

Visual Guide to Computer Systems

Ansible A Visual Guide

Sample Book — Chapters 1 to 2

Foundations, Reliable Playbooks, Maintainable Automation, and Safe Operations

Reiner Braun

Introduction

This book is designed for readers who are taking their first steps with Ansible, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

Read the chapters in order when you are new to Ansible. The sequence builds the required background step by step: first the execution model and practice environment, then playbook construction, reusable project structure, and finally safe team and fleet operations.

At the beginning of each chapter, review **What You'll Be Able to Do** before reading the chapter itself. It identifies the practical outcomes to keep in mind as you work through the explanations and examples.

Diagrams are used to clarify relationships, processes, structures, boundaries, and execution sequences that are harder to picture from prose alone. Read the surrounding text as the complete explanation; the image is a supporting view, not a replacement for the main text.

Run the code and hands-on examples yourself whenever a suitable disposable environment is available. Before each command, confirm the stated prerequisites and target scope, then compare the actual result with the expected result in the text.

Use each **Chapter Summary** and set of **Review Questions** to confirm that you understand and can recall the essential fundamentals. Return to the relevant section when an answer depends on memorized wording rather than a clear mental model.

Advanced-topic overviews do not need to be memorized at this stage. Treat them as signposts: they show what a capability is for, where it fits, and which topics to revisit when a future project requires them.

Table of Contents

Introduction

How to Use This Book

Part I — Build the Correct Mental Model

Chapter 1 — Ansible in One Mental Model

- What Ansible is designed to solve
- The three parts of a normal Ansible environment
- The vocabulary of automation content
- Agentless does not mean prerequisite-free
- ansible-core, the ansible package, and collections
- What happens during a playbook run
- Reading task results
- Desired state is the central idea
- The mental model to retain

Chapter 2 — Create a Safe Practice Environment

- Control-node and managed-node responsibilities
- Choose the package deliberately
- Install with pipx
- Use a virtual environment as an alternative
- Confirm the installation
- Establish SSH independently of Ansible
- Keep configuration inside the project
- Add two disposable targets
- Keep credentials out of project configuration
- Work with version awareness
- Diagnose the environment in the right order

Part I — Build the Correct Mental Model

Chapter 1 — Ansible in One Mental Model

What You'll Be Able to Do

- Explain how the control node, inventory, and managed nodes work together during an Ansible run.
- Distinguish playbooks, plays, tasks, modules, roles, plugins, and collections.
- Interpret common result states and explain why desired state and idempotency matter.

Ansible takes automation instructions from a control node, selects managed nodes from inventory, and runs ordered tasks to move those nodes toward a declared state. That sentence is the central model for the rest of this guide.

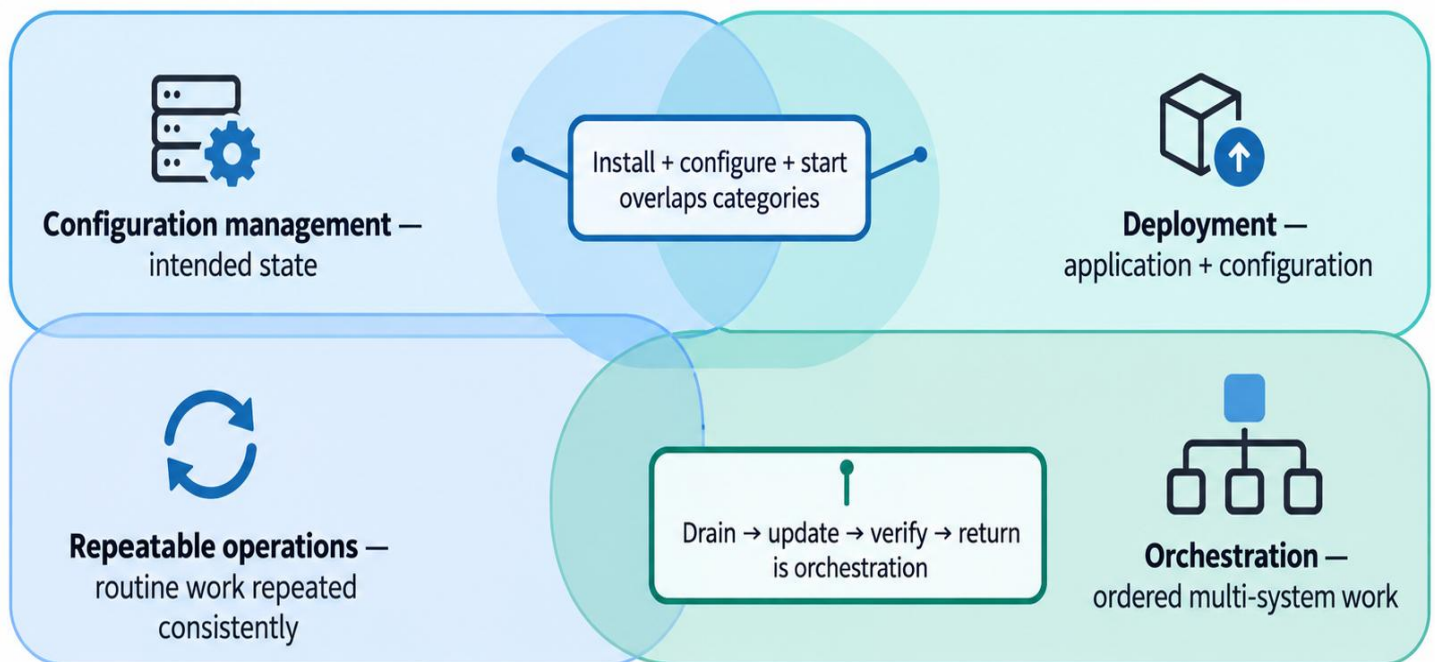
Ansible is useful because systems work is full of procedures that must be performed consistently: install a package, create an account, place a configuration file, start a service, or verify that an endpoint responds. A human can perform each step through an SSH session, but manual work becomes slow and inconsistent as the number of systems and repetitions grows. Ansible turns the procedure into reviewed, repeatable automation.

What Ansible is designed to solve

Ansible is commonly used for four related kinds of work:

Chapter 1: Ansible in One Mental Model

1-1 How Ansible Work Categories Overlap



Overlap is normal

001

Figure 1-1: How configuration management, reusable operations, deployment, and orchestration overlap

- **Configuration management** keeps operating-system and application settings in an intended state.
- **Reusable operations** perform routine work, such as applying a standard change to a fleet of servers.
- **Deployment** places application artifacts and supporting configuration on target systems.
- **Orchestration** coordinates ordered work across several systems or system tiers.

These categories overlap. A playbook that installs a web package, writes its configuration, and starts its service is both configuration management and deployment. A playbook that removes one web server from a load balancer, updates it, verifies it, and returns it to service is orchestration.

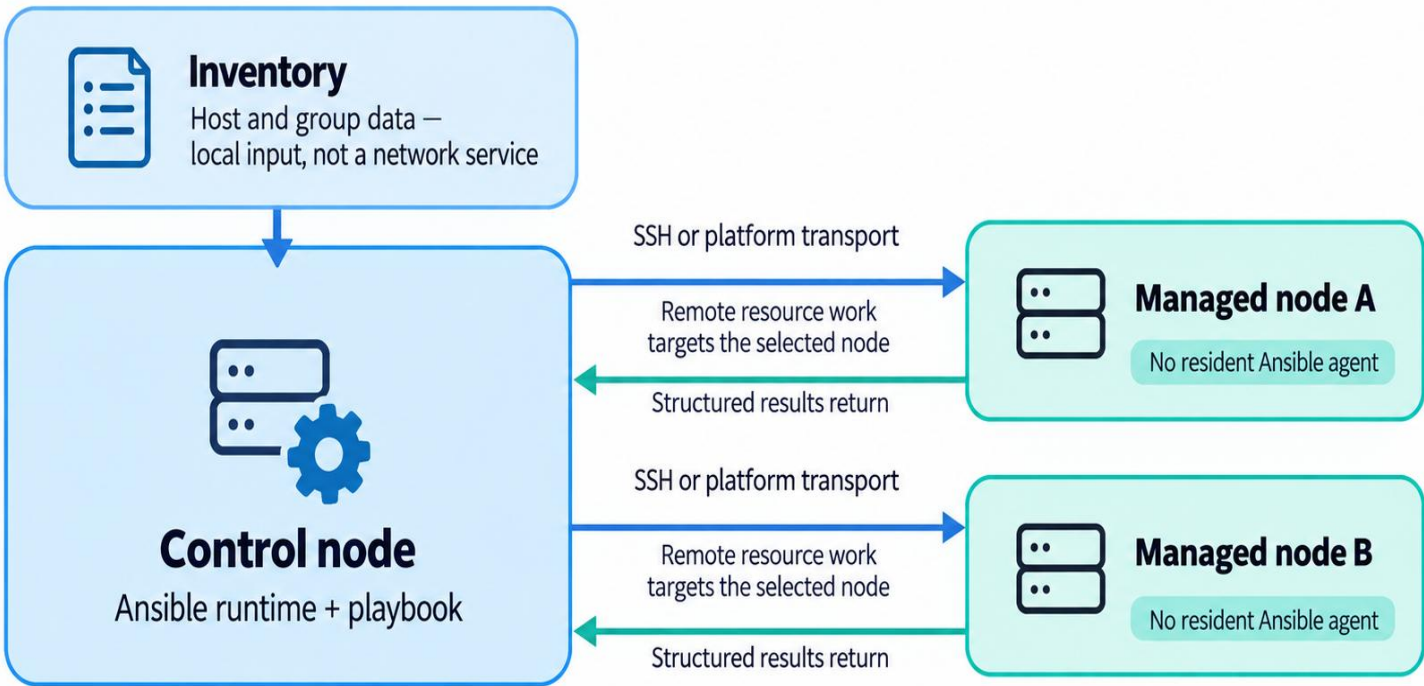
Ansible does not make an unsafe idea safe merely because it is automated. It also does not automatically provide a complete source of truth for every system, and it does not make arbitrary multi-system work transactional. A **transaction** is a set of changes that either all succeed or are all reversed. Ansible can validate, stop, retry, and perform defined recovery steps, but the engineer must design those behaviors.

The three parts of a normal Ansible environment

Most beginner Ansible environments have three main parts.

Chapter 1: Ansible in One Mental Model

1-2 Control Node, Inventory, and Managed Nodes



002

Figure 1-2: Control node, inventory, and managed-node topology

Part	Meaning	Practical question it answers
Control node	The system on which Ansible is installed and from which commands are run.	Where does the automation start?
Inventory	Data that identifies managed nodes, organizes them into groups, and can describe how Ansible connects to them.	Which systems are available, and how are they addressed?
Managed node	A remote system or device that Ansible controls.	Where does the work happen?

For the main examples in this guide, the control node is a UNIX-like system and the managed nodes are Linux systems reached over SSH. This is one common environment, not a rule for every Ansible use case.

An everyday analogy helps separate these parts. Imagine a site supervisor responsible for several buildings:

- The supervisor’s office is the **control node**.

- The roster of buildings and teams is the **inventory**.
- The buildings are the **managed nodes**.
- A work plan chooses certain buildings and lists the work to perform.
- Specialized tools perform each individual job.

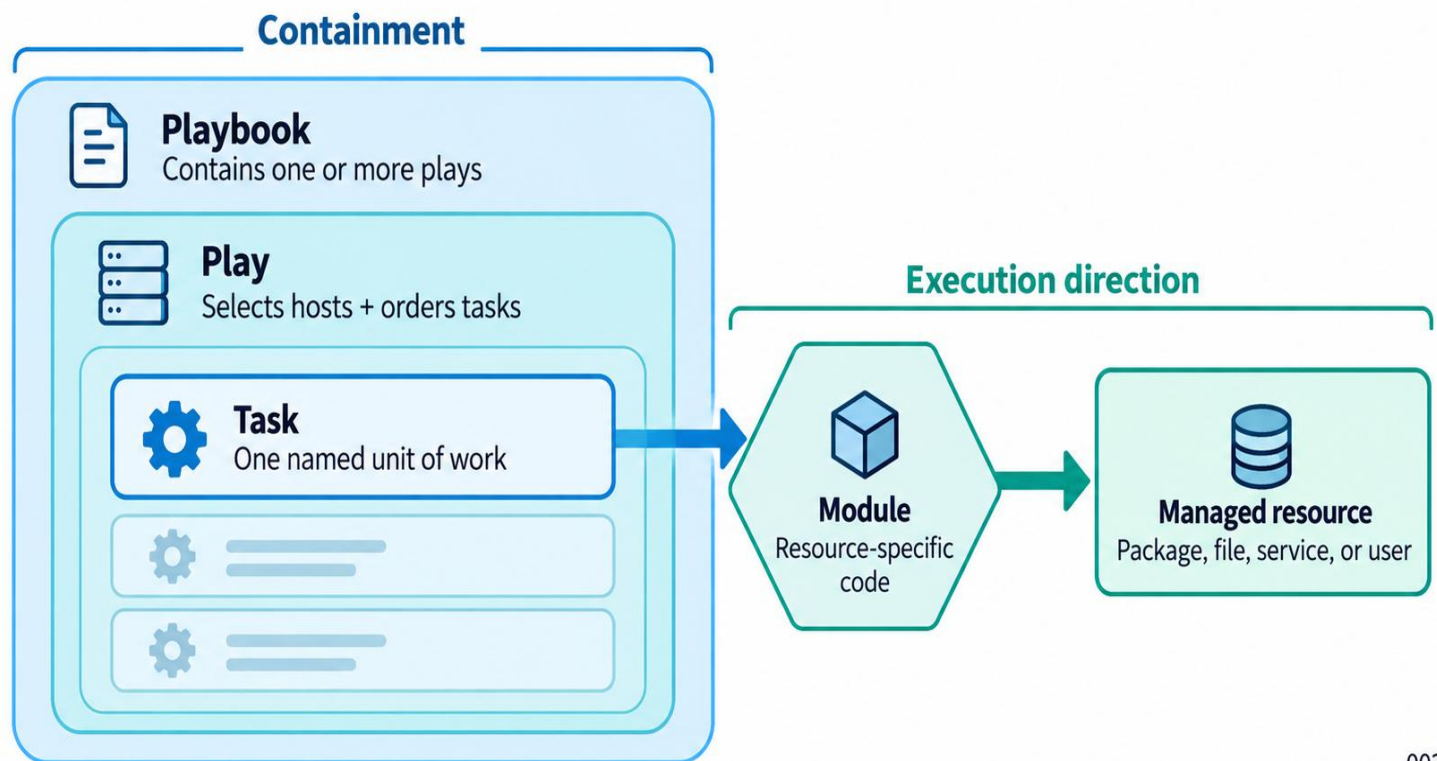
The supervisor does not move permanently into every building. In the same way, standard Ansible operation does not require a permanently running Ansible agent on every managed node.

The vocabulary of automation content

Ansible uses several related terms. Understanding their boundaries prevents many early mistakes.

Chapter 1: Ansible in One Mental Model

1-3 Playbooks, Plays, Tasks, and Modules



003

Figure 1-3: Containment and execution relationships among playbooks, plays, tasks, and modules

Term	Meaning
Playbook	A YAML document containing one or more plays. It records reusable automation.
Play	A mapping between a selected set of hosts and an ordered set of tasks.
Task	One named unit of work, normally calling one module.
Module	Code that performs a specific action, such as managing a package, file, service, or user.
Plugin	Code that extends how Ansible itself behaves, such as connecting to hosts, looking up data, or formatting output.
Role	A conventional directory structure for packaging related automation content for reuse.
Collection	A distributable namespace that can contain modules, plugins, roles, and related content.

A task is not the same as a shell command. A task usually asks a module to manage a resource. For example, a package task can express that `nginx` must be present. The package module inspects the target and installs the package only when necessary.

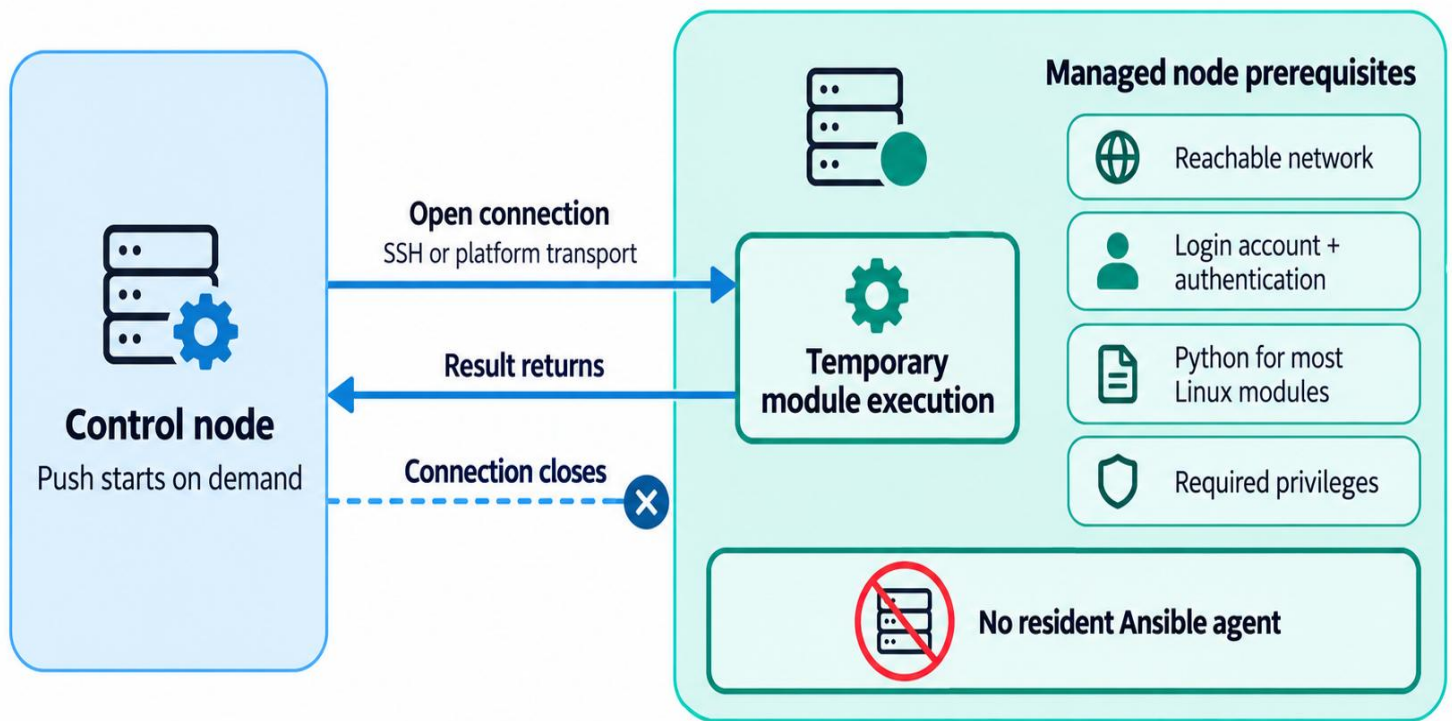
A module name such as `ansible.builtin.copy` is a **fully qualified collection name**, often abbreviated as FQCN. It contains the collection namespace (`ansible`), the collection name (`builtin`), and the module name (`copy`). Using FQCNs makes the source of a module explicit and avoids ambiguity when collections provide similarly named content.

Agentless does not mean prerequisite-free

Ansible is described as **agentless** because a standard managed node does not need a persistent Ansible service waiting for instructions. In the Linux-over-SSH model, the control node connects when work begins, transfers or invokes the required module logic, receives the result, and closes the connection when appropriate.

Chapter 1: Ansible in One Mental Model

1-4 Agentless Push and Prerequisites



004

Figure 1-4: Agentless push execution and managed-node prerequisites

That model still has prerequisites. A typical Linux managed node needs:

- Network reachability from the control node.
- An SSH service and a permitted login account.
- Authentication, normally through an SSH key or another approved method.
- A usable Python interpreter for most standard modules.
- Sufficient operating-system privileges for the requested change.

Some platforms and modules use different transports or have different runtime requirements. The important principle is that **agentless describes the absence of a permanently installed Ansible agent; it does not mean that any untouched device can be managed without preparation.**

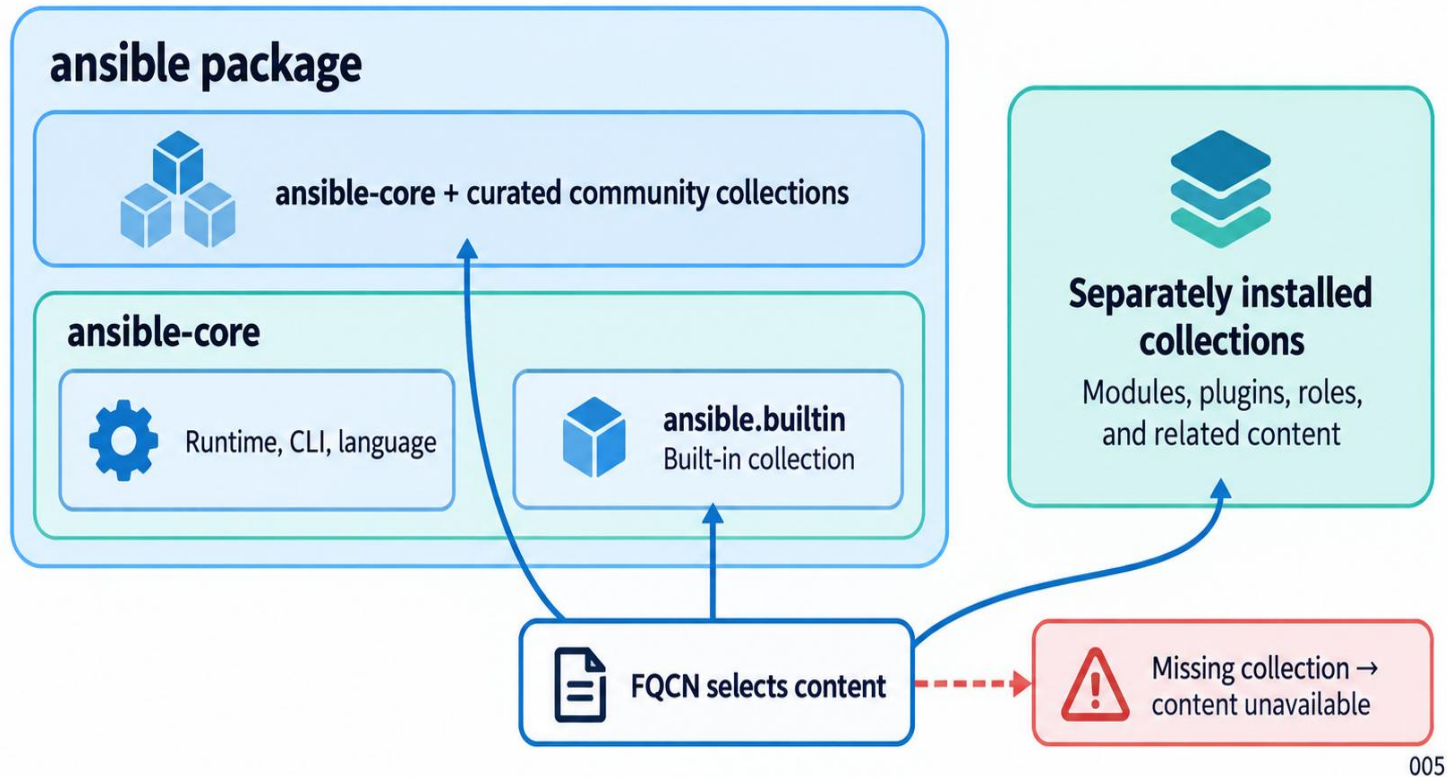
Ansible normally uses a **push-based workflow**: the control node initiates the run and pushes work toward the selected managed nodes. This is different from systems in which each managed node periodically pulls its own configuration.

ansible-core, the ansible package, and collections

The Ansible community distributes two commonly encountered Python packages:

Chapter 1: Ansible in One Mental Model

1-5 ansible-core, ansible, and Collections



005

Figure 1-5: Relationship among ansible-core, the ansible package, built-in content, and separately installed collections

- **ansible-core** provides the automation language, command-line tools, runtime, and the built-in collection.
- **ansible** includes ansible-core plus a broader community-curated set of collections.

Collections can also be installed separately. This separation matters when reading a project: a playbook may be valid YAML and valid Ansible syntax but still fail because the collection containing a required module is not installed.

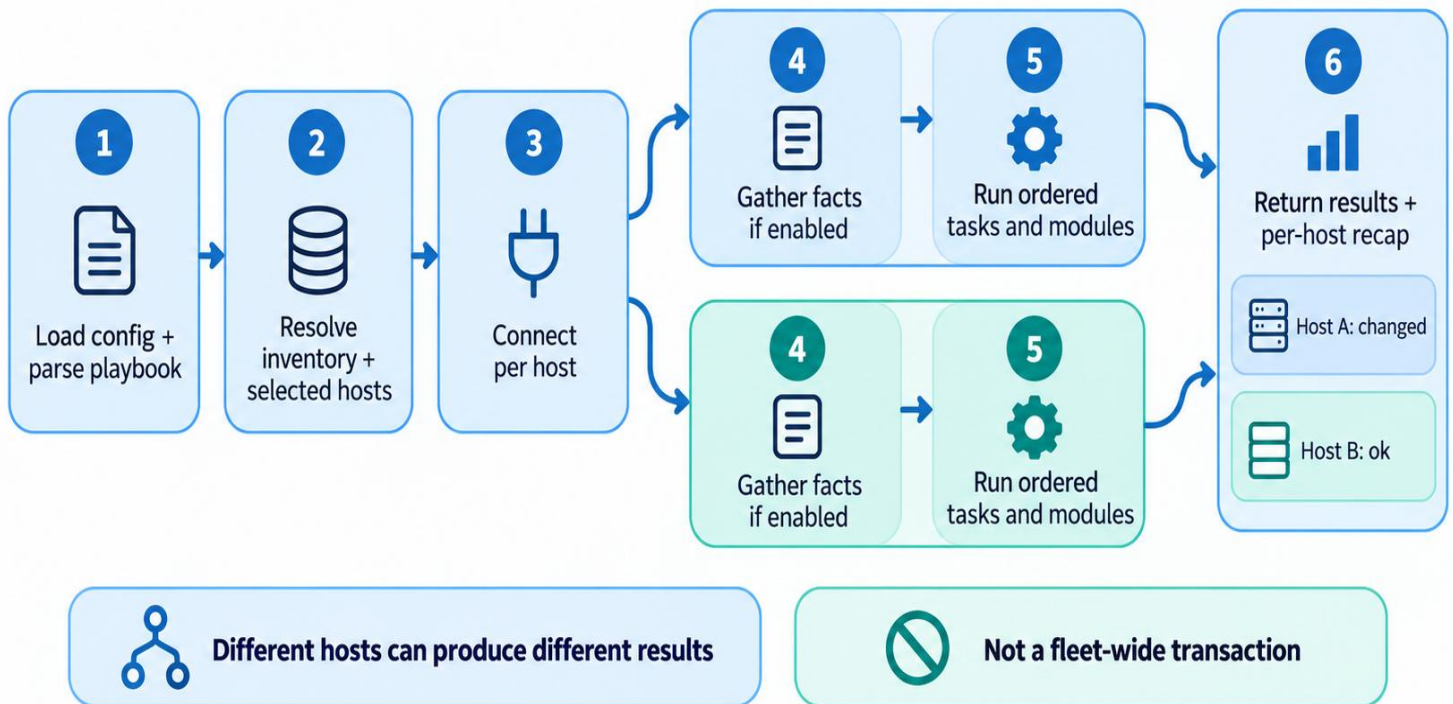
For a beginner lab, the full **ansible** package is convenient because it includes more content. A team project may choose **ansible-core** and install only explicitly pinned collections. The package choice does not change the basic playbook model.

What happens during a playbook run

With Ansible installed, inventory available, and a valid `configure_web.yml`, the command `ansible-playbook configure_web.yml` starts the playbook. The expected result follows this execution sequence:

Chapter 1: Ansible in One Mental Model

1-6 What Happens During a Playbook Run



006

Figure 1-6: Internal sequence of an ansible-playbook run

1. The control node loads Ansible configuration and parses the playbook.
2. Ansible loads inventory and resolves the hosts selected by each play's `hosts` pattern.
3. Ansible prepares connections to the selected managed nodes.
4. If fact gathering is enabled, Ansible collects system information from each host.
5. Ansible processes tasks in playbook order for the active hosts.
6. For each task, the selected module examines the relevant current state and performs work if needed.
7. The module returns structured result data to the control node.
8. Ansible displays task results and a recap for each host.

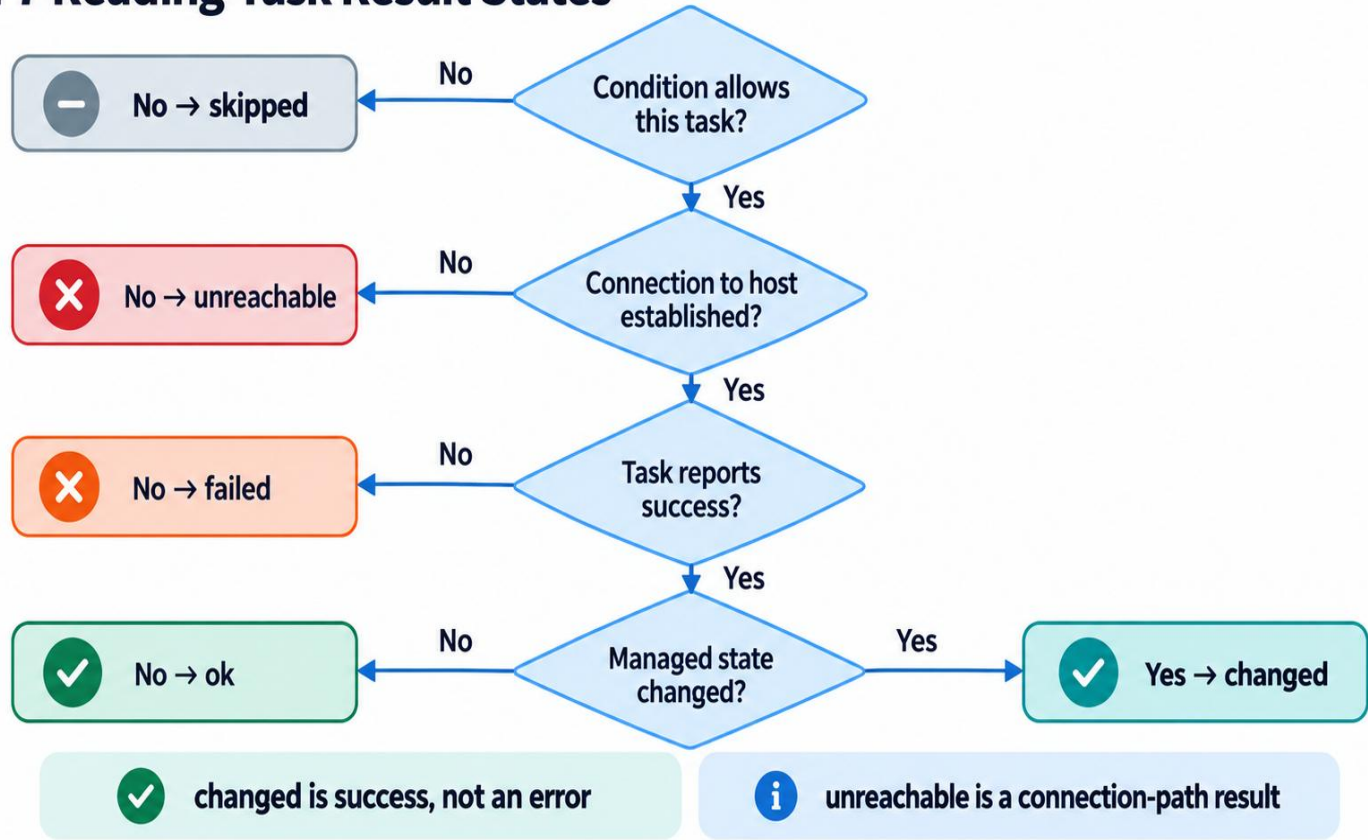
The exact connection and execution details vary by platform and plugin, but this sequence is the right starting model.

Suppose a task says that package `nginx` must be present. On one host, the package is absent, so the module installs it and reports a change. On another host, the same package is already installed, so the module reports success without changing the system. The task is identical; the result depends on the current state of each host.

Ansible output commonly uses the following result states:

Chapter 1: Ansible in One Mental Model

1-7 Reading Task Result States



007

Figure 1-7: Decision path for ok, changed, failed, unreachable, and skipped

State	Meaning
ok	The task succeeded and did not report a change.
changed	The task succeeded and reported that it changed the managed node.
failed	Ansible reached the host and ran the task, but the task did not meet its success conditions.
unreachable	Ansible could not establish the required connection or begin normal module execution.
skipped	Ansible deliberately did not run the task for that host, commonly because a condition was false.

The difference between `failed` and `unreachable` is operationally important. A failed package task suggests that Ansible connected but package management failed. An unreachable result suggests a connection, authentication, name-resolution, or transport problem before the task could run normally.

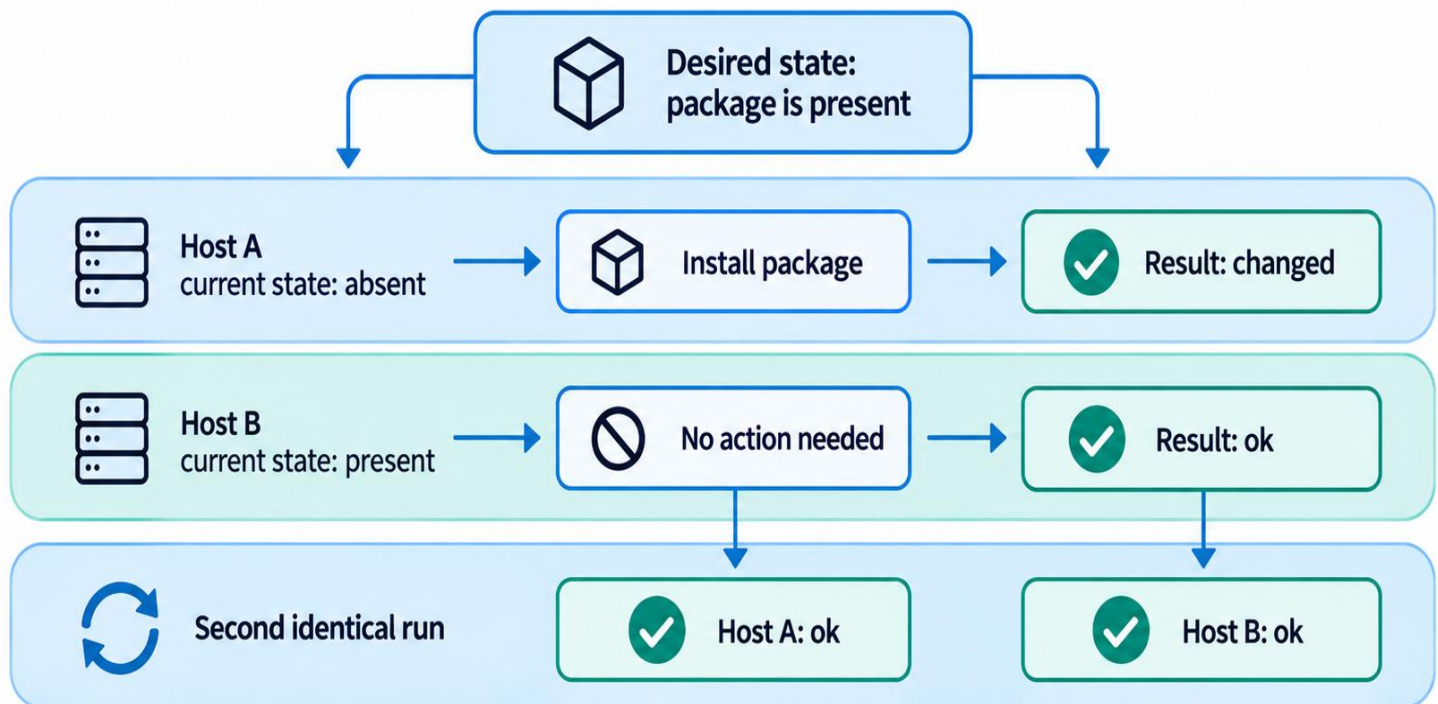
A successful task is not automatically a well-designed task. A shell command can exit with code zero every time while rewriting the same file on every run. That behavior may be successful in a narrow sense, but it is not necessarily safe or repeatable.

Desired state is the central idea

The strongest Ansible tasks describe **desired state**, meaning the condition that should be true after the task finishes. Examples include:

Chapter 1: Ansible in One Mental Model

1-8 Desired State and Idempotent Repeats



Idempotent repeat: intended state is preserved

008

Figure 1-8: Desired-state convergence and idempotent repeat execution

- A package is present.
- A directory exists with specified ownership and permissions.
- A service is enabled and running.
- A configuration file has approved content.

This is different from describing only an action such as “run the package installer.” State-oriented modules can usually inspect the system and decide whether action is required.

The thermostat analogy is useful. A thermostat is configured with a desired temperature. It does not turn on the heater forever; it measures the current temperature and acts only when needed. In the same way, a well-chosen Ansible module compares current state with desired state and changes the system only when necessary.

This property leads to **idempotency**, which means that repeating the same operation with the same inputs produces the same intended state without unnecessary additional change. Idempotency is not magic applied to every task. It depends on the module, its arguments, and the behavior of any external command or system it uses.

The mental model to retain

Inventory answers “where,” a **play** answers “which hosts and which ordered work,” a **task** answers “what unit of work,” and a **module** answers “how this resource is inspected and managed.”

When a run behaves unexpectedly, return to that model:

- Did Ansible load the intended inventory?
- Did the host pattern select the intended systems?
- Did the task call the appropriate module?
- Did the module receive values of the expected type?
- Did the connection account have the required access?
- Did the result report `unreachable`, `failed`, `changed`, or `ok`?

Those questions locate most beginner problems more effectively than immediately adding more verbosity or replacing a module with a shell command.

Chapter Summary

Ansible starts on a control node, resolves hosts through inventory, and executes ordered tasks through modules on managed nodes. Reliable automation depends on a correct execution model, meaningful result states, and state-oriented tasks that can be repeated safely.

Review Questions

1. What role does each of the control node, inventory, and managed node play?
2. How is a task different from the module it invokes?
3. Why are `failed` and `unreachable` different diagnostic states?
4. What must be true for a repeated task to be idempotent?

Chapter 2 — Create a Safe Practice Environment

What You'll Be Able to Do

- Create an isolated Ansible practice environment and confirm which runtime and configuration are active.
- Establish and verify SSH access, authentication, host-key trust, and managed-node prerequisites.
- Separate configuration from credentials and diagnose environment problems in a useful order.

A reliable learning environment isolates Ansible from the system Python, verifies ordinary SSH before Ansible is introduced, and keeps configuration local to a disposable practice project. This prevents environment problems from being mistaken for playbook problems.

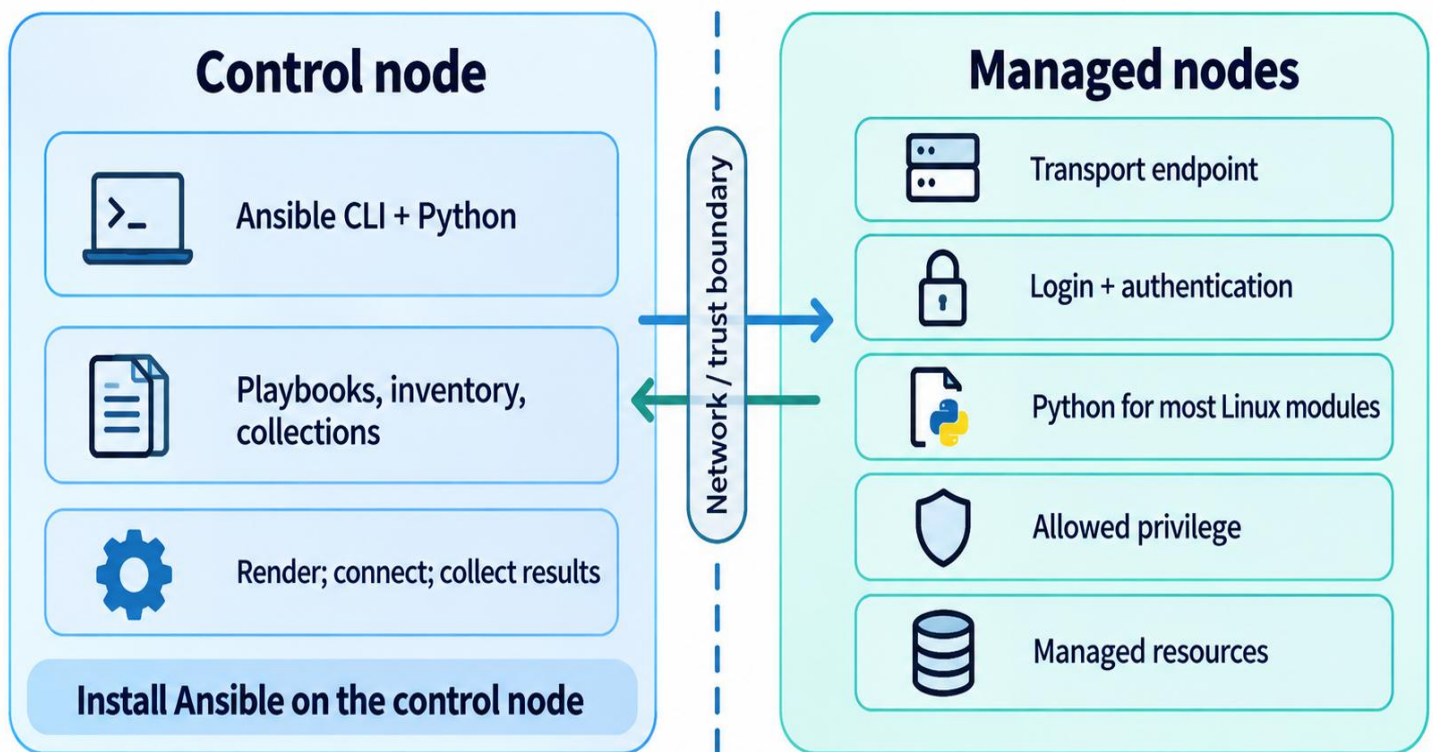
The main example in this chapter uses one UNIX-like control node and two disposable Linux managed nodes. Virtual machines, short-lived cloud instances, or dedicated lab servers are suitable. Containers can also be useful, but a minimal container may not provide SSH, Python, or a normal service manager, so it can add distractions for a first lab.

Control-node and managed-node responsibilities

The **control node** runs commands such as `ansible`, `ansible-inventory`, and `ansible-playbook`. A current Python installation is normally required on the control node because Ansible itself is a Python application.

Chapter 2: Create a Safe Practice Environment

2-1 Control-Node and Managed-Node Responsibilities



009

Figure 2-1: Responsibility split between the control node and managed nodes

A standard control node can be a Linux system, macOS, a BSD system, or Windows through a Windows Subsystem for Linux distribution. A Windows installation without WSL is not treated as a normal native Ansible control node in this learning path.

For the Linux-over-SSH examples, each **managed node** should provide:

- An IP address or resolvable DNS name reachable from the control node.

- A running SSH service.
- A login account intended for automation practice.
- A Python interpreter that supports the Ansible version being used.
- Permission to perform the lab changes, with privilege escalation configured only when needed.

Do not begin by pointing a new practice project at production systems. Disposable targets make it safe to inspect changes, make mistakes, rebuild, and repeat the same run.

Choose the package deliberately

Install the full `ansible` package when convenience and a broad starting set of collections are useful. Install `ansible-core` when the project intends to manage collection dependencies separately and keep the runtime minimal.

For a first lab, the full package is a practical choice. The examples still use modules from `ansible.builtin`, so they also work with an appropriate `ansible-core` installation.

Install with `pipx`

`pipx` installs Python command-line applications into isolated environments while exposing their commands on the user's `PATH`. This avoids mixing Ansible packages with the operating system's Python packages.

Prerequisites: Python and `pipx` are installed for the current user, and the `pipx` application directory is on `PATH`.

Purpose: Install the full Ansible community package in an isolated environment.

```
pipx install --include-deps ansible
```

Expected result: `pipx` creates an isolated environment and makes Ansible commands available to the current user. The exact installed version depends on the package version selected by the environment at installation time.

To install only the smaller runtime, use the following alternative.

Prerequisites: Python and `pipx` are installed, and no conflicting Ansible command is earlier on `PATH`.

Purpose: Install only the `ansible-core` runtime in an isolated environment.

```
pipx install ansible-core
```

Expected result: `pipx` exposes the core Ansible command-line tools without installing the broader set of collections included in the full `ansible` package.

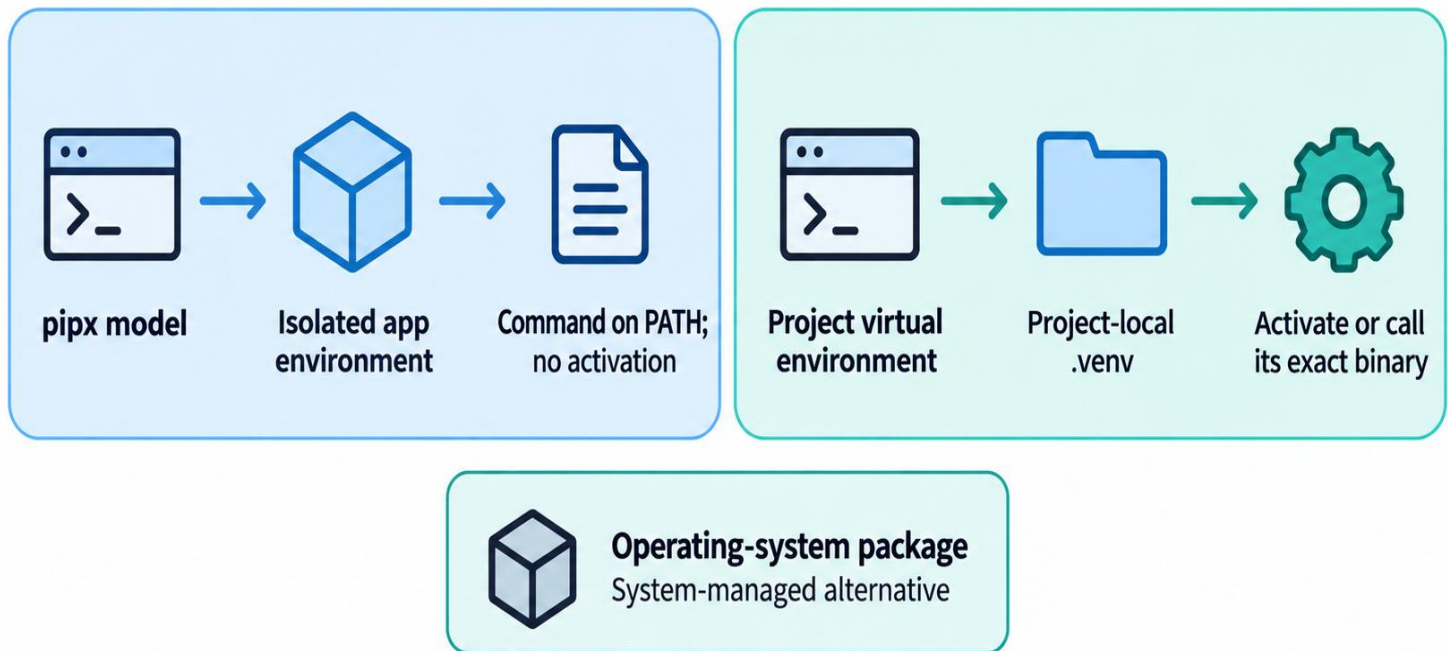
Do not install both packages into unrelated environments and then assume the shell will use the intended one. Confirm the command path and version after installation.

Use a virtual environment as an alternative

A Python **virtual environment** is an isolated directory containing its own Python package installation. It is suitable when a project deliberately activates a local environment before using Ansible.

Chapter 2: Create a Safe Practice Environment

2-4 Two Python Isolation Models



Install and run through the same environment

010

Figure 2-2: pipx and project virtual environments as two isolation models

Prerequisites: Python 3 includes the venv module, and the current directory is the intended practice project directory.

Purpose: Create an isolated environment, activate it, and install Ansible inside it.

```
python3 -m venv .venv
. .venv/bin/activate
python -m pip install --upgrade pip
python -m pip install ansible
```

Expected result: The shell prompt normally indicates that `.venv` is active, and Ansible commands resolve to executables inside `.venv`. The `.venv` directory should not be committed to source control.

Operating-system package managers may also provide Ansible. That can be appropriate when an organization standardizes on vendor packages, but the available version and update policy depend on the operating system. Treat that as an environment-specific choice rather than a universal installation method.

Confirm the installation

Prerequisites: Ansible was installed and its command directory is on PATH. If a virtual environment is used, it is active.

Purpose: Display the Ansible runtime version and the Python and configuration paths currently in use.

```
ansible --version
```

Expected result: The output identifies the installed `ansible-core` version, the Python interpreter, module and collection paths, and the configuration file if one is active. Even when the full `ansible` package is installed, this command reports the associated `ansible-core` runtime version.

The paths are as important as the version number. If the output points to an unexpected Python interpreter or configuration file, correct that before troubleshooting playbooks.

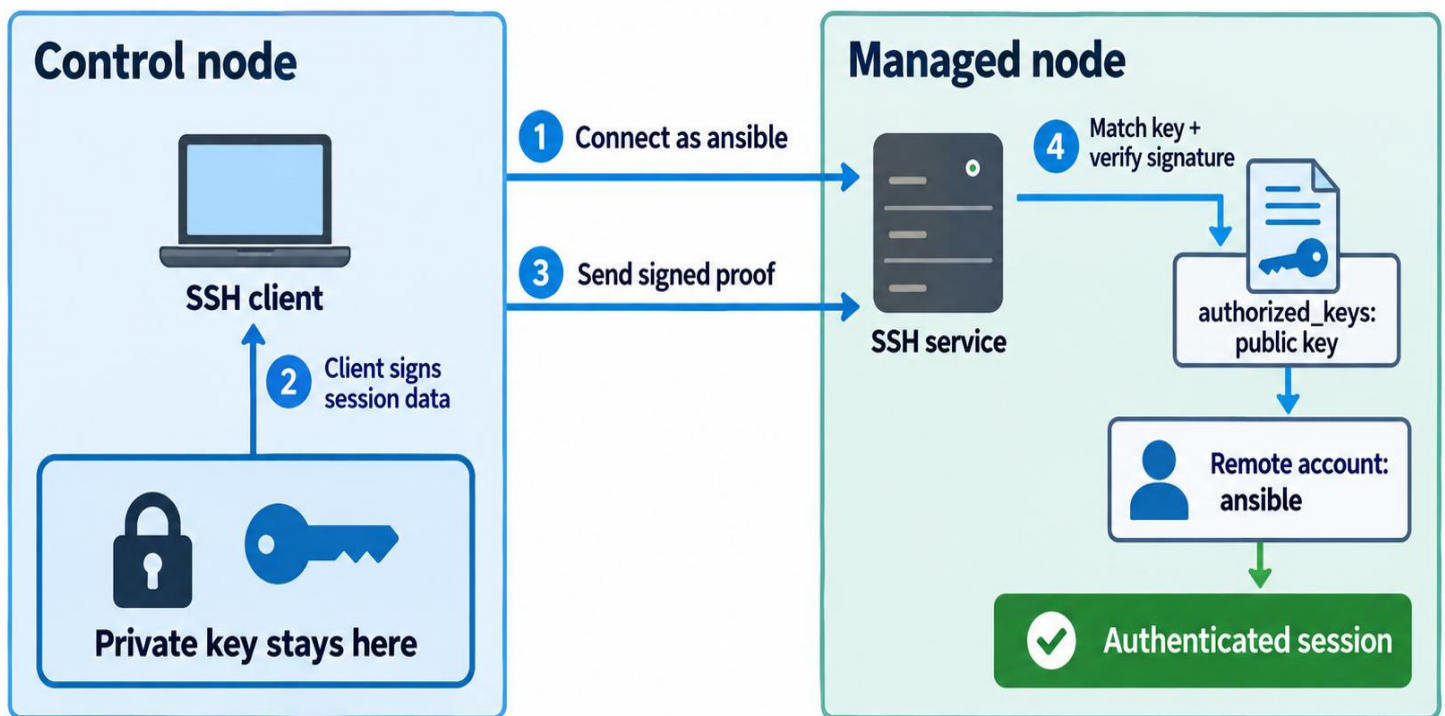
Establish SSH independently of Ansible

Ansible should not be the first tool used to test a new SSH path. Confirm the same user, key, hostname, and network route with the ordinary `ssh` client.

A typical lab uses a dedicated SSH key. The following is one example for OpenSSH environments.

Chapter 2: Create a Safe Practice Environment

2-6 SSH User Authentication Path



Never copy the private key to the managed node

011

Figure 2-3: SSH connection path from the control node to the lab account

Prerequisites: The OpenSSH client is installed, and `~/.ssh/ansible_lab` does not already contain a key that must be preserved.

Purpose: Create an Ed25519 key pair for the lab. A passphrase is recommended unless an approved credential system supplies equivalent protection.

```
ssh-keygen -t ed25519 -f ~/.ssh/ansible_lab
```

Expected result: The command creates a private key at `~/.ssh/ansible_lab` and a public key at `~/.ssh/ansible_lab.pub`. The private key must not be copied to managed nodes or committed to a repository.

Install the public key in the lab account's `authorized_keys` file using the method approved for the environment. Tools such as `ssh-copy-id` are convenient on some UNIX-like systems, but account provisioning methods differ.

Then test an ordinary connection. The address `192.0.2.11` is a documentation example and must be replaced by the actual lab address.

Prerequisites: The public key is authorized for user `ansible` on the managed node, and the node is reachable.

Purpose: Verify authentication and inspect the server's host key before Ansible is involved.

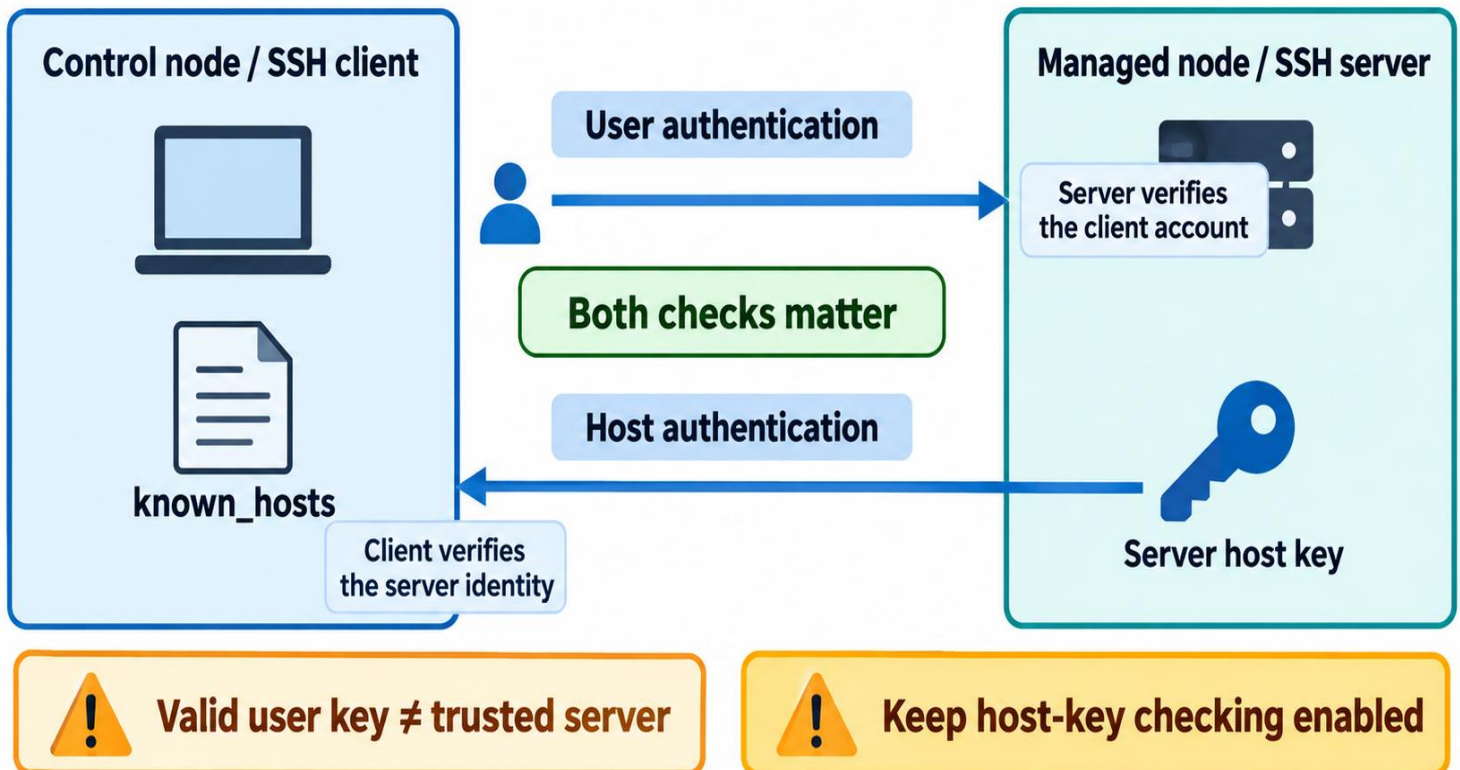
```
ssh -i ~/.ssh/ansible_lab ansible@192.0.2.11
```

Expected result: After the host key is verified and accepted according to local policy, the command opens a remote shell without requiring the account password. Exit the shell after confirming the remote identity with a command such as `id`.

Host-key verification checks that the SSH server presents the expected cryptographic identity. Disabling it globally may make a lab feel easier, but it teaches an unsafe default and can hide redirection to the wrong system. Verify fingerprints through a trusted source and keep normal `known_hosts` behavior enabled.

Chapter 2: Create a Safe Practice Environment

2-6 SSH Host-Key Verification and Trust



012

Figure 2-4: SSH host-key verification and known-host trust

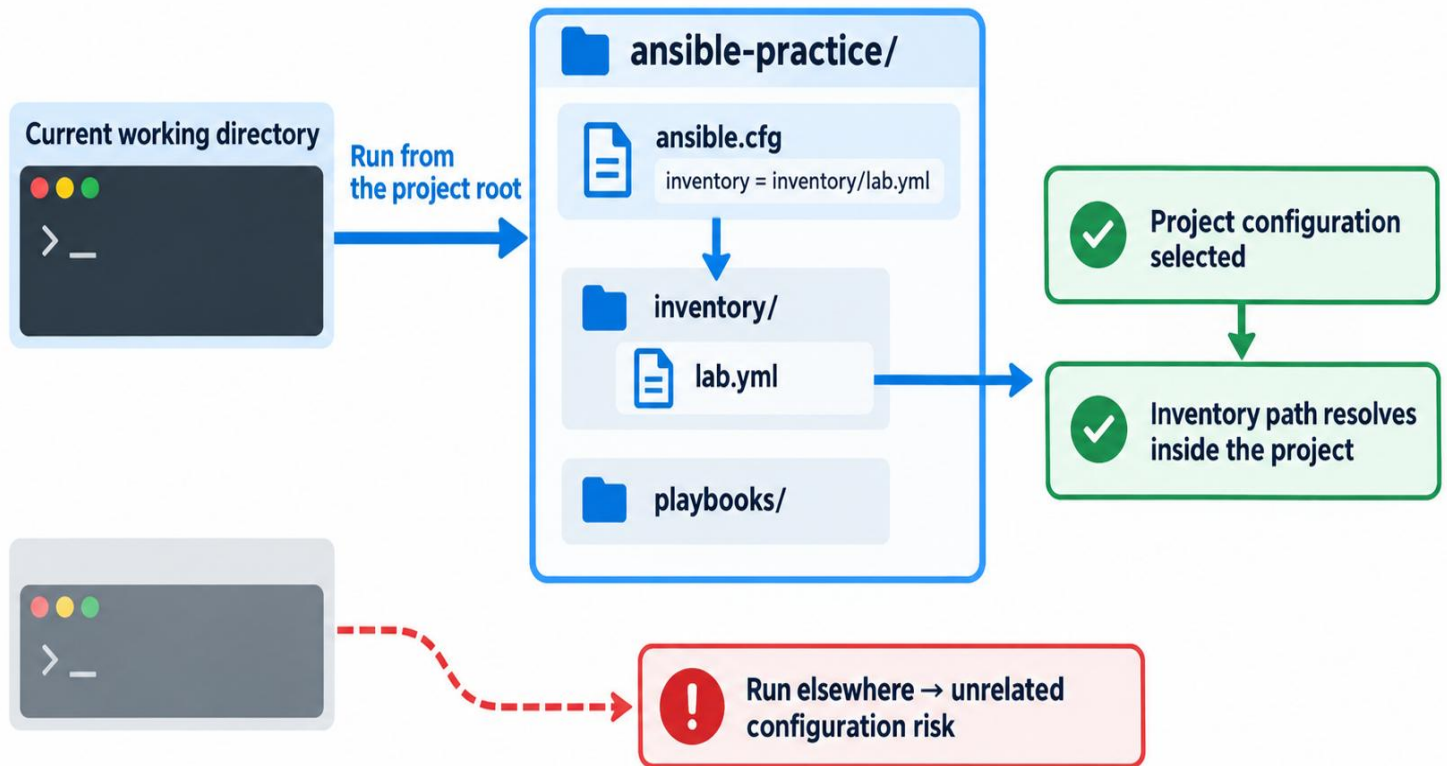
An SSH agent can hold the unlocked private key for the current session. This often avoids placing private-key paths in inventory and avoids repeated passphrase prompts. The exact agent workflow depends on the operating environment.

Keep configuration inside the project

Ansible searches several locations for configuration. A project-local `ansible.cfg` makes the lab's behavior visible and reduces accidental dependence on a user's unrelated global configuration.

Chapter 2: Create a Safe Practice Environment

2-7 Project-Local Configuration



013

Figure 2-5: Project-local configuration and practice-directory containment

A minimal practice structure can begin as follows:

```
ansible-practice/
├── ansible.cfg
├── inventory/
│   └── lab.yml
└── playbooks/
```

The project-local configuration can contain only the inventory location:

```
[defaults]
inventory = inventory/lab.yml
```

A minimal inventory for one host can be written in YAML:

```
all:
  hosts:
    lab-web-01:
      ansible_host: 192.0.2.11
      ansible_user: ansible
```

The name `lab-web-01` is an inventory alias. The `ansible_host` value is the actual address used for the connection. The `ansible_user` value identifies the SSH login account. Inventory structure is covered in detail later.

Run Ansible commands from the project directory so the local configuration is selected.

Prerequisites: The directory contains the preceding `ansible.cfg` and inventory, and Ansible is installed.

Purpose: Confirm which nondefault settings Ansible has loaded.

```
ansible-config dump --only-changed
```

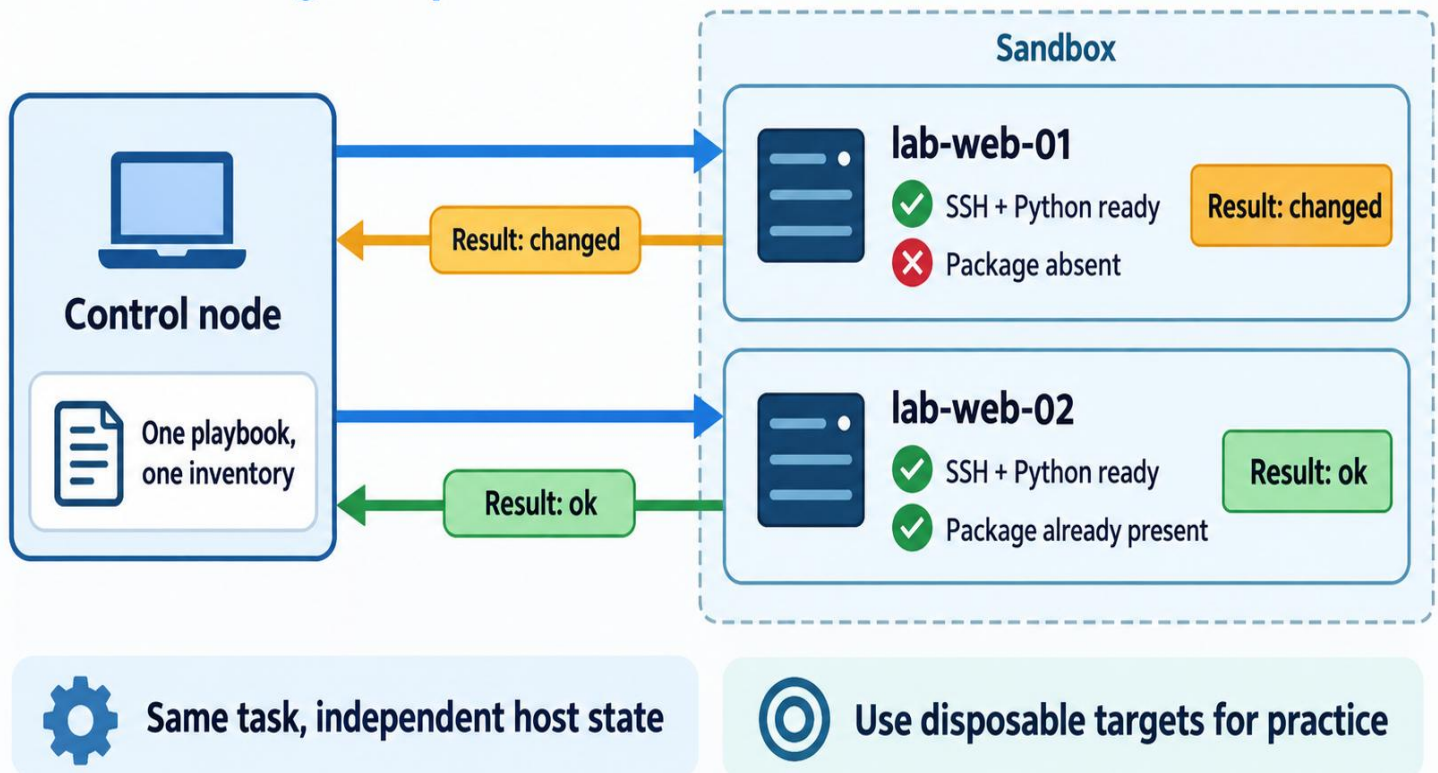
Expected result: The inventory setting points to the project inventory path. If unrelated settings appear, inspect the active configuration path shown by `ansible --version` and the environment variables in the shell.

Add two disposable targets

Two managed nodes are enough to reveal behavior that a single-host lab hides. For example, one node may succeed while the other is unreachable, or a package may already be installed on only one host.

Chapter 2: Create a Safe Practice Environment

2-8 A Two-Target Disposable Lab



014

Figure 2-6: Two-target disposable lab topology

Use addresses and hostnames from the actual lab. A simple two-host inventory is:

```
all:
  hosts:
    lab-web-01:
      ansible_host: 192.0.2.11
      ansible_user: ansible
    lab-web-02:
      ansible_host: 192.0.2.12
      ansible_user: ansible
```

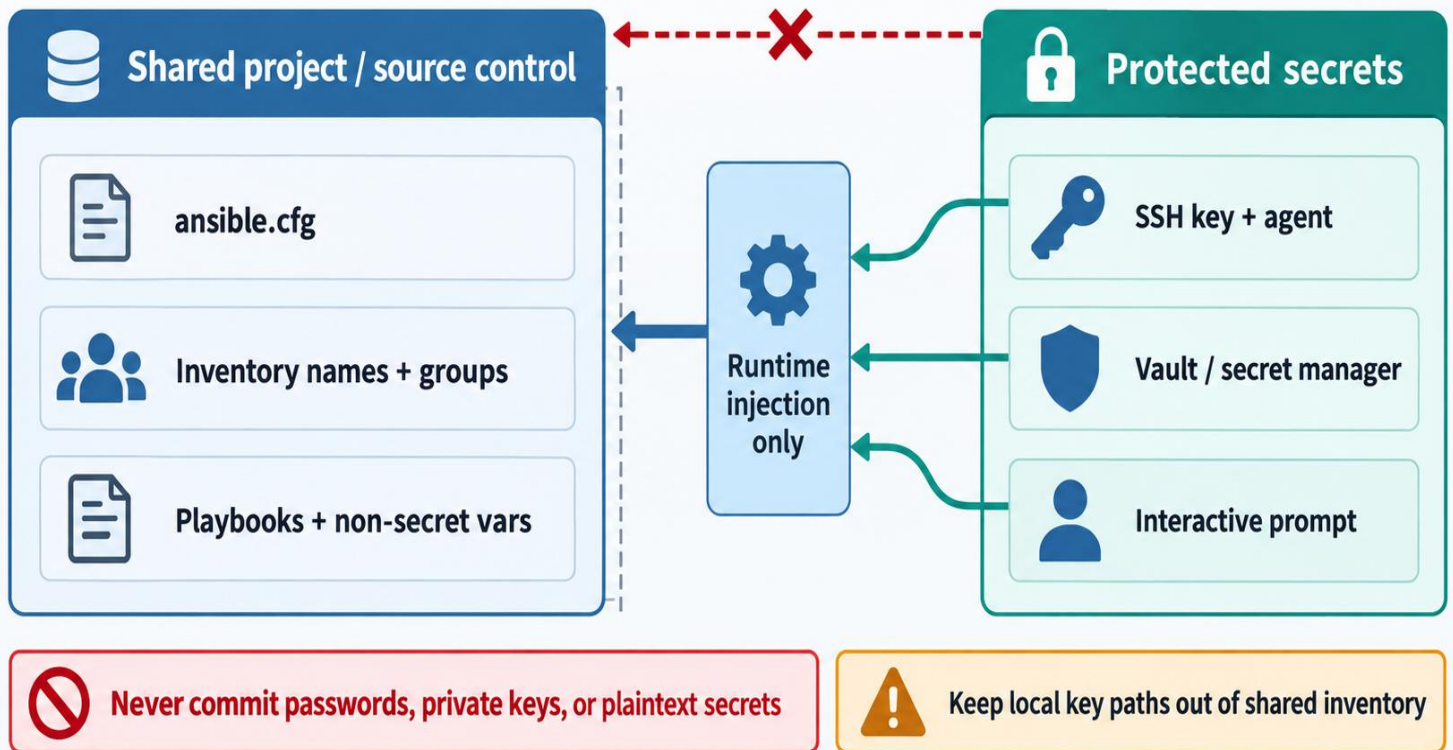
Before running Ansible, test SSH to both hosts with the same account. Also confirm that Python is available, for example with `python3 --version` in the remote shell. If the environment uses a nonstandard interpreter path, inventory can identify it later with `ansible_python_interpreter`.

Keep credentials out of project configuration

Do not store account passwords, private keys, or secret values directly in `ansible.cfg`, inventory committed to source control, or ordinary playbooks. A configuration file is not a credential store.

Chapter 2: Create a Safe Practice Environment

2-9 Keep Credentials Outside Project Configuration



015

Figure 2-7: Credential and configuration storage boundary

The same principle applies to convenience settings. Disabling host-key verification, embedding passwords, or connecting as a broad administrative account may remove an immediate obstacle, but it weakens the lab's value because the resulting habits do not transfer safely to team work.

Work with version awareness

Ansible develops over time. Python support ranges, deprecated syntax, module behavior, and collection content can change between releases. Avoid building the learning environment around an unrecorded global installation.

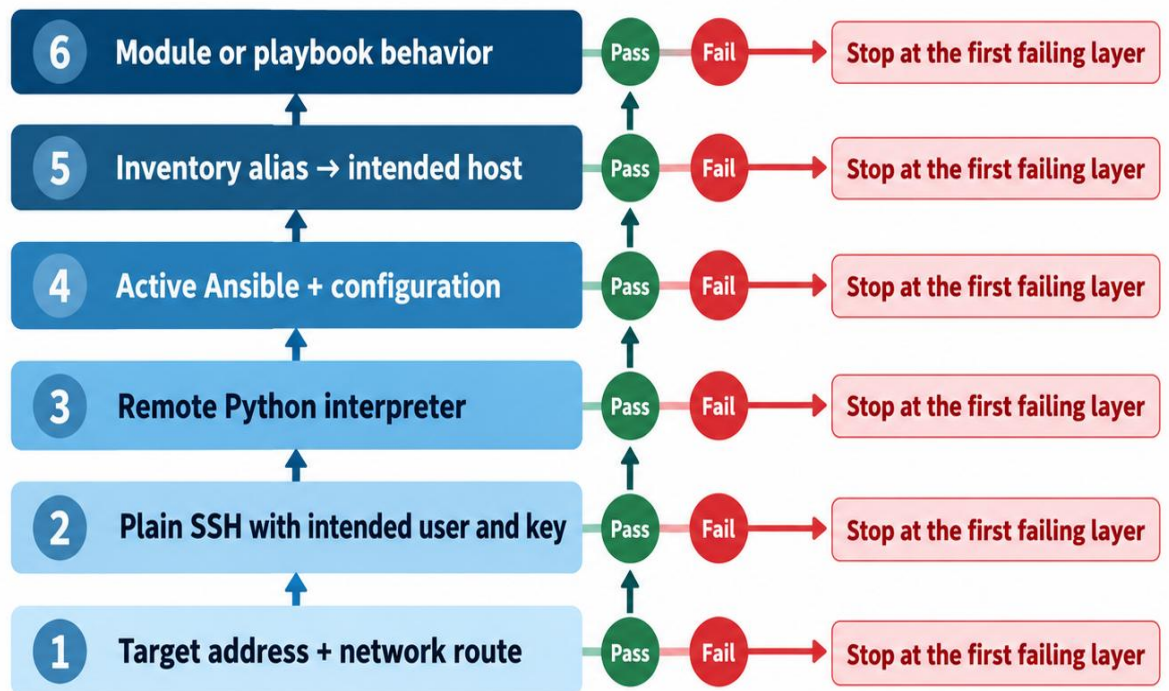
At minimum, retain the output of `ansible --version` with lab notes and use the same isolated environment while working through related examples. When joining a team, use the project's documented and pinned runtime rather than silently upgrading it on one workstation.

Diagnose the environment in the right order

When the first Ansible command fails, check the layers in this order:

Chapter 2: Create a Safe Practice Environment

2-11 Diagnose a New Environment in Order



Fix the lowest failing layer first

016

Figure 2-8: Correct diagnostic order for a new environment

1. Confirm the target address and network route.
2. Confirm ordinary SSH with the intended account and key.
3. Confirm the remote Python interpreter.
4. Confirm the active Ansible installation and configuration paths.
5. Confirm the inventory alias resolves to the intended host.
6. Only then investigate module or playbook behavior.

This order prevents a connection problem from turning into unnecessary playbook edits.

Chapter Summary

A useful practice environment isolates the control-side runtime, proves ordinary SSH first, and keeps project configuration separate from credentials. Troubleshooting should move from network and SSH prerequisites toward Ansible configuration and module behavior.

Review Questions

1. Why should ordinary SSH be tested before Ansible?
2. What information in `ansible --version` helps identify the active environment?
3. Why is project configuration an unsuitable place for passwords and private keys?