1.1

Infrastructure as code (IaC) is a newer way of configuring computer and network infrastructure [1]. IaC makes it possible to configure servers and networks without using physical hardware. Instead, cloud computing has become more common where hardware and network configurations are provisioned dynamically without any physical hardware.

IaC creates the infrastructure using source code [1]. The source code creates exactly the same infrastructure every time it is executed. This avoids snowflake servers, which are servers that have not been configured in exactly the same way as the others. This is a common problem with manual configuration. IaC, therefore, becomes a less error-prone way to configure infrastructure.

One of a few common IaC practices is the use of definition files [1]. These are the executable configuration files that are used to provide the infrastructure and its configurations. It is also important to configure everything in these files since manual configuration on the servers creates snowflake servers that do not have the same configuration as the definition files create.

Another common practice is the use of self-documentation. This is the process of using code (configuration files) as the documentation instead of human-readable documentation [1].

Using version control such as git for the configuration files is also a common practice [1]. This is used in order to keep a history of all changes to the infrastructure configuration files. This makes it easier to audit changes and find potential issues. It is also used in order to be able to create the infrastructure using the latest configuration.

It is also important to make small configuration changes [1]. Because errors from small changes are easier to find, it is recommended to make changes more often rather than combining a lot of configuration changes into one big update.

Lastly, it is a common practice to use tools that prevent systems from downtime during configuration changes [1]. For example, BlueGreenDeployment and ParallelChange are tools that allow smaller configuration changes without downtime.

The main problem that IaC aims to solve is to enforce consistency [2]. If IaC were not used every server would have to be configured manually according to manual instructions. Over time each server configuration would drift away from the manual instructions causing snowflake servers. These are servers that can not be recreated with exactly the same configuration. This can especially be a problem during deployment when another server is used in order to deploy an application that has only been deployed to a development server that has become a snowflake server

A benefit of using IaC is that developers do not have to manage infrastructure manually [3]. Instead, they can use templates to provision infrastructure for the applications they are working on. Another benefit is that time and money are saved when companies do not have to manually configure physical hardware anymore. Instead, virtualization in the cloud can be used with automated configurations. Some other benefits are the faster speed of deployment, fewer errors due to automated configurations and consistent infrastructure, and avoiding snowflake servers (servers with configuration drift).

IaC configurations are always idempotent [2]. This means that all infrastructure created with the same configuration files does always have exactly the same configuration after being created.

Creating test environments is a benefit that comes from IaC's idempotence [2]. It makes it easy to provision one or many test environments for applications during development.

IaC is also a benefit for DevOps, continuous integration, and continuous delivery [3]. The infrastructure becomes automated which removes work from developers and sysadmins. Without IaC sysadmins would have to provision infrastructure manually. CI/CD helps development teams and operations teams configure deployments in the same way both in development and deployment using IaC automation. This process avoids problems and inconsistencies that would occur without automation.

There are two types of languages used to write IaC [3]. The first type is a declarative language. Declarative languages describe the resources and configurations that the IaC tool should create. Because the state of the infrastructure is stored as a list it can also be removed easier compared to other types of languages. There are multiple declarative languages used for IaC [2]. Some declarative languages used by IaC tools are YAML, JSON, and XML.

Imperative languages are another type of language used by IaC. [3]. These types of IaC languages use commands that have to be run in a specific order instead of describing the infrastructure like declarative languages. Imperative languages make changes a lot harder because the user has to figure out how to implement the change. Declarative languages can make changes for the user.

1.2

Terraform is an infrastructure as code tool that is used in the cloud and on in-house servers [4]. The purpose of the tool is to configure infrastructure using code. Terraform can configure resources such as server hardware, storage, and networks.

Terraform can work with almost any cloud provider [4]. In order to work with a cloud provider Terraform needs a Terraform provider. A provider is a layer between Terraform and the cloud provider API, making it possible for Terraform to work with the cloud provider. The providers are created by the community and there exist thousands of different providers.

The Terraform workflow is split into three stages [4]. The first stage is the “write” stage. In this stage, infrastructure is defined by writing code in configuration files. For example, a virtual machine and a network could be configured for a web server.

The “plan” stage is the second stage [4]. In this stage, Terraform describes all changes that will be done to the infrastructure. This could be creating, updating, or destroying resources in the infrastructure.

The last stage is the “apply” stage. This is the last stage where all the planned changes are applied. Terraform is responsible for executing all changes in the correct order and not breaking any dependencies.

Terraform has many benefits [4]. Firstly it manages infrastructure making it easier to update. It makes changes safer by having a planning stage where all changes are listed before they are applied. The declarative configuration makes it easier to do changes compared to the imperative configuration where changes have to be written in the correct order. Another benefit is that Terraform has optional standardized configuration modules that can be used to make configurations faster. Lastly, Terraform configuration files can be shared and maintained by a version control system making them accessible by everyone publicly or in a team.

- Pros
    - Deploys configured servers faster than manual configuration.
    - Easy to recreate infrastructure.
    - Easier to make changes.
    - Compatible with most cloud providers.
- Cons
    - Unnecessary if an infrastructure configuration is only needed once and never again.

1.3

Ansible is used to manage the state of remote servers [5]. It consists of three parts. The first part is the Control node. This is the system which the Ansible application is run from. The second part is the Managed node. This is a system that Ansible manages. The last part is the Inventory which exists in the same system as the Control node. The inventory is used to list managed nodes in different categories.

Playbooks are collections of configurations that are applied to the managed nodes (server) [5]. A playbook consists of plays. These are things to execute. For example, tasks to run or variables. The playbooks are written using the YAML syntax [6].

These playbooks can be used to configure many servers [6]. For example, a playbook can execute the configuration on multiple managed nodes at once or be reused in the future.

Each play (task) in the playbook is executed from the top to the bottom [6]. If multiple managed nodes are defined in the inventory each task is executed on all the managed nodes before moving to the next task. If a task does not succeed on a managed node it will not get more tasks executed and the managed node will have to be recreated or manually investigated.

The benefit of using a configuration management tool like Ansible is to be able to configure one or many servers to the exact same state. The configuration files can be stored in version control and be reused or updated when needed. It is also faster to run a command to execute code instead of manually configuring hundreds of servers.

- Pros
    - Can configure many servers to the same state without doing it manually.
    - Configuration can be reused.
- Cons
    - It can take more time to write a playbook than configure one server manually if the same configuration is not going to be used again.

2.1

Continuous delivery is a continuation of continuous integration [7]. In continuous integration, the goal is to automate the process of building and testing the codebase in the main branch. This is done in order to make sure that the new changes do not affect something else in the application.

Continuous delivery begins after the codebase has been built and tested by continuous integration [8]. Continuous delivery manages a codebase (branch) that is always ready to be deployed to the production environment. The codebase that is managed by continuous delivery can then easily be deployed when needed.

There are a few things that continuous delivery aims to solve [7]. The process of deploying the application becomes easier. This is because of the automation of an always deployment-ready codebase. Without continuous delivery, the development team would have to create releases manually.

Another benefit of continuous delivery is the ability to release new versions of the application more often [7]. Because the continuous delivery codebase always is up to date it becomes easy and fast to deploy the new version to the production environment. This does also help with getting feedback from the users faster due to the more frequent releases.

Lastly, it helps to make it easier to make small changes [7]. This is because the process to deliver the application with a small change is automated. Compared to a manual process where the application would have needed to be checked, tested, and merged manually each time someone makes a small change.

There are many benefits of applying continuous delivery. It lowers the risk of releases since there is always a tested and deployment-ready codebase [9]. Manual work such as integration and testing is replaced with automated processes. This makes the time to market faster and it can save up to months of work. When the integration and testing phase becomes automated the developers have to work with it every day when making changes and merge requests. Since the testing is automated it does also help with higher quality code. This gives the developers more time to do higher-level testing and user research. The developers can do this by building new pipelines. Costs are also reduced due to automated delivery which reduces the amount of manual work. This is both during development and during its lifetime. The applications do also become better due to more feedback. This is due to the more frequent releases which makes it possible for users to give feedback continuously. Lastly, development teams become happier. This is because of the easier way to work using automation. Developers also get to work with the users more. This helps with feedback and they can work directly with the users more.

There are a few different basic principles for continuous delivery. The first principle is build quality in [10]. This is the process of working with feedback loops that find problems as fast as possible instead of finding problems later. For example, if we find a problem manually then we should also try to figure out a way to find the same problem automatically by using a unit test.

Another principle is to work in small batches [10]. In continuous delivery, the goal is to add an update to the deployment-ready codebase as fast as possible. In traditional software development, this could take months compared to minutes using continuous delivery if the pipeline does not find any issues. If there are issues found the developer will get feedback immediately when the pipeline fails. This reduces the time it takes to get feedback from weeks or months to minutes.

Using computers to perform repetitive tasks and people to solve problems is also a principle of continuous delivery [10]. This principle is used to avoid unnecessary repetitive work for developers. Instead of wasting time doing the same thing often a computer can do it instead. This does also help with reducing errors. Because humans are more likely to create errors when they are doing basic and repetitive tasks.

Relentlessly pursuing continuous improvement is a principle about always trying to improve things [10]. The process of improvement should be a part of a developer's daily work. Because there is always something that can be improved.

Lastly, everyone is responsible is a principle about the developer's responsibility to the software project [10]. Developers should not think that something is someone else's problem. Instead, they should work together as a team. There should not be someone that makes local changes that affect the entire project or organization. This could happen if management is not good or if there are incentives that reward bad behavior. For example, if developers get rewarded for the amount of code they find.

Continuous integration, delivery, and deployment are three development practices [7]. Continuous integration is the practice of merging code to the main branch often. Every merge is automatically built and tested. This is done in order to validate that there are not any integration issues when merging into the main branch.

Continuous delivery is a continuation of continuous integration [10]. Continuous delivery automates the release process after the staging and testing process. The application can then be released when needed by pressing a button. It can be daily, weekly, or when needed. Continuous delivery consists of four parts. Acceptance testing, deployment to staging, deployment to production, and smoke tests.

Continuous deployment is similar to continuous delivery [10]. Instead of needing manual user input in order to deploy the application, it happens automatically every time the pipeline is run successfully.

References:

[1] “InfrastructureAsCode”, *martinfowler.com*. [Online]. Available:https://martinfowler.com/bliki/InfrastructureAsCode.html.

[2] “What is infrastructure as code (IaC)?”, *Microsoft*. [Online]. Available:https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code.

[3] “What is infrastructure as code (IaC)?”, *Red Hat*. [Online]. Available:https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac.

[4] “What is Terraform?”, *hashicorp*. [Online]. Available:https://developer.hashicorp.com/terraform/intro.

[5] “Ansible concepts”, *Ansible*. [Online]. Available:https://docs.ansible.com/ansible/latest/user_guide/basic_concepts.html.

[6] “Ansible playbooks”, *Ansible*. [Online]. Available:https://docs.ansible.com/ansible/latest/user_guide/playbooks_intro.html.

[7] “Continous integration vs. delivery vs. deployment”, *Atlassian*. [Online]. Available:https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment.

[8] “What is CI/CD?”, *Red Hat*. [Online]. Available:https://www.redhat.com/en/topics/devops/what-is-ci-cd.

[9] “What is Continous Delivery?”, *Continuous Delivery*. [Online]. Available:https://continuousdelivery.com/.

[10] “principles”, *Continuous Delivery*. [Online]. Available:https://continuousdelivery.com/principles/.