**Infrastructure as Code: DevOps approach to deploy and**

**Configure resources on Amazon Web Services**

**by**

**ADITYA BISHT**

# Project submitted in partial fulfillment of the requirements for the degree of Master of Science in Information Sciences and Technologies

**Rochester Institute of Technology**

**B. Thomas Golisano College**

**of**

**Computing and Information Sciences**

**Department of Information Sciences and Technologies**

# 12/7/2017



**Abstract**

“The enabling idea of infrastructure as code that the systems and devices which are used to run software can treated as if they themselves are software” – Kief Morris, Cloud Evangelist.

Imagine a development project where the student developer themselves are coding the required infrastructure over a public cloud hosting service like Amazon Web Services to successfully host the project and iteratively changing the infrastructure as per the project requirements.

The project implements Infrastructure as Code using DevOps approach to configure and deploy various resources over Amazon Web Services. The infrastructure needed to successfully host a WordPress Website like an Apache Server, MySQL Database instance, PHP and Linux environment is coded using software engineering concepts like modularity and reusability using open source tools like Terraform and Ansible.

**Keywords**: Infrastructure as Code, DevOps Methodology, Amazon Web Services, Terraform and Ansible.

**Table of Contents**

|  |  |
| --- | --- |
| TOPICS | Page Number |
| Abstract | 1 |
| Table of Contents | 2 |
| Introduction | 4 |
| Problem | 5 |
| Problem Statement | 5 |
| Motivation to Study | 5 |
| Project Goals | 6 |
| Project Objectives | 6 |
| Literature Review | 7 |
| Methods | 8 |
| Development Scenario | 8 |
| Setting up Prerequisites | 11 |
| Implementation | 12 |
| Results | 13 |
| Evaluation | 17 |
| Reflections | 17 |
| Future Work | 18 |
| Appendix A – Amazon Web Services Terminology | 19 |
| Appendix B – Setting up Environment | 20 |
| Appendix C - Terraform Scripts to implement 1st set of infrastructures requirements. | 23 |
| Appendix D - Terraform and Ansible Scripts to implement 2nd set of infrastructures requirements. | 26 |
| Bibliography | 39 |

|  |
| --- |
|  |

**1 Introduction**

Student developers of RIT who have been following the agile software development methodology[[1]](#footnote-1) for their coding assignments and application development have been shipping out code changes to satisfy new project requirements in very short intervals of time. This has led to a very uneasy situation between the student developers and system administrators as the formers are expected to deliver new releases of their assignment, whereas the latter must keep production servers/systems stable and updated always. It’s been observed that system administrators are reluctant to accept the new underlying network infrastructure changes and tend to consume new code changes slower than developers would like to [2].

Though system administrators have been automating the deployment of middleware and applications to the production environment using bash or shell scripts to accommodate new requirements imposed on the production infrastructure. These types of automations fail to achieve the same level of maturity and quality, incurring an increased risk of compromising the stability of the deployments as they are not being developed using software engineering concepts like modularity and reusability leading servers to a state called Configuration Drift [1].

To overcome the phenomenon of Configuration Drift, student developers need to incorporate the software development methodology called DevOps which is a practice where development and operations engineers work in tandem throughout the software service lifecycle starting from design through the development process to production support. It is put into practice to remove their student developers over dependence from System Administrators [4].

Implementing Infrastructure as Code on public cloud services like Amazon Web Services, often called ‘Cloud Age’ is being considered as a natural advancement from Iron Age where to release a new application, the system administrator had to find some physical hardware like some specific servers on which the application can be deployed. Now in the Cloud Age even without dabbling with screwdrivers, never ending entangled wires, the application developer themselves can create virtual networks, virtual data centers and can provision, extend and tear them down with some software commands.

Programmable infrastructure aims to eliminate~~s~~ or at least hedge the error prone manual provisioning by deploying middleware infrastructure using configuration files by both system administrators and student developers.

**2 Problem**

**2.1 Problem Statement**

Student developers build their software applications in an iterative development in which the application code is designed, developed and tested in repeated cycles but at times the operators (System administrators) responsible for providing the underlying infrastructure required to host that application fails to keep up with the evolving application architecture.

**2.2 Motivation to Study**

Universities such as Rochester Institute of Technology (RIT), offers many undergraduate and graduate courses where students are asked to develop some web or mobile applications and host them over different architectures. In these scenarios, it is common for students to understand the need to iteratively change the underlying infrastructure depending upon the application requirements like the type of Operating System required, TCP/IP protocols to be implemented or to scale up or tear down the entire infrastructure depending upon the estimated web traffic.

To successfully prevail over the aforementioned scenarios, the developers can create their own desirable infrastructure over any public cloud services like AWS by implementing Infrastructure as Code (IaC) in which the entire desired infrastructure is scripted using code blocks and variables. This methodology of treating infrastructure as a piece of code will give developers immense agility to tweak it as per the evolving project requirements.

The capability of developing their own underlying application infrastructure over any public cloud services like Amazon Web Services by implementing it as code will keep developers away from “Shadow IT” in which the current services or infrastructure is different from the required one either in form of version or functionality.

**2.3 Project Goals**

The proposed project’s aim is to implement Infrastructure as Code for deploying and configuring various resources over a public cloud hosting service like Amazon Web Services using DevOps development methodology[[2]](#footnote-2).

**2.4 Project Objectives**

To implement Infrastructure as Code using DevOps development methodology. This will be achieved by scripting the required infrastructure over Amazon Web Services using open source tools like Terraform[[3]](#footnote-3) and Ansible[[4]](#footnote-4). These scripts will be used to deploy and configure resources like a WordPress Website along with its LAMP (Linux instance, Apache Server, MySQL Database Instance and PHP) stack over Amazon Web Services.

**3 Literature Review**

The earliest end user enterprise software applications were nothing but fossilized contraptions that were installed in a very primary environment like operating system, and some specific hardware for embedded systems. The developer’s duties were limited to build and test the software on the predefined infrastructure set up by the system administrator. There was no framework where the application developer could also setup his/her own required infrastructure and tweak it as the project progresses.

In the “Iron age” of IT, only organizations with deep pockets were expected to set up large data centers. The lower cost of deploying middleware infrastructure on public cloud based solutions and virtualization has enabled anyone with a credit card to set up separate development, test, and production servers, a database management system, a storage subsystem, a computation cluster, monitoring, and disaster recovery in a week [2].

Declaring IaC just as simple automation is unfair even though the concept behind IaC is deployment of scripts for middleware technologies to automate the manual processes. The main concept behind IaC is to get scripts that are free of errors, can be redeployed on multiple servers, can be rolled back in case of problems, and can be used by both operations and development teams. According to [1] the best practices of implementing IaC are 1) Deployment and configuration of infrastructure using source control 2) Apply various testing methodologies like unit testing, regression testing on the deployed middleware or infrastructure 3) Usage of scripts as documentation 4) Close cooperation between developers and system administrators or operations team.

Deployment of web applications on public service clouds with maximum optimization requires extensive evaluations of many metrics like costs of deploying, completion of various cloud configurations which tends to be a cumbersome and error prone processes and return on investments (ROI) [2] describes a methodology Cloud Workbench which is substantial implementation of public cloud services. The system defined is based on the notion of Infrastructure-as-Code, which is a state of the art concept to define IT infrastructure in a programmable manner which is reproducible, well-defined, and testable way.

**4 Methods**

**4.1 Development Scenario**

To implement Infrastructure as Code using DevOps methodology over Amazon Web Services. I have stimulated a development scenario where 2 developers have iteratively developed the Terraform and Ansible scripts to spin up the desired Infrastructure needed to successfully host the WordPress Website over AWS. The 2 developers are depicted as 2 different Linux machines both hosted on Oracle’s Virtual Box.

In this project, I have demonstrated the whole process of successfully deploying and configuring the resources for hosting the WordPress Website over Amazon cloud in an agile development process. This is achieved by adopting Continuous Integration and Deployment[[5]](#footnote-5), where these 2 developers have shared their latest Terraform and Ansible Scripts over GIT hub and uses Jenkins[[6]](#footnote-6), to fetch the latest set of scripts from GIT hub.

The 1st developer has developed initial Terraform Script (see Appendix C) to spin up the infrastructure over AWS based upon the initial set of project requirements which are starting an EC2 instance of type Ubuntu, a Virtual Private Cloud (VPC), an open internet gateway and to execute a remote provisioner on the EC2 instance to install Nginx server on it as shown in the code snippet below.

***resource "aws\_instance" "web" {***

***instance\_type = "t2.micro"***

***ami = "ami-fce3c696"***

***# remote provisioner on the instance after creating it to install Nginx.***

***provisioner "remote-exec" {***

***inline = [***

***"sudo apt-get -y update",***

***"sudo apt-get -y install nginx",***

***"sudo service nginx start"***

***]***

***}}***

# Code Snippet 1: Terraform Code snippet highlighting creation of EC2 instance named “web” and executing a remote provisioner to install Nginx Server on it.

Once the infrastructure as per the 1st set of project requirements is implemented over AWS using the Terraform Scripts. The 1st developer pushed them over the shared Git Hub repository using GIT command line utility.

In order to pull the latest, terraform scripts from the shared GIT Hub repository developed by the 1st developer on his/her machine. 2nd developer started a Jenkins job to fetch the scripts from GIT Hub.

The 2nd developer then edited the Terraform scripts and added an Ansible Script to implement the final and ultimate set of project requirements which are to spin up an EC2 instance of type Linux, an open internet gateway, an Amazon Relational Database Instance (RDS), a Virtual Private Cloud (VPC) and installing Apache Server, MySQL database instance and PHP over the Linux EC2 instance thus completing LAMP stack necessary to host the WordPress Website on it as shown in the code snippet below.

***---***

***- hosts: dev***

***become: yes***

***remote\_user: ec2-user***

***tasks:***

***- name: Install Apache.***

***yum: name={{ item }} state=present***

***with\_items:***

***- httpd***

***- php***

***- php-mysql***

***- name: Download WordPress***

***get\_url: url=http://wordpress.org/wordpress-latest.tar.gz dest=/var/www/html/wordpress.tar.gz force=yes***

***- name: Extract WordPress***

***command: "tar xzf /var/www/html/wordpress.tar.gz -C /var/www/html --strip-components 1"***

Code Snippet 2: Ansible Code snippet highlighting the process of installing WordPress website.

The 2nd developer after modifying terraform scripts and adding ansible script has again initialized terraform to build the final infrastructure to successfully deploy WordPress website over Amazon Web Services.

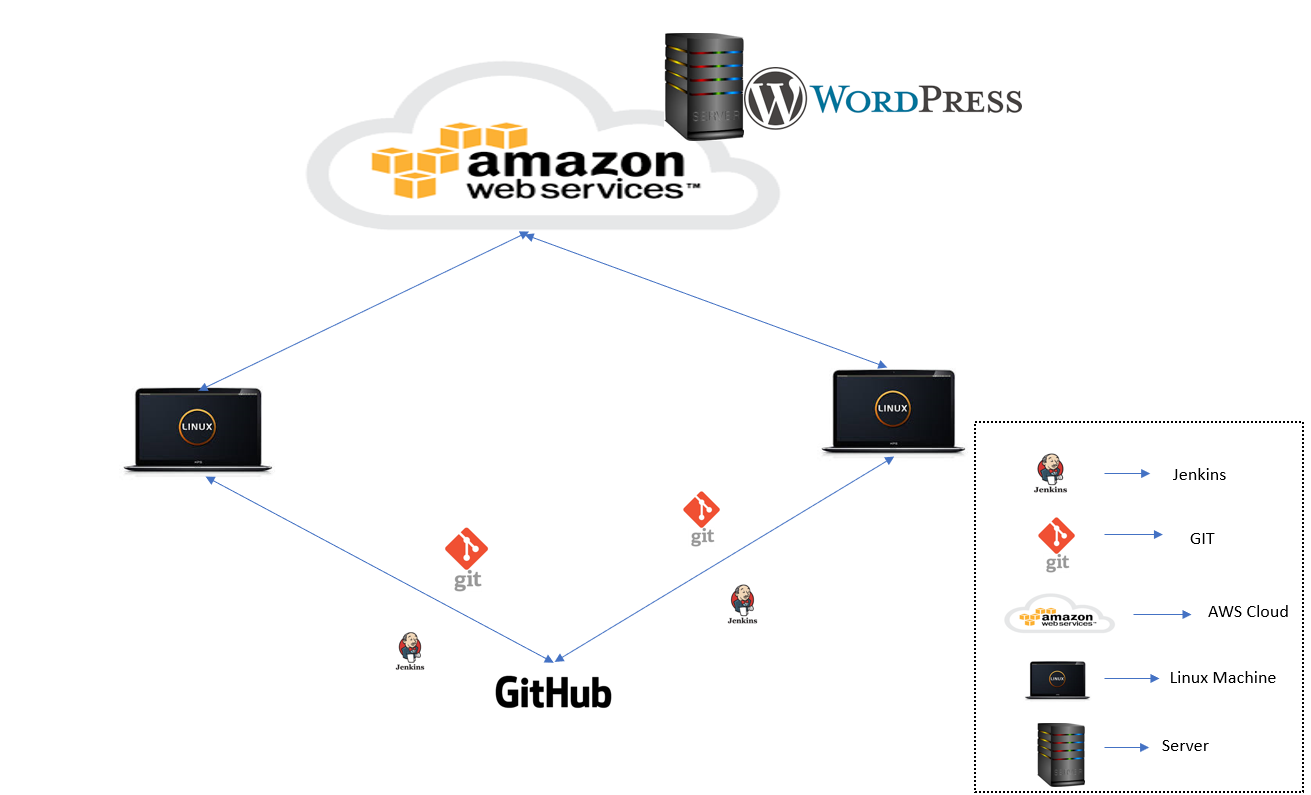
**4.2 Setting up Prerequisites**

The project was started by making an account on Amazon Web Services Account with one default user (Root User). The Root user comes with default permission of ALL ACCESS GRANTED. The Root user (Default User) has default privileges for all the resources that I have built like EC2 instance, Virtual Private Cloud (VPC), an open Internet Gateway, AWS Relational Database Service (RDS), AWS Subnets etc.

GIT Hub and Jenkins were used to demonstrate Continuous Integration and Development as changes to the infrastructure script were pushed over the GIT Hub repository from one virtual Linux machine and were pulled onto the 2nd Linux virtual machines using Jenkins.

The entire ecosystem which included AWS CLI, GIT, Terraform, Jenkins and Ansible on both of the virtual Linux machines along with Java SDK 8.0 and above was installed on both the Linux machines (see Appendix 1).

**4.3 Implementation**



**Figure 1:  Project Architecture highlighting workflow using various tools and technologies.**

As shown in the figure above system architecture, 2 virtual Linux machines were used to develop Terraform and Ansible scripts to automate the infrastructure build for successfully hosting a WordPress Website along with deploying and configuring various resources like Apache Server, MySQL database instance and PHP over AWS. The 2 Linux machines represents 2 individual developers who collaborated in developing those Terraform and Ansible Scripts using GIT Hub and Jenkins.

The 1st developer built an infrastructure over AWS based upon the 1st set of requirements which were to set up an EC2 instance of type ubuntu on which a Nginx server was installed using remote provisioner. Once the infrastructure was up and running the 1st developer pushed those Terraform scripts over a shared GIT Hub repository so that the 2nd developer can pull them and make necessary changes as per the next and final set of project requirements.

The 2nd developer used Jenkins to pull the Terraform scripts from GIT hub repository. The 2nd developer then edited the Terraform scripts for deploying some additional AWS resources like another EC2 instance of type Linux, AWS Security groups, an open internet gateway, Amazon Relational Database instance, both private and public subnets. And an additional Ansible script is added to deploy PHP, MySQL instance and Apache Server on the EC2 instance of type Linux to install and host a WordPress website.

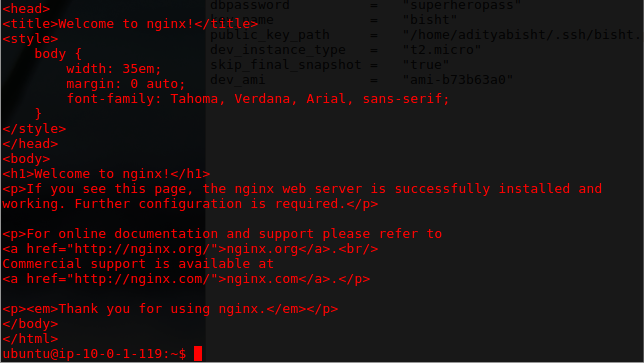
After the successful demonstration of the project Infrastructure as Code: DevOps approach to deploy and configure resources over Amazon Web Services the infrastructure is destroyed through Terraform script only.

**5 Results**

As stated in the Project Goals and Project Objectives, Infrastructure as Code using DevOps software development methodology was implemented by developing Terraform and Ansible scripts for successfully deploying a WordPress Website over Amazon Web Services cloud. A development scenario was stimulated depicting 2 developers working on 2 different virtual Linux machines and collaborating on those scripts using GIT hub and Jenkins.

These scripts were developed and executed in iterations to meet the infrastructure requirements of the project. For the 1st iteration of project development, terraform script had code blocks to add AWS resources like an EC2 instance of type ubuntu, an open internet gateway, a Virtual Private Cloud (VPC) and remote provisoner for installing a Nginx server over that EC2 instance.

Once the infrastructure was built over the AWS cloud, the 1st developer used SSH on the EC2 instance and checked the proper installation of the Nginx server using ***curl localhost*** command as shown in the figure below.



**Figure 2: SSH connection into the EC2 instance type of Ubuntu to verify the successful installation of NGINX server.**

The terraform scripts developed by the 1st developer are then pushed over to the shared GIT Hub repository using GIT command line utility.

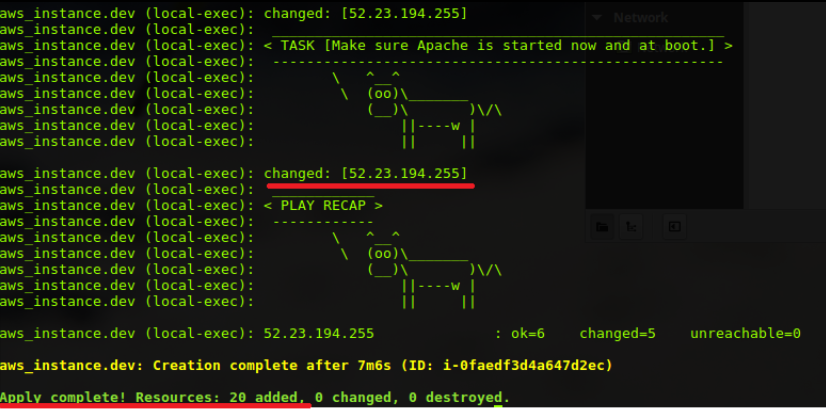
These scripts are then pulled onto the 2nd developer’s virtual Linux machine by starting a Jenkins job as shown in Figure 3 below. The ultimate set of project requirements which is to deploy a WordPress website over Amazon Web Services along with its LAMP Stack (Linux instance, Apache Server, MySQL database instance and PHP) are then implemented by adding some extra code blocks in to the terraform scripts for deploying some required resources like, both private and public subnets, an EC2 instance of type Linux, a Virtual Private Cloud associated with a security group. The edited terraform script had a code block to execute an Ansible script over the EC2 instance once created. This ansible scripts installs PHP, Apache Server, MySQL instance and installs WordPress Website over that EC2 instance.

A screenshot of a cell phone

Description generated with very high confidence

**Figure 3: Jenkins job to Clone/Push the latest code from GIT repository.**

Once the Terraform and Ansible scripts are successfully executed, the 2nd developer then took the IP address of the EC2 instance created from the Terraform Log as shown in the figure 4 below and then accessed the WordPress site hosted on it by putting the IP address on any preferred internet browser as shown in figure 5.



**Figure 4: Log highlighting the IP address of the EC2 instance on which the WordPress website is hosted.**

A picture containing indoor, table, plant, sitting

Description generated with very high confidence

**Figure 5: WordPress welcome page highlighted with the IP address****.**

**6 Evaluation**

According to [4], 60% of infrastructure failures are caused by human error or lack of applying DevOps approach to automate infrastructure build up and it’s also been estimated that the average hourly cost of an infrastructure failure is $100,000 +.

IT Organizations before switching from manual provisioning of infrastructure to using Infrastructure as Code, evaluates the latter methodology on 3 measures:

* Cost Reduction: Infrastructure as Code significantly saves countless manual hours by letting developers only spin up multiple production and test environments over public cloud thus making organizations leaner and financially profitable.
* Faster Execution: Automating infrastructure build up improves the execution time of migrating between different infrastructures as it significantly lowers a key metric called *Mean Time To Recovery (MTTR).* Codifying the infrastructure minimizes the impact of each failure and also minimizes the time to recover from failure.
* Risk: Infrastructure as Code removes the risk associated with human error like manual misconfiguration of servers. Eliminating this decrease sever downtime and increases underlying infrastructure reliability.

In the project**,** Codifying the required infrastructure enabled developers to get more efficient and predictable deployments of various resources. It also provided improved monitoring of those resources as terraform displayed the type and number of AWS resources to be deployed along with their configuration.

Infrastructure as Code can also be perceived as rollback scripts. Since the developers were using GIT Hub as their version control system they could grab any commit since the repository began and execute that script version to restore the state of the infrastructure as it was on any given day.

Rochester Institute of Technology has a graduate course called Database Management and Access which acts as an introductory course to Amazon Web Services cloud. In that course students are assigned host an Oracle Database instance over AWS and then perform Database Administrator tasks over that database. Though the exposure to AWS cloud is very limited in terms of AWS resources to be used but it usually takes around 3 lectures just to successfully setup the AWS infrastructure necessary to host the Oracle database instance. Using Infrastructure as Code can significantly reduce the time to spin up the infrastructure over Amazon cloud and since the students will be using terraform scripts to build the infrastructure it will also bring consistency to the infrastructure meaning every student will have access to the same required Infrastructure thus eliminating the risk associated with manual provisioning.

**7** **Reflections**

All the software’s used in my capstone project are open source software be it Terraform, Jenkins or Ansible and comes with a steep learning curve.

Hosting all these applications on Linux Operating System was a bit challenging due to their inter dependencies. Architecting the solution was also challenging as I had to incorporate Continuous Integration and Continuous Development to execute Infrastructure as Code in a DevOps software development methodology.

Developing the Terraform Script to build the infrastructure over Amazon Web Service was taxing as I had to envision what all infrastructure components like TCP/IP rules, Subnets, Security Policy along with Load Balancer will be required for successful deployment.

The practice of automating the underlying infrastructure or middleware technologies has contributed a lot in bringing the cloud revolution. As just one operation person can start 100s of machines at the execution of a script and can bring them to any desired state and configuration.  This enables the developers to truly leverage the elasticity of the cloud paradigm and disposability of cloud machines.

**8 Future Work**

The future iterations of this project can include to automate the infrastructure on other Public Cloud Hosting Service like Microsoft Azure, Google Cloud or Oracle Cloud.

The WordPress Website can have a real domain name like [www.qwerty.com](http://www.qwerty.com) using Amazon Route 53 which is a highly available and scalable cloud [Domain Name System (DNS)](https://aws.amazon.com/route53/what-is-dns/) web service but can cost significantly to developers.

Much complex applications like multi-tier Mobile Applications or Real Time Data Streaming applications can be deployed onto the public cloud services using other Configuration Management Tools like Chef, Puppet or Docker.

If the developers don’t want to rely on the open source Configuration Management Tools like Terraform, Ansible to automate the infrastructure and later deploy application on it. They have an option to get the Infrastructure build manually on Amazon web services and use AWS Cloud Formation tool to get the script of the infrastructure build. This is somewhat reverse engineering of the AWS infrastructure and can later be used as template for future iterations in infrastructure build.

**Appendices**

**Appendix A – Amazon Web Services Terminology**.

**EC2 Instance:** An EC2 instance is a virtual server in Amazon's Elastic Compute Cloud (**EC2**) for running applications on the Amazon Web Services (AWS) infrastructure.

**Virtual Private Cloud**: A virtual private cloud (VPC) is a virtual network dedicated to your AWS account. It is logically isolated from other virtual networks in the AWS Cloud. You can launch your AWS resources, such as Amazon EC2 instances, into your VPC. You can configure your VPC by modifying its IP address range, create subnets, and configure route tables, network gateways, and security settings.

**Route Table**: A route table contains a set of rules, called routes, that are used to determine where network traffic is directed.

**Amazon EC2 Key Pairs**: Amazon EC2 uses public–key cryptography to encrypt and decrypt login information. Public–key cryptography uses a public key to encrypt a piece of data, such as a password, then the recipient uses the private key to decrypt the data. The public and private keys are known as a key pair.

**AWS Relational Database Service**: Amazon Relational Database Service (Amazon RDS) is a web service that makes it easier to set up, operate, and scale a relational database in the cloud. It provides cost-efficient, resizable capacity for an industry-standard relational database and manages common database administration tasks.

**AWS Security Group**: A security group acts as a virtual firewall for your instance to control inbound and outbound traffic. When you launch an instance in a VPC, you can assign up to five security groups to the instance. Security groups act at the instance level, not the subnet level. Therefore, each instance in a subnet in your VPC could be assigned to a separate set of security groups. If you don't specify a group at launch time, the instance is automatically assigned to the default security group for the VPC.

**AWS CLI**: The AWS Command Line Interface (CLI) is a unified tool to manage your AWS services. With just one tool to download and configure, you can control multiple AWS services from the command line and automate them through scripts.

**Jenkins**: [Jenkins](http://jenkins-ci.org/) is an open-source [continuous integration](http://searchsoftwarequality.techtarget.com/definition/continuous-integration) software tool written in the [Java](http://searchsoa.techtarget.com/definition/Java) programming language for testing and reporting on isolated changes in a larger code base in real time. The software enables developers to find and solve defects in a code base rapidly and to automate testing of their builds.

**GITHUB**: GitHub is a Web-based Git version control repository hosting service. It is mostly used for computer code. It offers all the distributed version control and source code management functionality of Git as well as adding its own features.

**Appendix B – Setting up Environment**.

1. Install AWS CLI

A screenshot of a cell phone

Description generated with high confidence

1. Install BOTO

A screenshot of a cell phone

Description generated with very high confidence

1. Install GIT

A screenshot of a cell phone

Description generated with high confidence

1. Install Terraform

A screenshot of a cell phone

Description generated with very high confidence

1. Install ANSIBLE

A screenshot of a computer

Description generated with very high confidence

1. Create SSH Key

A screenshot of a cell phone

Description generated with very high confidence

**Appendix C - Terraform Scripts to implement 1st set of infrastructures requirements.**

1. **Main.tf**

|  |
| --- |
| provider "aws" { |
|  | region = "${var.aws\_region}" |
|  | profile = "${var.aws\_profile}" |
|  | } |
|  |  |
|  | resource "aws\_vpc" "vpc" { |
|  | cidr\_block = "10.0.0.0/16" |
|  | } |
|  |  |
|  | # Create an internet gateway to give our subnet access to the open internet |
|  | resource "aws\_internet\_gateway" "internet-gateway" { |
|  | vpc\_id = "${aws\_vpc.vpc.id}" |
|  | } |
|  |  |
|  | # Give the VPC internet access on its main route table |
|  | resource "aws\_route" "internet\_access" { |
|  | route\_table\_id = "${aws\_vpc.vpc.main\_route\_table\_id}" |
|  | destination\_cidr\_block = "0.0.0.0/0" |
|  | gateway\_id = "${aws\_internet\_gateway.internet-gateway.id}" |
|  | } |
|  |  |
|  | # Create a subnet to launch our instances into |
|  | resource "aws\_subnet" "default" { |
|  | vpc\_id = "${aws\_vpc.vpc.id}" |
|  | cidr\_block = "10.0.1.0/24" |
|  | map\_public\_ip\_on\_launch = true |
|  |  |
|  | tags { |
|  | Name = "Public" |
|  | } |
|  | } |
|  |  |
|  | # Our default security group to access |
|  | # instances over SSH and HTTP |
|  | resource "aws\_security\_group" "default" { |
|  | name = "terraform\_securitygroup" |
|  | description = "Used for public instances" |
|  | vpc\_id = "${aws\_vpc.vpc.id}" |
|  |  |
|  | # SSH access from anywhere |
|  | ingress { |
|  | from\_port = 22 |
|  | to\_port = 22 |
|  | protocol = "tcp" |
|  | cidr\_blocks = ["0.0.0.0/0"] |
|  | } |
|  |  |
|  | # HTTP access from the VPC |
|  | ingress { |
|  | from\_port = 80 |
|  | to\_port = 80 |
|  | protocol = "tcp" |
|  | cidr\_blocks = ["10.0.0.0/16"] |
|  | } |
|  |  |
|  | # outbound internet access |
|  | egress { |
|  | from\_port = 0 |
|  | to\_port = 0 |
|  | protocol = "-1" # all protocols |
|  | cidr\_blocks = ["0.0.0.0/0"] |
|  | } |
|  | } |
|  |  |
|  | resource "aws\_key\_pair" "auth" { |
|  | key\_name = "${var.key\_name}" |
|  | public\_key = "${file(var.public\_key\_path)}" |
|  | } |
|  |  |
|  | resource "aws\_instance" "web" { |
|  | instance\_type = "t2.micro" |
|  | ami = "ami-fce3c696" |
|  |  |
|  | key\_name = "${aws\_key\_pair.auth.id}" |
|  | vpc\_security\_group\_ids = ["${aws\_security\_group.default.id}"] |
|  |  |
|  | # We're going to launch into the public subnet for this. |
|  | # Normally, in production environments, webservers would be in |
|  | # private subnets. |
|  | subnet\_id = "${aws\_subnet.default.id}" |
|  |  |
|  | # The connection block tells our provisioner how to |
|  | # communicate with the instance |
|  | connection { |
|  | user = "ubuntu" |
|  | } |
|  |  |
|  | # We run a remote provisioner on the instance after creating it |
|  | # to install Nginx. By default, this should be on port 80 |
|  | provisioner "remote-exec" { |
|  | inline = [ |
|  | "sudo apt-get -y update", |
|  | "sudo apt-get -y install nginx", |
|  | "sudo service nginx start" |
|  | ] |
|  | } |
|  | } |

1. **variables.tf**

|  |
| --- |
| Variable"aws\_region"{} |
|  | variable "aws\_profile" {} |
|  | variable "key\_name" {} |
|  | variable "public\_key\_path" {} |

1. **terraform.tfvars**

aws\_profile = "superhero"

aws\_region = "us-east-1"

key\_name = "bisht"

public\_key\_path = "/home/adityabisht/.ssh/bisht.pub"

**Appendix D - Terraform and Ansible Scripts to implement 2nd set of infrastructures requirements.**

1. **Main.tf**

provider "aws" {

region = "${var.aws\_region}"

profile = "${var.aws\_profile}"

}

data "aws\_availability\_zones" "available" {}

# VPC

resource "aws\_vpc" "vpc" {

cidr\_block = "10.1.0.0/16"

}

#internet gateway

resource "aws\_internet\_gateway" "internet\_gateway" {

vpc\_id = "${aws\_vpc.vpc.id}"

}

# Route tables

resource "aws\_route\_table" "public" {

vpc\_id = "${aws\_vpc.vpc.id}"

route {

cidr\_block = "0.0.0.0/0"

gateway\_id = "${aws\_internet\_gateway.internet\_gateway.id}"

}

tags {

Name = "public"

}

}

resource "aws\_default\_route\_table" "private" {

default\_route\_table\_id = "${aws\_vpc.vpc.default\_route\_table\_id}"

tags {

Name = "private"

}

}

resource "aws\_subnet" "public" {

vpc\_id = "${aws\_vpc.vpc.id}"

cidr\_block = "10.1.1.0/24"

map\_public\_ip\_on\_launch = true

availability\_zone = "${data.aws\_availability\_zones.available.names[0]}"

tags {

Name = "public"

}

}

resource "aws\_subnet" "private1" {

vpc\_id = "${aws\_vpc.vpc.id}"

cidr\_block = "10.1.2.0/24"

map\_public\_ip\_on\_launch = false

availability\_zone = "${data.aws\_availability\_zones.available.names[1]}"

tags {

Name = "private1"

}

}

resource "aws\_subnet" "private2" {

vpc\_id = "${aws\_vpc.vpc.id}"

cidr\_block = "10.1.3.0/24"

map\_public\_ip\_on\_launch = false

availability\_zone = "${data.aws\_availability\_zones.available.names[0]}"

tags {

Name = "private2"

}

}

resource "aws\_subnet" "rds1" {

vpc\_id = "${aws\_vpc.vpc.id}"

cidr\_block = "10.1.4.0/24"

map\_public\_ip\_on\_launch = false

availability\_zone = "${data.aws\_availability\_zones.available.names[0]}"

tags {

Name = "rds1"

}

}

resource "aws\_subnet" "rds2" {

vpc\_id = "${aws\_vpc.vpc.id}"

cidr\_block = "10.1.5.0/24"

map\_public\_ip\_on\_launch = false

availability\_zone = "${data.aws\_availability\_zones.available.names[1]}"

tags {

Name = "rds2"

}

}

resource "aws\_subnet" "rds3" {

vpc\_id = "${aws\_vpc.vpc.id}"

cidr\_block = "10.1.6.0/24"

map\_public\_ip\_on\_launch = false

availability\_zone = "${data.aws\_availability\_zones.available.names[2]}"

tags {

Name = "rds3"

}

}

# Subnet Associations

resource "aws\_route\_table\_association" "public\_assoc" {

subnet\_id = "${aws\_subnet.public.id}"

route\_table\_id = "${aws\_route\_table.public.id}"

}

resource "aws\_route\_table\_association" "private1\_assoc" {

subnet\_id = "${aws\_subnet.private1.id}"

route\_table\_id = "${aws\_route\_table.public.id}"

}

resource "aws\_route\_table\_association" "private2\_assoc" {

subnet\_id = "${aws\_subnet.private2.id}"

route\_table\_id = "${aws\_route\_table.public.id}"

}

resource "aws\_db\_subnet\_group" "rds\_subnetgroup" {

name = "rds\_subnetgroup"

subnet\_ids = ["${aws\_subnet.rds1.id}", "${aws\_subnet.rds2.id}", "${aws\_subnet.rds3.id}"]

tags {

Name = "rds\_sng"

}

}

#Security groups

resource "aws\_security\_group" "public" {

name = "sg\_public"

description = "Used for public and private instances for load balancer access"

vpc\_id = "${aws\_vpc.vpc.id}"

#SSH

ingress {

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["${var.localip}"]

}

#HTTP

ingress {

from\_port = 80

to\_port = 80

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

#Outbound internet access

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

}

#Private Security Group

resource "aws\_security\_group" "private" {

name = "sg\_private"

description = "Used for private instances"

vpc\_id = "${aws\_vpc.vpc.id}"

# Access from other security groups

ingress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["10.1.0.0/16"]

}

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

}

#RDS Security Group

resource "aws\_security\_group" "RDS" {

name= "sg\_rds"

description = "Used for DB instances"

vpc\_id = "${aws\_vpc.vpc.id}"

# SQL access from public/private security group

ingress {

from\_port = 3306

to\_port = 3306

protocol = "tcp"

security\_groups = ["${aws\_security\_group.public.id}", "${aws\_security\_group.private.id}"]

}

}

resource "aws\_db\_instance" "db" {

allocated\_storage = 10

engine = "mysql"

engine\_version = "5.6.27"

instance\_class = "${var.db\_instance\_class}"

name = "${var.dbname}"

username = "${var.dbuser}"

password = "${var.dbpassword}"

db\_subnet\_group\_name = "${aws\_db\_subnet\_group.rds\_subnetgroup.name}"

skip\_final\_snapshot = "${var.skip\_final\_snapshot}"

vpc\_security\_group\_ids = ["${aws\_security\_group.RDS.id}"]

}

# key pair

resource "aws\_key\_pair" "auth" {

key\_name ="${var.key\_name}"

public\_key = "${file(var.public\_key\_path)}"

}

# server

resource "aws\_instance" "dev" {

instance\_type = "${var.dev\_instance\_type}"

ami = "${var.dev\_ami}"

tags {

Name = "wordpress-instance"

}

key\_name = "${aws\_key\_pair.auth.id}"

vpc\_security\_group\_ids = ["${aws\_security\_group.public.id}"]

subnet\_id = "${aws\_subnet.public.id}"

provisioner "local-exec" {

command = <<EOD

cat <<EOF > aws\_hosts

[dev]

${aws\_instance.dev.public\_ip}

EOF

EOD

}

provisioner "local-exec" {

command = "sleep 6m && ansible-playbook -i aws\_hosts wordpress.yml"

}

}

1. **Variables.tf**

variable "aws\_region" {}

variable "aws\_profile" {}

variable "localip" {}

variable "db\_instance\_class" {}

variable "dbname" {}

variable "dbuser" {}

variable "dbpassword" {}

variable "key\_name" {}

variable "public\_key\_path" {}

variable "dev\_instance\_type" {}

variable "skip\_final\_snapshot" {}

variable "dev\_ami"{}

1. **Terraform.tfvars**

aws\_profile = "superhero"

aws\_region = "us-east-1"

localip = "0.0.0.0/0"

db\_instance\_class = "db.t2.small"

dbname = "superherodb"

dbuser = "superhero"

dbpassword = "superheropass"

key\_name = "bisht"

public\_key\_path = "/home/adityabisht/.ssh/bisht.pub"

dev\_instance\_type = "t2.micro"

skip\_final\_snapshot = "true"

dev\_ami = "ami-b73b63a0"

1. **WordPress.yml**

---

- hosts: dev

become: yes

remote\_user: ec2-user

tasks:

- name: Install Apache.

yum: name={{ item }} state=present

with\_items:

- httpd

- php

- php-mysql

- name: Download WordPress

get\_url: url=http://wordpress.org/wordpress-latest.tar.gz dest=/var/www/html/wordpress.tar.gz force=yes

- name: Extract WordPress

command: "tar xzf /var/www/html/wordpress.tar.gz -C /var/www/html --strip-components 1"

- name: Make my directory tree readable

file:

path: /var/www/html/

mode: u=rwX,g=rX,o=rX

recurse: yes

owner: apache

group: apache

- name: Make sure Apache is started now and at boot.

service: name=httpd state=started enabled=yes

**Bibliography**

1. Hummer, W., Rosenberg, F., Oliveira, F., & Eilam, T. (2013, December). Testing idempotence for infrastructure as code. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing* (pp. 368-388). Springer, Berlin, Heidelberg.
2. Hüttermann, M. (2012). Beginning DevOps for Developers. DevOps for Developers, 3-13.
3. Jabbari, R., bin Ali, N., Petersen, K., & Tanveer, B. (2016, May). What is DevOps?: A Systematic Mapping Study on Definitions and Practices. In *Proceedings of the Scientific Workshop Proceedings of XP2016* (p. 12). ACM
4. MacVittie, L. (2015, February 25). How Deploy Frequency Impacts Infrastructure Stability. Retrieved December 07, 2017, from https://devops.com/how-deploy-frequency-impacts-infrastructure-stability/
5. Null, C. (2017, April 07). Infrastructure as code is essential to your DevOps practices. Retrieved August 28, 2017, from https://techbeacon.com/infrastructure-code-engine-heart-devops
6. Scheuner, Joel, et al. "Cloud workbench: Benchmarking iaas providers based on infrastructure-as-code." *Proceedings of the 24th International Conference on World Wide Web*. ACM, 2015.
7. Spinellis, D. (2012). Don't install software by hand. *IEEE software*, *29*(4), 86-87.
8. MacVittie, L. (2015, February 25). How Deploy Frequency Impacts Infrastructure Stability. Retrieved December 07, 2017, from https://devops.com/how-deploy-frequency-impacts-infrastructure-stability/

1. Agile Software Development Methodology: The Agile software development life cycle is based upon the iterative and incremental process models, and focuses upon adaptability to changing product requirements. [↑](#footnote-ref-1)
2. DevOps development methodology: It is a software engineering practice that aims at unifying software development (Dev) and software operation (Ops). [↑](#footnote-ref-2)
3. Terraform: It is an infrastructure as code software. It allows users to define a datacenter infrastructure in a high-level configuration language, from which it can create an execution plan to build the infrastructure in a service provider such as AWS. [↑](#footnote-ref-3)
4. Ansible: It is a radically simple IT automation platform that makes your applications and systems easier to deploy. [↑](#footnote-ref-4)
5. Continuous integration: When application developers merge their script changes back to the shared repository as often as possible.

   Continuous delivery: It is an extension of continuous integration to make sure that the latest changes to the application are quickly available for usage. [↑](#footnote-ref-5)
6. Jenkins: It is an extensible automation server and can be used as a simple Continuous Integration server or turned into the Continuous Delivery hub for any project. [↑](#footnote-ref-6)