Tampere University

Tommi Aalto

FLEXIBLE ENVIRONMENT PROVISIONING WITH INFRASTRUCTURE AS CODE TOOLS

Master's thesis
Faculty of Information Technology and Communication Sciences
Examiners: Kari Systä, Terhi Kilamo
September 2022

ABSTRACT

Tommi Aalto: Flexible environment provisioning with Infrastructure as Code tools
Master's thesis
Tampere University
Master's Programme in information Technology
September 2022

Infrastructure as Code is a method where automation is extended from already popular DevOps methods also to manage IT infrastructure. Automation in fundamental role when software is desired to be delivered to the customers faster than before and without additional human interaction.

Now the automation is extended to management of servers, to enable more efficient use of employee's resources and to reduce the dependency of software developers from other teams of the organization, enabling more agile software development and testing.

Infrastructure as code consist of multiple sectors of infrastructure management, like configuration management and orchestration, which are introduced through literature review. In this thesis the main topics are server provisioning, and installation and configuration of the services. For these purposes sufficient tools and their usage are introduced.

In this thesis, virtual machines are provisioned to a virtual datacenter and software is installed and configured to those VMs. The goal is to create re-usable Terraform and Ansible configurations, that can then be used to, flexibly and with low effort, create new environment to support testing and quality assurance. At the same time automatic deployment is enabled for individual components straight from pipeline. The done work also enables Infrastructure as Code solutions to be used in Production environment in the future.

Keywords: Infrastructure as Code, Terraform, Ansible

TIIVISTELMÄ

Tommi Aalto: Kettärä ympäristön provisiointi infrastruktuuri koodina -työkaluilla
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Syyskuu 2022

Infrastruktuuri koodina on menetelmä, jossa automaatiota ulotetaan jo tavaksi muodostuneista DevOps käytännöistä myös IT infrastruktuurin hallintaan. Automaatio on keskeisessä osassa, kun ohjelmistoja halutaan toimittaa loppukäyttäjälle yhä nopeammin ja ilman ylimääräistä ihmisten vuorovaikutusta.

Nyt myös palvelinympäristöjen hallinnassa halutaan ottaa käyttöön automaatiota, jotta työntekijöiden resursseja voidaan käyttää tehokkaammin ja ohjelmistokehittäjien riippuvuus muista organisaation osista vähenee mahdollistaen ketterämmän ohjelmistotuotannon ja testaamisen.

Infrastruktuuri koodina käsittää useita erityyppisiä infrastruktuurin hallintaan liittyviä osa-alueita, kuten konfiguraatioiden hallintaa tai orkestrointia, joihin tutustutaan kirjallisuuden kautta. Tässä työssä pääaiheena on palvelunympäristö provisiointi ja palveluiden asentaminen ja konfigurointi. Tätä varten työssä tutustutaan sopiviin työkaluihin ja niiden käyttöön

Tässä työssä toteutetaan virtuaalikoneet virtuaalipalvelimelle sekä asennetaan että konfiguroidaan sovellukset luoduille virtuaalikoneille. Tavoitteena ovat uudelleenkäytettävät Ansible ja Terraform konfiguraatiot, joilla ketterästi luodaan testauksen ja laadunvarmistuksen tueksi valmiita ympäristöjä nopeasti ja pienellä vaivalla. Samalla mahdollistetaan automaattinen asentaminen yksittäisille komponenteille suoraan työjonosta. Työ mahdollistaa myös tulevaisuudessa Infrastruktuuri koodina käytäntöjen laajentamisen myös tuotantoympäristöihin.

Avainsanat: Infrastruktuuri koodina, Terraform, Ansible

PREFACE

Thanks to OpenText and the people involved for the possibility to do this project as a master's thesis for the company. Thanks to thesis supervisor Kari for all the patience and help on the way. Special thanks to my parents, Monika, and Jouni for pressure and support throughout the studies and thesis project.

Tampere, 14 September 2022

Tommi Aalto

CONTENTS

LIST OF SYMBOLS AND ABBREVIATIONS

SDLC	Software Development Lifecycle
QA	Quality Assurance
DevOps	Development and Operations
CI	Continuous Integration
CD	Continuous Deployment / Continuous Delivery
VM	Virtual machine
vApp	Virtual application, vCloud Director component
IaC	Infrastructure as Code

1. INTRODUCTION

Infrastructure as Code is a new way of doing IT infrastructure management. Usually, software companies have separate teams for infrastructure management, and the environments are built manually after careful consideration of the requirements. However, this thesis describes a modern approach to infrastructure management, whereby using modern tools, the environment is configured in code, and changes to the environment are applied by the tools automatically. The responsibility of building the environment is now on also in the hands of development team instead of only being a job of a separate infrastructure management team or operations.

The first implementation of IaC is building a quality assurance (QA) environment for the product, to support DevOps practices and the software development lifecycle (SDLC). The new QA environment is an essential part in the evolution of the platform and ensuring that the code in the Production environment is reliable. All new code will be deployed to QA prior to production. After that, new environments for development, testing or quality assurance purposes can be set up flexibly with the configuration used for QA environment provisioning.

To accomplish the above, this thesis introduces two state-of-the-art tools for Infrastructure as Code solution: Terraform and Ansible, and both provide different view for the process. Terraform is infrastructure management tool that is used to build the virtual machines and other logical and physical components of the platform. Ansible is configuration management tool, that is used to deploy and configure software on virtual machines created with Terraform. Additionally, CI/CD pipeline is built for each tool using GitLab CI to take infrastructure automation even further.

In this thesis, Infrastructure as Code solutions are introduced and discussed. Prior research is used to take multiple points of view into account in the design process, and to evaluate the possible benefit of DevOps practices. Different options for the work are discussed and the tools are introduced. Benefits of the used solutions are evaluated against the traditional or alternative practices of managing infrastructure. The requirements of the environment are set, and the first implementation of IaC is built to the VMWare vCloud Director environment using Terraform. Also, the software is deployed and configured to the created environment using Ansible. Later, the environment can be expanded following the learned processes and practices.

The objective of this thesis is to find out if Infrastructure as Code solution can help testing by automating environment provisioning with minimal effort. The purpose is also to describe the process of the development and to provide a hands-on example of the work. This thesis clarifies if a legacy platform can benefit from a newly built QA and testing environment and analyses how the Infrastructure as Code solutions are affecting the development process.

This thesis describes the target product, ANI Platform, in the next chapter. Through the introduction, the current state and challenges are evaluated, and desired changes are introduced. Project scope is also mitigated in chapter two. Chapter three introduces Infrastructure as Code through DevOps. Different types of IaC are presented. Terraform and Ansible are introduced in detail by looking into the main concepts of both tools. Also GitLab and CI/CD are introduced. The process of building IaC configurations and CI/CD and creation of new environment, are then explained in detail in chapter four. Results are then evaluated and concluded in last two chapters.

2. MOTIVATION AND BACKGROUND

This chapter describes the motivation behind the work that has been done. To understand the goals of the project, the state of the platform needs to be understood. First section describes the platform in general and the processes implemented in the software development lifecycle. In the second chapter, research task and the desired changes are introduced. The goals of the project are discussed in third section along with possible outcomes and result and how they will effect on challenges that were discussed in first section. Fourth section defines the things that are not included in this project, limits the thesis scope, and clarifies what will happen after this project is completed.

2.1 The current state of the platform

OpenText Liaison ANILinker, in short, ANI is a B2B EDI messaging platform, that has a wide range of components of different technologies. In total, the platform contains approximately 90 different components. Up on that, a database, lot of configurations, and lots of customer solutions. The Platform is a legacy system, which can be described in other terms as existing project that on all aspects is hard to maintain because of the code itself, infrastructure, dependencies, documentation, or build tools. [1] The platform was launched in the 90’s and has been expanding since. In recent years, the platform has been placed into a maintenance mode and effort on new development has decreased.

The platform can be divided into three main parts that all hold many different components but are different based on the development responsibilities: ANI Internal Core components, External components, Third party components. The ANI Internal Core components contains the components that handle data inside the platform. These components are developed by ANI development team. External components are also developed by OpenText, but different teams. These systems provide additional capabilities to Ani Platform such as visibility or connectivity. Third party components are software that are not developed in OpenText but are important parts of ANI Platform. These components provide form example standardized ways of transferring data in and out of the platform. In this project we only focus on The ANI Internal Core components to mitigate the project scope.

ANI Internal Core components part of ANI is mostly based on Windows operating system, Microsoft SQL Server, and .NET Framework. In addition to Windows, Linux servers support some of the software. Also, some components are built in Java, and run on TomCat servers. Frameworks from .NET Framework 3.5 to .NET 6 are in use, but some of the oldest components of the platform are written in VB6 and ASP 2. Most of the code is written in C# or Visual Basic. Also, lot of custom libraries are developed to hold the business logic and the database entities. Additionally, platform has a lot of custom plugins that are mostly maintained by Professional Services and Customer Support teams that also contain majority of the business logic of the platform. On top of that, custom configurations and mappings are also important. Simplified architecture of the platform is described in the figure 1.

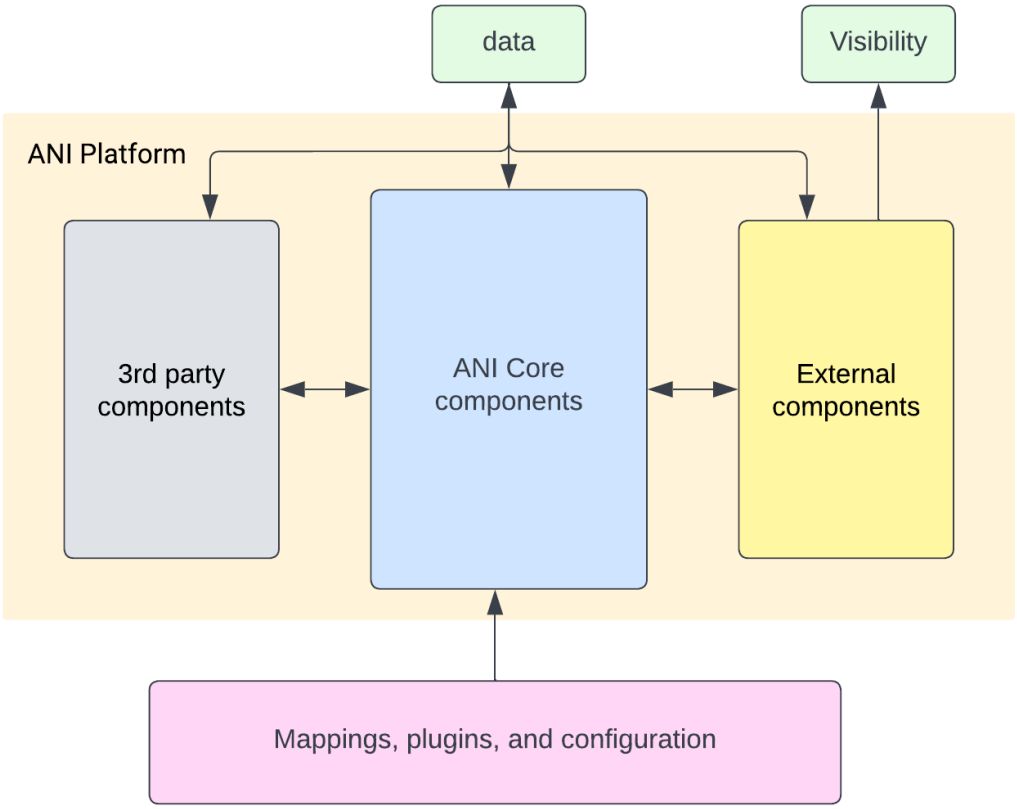

Figure 1. Different parts of ANI platform explained.

ANI Platform consist of three environments that are copies of the same platform but for different use cases, Test, QA, and Production. So called lower environments: Test and QA, have same setup: Same service groups, services, and number of VMs. Higher environment, Production has the same service groups and services deployed but the

capacity is different because of volume of data to be processed. Additionally, Production also includes all implemented configuration and data transfer protocols. Test and QA are used by multiple teams to test new software, configurations, and mappings, while Production is the live environment for customers.

Operating the platform requires work of multiple teams. Development team is responsible of the new code, bug fixes and testing in unit test level as well as integration level. Operations -team are responsible of deploying, configuring, and monitoring the Production environment of the platform. They should be able to troubleshoot issues and fixing them by configurations and maintenance tasks. Customer support team is responsible of the first line troubleshooting and support request handling. Professional services team is responsible of customer specific implementations to the platform and managing the customer relations. All these teams have some sort of access to the code or configuration of the platform and the responsibilities must be addressed.

Development team uses many common tools in their work such as git for version control, GitLab CI and Nant for build automation to mention few. Lot of things could be improved in development process by clarifying working habits and extending automation. Test automation is very rarely implemented although various kinds of tests exist. Packaged software is distributed also by manual manners. Operations team also deploys and configures applications and servers mostly manually even though some Ansible and PowerShell scripts are in use. Documentation is made by many teams individually and it is distributed to many sources such as code repositories

2.2 Research objective

In this thesis the focus is to develop an IaC configurations that enable creating new environments in flexible and automated means to enable better testing and bug fixing capabilities. Suitable tools for such task will be evaluated during the research and the work will be carried out using Terraform and Ansible. The first environment to be developed is the new QA environment that will be used for testing purposes when finished and will be replacing the existing QA environment. After QA, IaC configurations are used to create other environment flexibly, using the same configurations. Additionally, the work that is done helps to develop SDLC and automate many tasks of multiple teams. The biggest benefit is to enable operations to start improving automated deployments. The configurations Improvements are expected also in Documentation. Following issue are addressed with these changes.

- Use IaC to flexibly deploy new environments.
- Enable IaC on Production deployments.
- Use newest versions of operating systems, servers, and third-party software in QA environment, to test the changes that can be later be deployed to Production.
- Deprecate and get rid of old Test and QA.
- Make SDLC more automated.
- Improve testing ability and reliability.
- Save developers time when less time is needed for bugfixes and debugging.
- Documentation of service installation and configuration.

The information search process of this thesis is based on the research questions "How to build software infrastructure and deploy software automatically by using Infrastructure as Code tools?", “How to use IaC tools to develop flexibly repeatable environment?” and “Are Terraform and Ansible suitable to be used in Production environment by Operations team?”. The topic is also related to software quality assurance and DevOps which were also used as a reference keyword for information search. Key topics are “Infrastructure as Code”, “IaC”, “Terraform”, “Ansible”, “DevOps“, “Automation”, “software development lifecycle”, “SDLC”, “Continuous integration”, “Continuous Delivery”, “Continuous Deployment”

Books, articles, and theses are used to gather information about IaC solutions, DevOps, and other topics of the thesis to form an idea of the effects of IaC in the software industry, and how people and companies are using it to make a change. For Ansible and Terraform implementation the most useful resources are the official documentation of the products. The concepts of the tools and the abilities and disadvantages are clarified.

2.3 The desired changes and their effects

When the changes described in previous chapter have been implemented and the work is completed, there are multiple things that are better than they were before the start of the project.

The main goal is to enable developers to build new environments for different testing purposes and only for short term benefits, for example, testing a new third-party

integration or custom solution for a single customer. Otherwise, it might be a safe choice to test new features in completely isolated new environment without taking a risk of breaking the commonly used QA or Dev environment.

The other part is the new QA environment and the benefits of infrastructure automation and automatic deployment. The outcome of this project is to develop the QA platform to the state where testing of some of the most important and most frequently changed components is possible before deployment to Production environment. The QA environment will be running on newer version of operating systems, webservers, and databases. Also new frameworks of .NET or Java will be used. The main purpose of the update is to enable changes to newer versions and frameworks also in production. Production instance will later go through series of updates, and it will be moved to newer environment as well. New QA environment makes possible to test each component on the new environment in a trustworthy matter. With the knowledge gathered during this project it is easy to continue expand the QA environment to eventually include all the component in the platform. That will continue to expand the ability to do comprehensive end to end testing.

The new QA environment will be built using Infrastructure as Code tools that are used to automate the whole environment creation process. The environment is configured in code and the tools automatically provision the desired environment. Once the QA environment is set up, IaC tools enable building other similar environment with minimum effort. Previously, a lot of manual effort would be required for building, and keeping all the environments similar, would take considerable amount of work, which would mean less time for the actual development work. Also changes to environment are easily deployed by making changes to the code leading to CI pipeline provisioning the desired environmental components. This could happen for example in case of lack of computing capacity or memory in the platform, or if completely new components are added to the platform. In that case it is easy to add a new virtual machine with required computing capacity and then install and configure the component automatically. As a result of new QA environment, the old Test and QA environments will be deprecated once this project reaches a state when it is possible.

One part of making the platform better is the development of the SDLC process. Each component can be deployed automatically using Ansible playbooks. Additionally, SDLC of a single component should include building, testing, packing, and releasing the software. When each component has a pipeline to perform these steps, the time from developer changing the code to the software being deployed to Productions should decrease. As part of that the software packages are published to a package manager

software from where the distribution and deployment is easy, either by using automation or manual effort.

Operations team, that is responsible of Production environment, has been working with manual practices throughout the history of platform. One goal of this project is also to introduce IaC to Operations and enable automation for also Production environment provisioning, deployment, and configuration. This would enable complete continuous deployment for ANI Platform in the future, where new software can be released on shorter cycle.

Code of the IaC solutions also work as a documentation of the platform. The development team will have better control of the QA environment when everything is documented in the code. Ansible playbooks contain detailed instructions for every component about installation and configuration.

2.4 Out of scope

While this project concentrates to build a functional QA environment with deducted number of components and automation for the environment provisioning, there will be lot of work to continue with in the future. The automation can be extended for all the component in the platform, also concerning 3rd party components and the required configuration. In the end, the QA platform has all the functionalities as the production.

The scope of this project is not to develop applications of the platform themselves. The assumption during this project is that there is a package, usually an installer, for all the software that we want to install to the new QA environment. Also, CI/CD pipelines for the applications are out of scope.

One big part of provisioning of a new environment is a database migration. Each individual environment should have its own separated database instance. Ansible and Terraform are not suitable for migrating databases, but there are many ways to accomplish a good result. There are tools that are designed for database operations, for example Flyway, but in OpenText the database administration is handled completely by a separate team.

The most essential action after this project is to extend the environment provisioning and deployment automation also to Production environment. The work done during this project should be compatible to be used also in Production with slight modifications, and as the knowledge of the tools has been gathered, it should be also expanded to other teams and environments. That can help in multiple scenarios, including new

feature implementations, config changes, operating system and server updates, and disaster recovery.

3. INFRASTRUCTURE AS CODE

This chapter describes the terminology and the basic theoretical background of the topics of the thesis. At first, terms DevOps and CI/CD are explained by the benefits that they have brought to software development during its evolution. After that, deeper look into IaC is taken describing the roles of different teams and traditional ways of operations compared to the IaC solutions. Finally, the IaC tools and their features are introduced using official documentation.

3.1 From DevOps to Infrastructure as Code

Manual work of operations (ops) in a software company has led to increasing difficulties of managing software in different environments when the scale of product increased. Humans also make mistakes and software can drift to a state where the configuration is not managed properly on all the instances. Because of these flaws, software releases tend to become less frequent by time. DevOps is a result of software development becoming vastly more agile and efficient, by implementing ideas and processes where operations work is automated and thought differently. DevOps is about delivering software faster and more efficiently to the customer and making more maintainable and better software. [2]

Software is delivered through a release pipeline consist of different number of environments that are configured to serve different purposes. A part of DevOps is quality control or quality assurance, which is a way of testing the software in multiple environments during the release pipeline before deployment to Production. Naming the environments is not standardized but they serve different purpose in each stage,

DevOps helps to handle multi environment release pipelines by introducing deployment automation. Before DevOps development team would make a software that would be tested by quality assurance team and then deployed by operations. By automation, all developers are capable of testing and deploying their own code whenever possible through release pipelines. While working in multiple environments of the release pipeline, automation is in even bigger role by a saving ops effort in multiple stages. [3]

Deployment automation is described often with terms Continuous Delivery (CD) and Continuous Deployment (CD). A difference between them by manual intervention in a release pipeline after a code change, usually a merge to a master branch, by a human

to deploy to the final Production environment. If the human interaction is used, the process can be called Continuous Delivery but not Continuous Deployment. [4]

DevOps significantly reduces the effort needed from Operations to deploy software, but DevOps is not about invalidating ops. Operations will still have the knowledge about the environment, they will have time to work on reliability and maintainability and monitoring.

Infrastructure as Code takes one core value of DevOps, automation, into consideration and changes the way to manage infrastructure. Not by clicking and scripting, but by code. In the end almost everything can be automated. Different tools are being develop for different types of IaC tasks. There are 5 types of IaC tools:

- Ad hoc scripts
- Configuration management tools
- Server templating tools
- Orchestration tools
- Provisioning tools

Ad hoc scripts are the most straight forward way of automating infrastructure tasks. Benefits of them are ability to use popular languages and to make specific things. They are relatively easy to get started but when tasks get more complex, they are hard to maintain and keep up to date.

Configuration management tools are designed to manage software on existing servers. They offer many advantages compared to ad hoc scripts such as coding conventions, idempotence and distribution, which means they force their users to follow patterns and good practices and avoid repeating, while offering scaling for many purposes. Chef, Puppet, and Ansible are examples of this kind of tools.

Server templating tools are used to pack an existing server setup into an image, either a VM image or a container such as Docker, that can be then easily distribute across environments, preferably by using other Infrastructure management tools.

Also orchestration tools come handy when dealing with packed images. Along with deployment management they also keep track of the health of the running images as well as handle load balancing and monitoring tasks. Kubernetes or Amazon Elastic Container Service (Amazon ECS) are good examples of orchestration tools.

Instead of tools described above, that manage the running software, provision tools, like Terraform and Pulumi, are used to manage the servers themselves, such as AWS instances, VMs, Network structures and other. [5]

In following chapters, the most suitable tools for tasks of this project, Terraform and Ansible, and their main concepts are introduced along with comparison to other similar tools. With the gathered information it is possible to efficiently build an infrastructure using those tools. The last chapter focusses on automating IaC with the practices learned from DevOps and CI/CD and with the designated tool for that, GitLab CI.

3.2 Terraform

HashiCorp Terraform is an Infrastructure as Code (IaC) tool that allows you to build, change, and version infrastructure safely and efficiently. It was initially launched in 2014 and version 1.0 was released in 2021. In terms of IaC tools Terraform can be classified as provisioning tool, meaning that the tool does not define what is running on a server, but instead is used to create the server, VM, or almost any other aspect of IT infrastructure itself. It offers possibility to manage Kubernetes clusters, network infrastructure, policies, and multi-cloud environments. Additionally Terraform can perform tasks also outside of the scope of IaC. Interesting Terraform features are for example secrets management and Virtual Machine Image management. [6] [7] [8]

Terraform is used with Terraform CLI. To provision infrastructure with Terraform the most important commands are *terraform init, plan* and *apply*. *terraform init* initializes the working directory that holds the configuration files. After running *init*, user is able to run other commands. *terraform plan* compares the desired state defined in the configuration file to the current state defined in the state files and in the real physical infrastructure and creates a plan that describes the necessary changes to achieve the desired state in the real-life environment. The changes in that plan can then be carried out by running *terraform apply* command. [9]

3.2.1 Main concepts

Terraform includes many concepts that are used in different purposes. Terraform concepts provider, state, backend, and module are introduced in this chapter and later used in the configurations.

The most important base concept of Terraform are providers. Providers are plugins that are distributed separately from Terraform. Providers are the way of Terraform

interacting with different environments, such as different Cloud Providers, and their APIs to manage the desired environment. There are over 1700 providers written by HashiCorp and Terraform community, for managing different services and resources. Every Terraform configuration must declare which provider they are using. In this project, the QA environment is built in VMWare vCloud Director environment, meaning that the provider for that is “vcd”. It is verified provider which means it is managed by third-party partner, in this case, VMWare. [10]

Provider documentation offers information about how to write configurations for this specific provider. Each provider has custom configuration options to interact with the specific API that the provider has been designed for. A single Terraform configuration can include multiple providers to interact with multiple systems. If an environment that is built on VMWare, would be built on another platform, such as AWS or Azure, using the same configuration is not possible. The providers define the possible option for one single platform. Without providers, Terraform cannot manage any kind of infrastructure.

Each provider comes with a different set of resources and data sources. Resources are the real-life elements that are possible to be created with the provider. Each resource then has arguments that can be used to define the resources options that match the possibilities that are available also in real-life platform itself. For example, in vCloud Director, it is possible to create networks, vApps, VMs and other type of objects and configure them like they are configurable in vCloud Director UI. Data sources also match the objects in the provider but are used to gather information about the objects in the environment, that are often not created by Terraform or by a different Terraform configuration, to be used when creating new resources. [11]

State is another important feature of Terraform. It is used to map Terraform configurations to real life components such as servers or VMs. It is also used to improve performance of the jobs Terraform is running during provisioning. By default, the state is stored in the working directory of Terraform configurations in a file called "terraform.tfstate". When Terraform plan command is executed, Terraform compares the state that represent the current infrastructure to the newest configuration and creates a plan that describes which items should be created, modified, or destroyed and what will be the desired state after those modifications. After Terraform apply has been run the new state is saved to be used in the next plan.

State can be managed in the working directory as long as changes are made only by one person or from one computer. To enable working in teams, all team members must have access to the latest state to make successful modifications. For that purpose,

Terraform state can be stored in a remote location where it will be updates after each apply or destroy command. Terraform calls it the remote state. Remote state can be stored in Terraform Cloud or a backend that can be a third-party solution with different features such as authentication. Backend is defined in the Terraform configuration along with all the other options. [12]

Terraform Module is a group of resources that are used together. Each Terraform configuration has at least one module called root module. Other Terraform configurations can call modules to include them in the configuration, to re-use existing Terraform configurations. Modules can be included from multiple different sources, like remote storages or local path. Variables are used when using modules to customize a module to match requirements by passing them down in a module declaration. They are declared in Terraform configuration under *variable* section with a possibility to declare a type, description and other arguments, and the values are stored in a file with *.tfvars* -extension as a key value pairs. Modules also benefit of output values, that are used to fetch information about the infrastructure to be used in a Terraform configuration and they work as a communication channel between root module and other modules. [13]

3.2.2 Other solutions

Terraform has been compared with other tools that can solve similar problems, that can be handled with Terraform as well, and many other tools that overlap with some of the features of Terraform. Terraform documentation provides comparison with at least Chef, Puppet, CloudFormation, Heat, Boto, Fog, and custom solutions. Chef and Puppet are configuration management tools that compare better with Ansible and Terraform is not meant to be used as configuration management tool. CloudFormation and Heat Provide similar approach than Terraform as they represent the infrastructure in a config file. Boto and Fog give low level access to APIs while Terraform is providing high level syntax for managing the infrastructure with those APIs through providers. [14]

The closest alternative solutions are other provisioning tools like Pulumi, that also provides option to manage multiple different platforms. Different cloud platforms Like Azure or AWS offer option for provisioning using IaC. For example, Azure Resource Manager for Azure and AWS CloudFormation use declarative templates to provide IaC. Although using cloud platforms for this project is not possible, the only considerable solutions are Terraform and Pulumi. [15] [16] [17]

3.3 Ansible

Ansible was developed in 2012 by Michael DeHaan who also founded Ansible Inc. Ansible is open-source Project. Red Hat acquired Ansible Inc. in 2015 and has since been developing Red Hat Ansible platform. In this thesis the word Ansible is used to refer to the tool, and features of Red Hat Ansible Automation Platform or Ansible Inc. are not discussed.

Ansible is a simple IT automation tool that provides capabilities for application deployment and configuration, and many other IT needs. Ansible uses simple human readable language YAML, to describe the tasks. In this project, Ansible is used for configuration management and application deployment for the platform. In the terms of IaC tool type, Ansible has been classified as a configuration management tool. Other configuration management tools are for example Chef, Puppet, and SaltStack. The main concepts of Ansible; inventory, playbook, and role are introduced in following chapters. [18]

3.3.1 Building the inventory

It is important to be able to install and configure a component of the platform similarly in lower environments and in production. In context of Ansible it means that the same playbooks and roles can be used in all environments. Different environment run in different hosts and IP addresses, and number of nodes can be different. In Ansible those hosts are managed by configuring an inventory. Inventory is a list, or group of lists, that define the hosts. Inventory can be defined in many formats that depends on the used plugins. The two common formats are INI and YAML. Inventory file in INI format defines a single host as shown in figure 2 below. The same example also shows how to group hosts under a name. [19] [20]

```
dc1-hostnameX.com

[component1]
dc1-hostname01.com
dc1-hostname02.com

[component2]
dc1-hostname01.com
```

Figure 2. *Simple group and host definition in Ansible inventory using INI format.*

Single host can be defined on one line, but more convenient way is to name a host based on the name, use case or any other meaningful way. Same hostnames can be defined multiple times on different groups. Imagine having a single host and wanting to deploy two different services there. There are multiple reasons not to use a single group for both services. After defining a single service as a smallest possible unit, those services can then be defined in multiple other groups based on multiple use cases. Simple group definition is described in figure 3.

```
[servicegroup1:children]
component1
component2

[windows:children]
component2
component3
```

Figure 3. *Groups are defined using children directive.*

Groups can be set to serve multiple purposes and can be based on for example location, environment, type, operating system, or many other variables. For example, two web services that run on different servers thus do not belong to same group by locations, could still be added to same group defining then as webservices. Then if both of those services need supporting software or configuration, such as web server setup and configuration, a playbook can be created to target that group and both services can have same configuration. There are two default groups *all*, and *ungrouped.* [21]

Variables are essential part of enabling using the same roles and playbooks in different environments. For example, a SQL Server instance in different environment has a different name and that name is used to configure multiple components in all environments. By having the value defined as a variable and referenced in playbooks by variable name, playbooks can be reused in all the environments. In figure 4 is described a simple use of variables.

```
[all:vars]
sql=SQL-INSTANCE

[windows:vars]
# Ansible connection vars
ansible_user=User
ansible_password=Pass

[component1:vars]
app_user=App_user
```

Figure 4. *Variables can be defined for single host or wider groups using vars directive.*

Ansible Vault can be used to encrypt variables and files that contain sensitive content like passwords. *ansible-vault* is a command line tool that lets user encrypt and decrypt files by using a password, that can be then stored in a safe location. It lets user to turn passwords into crypted string, that can then be visible in the variable file. When running the playbooks, Ansible has an option to decrypt secrets on-the-fly when provisioning the target. [22]

Variables are merged following a set of rules defined by Ansible. The order from lowest to highest is:

- all group (because it is the 'parent' of all other groups)
- parent group
- child group
- host

There are multiple ways to achieve using the same playbooks in different environments. Ansible documentation introduces three types of inventory setup:

1. one inventory per environment
2. group by function
3. group by location

[23] [24]

The option one requires having multiple inventory folders. For example, creating folders *inventory_DEV, inventory_QA,* and *inventory_PROD* and having *hosts* -files in each of the with environment specific hosts, groups, and variables. Different inventory file can be then described at the command line using the -i <path> option in the *ansible*

command. Second option is to divide all hosts to groups that match their function in the environment. For example, web services, gateways, and processing software could be separated. Third option divides groups by the location, meaning that different environments are in different datacenters or other physical locations. [19]

An ansible concept “patterns” are then used to define which groups or host the playbooks target. To enable using the same playbooks in different environments, each playbook must define a host that is present in each inventory or environment. The host must be declared either in *ansible* or *ansible-playbook* command as the second element, or in a playbook, in *hosts* -field, as shown in figure 5. It is more convenient to have the pattern set up in the playbook to enable running all playbooks with similar commands and make possible to include playbooks withing each other. Then, if all inventories share the same structure, patterns are easy to manage. [25]

```
- name: <playbook_name>
  hosts: <pattern>
```

Figure 5. *Host definition in playbook.*

3.3.2 Using playbooks and roles

The main unit of configuration on Ansible is playbook. Playbooks are sets of steps that are performed in the desired process of deploying and configuring IT infrastructure. The complexity of tasks varies from simple to advanced. Playbooks are developed in human readable language, usually YAML, and have file extension *.yml.* Tasks are units of work, that are defined inside a playbook, and they invoke modules, that executes the actual tasks, that Ansible is dedicated to do. [26]

Modules are the actual scripts that Ansible runs locally or on remote. They interact with local machine, APIs, and remote systems to complete the desired tasks. Modules are grouped into collections. Windows specific modules are in Ansible.Windows-collection. Those modules provide functionality for many basic Windows configuration tasks such as *win_shell* for running PowerShell scripts, *win_dsc* for Executing PowerShell Modules, or *win_package* to download an installer file from network locations and running the installer with necessary arguments. Collections can be browsed and installed through Ansible Galaxy. Both modules and collections can be developed easily if no suitable one is found. [27] [28]

Playbooks can be written in a single file where all the task for one goal are set and that is the easiest way of getting started with Ansible. That will eventually cause writing the

same tasks multiple times. Ansible is designed to be re-used, so that a playbook or a role can include or import tasks, variables, roles, or other plays, like shown in figure 6. In this project the whole platform is deployed at once using a playbook that imports all the individual playbooks that are used to deploy a single application or configure a feature for wider group.

```
- import_playbook: webservers.yml
- import_playbook: databases.yml
```

Figure 6. *Including playbooks to another playbook.*

Many individual playbooks use roles to include re-usable tasks to a play. Roles are repeatable sets of tasks that can be used to template similar set of tasks. Roles are called from a playbook, and they are assigned with set of variables that differ between playbooks. Role performs the same tasks for different purposes based on the variables given when called. Variable defaults, files, templates etc. can be set for each Role to be used. For example, a role can be used to deploy and configure a web site and web application in IIS server. The role can have default values for those tasks such as web application names or server port, but if required, those values can also be set from a play that calls the role. Roles should be made from small parts that then are called from bigger playbooks or other roles to avoid writing duplicate code. [29]

Templating in Ansible is a way to help configuring applications of other assets by injecting variables into files. Many applications can be configured with a configuration file that holds environment specific values. Those values can be set as a variable, and by environment variables, inject the correct one to the file. Ansible uses Jinja2 templating language where variables are declared between curly braces as shown in figure 7. [30]

```
{{ a_variable }}
```

Figure 7. *Injecting variable to a configuration.*

3.3.3 Executing Ansible

In Ansible, control node is the machine that runs Ansible. Any machine with Python 3.8 or newer installed can be used, except Windows machines are not supported. With Ansible it is possible to execute multiple commands. The most important ones are *ansible* that is used to run single tasks and modules, and *ansible-playbook*, that is used to execute playbooks, that can contain multiple tasks and modules. [31] [32] [33] [34]

Both commands can take multiple arguments that are defined in the documentation. Target inventory is defined by using *-I, --inventory, --inventory-file* -option, This is important when using certain patterns that require choosing the inventory based on the environment. Also, *-l, --limit* -option is useful when wanting to target a host outside the pattern defined inside the playbook. Ansible connects to the target machine by using either *ssh* or to Windows machines using *winRM*. These options can also be assigned through command line options or preferably on variables in the inventory. [34]

When Ansible commands are run, Ansible prints an output to console about each play and task. Ansible prints output to the console of your control node, and it is also possible to save the output to separate file. Ansible prints the name of each play and task with the name of the targeted host and status of the task: Ok, changed, skipped, rescued, ignored, or failed. A recap of all executed tasks is displayed after all the tasks have been executed, showing number of tasks with different statuses. Example of output of *ansible-playbook* -command is shown in figure 8.

```
$ ansible-playbook -i inventory_test playbooks/install_application.yml

PLAY [Install Application] ***************************************
TASK [Gathering Facts] *******************************************
ok: [hostname.net]

TASK [Play to do something] **************************************
changed: [hostname.net]

PLAY RCAP*********************************************************
hostname.net:
ok=2    changed=1    unreachable=0    failed=0    skipped=0    res-
cued=0    ignored=0
```

Figure 8. *Output of ansible-playbook -command.*

3.4 Infrastructure automation using GitLab CI/CD

Essential part of this project is to enable infrastructure creation and software deployment by automated means. Terraform and Ansible alone provide advanced automation abilities for multiple purposes in means of the need of running only a few commands from your machine. Using CI/CD tools take the automation even further by totally removing manual effort after the code changes. In practice that means building CI/CD pipelines for both Ansible and Terraform configurations.

As a part of DevOps software has a pipeline that is used to automate different tasks such as build, test and deployment. Same kind of pipeline can be developed for IaC solution. There are multiple solutions that enable building pipelines and most of them offer similar capabilities. Commonly used tools are for example GitLab CI, Jenkins, Travis, Bamboo, and Azure DevOps. Jenkins, GitLab, and Bamboo has been compared for software delivery without finding significant weaknesses among the tools. The tools offer a pipeline structure where the user can define stages that are executed in certain order. Usually, stages include option to run scripts on different shell programs or programming languages, that are used to execute the desired operations for the software. [35]

Jenkins have been used as a build and deployment automation tool in OpenText products. Jenkins uses Groovy programming language to declare the pipeline steps instead of YAML syntax in GitLab CI. For the purposes of this project the CI/DC pipelines are built in GitLab CI which is a standard solution for OpenText products and has required capabilities for executing desired tasks in a pipeline. GitLab is DevOps platform that is mostly used as a git repository. It provides many useful features such as merge requests, and issue management. As part of GitLab, Gitlab CI/CD is tool for managing continuous software engineering methodologies that enables automatically build, test, deploy, and monitor applications. It includes concepts like *pipelines, artifacts,* and *runners,* that are later described. It enables simple YAML based configuration and user interface for managing the pipelines. [36]

GitLab Runner is an application that works with GitLab CI/CD to run jobs in a pipeline. It uses executors to run the jobs in the pipelines. Docker executor runs jobs inside Docker containers enabling stable Linux environment for using all tools. [37] [38] [39]

Terraform and Ansible have requirements for the environment that they are running on. That means that the Gitlab runner needs to be configured to serve operations of both tools. Ansible cannot run on Windows environment and for both tools ssh and winRM

connectivity need to be working. Gitlab provides default template for Terraform pipelines that has necessary steps for executing Terraform. [40] [41]

When changes to the environment is needed, and when using IaC tools, All the changes are made to the code files of the tools. The developer creates the new features in feature branch in the code and commits the changes to git. With GitLab, the developer then opens a merge request for the changes to be merged into the main branch. GitLab CI pipeline can run test and checks for the changes. For example, Terraform plan can be generated for review or dry runs can be performed. Other developers must then approve the changes before merge. After merge, GitLab CI pipeline completes the environment provisioning to desired environment.

4. IMPLEMENTATION PROCESS

After the theoretical background of IaC, Terraform, and Ansible had been examined, the next step is to build up the environment. This chapter describes the build process. The plan was to implement total of three environments: Test, QA, and Dev. Test would be the first to implement and would be used to test Terraform and Ansible scripts and do manual testing and configuration on the environment before committing the changes to version control. Then by using CI/CD, the QA environment would be built directly from the code using GitLab and GitLab CI pipelines. In the end when also QA has been proved working, the Dev environment would be set up using all working parts that are deployed to QA. The actual application development would be done using Dev environment, and the tests could be run on QA. Test environment would remain as a sandbox for testing Terraform and Ansible scripts.

Code repositories were created for Terraform configurations and for Ansible playbooks. First, the Terraform was configured and the first version of environment including virtual machines were created. After that the platform deployment using Ansible started piece by piece and the infrastructure pipeline was used to bring new features to use.

When first Terraform configurations and Ansible playbooks were ready, building the pipelines started. Then after the initialization, all changes could be deployed through CI/CD pipelines. Also pipeline for an individual component was built to test the scenario of deploying a single component.

In the end a completely new environment is created based on the developed IaC solutions to evaluate the goals of the thesis that was to be able to recreate environments in flexible way. The steps to create new environment are described and the problems and challenges of the solution are evaluated. Also, the use cases for completely new environments are determined.

4.1 Requirements

The main idea of the QA environment is to be possible to test the software of the platform before deployment to Production environment. It should be the best possible copy of the Production environment. To reduce the scope of this thesis the whole Production environment is not replicated to QA. In this project, the goal was to build a system that can be tested in a way that proves that the few most frequently updated

and most critical components of the platform work. The main functional requirements were set as follows:

1. Management application can connect to the platform and the data can be seen and managed through the UI.
2. Messages (data) can be sent to the platform and platform processes them as it should.
3. A web application is operational and can interact with the platform by receiving and sending messages.
4. Possibility to easily continue deploying of other applications and expanding the environment.

To achieve those requirements, various components need to be installed to the environment. Applications serve different purposes of the platform and are in some cases dependent on one another. All these pieces of the platform are directly installed and configured on the VMs of the platform.

The same requirements apply to all environments. Each of them needs to have the same VMs and software installed. All the environments are built in the same virtual datacenter in a vCloud Director. To separate the environments from each other, individual VMs of each environment are placed in a separate vApp, which is a construct that consist of VMs that provide useful features for managing them. [42]

4.1.1 Limitations

While gathering the requirements of the project, there are lots of things that can be affected in the process by making a choice. There are also things that cannot be affected directly in this project but would have other options in other circumstances. Project like this is made for a company that has established principles and tools for special operations. At OpenText and in this project couple of things cannot be affected on. VM Ware vCloud Director is used as a cloud platform where the VMs are built. It is the standard solution for OpenText lab environments, and it is used by multiple teams across organization. Also, GitLab and GitLab CI/CD are used as a git repository and as a build automation and CI/CD tool.

An SQL Server database is the most crucial part of ANI Platform, and one is needed to be set up for each individual environment. Database management is out of scope of this project and is also handled by another team within OpenText. Most applications

need a connection to database, and it is assumed that the database is fully functional and ready to serve the needs of the applications in QA environment. Also, Management application is an application that user installs to users own computer and connect to ANI Platform directly from the desktop. Because of that, it is not part of the Ansible and Terraform configurations. Although it is dependent of few components described in this project.

The operating system images are included in the vCloud Director environment and are maintained by another team within the corporation. Images come with standard OpenText features and configuration that help in the deployment, such as proxy server configuration. It is trusted in this project that the images are valid and secure and changes to the is out of scope of this project.

OpenText has a self-managed instances of JFrog Artifactory which is a DevOps tool for package management for storing binaries like software installers, executables, library packages, Docker images, etc. A generic repository was found for this project to hold the Terraform state files, in other words, Artifactory was the backend solution for Terraform. Artifactory is also a centralized location to store application installers that are used within Ansible playbooks. [43]

4.1.2 Networking and VMs

To serve the needs of the QA environment, components of the platform were divided to five service groups that each serve a simple and isolated purpose in the platform. One VM was set up for each service group of the platform. Components could easily be divided to these groups based on their task in the platform. VMs and Service groups are named as follows.

- sql
- core
- web
- gateway
- utility

All VMs use Windows Server 2019 image and computing capacity of 8GB RAM, 4 cores, and 40GB of disk space. While Production environment has strict requirements for computing capacity, requirement for the VMs in QA environment were not set and default VM sizing was used. Processing power, memory and disk space are easy to

increase if needed in the future, with Terraform without even need to reboot the VM, by using hot update option.

All VMs are also by default connected to OpenText Lab network has restricted access that is protected by firewalls. That kind of network structure enables efficient development while being secured from outside world. Environment could also be built in isolated network when a jump host or Ansible control node would be the only way of accessing the VMs.

4.2 Infrastructure

Terraform repository is initialized by creating a directory structure of necessary files shown in figure 9. The folder contains Terraform configurations and other necessary files such as *.gitlab-ci.yml* for GitLab CI pipeline configuration and *CHANGELOG.md* for keeping track of changes. *README.md* is the main place for documentation of the solution.

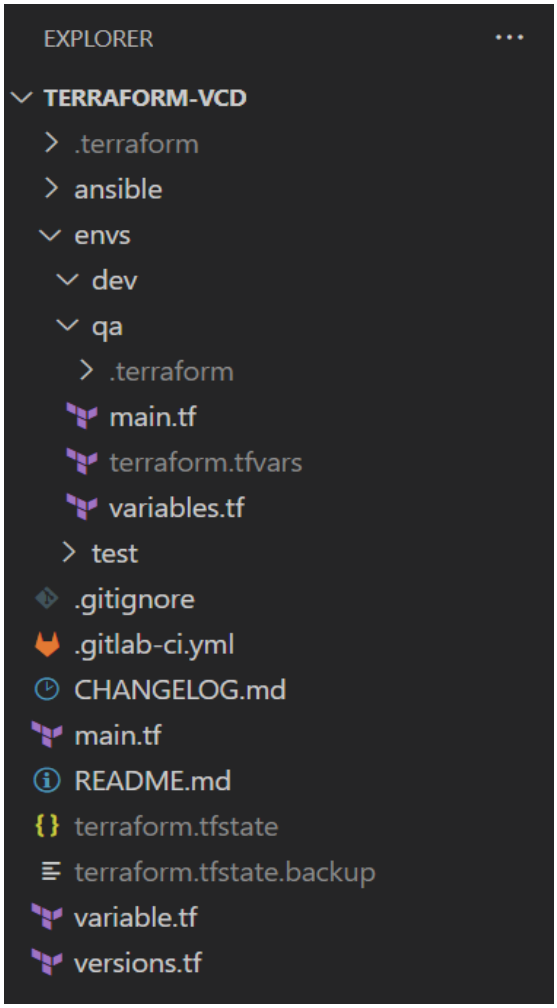

Figure 9. *Terraform projects folder structure. The environment specific variables are in envs/.. folders and main.tf in folder root is used*

Each environment is configured in their own folder that contains the necessary options and environment specific variables to be used to create the infrastructure. Following code is in the *main.tf* file in each of the environment folders, Test, QA, and Dev. In the *main.tf,* Terraform provider is declared in *terraform* block as shown in figure 10. The provider for VMWare vCloud Director is *vcd* Also *required_version* option is declared here, that tells what version of Terraform is used. Backend is configured in the same section with required options. [44] [45]

```
terraform {
  required_providers {
    vcd = {
      source = "vmware/vcd"
      version = "~> 3.0.0"
    }
  }
  required_version = ">= 0.13"
  backend "artifactory" {
    username = "your_artifactory_username"
    password = "your_artifactory_password"
    url      = "https://artifactory.instance.net/artifactory"
    repo     = "artifactory_repository"
    subpath  = "repository_path"
  }
}
```

Figure 10. *Terraform block defines provider and backend.*

Settings for the provider are declared according to the provider specific documentation. Each provider has different configuration options. For vCloud Director, the options are presented in figure 11.

```
provider "vcd" {
  user                 = var.vcd_user
  password             = var.vcd_pass
  auth_type            = "integrated"
  org                  = var.vcd_org
  vdc                  = var.vcd_vdc
  url                  = var.vcd_url
  max_retry_timeout    = var.vcd_max_retry_timeout
  allow_unverified_ssl = var.vcd_allow_unverified_ssl
}
```

Figure 11. *Vcd provider declaration.*

Each environment uses the same configuration module, that is located in the projects root folder. In each environment the source of the module must be declared. *source* option point to the path where Terraform can find the *main.tf* file of the module. Variables from the parent, environment specific, configuration must be injected to the module by declaring the as shown in figure 12 below.

```
module "ani-qa-env" {
    source = "../../../terraform-vcd"

    vcd_env = var.vcd_env
    env_abbr = var.env_abbr
    vcd_user = var.vcd_user
    vcd_pass = var.vcd_pass
    vcd_url = var.vcd_url
    vcd_max_retry_timeout = var.vcd_max_retry_timeout
    vcd_allow_unverified_ssl = var.vcd_allow_unverified_ssl

    vcd_org = var.vcd_org
    vcd_vdc = var.vcd_vdc
}
```

Figure 12. *Module usage.*

The main module that contains the configuration holds the resources that Terraform creates in the vCloud Directors virtual Datacenter. Each environment is built inside logical group in vCloud Director called vApp. Terraform declaration of vApp is shown in figure 13. Then each vApp hold the VMs designated for the specific environment. In the Terraform configuration the vApp is declared in following way. The environments name is injected to the vApp name using a variable.

```
# vApp
resource "vcd_vapp" "ANI_vApp" {
  name = "ANI_${var.vcd_env}_env_vApp"
}
```

Figure 13. *Declaring vApp resource.*

The VMs are declared using resource *vcd_vapp_vm,* as shown in figure 14. Each VM must be assigned to a vApp using *vapp_name.*

```
resource "vcd_vapp_vm" "web" {
  vapp_name     = vcd_vapp.web.name
  name          = "web"
  catalog_name  = "my-catalog"
  template_name = "photon-os"
  memory        = 1024
  cpus          = 1
}
```

Figure 14. *VM declaration for vcd provider.*

Also other vCloud objects can be created using Terraform Resources. All the configuration options for VMs are documented in the provider documentation.

There are also issues with Terraform configurations when something is not working as intended. The VM configuration is heavily dependent on the VM image. Terraform, or vCloud Director do not provide a VM images. That means that the organization must upload their own images to the vCloud Director and the create VMs based on those images. It is possible to select any of the templates in the libraries of the vCloud Director in Terraform configuration and use those in VM creation. In some cases that means that some of the features of Terraform does not work as expected. Setting a *computer_name* parameter in Linux VMs changed the hostname of the VM to that value, but in Windows images the value was not set as expected. Therefore running scripts after resource creation was necessary to set the hostnames correct for windows machines. Also, there are multiple options for the backend for Terraform, and the current solution with Artifactory backend is deprecated. Other solutions for example Gitlab managed Terraform state could be updated in the future. [45] [46]

4.3 Application deployment

The Ansible installation and configuration buildout could be divided into four different parts:

1. Prepare VM for applications installation
2. Installations of an application
3. Configuration of the application
4. Additional configuration in the environment

First step includes for example turning on and off Windows features or creating folders form applications. Next two parts are directly tied to a single component, a single installer, or a software package, that can be delivered as a single unit, that usually also contains a configuration file or another way of configuration. The fourth part is usually more complex than first three, as there are multiple types of configurations an application can require and many of them can be required cotemporally. Common additional configurations in this project are for example IIS Server configuration, Registry keys, Environmental variables, registering dll's, or creating a Windows service.

The process with Ansible is different from Terraform development. While Terraform configuration are meant to be used only for Test, QA, and Dev environments, Ansible playbooks are meant to be used also for Production deployments. Production deployments are done by operations team and requires cooperation between development and operations. The most important part of the work is to use Ansible patterns that are described in chapter 3.3.1 in a way that enables using the same configurations in Production and QA.

The structure of different environments and especially their differences determine the possible patterns that are usable in each case. In this project the solution is to have each playbook define the pattern in *hosts* field. The name of the pattern must be the name of the component the play installs, or a name of the software it is configuring. For example, playbook that installs and configures a component called Core Application, the pattern in the playbook is *core_application*. Then in the inventory the correct host is defined and set to groups. Other example is an IIS Server installation and configuration. The pattern in the playbook is iis_server. In the inventory group iis_server can include multiple hosts and other groups.

Because of the great differences between QA and PROD environment it is not possible to set the patterns in playbooks for wider groups. For example, if two webservices are on different service groups in Production but on the same one in QA, the deployment

using service groups as a pattern is not possible without changing the pattern before running Ansible commands or defining the pattern in the command. In complex system like this remembering possible patterns in all environments every time when running the playbooks is challenging and should be avoided. Instead using the patterns defined in the playbook YAML. The differences between setup in QA and Production is shown in table 1.

Table 1. *QA and Production set up*

COUNT	QA	PROD
VM	5	90
SERVICES	15	100
SERVICE GROUPS	5	30

QA and Production have different set of resources allocated. The biggest difference is the number of VMs in the environment. Production environment needs to be capable to handle large volumes of data and be fast in these operations. QA environment is used for occasionally testing and just a small number of transactions needs to be processed at a time. Eventually, all the services are installed to both QA and Production environments. During this project, the number of services in QA has been limited to 15, to define the scope of the work. Production environment has two to ten VMs per service group where the QA will have one. To design a QA environment based on production, some service groups need to be combined. When service groups are combined, that means that all the services on those groups will be deployed on the same VM in the QA environment. This is part of the planning process, and the plans must be changed if issues on performance or stability of the environment come up when more services are deployed to QA.

A service is a software component or an application, that is running on a VM. Many services from a service group. Service Group is a set of services that work together and form a logical group from the functionality perspective. Each service group can have multiple instances, and all the instances hold the same services. Each Service group has a designated number of VMs that are assigned to it. VM is a base unit of the platform. Each VM is part of a server group. Each VM can have one or multiple

services installed to it, and those services belong to same service group. There can be multiple VMs with exactly same setup.

The development process for Ansible playbooks was iterative and based on try and error, and existing documentation. The work usually followed a pattern:

1. Pick an application (service) to install.
2. Choose to which service group the service belongs.
3. Gather installers, config files and other resources.
4. Follow existing instructions of manual installation and document the steps carefully.
5. Transfer those manual steps into Ansible tasks.
6. Implement and test Ansible playbooks.
7. Confirm that component is working as desired.
8. If bugs are found, try fixing the manually and after fix is found, implement then in Ansible. Document the steps.

The Ansible Inventory is built based on the analysis on patterns in chapter 3.3.1. All playbooks must declare the host in a low level, meaning a host is the name of the installed application. Then by organizing the inventory file, that is named by environment name, those hosts are set to wider groups differently based on the environment's requirements. In QA environment, services would be divided into five service groups in five hosts. Inventory was built in INI format. After inventory setup, Ansible task are designed. Typical component would have following steps in the playbook:

1. Prepare VM for installation, for example create folders, turn on windows features, or configure IIS.
2. Download installer or zipped folder from network source.
3. Copy installer or unzip folder to installation location.
4. Run installer or executable.
5. Modify configuration file in the installation directory.
6. Create Windows service based on executable or create IIS web site and web application.
7. Configure services start up settings.

8. Start service or a web application.

When working in a Windows environment, most of the tasks can be performed by using ansible.windows -collection that has been described in chapter 3.3.2. For example, preparing the VM by installing IIS using *win_feature* -module is shown in figure 15. [47]

```
- name: <playbook_name>
  hosts: <pattern>
  tasks:
    - name: Install IIS
      win_feature:
        name: Web-Server
        state: present
```

Figure 15. *Ansible task using win_feature module.*

As described in chapter 4.1.1, JFrog Artifactory is used to store released packages. That means they need to be downloaded from the network location for the installation. Downloading application installer from network location can be done using *win_package* -module as shown in figure 16.

```
  - name: Install Application
    win_package:
      path: "{{ network_path }}\\application.exe"
      arguments: /S
      product_id: 1231238
```

Figure 16. *Usage of win_package module.*

Application configuration is mostly done using config files. The same files are used in every environment. But some configurable values must be injected to files that are environmental specific. Injecting variable to files can be done using *win_template* -module, as shown in figure 17.

```
- name: Configure Application
  win_template:
     src: ../config.xml
     dest: "{{ installation_folder }}\\exe.config"
```

Figure 17. *Usage of win_template module.*

During the build-up, lot of other tools are needed outside of the scope of IaC. Because of the VMs being Windows, for example Windows package manager Chocolatey

comes handy executing some of the tasks. There is also a Ansible module for such tasks: *win_chocolatey*. Powershell and Windows command prompt are also important tools when working with Windows environment. Lots of tasks can only be executed by running a script. [48]

After all necessary tasks have been implemented to a playbook, Ansible playbook command is executed. If the running of playbooks is successful, the results are tested by processing messages or browsing to a web application. If errors appear, the fixes are implemented to the playbooks and run again. In some cases, fixing error is initially easier by manual matters, but when the fix is found, it is implemented to playbooks as well. Sometimes Ansible also gives an OK status to a task even the task is not successful these errors are usually caused by using scripts in a task instead of a designated module. Ansible cannot always get the error status and a good error message to user. However, most of the modules and tasks work without problems and if problems occur the errors are comprehensive.

4.4 Automation and CI/CD

As described earlier, Terraform and Ansible configurations are stored in GitLab, and GitLab CI/CD is used as a build automation tool. A build pipeline can be created for the project by adding a *.gitlab-ci.yml* file to the project root. *.gitlab-ci.yml* file describes all options in the pipeline. [49]

Configuration of the runner is not part of this projects but understanding the concept is important when designing a CI/CD pipeline. OpenText's GitLab instance has multiple runners that can be used parallel. The task then is to determine a Docker image to run the jobs in. As there is not dedicated GitLab runner set up for this project's purposes, it is the easiest and recommended way to use a docker runner with a suitable Docker image for each job. The operative GitLab instance has runners to work as Docker runners, so setting them up is not necessary. The requirements for the image are to ability to run both Terraform and Ansible commands.

Terraform pipeline consist of four stages. First step prepares the image for the execution of Terraform commands. Next steps are for running the Terraform commands *init, plan,* and *apply.* The pipeline uses custom made, python-based Docker image that has Terraform, Ansible, and ssh-agent installed. That enables performing Terraform and Ansible tasks in isolated system. Pipeline steps are described in figure 18 and the view in GitLab is shown in figure 19.

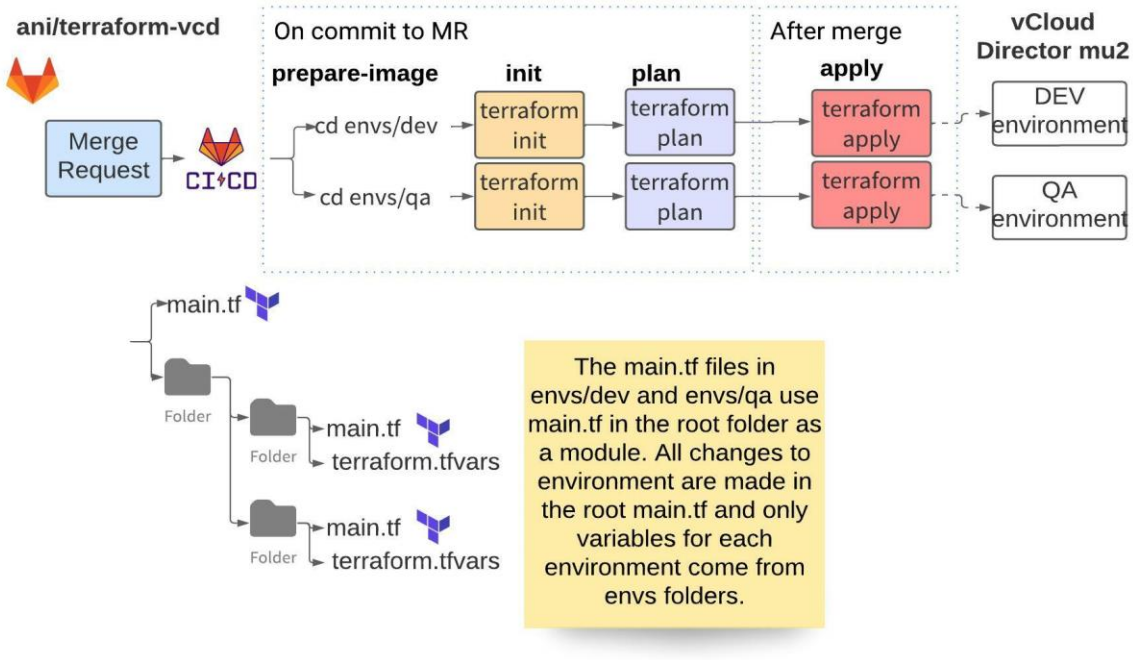

Figure 18. *Terraform pipeline structure.*

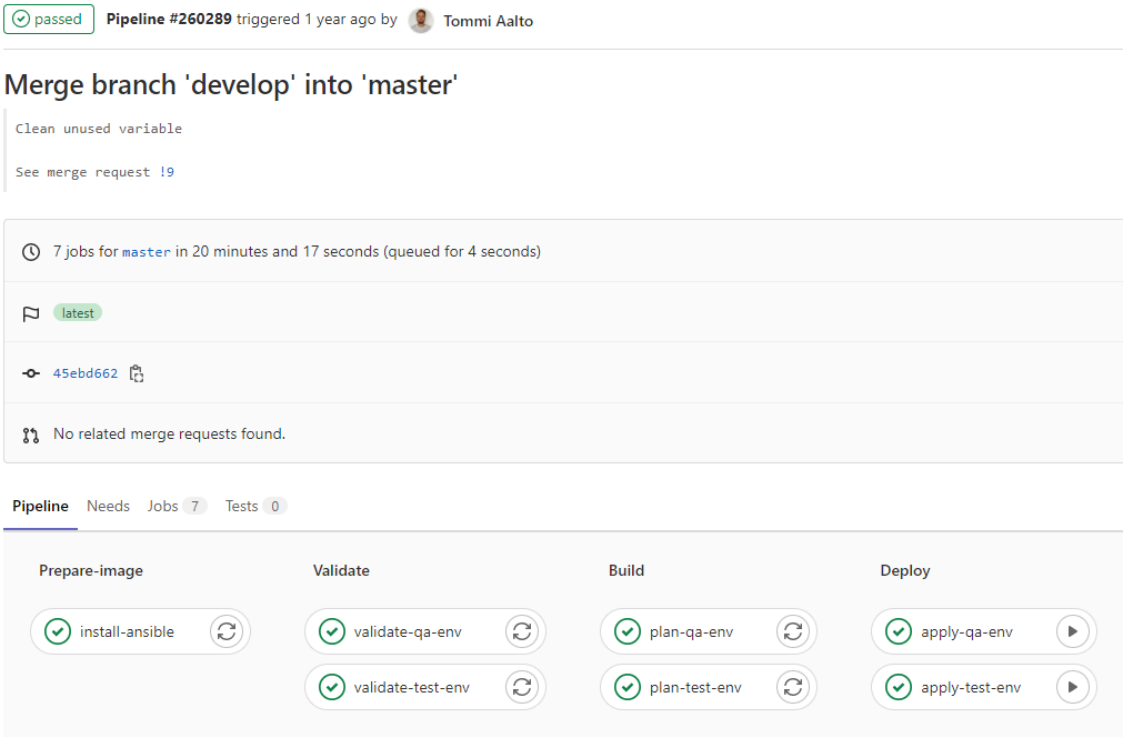

Figure 19 *Terraform pipeline after merge to master.*

Pipeline of the Ansible repository runs on each commit to a merge request, as shown in figure 20. Before the merge request is ready to be merged, it is not desired to deploy anything with draft code. Before merge, the pipeline runs a connectivity check on the Ansible control node to make sure that the changes can be deployed when they are ready. After the merge, the task is first to make the configurations available on the control node, because all Ansible deployments are run in an Ansible control node VM. The latest code is always in the *master* branch of the repository and in the pipeline, the lates code will be cloned or pulled to the VM using git. After the latest configurations have been fetched, the next job takes an ssh-connection to control node an executes Ansible commands. When making changes to Ansible configurations, all the changes should be provisioned to all environments. The repository has a *platform.yml* file that includes all playbooks that are needed to provision a functional environment, that is executed in the Ansible pipeline after a merge to *master.*

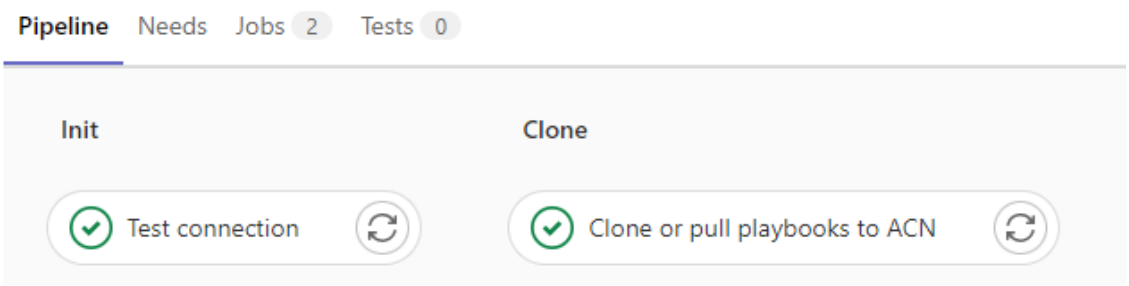

Figure 20 *Ansible pipeline after merge to master.*

4.5 Single component deployment

Each component of the platform has its own CI/CD pipeline that is used to automatically build, test, and deploy the application. Deployment of an individual component is done with the same Ansible configurations than the whole platform. When deploying the whole platform, the playbook used calls other playbooks and includes them in the play. In the single component scenario, Only the component specific playbook is called to perform just certain tasks. For example, when deploying Application to QA environment the *ansible-playbook* -command is run with necessary flags.

ansible-playbook -i inventory_qa application.yml

4.6 Creating new environment

The aim of this thesis is to flexibly create new environments for testing, development, or other purposes. Terraform and Ansible require different kind of changes to the configuration to enable creating a new environment. For both tools, the changes can be managed by copying configurations or environmental variables from another environment's solution and changing the values according to new environment's requirements.

Terraforms abilities are based on the providers thus using the same configuration on some other environment or cloud provider than VMWare vCloud Director is not possible. That sets a condition for Terraform deployments. If a vCloud Director is available, a Virtual Datacenter in vCloud Director is needed to be set up. It is not required to be same used for QA and Test environments. As described in chapter 4.3 also VM images that are used are sometimes different by their features and Terraform cannot guarantee that all images work the same with same Terraform configuration. If using different vCloud Director, the virtual datacenter, catalogues, and image names can be configured in the config files.

In the use case of ANI Platform, the creation of new environment is easy because of existing vCloud Director platform with required computing capacity, VM images, and CI tools for automated provisioning. The existing environment configs can be used when creating new environment with Terraform. As described earlier, each environment has its own folder in the repository that holds the configuration. Those configurations then use the *main.tf* file in the root to apply desired changes. To create new environment, copy the content of another environment specific folder to a new folder that is named after the new environment and change the values in the *.tfvars* -file to match new environment. Then *terraform init, plan,* and *apply* needs to be run from the newly created directory.

Ansible is more flexible. The datacenter under the VMs can be any. Nevertheless, playbooks always target a host with certain operating system, in this case Windows. However, host files in the inventory can be configured to match different set of VMs than the QA setup. That is why Ansible playbooks can also be used in future for higher environment deployment and configuration. In Ansible repository provisioning to new environment can be done by creating a new directory for the new environments hosts file to the root of the repository, copying the *hosts* -file from another environment's *hosts* -file and changing the values based on environment's requirements. The service groups and the number of hosts per application might vary per environment.

Deployment is done by running *ansible-playbook* command with targeting the new host file directory.

ansible-playbook -i inventory_env_name playbook_name

Both Terraform and Ansible configuration also contain GitLab CI pipeline configuration file. A new environment is also possible to create without using any CI/CD solution by running command independently. In the Gitlab configuration, the default pipeline uses an image for the build making it a requirement to have a Docker runner set up in the GitLab instance. Terraform requires an environmental variable to be set in GitLab settings. The variable contains the *.tfvars* -file that is normally located in the project root but is excluded from git because of sensitive values. It is used by terraform plan stage to inject configurations with variables.

Changes are carried out through GitLab merge requests. After changes have been made to Terraform and Ansible configurations. A merge request should be opened and GitLab CI performs initial checks for the code. After the merge Request is merged, the changes are applied to all environments.

5. EVALUATION OF THE SOLUTION

This project started with research of Infrastructure as Code where the history, terminology, and tools for IaC were clarified and compared. After the research, it was possible to determine what can be done with IaC and what cannot be done. Also, suitable tools for different tasks were selected. This knowledge was then evaluated to the requirements of the QA environment build out project, and the project was carried out using the most comprehensive tools for the tasks, Terraform and Ansible.

Terraform and Ansible performed well for the designated tasks. Terraform offered a provider for the vCloud Director that was well documented and easy to use. The VM images caused some issues with the configuration. Although the images are designed outside of the scope of the project, it would be an interesting topic to build also the images using server templating tools like Packer.

Ansible proves also to be a powerful tool form configuration management. It offers variety of modules for Windows tasks along with many for other operating systems. It is easy to get started with but offers possibility to extend the configurations to more complex sets of operations by using complex roles, variables, and patterns. The greatest benefit of Ansible is that it is more flexibly usable in other environments than the one it was developed against by simply building an inventory and using patterns. Operations need to be involved on building other environments, to help them to gain knowledge about IaC and start changing the way they work.

The build project of the QA environment was a success. The requirements set for the QA environment were met. Management applications can connect to the platform and manage configurations. Messages can be processed through platform and web application can communicate with platform by sending and receiving transactions. Now the platform is running on newer operating systems and software versions and development, operations, and other teams can run tests on it. The QA environment can be extended from now on to include more and more of the components of the platform to enable comprehensive end to end testing. In the end the old Test an QA environment can be deprecated.

Changes to the environments that are created using IaC is happening through GitLab merge requests and CI/CD. When changes are needed, for example new component or a version is needed to be deployed, developer makes code changes that are approved by other developer and after merge the changes are automatically

provisioned through pipeline. Also individual applications in the environment can now be deployed automatically from the component specific pipeline using the same playbooks that can are used when deploying the whole environment.

This project enabled creating new environments for testing and QA purposes fast and flexibly. Creation of completely new environment based on the QA setup is easy and is made by making required configuration changes to both separate code repositories. The same merge request procedure and pipeline will provision the new environment.

Certainly, other options beyond vCloud Director are possible. Public cloud providers such as Microsoft Azure, AWS or Google Cloud all provide an environment for setting up VMs. Terraform offers providers to work with each of those platforms and Ansible requires just similar VMs. Azure would be an excellent choice for Windows based Platform like ANI. IaC could also be used in case of containerization and orchestration of the platform in the future.

This project also improved the state of documentation of the platform. Every detail about installation and configuration of each component and their dependencies are written in the Ansible configurations.

As described above, using IaC brought many benefits and increased the quality of the platform and its testing abilities. The use of IaC should be extended also to Production deployments and could be used if platform develops towards containerized environment. Also server templating could be considered already now to improve image stability.

6. CONCLUSION

Infrastructure as Code is an interesting topic to investigate. It is certain that using IaC tools can bring benefits throughout software development lifecycle. Some of the tools, like server templating tools, like Docker, and orchestration tools, like Kubernetes are hot topics in software engineering. Also Terraform is gaining increasing popularity for different kind of tasks.

The main result of this thesis is the reusable Terraform and Ansible configurations. The other result of this thesis can be evaluated against the desired changes described in research objective. Now it is possible to deploy new environments flexibly, fast, and low effort with the configurations that were created. The usage of the configurations can be extended to Production environment provisioning as well if such decisions are made. The QA environment is running on new operating systems, servers, and third-party software, and it is easy to start extending the QA environment to eventually include all the components of the platform. This project also takes the organizations closer to the goal of deprecations of old QA environment.

The components in the scope of this project can now all be deployed automatically and the work to improve component pipelines can continue and automation will be extended in component pipelines. Teams can now test software on more reliable matter and customers benefit from more reliable software. Also, documentation is better when IaC configurations are also detailed instructions for deployment and configuration.

Infrastructure as Code tools offer a great chance to improve the processes currently implemented in the SLDC. While already small changes like automated deployment brings significant improvements, Configuration management and server templating can help operations to save time to focus on monitoring and improving stability of the platform. The tools and methods used in this project should be used also in Production environment. Also, any kind of deployment of new software or a configuration should be done by using automation. All manual work should be transformed into automated tasks. That takes a lot of effort at first, but IaC is an upfront investment.

7. REFERENCES

[1] C. Birchall, “1.1. Definition of a legacy project,” in *Re-engineering legacy software*, 2016.

[2] Y. Brikman, “The Rise of DevOps,” in *Terraform: Up and Running, 3rd Edition.*, O’Reilly Media, Inc., 2022.

[3] F. N. Emily Freeman, “Embracing the New Development Life Cycle,” in *DevOps*, Hoboken, New Jersey, 2019.

[4] P. Hodgson, “The Path to Production,” in *Continuous delivery in the wild*, Seabastopol, CA, O'Reilly Media, 2020.

[5] Y. Brikman, “What Is Infrastructure as Code?,” in *Terraform: Up and Running, 3rd Edition*, O'Reilly Media, Inc., 2022.

[6] Laura Paciliom, Hashicorp, “What is Terraform?,” Hashicorp, 22 1 2022. [Online]. Available: https://www.terraform.io/intro. [Accessed 14 2 2022].

[7] K. Ruddy, “Announcing HashiCorp Terraform 1.0 General Availability,” HashiCorp, 8 6 2021. [Online]. Available: https://www.hashicorp.com/blog/announcing-hashicorp-terraform-1-0-general-availability. [Accessed 4 9 2022].

[8] Y. Brikman, “Provisioning Tools,” in *Terraform: Up and Running, 3rd Edition*, O'Reilly Media, Inc..

[9] HashiCorp, “Basic CLI Features,” [Online]. Available: https://www.terraform.io/cli/commands. [Accessed 1 7 2022].

[10] HashiCorp, “vcd,” 2022. [Online]. Available: https://registry.terraform.io/providers/vmware/vcd/3.6.0. [Accessed 1 7 2022].

[11] HashiCorp, “Data Sources,” [Online]. Available: https://www.terraform.io/language/data-sources. [Accessed 22 7 2022].

[12] HashiCorp, “Remote State,” [Online]. Available: https://www.terraform.io/language/state/remote. [Accessed 1 7 2022].

[13] HashiCorp, “Modules,” [Online]. Available: https://www.terraform.io/language/modules. [Accessed 22 7 2022].

[14] HashiCorp, “Terraform vs. Alternatives,” [Online]. Available: https://www.terraform.io/intro/vs. [Accessed 7 1 2022].

[15] Microsoft, “What is Azure Resource Manager?,” 2022. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/overview. [Accessed 30 8 2022].

[16] Amazon Web Services, Inc., “AWS CloudFormation,” 2022. [Online]. Available: https://aws.amazon.com/cloudformation/. [Accessed 30 8 2022].

[17] Pulumi, “Pulumi vs. Terraform,” 2022. [Online]. Available: https://www.pulumi.com/docs/intro/vs/terraform/. [Accessed 30 8 2022].

[18] Red Hat, Inc., “How Ansible Works,” 2020. [Online]. Available: https://www.ansible.com/overview/how-ansible-works. [Accessed 14 2 2022].

[19] Ansible project contributors, “How to build your inventory,” 21 12 2021. [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html. [Accessed 9 3 2022].

[20] Ansible project contributors, “Inventory basics: formats, hosts, and groups,” 27 4 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#inventory-basics-formats-hosts-and-groups. [Accessed 29 4 2022].

[21] Ansible project contributors, “How to build your inventory,” 27 4 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#default-groups. [Accessed 29 4 2022].

[22] Ansible project contributors, “Encrypting content with Ansible Vault,” Red Hat, [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/vault.html. [Accessed 12 8 2022].

[23] Ansible project contributors, “How to build your inventory,” 27 4 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#how-variables-are-merged. [Accessed 29 4 2022].

[24] Ansible project contributors, “How to build your inventory,” 27 4 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#inventory-setup-examples. [Accessed 29 4 2022].

[25] Ansible project contributors, “Patterns: targeting hosts and groups,” 21 12 2021. [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/intro_patterns.html#intro-patterns. [Accessed 9 3 2022].

[26] Ansible project contributors, “Working with playbooks,” 21 12 2021. [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/playbooks.html. [Accessed 9 3 2022].

[27] Ansible project contributors, “Developing modules,” Red Hat, [Online]. Available: https://docs.ansible.com/ansible/latest/dev_guide/developing_modules_general.html#developing-modules-general. [Accessed 12 8 2022].

[28] Ansible project contributors, “Using collections,” Red Hat, 5 8 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/collections_using.html. [Accessed 12 8 2022].

[29] Ansible project contributors, “Re-using Ansible artifacts,” [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/playbooks_reuse.html#playbooks-reuse. [Accessed 15 7 2022].

[30] Pallets, “Template Designer Documentation,” [Online]. Available: https://jinja.palletsprojects.com/en/latest/templates/. [Accessed 15 7 2022].

[31] Ansible project contributors, “Control node requirements,” 21 12 2021. [Online]. Available: https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html#control-node-requirements. [Accessed 14 2 2022].

[32] Ansible project contributors, “Working with command line tools,” 26 9 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/user_guide/command_line_tools.html. [Accessed 1 9 2022].

[33] Ansible project contributors, “ansible,” 26 8 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/cli/ansible.html. [Accessed 1 9 2022].

[34] Ansible project contributors, “ansible-playbook,” 26 8 2022. [Online]. Available: https://docs.ansible.com/ansible/latest/cli/ansible-playbook.html. [Accessed 1 9 2022].

[35] J. Salojärvi, *Työkalujen vertailu,* Tampere: Tampereen Ammattikorkeakoulu, 2021.

[36] “GitLab CI/CD,” [Online]. Available: https://docs.gitlab.com/ee/ci/index.html#concepts. [Accessed 29 4 2022].

[37] “GitLab Runner,” [Online]. Available: https://docs.gitlab.com/runner/. [Accessed 29 4 2022].

[38] “Executors,” [Online]. Available: https://docs.gitlab.com/runner/executors/index.html. [Accessed 29 4 2022].

[39] “Run your CI/CD jobs in Docker containers,” [Online]. Available: https://docs.gitlab.com/ee/ci/docker/using_docker_images.html. [Accessed 29 4 2022].

[40] GitLab, “Infrastructure as Code with Terraform and GitLab,” GitLab, [Online]. Available: https://docs.gitlab.com/ee/user/infrastructure/iac/. [Accessed 12 8 2022].

[41] GitLab, “Terraform.latest.gitlab-ci.yml,” GitLab, [Online]. Available: https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Terraform.latest.gitlab-ci.yml. [Accessed 12 8 2022].

[42] D. Langenhan, “Chapter 3. Better vApps,” in *VMware vCloud Director Cookbook*, Birmingham, Packt Publishing, 2013.

[43] E. B. Salomon, “JFrog Artifactory,” 21 4 2021. [Online]. Available: https://www.jfrog.com/confluence/display/JFROG/JFrog+Artifactory. [Accessed 14 2 2022].

[44] HashiCorp, “How to use this provider,” [Online]. Available: https://registry.terraform.io/providers/vmware/vcd/3.0.0/docs. [Accessed 15 7 2022].

[45] HashiCorp, “Artifactory,” [Online]. Available: https://www.terraform.io/language/settings/backends/artifactory. [Accessed 15 7 2022].

[46] GitLab, “GitLab-managed Terraform state,” [Online]. Available: https://docs.gitlab.com/ee/user/infrastructure/iac/terraform_state.html. [Accessed 1 9 2022].

[47] Ansible project contributors, “Ansible.Windows,” 21 12 2021. [Online]. Available: https://docs.ansible.com/ansible/latest/collections/ansible/windows/index.html. [Accessed 14 2 2022].

[48] Red Hat, Inc., “win_chocolatey - Manage packages using chocolatey,” 2018. [Online]. Available: https://docs.ansible.com/ansible/2.6/modules/win_chocolatey_module.html. [Accessed 1 9 2022].

[49] “The .gitlab-ci.yml file,” [Online]. Available: https://docs.gitlab.com/ee/ci/yaml/gitlab_ci_yaml.html. [Accessed 29 4 2022].