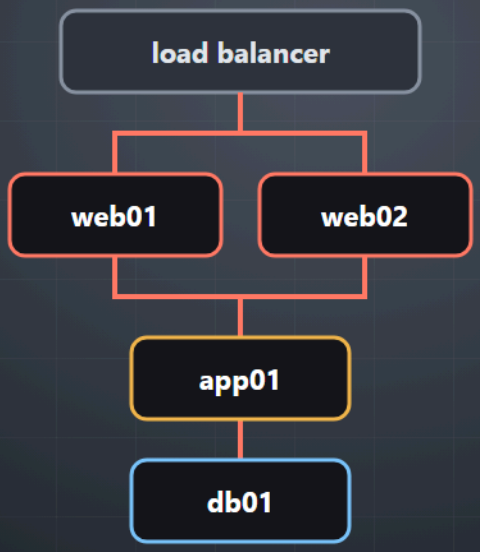
web.yml

```yaml
---
- name: Configure web tier
  hosts: web
  serial: 1
  become: true
  roles:
    - nginx
  post_tasks:
    - name: Check /lb-health
```




```
$ ansible-playbook site.yml
TASK [nginx : Render the Brightloom site config] ****
changed: [stg-web01]
ok: [stg-web02]
PLAY RECAP *********************************************
stg-web01 : ok=12 changed=1 unreachable=0 failed=0
```

A COMPLETE, PRACTICAL GUIDE

# Ansible Automation

## Configure servers, deploy applications, and manage infrastructure the repeatable way

Inventories · Playbooks · Jinja2 · Roles · Collections · Vault
Rolling Deploys · Molecule · CI/CD · Execution Environments

YAML · SSH · NGINX · POSTGRESQL · LINUX

THENEXBOOKS

# Ansible Automation

*Configure Servers, Deploy Applications, and Manage Infrastructure the Repeatable Way*
*From your first ad hoc command to a tested, multi-environment platform — built step by step*

First Edition

THENEXBOOKS

**Ansible Automation**
Configure Servers, Deploy Applications, and Manage Infrastructure the Repeatable Way

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples, playbooks, roles, and templates may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Ansible, its collections, and the operating systems and services it manages change over time; module options, default settings, command output, and package versions can differ between releases. Always consult the official documentation for the versions you use, test changes in a non-production environment first, and keep backups before running automation that changes servers or data.

Ansible and Red Hat are trademarks or registered trademarks of Red Hat, Inc. in the United States and other countries. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. Ubuntu is a registered trademark of Canonical Ltd. NGINX is a trademark of F5, Inc. PostgreSQL is a registered trademark of the PostgreSQL Community Association. Python is a registered trademark of the Python Software Foundation. Microsoft, Windows, and Visual Studio Code are trademarks of the Microsoft group of companies. macOS is a trademark of Apple Inc. GitHub is a trademark of GitHub, Inc. Amazon Web Services and AWS are trademarks of Amazon.com, Inc. or its affiliates. Terraform is a trademark of HashiCorp, Inc. Docker is a trademark of Docker, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by Red Hat, Inc. or any other company mentioned.

Brightloom, the SaaS company used as the running example in this book, is fictional, as are its product Brightloom Invoices, its employees, servers, IP addresses, releases, and all sample data. Domain names use the reserved example.com domain, and IP addresses are private ranges chosen for illustration.

# Preface

Most server problems do not start with a dramatic failure. They start with a small, reasonable change made by hand: a package installed over SSH to fix an urgent bug, a configuration file tweaked on one web server but not the other, a setting copied from a wiki page that was last updated two years ago. Each change makes sense on its own. Together they produce servers that nobody can rebuild with confidence, staging environments that no longer resemble production, and a deployment process that lives in one person's memory.

Ansible exists to replace that kind of work with something repeatable. You describe the state you want — which packages are installed, which files contain what, which services are running, which version of your application is live — in plain, readable YAML files. Ansible connects to your servers over SSH, compares reality with your description, and changes only what differs. Run it once and the servers are configured; run it again and nothing changes, because nothing needs to. The same files configure staging and production, live in version control next to your application, and can be reviewed like any other code.

This book teaches Ansible from the very beginning and takes you all the way to a production-quality automation project. We do not just list modules. We build one real platform step by step, introducing each feature at the moment it solves a problem we actually have, and explaining *why* it works the way it does, so you can design automation for your own systems with confidence.

## Who This Book Is For

- **Developers** who deploy their own applications and want servers and releases they can reproduce instead of hand-tune.
- **System administrators** who manage fleets of Linux servers and want to turn runbooks and shell scripts into reliable, reviewable automation.
- **DevOps and platform engineers** who need a solid, practical grounding in roles, collections, Vault, testing, CI/CD, and multi-environment workflows.
- **IT generalists and students** who know basic Linux and SSH and want a skill that pays off on every server they touch.

## What You Will Learn

- How Ansible works — control node, managed nodes, inventory, modules, and playbooks — and how to set up a control node on Linux, macOS, or Windows with WSL.
- How to describe your servers in inventories, run ad hoc commands, and write idempotent playbooks with variables, facts, templates, conditionals, loops, and handlers.
- How to handle errors deliberately with blocks, assertions, and health checks, and how to organize growing automation with imports, includes, and tags.

- How to build reusable roles for a real three-tier application: nginx web servers, a Python application server, and PostgreSQL — and how to package them with collections.
- How to protect secrets with Ansible Vault, harden users and SSH, manage firewalls and TLS, and keep staging and production separate and safe.
- How to deploy applications with versioned releases, rollbacks, rolling updates, and health checks, and how to run day-2 operations such as patching and backups.
- How to make automation fast, testable, and trustworthy: performance tuning, linting, Molecule, CI/CD pipelines, execution environments, debugging, and custom plugins.

## How This Book Is Organized

**Part I — Foundations** (Chapters 1–5) explains what Ansible is, sets up your control node and lab, and covers inventories, ad hoc commands, and your first playbook. **Part II — Writing Playbooks** (Chapters 6–10) covers variables and facts, Jinja2 templates, conditionals, loops, handlers, error handling, and organizing playbooks. **Part III — Roles and Reuse** (Chapters 11–15) builds the roles for each tier of the running example and covers collections and Ansible Galaxy. **Part IV — Secrets, Security, and Environments** (Chapters 16–20) covers Ansible Vault, users and SSH hardening, firewalls and TLS, managing staging and production, and dynamic inventory. **Part V — Deploying and Operating** (Chapters 21–24) covers application deployment, rolling updates, day-2 operations, and performance. **Part VI — Quality and Production** (Chapters 25–28) covers testing, CI/CD and execution environments, debugging and extending Ansible, and a complete case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with a **Try It Yourself** exercise list and a **Key Takeaways** box for quick review.

## The Running Example: Brightloom

Throughout the book we automate the infrastructure of **Brightloom**, a fictional SaaS company whose product, Brightloom Invoices, helps freelancers and small agencies send invoices and get paid. Its application is a classic three-tier web app: two nginx web servers behind a load balancer, one application server running a Python web application under Gunicorn and systemd, and one PostgreSQL database server — in both a **staging** and a **production** environment. When we meet Brightloom, its servers were configured by hand from a wiki page and the two environments have drifted apart. By the end of the book, a single Git repository, `brightloom-infra`, builds any environment from scratch, deploys and rolls back releases, patches and backs up servers, and is tested automatically on every change. You build the staging environment yourself in a small lab of four virtual machines, so every step can be followed on real servers.

## Versions and Conventions

Examples target **recent versions of ansible-core** together with the `ansible` community package, and managed nodes running **Ubuntu Server 24.04 LTS**; notes point out where Red Hat family systems differ. The Ansible control node must run on **Linux or macOS** — on Windows, use **WSL 2**, as Chapter 2 explains. Ansible changes steadily: module options are added, defaults evolve, and recent releases are stricter about some templating patterns. Command output in this book is shown as *what you should see something like*; your version numbers, timings, and spacing will differ. When something behaves differently on your system, `ansible-doc` and the official documentation for your installed version are the final authority.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause outages, data loss, or security problems.

Code, file paths, and commands appear in `monospace`; long lines are wrapped to fit the page. Commands you type on the control node start with a `$` prompt and run from the project folder `~/brightloom-infra`. Modules are always written with their fully qualified collection name, such as `ansible.builtin.template`, and file captions above each listing show the file's path in the project.

# Contents

Preface

PART I — FOUNDATIONS

1 Why Ansible: Automation the Repeatable Way

2 Setting Up Your Control Node and Lab

3 Inventories: Hosts, Groups, and Environments

4 Ad Hoc Commands and Essential Modules

5 Your First Playbook: Plays, Tasks, and Idempotency

PART II — WRITING PLAYBOOKS

6 Variables, Facts, and Precedence

7 Jinja2 Templates and Filters

8 Conditionals, Loops, and Handlers

9 Error Handling, Blocks, and Task Control

10 Organizing Playbooks: Imports, Includes, and Tags

PART III — ROLES AND REUSE

11 Roles: Structure, Defaults, and Your First Role

12 The Web Tier: Building the nginx Role

13 The Database Tier: Building the PostgreSQL Role

14 The App Tier: Building the Application Role

15 Collections, Ansible Galaxy, and Dependencies

PART IV — SECRETS, SECURITY, AND ENVIRONMENTS

16 Ansible Vault and Secrets Management

17 Users, SSH, and Server Hardening

18 Firewalls, TLS, and Network Security

19 Managing Staging and Production Safely

20 Dynamic Inventory and Cloud Hosts

**PART V — DEPLOYING AND OPERATING**

21 Deploying Applications: Releases, Symlinks, and Rollbacks

22 Rolling Updates and Zero-Downtime Changes

23 Day-2 Operations: Patching, Backups, and Maintenance

24 Performance and Scale: Forks, Strategies, Async, and Caching

**PART VI — QUALITY AND PRODUCTION**

25 Testing Ansible: Linting, Check Mode, and Molecule

26 CI/CD for Ansible: Pipelines, Execution Environments, and Navigator

27 Debugging and Extending Ansible: Troubleshooting, Filters, and Modules

28 Production Case Study: The Complete Brightloom Platform

**APPENDICES**

A Ansible Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

# Foundations

What Ansible is and why repeatable automation beats hand-configured servers, setting up a control node and a four-server lab, describing servers in inventories, running ad hoc commands with essential modules, and writing your first idempotent playbook.

1 · Why Ansible

2 · Control Node and Lab

3 · Inventories

4 · Ad Hoc Commands and Modules

5 · Your First Playbook

CHAPTER 1 · PART I — FOUNDATIONS

# Why Ansible: Automation the Repeatable Way

*Servers that are configured by hand slowly drift apart, and nobody can say exactly how they got the way they are. In this chapter we meet Brightloom, see why its hand-built servers became a liability, and learn how Ansible turns server setup into readable, repeatable, reviewable files.*

---

## A Tuesday Night at Brightloom

Brightloom is a small software company with one product, **Brightloom Invoices**, an online invoicing app used by freelancers and small agencies. Customers create clients, send invoices, and get paid. The product is a classic three-tier web application: two nginx web servers sit behind a managed load balancer, one application server runs the Python app under Gunicorn, and one PostgreSQL server stores everything. There is a staging copy of the whole stack and a production copy.

Like many young companies, Brightloom built its servers by hand, by logging in over SSH and following a wiki page called "How to set up a web server" from top to bottom. The page grew, steps were edited, and a few were quietly skipped.

**Maya Torres**, the platform engineer, remembers the night that changed things. Production started returning errors on large invoice uploads. A developer logged in to `prod-web01`, raised the upload limit in the nginx configuration, reloaded nginx, and the problem went away. Two weeks later the hosting provider retired the underlying hardware and the web server was rebuilt from the wiki page. The upload limit was not on the wiki page. The errors came back, and it took most of an evening to remember why.

Around the same time, **Jordan Reyes**, who ships the application releases, noticed that a bug reproduced in production but not in staging. After a long investigation the cause turned out to be a different package version on the two application servers. Nobody had decided that staging and production should differ; they had simply been built months apart, by different people, from different revisions of the same wiki page.

**Priya Shah**, the head of engineering, asked for something simple to describe and hard to achieve by hand: every change to a server should be repeatable, reviewable, and recorded. If a setting matters, it should live somewhere that survives a rebuild, and anyone on the team should be able to see who changed it and why.

That is the job of this book. We work alongside Maya as she replaces the wiki page with an Ansible project called `brightloom-infra`, a Git repository that describes every Brightloom server and every deployment step as code. By the final chapter, building a fresh staging environment, rolling out a new release, patching servers, and taking database backups are all single commands that do the same thing every time.

## The Problems With Hand-Built Servers

Brightloom's story is not unusual, and none of the people in it were careless. Hand configuration fails for structural reasons, and it helps to name them, because each one maps to a feature of Ansible we will use later.

### Snowflake servers

A *snowflake* server is one that is unique in ways nobody fully understands. It works, but you are afraid to touch it, and you could not rebuild it from scratch with confidence. Every manual fix that is not written down makes a server more of a snowflake. The production web server with the hand-edited upload limit was a snowflake: its real configuration existed only on its own disk.

### Configuration drift

*Configuration drift* is the slow divergence between servers that should be identical. Two web servers built from the same instructions drift apart when one gets an emergency fix and the other does not, when one is patched on Monday and the other on Friday, or when someone tries an experiment and forgets to undo it. Staging and production drift apart in the same way, which defeats the purpose of having a staging environment at all. If staging does not match production, a successful test in staging proves very little.

### Runbooks that rot

A wiki runbook is better than nothing, but it has no way to check itself. Nothing tells you when step 7 no longer works on the current Ubuntu release, or when the order of two steps matters. A runbook also depends on the person following it: one person types a command slightly differently, another skips a step because the server "already has that". The result is a set of servers that were all built "from the runbook" and are all different.

### No audit trail, slow rebuilds

When changes are made over SSH, the only record is a shell history file on the server, if it survives. You cannot review a change before it happens or answer "what changed on production last Thursday?" And if a rebuild takes a day of careful work, you avoid rebuilding, so servers get older and more fragile.

## Configuration Management and Infrastructure as Code

The industry answer to these problems has two names that overlap heavily. **Configuration management** is the practice of describing how systems should be configured and using a tool to make them match. **Infrastructure as code** (IaC) is the broader idea that infrastructure definitions should be text files, kept in version control and treated with the same care as application code: reviewed, tested, and released.

Put together, they change the way a team works. Instead of logging in to a server and editing a file, Maya edits a template in the `brightloom-infra` repository, opens a pull request, has a colleague review it, applies it to staging, checks the result, and then applies exactly the same change to production. The repository becomes the source of truth. If a server is lost, its replacement is built from the repository, and the upload limit that caused so much trouble is there because it is written in a file that every rebuild uses.

This gives you repeatability (the same files produce the same result everywhere), reviewability (changes are Git diffs), history (who changed what and why), documentation that cannot rot (the code that runs *is* the description), and speed (one server or forty takes the same effort from you).

## Idempotency and Desired State

The single most important idea in this book is **idempotency**. An operation is idempotent if running it once has the same effect as running it many times. Pressing the "floor 3" button in an elevator is idempotent: pressing it again does not send you to floor 6. Adding 1 to a number is not idempotent: every run changes the result.

Shell scripts are naturally *imperative*: they say "do this, then do that". Imperative steps are often not idempotent. Consider a fragment that might have appeared on Brightloom's wiki page, turned into a script:

setup-web.sh (the old way)

```
#!/bin/bash
# From the wiki: "Set up a web server"
apt-get update
apt-get install -y nginx
echo "client_max_body_size 10m;" >> /etc/nginx/conf.d/brightloom.conf
systemctl restart nginx
```

Run it once on a fresh server and it works. Run it a second time and the `echo` line appends the same directive again, so the file now contains it twice and nginx refuses to start with a "directive is duplicate" error. The restart also happens every time, whether or not anything changed, which briefly interrupts traffic for no reason. You can make shell scripts idempotent, but you have to write every check yourself: "if the line is not already there, add it", "if the file changed, restart". That code is tedious, easy to get wrong, and different in every script.

Ansible takes a different approach. Instead of describing steps, most Ansible tasks describe a **desired state**: "the package `nginx` should be present", "this file should have this content", "this service should be running". Each task is carried out by a **module**, a small program that knows how to check the current state and change it only if it differs from what you asked for. If the package is already installed, the module reports `ok` and does nothing. If it is missing, the module installs it and reports `changed`.

So you can run the same playbook as often as you like: a fresh server gets all the work, a correct server reports nothing changed, and a drifted server has only the drifted parts fixed.

Ansible also has a built-in way to act only when something changed. A task can *notify* a **handler**, such as "Reload nginx", and the handler runs only if the task actually changed something. Chapter 8 covers handlers in detail; for now, notice that the "restart every time" problem in the shell script goes away.

> **TIP**
>
> A good habit from day one: run every playbook twice when you write it. The first run should make changes; the second run should report zero changes. If the second run still reports changes, some task is not idempotent, and it will cause noise (and sometimes outages) later. We will practice this throughout the book.

We say Ansible is "declarative-ish" because it is a blend. Individual modules are declarative: you state the result you want. But a playbook is an ordered list of tasks that runs from top to bottom, so *order* is explicit and predictable. Many people find this blend easy to learn: it reads like a runbook, but each step is safe to repeat.

## How Ansible Works

Ansible has a small number of moving parts. Once you know them, most of the rest of the tool falls into place.

### The control node

The **control node** is the machine where Ansible is installed and from which you run it. For Maya it is her Linux laptop. It can be any Linux or macOS machine with Python. On Windows, the control node runs inside **WSL 2** (Windows Subsystem for Linux), because Ansible does not run natively on Windows as a control node. Chapter 2 sets up a control node step by step.

### Managed nodes

The servers Ansible configures are **managed nodes**, sometimes called hosts. Brightloom's managed nodes run Ubuntu Server 24.04 LTS. A Linux managed node needs only two things: an SSH server that the control node can log in to, and Python 3, which Ubuntu ships by default. There is no Ansible software to install on the managed nodes, no background service to keep running, and no extra port to open. This is what people mean when they call Ansible **agentless**.

### The inventory

The **inventory** is the list of managed nodes, organized into groups, with any connection details and variables they need. Brightloom's inventory groups servers by tier (`web`, `app`, `db`) and by environment (`staging`, `production`). Chapter 3 builds it.

### Modules

**Modules** are the units of work. There are modules for installing packages (`ansible.builtin.apt`), copying files (`ansible.builtin.copy`), rendering templates (`ansible.builtin.template`), managing services (`ansible.builtin.service`), creating users (`ansible.builtin.user`), and thousands of other jobs, from PostgreSQL databases to cloud resources. Chapter 4 introduces the essential ones.

### Playbooks, plays, and tasks

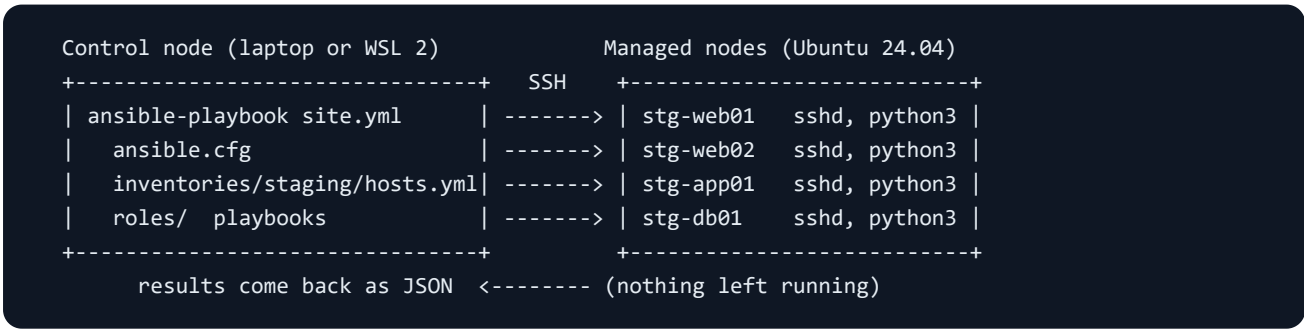
A **playbook** is a YAML file containing one or more **plays**. Each play maps a group of hosts to a list of **tasks**, and each task calls one module with some arguments. Later we group related tasks, templates, and handlers into **roles**, which are reusable building blocks such as "nginx" or "postgresql".

### What happens when you run a task

When you run a playbook, Ansible reads the inventory and the playbook on the control node, then works through the tasks in order. For each task and each targeted host, it does roughly the following:

1. Opens (or reuses) an SSH connection to the host, using the standard OpenSSH client and your SSH keys.
2. Packages the module code together with the task's arguments.
3. Copies that package to the managed node, usually into a temporary directory under the remote user's home folder.
4. Runs it with the node's Python interpreter, with `sudo` if the task asks for privilege escalation.
5. Collects the result, a small JSON document that says whether the task changed anything, failed, or was fine.
6. Removes the temporary files and moves on.

The picture below shows the overall flow for Brightloom's staging environment.



Ansible runs the same task on several hosts at once. How many it handles in parallel is controlled by a setting called **forks**, which defaults to 5; Brightloom raises it to 10 in Chapter 2. With the default *linear* strategy, every host finishes a task before any host starts the next one, which makes runs easy to follow. Chapter 24 explores other strategies and performance tuning.

The agentless design means there is nothing to install on the servers before you start, and it reuses SSH, which is already hardened and allowed through firewalls. The trade-off is that Ansible acts only when someone (or some scheduler) runs it.

## Push and Pull

Ansible is **push-based**: the control node connects out to the managed nodes and pushes changes when you run a command. Some other tools are **pull-based**: an agent on each server periodically contacts a central server, downloads its desired configuration, and applies it.

Push suits teams that want changes to happen when a person or pipeline decides, with output visible right away, and it makes ordering across machines easy ("database first, then app, then web, one at a time"). Pull suits huge fleets that must converge on their own. Ansible has a pull mode, `ansible-pull`, which checks out a Git repository on the managed node and applies a playbook locally; this book uses push throughout.

## Where Ansible Fits Among Other Tools

Ansible is not the only infrastructure tool, and it works best when you know what it is good at. The short version: Ansible is excellent at *configuring* machines that exist, *deploying* applications to them, and *orchestrating* multi-step operations across them in a specific order.

### Terraform and OpenTofu

Terraform and its open-source fork OpenTofu focus on *provisioning*: creating cloud resources such as virtual machines, networks, load balancers, and DNS records by calling provider APIs. They keep a state file that maps your code to real resources, and they are very good at planning and applying changes to that kind of infrastructure. Ansible can create cloud resources too, but most teams find the split natural: Terraform or OpenTofu creates the machines, and Ansible configures what runs on them. Chapter 20 shows how cloud tags become the contract between the two.

### cloud-init

cloud-init is built into most cloud images and runs once, at first boot, using "user data" you provide when creating a VM. It is ideal for the very first steps: creating a user, adding an SSH key, setting a hostname. It is not designed for ongoing management. A common pattern, which we use in Chapter 2 for the lab, is to let cloud-init create the automation user and key, and then let Ansible do everything else.

### Puppet, Chef, and Salt

Puppet, Chef, and Salt are mature configuration-management tools. They typically run an agent on each server that talks to a central server, which suits organizations that want continuous enforcement (Salt also has an SSH mode). Ansible's appeal is a lower barrier to entry: no agents, YAML instead of a programming language, and a short path from "I have SSH access" to "I am automating". Shell scripts remain useful for small jobs, but Ansible gives you inventory, parallelism, idempotent modules, and templates without rebuilding them in every script.

| Tool | Main job | How it runs | On the server |
|---|---|---|---|
| Ansible | Configure, deploy, orchestrate | Push over SSH | SSH and Python only |
| Terraform / OpenTofu | Provision cloud resources | API calls plus a state file | Nothing |
| cloud-init | First-boot setup | Once, at boot, from user data | Built into the image |
| Puppet / Chef / Salt | Configuration management | Usually agent and server | Usually an agent |
| Shell scripts | Anything, step by step | Whatever you write | The script itself |

None of these choices is permanent or exclusive. Plenty of teams use Terraform, cloud-init, and Ansible together, each for the part it handles best. Brightloom's hosting provider supplies the load balancer and the VMs; this book concentrates on everything Ansible does once the machines exist.

## ansible-core, the ansible Package, and Collections

When you install Ansible you will meet three names that are easy to confuse. It is worth getting them straight now, because they show up in documentation, error messages, and version numbers.

- **ansible-core** is the engine. It contains the command-line tools (`ansible`, `ansible-playbook`, `ansible-inventory`, `ansible-doc`, `ansible-vault`, `ansible-galaxy`, and others), the playbook language, and one collection of essential modules and plugins called `ansible.builtin`.
- **Collections** are the distribution format for everything else: modules, plugins, and roles grouped under a namespace. Examples we will use are `ansible.posix` (SSH authorized keys, sysctl, firewalld), `community.general` (ufw, timezone, and much more), `community.postgresql` (databases and users), and `community.crypto` (keys and certificates).
- The `ansible` **community package** bundles a specific ansible-core release together with a curated set of popular collections. Installing it gives you a "batteries included" setup.

The two have separate version numbers, and the community package version is not the same as the ansible-core version inside it. Rather than memorize a mapping, run `ansible --version`: it shows the core version in square brackets, as in `ansible [core 2.x.y]`. This book targets recent versions of ansible-core, with 2.17 and later as our baseline.

Collections are also why you will see long module names throughout the book. Every module has a **fully qualified collection name** (FQCN): the namespace, the collection name, and the module name, separated by dots. `ansible.builtin.apt` is the `apt` module from the `ansible.builtin` collection; `community.general.ufw` is the `ufw` module from `community.general`. Ansible still accepts many short names such as plain `apt`, but short names can be ambiguous when two collections provide a module with the same name, and linting tools flag them. We use the FQCN everywhere, and we recommend you do the same.

> **WARNING**
>
> Blog posts and older examples often use short module names, `sudo: yes`, `with_items`, and other legacy syntax. Much of it still works, but some of it has been removed, and recent versions of ansible-core are stricter about templating and conditionals than older ones. When you copy an example from the internet, translate it to modern style (FQCN, `become: true`, `loop`) and test it in staging before trusting it.

## A First Look at a Playbook

You will write your first real playbook in Chapter 5, after setting up a control node and an inventory. But it helps to see the shape of one now. Here is a tiny playbook that makes sure every server keeps accurate time, which matters for logs, TLS certificates, and invoice timestamps.

A preview (we build the real version in Chapter 5)

```
---
- name: Keep time in sync on every server
  hosts: all
  become: true
  tasks:
    - name: Install chrony
      ansible.builtin.apt:
        name: chrony
        state: present
        update_cache: true

    - name: Ensure chrony is running and enabled at boot
      ansible.builtin.service:
        name: chrony
        state: started
        enabled: true
```

Even without knowing any Ansible, you can probably read it. The file starts with `---`, the YAML document marker. It contains one play, named "Keep time in sync on every server", which targets `hosts: all`, meaning every server in the inventory. The line `become: true` tells Ansible to use privilege escalation (sudo) because installing packages needs root. Then come two tasks. The first asks

the `apt` module for a desired state: the `chrony` package should be present, and the package cache should be refreshed first. The second asks the `service` module for another desired state: chrony should be running now and enabled at boot.

Notice what is *not* there. There is no "if chrony is not installed then install it" logic, and no check for whether the service is already running. The modules handle that. If you run this playbook against a server that already has chrony installed and running, both tasks report `ok` and nothing changes.

When you run a playbook, Ansible prints a section per play and per task, a status per host, and a final recap. You will see something like this:

Output (abbreviated)

```
PLAY [Keep time in sync on every server] ******************************

TASK [Gathering Facts] ************************************************
ok: [stg-web01]
ok: [stg-web02]
ok: [stg-app01]
ok: [stg-db01]

TASK [Install chrony] *************************************************
changed: [stg-web01]
changed: [stg-web02]
changed: [stg-app01]
changed: [stg-db01]

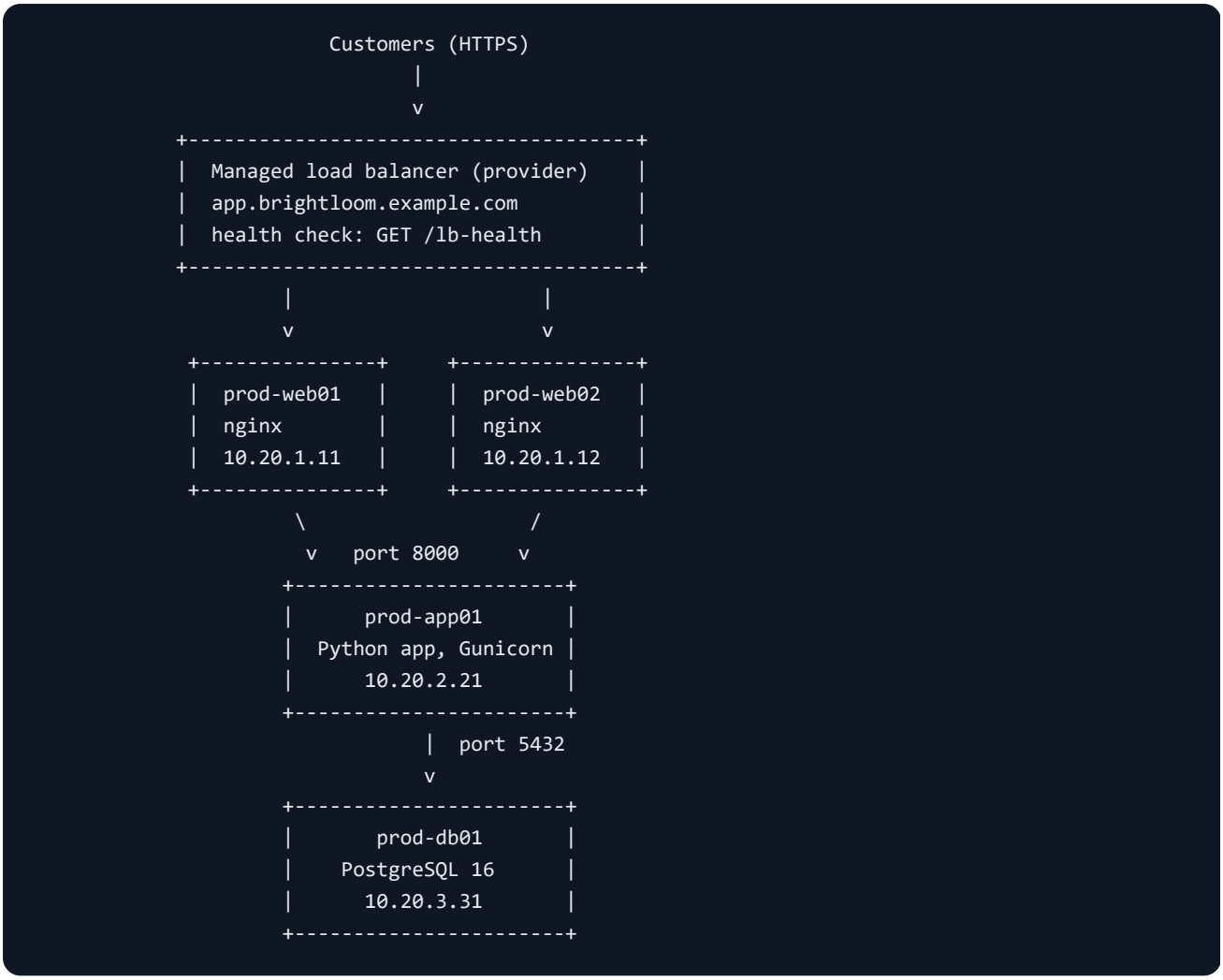
TASK [Ensure chrony is running and enabled at boot] *******************
ok: [stg-web01]
ok: [stg-web02]
ok: [stg-app01]
ok: [stg-db01]

PLAY RECAP ************************************************************
stg-app01 : ok=3  changed=1  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
stg-db01  : ok=3  changed=1  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
stg-web01 : ok=3  changed=1  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
stg-web02 : ok=3  changed=1  unreachable=0  failed=0  skipped=0  rescued=0  ignored=0
```

The "Gathering Facts" task runs automatically at the start of each play: Ansible collects details about each host, such as its operating system and IP addresses, so later tasks can use them. Installing chrony reported `changed`. The service task reported `ok` because, on Ubuntu, installing chrony also starts and enables it, so the desired state was already true. Run the playbook a second time and the recap shows `changed=0` for every host. That second, quiet run is idempotency in action.

## Meet the Brightloom Platform

Throughout the book we automate one realistic system rather than a collection of unrelated examples. Every chapter adds a piece, and by Chapter 28 the pieces form a complete platform. Here is the production architecture.




The load balancer belongs to the hosting provider and is outside Ansible's scope. It sends public HTTPS traffic to the two web servers and checks the path `/lb-health` on each one to decide whether it should receive traffic. The web servers run nginx, terminate TLS, and pass requests to the application server on port 8000. The application server runs Brightloom Invoices, a Python web app served by Gunicorn and managed by systemd. The database server runs PostgreSQL 16 from Ubuntu's own repositories.

Staging is the same shape with different names and addresses. Here is the full list of servers we manage.

| Inventory host | Environment | Group | Address | Runs |
|---|---|---|---|---|
| stg-web01 | staging | web | 10.10.1.11 | nginx |
| stg-web02 | staging | web | 10.10.1.12 | nginx |
| stg-app01 | staging | app | 10.10.2.21 | Gunicorn app |
| stg-db01 | staging | db | 10.10.3.31 | PostgreSQL 16 |
| prod-web01 | production | web | 10.20.1.11 | nginx |
| prod-web02 | production | web | 10.20.1.12 | nginx |

| | | | | |
|---|---|---|---|---|
| prod-app01 | production | app | 10.20.2.21 | Gunicorn app |
| prod-db01 | production | db | 10.20.3.31 | PostgreSQL 16 |

All managed nodes run Ubuntu Server 24.04 LTS, listen for SSH on port 22, and use the Python 3.12 that ships with the operating system. Ansible logs in as a dedicated automation user called `deploy`, which has passwordless sudo. The control node and the management network live in `10.99.0.0/24`, and later chapters allow SSH only from there.

You will not need a production environment of your own. In Chapter 2 you build a **lab** of four small Ubuntu virtual machines, and that lab *is* your staging environment: you simply put your VMs' addresses into the staging inventory. Production appears in the book as the environment we are careful with, so we can discuss safeguards realistically without asking you to own one.

> **NOTE**
>
> Brightloom, its people, its domain `brightloom.example.com`, and its IP addresses are all fictional. The `example.com` domain is reserved for documentation, and the `10.x.x.x` ranges are private addresses. Replace them with your own values when you apply these ideas at work.

## The Plan for This Book

The book is organized in six parts that follow the path Maya takes in replacing the wiki page.

- **Part I, Foundations (Chapters 1 to 5):** control node, lab, inventories, ad hoc commands, first playbook.
- **Part II, Writing Playbooks (Chapters 6 to 10):** variables, templates, conditionals, loops, handlers, errors, and organization.
- **Part III, Roles and Reuse (Chapters 11 to 15):** the `common`, `nginx`, `postgresql`, and `brightloom_app` roles, plus collections.
- **Part IV, Secrets, Security, and Environments (Chapters 16 to 20):** Vault, hardening, firewalls, TLS, environments, and dynamic inventory.
- **Part V, Deploying and Operating (Chapters 21 to 24):** releases and rollbacks, rolling updates, patching, backups, and performance.
- **Part VI, Quality and Production (Chapters 25 to 28):** testing, CI/CD, debugging, extending, and the complete platform.

By the end, `brightloom-infra` will contain a `site.yml` that builds every server from a plain Ubuntu install, a `deploy.yml` that ships releases with one command, a `rollback.yml` for the bad days, and the safety checks that make those commands trustworthy.