Ansible

for Network Engineers

From First Playbook to Production Practice

JOZEF BAROŠ

Ansible

for Network Engineers

From First Playbook to Production Practice

Examples target ansible-core 2.21 and Python 3.12+

Cisco IOS-XE, NX-OS and IOS-XR, with Juniper Junos and Fortinet FortiOS

First edition

JOZEF BAROŠ

Ansible for Network Engineers

From First Playbook to Production Practice

First edition, 2026. Written and produced by Jozef Baroš under the ProDigitalJ imprint.

TRADEMARKS

Ansible and Red Hat Ansible Automation Platform are trademarks of Red Hat, Inc. Cisco, IOS, IOS-XE, IOS-XR, NX-OS, Catalyst, Nexus and DevNet are trademarks of Cisco Systems, Inc. Junos and Juniper Networks are trademarks of Juniper Networks, Inc. FortiGate and FortiOS are trademarks of Fortinet, Inc. All other product names, logos and brands are the property of their respective owners and are used here for identification only. This book is an independent publication and is neither affiliated with, authorised by, sponsored by, nor endorsed by any of them.

DISCLAIMER — PLEASE READ

The commands, playbooks and configurations in this book can change or disrupt a production network. Several of them are destructive by design: the overridden state removes configuration you did not list, deleted with no arguments removes every resource of its type, configure replace can remove your own management access, and a software upgrade reloads the device. They are presented that way deliberately, because pretending otherwise would make the book less useful and considerably more dangerous.

Every example is intended to be run first in a laboratory, then against a single device in check mode, and only then against anything that matters. Test everything. Verify against your vendor's current documentation, which is authoritative where this book and the vendor disagree. Follow your organisation's change management process; nothing here replaces it.

The author accepts no liability for any loss, outage, data loss or damage of any kind arising from the use or misuse of the material in this book. You are responsible for what you run on your network.

VERSIONS

Examples target ansible-core 2.21 with Python 3.12 or newer on the control node, and the cisco.ios, cisco.nxos, cisco.iosxr, junipernetworks.junos and fortinet.fortios collections at the versions pinned in Appendix C. Collections release independently of the engine and behaviour does change between major versions; where a difference matters, the text says so.

Found an error, or something that no longer matches reality? Corrections are genuinely welcome and are the fastest way to make the next edition better.

Preface

Why this book exists, and how to read it

Most network engineers arrive at automation the same way. Something breaks at two in the morning, or a change window overruns, or an auditor asks a question nobody can answer, and it becomes obvious that the network has no authoritative description of itself. The configuration is on the devices. What the configuration is supposed to be lives in a design document that stopped being true in 2019, a spreadsheet, and the memory of whoever was on the project.

This book is about closing that gap. Its central claim is a single sentence: make the description of the network the thing that configures the network. Everything else — the modules, the templates, the pipelines — is machinery in service of that idea. Ansible is the vehicle, and a good one, but the techniques that will still matter in ten years are not the module names.

WHO THIS IS FOR

Network engineers with no Ansible experience, and network automation engineers who want the parts nobody teaches: how to design a data model that survives its second year, how to phase a rollout so nobody loses access, and how to build something a colleague can run when you are on holiday. You are assumed to know networking. You are not assumed to know Python, YAML, Git, or what a playbook is.

It is Cisco-centred by design — IOS-XE, NX-OS and IOS-XR — with Juniper Junos and Fortinet FortiOS appearing where the contrast teaches something worth knowing, which is mostly about commit models rather than syntax.

HOW IT IS ORGANISED

Parts I and II build the foundation: why manual configuration is expensive, and then Ansible itself — inventory, playbooks, variables, roles and error handling — with almost nothing that is network-specific, because those concepts are the same for a switch and a web server.

Part III is where it becomes network automation: connection plugins, credentials, the Cisco collections, and Chapter 15 on resource modules and

their seven states, which is the most important chapter in the book.

Parts IV and V turn that into a practice: a source of truth, templates that generate whole devices, drift detection, backups, documentation, upgrades and validation.

Part VI is three complete projects — campus 802.1X, a data centre fabric with WAN policy, and multi-vendor security — taken from a stated requirement to a validated rollout.

Part VII is about doing all of it as a team: Git workflow, testing, CI/CD, the automation platform, credentials, and how a practice survives the person who started it.

HOW TO READ IT

Straight through, if you are starting from nothing. Every chapter assumes the ones before it, and the sequence is deliberate: the read-only, zero-risk work comes before anything that can break a device, because that ordering is also the right adoption order in a real organisation.

If you already automate networks, read Chapter 15 first, then Chapters 19 and 21, and use the rest as reference. Those three carry most of what separates a repository of clever playbooks from an automation practice.

THE EXERCISES

There are no quizzes. Every sub-section ends with exercises that include the complete source code, inline — a task, the constraints, a full solution, the command that verifies it, and an explanation of why it works. Some chapters end with a stretch exercise that has hints and no solution, because the answer depends on your network.

The exercises are the book. Reading them is worth something; running them is worth considerably more, and several of them — the restore drill, the credential rotation, the independent-run test — are designed to find problems that no amount of reading will.

CONVENTIONS

Code blocks are set in a monospaced face with three registers: commands you type, output the device returns, and comments in a muted italic. Comments are used heavily, and they explain the construct rather than restating the line.

Four kinds of highlighted box appear throughout: Did you know? for background worth having, Exam Tip for the thing that is easy to get wrong, Common Pitfall for the mistake most people make once, and From the Field for what actually happens in production. The pitfalls are the ones to read twice.

A WORD ABOUT WHERE YOUR TIME WILL GO

In each of the three projects in Part VI, the automation is roughly a fifth of the effort. The rest goes on identifying endpoints nobody inventoried, agreeing numbering conventions, and persuading rule owners to justify what they own. That is not a failure of automation — it is what automation reveals. The tooling makes the undecided things visible, and then somebody still has to decide them.

Network automation does not replace network engineering. It demands more of it, in writing, where somebody else can read it.

Jozef Baroš

Contents

Contents (continued)

PART ONE

Foundations

Why automation, what Ansible is, and the ground you need under you

Four chapters stand between you and your first playbook, and none of them are filler. The first explains why the way you configure networks today does not scale, in terms you will recognise from your own change windows. The second places Ansible precisely among the alternatives, including the cases where it is the wrong tool. The third gives you the three languages you will read every day — YAML, structured data, and the transports that carry configuration onto a device. The fourth builds the lab that every later chapter assumes.

If you have automated nothing before, start here and read straight through. If you already write Python against network devices, Chapter 3 will be a skim and Chapter 4 will still be worth twenty minutes, because the lab conventions established there are used for the rest of the book.

Chapter	What it gives you	Hands-on
1 The Case for Network Automation	Language for the problem, and an honest cost model	Audit exercises
2 Ansible in the Network Landscape	A correct mental model of the agentless architecture	Comparison work
3 The Prerequisites You Actually Need	YAML, data structures, SSH / NETCONF / RESTCONF, Git	Code exercises
4 Building Your Lab	A working control node and five reachable devices	Full lab build

🌍 **From the Field — A note on versions**

Every command and playbook in this book was written against **ansible-core 2.21**, released in May 2026, which requires **Python 3.12 or newer** on the control node. Where a large installed base still runs an older branch — Ansible Automation Platform execution environments commonly ship 2.18 or 2.20 — the text calls out the difference at the point where the syntax

> actually diverges. Version drift is a real cost in automation, and pretending otherwise would be a poor start to a book about consistency.

Chapter 1 — The Case for Network Automation

You do not need to be sold on automation to read this book, and this chapter is not a sales pitch. It is an attempt to name a problem precisely, because a problem you can name is a problem you can measure, and a problem you can measure is one you can argue about with a manager who controls a budget.

The problem is not that typing is slow. Network engineers are fast typists. The problem is that a network configured by hand has no authoritative description of itself, and everything expensive follows from that.

1.1 What Manual Configuration Actually Costs

Configuration drift, defined properly

Configuration drift is the gap between how a device is configured and how you believe it is configured. It is not caused by carelessness. It is caused by the fact that every manual change is an independent act of translation: an engineer reads an intention — in a ticket, a design document, or their own head — and retypes it into a terminal. Translation has an error rate. Repeat the translation across forty devices at two in the morning and the error rate stops being theoretical.

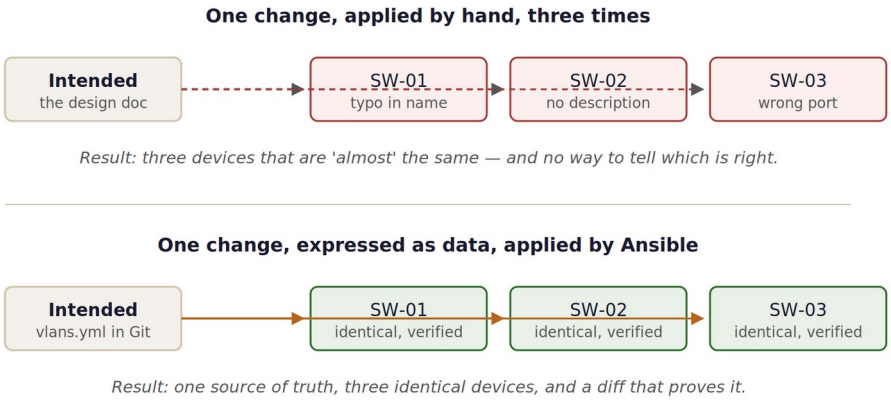

Figure 1.1 The same change, applied by hand and applied from data. The difference is not speed — it is whether the result is knowable.

Notice what the lower half of Figure 1.1 does not claim. It does not claim the automated change is faster to write; the first time, it is slower. It claims that the intended state exists as a file, that the file was applied identically to every device, and that you can prove it. That proof is the whole product.

> ⚠️ **Common Pitfall — Drift is not fixed by better documentation**
>
> The instinctive response to drift is a wiki page, a spreadsheet, or a Visio diagram that records what the network should look like. These decay because nothing forces them to stay true — a device does not reject a change because the spreadsheet disagrees. Documentation that is not executed is documentation that is wrong. The fix is to make the description of the network the thing that configures the network, so that a stale description produces a visible failure instead of a silent lie.

The four costs, in the order they hurt

When you build a business case, resist the urge to lead with hours saved. It is the weakest of the four costs and the easiest for a sceptical manager to dismiss.

- **Risk of outage.** The majority of network incidents are caused by changes, and the majority of those by human error during the change. A process that removes retyping removes a whole class of incident.
- **Time to recover.** When a device fails and is replaced, how long does it take to restore its exact configuration? If the answer involves reading a backup file and re-entering it by hand, the answer is hours. If the configuration is generated from data, the answer is minutes.
- **Audit and compliance.** Prove that SNMP community strings are identical on all 400 access switches. By hand, this is a week of work and the result is stale by the time you finish. Automated, it is a report you can run before the meeting starts.
- **Engineer hours.** Real, but the least interesting, because it is the one your manager has already discounted.

> 🌍 **From the Field — What actually convinces management**
> The number that lands is rarely the hours saved — it is the change failure rate and the mean time to restore service. Both are already tracked by most operations teams, both are already reported upward, and both improve visibly within a quarter of adopting even basic automation. Frame your first project as risk reduction with a measurable before-and-after, and you will get the second project funded.

Why scripts alone did not solve this

Most network engineers have already written automation. It is usually a Python script, or an Expect script, or a spreadsheet of commands pasted into a terminal. These work, and dismissing them would be dishonest. What they lack is not capability but **structure**: there is no shared convention for where the list of devices lives, how credentials are supplied, what happens when a device is unreachable, or how someone else runs your script safely. Every script solves those problems again, differently, and only the author knows how.

Ansible's real contribution is not that it can send a command to a switch — Netmiko has done that for years. It is that it imposes an opinionated structure on the boring parts, so that two engineers who have never met can read each other's automation.

> 💡 **Did you know? — The word comes from science fiction**
> Ansible is named after a fictional device from Ursula K. Le Guin's novels: a communicator that transmits instantly across any distance. Michael DeHaan chose the name for the original project in 2012. The metaphor is apt — the tool's defining trait is that it reaches remote systems without anything having been placed there first.

> **Summary**
> Manual configuration is expensive not because typing is slow but because it leaves the

network without an authoritative description of itself. Configuration drift — the gap between actual and believed state — grows with every hand-applied change and cannot be closed with documentation, because documentation is not enforced by anything. The costs fall in four categories, of which outage risk and recovery time are the two that persuade budget holders. Scripts partially address the problem but each one reinvents the surrounding conventions, which is precisely the gap Ansible fills.

Key Takeaways

Drift is the gap between actual and believed configuration, and it is structural, not sloppy. **Documentation that is not executed will be wrong.** Lead a business case with **change failure rate and MTTR**, not hours saved. Ansible's value is **shared structure**, not raw capability — Python could already reach the device.

🔧 Exercise 1.1 — Cost your own last change window

Before writing any automation, quantify what you are replacing. Take the most recent multi-device change you performed by hand.

Your turn. Build it so that:

1. Record the number of devices touched and the wall-clock duration of the window.
2. Count the discrete commands you typed, including the ones you typed twice.
3. Note any device where the result differed from the others, and why.
4. Estimate what a second engineer would need in order to repeat the change identically without asking you a question.

SOLUTION

A simple worksheet you can keep in the repo you are about to create

```
# change-audit.yml  --  fill one of these in after every manual change window
change:
  ticket: CHG0041982
  date: 2026-08-14
  devices_touched: 18
  duration_minutes: 195
  commands_typed: 306
  engineers: 2

  # the honest part
  devices_that_differed: 2
  cause_of_difference: "interface description copied from the wrong site"
  rollback_needed: false

  # the question that matters most
```

```
  reproducible_by_someone_else: false
  reason: "the VLAN-to-site mapping only exists in my notes"
```

VERIFY

```
# Three of these files is enough to see a pattern. Ten is an argument.
$ ls change-audit-*.yml | wc -l
$ grep -c "reproducible_by_someone_else: false" change-audit-*.yml
```

WHY IT WORKS

The field that matters is the last one. Duration and command counts make a nice slide, but **reproducibility** is the property automation actually delivers, and it is the one that fails most often in manual work. If the honest answer is that nobody else could repeat your change without interviewing you, then the knowledge is in your head rather than in the network, and the organisation is one resignation away from losing it.

Keeping these files in YAML is not an accident. You are about to spend an entire book reading and writing this format, and using it for something trivially useful first is the cheapest possible introduction.

🔧 Exercise 1.2 — Find drift you already have

Drift is easier to believe once you have seen your own. Pick a configuration element that should be identical across a group of devices — NTP servers, SNMP communities, AAA servers, syslog destinations, or the login banner — and compare it across at least five devices of the same role.

Your turn. Build it so that:

1. Collect the relevant configuration section from each device by any means you already have.
2. Normalise the output so ordering and whitespace do not create false differences.
3. Report which devices deviate from the majority.

SOLUTION

A deliberately low-tech comparison — no Ansible required yet

```
#!/usr/bin/env bash
# drift-check.sh  --  compare one config section across a set of devices
```

```
# usage: ./drift-check.sh "ntp server" core-sw1 core-sw2 core-sw3 core-sw4
core-sw5

SECTION="$1"; shift

for DEV in "$@"; do
    # ssh runs the command and exits; adjust the command for your platform
    ssh -o StrictHostKeyChecking=no "$DEV" "show running-config | include ^$
{SECTION}" \
        | tr -s ' ' | sort > "/tmp/drift.${DEV}.txt"
done

echo "=== unique configurations found ==="
md5sum /tmp/drift.*.txt | awk '{print $1}' | sort | uniq -c | sort -rn

echo
echo "=== devices grouped by fingerprint ==="
for F in /tmp/drift.*.txt; do
    printf "%s  %s\n" "$(md5sum "$F" | cut -c1-8)" "$(basename "$F" .txt | cut
-d. -f2)"
done | sort
```

VERIFY

```
$ ./drift-check.sh "ntp server" core-sw1 core-sw2 core-sw3 core-sw4 core-sw5

=== unique configurations found ===
      4 9f2a1c4e8b7d3a6f5c2e1b0a9d8c7f6e
      1 3c7e9a2b1f4d8c6a5e0b3d7f2a9c4e1b     <-- one device disagrees
```

WHY IT WORKS

The script is intentionally crude: raw SSH, a hash, and a count. It is here to make a point that gets lost once tooling gets sophisticated — **drift detection is fundamentally about comparing a normalised representation of state across a group**. Everything Ansible adds to this (inventory, structured facts, real parsing, reporting) is refinement, not a different idea.

The normalisation step is the part people skip and then regret. Devices return configuration lines in an order that is stable per platform but not guaranteed across software versions, and a stray double space will produce a different hash for two identical configurations. Sorting and squeezing whitespace removes both classes of false positive.

If every device came back identical, run it again against a group that has been in production for longer, or pick an element that engineers edit under time pressure. Drift accumulates fastest in whatever is touched during incidents.

1.2 What Automation Changes, and What It Does Not

The honest limits

A book that only argues for its subject is a brochure. Three things are worth saying plainly before you invest weeks of learning.

Automation does not remove the need to understand the network. It amplifies whatever understanding you have. An engineer who does not know why a BGP session is failing will not be helped by the ability to reconfigure four hundred sessions in a minute — they will be harmed by it. Every chapter in this book assumes the network knowledge and adds the tooling on top.

Automation moves work rather than eliminating it. You will stop typing VLAN commands and start maintaining data files, templates, and pipelines. This is a good trade, because the new work is reviewable, versioned, and shared, but it is a trade and not a subtraction. Teams that budget zero hours for maintaining their automation abandon it within a year.

Automation makes mistakes faster. This is the one that deserves fear. A wrong command typed by hand affects one device; the same error in a playbook affects every device in the inventory, in parallel, in under a minute. The discipline that this book builds — check mode, staged rollouts, pre- and post-change validation, peer review — exists precisely because the blast radius grows with the leverage.

> ⚠️ **Common Pitfall — Do not automate a process you do not trust**
>
> The temptation with a new tool is to point it at the messiest, most repetitive task you have. That task is usually messy because the process behind it is unclear, and automating an unclear process produces a fast, confident, wrong result. Fix the process on paper first, then encode it. If you cannot write down the rule that decides which VLAN a port belongs to, you are not ready to automate port configuration.

Choosing a first project

The right first automation project has four properties: it is read-only, it runs against many devices, its output is immediately useful to someone other than you, and its failure mode is embarrassment rather than an outage. Configuration backup and inventory reporting both qualify, which is why they appear early in almost every successful adoption story and why Part V of this book treats them properly.

The wrong first project is a configuration push to production during a change window, no matter how simple the change looks. Not because you cannot do it, but because you will have no track record to fall back on when a colleague reasonably asks whether the tool can be trusted.

✅ **Exam Tip — A useful adoption sequence**

Read-only reporting → configuration backup → compliance checking without remediation → configuration push in a lab → configuration push to a small production group → full deployment. Each step earns the credibility that funds the next. Skipping to step five is the single most common way automation initiatives die inside network teams.

Summary

Automation amplifies existing understanding rather than substituting for it, converts configuration work into data and template maintenance rather than eliminating work, and increases the blast radius of every mistake in proportion to the leverage it grants. These are not reasons to avoid it, but they determine how you should adopt it: encode only processes you can already articulate, and sequence your first projects so that each one earns the trust needed for the next. Read-only work first, production configuration pushes last.

Key Takeaways

Automation is a **force multiplier on understanding**, not a replacement for it. Work is **moved into data and templates**, not removed — budget for its maintenance. **Blast radius scales with leverage**, which is why validation discipline is mandatory rather than optional. Start with **read-only reporting and backup**; earn the right to push configuration.

🔧 **Exercise 1.3 — Write the rule before you write the code**

Pick a decision you make routinely by judgement rather than by rule — which VLAN a new access port belongs to, which QoS policy a WAN circuit gets, or which devices are in scope for a given software upgrade. Write the rule down completely enough that a colleague could apply it without asking you anything.

Your turn. Build it so that:

1. Express the decision as data plus a rule, not as prose.
2. Include the exceptions, because there are always exceptions.
3. Make the default explicit — what happens when nothing matches?

SOLUTION

An articulated rule, expressed the way Ansible will later consume it

```
# port-policy.yml  --  the rule that currently lives in your head
port_policy:
  # match on the interface description convention already in use
  rules:
    - match: "^AP-"          # access point uplinks
      vlan: 110
      voice_vlan: null
      portfast: true
```

```
    description_prefix: "AP"

  - match: "^PRN-"          # printers
    vlan: 130
    voice_vlan: null
    portfast: true

  - match: "^PHN-"          # phones with a daisy-chained PC
    vlan: 120
    voice_vlan: 121
    portfast: true

# the exceptions you would otherwise remember, and eventually forget
exceptions:
  - device: core-sw2
    interface: GigabitEthernet1/0/47
    reason: "legacy badge reader, cannot be moved until 2027 refresh"
    vlan: 999

# the case people forget to specify
default:
  vlan: 100
  voice_vlan: null
  portfast: true
  shutdown: false
```

VERIFY

```
# There is nothing to run yet -- the test is human.
# Hand the file to a colleague with this question:
#   "Port Gi1/0/12 on core-sw1 is described PHN-3F-DESK-14. What goes on it?"
# If they answer without asking you a follow-up question, the rule is complete.
```

WHY IT WORKS

This exercise produces no automation and is still the most valuable one in the chapter. The file you have written is a **source of truth** in miniature: data that describes intent, separated from the mechanism that applies it. Part IV of this book is entirely about growing files like this into a model that drives real configuration, and the hard part there is never the YAML — it is the articulation you just did.

Pay attention to how uncomfortable the **exceptions** block felt to write. Exceptions are where undocumented knowledge concentrates, and an automation project that has no place to record

them will either break on contact with reality or quietly overwrite something that mattered. Give exceptions a home from the first day.

The **default** matters just as much. Manual processes handle unmatched cases by an engineer noticing and thinking; automated processes handle them by doing whatever the code happens to do. Making the default explicit converts a silent behaviour into a decision.

> 🔧 **Stretch Exercise 1.A — Build the argument you will actually have to make**
>
> Using the audit data from Exercise 1.1 and the drift results from Exercise 1.2, draft a one-page proposal for a first automation project in your own environment. Assume the reader is a manager who has heard the word automation too many times and controls one week of your time.
>
> **No solution is printed for this one.** Hints:
>
> 1. Lead with a number that is already on your team's dashboard — change failure rate, MTTR, or audit findings — not with engineer hours.
> 2. Propose something read-only. The proposal is more credible when the downside is bounded and you say so explicitly.
> 3. Name what you will not do in week one. Scoping out production configuration changes is a feature of the proposal, not a weakness.
> 4. State how you will measure the result, and pick a metric you can produce without asking anyone else for data.

End of the free sample

You have read Chapter 1 of thirty-four.

7 parts · 34 chapters · 5 appendices · 513 pages

50 exercises with complete, runnable source code, inline

23 stretch exercises · 41 original diagrams

Cisco IOS-XE, NX-OS and IOS-XR; Juniper Junos; Fortinet FortiOS

Three end-to-end projects: campus 802.1X, DC fabric, multi-vendor security

Get the complete book

leanpub.com/ansible-network

JOZEF BAROŠ

ProDigitalJ