Ansible Detailed Notes

Table of Contents

---

1. Introduction to Ansible

Theoretical

Ansible is an open-source automation tool designed for configuration management, application deployment, and task orchestration. As a configuration management tool, it ensures systems are configured to a desired state consistently and idempotently. Ansible is agentless, using SSH (for Linux/Unix) or WinRM (for Windows) to communicate with

managed nodes, and it employs YAML for defining automation tasks, making it accessible to both developers and system administrators.

Key Characteristics:

- **Agentless**: No software installation required on managed nodes.
- **Idempotent**: Tasks are applied only when needed, avoiding redundant changes.
- **Push-Based**: Configurations are pushed from the control node to managed nodes.
- **Human-Readable**: Uses YAML, which is simple and easy to understand.

Practical

Command:

ansible --version

- Displays the Ansible version and configuration details.

Example:

```
$ ansible --version
ansible [core 2.15.3]
  config file = /etc/ansible/ansible.cfg
  python version = 3.9.2
```

---

2. Why Ansible for Configuration Management?

Theoretical

Ansible is a preferred configuration management tool due to its simplicity, flexibility, and robust feature set. Unlike other tools like Puppet or Chef, which require agents and complex setups, Ansible leverages existing SSH infrastructure, reducing overhead. Its YAML-based playbooks enable Infrastructure as Code (IaC), allowing version control and collaboration.

Reasons:

- **Ease of Use**: Minimal learning curve with YAML syntax.
- **No Agents**: Reduces complexity and security risks.
- **Extensibility**: Supports custom modules and plugins.
- **Community Support**: Large community and extensive documentation.
- **Multi-Platform**: Manages Linux, Windows, and network devices.

Practical

Command:

ansible all -m ping

- Tests connectivity to all hosts in the inventory, verifying SSH setup.

Example:

```
$ ansible all -m ping
webserver1 | SUCCESS => {
   "changed": false,
   "ping": "pong"
}
```

---

3. Benefits for DevOps

Theoretical

Ansible aligns with DevOps principles by enabling automation, collaboration, and continuous delivery. It bridges the gap between development and operations by providing a unified tool for managing infrastructure and applications.

Benefits:

- **Infrastructure as Code**: Define and version infrastructure in YAML.
- **Consistency**: Ensures uniform configurations across environments.
- **Automation**: Reduces manual errors and speeds up deployments.
- **Scalability**: Manages small to large-scale infrastructures.
- **Integration**: Works with CI/CD pipelines (e.g., Jenkins, GitLab).
- **Collaboration**: Playbooks are shareable and understandable by cross-functional teams.

Practical

Example Use Case: Automating the configuration of a web server cluster ensures consistent package versions, configurations, and service states across all nodes.

---

4. Installation of Ansible

Theoretical

Ansible is typically installed on a control node (e.g., a Linux machine). Managed nodes require no additional software, only SSH access and Python (for Linux) or PowerShell (for

Windows). Ansible supports various installation methods, including package managers, pip, and source.

Practical

Prerequisites

- **Control Node**: Linux (e.g., Ubuntu, CentOS), macOS, or Windows with WSL.
- **Python**: Version 3.8+ recommended.
- **SSH**: For Linux/Unix managed nodes.
- **WinRM**: For Windows managed nodes (if applicable).

Installation Steps (Ubuntu) :

Update Package Index:
sudo apt update

Install Software Properties Common:
sudo apt install software-properties-common

Add Ansible PPA (for the latest version):
sudo add-apt-repository --yes --update ppa:ansible/ansible

Install Ansible:
sudo apt install ansible

Verify Installation:
ansible --version

Installation Steps (CentOS/RHEL)

Install EPEL Repository:
sudo yum install epel-release

Install Ansible:
sudo yum install ansible

Verify Installation:
ansible --version

Installation via pip (Cross-Platform)

Install pip:
sudo apt install python3-pip # Ubuntu
sudo yum install python3-pip # CentOS

Install Ansible:
pip3 install ansible

Verify Installation:
ansible --version

Post-Installation

Create Ansible Configuration File (optional):
mkdir ~/ansible
touch ~/ansible/ansible.cfg

Sample `ansible.cfg`:
[defaults]
inventory = ./inventory
remote_user = ansible
private_key_file = ~/.ssh/ansible_key
host_key_checking = False

Example:

$ sudo apt update
$ sudo apt install software-properties-common
$ sudo add-apt-repository --yes --update ppa:ansible/ansible
$ sudo apt install ansible
$ ansible --version
ansible [core 2.15.3]

---

5. Ansible Architecture

Theoretical

Ansible's architecture is straightforward and agentless, consisting of:

- **Control Node**: The machine running Ansible, executing commands and playbooks.
- **Managed Nodes**: Remote systems managed via SSH (Linux) or WinRM (Windows).
- **Inventory**: A file or dynamic source listing managed nodes.
- **Modules**: Reusable units of code for tasks (e.g., `apt`, `copy`).
- **Playbooks**: YAML files defining automation workflows.
- **Plugins**: Extend functionality (e.g., connection, callback plugins).
- **Ansible Core**: The engine orchestrating tasks.

Communication:

- Ansible pushes configurations to managed nodes using SSH or WinRM.

- No agents or daemons run on managed nodes, reducing overhead.

Diagram:

```
Control Node --> SSH/WinRM --> Managed Nodes
  |                    |
Inventory              Modules
  |                    |
Playbooks              Plugins
```

Practical

Command:

ansible-config dump

- Displays Ansible configuration settings.

Example:

```
$ ansible-config dump | grep INVENTORY
INVENTORY_ENABLED = ['host_list', 'script', 'auto', 'yaml', 'ini']
```

---

6. Features of Ansible

Theoretical

Ansible's feature set makes it a powerful configuration management tool:

- **Agentless**: Uses existing SSH/WinRM infrastructure.
- **YAML Playbooks**: Simple, human-readable automation scripts.
- **Idempotency**: Ensures consistent system states.
- **Modules**: Pre-built tasks for system management.
- **Extensibility**: Custom modules/plugins in Python.
- **Ansible Galaxy**: Repository for reusable roles/collections.
- **Ansible Vault**: Encrypts sensitive data.
- **Dynamic Inventory**: Integrates with cloud providers (e.g., AWS, Azure).
- **Error Handling**: Robust mechanisms for task failures.
- **Policy as Code**: Enforces compliance and security policies.

Practical

Command:

ansible-doc -l

- Lists all available modules.

Example:

$ ansible-doc -l | grep apt
apt Installs, upgrades, or removes packages using apt
apt_key Add or remove an apt key
apt_repository Add or remove an apt repository

---

7. Passwordless Authentication

Theoretical

Ansible relies on SSH for secure communication with managed nodes. Passwordless authentication using SSH keys is preferred for scalability and security. Alternatively, password-based authentication can be used, but it's less secure and requires manual intervention.

Practical

Steps for Passwordless Authentication
Generate SSH Key Pair:
ssh-keygen -t rsa -b 4096 -f ~/.ssh/ansible_key

- Press Enter for no passphrase (or set one for added security).

Copy Public Key to Managed Nodes:
ssh-copy-id -i ~/.ssh/ansible_key.pub ansible@managed_node

- Alternatively, append the public key to `~/.ssh/authorized_keys` on the managed node.

Update Ansible Configuration:
Edit `ansible.cfg`:
[defaults]
private_key_file = ~/.ssh/ansible_key
Test Connectivity:
ansible all -m ping

Password-Based Authentication (Not Recommended)
Specify in inventory:
[webservers]
webserver1 ansible_user=admin ansible_password=secret

Or use `--ask-pass`:
ansible all -m ping --ask-pass

Example:

```
$ ssh-keygen -t rsa -b 4096 -f ~/.ssh/ansible_key
Generating public/private rsa key pair.
Enter passphrase (empty for no passphrase):
...
$ ssh-copy-id -i ~/.ssh/ansible_key.pub ansible@webserver1
...
$ ansible webserver1 -m ping
webserver1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

---

8. YAML Architecture

Theoretical

YAML (YAML Ain't Markup Language) is the foundation of Ansible's playbooks, inventories, and variable files. Its simplicity and hierarchical structure make it ideal for defining complex configurations.

Key YAML Elements:

- **Key-Value Pairs**: `key: value`
- **Lists**: `- item`
- **Nested Structures**: Indented blocks.
- **Comments**: `# Comment`.

Practical

Example YAML:

```
---
# Playbook example
- name: Install Nginx
  hosts: webservers
  tasks:
    - name: Install Nginx package
      apt:
        name: nginx
```

```
    state: present
```

Validation:

```
ansible-playbook playbook.yml --syntax-check
```

Best Practices:

- Use 2-space indentation.
- Avoid tabs; use spaces.
- Keep YAML files concise and modular.

---

9. Inventory Files

Theoretical

Inventory files define the hosts and groups Ansible manages. They can be static (INI/YAML) or dynamic (generated by scripts or cloud plugins).

Types:

- **Static Inventory**: Manually defined hosts.
- **Dynamic Inventory**: Generated from sources like AWS, Azure, or scripts.

Practical

Static Inventory (INI)

```
[webservers]
webserver1 ansible_host=192.168.1.10 ansible_user=ansible
webserver2 ansible_host=192.168.1.11 ansible_user=ansible

[dbservers]
dbserver1 ansible_host=192.168.1.20 ansible_user=ansible
```

Static Inventory (YAML)

```
all:
  children:
    webservers:
      hosts:
        webserver1:
          ansible_host: 192.168.1.10
          ansible_user: ansible
        webserver2:
```

```
        ansible_host: 192.168.1.11
        ansible_user: ansible
    dbservers:
      hosts:
        dbserver1:
          ansible_host: 192.168.1.20
          ansible_user: ansible
```

Dynamic Inventory
Example (AWS):
ansible-inventory -i aws_ec2.yml --graph

Commands:

List Inventory:
ansible-inventory --list

Graph Inventory:
ansible-inventory --graph

Example:

```
$ ansible-inventory --list
{
    "_meta": {
        "hostvars": {}
    },
    "all": {
        "children": ["ungrouped", "webservers", "dbservers"]
    },
    "webservers": {
        "hosts": ["webserver1", "webserver2"]
    },
    "dbservers": {
        "hosts": ["dbserver1"]
    }
}
```

---

10. Ad-Hoc Commands

Theoretical

Ad-hoc commands are one-off tasks executed directly from the command line using the `ansible` command. They are useful for quick tasks like checking system status, installing packages, or rebooting servers.

Practical

Syntax:

ansible <host_pattern> -m <module> -a "<arguments>"

Common Modules:

- `ping`: Test connectivity.
- `command`: Run shell commands.
- `file`: Manage files/directories.
- `apt`/`yum`: Manage packages.

Examples:

Check Disk Space:
ansible all -m command -a "df -h"

Install Package:
ansible webservers -m apt -a "name=nginx state=present" --become

Create Directory:
ansible webservers -m file -a "path=/app state=directory"

Output:

$ ansible webservers -m command -a "df -h"
webserver1 | SUCCESS | rc=0 >>
Filesystem Size Used Avail Use% Mounted on
/dev/sda1 20G 5.0G 15G 26% /
...

---

11. Playbooks

Theoretical

Playbooks are YAML files that define a series of tasks to achieve a desired system state. They are the core of Ansible's configuration management, allowing complex workflows to be scripted.

Practical

Example Playbook (`install_nginx.yml`):

```
---
- name: Install and configure Nginx
  hosts: webservers
  become: yes
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
    - name: Start Nginx
      service:
        name: nginx
        state: started
        enabled: yes
```

Command:

```
ansible-playbook install_nginx.yml
```

Dry Run:

```
ansible-playbook install_nginx.yml --check
```

---

12. Playbook Structure

Theoretical

A playbook typically includes:

- **Name**: Descriptive name.
- **Hosts**: Target hosts/groups.
- **Become**: Sudo escalation.
- **Vars**: Inline or included variables.
- **Tasks**: List of tasks using modules.
- **Handlers**: Tasks triggered by changes.

Practical

Example:

```
---
- name: Configure web server
  hosts: webservers
  become: yes
  vars:
    http_port: 80
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
      notify: Restart Nginx
    - name: Copy configuration
      copy:
        src: nginx.conf
        dest: /etc/nginx/nginx.conf
      notify: Restart Nginx
  handlers:
    - name: Restart Nginx
      service:
        name: nginx
        state: restarted
```

Command:

```
ansible-playbook webserver.yml
```

---

13. Roles and Folder Structure

Theoretical

Roles modularize playbooks into reusable components, improving maintainability and scalability.

Folder Structure:

```
my_role/
├── defaults/
│   └── main.yml       # Default variables (low precedence)
├── files/
│   └── nginx.conf     # Static files
├── handlers/
│   └── main.yml       # Handlers
├── tasks/
│   └── main.yml       # Main tasks
```

```
├── templates/
│   └── nginx.j2       # Jinja2 templates
├── vars/
│   └── main.yml       # Variables (high precedence)
└── meta/
    └── main.yml       # Role metadata
```

Purpose:

- **defaults**: Low-precedence variables.
- **files**: Static files for copying.
- **handlers**: Tasks triggered by `notify`.
- **tasks**: Core tasks.
- **templates**: Dynamic files with Jinja2.
- **vars**: High-precedence variables.
- **meta**: Role dependencies.

Practical

Create Role:

ansible-galaxy role init my_role

Example Role (`roles/webserver/tasks/main.yml`):

```
---
- name: Install Nginx
  apt:
    name: nginx
    state: present
- name: Copy Nginx config
  template:
    src: nginx.j2
    dest: /etc/nginx/nginx.conf
  notify: Restart Nginx
```

Playbook Using Role:

```
---
- name: Deploy web server
  hosts: webservers
  roles:
    - my_role
```

Command:

ansible-playbook playbook.yml

---

14. Ansible Galaxy

Theoretical

Ansible Galaxy is a community repository for sharing roles and collections, enabling reuse of pre-built automation content.

Practical

Commands:

Search:
ansible-galaxy search nginx

Install Role:
ansible-galaxy role install geerlingguy.nginx

List Roles:
ansible-galaxy role list

Example:

$ ansible-galaxy role install geerlingguy.nginx
- downloading role 'nginx', owned by geerlingguy
- role 'geerlingguy.nginx' successfully installed

---

15. Ansible Collections

Theoretical

Collections bundle roles, modules, and plugins into a single package, improving modularity and distribution.

Structure:

namespace.collection_name/
├── docs/
├── plugins/
│ ├── modules/
│ ├── inventory/
│ └── connection/

```
├── roles/
└── README.md
```

Practical

Commands:

Install Collection:
ansible-galaxy collection install community.general

List Collections:
ansible-galaxy collection list

Example:

$ ansible-galaxy collection install community.general
Installing 'community.general:8.3.0' to
'/home/user/.ansible/collections/ansible_collections/community/general'

---

16. Ansible Vault

Theoretical

Ansible Vault encrypts sensitive data (e.g., passwords, API keys) to secure playbooks and variable files.

Practical

Commands:

Create Encrypted File:
ansible-vault create secrets.yml

Edit Encrypted File:
ansible-vault edit secrets.yml

Encrypt File:
ansible-vault encrypt secrets.yml

Run Playbook with Vault:
ansible-playbook playbook.yml --ask-vault-pass

Example (`secrets.yml`):

---

db_password: mysecretpassword

Playbook:

```
---
- name: Use secret
  hosts: dbservers
  vars_files:
    - secrets.yml
  tasks:
    - name: Configure database
      mysql_user:
        name: admin
        password: "{{ db_password }}"
        state: present
```

---

17. Variables and Variable Precedence

Theoretical

Variables enable dynamic configurations. They can be defined in multiple places, with a clear precedence order.

Variable Types:

- **Playbook Vars**: Inline or in `vars_files`.
- **Role Vars**: In `defaults` or `vars`.
- **Inventory Vars**: Host or group variables.
- **Facts**: System information (e.g., `ansible_facts`).
- **Extra Vars**: CLI (`-e`).

Precedence (highest to lowest):

1. Extra variables (`-e`)
2. Task variables
3. Block variables
4. Role variables (`vars/main.yml`)
5. Group variables
6. Host variables
7. Role defaults (`defaults/main.yml`)
8. Facts

Practical

Example:

```
---
- name: Use variables
  hosts: webservers
  vars:
    app_port: 8080
  tasks:
    - name: Configure app
      debug:
        msg: "Application runs on port {{ app_port }}"
```

Command:

```
ansible-playbook playbook.yml -e "app_port=9090"
```

---

18. Conditionals

Theoretical

Conditionals allow tasks to execute based on specific conditions, enhancing playbook flexibility.

Practical

Example:

```
---
- name: Install package based on OS
  hosts: all
  tasks:
    - name: Install Nginx on Ubuntu
      apt:
        name: nginx
        state: present
      when: ansible_os_family == "Debian"
    - name: Install Nginx on CentOS
      yum:
        name: nginx
        state: present
      when: ansible_os_family == "RedHat"
```

Command:

ansible-playbook conditional.yml

---

19. Loops

Theoretical

Loops allow repetitive tasks to be performed over a list of items, reducing playbook verbosity.

Practical

Example:

```
---
- name: Install multiple packages
  hosts: webservers
  tasks:
    - name: Install packages
      apt:
        name: "{{ item }}"
        state: present
      loop:
        - nginx
        - vim
        - curl
```

Command:

ansible-playbook loop.yml

---

20. Delegation

Theoretical

Delegation allows tasks to be executed on a different host than the target, useful for tasks like load balancer updates or database migrations.

Practical

Example:

```
---
- name: Delegate task
```

```
  hosts: webservers
  tasks:
    - name: Update load balancer
      command: /usr/bin/update_lb
      delegate_to: loadbalancer1
```

Command:

ansible-playbook delegate.yml

---

21. Error Handling

Theoretical

Ansible provides mechanisms to handle task failures, ensuring robust automation.

Features:

- **Ignore Errors**: Continue despite failures.
- **Failed When**: Custom failure conditions.
- **Block and Rescue**: Handle errors with recovery tasks.

Practical

Example:

```
---
- name: Handle errors
  hosts: webservers
  tasks:
    - block:
        - name: Task that might fail
          command: /bin/false
      rescue:
        - name: Recovery task
          debug:
            msg: "Task failed, recovering"
      always:
        - name: Always run
          debug:
            msg: "This runs regardless"
```

Command:

ansible-playbook error_handling.yml

---

22. Tags

Theoretical

Tags allow selective execution of tasks or plays, improving playbook efficiency during testing or partial runs.

Practical

Example:

```
---
- name: Tagged playbook
  hosts: webservers
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
      tags: install
    - name: Configure Nginx
      copy:
        src: nginx.conf
        dest: /etc/nginx/nginx.conf
      tags: configure
```

Commands:

Run specific tags:
ansible-playbook playbook.yml --tags install

Skip tags:
ansible-playbook playbook.yml --skip-tags configure

---

23. Policy as Code

Theoretical

Policy as Code enforces compliance and security policies through automation. Ansible ensures systems adhere to standards like CIS, NIST, or organizational rules.

Practical

Example (`security_policy.yml`):

```
---
- name: Enforce security policy
  hosts: all
  become: yes
  tasks:
    - name: Disable root SSH login
      lineinfile:
        path: /etc/ssh/sshd_config
        regexp: '^PermitRootLogin'
        line: 'PermitRootLogin no'
      notify: Restart SSH
    - name: Enable firewall
      service:
        name: ufw
        state: started
        enabled: yes
  handlers:
    - name: Restart SSH
      service:
        name: sshd
        state: restarted
```

Command:

```
ansible-playbook security_policy.yml
```

Validation:

```
ansible-lint security_policy.yml
```

---

24. Ansible as a Configuration Management Tool

Theoretical

As a configuration management tool, Ansible ensures systems are in a desired state by defining configurations in playbooks. It supports:

- **Idempotency**: Applies changes only when needed.

- **Declarative Approach**: Define the end state, not the steps.
- **Version Control**: Store playbooks in Git for traceability.
- **Compliance**: Enforce policies and standards.
- **Multi-Environment**: Manage dev, test, and prod environments consistently.

Comparison with Other Tools:

Feature	Ansible	Puppet	Chef
Agentless	Yes	No	No
Language	YAML	Ruby	Ruby
Setup Complexity	Low	High	High
Learning Curve	Easy	Moderate	Moderate

Practical

Ansible's configuration management is demonstrated through playbooks that install software, configure services, and enforce policies across nodes.

---

25. Sample Project: Configuring a Web Server Cluster

Objective

Configure a cluster of web servers running Nginx with a custom configuration, ensuring consistency and security.

Prerequisites

- Control node with Ansible installed.
- Managed nodes (e.g., Ubuntu servers) with SSH access.
- SSH key-based authentication.

Project Structure

```
web_cluster/
├── inventory.yml
├── ansible.cfg
├── playbooks/
│   └── deploy_web.yml
├── roles/
│   └── webserver/
│       ├── tasks/
│       │   └── main.yml
```

```
│       ├── templates/
│       │   └── nginx.j2
│       ├── handlers/
│       │   └── main.yml
│       └── defaults/
│           └── main.yml
└── files/
    └── index.html
```

Inventory (`inventory.yml`)

```
all:
  children:
    webservers:
      hosts:
        webserver1:
          ansible_host: 192.168.1.10
          ansible_user: ansible
        webserver2:
          ansible_host: 192.168.1.11
          ansible_user: ansible
```

Ansible Configuration (`ansible.cfg`)

```
[defaults]
inventory = ./inventory.yml
remote_user = ansible
private_key_file = ~/.ssh/ansible_key
host_key_checking = False
```

Role Defaults (`roles/webserver/defaults/main.yml`)

```
---
http_port: 80
```

Role Tasks (`roles/webserver/tasks/main.yml`)

```
---
- name: Install Nginx
  apt:
    name: nginx
    state: present
- name: Copy Nginx configuration
  template:
    src: nginx.j2
    dest: /etc/nginx/nginx.conf
  notify: Restart Nginx
```

```
- name: Copy website content
  copy:
    src: index.html
    dest: /var/www/html/index.html
- name: Ensure Nginx is running
  service:
    name: nginx
    state: started
    enabled: yes
```

Role Template (`roles/webserver/templates/nginx.j2`)

```
server {
    listen {{ http_port }};
    server_name localhost;
    root /var/www/html;
    index index.html;
}
```

Role Handler (`roles/webserver/handlers/main.yml`)

```
---
- name: Restart Nginx
  service:
    name: nginx
    state: restarted
```

Static File (`files/index.html`)

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome</title>
</head>
<body>
    <h1>Welcome to the Web Server!</h1>
</body>
</html>
```

Playbook (`playbooks/deploy_web.yml`)

```
---
- name: Deploy web server cluster
  hosts: webservers
  become: yes
  roles:
    - webserver
```

Execution

Validate Syntax:
ansible-playbook playbooks/deploy_web.yml --syntax-check

Dry Run:
ansible-playbook playbooks/deploy_web.yml --check

Run Playbook:
ansible-playbook playbooks/deploy_web.yml

1. **Verify**:
    - Access `http://192.168.1.10` and `http://192.168.1.11` to confirm the website is live.

Output:

```
$ ansible-playbook playbooks/deploy_web.yml
PLAY [Deploy web server cluster] ***********************************************
TASK [webserver : Install Nginx] ***********************************************
changed: [webserver1]
changed: [webserver2]
...
PLAY RECAP *********************************************************************
webserver1 : ok=4    changed=3    unreachable=0    failed=0
webserver2 : ok=4    changed=3    unreachable=0    failed=0
```

---

26. Best Practices

- **Modularize with Roles**: Break playbooks into reusable roles.
- **Use Version Control**: Store playbooks in Git.
- **Validate Playbooks**: Use `ansible-lint` and `--syntax-check`.
- **Encrypt Sensitive Data**: Use Ansible Vault.
- **Minimize Privileges**: Use `become` only when necessary.
- **Document Playbooks**: Add clear comments and names.
- **Test in Staging**: Use `--check` mode before production.
- **Leverage Tags**: For selective task execution.
- **Optimize Performance**: Use async tasks for long-running operations.

---

27. Troubleshooting and Debugging

Theoretical

Ansible provides tools to diagnose issues in playbooks and tasks.

Practical

Commands:

Verbose Output:
ansible-playbook playbook.yml -v

- Add more `v` for increased verbosity (`-vv`, `-vvv`).

Debug Module:
- name: Print variable
  debug:
    msg: "Value is {{ my_var }}"

Check Mode:
ansible-playbook playbook.yml --check

Syntax Check:
ansible-playbook playbook.yml --syntax-check

Ansible Lint:
ansible-lint playbook.yml

Common Issues:

- **SSH Errors**: Verify key-based authentication and `ansible.cfg`.
- **Module Failures**: Check module parameters with `ansible-doc <module>`.
- **Inventory Issues**: Validate with `ansible-inventory --list`.
- **Variable Errors**: Use `debug` to inspect values.

Example:

$ ansible-playbook playbook.yml -vvv
...
TASK [Install Nginx] ***
changed: [webserver1] => {"changed": true, "name": "nginx", "state": "present"}
