

ANSIBLE

What Is Ansible?

Ansible is an open-source IT automation tool developed by Red Hat that simplifies tasks like:

- > Configuration management
- > Application deployment
- > Infrastructure provisioning
- > Orchestration of IT environments



Simple



Easy to Use



Flexible

Ansible are written in python.

It uses simple YAML syntax (called **Playbooks**) and connects to systems over SSH (clientless/no agent installation required). Everything in ansible is human readable format that makes it easier.

Control Node

Def: The system where you install and run Ansible is called the **control node**. (Accessible nodes – Managed nodes)
Or A task execution engine target the local system and remote systems.

When u run ansible on your system, in order to control other systems. You tend to call that local system running ansible your control node.

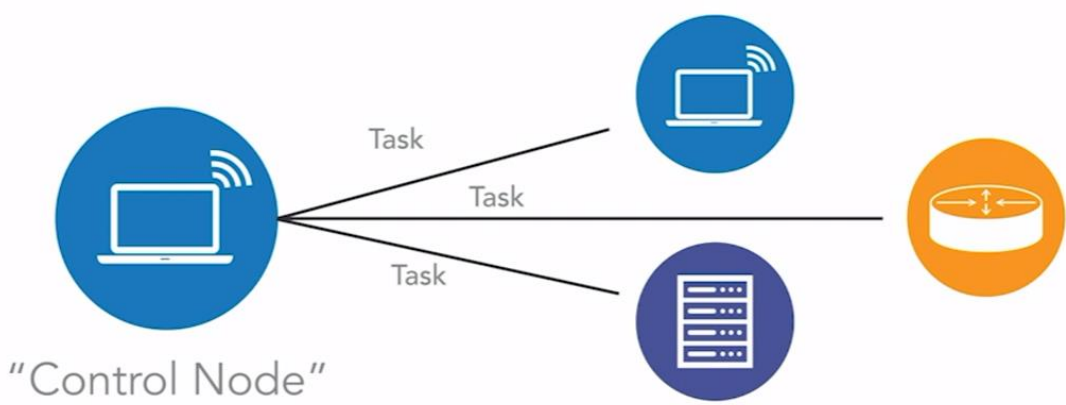
Fleet Management

Red Hat Ansible Tower

- Graphical UX
- Low-cost fleet management
- Not free software

AWX

- Graphical UX
- Free software
- Open-source



```
# Sample Ansible playbook
- hosts: all

roles:
  - common

# Configure and deploy database servers.
- hosts: dbservers

roles:
  - db

# Configure and deploy the web servers.
- hosts: webservers

roles:
  - base-apache
  - web

# Configure and deploy the load balancer(s).
- hosts: lbserver

roles:
  - haproxy

# Configure and deploy the Nagios monitoring node(s).
- hosts: monitoring

roles:
  - base-apache
  - nagios
```

Life before Ansible

Many Years Ago

- Server admins siloed
- "By hand" management
- Server tools developed
- Virtualization drove push for automation

Recently

- The growth of DevOps
- Server teams work with development staff
- Ease and transparency of collaboration

Playbooks

Ansible playbooks are like instruction manuals for your servers. They help you:

- Automate tasks (install software, configure settings, deploy apps).
- Keep everything consistent (no mistakes when setting up multiple servers).
- Save time (run the same steps over and over without doing it manually).

Why Use Playbooks?

i. Repeatable

- Run the same steps on 1 server or 1000 servers—exactly the same way every time.

ii. Easy to Share & Reuse

- Save playbooks as YAML files (like a text file).
- Store them in Git (like a shared folder for code) so your team can use them.

iii. Great for Complex Apps

- Need to set up a web server + database + firewall rules? A playbook can do it all at once.

Ansible: The History

- Familiar approach; leverage existing scripts
- Released in 2012 by Michael DeHaan
- Ansible Inc. was purchased by Red Hat in 2015 (Ansible Consulting and Ansible Tower)



Ansible Consulting

Assistance with Enterprise
Implementation



Ansible Tower

Graphical Interface
Software

Original Goals of Ansible

1. Clear
2. Fast
3. Complete
4. Efficient
5. Secure

Ansible Ease of Use

- Leverages modules for straightforward functionality
- Plugins to give additional capabilities
- Dynamic inventories

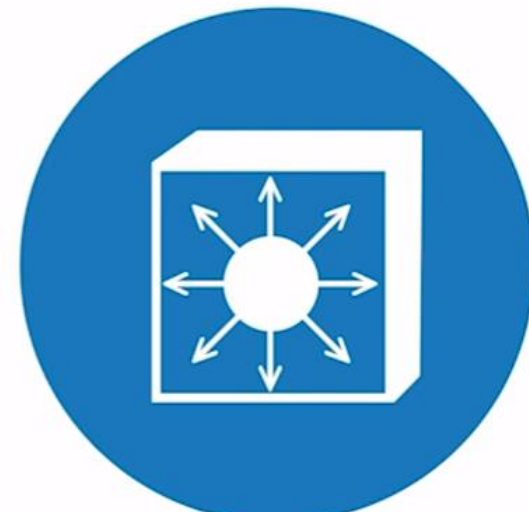
Advantages to Ansible



Easy to Learn



Python



Agentless

PLUGINS AND ROLES

In Ansible, both Plugins and Roles help modularize and extend functionality—but they serve very different purposes:

 Ansible Plugins (Extend Ansible's Core Functionality)

What is a Plugin?

A plugin is a piece of code that extends how Ansible behaves. Ansible uses many types of plugins internally—for logging, output formatting, caching, connection handling, etc.

Common Types of Plugins:


Plugin Type	Description
Connection Plugin	Controls how Ansible connects to nodes (e.g., SSH, local, docker).
Callback Plugin	Controls how output is displayed (e.g., default, JSON, minimal).
Lookup Plugin	Fetches external data (e.g., from files, secrets managers).
Filter Plugin	Adds custom Jinja2 filters (like custom string formatting).
Inventory Plugin	Dynamically generates inventory from sources like AWS, GCP, YAML.
Action Plugin	Controls task execution behavior.
Cache Plugin	Caches gathered facts for speed.

 Example: Callback Plugin to Show JSON Output
ANSIBLE_STDOUT_CALLBACK=json ansible-playbook site.yml

Ansible Roles (Organize Reusable Configurations)

What is a Role?

A role is a structured way to group variables, tasks, handlers, templates, files, and modules—so they are reusable across multiple playbooks.

 Example: Using a Role in a Playbook
yaml
- name: Configure web server
 hosts: webservers
 roles:
 - apache

If the role is in roles/apache/, Ansible will:
Run tasks from apache/tasks/main.yml
Use variables and templates as needed

Ansible Commands

```
[ec2-user@ip-10-0-1-184 mystuff]$ cd /etc/ansible
[ec2-user@ip-10-0-1-184 ansible]$ ls
ansible.cfg  hosts  roles
[ec2-user@ip-10-0-1-184 ansible]$ vim ansible.cfg
```

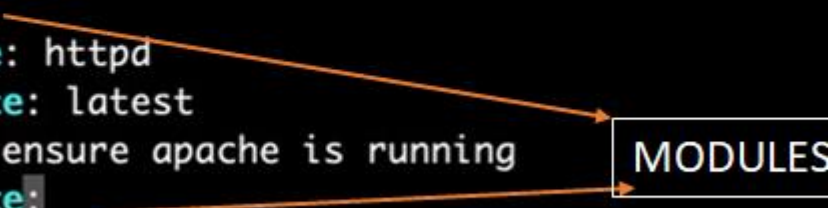
To escape out of the file after entering:
q!

To read a file in linux:
vim <file.ext>

To run ansible file all of the host inside the inventoy:
(here we can see the problem)
ansible-playbook <file.yml>

s

```
name: sampleplaybook
hosts: all
become: yes
become_user: root
tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: ensure apache is running
    service:
      name: httpd
      state: started
```



We can check a particular host device by ping:

```
[ec2-user@ip-10-0-1-184 ~]$ ansible myservers -m ping
10.0.1.184 | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python"
  },
  "changed": false,
  "ping": "pong"
}
ubuntu@10.0.1.79 | SUCCESS => {
  "ansible_facts": {
    "discovered_interpreter_python": "/usr/bin/python3"
  },
  "changed": false,
  "ping": "pong"
}
[ec2-user@ip-10-0-1-184 ~]$
```

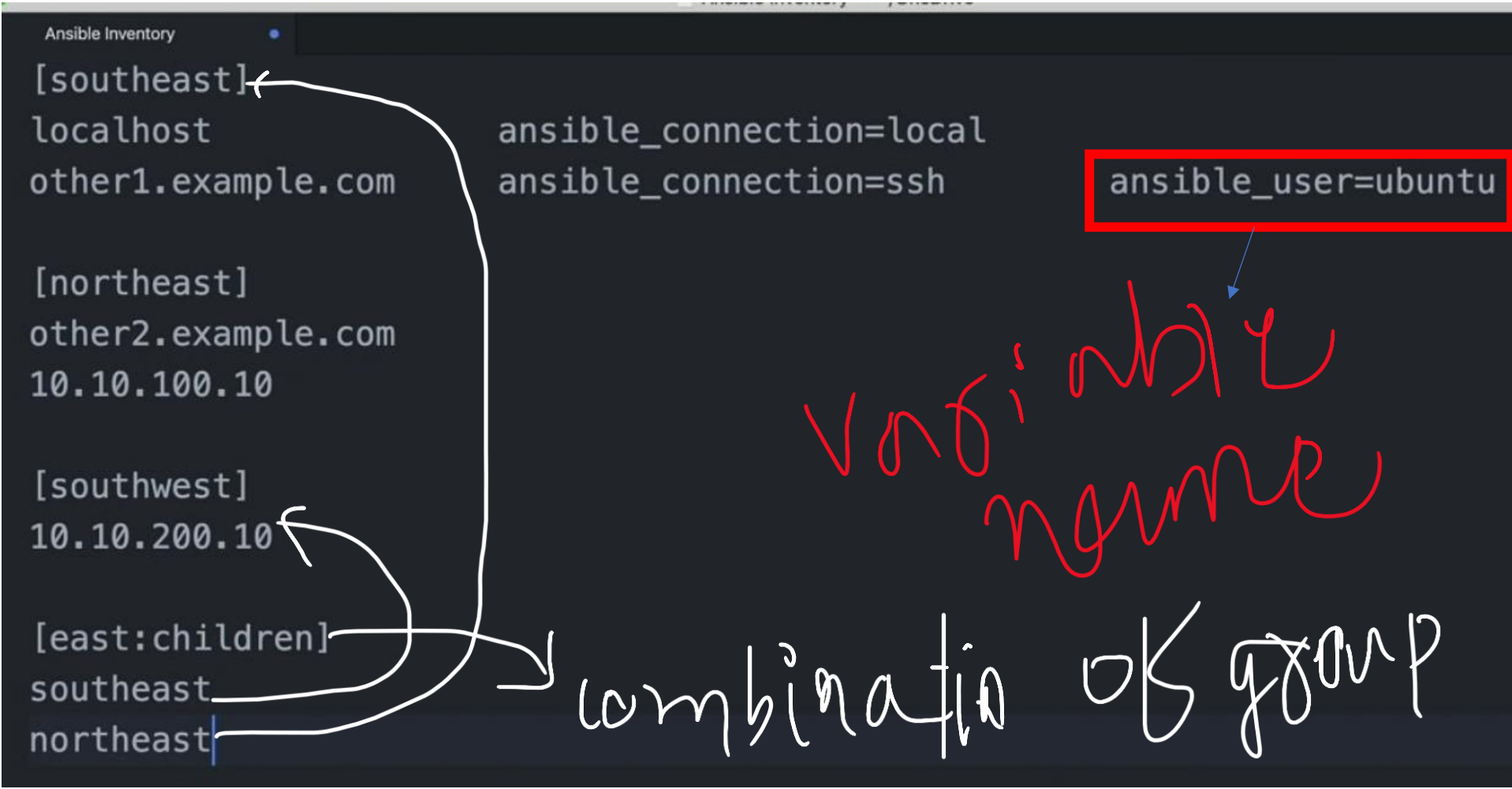
```
$ sudo apt update
$ sudo apt install software-properties-common
$ sudo apt-add-repository --yes --update ppa:ansible/ansible
$ sudo apt install ansible
```

Challenge: Install Ansible

- Install Ansible on the supported platform of your choice
- Verify the version of Ansible you installed
- Use Ansible to perform a ping test against localhost

Working with hosts and Variables in Ansible

Mainly used in inventory file for our hosts.



Here, all the square brackets[northeast.....] are all groups by which we can access a number of hosts.

localhost ansible_connection=local ===== here we only need to use local to access
other1.example.com ansible_connection=ssh ===== here we can use ssh to access the host

[east:children] ===== it is like a class/group where we can combine multiple group

```
[webservers]
web1 ansible_host=192.168.1.10
web2 ansible_host=192.168.1.11

[dbservers]
db1 ansible_host=192.168.1.20
db2 ansible_host=192.168.1.21

[production:children]
webservers
dbservers
```

MODULES

```
TonySequeira@ANTHONYs-MacBook-Pro ~ % ansible localhost -m find -a "paths=Downloads file_type=file"
localhost | SUCCESS => {
  "changed": false,
  "examined": 4,
  "files": [
    {
      "atime": 1592022585.581509,
      "ctime": 1592022585.6932812,
      "inode": 16777328
```


Working With Playbooks in Ansible

Mainly used in inventory file for our hosts.

```
TonySequeira@ANTHONYs-MacBook-Pro ~ % ansible localhost -m find -a "paths=Downloads file_type=file"
localhost | SUCCESS => {
  "changed": false,
  "examined": 4,
  "files": [
    {
      "atime": 1592022585.581509,
      "ctime": 1592022585.6932812,
      "inode": 16777328
```

first.yml

```
---
- name: "My first play"
  hosts: localhost

  tasks:

    - name: "test reachabiliy"
      ping:

    - name: "install stress"
      homebrew:
        name: stress
        state: present
```

Challenge: Write a Playbook in Ansible

- Create two text files in the directory of your choosing
- Create a playbook with a single play that consists of two tasks run against your local Ansible control node
- The first task should ping the local Ansible system
- The second task should display the details of the two text files
- Ensure you run the playbook so that you can see the output of the modules used

File search:

```
sudo ansible localhost -m find -a "paths=. file_type=file patterns=shan.yml"
```

run yml file:

```
ansible-playbook wifi_ping.yml
```

Ping your device task:

```
- name: ping to the local host
  hosts: localhost
  tasks:
    - name: ping local
      ping:
```



1. Create an inventory file: tells the particular device ip to manage:
touch inventory.ini



What is an Ansible Playbook?

- A **Playbook** is a YAML file containing one or more “plays.”
- Each **play** defines:
 - Which **hosts** to run on,
 - What **tasks** to perform on those hosts,
 - And how to perform them.
- Playbooks automate complex workflows and configurations.



What is an Inventory in Ansible?

An **inventory** is a file where you list all the machines (hosts) Ansible will manage.



Ping other hosts

Create host file /etc/ansible
nano <filename.ini>

insert:

[myrouter]

10.11.11.1 ansible_user=admin ansible_password=yourpassword
ansible_connection=ssh

Create two text files and ping localhost

- name: Create two text files

hosts: localhost

tasks:

- name: ping localhost

ping:

- name: Create file1.txt

copy:

content: "This is file 1"

dest: "./file1.txt"

- name: Create file2.txt

copy:

content: "This is file 2"

dest: "./file2.txt"

```
---
- name: "Challenge answers"
  hosts: localhost

  tasks:

    - name: "test reachabiliy"
      ping:

    - name: "find module"
      find:
        path: ~/Challenges
        file_type: file
```

Any file present inside the path
(challenges) , it will just find.

run to check the file:

ansible-playbook <filename.yml> -v



How ansible access remote-system

- For Linux/Unix (SSH)
- For Windows (WinRM)

Automation and Orchestration

- Automation refers to a single task.
- Orchestration refers to the management of many automated tasks.
- Often a complicated ordering with dependencies.

```
---
- name: "Orchestration Example"
  hosts: logicservers
  serial: 1

  tasks:

    - name: "Shutdown Server"
      debug:
        msg: "Shutdown {{ inventory_hostname }}"

    - name: "Upgrade Firmware"
      debug:
        msg: "Upgrade {{ inventory_hostname }}"

    - name: "Start Server"
      debug:
        msg: "Start {{ inventory_hostname }}"
```

```
[logicservers]
server1  ansible_host=127.0.0.1  ansible_connection=local  deprecation_warnings=false
server2  ansible_host=127.0.0.2  ansible_connection=local  INTERPRETER_PYTHON=auto_silent
server3  ansible_host=127.0.0.3  ansible_connection=local
server4  ansible_host=127.0.0.4  ansible_connection=local
```

```
TonySequeira@ANTHONYs-MacBook-Pro Documents % vim myhosts
TonySequeira@ANTHONYs-MacBook-Pro Documents % ansible-playbook orchestrate.yml -i myhosts

PLAY [Orchestration Example] *****
```



Automation vs. Orchestration: Key Differences & Tools

Automation and orchestration are critical in DevOps, Cloud, and IT Ops, but they serve different purposes. Here’s a breakdown:

1. Automation (Task-Level)

Definition: Automating individual tasks (e.g., restarting a service, installing software).

Goal: Replace manual, repetitive work with scripts/tools.

Examples of Automation

- Ansible (Run ad-hoc commands or playbooks)
ansible webserver -m service -a "name=nginx state=restarted"
- Bash/Python Scripts (Custom automation)
- Cron Jobs (Scheduled tasks)

2. Orchestration (Workflow-Level)

Definition: Coordinating multiple automated tasks across systems to achieve a larger goal.

Goal: Manage complex workflows (e.g., deploying an app, scaling cloud resources).

Examples of Orchestration

- Kubernetes (K8s) – Manages containers, scaling, networking.
- Terraform – Provisions cloud infrastructure in order.
- CI/CD Pipelines (Jenkins, GitHub Actions) – Automates testing → deployment.

Key Differences

Feature	Automation	Orchestration
Scope	Single task	Multi-step workflow
Tools	Ansible, Bash	Kubernetes, Terraform
Example	Install Nginx	Deploy full app (DB + Web + LB)



Orchestration in Ansible

Orchestration refers to coordinating and managing multiple tasks, plays, or playbooks across different systems in a specific order to achieve a desired state. It involves:

Managing dependencies between tasks.

Controlling the order of operations across multiple hosts.

Handling rolling updates, canary deployments, and complex workflows.

Serialization in Ansible

Serialization is a subset of orchestration where tasks are executed one after another (sequentially) rather than in parallel. This is controlled using:

serial keyword (to limit how many hosts run at once).

strategy: linear (default, runs task on all hosts before moving to the next task).

strategy: free (allows hosts to run independently).

Key Difference

Orchestration = Managing workflows across multiple systems (e.g., deploying app servers only after the database is ready).

Serialization = Controlling how many hosts execute a task at a time (e.g., updating servers in batches of 2).



Ansible For System Configuration

- Designed to be minimal in nature
- Agentless
- SSH
- State Driven – It can examine the state of machine and it can decide whether or not actions need to be taken
- Idempotent – we can safely execute actions with ansible against systems and not have detrimental effect on those systems.

Orchestration

- name: Test Local Ansible Orchestration

hosts: local

gather_facts: yes

become: yes # Run as sudo if needed means root level access

tasks:

- name: Create a test directory

file:

path: /tmp/ansible_test

state: directory

mode: '0755'

- name: Write a test file

copy:

content: "Hello, Ansible running locally!"

dest: /tmp/ansible_test/test.txt

- name: Install a package (e.g., htop)

apt: # For Debian/Ubuntu

name: htop

state: present

when: ansible_os_family == 'Debian'

- name: Print a message

debug:

msg: "Ansible is working locally!"



e.g: Ansible For System Configuration

```
---
- hosts: all
  become: yes
  name: "NTP configuration"

  tasks:
    - name: "Ensure NTP is installed"
      apt:
        name:
          - ntp
        state: present

    - name: "Ensure NTP is started now and at boot"
      service:
        name: ntp
        state: started
        enabled: yes
```

hosts: all

Meaning: Specifies the target hosts for the playbook. U can also define it for a single system like webserver.

become: yes

Meaning: Enables privilege escalation (run tasks as root or another privileged user).

Details:

Equivalent to using sudo on Linux.

Required for tasks like installing packages (apt) or managing services (service).

Taks

```
- name: "Ensure NTP is installed"
```

```
apt:
```

```
  name:
```

```
    - ntp
```

```
  state: present
```

name (task description): Explains the task's purpose (for logging/debugging).

apt: Ansible module to manage packages on Debian/Ubuntu.

name: ntp: Specifies the package to install.

state: present: Ensures the package is installed (alternatives: absent, latest).

```
- name: "Ensure NTP is started now and at boot"
```

```
service:
```

```
  name: ntp                //chronyd -- same
```

```
  state: started
```

```
  enabled: yes
```

service: Ansible module to manage services.

name: ntp: Targets the ntp service.

state: started: Ensures the service is running immediately.

enabled: yes: Ensures the service starts automatically at boot.

e.g: Ansible For System Configuration

```
---
- name: "React with Change Example"
  hosts: webserver
  serial: 1

  tasks:

    - name: "Install nginx"
      debug:
        msg: "Install nginx on: {{ inventory_hostname }}"

    - name: "Upgrade nginx"
      debug:
        msg: "Upgrade nginx on: {{ inventory_hostname }}"

    - name: "Configure nginx"
      debug:
        msg: "Start {{ inventory_hostname }}"
      notify: restart nginx
      # changed_when: True

    - name: "Verify nginx"
      debug:
        msg: "Verify: {{ inventory_hostname }}"

  handlers:
    - name: restart nginx
      debug:
        msg: "CALLED HANDLER FOR RESTART"
```

```
TonySequeira@ANTHONYs-MacBook-Pro Documents % ansible-playbook change.yml -i myhosts
```

In Ansible, handlers are special tasks that only execute when notified by another task. However, if the `changed_when` condition is not properly configured, the handler will not trigger, even if a task reports a change.

Task: Updates a config file only if changes are detected.

Handler: Restarts the service when notified.

Verification: Use `-v` or `--check` to validate behavior before full execution.

i. Remove changed_when (Recommended)

Let Ansible detect changes naturally:

tasks:

```
- name: Update config file # Descriptive task name
```

```
  template:
```

```
    src: config.j2      # Jinja2 template source
```

```
    dest: /etc/app/config.conf # Destination path on remote system
```

```
    owner: root        # Optional: Set file ownership
```

```
    group: root        # Optional: Set file group
```

```
    mode: '0644'       # Optional: Set file permissions (e.g., rw-r--r--)
```

```
  notify: Restart service # Notifies the handler if the file changes
```

ii. Use changed_when Correctly

If you must use changed_when, ensure it evaluates to true when a change occurs:

yaml

tasks:

```
- name: Update config file (conditional change)
```

```
  template:
```

```
    src: config.j2
```

```
    dest: /etc/app/config.conf
```

```
  notify: Restart service
```

```
  changed_when: "'modified' in template_result"
```

```
  # Custom logic to detect    changes
```

iii. Force Handler Execution

If you always want the handler to run (even without changes), use meta:

tasks:

```
- name: Force handler execution
```

```
  meta: flush_handlers
```

Infrastructure Management With Ansible



**Build Infrastructure
from the Ground Up**



**Manage Physical
Infrastructure**



**Automate
Virtualization**



**Wide OS
Support**

How Ansible Works with

i. Build Cloud Infrastructure Automatically

- Use Ansible playbooks to create servers, networks, and storage in the cloud—like magic!
- No manual setup: Ansible can provision the hardware (VMs, containers, etc.) for you.

ii. Dynamic Inventory = Real Time Tracking

- Ansible's dynamic inventory updates itself when you add/remove cloud servers.
- Example: If AWS creates a new VM, Ansible instantly knows and can manage it.

iii. Manage Virtualization Easily

- Ansible sets up the physical parts (servers, networks) first.
- Then, it automates virtualization (like Docker, Kubernetes, or VMware) on top.

iv. Works with Any OS

- In the cloud, you might use Linux, Windows, or macOS.
- Ansible supports all of them, making it perfect for mixed environments.

Repeating Tasks Across Fleets With Ansible



An ad-hoc command is a quick, one-time task you run with Ansible—without writing a full playbook.

Think of it like:

- Typing a single command in the terminal to do something fast.
- Useful for quick checks or simple changes (e.g., restarting a service, checking disk space).

Example Commands:

Ping all servers (check connectivity):

```
ansible all -m ping
```

for particular inventory: add “filename” at the end

Restart a service (e.g., Nginx):

```
ansible webservers -m service -a "name=nginx state=restarted" -b  
(-b = run as admin/sudo)
```

Check disk space:

```
ansible db_servers -m shell -a "df -h"
```

Install a package (e.g., htop):

```
ansible all -m apt -a "name=htop state=present" -b
```

When to Use Ad-Hoc vs. Playbooks?

Ad-Hoc Commands

Quick, one-time tasks

Testing/debugging

Just do this now

Playbooks

Repeatable, complex setups

Full automation

Always do this correctly

Execute the playbook on **all servers**

```
ansible-playbook -i hosts.ini main.yml
```

Execute the playbook on **single servers**

```
ansible-playbook -i hosts.ini main.yml --limit webservers
```

```
- name: "React with Change Example"
  hosts: webservers
  serial: 1

  tasks:
    - name: "Install nginx"
      debug:
        msg: "Install nginx on: {{ inventory_hostname }}"

    - name: "Upgrade nginx"
      debug:
        msg: "Upgrade nginx on: {{ inventory_hostname }}"

    - name: "Configure nginx"
      debug:
        msg: "Start {{ inventory_hostname }}"
      notify: restart nginx
```

Serial: 1

Everything run at one host then move to the next host

Strategy: free (available inside ansible)

Each host runs independently.

As soon as a host finishes a task, it can proceed to the next one — without waiting for the other hosts

Folk tell in how many systems, it should execute:
Ansible-playbook file.yml -f 10 myhost