.41.0 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

The above image is the output of the terraform init command. Since terraform has been initialized successfully, we will move on to the next command :

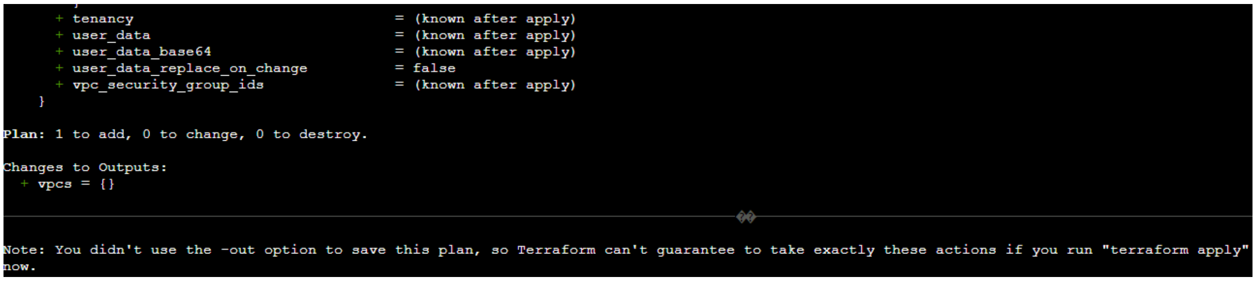
```
[root@ip-172-31-16-162 Terraform-thesis-project]# terraform plan
```



```
[root@ip-172-31-16-162 Terraform-thesis-project]# terraform plan

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbol
  + create

Terraform will perform the following actions:

  # aws_instance.thesis_server will be created
  + resource "aws_instance" "thesis_server" {
      + ami                                  = "ami-03484a09b43a06725"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t2.micro"
      + ipv6_address_count                   = (known after apply)
```



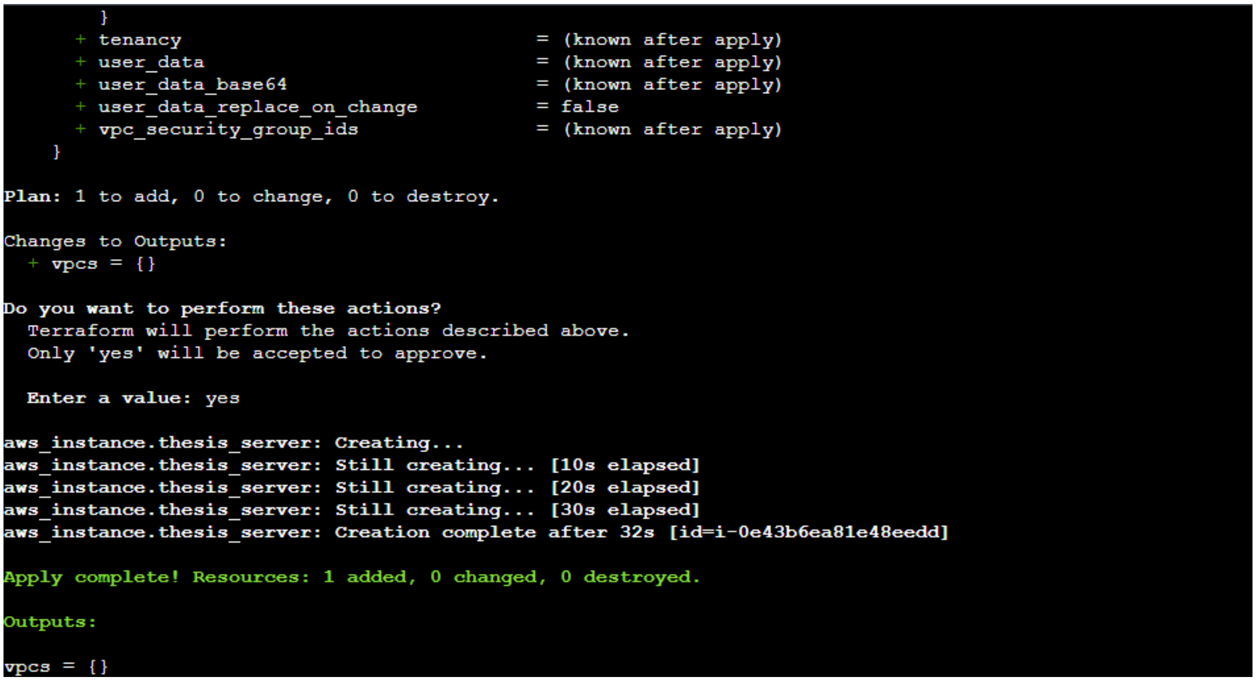
```
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
      + key_name                             = (known after apply)
      + monitoring                           = (known after apply)
      + outpost_arn                          = (known after apply)
      + password_data                        = (known after apply)
      + placement_group                      = (known after apply)
      + placement_partition_number           = (known after apply)
      + primary_network_interface_id         = (known after apply)
      + private_dns                          = (known after apply)
      + private_ip                           = (known after apply)
      + public_dns                           = (known after apply)
      + public_ip                            = (known after apply)
      + secondary_private_ips                = (known after apply)
      + security_groups                      = (known after apply)
      + source_dest_check                    = true
      + spot_instance_request_id             = (known after apply)
      + subnet_id                            = (known after apply)
      + tags                                 = {
          + "Name" = "exampleserver"
        }
      + tags_all                             = {
          + "Name" = "exampleserver"
        }
```



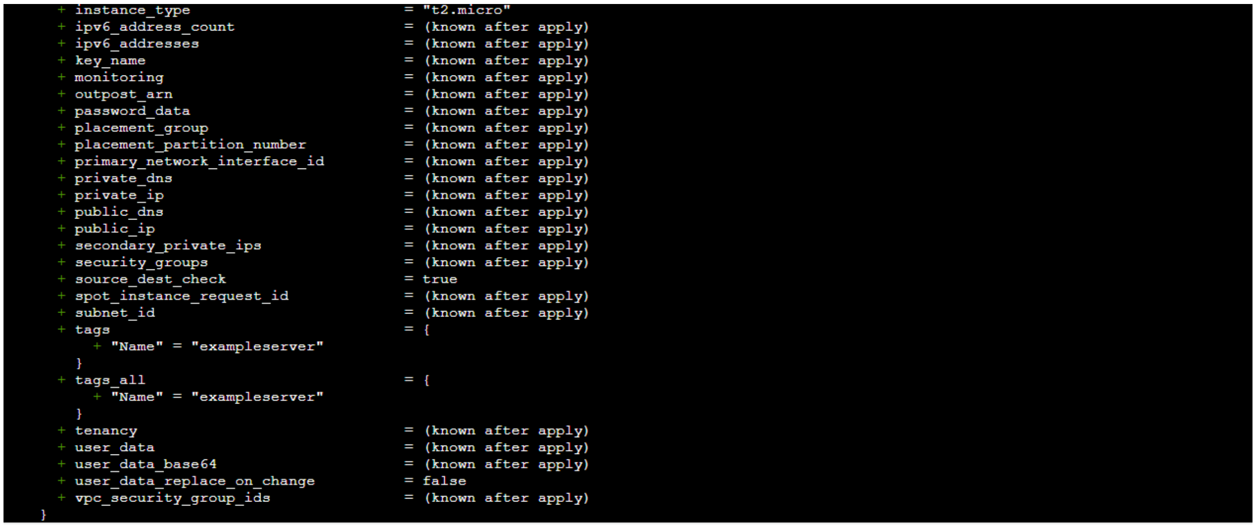
```
      + tenancy                              = (known after apply)
      + user_data                            = (known after apply)
      + user_data_base64                     = (known after apply)
      + user_data_replace_on_change          = false
      + vpc_security_group_ids               = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + vpcs = {}

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform apply"
now.
```

The above image is the output of the terraform plan command. Since the planning has been successful, we will move on to the next command :

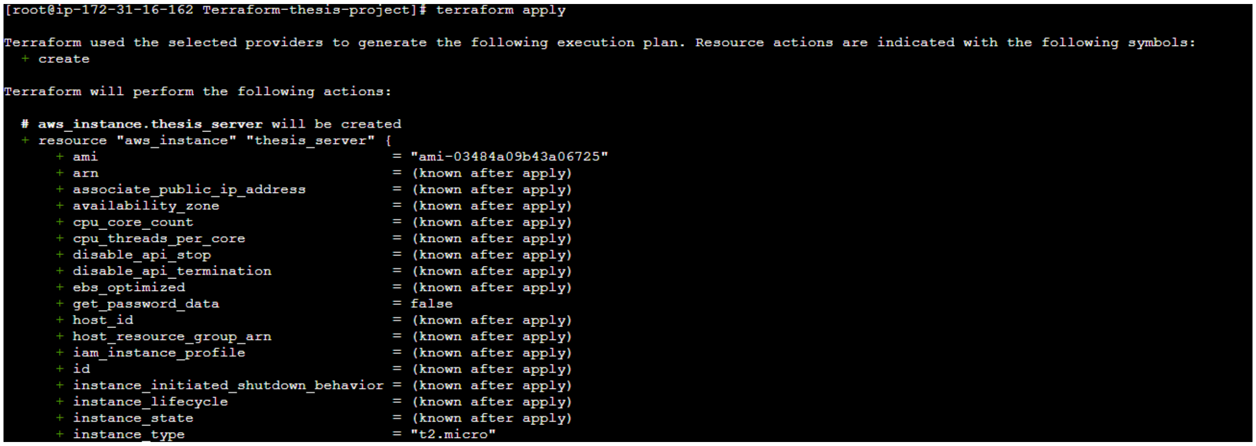
```
[root@ip-172-31-16-162 Terraform-thesis-project]# terraform apply
```



```
        }
      + tenancy                              = (known after apply)
      + user_data                            = (known after apply)
      + user_data_base64                     = (known after apply)
      + user_data_replace_on_change          = false
      + vpc_security_group_ids               = (known after apply)
    }

Plan: 1 to add, 0 to change, 0 to destroy.

Changes to Outputs:
  + vpcs = {}

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

aws_instance.thesis_server: Creating...
aws_instance.thesis_server: Still creating... [10s elapsed]
aws_instance.thesis_server: Still creating... [20s elapsed]
aws_instance.thesis_server: Still creating... [30s elapsed]
aws_instance.thesis_server: Creation complete after 32s [id=i-0e43b6ea81e48eedd]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.

Outputs:

vpcs = {}
```



```
      + instance_type                        = "t2.micro"
      + ipv6_address_count                   = (known after apply)
      + ipv6_addresses                       = (known after apply)
      + key_name                             = (known after apply)
      + monitoring                           = (known after apply)
      + outpost_arn                          = (known after apply)
      + password_data                        = (known after apply)
      + placement_group                      = (known after apply)
      + placement_partition_number           = (known after apply)
      + primary_network_interface_id         = (known after apply)
      + private_dns                          = (known after apply)
      + private_ip                           = (known after apply)
      + public_dns                           = (known after apply)
      + public_ip                            = (known after apply)
      + secondary_private_ips                = (known after apply)
      + security_groups                      = (known after apply)
      + source_dest_check                    = true
      + spot_instance_request_id             = (known after apply)
      + subnet_id                            = (known after apply)
      + tags                                 = {
          + "Name" = "exampleserver"
        }
      + tags_all                             = {
          + "Name" = "exampleserver"
        }
      + tenancy                              = (known after apply)
      + user_data                            = (known after apply)
      + user_data_base64                     = (known after apply)
      + user_data_replace_on_change          = false
      + vpc_security_group_ids               = (known after apply)
    }
```



```
[root@ip-172-31-16-162 Terraform-thesis-project]# terraform apply

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # aws_instance.thesis_server will be created
  + resource "aws_instance" "thesis_server" {
      + ami                                  = "ami-03484a09b43a06725"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
      + instance_initiated_shutdown_behavior = (known after apply)
      + instance_lifecycle                   = (known after apply)
      + instance_state                       = (known after apply)
      + instance_type                        = "t2.micro"
```

Configuration using Ansible

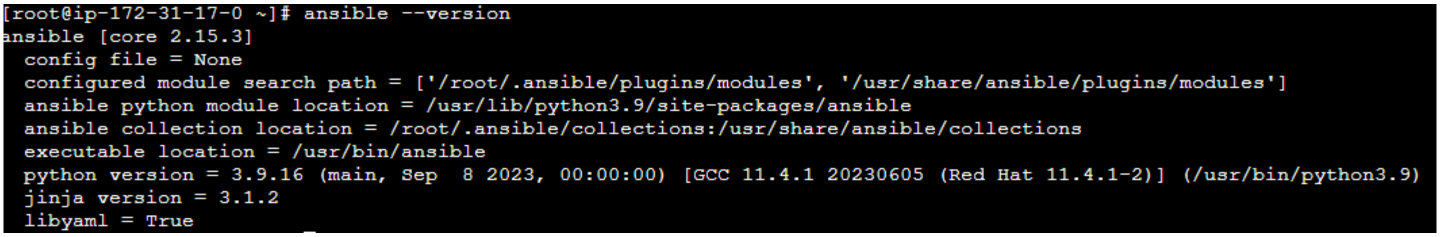
The first thing we need to do is install ansible and also to make sure that we have python and pip install. The server we will use to run ansible needs to have both ansible and python to be installed. The servers we will manage do not need to have ansible install but they will need to have python installed in them.

Inventory

We will keep the servers we want to manage here. The external ip addresses of the server will be stored in the file /etc/hosts

```
server1.thesis.com

server2.thesis.com
```

```
[root@ip-172-31-17-0 ~]# yum install ansible
```



```
[root@ip-172-31-17-0 ~]# ansible --version
ansible [core 2.15.3]
  config file = None
  configured module search path = ['/root/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3.9/site-packages/ansible
  ansible collection location = /root/.ansible/collections:/usr/share/ansible/collections
  executable location = /usr/bin/ansible
  python version = 3.9.16 (main, Sep  8 2023, 00:00:00) [GCC 11.4.1 20230605 (Red Hat 11.4.1-2)] (/usr/bin/python3.9)
  jinja version = 3.1.2
  libyaml = True
```

The above output shows that we have ansible already installed

The below configuration file will enable us to have root access and enable us to login with the help of the ssh key when we will run our playbooks

Ansible.cfg

```
[defaults]

inventory = inventory

remote_user = ec2-user
```

```
host_key_checking = false
deprecation_warning = false

[privilege_escalation]
become = True
become_method = sudo
become_user = root
become_ask_pass = False
```

install-apache.yml

The below script will install the httpd server and start the service, which can be used as a web server

```
---
- name: install httpd
  hosts: all
  tasks:
    - name: install httpd server
      yum:
        name: httpd
        state: present
    - name: create an index.html
      copy:
        content: 'welcome to the thesis web server'
        dest: /var/www/html/index.html
    - name: start the service
      service:
        name: httpd
        state: started
        enabled: true
```

selinux.yml

The below script will enable selinux which is necessary to regulate what each user, process, and daemon is allowed to do on the system. Daemons with well-defined data access and activity privileges, such web servers and database engines, are contained by it.

```
---
- hosts: all
  tasks:
  - name: Enable SELinux for security
    ansible.posix.selinux:
      policy: targeted
      state: enforcing
```

user.yml

The below script will be used to create a user and group

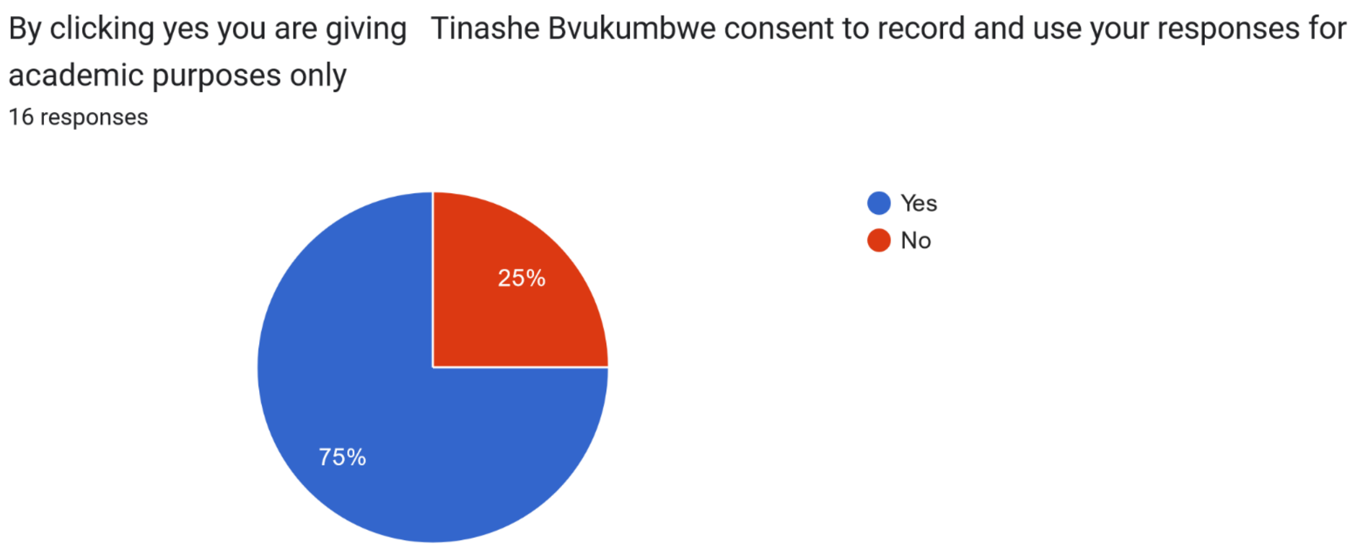
```
---
- name: I am going to create users and groups using this playbook
  hosts: all
  tasks:
   - name: create an admin group
     group:
      name: admin
      state: present
   - name: create admin users
     user:
      name: tinashe
      comment: admin
```

```
        groups: admin
```

Conclusion

Conclusion

For the conclusion of this research paper, l created a survey to ask about the experiences of users, and below are the questions and results

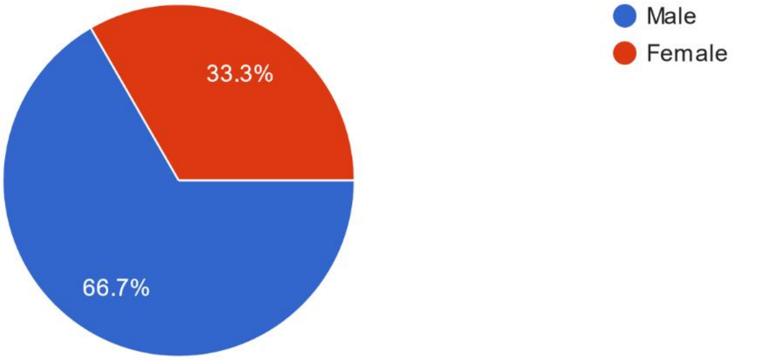


By clicking yes you are giving Tinashe Bvukumbwe consent to record and use your responses for academic purposes only

16 responses

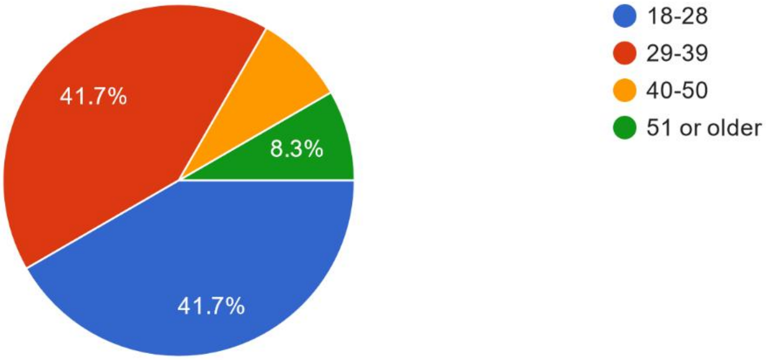
1. What is your gender ?

12 responses



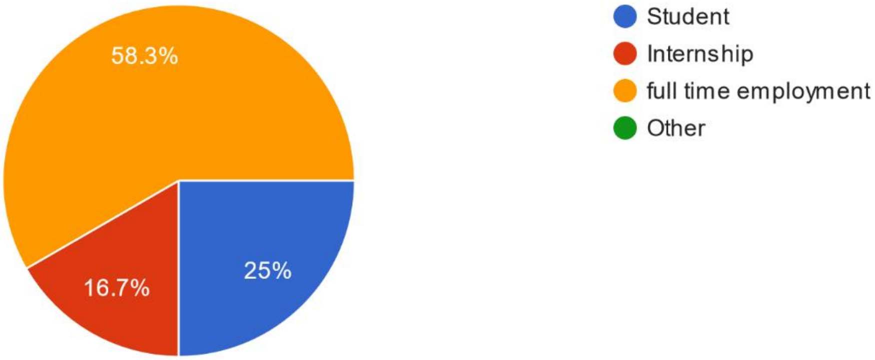
2.What is your age?

12 responses



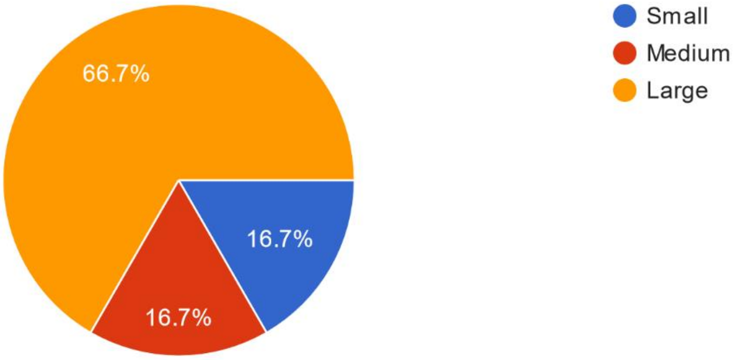
3.What do you do for a living?

12 responses



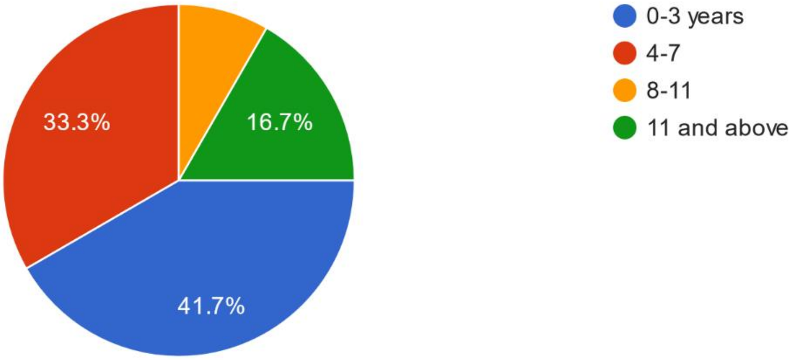
4.How would you categorize the size of your company ?

12 responses



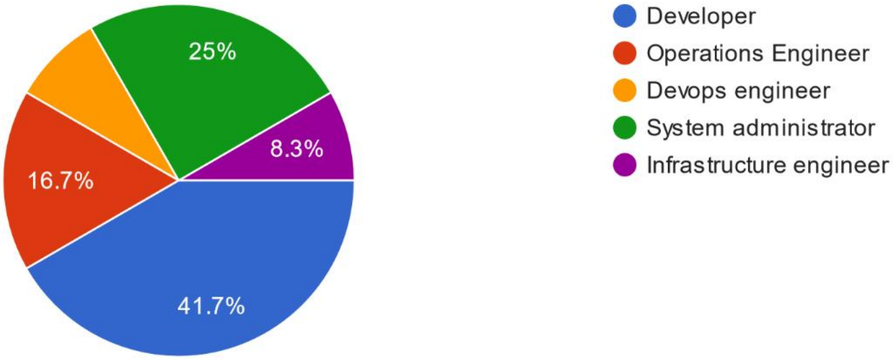
7.How long have you been in the IT industry?

12 responses



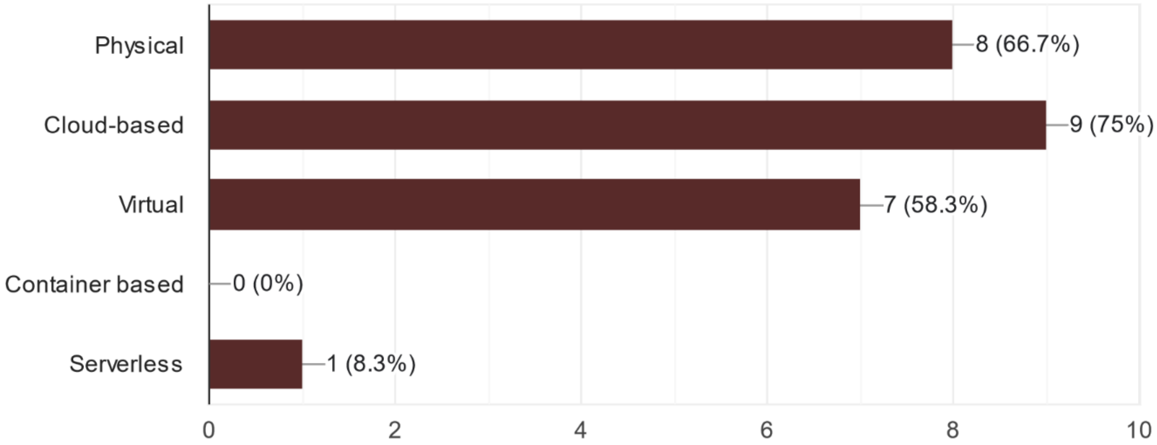
6.What is your job title?

12 responses



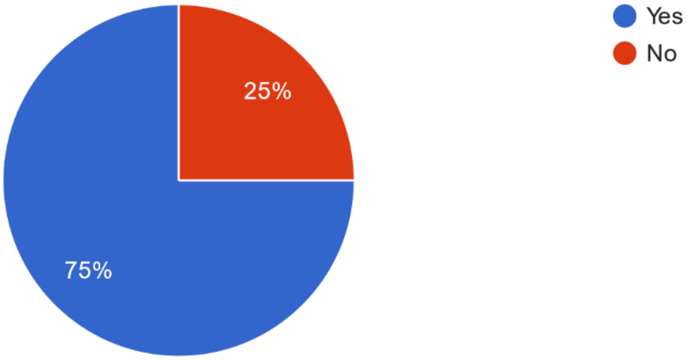
5. What type of servers does your company currently use?

12 responses



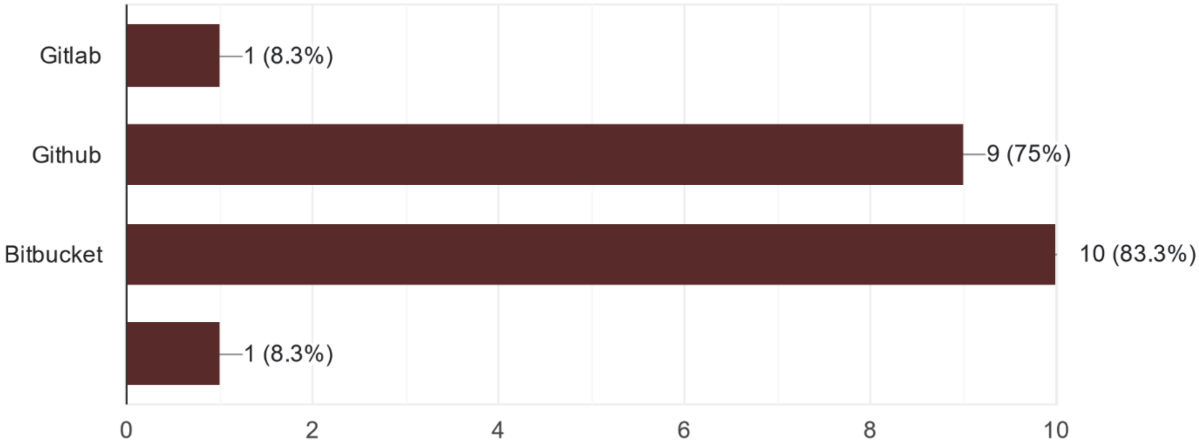
8.Are you familiar with the concept of Version control system ?

12 responses



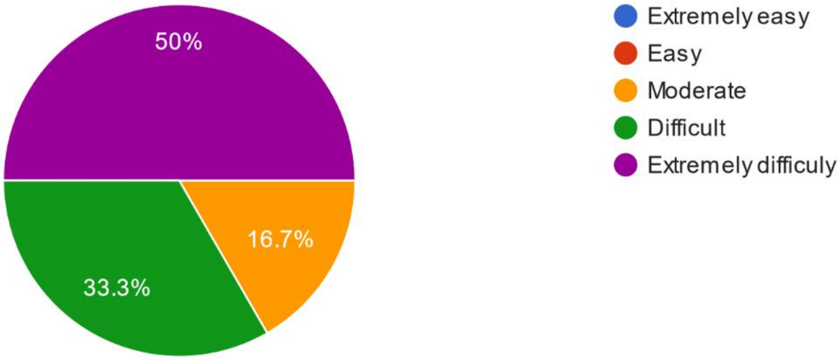
9.Mark the Version control systems you are familiar with ?

12 responses



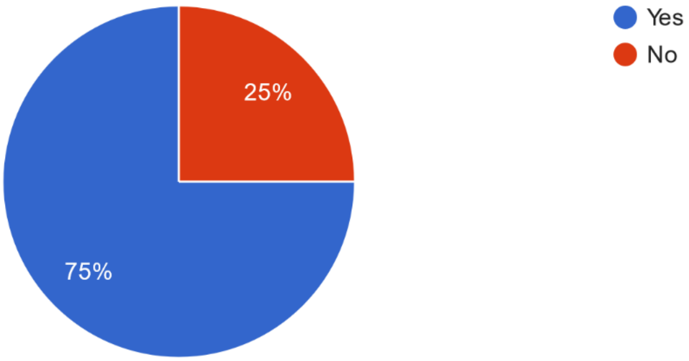
10.Is it difficult to get started with gitops principles?

12 responses



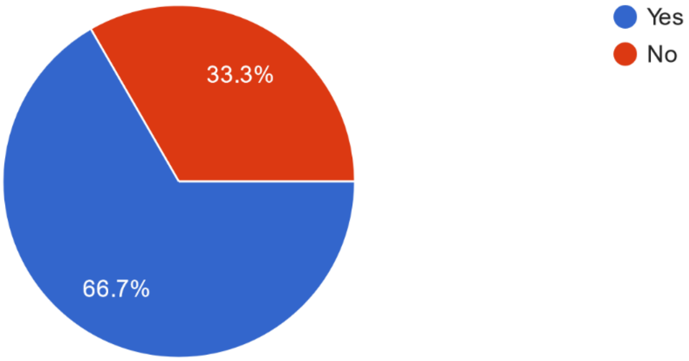
11. Have you used Ansible or Terraform in your infrastructure setup and configuration?

12 responses



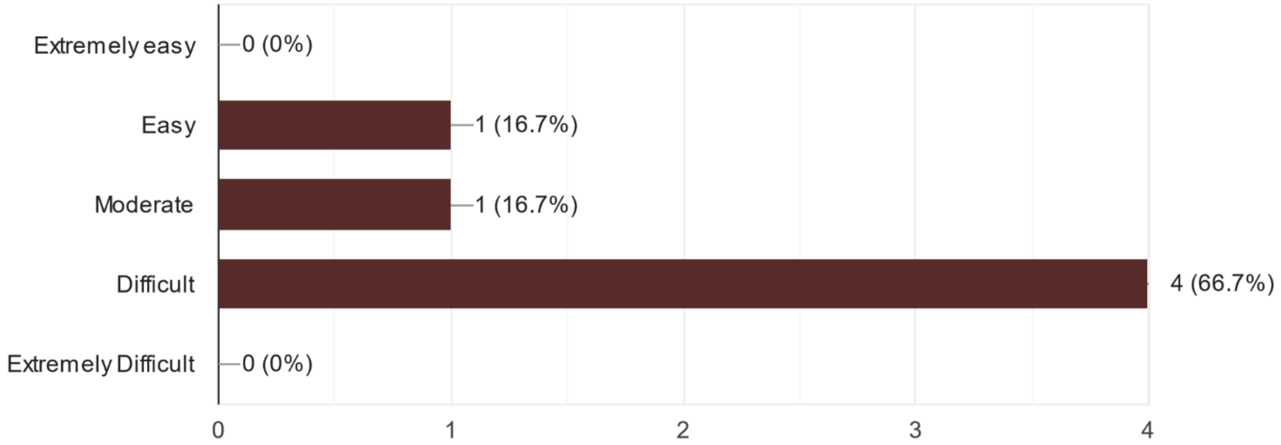
12. Have you or your team undergone any training for using Ansible and Terraform? ?

9 responses



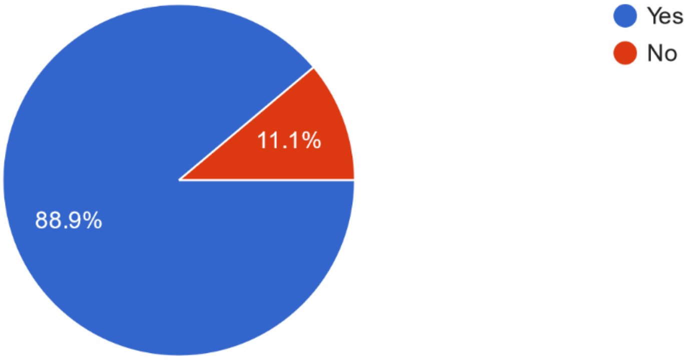
13.In your own opinion how was the training?

6 responses



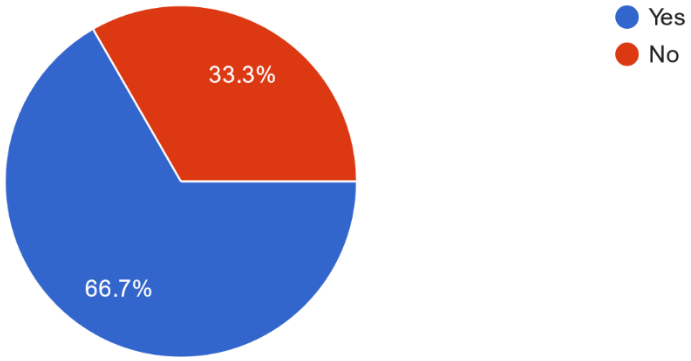
14.Is it efficient to use ansible and terraform when setting up linux servers?

9 responses



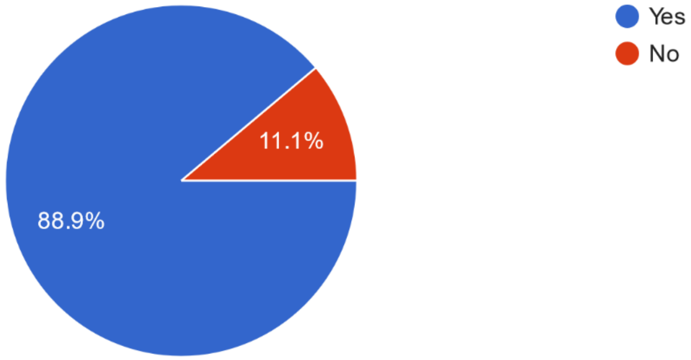
15.Have you faced any challenges or limitations while using Ansible or Terraform?

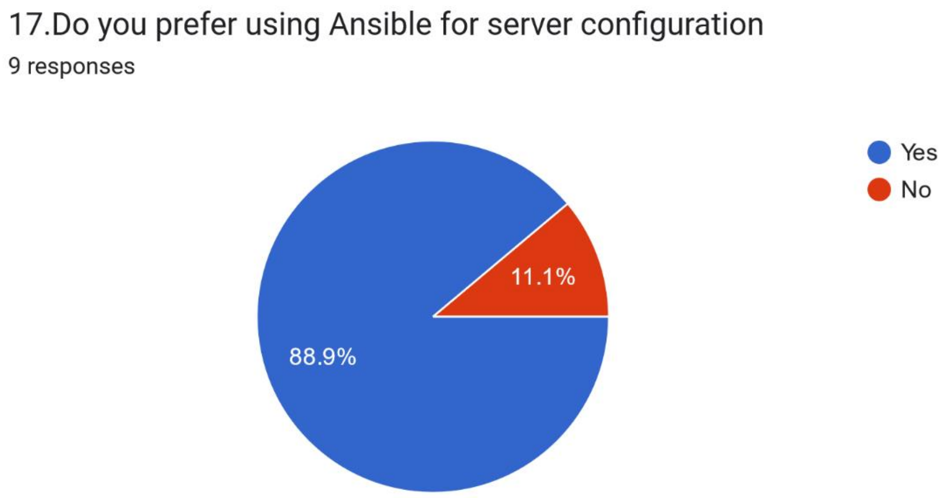
9 responses



16.Do you prefer using terraform for server provisioning

9 responses





17.Do you prefer using Ansible for server configuration

9 responses

From the survey above we can see that getting started with ansible and terraform is difficult. But people in the IT industry are still finding it useful to learn it since it will make life easier for them once they learn it.

References

https://www.redhat.com/en/technologies/management/ansible/what-is-ansible
https://www.redhat.com/en/technologies/management/ansible
https://docs.ansible.com/ansible/latest/getting_started/index.html

https://codefresh.io/learn/gitops/ https://about.gitlab.com/topics/gitops/

https://www.redhat.com/en/technologies/management/ansible/hybrid-cloud
https://www.ibm.com/topics/cloud-computing

https://aws.amazon.com/what-is-cloud-computing/ https://www.linux.com/what-is-linux/

https://developer.hashicorp.com/terraform/intro
https://www.atlassian.com/git/tutorials/gitops

https://www.ibm.com/topics/terraform