Anssi Knuuttila

Managing IT-infrastructure with automation tools

Metropolia University of Applied Sciences
Master of Engineering
Information Technology
Master's Thesis
7 March 2025

PREFACE

I made a career shift from another industry to IT some years ago and I have had a chance to learn a lot. Studying bachelor's degree in IT made me understand how much there is in a subject of IT, specialization studies grew my knowledge on a single piece of the puzzle and creating a master's thesis made me understand a single narrow subject better. Working on IT means that one has to learn new all the time as the IT environments are in a constantly evolving state. Learning is what I like to do and hopefully I was able to share my learnings to others by creating this thesis.

At the moment I am writing the last pages of this study. Learning to use Ansible has been captivating and motivating. Still there is much to learn and hopefully in the future I get to work with Ansible and other automation systems.

Studying while being almost 40 years old has been surprisingly motivating. Maybe I have gained more learning capabilities while growing up. Having friends that were doing their own studies on different industries made me want to make a push and do this. Thanks for the ones that cheered me. Studying and working at the same time really disturbs the balance between rest and work. Knowing people that are going towards the same goal helps on a journey.

Sincerely,

Anssi Knuuttila
Lahti Finland, 7.3.2025

Abstract

Author:	Anssi Knuuttila
Title:	Managing IT-infrastructure with automation tools
Number of Pages:	40 pages + 2 on the appendix
Date:	7 March 2025
Degree:	Master of Engineering
Degree Programme:	Information Technology
Professional Major:	Networking and Services / Medical Technology
Supervisors:	Ville Jääskeläinen, Principal Lecturer

Growing IT-Infrastructure steered Rejlers Finland Oy to find solutions to automate system administration tasks. Finding solutions to the most common administrative tasks such as updating servers and third-party software would free up the workload from the system administration team. With automation the system administrators could improve their response time with the new tasks provided and the automated updates would happen more frequently, improving security.

The Ansible IT-infrastructure management tool was installed and apprehensive testing with the tool was conducted. The goal of this research was to get to know the Ansible tool, find ways to automate updating tasks on the Linux operating systems and find ways to utilize the installed tool.

As predicted, Ansible does have specific use cases that help automate the mundane administrative tasks and make the system administrator team more efficient.

Keywords: IT-infrastructure, system administration, ansible, automation, Linux

The originality of this thesis has been checked using Turnitin Originality Check service.

Contents

List of Abbreviations

1 Introduction 1

2 Automating of IT Administrative Tasks 3

2.1 Ansible Automation Tool 4

3 Utilizing Ansible Automation Tool 6

3.1 SSH keys with Ansible 6

3.2 Ansible Vault 12

3.3 Ansible Configuration 15

3.4 YAML format in Ansible 17

3.5 Inventory 20

3.6 Modules 23

3.7 Playbooks 26

3.8 Updating Ansible 34

3.9 Collections 35

4 Insights of Using Ansible 37

5 Discussions and Conclusions 39

5.1 Discussions 39

5.2 Recommendations for Future Studies 40

References 41

Appendix 1: Examples of Gathered Facts 1

List of Abbreviations

AES	Advanced Encryption Standard
API	Application Programming Interface
APT	Advanced Packet Tool
CLI	Command Line Interface
CPU	Central Processing Unit
DNS	Domain Name System
DSO	Distribution System Operator in energy industry
EV	Environment Variable
GUI	Graphical User Interface
IAC	Infrastructure As Code
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
INI	INItialization
IoT	Internet of Things
IT	Information Technology
JSON	JavaScript Object Notation
OS	Operating System
OWASP	Open Worldwide Application Security Project
PBKDF2	Password-Based Key Derivation Function 2
PID	Process Identification Number
PING	Packet Internet Groper
PIP	Pip Installs Packages
PPA	Personal Packet Archive
RAM	Random Access Memory
SHA	Secure Hash Algorithm
SSH	Secure Shell
SUDO	Super User Do, privilege escalation
XML	eXtensible Markup Language
YAML	Yet Another Markup Language

1 Introduction

This study was made for the Rejlers Finland Oy Sustainable Energy Solutions Divisions Metering Services business unit. Rejlers is an engineering company that operates in the Nordic countries and in the middle east. Rejlers provides engineering services in multiple sectors including energy and power, industry, infrastructure and environment, buildings and properties, communication and security and technical management consulting. The Metering Services business unit works in the energy industry and with the building and properties sector offering solutions to the Distribution System Operators (DSO) and the property owners. The Measurement Services offers customers remote meter reading, Internet of Things (IoT) devices, measurement data management and exchange, customer and billing systems, energy reporting and analytics, energy management, emission and sustainability reporting, electrical power quality measurements and balance settlements.

In the energy sector there are multiple substantial changes happening that are all affecting the Information Technology (IT) infrastructure on various levels and every actor in the energy sector must create new IT-systems to keep up with the development. Major changes happening are electricity metering data resolution change from 1 hour to 15 minutes that quadruples the amount data handled by the collection and management systems, more frequent deliveries of validated data to Finnish Datahub, meter changes from basic meters to smart meters, load management systems and DSO requirements for more real time data reporting.

Changes happening in the energy sector heavily affect the IT-infrastructure and the traditional ways to handle operations are becoming more time consuming as the number of servers and services are constantly growing. This study was made to gain knowledge of the Ansible open-source automation tool and find use cases to automate the basic system administration tasks and free up workload from the system administration team.

This study is limited to the basic tasks used to update and configure Ubuntu Linux version 22.04.4.

This study had to be efficient and provide real life usable material in a tight schedule. An efficient research design was crafted:

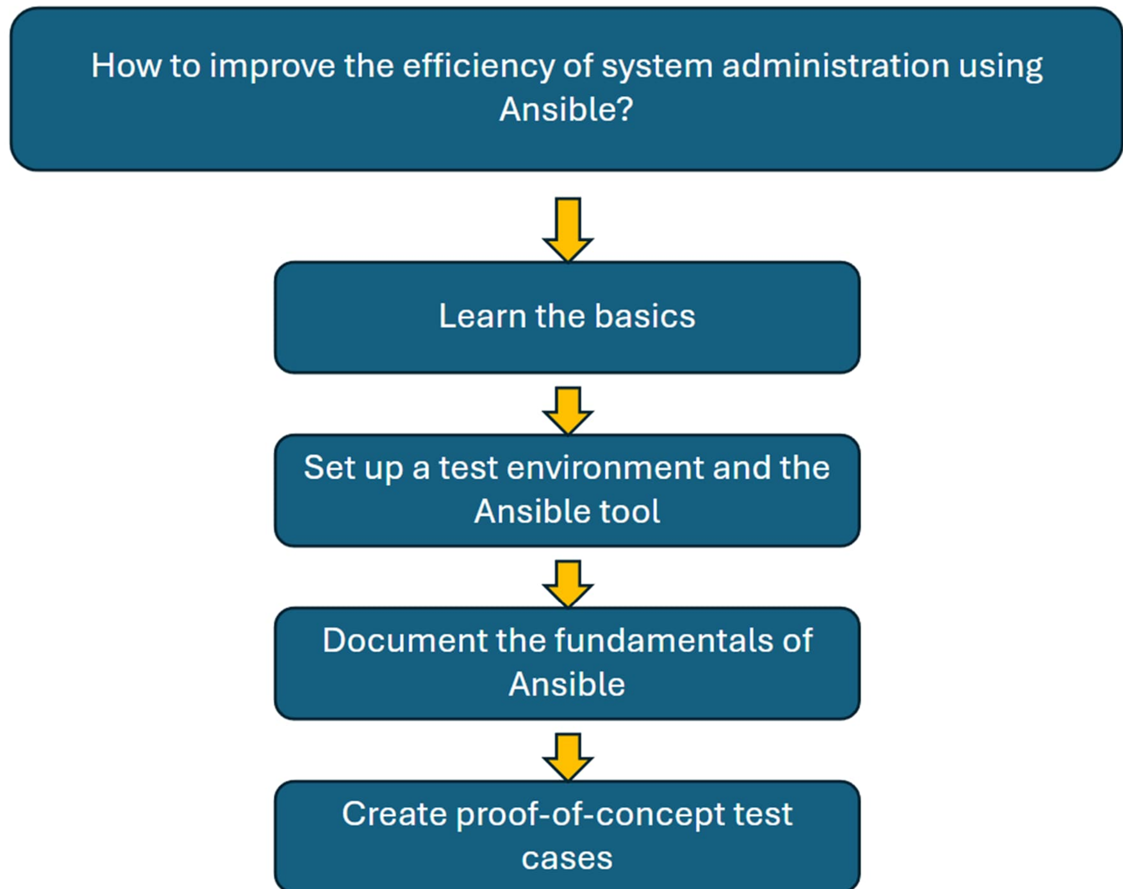

Figure 1. Research Design.

The goal of this study was to give a reader a basic understanding of the fundamentals of Ansible and capability to create basic operations.

2 Automating of IT Administrative Tasks

Using automation tools to help manage the administrative tasks is becoming increasingly popular as the size of the IT infrastructure is constantly growing and the tools to do automation with are improving. Most of the tools are Infrastructure As Code (IAC) tools that are used by writing a configuration file that the tool uses to handle the system provisioning, configuration, and management. Using the IAC tools streamlines the workflow of the administrative tasks as executing created automations does the same procedures and can be applied to multiple target systems simultaneously. Repeatability using the IAC tools creates systems that are equivalent to each other when desired. Setting up the systems with the IAC tools is efficient. When the configuration files are once created, updating the systems with the same IAC configuration helps keeping the systems compliant to required security frameworks.

Automation before the automation tools has been made by programming custom software. Python has been a popular programming language to automate the system administration tasks with. Python's characteristics, that includes ease of use, lots of libraries, open source, and a big community of developers, has made it a popular programming language when it comes to automations. It makes sense that the programming language used to build the Ansible automation tool was Python.

Common use cases to apply automation includes configuring servers and network devices, installing software and updates, application deployment, managing cloud environments. Even if some tasks are only performed once, it might be beneficial to do it with an automation tool, because having done the task once forces the user to create a template that can be used another time. Also doing the administrative work with an automation tool will execute commands on a target server quick and downtimes and possibilities for the human error are reduced.

2.1 Ansible Automation Tool

Ansible is a powerful open-source automation tool, later referred as Ansible, that can be used to automate any repetitive task worth automating. Ansible was created in 2012 by a software developer Micheal DeHaan as a side job when he wanted to create a tool that could simplify and streamline a process of managing multiple servers simultaneously. The release was an instant success and Ansible managed to gather community contributors to create more features and extend Ansible's capabilities. Red Hat acquired Ansible in 2015 and continued to open-source Ansible community edition. Red Hat developed their own commercial version of Ansible, later known as Red Hat Ansible Automation Platform, that has more features and support. Graphical User Interfaces (GUI) for Ansible are also available. Red Hat offers Ansible Tower as a commercial product, Ansible AWX and Ansible Semaphore are open-source projects that offers a GUI. This study is limited to Ansible community edition *ansible-core* version 2.12.0 and later in this study the version is upgraded to 2.17.9.

Ansible is easy to use Command Line Interface (CLI) tool made with Python that is configurable with Yet Another Markup Language (YAML). The YAML markup language was chosen instead of the eXtensible Markup Language (XML) or the JavaScript Object Notation (JSON) because of its simplicity and because it is easier for the humans to read and understand. Ansible is installed on a single server, later referred as the Ansible server, that has the access to target servers via Secure Shell (SSH). On a target server a user must be created with Super User Do (SUDO) privileges and while using Ansible there is an option to write the SUDO password manually. For better automation, an SSH-key should be generated, and the user's target server password should be stored in the Ansible Vault.

Ansible uses *inventory*, *modules*, and *playbooks* to interact with target servers. The *inventory* file contains the target server information in YAML-format and the servers can be divided into groups. *Modules* are code binaries that Ansible executes to gather data or insert commands on the target servers and can be

used via CLI. *Playbooks* can be made in YAML-format to perform multiple *module* actions and logic can be inserted to the *playbooks* to help create more complex automation plays.

Setting up Ansible is streamlined and easy. The target servers do not require separate agents, like many systems do, and all the actions are performed via SSH.

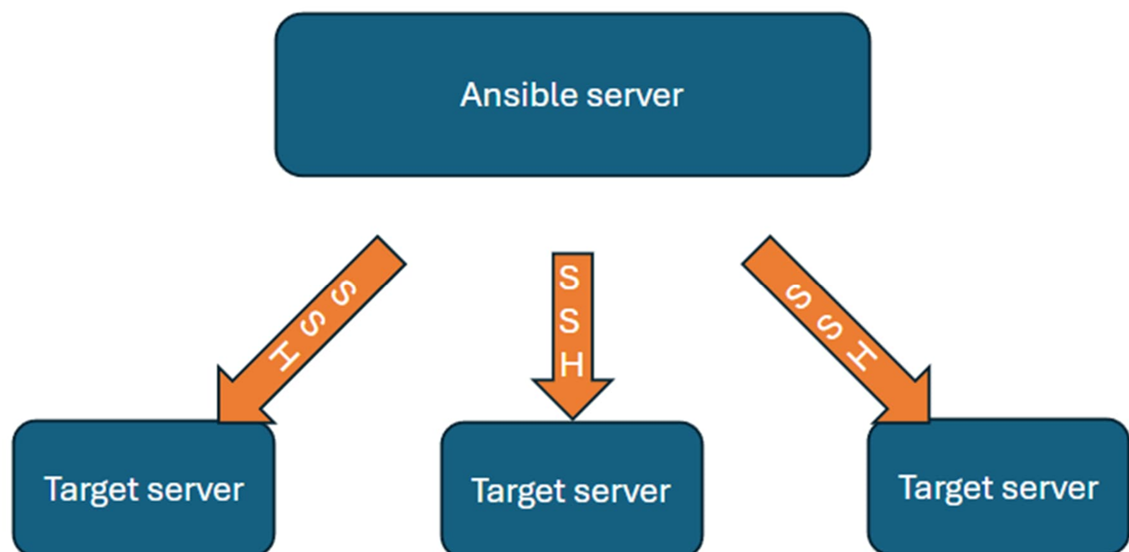

Figure 2. Ansible Server Setup.

3 Utilizing Ansible Automation Tool

Getting to know Ansible requires getting familiar with the basic concepts of how it works and how it is configured. Creating automations with Ansible can be done in numerous ways and understanding how the fundamental blocks of Ansible work and how the information is shared between one other is crucial. Setting up Ansible in a way that reduces repetition while creating the automations and reduces the risk of doing what was not supposed to be done requires some work, but getting to know the fundamentals makes automating the tasks easy. Ansible Documentation [2] is comprehensive and detailed information about how Ansible is set up is available. Red Hat Documentation [5] offers more information.

3.1 SSH keys with Ansible

SSH keys are a secure way to use SSH without a need of typing passwords when using SSH to login to remote servers. SSH keys provides the possibility to run Ansible *playbooks* that use escalated privileges on a scheduled job. SSH keys uses public key authentication that generates private key and public key. SSH keys are default assumption of preferred connection method for Ansible. SSH Academy [1] provides a lot of useful and detailed information how to set up SSH connections securely.

Before generating an SSH key, a user must be created to the Ansible server and to the target server. The user is created using the following CLI command on a target server:

```
sudo adduser tester
> Adding user `tester' ...
> Adding new group `tester' (1006) ...
> Adding new user `tester' (1006) with group `tester' ...
> Creating home directory `/home/tester' ...
> Copying files from `/etc/skel' ...
> New password:
> Retype new password:
> passwd: password updated successfully
> Changing the user information for tester
> Enter the new value, or press ENTER for the default
>         Full Name []:
>         Room Number []:
>         Work Phone []:
>         Home Phone []:
>         Other []:
> Is the information correct? [Y/n] Y
```

In this command privileges are escalated, *tester* user is created. Home directory for *tester* user is created and password and additional information is requested.

After creating user for the Ansible server and the target server, in the target server the created user must be added to the *sudoers* group. Users part of this group can escalate privileges and work on administrative tasks. User can be added to *sudoers* group, using a user with SUDO rights, with the command:

```
sudo usermod –aG sudo tester
```

Verification of the created user and groups that the user is part of can be done with command:

```
id tester
> uid=1001(tester) gid=1001(tester) groups=1001(tester),27(sudo)
```

After the user is created, an SSH key can be created. As the user *tester* the SSH key pair can be generated with the CLI command:

```
ssh-keygen –t ed25519 –C "Tester ssh key"
> Enter file in which to save the key (/home/tester/.ssh/id_ed25519):
> Created directory '/home/tester/.ssh'.
> Enter passphrase (empty for no passphrase):
> Enter same passphrase again:
> Your identification has been saved in /home/tester/.ssh/id_ed25519
> Your public key has been saved in /home/tester/.ssh/id_ed25519.pub
```

The command generates the key using the flag *t* that selects the used algorithm. The *C* flag is a comment for the key. After giving the command, file name is asked. If no file name is given, a file with the name shown in the prompt is created. Using a different file name than suggested is possible but will lead to some changes with commands when continuing to set up connection. The *ed25519* is the newest algorithm added to *OpenSSH* at the time of drafting this thesis and is the most secure compared to other options:

```
rsa          #An old algorithm that factors large numbers.
rsa –b 4096  #Rsa with a larger bit size to improve security.
dsa          #An old Digital Signature Algorithm used by US government.
ecdsa        #A new dsa that uses elliptic curves. Bit size 521 recommended.
```

Using other algorithm options can be less secure but mandatory in some use cases as support for *ed25519* algorithm is not yet universal. If *ed25519* cannot be used, then default should be *rsa* with flag *b* raised up using value 4096 that represents the number of bits used in key size.

Generating the key creates two files to the *.ssh* folder in user's home directory.

```
Ls-l ~/.ssh
> total 16
> -rw------- 1 tester tester  399 Feb 23 14:01 id_ed25519
> -rw-r--r-- 1 tester tester   88 Feb 23 14:01 id_ed25519.pub
```

The larger file of the key is the private key and the file with *.pub* at the end of the file name is the public key.

After the keys has been created in the Ansible server the public key has to be sent to the target server as the created user with the CLI command:

```
ssh-copy-id -i ~/.ssh/key_name.pub target_server_ip_address
> /usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed:
> "/home/tester/.ssh/id_ed25519.pub"
> ED25519 key fingerprint is
> SHA256:xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx.
> This key is not known by any other names
> Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
> /usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to
> filter out any that are already installed
> /usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are
> prompted now it is to install the new keys
> tester@testi's password:
>
> Number of key(s) added: 1
>
> Now try logging into the machine, with:   "ssh 'testi'"
> and check to make sure that only the key(s) you wanted were added.
```

If Domain Name System (DNS) is used or target server information is available in the Ansible servers */etc/hosts* file a target server name can be used instead of Internet Protocol (IP) address. User must give a password to the target server and the key is added to the Ansible server's *known_hosts* file:

```
Ls-l ~/.ssh
> total 16
> -rw------- 1 tester tester  399 Feb 23 14:01 id_ed25519
> -rw-r--r-- 1 tester tester   88 Feb 23 14:01 id_ed25519.pub
> -rw------- 1 tester tester 978 Feb 23 14:31 known_hosts
> -rw-r--r-- 1 tester tester 142 Feb 23 14:31 known_hosts.old
```

In the target server *authorized_keys* file is created to store keys that are sent:

```
ls -l ~/.ssh
> total 4
> -rw------- 1 tester tester 88 Feb 23 14:31 authorized_keys
```

When keys are introduced between Ansible server and target server, the user can use SSH to login from the Ansible server to the target server with the CLI command:

```
ssh –i ~/.ssh/key_name target_server_ip_address
```

If default key names were used the SSH client will find them automatically and user can log in to the target server with the CLI command:

```
ssh target_server
```

If key name was changed while creating the key the SSH client does not find the key and defaults to asking a password when logging in to another server. This can be fixed by changing the file names to the SSH client default format:

```
id_rsa, id_dsa, id_ecdsa, id_ed25519
```

If different file names for public and private keys are required, an SSH client can be informed with inserting a line to */etc/ssh/ssh_config* file:

```
...
#   IdentityFile ~/.ssh/id_rsa
#   IdentityFile ~/.ssh/id_dsa
#   IdentityFile ~/.ssh/id_ecdsa
#   IdentityFile ~/.ssh/id_ed25519
    IdentityFile ~/.ssh/ansible_no_pass
...
```

After inserting this line to */etc/ssh/ssh_config* file the SSH client finds the keys like it finds the default keys and logging in is simpler, persistent, and optimal for automations and tasks with *cron* that requires SSH connections.

If passphrase was given while generating the SSH key, it will be prompted, but if passphrase was not given the user does get a direct login to the target server.

If */etc/ssh/ssh_config* file cannot be edited and key names are required to differ from default file names, a user can use connect to target servers with an SSH agent. SSH agent provides Single Sign-On (SSO) functionality between servers and manages user's identity keys and passphrases. SSH agent can be activated with the CLI command:

```
eval $(ssh-agent) && ssh-add key_path_and_name_if_not_default
```

If SSH key was generated with a default key name the *ssh-add* tool will find the key from the user's home directory but if key name was changed it must be given in the command. If passphrase was given while generating the key, it must be given after a prompt from the tool. When the SSH agent is set up the user can open SSH connections between the servers using the CLI command:

```
ssh target_server_ip_address_or_domain_name
> Welcome to Ubuntu 22.04.4 LTS (GNU/Linux 5.15.0-133-generic x86_64)
```

SSH agent must be opened, or configured to be opened, at the start of every session when the user logs in. Not killing SSH agent process will lead to growing amount of SSH agent processes on a server and can become an issue over time. This happens because *eval $(ssh-agent)* starts an SSH agent process and exports Environment Variables (EV) to current shell session required by SSH client. Logging out a user terminates a shell session and loses EV: s and SSH agent process gets orphaned. Running commands with *cron*, a Linux job scheduler, that requires SSH agent functionality requires handling this problem with a persistent *tmux,* a terminal multiplexer, session or with a shell script that checks whether SSH agent is running and usable and either starts a process or uses an existing process and exports environment variables required by the SSH agent. Setting this functionality is not on the scope of this study. Killing an SSH agent process can be done with the CLI command:

```
ssh-agent –k
```

Secondary way of killing an SSH agent process is to find a Process Identification Number (PID) for the SSH agent process and kill the process with a *kill* command.

3.2 Ansible Vault

Ansible Vault is used with a CLI command *ansible-vault,* and it comes standard with the *ansible* or *ansible-core* installation packages. Ansible Vault is a CLI tool used to store secrets in variables that can be used in the *inventory* file configuration or in *playbooks*. It is strongly advisable to crypt all the passwords and keys so that they are not visible in a plaintext format. Vault uses 256-bit Advanced Encryption Standard (AES) to protect encrypted files. Brute-forcing passwords of encrypted files has been made even more difficult by using Password-Based Key Derivation Function 2 (PBKDF2) which uses salted 256-bit Secure Hash Algorithm (SHA) multiple iterations. The current Open Worldwide Application Security Project (OWASP) recommendation for the number of iterations is 600,000 OWASP [3]. The iteration number recommendation has grown over the years following the trend of growing processing power. Salting a password, any string, or file means adding characters to it. Vault keeps track of added random salt to be able to keep track of the calculations. Using this method ensures that even if encrypted files are using the same password, the encrypted data is different and incomparable.

Using Ansible Vault requires a Vault key. Vault key should be a strong password that should not get lost. Vault key can be stored in an Ansible server in a file that is accessible by Ansible. File permissions should be set to read and write only for the ansible user. Vault key file can be generated with the not so good CLI command:

```
echo "a_strong_password" > vault_key && chmod 600 vault_key
```

Using the *echo* method can seem tempting at first, as one liners are the most satisfying way of using Linux, but it leaves the password in a plain text to the

user's *~/.bash_history* log that stores CLI commands used by the user. Having a password information in this log may lead to security risks. This example was included in this study because this line is found all over the internet and needed to be called out. A preferred way of creating a vault_key file is made with a text editor:

```
nano vault_key # opens an editor and password is typed and file saved
chmod 600 vault_key # sets the rights to read and write for the owner only
```

Any file can be encrypted and decrypted using a file that contains the vault key. Encrypting a file using the *vault_key* file is made with the CLI command:

```
ansible-vault encrypt –-vault-password-file vault_key filename
```

Viewing the content of an encrypted file is possible but the content is protected by the Ansible Vault:

```
cat encryption_test
> $ANSIBLE_VAULT;1.1;AES256
> 63353938333261633134666237303465373533366463623232396139323632653836373 46
> 36563313764333032373236306437333663656461633838376431360a386564616435653 7
> 62366534643361356531336461326465633332343961353366363163363064623435333 36
> 33566386630636136653137333339383664383 40a62376637363936643638656366323765
> 373764363333326366333133316 6 6262
```

Decrypting an encrypted file using the *vault_key* file is made with the CLI command:

```
ansible-vault decrypt –-vault-password-file vault_key filename
```

Decrypted file's content is decrypted a readable format:

```
cat encryption_test
> stored_secret
```

If variable *vault_password_file* is configured in the Ansible configuration file, commands should be used without *–vault-password-file* option:

```
ansible-vault encrypt filename
ansible-vault decrypt filename
```

Viewing the contents of an encrypted file does not require decrypting of the file and encrypting it again after viewing it. A more efficient way of viewing the contents of an encrypted file can be made with a command:

```
ansible-vault view filename   # when vault password is set in configuration
ansible-vault view –vault-password-file vault_key filename   # when it is not
```

Encrypting sensitive information is useful with automated *playbooks*. Setting up an SSH connection to use SSH keys allows users to log in to the target server without prompting a password, but privilege escalation on a target server, required for doing administrative tasks, prompts the user to give their password so Ansible must have it. *Playbooks* can be run using a *–ask-become-pass* option that prompts user to give a password at the beginning of running a *playbook*, but a more automated way of giving a password to Ansible can be done by storing the password in an encrypted file. File containing a password can be made with commands:

```
nano become_pass
     File content should be: ansible_become_pass=user’s_password
chmod 600 become_pass
ansible-vault encrypt become_pass
```

After the encrypted password file is created and the vault key is set up in the configuration file the password can be used in *playbooks* like this:

```
---
- name: Beginning of a playbook that uses encrypted password
  hosts: linux_servers
  become: true
  vars_files:
    - become_pass
...
```

Using the encrypted variables in the *playbooks* is a good method, but for a repetitive use it is better to call variables in the *inventory* file. Using this method

requires *become_pass* file to contain only the password. An example of assigning variables in the *inventory* file looks like this:

```
---
test_servers:
  hosts:
    192.168.1.189:
    192.168.1.190:
  vars:
    ansible_become_password: "{{ lookup('file', become_pass) | trim }}"
...
```

If the targeted servers requires unique passwords, they can be stored in their own files for example *become_pass_189* and *become_pass_190.* Encrypted passwords can be introduced in the *inventory* file like this:

```
---
test_servers:
  hosts:
    192.168.1.189:
      ansible_become_password: "{{ lookup('file', become_pass_189) | trim }}"
    192.168.1.190:
      ansible_become_password: "{{ lookup('file', become_pass_190) | trim }}"
...
```

3.3 Ansible Configuration

Ansible is configured with *ansible-config* tool and configurations are stored in the *ansible.cfg* file. Multiple *ansible.cfg* files can exist allowing *playbooks* to use different configurations. Configuration files must be on a directory that has access restrictions. If a directory is world-writable Ansible does not read the *ansible.cfg* file. Restricting access to the directories containing the configuration files is mandatory to securely operate Ansible as changing the configuration file details can lead to undesirable events.

Installing *ansible-core* package does not create a configuration file. The *ansible.cfg* file can be created using *ansible-config* tool using command:

```
ansible-config init –disabled > ansible.cfg
```

The *ansible.cfg* file can be edited in a text editor. The file contains a list of variables in an INItialization (INI) format. Ansible can be used without *ansible.cfg* file but professionally managed configuration reduces repetition in *playbooks* and CLI commands. Well documented configuration file with explanations of each variable is a benefit of using ansible-config tool. Alternative way of creating a configuration file is to create a file named *ansible.cfg* and create variables it manually. An example of a basic configuration file looks like this:

```
cat ansible.cfg
> [defaults]
> inventory = ./inventory
> remote_user = ansible
> ask_pass = false
> host_key_checking = false
> retry_files_enabled = false
> forks = 5
> timeout = 30
> log_path = ./ansible.log
> stdout_callback = default
> vault_password_file = ./vault_key
> become_password_file = ./ansible_become_pass
> verbosity = 2
>
> [privilege_escalation]
> become = true
> become_method = sudo
> become_ask_pass = false
>
> [ssh_connection]
> pipelining = true
> ssh_args = -o ControlMasted=auto –o ControlPersist=60s
```

The *defaults* section includes paths to files and variables otherwise required to use in *playbooks* or in CLI commands. Mentioning variable log_path and giving it a file as a value starts the logging functionality. Logging and standard out

verbosity, amount of logging is adjusted with an integer from 0 to 4 where 0 is minimal output and 4 is used for debugging purposes. Logging with high verbosity level can be viewed as a security risk as sensitive information can get logged. Secure handling of logs is mandatory.

The *privilege_escalation* section introduces Ansible's term *become*. When Ansible connects to a target server to run administrative tasks it needs SUDO rights. Operating with SUDO commands raises a prompt from the target server and password is asked. Password can be stored in the file mentioned in the *become_password_file* variable. The password file needs to be encrypted with Ansible Vault and Vault password is given to Ansible in the variable vault_password_file.

Configuration variables like *inventory* are defaults that Ansible uses. Having this configuration set and using Ansible command to do a connection test to all hosts in the *inventory* uses default *inventory file* mentioned in the *ansible.cfg* file:

```
ansible -m ping all # uses inventory file mentioned in ansible.cfg
```

3.4 YAML format in Ansible

YAML is a markup language like XML or JSON, and it is made to be more human readable and writable using fewer special characters present in the other markup language formats. YAML version used when making this study is 1.2. Like general purpose programming language Python YAML uses indentation and line changes to structure the content and actions represented in the file. Because Ansible was written in Python, the use of YAML in managing and creating with Ansible is a reasonable selection from the pool of markup languages. YAML format can represent lists, dictionaries, key value pairs, scalars, variables and Boolean values. Ansible Documentation [2], YAML specification [4].

A brief list presentation of programming languages in a YAML format can be represented like this:

```yaml
---
# Comments are made using hashtag
- Python
- JavaScript
- C#
- Go
...
```

YAML files start with three dashes representing the start of the file and ends with three dots. A dictionary of an employee in YAML can be represented like this:

```yaml
---
employee:
  name: Test User
  department: IT
...
```

When listing attributes that are subcategories of the first category, in this example the attribute *name* is a subcategory of *employee*, two blank spaces are added. A combination of dictionaries and lists can be represented like this:

```yaml
---
employee:
  name: Test User
  department: IT
  projects:
    - Integrations
    - Databases
...
```

Key value pairs and Booleans in YAML format can be represented like this:

```yaml
---
boolean_value: true
key_value_pair: "value"
...
```

Scalars helps with making YAML format more readable. When printing text for the output Literal Block Scalars adds new lines and Folded Block Scalars are represented in multiple lines but prints as a single line:

```
---
multiple_lines: |
  These lines
  are on
  separate lines

single_line: >
  These lines
  are on
  a same line
...
```

Variables are helpful tools in YAML working with Ansible. Creating *playbooks* without variables is possible but leads to repetitive writing and unnecessary long *playbook* files. Variables can be introduced in the *inventory* file and can be referenced in the *playbook* file. In the *inventory* file variables can be represented like this:

```
---
linux_servers:
  hosts:
    192.168.1.189:
      variable_example: text_1
    192.168.1.190:
      variable_example: text_2
...
```

Variables represented in the *inventory* file can be referred in a *playbook* file like this:

```
---
- name: Playbook that uses variables
  hosts: linux_servers
  become: no
  tasks:
    - name: Print variable_example of each server in hosts
      debug:
        msg: "Variable has a value of: {{ variable_example }}"
...
```

Executing this *playbook* using this *inventory* file with an *ansible-playbook* CLI command will output text string for each host represented in the *inventory* file and text from each hosts' variable will be printed with default Ansible reporting format:

```
ansible-playbook –i inventory example_playbook
> PLAY [Playbook that uses variables]
> TASK [Gathering Facts]
> ok: [192.168.1.190]
> ok: [192.168.1.189]
> TASK [Print variable_example of each server in hosts]
> ok: [192.168.1.189] => { "msg": "Variable has a value of: text_1" }
> ok: [192.168.1.190] => { "msg": "Variable has a value of: text_2" }
PLAY RECAP
192.168.1.189 :
ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
192.168.1.190 :
ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

The output from this execution of an example *playbook* tells that the *playbook* was executed successfully. Basic execution information of the execution was printed and configured message from the *playbook* was printed to the output as supposed.

3.5 Inventory

An *inventory* file is Ansible's configuration file that holds information of the target servers. An *inventory* file can be configured with an INI format or with YAML. The target server names can be written using IP addresses or with server names if

DNS is implemented or if target server info is provided in the */etc/hosts* file. A basic *inventory* file with both formats look like this:

```
INI:
[test_servers]
192.168.1.189
192.168.1.190

[production_servers]
192.168.1.191
192.168.1.192

YAML:
---
test_servers:
  hosts:
    192.168.1.189:
    192.168.1.190:
production_servers:
  hosts:
    192.168.1.191:
    192.168.1.192:
...
```

In both cases the target servers are divided into groups and when writing *playbooks* the plays can be directed into these groups. This study focuses on using YAML format because YAML is a familiar format in Ansible. If target servers have a similar naming pattern a shortcut can be used to refer to multiple target servers at the same line. This *inventory* file refers to the *test-server-1*, *test-server-2* till the *test-server-10*:

```
---
test_servers:
  hosts:
    test-server-[1:10]:
 production_servers:
  hosts:
    production-server-[1:10]:
...
```

An *inventory* file can contain variables that are assigned to a group or to a server. Variables can be used in *playbooks* when logic is implemented, and some variables contains username and password information. Passwords should be encrypted with Ansible vault. In this example the test servers are using a same set of usernames and passwords, and the production servers are using the same username, but different passwords:

```
---
test_servers:
  hosts:
    test-server-[1:2]:
  vars:
    ansible_user: test_user
    ansible_become_pass: "{{ lookup('file', 'test_servers_pass) | trim }}"

 production_servers:
  hosts:
    production-server-1:
      ansible_become_pass: "{{ lookup('file', 'prod_server_1_pass) | trim }}"
    production-server-2:
      ansible_become_pass: "{{ lookup('file', 'prod_server_2_pass) | trim }}"
  vars:
    ansible_user: john_smith
...
```

When Ansible *playbooks* or *modules* are used with Ansible CLI tools, the *inventory* file is referred to with a flag *i* like in this command:

```
ansible -i inventory -m gather_facts all
```

The default *inventory* file can be configured in the *ansible.cfg* file and then the use of flag *i* is no longer necessary:

```
ansible -m gather_facts all
```

Having the exact same configuration and using a command that assigns an *inventory* file overrides the use of default *inventory.* This example uses an *inventory* file named *hosts:*

```
ansible -i hosts -m ping test_servers
```

Multiple *inventory* files can be used simultaneously by adding *i* flags:

```
ansible -i hosts -i inventory -m ping production_servers
```

In the Ansible CLI commands the last parameter defines a group that the command will target. If an *inventory* file contains groups *test_servers, production_servers* and there are some target servers without an assigned group, there is a possibility to target them separately or all together using arguments *test_servers*, *production_servers*, *all* or *ungrouped*. Ansible Documenation [2]

3.6 Modules

Ansible uses *modules* to make actions on a target server. *Modules* are usually Python coded tools that do a specific task like executes system commands or interacts with an Application Programming Interface (API). *Modules* comes in a library provided by Ansible. *Modules* can be used as a standalone tool using a CLI and are used when crafting *playbooks*.

There are thousands of *modules* available to use in Ansible and creating *modules* is a possibility. *Modules* are available to use in *collections* offered by repositories like *Ansible Automation Hub* and *Ansible Galaxy* and many more repositories

exists. Interacting with databases, docker and other common services is made possible and straightforward with Ansible.

Installation of *ansible-core package* includes an *Ansible.Builtin collection* that contains 200+ *modules* allowing user to build effective *playbooks* for basic tasks. Installation of *ansible* package comes with a more comprehensive set of *collections*.

Ansible offers an CLI tool called *ansible-doc* that prints out documentation of a *module*. Ansible also has extensive documentation in *docs.ansible.com* and finding information is easy. Ansible tool *ansible-doc* can be used like this:

```
ansible-doc module_name
```

Modules can be used with a CLI command using a flag *m*, *module* name that is going to be used and group of servers that the *module* is going to target:

```
ansible –i inventory –m ping test_servers
```

Some *modules* take arguments that are given with the CLI command. This is an example command that uses *command module* with an argument *whoami* that asks the operating system what user is currently active in this shell session:

```
ansible –i inventory –m command –a "whoami" test_servers
> test-server-1 | CHANGED | rc=0 >>
> test_user
> test-server-2 | CHANGED | rc=0 >>
> test_user
```

Modules are used in *playbooks* using YAML format and usage of installing a *nano* text editor with *module package* is shown in this *playbook* example:

```
---
- name: Playbook that installs nano text editor
  hosts: test_servers
  become: yes
  tasks:
    - name: This package module installs nano
      package:
        name:
          - nano
        state: latest
...
```

Running a *module ping* is a testing tool that connects to the target server and verifies ability to log on and existence of required Python tools in the target server. If the *module* executes successfully, a *pong* will be responded by the target server. Having the same name as the Packet Internet Groper (PING) utility that sends Internet Control Message Protocol (ICMP) packets may be confusing as the technology and intent behind the operation differs from each other. Running a *module* with *ansible* CLI tool will perform the task intended and a report of an execution will be printed to the terminal:

```
ansible -i inventory -m ping test_servers
> 192.168.1.189 | SUCCESS => {
>     "ansible_facts": {
>           "discovered_interpreter_python": "/usr/bin/python3"
>      },
>     "changed": false,
>     "ping": "pong"
>     }
> 192.168.1.190 | SUCCESS => {
>     "ansible_facts": {
>           "discovered_interpreter_python": "/usr/bin/python3"
>     },
>     "changed": false,
>     "ping": "pong"
>     }
```

Ansible Documentation [2]

3.7 Playbooks

Playbooks in Ansible gathers concepts previously explored in this study to *plays* that have more functionality and logic. Utilizing *playbooks* effectively requires basic understanding of the fundamentals of Ansible including but not limited to: A user with correctly configured SSH capability and SUDO privileges, passwords in the *Ansible Vault*, an effective *inventory* file and basic configuration in the *ansible.cfg* file. *Playbooks* can be done in numerous ways and the way the previously mentioned Ansible utilities are configured affects the use of *playbooks* in a great manner.

Ansible *playbook* hierarchy can be represented with the concepts of a *playbook*, a *play*, a *task,* and a *module*. A playbook is a list of *plays*, a *play* is a list of *tasks* a *task* is a configuration for a *module* and a *module* is a code block that executes commands. Lists in a *playbook* are executed in an order that they are represented.

When an Ansible *Playbook* is executed, a first thing that is executed is a *module gather_facts*. This *module* is not mentioned in a *playbook,* but it is an integral part of Ansible, and the information gathered can be used while creating logic for a *playbook*. The *gather_facts module* gathers close to a thousand details from the target server in a JSON format. Details gathered are extensive and numerous including but not limited to system information, network information, hardware information, user and security information and package and service information. Examples of the variables provided are presented in the appendix 1.

Examples in this section uses an *inventory* file and an *ansible.cfg* file. Target server information is added to the */etc/hosts* file. The hosts configuration file */etc/hosts* added lines looks like this:

```
#ansible test servers (ip server_name)

192.168.1.189 testi
192.168.1.190 testi2
```

An *inventory* file is configured like this:

```
---
test_servers:
  hosts:
    testi:
    testi2:
   vars:
     ansible_become_pass: "{{ lookup('file', 'become_pass') | trim }}" ...
...
```

An *ansible.cfg* file is configured like this:

```
[defaults]
inventory = inventory
remote_user = ansible
ask_pass = false
ask_become_pass = false
host_key_checking = false
timeout = 30
forks = 5
retry_files_enabled = false
gathering = smart
log_path = ansible.log
vault_password_file = vault_key

[privilege_escalation]
become = true
become_method = sudo
become_ask_pass = false
become_password_file = become_pass

[ssh_connection]
pipelining = true
ssh_args = -o ControlMaster=auto -o ControlPersist=60s
```

Ansible offers a CLI tool *ansible-playbook* that is used to run *playbooks*. The tool can be used to verify crafted playbooks before trying to execute them. Executing a *playbook* with a *syntax-check* flag validates YAML syntax and gives suggestions for fixes. Example of a poorly indented *playbook* looks like this:

```
---
- name: Playbook that tests connections
 hosts: test_servers
  become: yes
  tasks:
    - name: use ping module to connect to hosts
      ping:
...
```

Executing a syntax validation check with the *ansible-playbook* CLI tool raises errors:

```
ansible-playbook --syntax-check ping_test.yml
> ERROR! We were unable to read either as JSON nor YAML, these are the errors
> we got from each:
> JSON: Expecting value: line 1 column 1 (char 0)
> Syntax Error while loading YAML. did not find expected '-' indicator
> The error appears to be in '/home/ansible/ansible_core/ping_test.yml': line
> 3, column 2, but may be elsewhere in the file depending on the exact syntax
> problem.
> The offending line appears to be:
>     name: Playbook that tests connections
>    hosts: test_servers
>    ^ here
```

The syntax check noticed the poorly indented line. Inserting a blank space will correct the error and syntax validation check clears:

```
ansible-playbook --syntax-check ping_test.yml
> playbook: ping_test.yml
```

Using a *check* flag with the *ansible-playbook* CLI tool will execute a dry run without making any changes to the target servers. Ansible will connect to the target servers, gathers facts, and tries to identify potential problems that might arise. Executing a check for this test *playbook,* adding verbosity with the flag *v,* does not trigger any potential problems:

```
ansible-playbook –check -v ping_test.yml
> Using /home/ansible/ansible_core/ansible.cfg as config file
> PLAY [Playbook that tests connections]***************************
> TASK [Gathering Facts]*******************************************
> ok: [testi2]
> ok: [testi]
> TASK [use ping module to connect to hosts] **********************
> ok: [testi2] => {"changed": false, "ping": "pong"}
> ok: [testi] => {"changed": false, "ping": "pong"}
PLAY RECAP *******************************************************
> testi : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
> ignored=0
> testi2 : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
> ignored=0
```

This *playbook* example does not do any changes to the target server so output of the real execution of the *playbook* without a *check* flag does not differ from the previous execution.

Creating a *playbook* that updates all the services in the target servers is made using a *packet module*. The *packet module* checks the target servers preferred packet manager tool and makes managing different distributions easy. The preferred packet manager is selected when facts are gathered from the target server. The variable *ansible_pkg_mgr*, mentioned in the appendix 1, has the preferred packet manager information available. Checking if reboot is required after the updates happens differently between Linux distributions and some logic must be applied. Updating the services is done with one task, checking if reboot is required is done on a second task and third task will handle the reboot if required:

```yaml
---
- name: Update and reboot if required
  hosts: test_servers
  become: yes
  tasks:
    - name: Update the servers
      package:
        name: "*"
        state: latest
    - name: Check if reboot is required
      stat:
        path: /run/reboot-required
      register: reboot_required
      when: ansible_os_family == "Debian"
    - name: Reboot the target server when required
      reboot:
      when: reboot_required.stat.exists
...
```

Executing the syntax check and a dry run is strongly advisable. In this scenario the tests were executed successfully, and a full run of the *playbook* can be executed:

```
ansible-playbook -v update_and_reboot.yml
> Using /home/ansible/ansible_core/ansible.cfg as config file
> PLAY [Update and reboot if required] *******************************
> TASK [Gathering Facts] *********************************************
> ok: [testi]
> ok: [testi2]
> TASK [Update the servers] ******************************************
> changed: [testi] + Extensive amount of logging due the verbosity level
> changed: [testi2] + Extensive amount of logging due the verbosity level
> TASK [Check if reboot is required] *********************************
> ok: [testi2] => {"changed": false, "stat": {"exists": false}}
> ok: [testi] => {"changed": false, "stat": {"exists": false}}
> TASK [Reboot the target server when required] **********************
> skipping: [testi] => {"changed": false, "skip_reason":
> "Conditional result was False"}
> skipping: [testi2] => {"changed": false, "skip_reason":
> "Conditional result was False"}
> PLAY RECAP *********************************************************
> testi : ok=3 changed=1 unreachable=0 failed=0 skipped=1 rescued=0
> ignored=0
> testi2 : ok=3 changed=1 unreachable=0 failed=0 skipped=1 rescued=0
> ignored=0
```

This execution output is heavily reduced as the flag *v* added verbosity and listed all the actions that took place during the execution of this *playbook*. The *playbook* was executed successfully and there are some things that is worth mentioning. This *playbook* introduces the term *when* and it is used in two tasks. First usage of *when* is made in the second task which uses a variable *ansible_os_family*, and it is set to check that the operations are performed on a *debian* family Operating System (OS). If the condition is met, like it is in this example, a variable *reboot_required* is created and if the file */run/reboot-required* exists, the value of the variable will be *true*. Because that file was missing after the first task and the second task did not find it, the execution of the third task was skipped due to the condition set in the *when* clause. Critical thinking must be applied when it comes to the fact that some of the services might require reboot even though the */run/reboot-required* file was not created during the update, but this is outside of the scope of this study.

Ansible could execute the updates to both servers simultaneously because in the *ansible.cfg* file the variable *forks* was set to 5 simultaneous connections. The number of *forks* that Ansible can handle is dependent on the number of Central Processing Units (CPU) and amount of Random Access Memory (RAM).

Installing software to the target server can be made with the same *package module*. Targeting multiple servers with different software to be installed can be done with a single *play* in a *playbook*. In this *inventory* file, a preferred software of choice is selected in a variable:

```
---
test_servers:
  hosts:
    testi:
      install_this: postgresql
    testi2:
      install_this: podman
   vars:
     ansible_become_pass: "{{ lookup('file', 'become_pass') | trim }}" ...
...
```

A *playbook* that uses the *install_this* variable mentioned in the *inventory* file can be made and software in the *inventory* file will be installed to their assigned servers. At the same time a text editor *nano* is installed. When using variables, there is a chance that some servers in the *inventory* list does not have this variable assigned. Setting up a default value will let the *playbook* execute without errors:

```yaml
---
- name: Update and reboot if required
  hosts: test_servers
  become: yes
  tasks:
    - name: Update the servers
      package:
        name:
          - "{{ install_this | default('vim') }}"
          -  "nano"
        state: latest
    - name: Check if reboot is required
      stat:
        path: /run/reboot-required
      register: reboot_required
      when: ansible_os_family == "Debian"
    - name: Reboot the target server when required
      reboot:
      when: reboot_required.stat.exists | default(False)
...
```

After executing the syntax check and a dry run, the *playbook* can be executed. In this example the reboot of both servers was required. When reboot is required in Ubuntu servers, the file named reboot-required will appear in the /run directory. The execution was successful and performing a check on the target servers shows that the preferred software and nano was installed:

```
# Installations on server testi
# Checking the postgresql package installation and it is found
ansible@testi:~$ dpkg -l | grep postgresql
> ii  postgresql               14+238                  all
>  object-relational SQL database (supported version
> ii  postgresql-14 14.17-0ubuntu0.22.04.1             amd64
> The World's Most Advanced Open Source Relational Database
> ii  postgresql-client-14   14.17-0ubuntu0.22.04.1      amd64
> front-end programs for PostgreSQL 14
> ii  postgresql-client-common  238                     all
> manager for multiple PostgreSQL client versions
> ii  postgresql-common        238                      all
> PostgreSQL database-cluster manager

# Checking the docker installation and it is not found
ansible@testi:~$ dpkg -l | grep docker

# Checking if the command nano is found
ansible@testi:~$ which nano
> /usr/bin/nano

# Installations on server testi2
# Checking the postgresql package installation and it is not found
ansible@testi2:~$ dpkg -l | grep postgresql

# Checking if the podman command is found:
ansible@testi2:~$ which podman
> /usr/bin/podman

# Checking if the nano command is found:
ansible@testi2:~$ which nano
> /usr/bin/nano
```

Executing the same *playbook* again will update the specified software to their latest version. The latest version is specified in the *playbook* task that installs the software with the parameter *state*. Ansible Documentation [2]

3.8 Updating Ansible

The *ansible-core* used in this study was installed on an Ubuntu server version 22.04 using the Advanced Packet Tool (APT) The latest *ansible-core* package

version available is 2.12.0. Using collections requires newer version of the *ansible-core* and in Ubuntu server version 24.04 the newer version is available when using APT. Installing a newer version of the *ansible-core* to the Ubuntu 22.04 can be done with Pip Installs Packages (PIP) tool that is used to manage Python libraries, or by utilizing the official Ansible Personal Packet Archive (PPA). Installing the Ansible PPA can be done with the commands:

```
sudo apt update
sudo apt install software-properties-common
sudo add-apt-repository –yes –update ppa:ansible/ansible
sudo apt update && sudo apt upgrade ansible-core
```

Adding repositories is outside of the scope of this study, but if the third command times out there is a solution:

```
# add the repository manually in to a new file that apt checks
echo "deb http://ppa.launchpad.net/ansible/ansible/ubuntu jammy main" | sudo
tee /etc/apt/sources.list.d/ansible.list
# Download and add the key that verifies package authenticity
Sudo apt-key adv –keyserver keyserver.ubuntu.com --recv-keys 93C4A3FD7BB9C367
# Now ansible-core can be upgraded
sudo apt update && sudo apt upgrade ansible-core
```

After these instructions, the *ansible-core* package is upgraded to use a newer version. The *ansible-core* version used in this thesis from now on is 2.17.9. Ansible Documentation [2]

3.9 Collections

Installation of the *ansible-core* package comes default with the collection *ansible.builtin*. When more than default functionality is required, more collections need to be installed. Some collections might need an upgraded version of the *ansible-core*. Collections needs to have a directory that is set up in the *ansible.cfg* file. If the *collections_path* variable is not set in the configuration file, a default file path is used. A path where the collections should be installed can be found using the command:

```
ansible-config dump | grep COLLECTIONS_PATH
> COLLECTIONS_PATHS(default) = ['/home/ansible/.ansible/collections',
> '/usr/share/ansible/collections']
```

Ansible has a CLI tool *ansible-galaxy*, that is used to handle managing collections. The */home/ansible/.ansible/collections*, or another directory mentioned in the configuration file instead of using the defaults options, directory needs to be created and after the creation the installing of the *ansible.windows* collection can be done:

```
ansible-galaxy collection install ansible.windows -p
/home/ansible/.ansible/collections -vvvv
```

Installing a collection can take a significant amount of time. Installed collections can be listed using *ansible-galaxy* CLI tool with a command:

```
ansible-galaxy collection list
> # /home/ansible/.ansible/collections/ansible_collections
> Collection    Version
> ------------ --------
> ansible.windows 2.7.0
```

Ansible Documentation [2]

4 Insights of Using Ansible

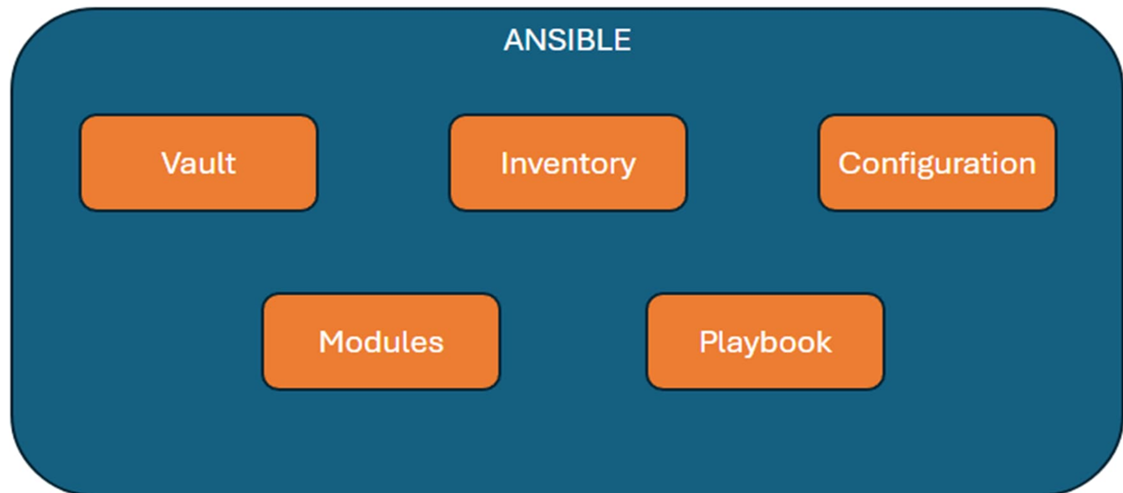

Figure 3. Ansible Concepts.

Using YAML as a markup language for the *inventory* file and for *playbooks* makes reading and interacting with the files easy. Creating the files requires focus with indentations. Creating with YAML on Ansible can be made exponentially easier using Visual Studio Code, one of the most popular Integrated Development Environments (IDE), and installing an Ansible extension. The extension validates YAML on the go and documentation is being provided on the IDE.

Creating the *inventory* file, or *inventory* files with YAML can be made in numerous ways and crafting the kind of file that best suits the target environment requires some thought process. Referencing the password and other variables in the *inventory* file makes creating *playbooks* less repetitive.

Having a robust *ansible.cfg* file makes creating the *inventory* file and the *playbook* files less repetitive. When running *modules* or *playbooks,* Ansible will first check the working directory to find the *ansible.cfg* file. If not found Ansible will check for the */etc/ansible/* path and use that configuration. If not found in */etc/ansible/* the default configuration is used. This can be confusing and making sure that the Ansible CLI commands are used in a correct working directory is important.

Using Ansible and creating automations will automatically create documentation in a form of the *playbook* files. Creating a *playbook* even for a one-time usage has benefits because of the documentation point of view. Other benefits of doing administrative work with Ansible are reduced possibility of a human error and less downtime on the target servers or devices.

Running the Ansible *playbooks* with increased verbosity and having logging set up in the *ansible.cfg* file creates one log of all the tasks done to target servers or devices. The collective logging can speed up working when there is no need to log on to target servers to view packet manager logging.

When it comes to documentation with Ansible, the creators have been doing an excellent job providing a lot of content in their website *docs.ansible.com*. Documentation includes information of the fundamental concepts used in Ansible, how to set up Ansible, and the thousands of *modules* available are described clearly with examples how to use them correctly. Information required to build even the complex automations can be found easily.

Ansible has become a popular tool to make automations with and has a market share of over 30 percent of the configuration management market. Community that uses Ansible is big and searching information outside of the documentation is easy. Community members have made a lot of tutorials in various formats to consume and sites like *stackoverflow.com* has a lot of questions that has been answered by the community.

5 Discussions and Conclusions

This study's purpose was to help increase the efficiency when it comes to the system administration and specifically when managing the Linux servers. The objective was to learn to use Ansible, create a documentation of the fundamentals of Ansible and create some proof-of-concept real life use cases.

The objectives of the study were met, and the system administration team has begun to utilize Ansible in their administrative work. Utilizing Ansible is effective and some of the administrative tasks are now done with it. Utilizing Ansible makes the system administration team more efficient and reduces the response time when it comes to the new tasks provided.

Learning to use Ansible was a captivating experience. Ansible is built in a way that has the flexibility and the possibility to automate tasks from the basic tasks to the specific use cases quite effortlessly. Learning the fundamentals of Ansible is straightforward, as it is separated into multiple building blocks. Gaining the knowledge of how the building blocks of Ansible react with each other is crucial. Ansible is a platform for a system administrator that wants to have the control over the administration process instead of giving the control to the software.

5.1 Discussions

Sanket Joshi, from the Vaasa University of Applied Sciences studied Terraform and the use of YAML in the thesis "Simplifying Infrastructure Management with Terraform and YAML Configuration (2024)". Ansible and Terraform both uses YAML, and the study gets to same conclusions as this study when it comes to working with administrative tasks on scale using YAML: "It is easier for teams to update and manage the infrastructure, which is crucial for long-term operational sustainability." [6 pp. 40]. Working with YAML files is user friendly and the ability to target multiple servers with the same working instructions provided by the Ansible *playbook* does scale up and make administrating servers and devices easy.

Mikael Holm studied use of Terraform automation tool in software deployment in the thesis "Code-based software deployment in cloud environment" and found the use of IAC tool and integration with GitHub actions well suited for the task [7 pp 55-57]. IAC tools like Terraform and Ansible both have cloud capabilities and use of the tools are an effective way to manage all kinds of environments.

Rasmus Mäki studied the use of Ansible in the Thesis "improving Efficiency and Minimizing Errors Through Automation (2024)" and came to conclusion that the use of automation tools will reduce human made errors. Doing automation tasks with the Ansible automation tool will automatically document the work [8 pp 42-43]. Created *playbooks* are a good form of documentation and reusability does make the use of automation tools an effective solution to tasks made in system administration.

5.2 Recommendations for Future Studies

There are multiple IAC tools and Ansible has the commercial product also. Hopefully more studies will be made in the future that research the subject of automating IT infrastructure, and they will be made in a manner that teaches readers to utilize the products effectively. Experiences of using automation tools for configuring hundreds or thousands of devices would be interesting. A guide that focuses on finding what tool is suitable for different environments and scenarios would also be helpful.

References

1. SSH Academy, ssh.com/academy
2. Ansible documentation, docs.ansible.com
3. OWASP, owasp.org
4. YAML specification, yaml.org/spec/1.2.2/
5. Red Hat documentation, docs.redhat.com
6. Sanket Joshi, (2024) Simplifying Infrastructure Management with Terraform and YAML Configuration
7. Mikael Holm, (2024) Code-based software deployment in cloud environment
8. Rasmus Mäki, (2024) Improving Efficiency and Minimizing Errors Through Automation

Appendix 1: Examples of Gathered Facts

This appendix includes the examples of variables collected with the *module gather_facts*. Because the amount of the information collected from the target server is vast, only a few categories and some variables from the selected categories are listed. The facts were gathered from the test server 192.168.1.189 that was used in the VirtualBox during the making of this study. The server was deleted after making this study.

System information:

```
    "ansible_distribution": "Ubuntu",
    "ansible_distribution_file_parsed": true,
    "ansible_distribution_file_path": "/etc/os-release",
    "ansible_distribution_file_variety": "Debian",
    "ansible_distribution_version": "22.04",
...
    "ansible_kernel": "5.15.0-133-generic",
    "ansible_kernel_version": "#144-Ubuntu SMP Fri Feb 7 20:47:38 UTC 2025",
...
    "ansible_architecture": "x86_64",
...
    "ansible_os_family": "Debian",
```

Network information:

```
            "ipv4": {
                "address": "192.168.1.189",
                "broadcast": "",
                "netmask": "255.255.255.0",
                "network": "192.168.1.0"
...
            "ansible_fqdn": "testi",
            "ansible_hostname": "testi",
```

Hardware information:

```
        "ansible_processor": [
            "0",
            "GenuineIntel",
            "Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz"
...
        "ansible_memtotal_mb": 1964,
...
        "ansible_virtualization_type": "virtualbox",
```

User and security information:

```
        "ansible_user_dir": "/home/ansible",
        "ansible_user_gecos": "ansible user,,,",
        "ansible_user_gid": 1001,
        "ansible_user_id": "ansible",
        "ansible_user_shell": "/bin/bash",
        "ansible_user_uid": 1001,
...
        "ansible_selinux": {
            "status": "disabled"
        },
...
    "ansible_ssh_host_key_ed25519_public":
      "AAAAC3NzaC1lZDI1NTE5AAAAIMgr+jjL0zE14v+jBQ0njZJdfRXifk+f8MuMzIAbM0R/",
    "ansible_ssh_host_key_ed25519_public_keytype": "ssh-ed25519",
```

Package and service information:

```
        "ansible_python_version": "3.10.12",
...
        "ansible_pkg_mgr": "apt",
...
        "ansible_service_mgr": "systemd"
```

Metropolia