Platforms for automatic server configuration

1.1.1 What is the concept of Infrastructure as Code (IaC)?

Put it simple infrastructure as code or IaC handles creation/destruction/changes of infrastructure by using code instead of manually configuring VMs and other server resources in the cloud. When using IaC a config file (or multiple config files) the definition of what is supposed to be created (and how) is defined in the configuration files and is thus easier to maintain, reuse and understand.

1.1.2 The problems that IaC aims to solve and/or benefits that IaC brings to a software project

Self documented - When deploying infrastructure through code using different configuration files, what is actually created and how the infrastructure is configured becomes self documented through the configurations, meaning that if an external developer skilled in the syntax of the tools used would read the configurations, he/she would make sense of what is going to happen/how the infrastructure is going to be configured.

Version controlled - Since the users are writing how to structure/create the infrastructure using configuration files, these files can be version controlled using i.e github/gitlab or any other similar tool to enable, among other things, change history (who changed what and did all test pass related to the change), branching and merging (allowing several users to work on the same infrastructure), traceability (with tools like Jira and other devops tools).

Faster/easier - Using IaC it will become easier and more automated to create/rebuild infrastructure. It would also enable abstracting away *how* to actually perform the actual building which becomes the responsibility of the tools used (i.e terraform) to create the infrastructure. Since in IaC it is the responsibility of *applications* to do the actual heavy lifting, there are no dependencies to i.e IT personnel creating/configuring resources. Furthermore, using IaC would enable the flexibility of recreating/destroying/moving/resizing infrastructure, which is one example that has happened at my current workplace where we used Terraform to move our current infrastructure hosted at a local Azure instance to another tenant/cloud with more or less just a single command.

Consistent - By using a declarative approach (more on that below) the end state is defined, which means that you end up with the same state no matter the way which was taken to get there. If you deploy/destroy several times, in theory it should be created to the same state every single time. Furthermore, since tracking state enables you to only change differences, whenever someone does a manual change to some part of the infrastructure, it would be easy to get back to the defined end state in the configuration.

1.1.3 Two types of languages to write IaC.

In general there are two types of approaches to writing IaC code. Either they use a Declarative programming language (functional), meaning that as a user you define the end state, i.e how you want the environment to look like in the end. Terraform, Pulumi and Bicep are all tools that use a declarative approach.

The other way to write IaC is using an Imperative programming language, where instead of defining the end state, you define which steps need to be executed in order to reach the state you want. Chef is an example of a tool using the imperative approach and while being mostly declarative, Ansible provides some support for writing imperative code as an example.

These types all come with both advantages and disadvantages. I.e a Declarative programming language is repeatable, meaning it doesn't matter how many times the code is run, it will reach the same end result. At the same time by using a declarative programming language you give up control (i.e which steps are needed to get to a certain point). An Imperative programming language requires more scripting knowledge and a planned defined path to take (i.e doing things in the right order), while at the same time giving you more freedom to configure in the way you think is best.

1.2.1 Describe Terraform and the problems that it is constructed to solve.

Terraform is an IaC application developed by HashiCorp that tries to simplify provisioning infrastructure by abstracting away how configuring/provisioning is performed.

The user simply adds configurations defined in the “terraform programming language” to one/or many configuration files to define which end state they want to end up at. Terraform reads the configuration (and which provider(s) are stated), checks the current state of the infrastructure to define what needs to be destroyed/changed/created and creates an execution plan by translating the configuration to different api calls that need to be performed to what infrastructure provider (and in what order).

As far as I understood from terraforms documentation they have developed handling of their own to different providers as well as allowed for the community to provide their own handlings. This enables terraform to communicate to over 1700 providers, making it very flexible, but also a bit of a jungle (i.e have I selected the best provider handler out of the ones that do exist).

1.2.2 Provide a list of Pros and Cons of Terraform according to your own experience using it.

- The programming language was simple and syntax was intuitive. However it would be interesting to try out i.e Pulumi which has a strength over Terraform in which it can be configured using many of the popular programming languages.
- One of the most challenging parts of using Terraform was to find proper documentation on how to do certain actions. Many times I had to guess what I needed to do, or to look at external learning material in order to properly set up everything using Terraform.
- When performing my provision of the 1.4 task I created a temporary resource to act as my personal computer. However since Terraform is using a descriptive language instead of Imperative, I couldn’t create temporary resources in a good way (ie. destroying the “new computer” i created to provision after Terraform finished executing all actions in the configuration).

1.3.1 Describe Ansible and the problems that it is constructed to solve

In general Ansible is an open source tool that can be used for many things, among other things, automated provisioning (similar to Terraform), configuration, orchestration and deployment. In this assignment Ansible was used to orchestrate different playbooks in order to automate the process of installing/configuring nginx on a server.

Provisioning/Configuration - Similar to Terraform the desired state of one's infrastructure and applications can be defined and automatically maintained by using Ansible.

Orchestration - As mentioned above, orchestration can be used to ensure that different applications get installed/configured in different hosts at different times.

Deployment - Using i.e a CI/CD pipeline some of Ansibles features enables the automatic running of different tasks to make sure everything is done in the same manner throughout every deployment.

1.3.2 Provide a list of the Pros and Cons of Ansible according to your own experience using it.

- When using Ansible I was surprised there was no “easy” way to install/use it with my Windows machine. This either forces you to use a linux machine, an emulator, to configure differently, or to use another application altogether to perform configuration.
- I used Ansible together with Terraform and once I understood how and what Ansible needed to properly operate it was quite easy to adapt and use the two together. I.e I created inventory/variable files with the ips that terraform creates to use with Ansible.
- I like the idea of configuring all the files locally and have Ansible replace these when doing the provisioning. This makes it in my opinion easier to maintain.
- It was generally easy to manage things using yaml files, as this format is commonly used. However the hardest part was getting used to indentation being important.

1.4 The following considerations were made to setup.

- I didn't have a Linux computer and didn't opt into using an emulator for Windows. As such I create a VM to use as my "personal computer" for the sake of provisioning the machines in the environment.
- This is the reason why a copy of the .pem file is made to the new VM to act as the provisioning computer. There are other ways to do this as stated below, but due to time constraints I did not pursue these options.
- If I would have had a Linux computer with Ansible pre-installed, the following sections in the provisioning.tf file could have been removed

```
connection {
    user        = "ubuntu"
    private_key = file(var.identity_file)
    host        = openstack_networking_floatingip_v2.fip_1.address
  }

  provisioner "remote-exec" {
    inline = [
      "echo Updating system and installing Ansible...!",
      "sudo rm -r /var/lib/apt/lists/*",
      "sudo apt -q update",
      "sudo apt install -q ansible -y",
      "sudo ssh-keyscan -H
${openstack_networking_floatingip_v2.fip_2.address} >> ~/.ssh/known_hosts",
      "sudo mkdir ~/ansible/",
      "sudo chmod -R 777 ~/ansible",
      "echo Finished setting up Ansible"
    ]
  }

  provisioner "file" {
    source      = "../ansible/"
    destination = "/home/ubuntu/ansible/"
  }

  provisioner "file" {
    source      = var.identity_file
    destination = "/home/ubuntu/ssh.pem"
  }

  provisioner "remote-exec" {
    inline = [
      "sudo chmod -R 600 /home/ubuntu/ssh.pem",
      "ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook -u ubuntu -i
~/ansible/inventory --private-key /home/ubuntu/ssh.pem
~/ansible/ansible-master.yaml",
      "echo Finished running Ansible Playbooks!"
    ]
  }
```

- And replaced with a local-exec section as suggested below (please note that the below is untested but provides an example):

```
provisioner "local-exec" {
    command = "ssh -i ${var.identity_file}
ubuntu@${openstack_networking_floatingip_v2.fip_1.address}
ANSIBLE_HOST_KEY_CHECKING=False ansible-playbook -u ubuntu -i
../ansible/inventory --private-key ${var.identity_file}
../ansible/ansible-master.yaml"
  }
```

- Another approach could be to have Terraform create local SSH keys and install these on a “provisioning machine” (private key) as well as each machine that needs to be provisioned/configured (public key), in order to only allow the provisioning machine access to the machines needing configuring.
- I also opted for letting terraform create ansible variable files as well as inventory and also be responsible for running the ansible commands (instead of using the bash script for this).

Continuous Delivery

2.1.1 The problems that Continuous Delivery aims to solve

CD is an approach where the development teams tries to shorten the deployment cycle by ensuring quality, speedy deliveries and a decreased manual work. It does this through building, testing and releasing with increased speed and frequency through automation and automated tasks/testing.

2.1.2 The benefits of applying Continuous Delivery to a software project.

Some of the benefits of using CD in development is improved productivity (i.e perform automated testing and tasks allowing developers to focus on development), increased quality (running tests for a feature developed a long time ago to ensure it still works when implementing a change of a newer feature) as well as a faster feedback loop from end users (i.e the development from start to release will be sped up through the use of CD, and thus the development team can receive feedback from end users faster).

2.1.3 Four of the basic principles necessary to adopt in a software project to do “Continuous delivery” successfully.

The different principles according to Atlassian are “Repeatable reliable process”, “Automate everything”, “Version control”, “Build in quality”, “Do the hardest parts first”, “Everyone is responsible”, “Done means released” and “Continuous improvement”. These differ a bit from the principles mentioned by https://continuousdelivery.com, but as the list form Atlassian is more extensive I have opted to explain the below a bit further.

Repeatable reliable process -Throughout an applications development life cycle, performing the same tasks over and over again (i.e through an automated script or deployment pipeline) is one of the key points to successfully adopt continuous delivery. The other key point in this area is a reliable process, meaning that it doesn’t matter if the change is made to production, stage or the development environment, the same principles/tasks are run independent of the environment. This would mean that the same tests are run in development and in production and would reliably catch issues before deployment. If one test is run in i.e development but is skipped in stage errors could slip through.

Automate everything - Time is money in a project and things that can be automated should be automated as far as possible. A task is not fully repeatable and reliable until it is automated, since there is always the risk of human error and deviating from areas of the task to be executed.

Version control - One of the most important tools used in any development project. Since this allows the consolidation of a common codebase, the ability to undo changes, cherry pick on which features gets released next and many more features.

Build in quality - According to Atlassian quality is not an afterthought but something that should be baked into every delivery to end users through automated testing making sure that the code is free from bugs/errors and is meeting clients/users expectation. Another important aspect is closing the feedback loop from end users and implementing change to the process.

2.1.4 The difference between Continuous Integration, Continuous Delivery, and Continuous Deployment

Continuous Integration - Is the process of automatically validating development changes by creating builds and running automated tests, to ensure that there are no integration issues when merging different areas of committed code to a master/release branch. In order for CI to work properly it requires the use of automated tests (i.e unit tests) in a majority of the application, to ensure that changes to one part doesn't break another part of the integration.

Continuous Delivery - Takes the process one step further from CI by including building a release candidate that can be tested and or released (however usually requiring manual input in the form of an approval or similar for it to go the whole way to production)

Continuous Deployment - Is similar to Continuous Delivery through the fact that the delivery is often automatic meaning there is no manual input to release the application all the way to the production environment.