Ansible for Network Automation: A Concise Guide

Lucio Jankok

October 28, 2023

Contact

Contact	Phone Number	Email
Lucio Jankok	06-11118887	ljankok@systemwise.nl

History

Version No.	Issue Date	Status	Reason for Change
0.1	02-08-2021	Initial	First Draft
0.2	02-08-2021	Under Review	Ready for Peer Review
0.3	20-01-2022	Final	First Release
0.4	24-01-2022	Published	Additional Use Cases
0.5	27-01-2022	Published	Content Reorganization
0.6	22-07-2022	Published	Text Updated
0.7	26-10-2023	Published	Content Updated

Review

Reviewer Details	Version No.	Date

Abstract

This guide provides a practical introduction to implementing network automation with Ansible, with emphasis on brownfield constraints common in enterprise networks. It focuses on a decision hierarchy for selecting Ansible network resource modules, parsing modules, and command modules; and it provides concrete playbook patterns that prioritize readability, determinism, and reuse.

Scope and assumptions

- **Audience:** network engineers who already understand automation concepts, but are new to Ansible.
- **Scope:** Ansible concepts, module categories relevant to network automation, usage levels, brownfield provisioning method, essential constructs, and playbook design via representative use cases.
- **Non-goals:** deep Ansible internals, Python module authoring, and vendor-specific design guides beyond what is required to explain the patterns.

Terminology and formatting conventions

- **Concepts** use Title Case (e.g., Managed Node, Control Node).
- **Filesystem constructs** use monospace (e.g., `group_vars`, `vars`, `defaults`).
- **Module names** are written as module identifiers (e.g., `cisco.ios.ios_command`).
- Code blocks are shown verbatim; YAML and CLI output appear in a monospaced listing.

Production readiness criteria

Throughout this guide, “production ready” means that automation is:

- **Idempotent:** repeated execution converges to the same desired state.
- **Deterministic:** behavior does not depend on incidental device state not explicitly checked.
- **Failure-aware:** errors are handled intentionally (assertions, guarded blocks, clear stop conditions).
- **Observable:** outputs are logged meaningfully; critical decisions are inspectable.
- **Safe:** changes are bounded; destructive operations are explicit; reload/upgrade steps include gates.

Contents

Part I

Foundations

Chapter 1

Introduction

Learning outcomes

After this section, you can:

- describe the scope and target audience of this guide,
- navigate the document based on its topic map,
- understand how later sections build toward concrete playbook patterns.

After reading and understanding this guide, you will have a basic understanding of how network automation is implemented with Ansible. I wrote this document because I could not find a short, concise guide that covers the essentials of Ansible for network automation.

The target audience is the network engineer who is experienced in network automation, but not particularly with Ansible.

In this document the following topics are presented:

Topics	**Elaboration**
1. Ansible Concepts	Ansible terminology explained and interrelationships depicted.
2. Ansible Modules	Ansible approach to hiding complexity explained.
3. Ansible Usage	Different levels of usage and implementation types explained.
4. Brownfield Provisioning	Best-practice approach for brownfield scenarios presented.
5. Ansible Constructs	Nuts and bolts of Ansible for network automation explained.
6. Playbooks	Design guidance presented through specific use cases.

Chapter 2

Ansible Concepts

Learning outcomes

After this section, you can:

- define the core objects in Ansible network automation (Control Node, Managed Node, Inventory, Playbook, Task, Module),
- explain where variables enter the execution path,
- recognize the separation between playbook logic and module implementation.

- **Control Node** — the Ansible machine that executes playbooks.
- **Managed Node** — a host in the inventory (for network automation: typically a network device).
- **Inventory** — the list of Managed Nodes.
- **Collection** — a packaged set of plugins, roles, and modules.
- **Modules** — the units of code execution.
- **Task** — a unit of action that invokes a module.
- **Playbook** — an ordered list of tasks; can include variables.

Diagram 1: Ansible concepts interrelationship

Figure 2.1 provides a compact map of the main objects (playbook, tasks, modules, inventory, variables) and how they combine to produce device-level changes. In this document, use it primarily as an orientation aid: (1) execution flows left-to-right (Control Node → Playbook → Tasks → Modules), and (2) variables are injected from multiple sources into modules and templating.

2.1 Ansible modules: the unit of code execution

Design rule

Keep playbooks declarative by pushing parsing, platform specifics, and procedural logic into modules.

The Ansible approach is to move parsing, regular expressions, and vendor/platform specifics into modules. A module either:

- returns structured data (variables or lists/dictionaries of key–value pairs), or
- performs a platform operation (configuration change, state query, file transfer, etc.).

Modules are therefore intended to keep playbooks readable by extracting complexity into reusable code. Modules are typically written in a procedural language and can be platform specific.

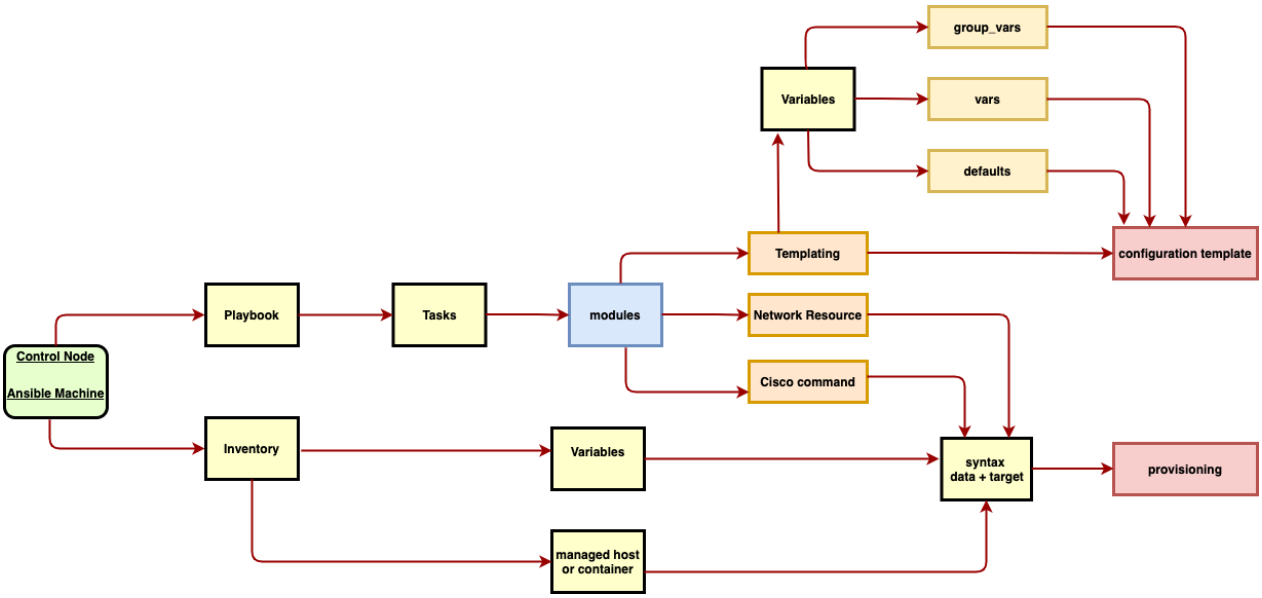

Figure 2.1: Ansible concepts interrelationship (execution flow and variable injection).

> A module's job is not only to parse output programmatically, but also to make the playbook syntax as platform-agnostic as possible.

In practice, Ansible enables a declarative style: tasks specify **what** you want rather than the low-level **how**. Ansible uses YAML for playbooks and Jinja2 for templates; combined, they form a common approach to **Infrastructure as Code**.

2.2 The Ansible constructs you need to know

Always parse device-specific configuration using Ansible modules in order to take advantage of native Ansible constructs such as:

1. the `when` clause for conditional execution, and
2. the `assert` module for configuration and operational-state compliance checks.

The module types used most in network automation are:

1. the `template` module to generate configuration (Infrastructure as Code approach),
2. command/config modules to generate and apply platform-specific commands on Managed Nodes,
3. platform facts modules to acquire information in a structured manner, and
4. the `ansible.utils.cli_parse` module to parse unstructured output into structured output.

Parsing engines supported by `cli_parse`

1. Ansible native
2. pyATS (core of Cisco's test automation tooling)
3. TextFSM (template-based parsing)
4. TTP
5. JSON
6. `ntc_templates` (TextFSM template collection)

These module types are documented on the Ansible documentation site: Ansible Docs.

2.3 The playbook as the center of execution

Design rule

A playbook should read like a deterministic decision procedure: discover state, decide, converge, validate.

The playbook dictates which modules and variables are needed.

- Ansible provides **roles** to organize playbooks into a known directory structure.
- The only required sub-directory inside a role directory is `tasks`.
- The only required file in `tasks` is `main.yml`, which is the default playbook entrypoint for the role.
- Tasks determine whether other directories are required (e.g., variables, templates, additional task files).

On the Ansible documentation website, roles are defined as follows:

> Roles are ways of automatically loading certain var-files, tasks and handlers, based on a known file structure. Grouping content by roles also allows easy sharing of roles with other users.

Key takeaways

- Playbooks express decision logic; modules encapsulate procedural complexity.
- Determinism requires explicit state discovery and explicit validation.

Chapter 3

Ansible Usage

Learning outcomes

After this section, you can:

- distinguish user roles in Ansible usage (module author, playbook author, playbook operator),
- explain how greenfield and brownfield constraints affect playbook design,
- apply a deterministic module-selection hierarchy in brownfield environments.

Ansible has different levels of usage:

- module writers using a procedural language,
- playbook writers using modules to automate network tasks, and
- users executing playbooks to perform automated network tasks.

For network automation with Ansible, the following use cases are in scope of this document:

- Apply **greenfield** provisioning using dynamic configuration templates.
- Apply **brownfield** provisioning using a mixture of templates and network resource modules.
- Use a single source of truth from which provisioning is pushed (Infrastructure as Code, fully implemented).
- Generate network documentation using network resource modules.

3.1 Provisioning method in brownfield settings

Design rule

In brownfield automation, never act before you can explain the current state in structured terms.

In brownfield provisioning, execution constraints are imposed by the current state of the Managed Node. Complexity emerges because there is no single source of truth. Therefore, device state must be determined before provisioning can be applied predictably.

Brownfield provisioning procedure:

- Query the current state of the Managed Node (hardware, configuration state, running state).
- Use network resource modules to capture relevant information.
- If a network resource module is not available, use `cli_parse` with an appropriate parsing engine (e.g., pyATS) to obtain structured data.
- If no parser is available, use a command module as the last resort and extract the needed values.

This yields the following best practices:

1. Use network resource modules whenever possible.
2. Use `cli_parse` only if no network resource module is available for the task.
3. Use a command module only if no `cli_parse` parsing engine is available for the required output.

In a brownfield situation, we need to know Managed Node state, hardware settings, and specific configuration settings. For example, license status might determine whether further actions are required.

The last-resort action to acquire device configuration data is to use a command module.

Last-resort workflow for command module output

The command module typically returns lists of lists, which can be used as a starting point for selection and decision-making. A practical workflow is:

- Use `register` to store output in a variable.
- Use `debug` to print variable content.
- Inspect the output structure and select the minimum required value(s).

Later in this document, this procedure is shown in a real use case.

Part II

Deterministic Automation

Chapter 4

Use cases for automation

Learning outcomes

After this section, you can:

- map common enterprise automation goals to Ansible playbook patterns,
- recognize how "discover → decide → converge → validate" appears in each use case,
- use the examples as a template for your own production playbooks.

1. Configuration — apply configuration in a brownfield scenario.
2. Reporting — create dynamic reports on configuration and/or running status.
3. Templating — produce device configurations according to company policy and standards.
4. Device Upgrade — apply an operating system upgrade.
5. Compliance — verify device configurations against company policy and standards.

4.1 Use case: Configuration (last-resort module)

Learning outcomes

After this section, you can:

- apply the brownfield module-selection hierarchy in a constrained scenario,
- extract and validate a minimal decision variable from command output,
- gate configuration changes deterministically,
- harden a last-resort workflow for production use.

Design rule

Treat command-module parsing as a last resort: extract the minimum required value, validate the structure explicitly, and gate all changes using that validated state.

Problem statement

Implement on-premise Smart Licensing provisioning in a brownfield environment where:

- no single source of truth exists,
- devices are already operational,
- licensing state must be determined before action.

If the device is already licensed, it must be skipped.

Brownfield decision hierarchy applied

1. Is a network resource module available for Smart Licensing?
   No.
2. Is a supported structured parser available (e.g., pyATS/Genie)?
   No.
3. Therefore: use a command module and extract only the required decision variable.

Last-resort module:

`cisco.ios.ios_command`

Command used:

```
show license summary
```

Phase 1: Discover

Inspect the raw command output to understand its structure.

```
- name: Register output of "show license summary"
ios_command:
commands: show license summary
register: license

- name: Print out the license variable
debug:
var: license
```

Relevant excerpt:

```
"stdout_lines": [
[
"Smart Licensing is ENABLED",
"",
"Registration:",
"  Status: REGISTERED",
...
```

Because only one command is executed:

- `stdout_lines[0]` refers to the first command,
- index `[3]` refers to the status line (zero-based).

Minimal extraction target:

`license.stdout_lines[0][3]`

Phase 2: Decide

Validate structure before trusting positional indices.

```
- name: Register output of show license summary
ios_command:
commands: show license summary
register: license

- name: Assert expected output structure
assert:
that:
- license.stdout_lines is defined
- license.stdout_lines | length > 0
- license.stdout_lines[0] | length > 3
```

```
fail_msg: "Unexpected license output structure"

- name: Extract license status
set_fact:
status: "{{ license.stdout_lines[0][3] }}"

- name: Print out the license status variable
debug:
var: status
```

Example result:

```
ok: [redacted] => {
  "status": "  Status: REGISTERED"
}
```

Key takeaways

- Never gate changes on unvalidated parsing.
- Assert structure before trusting positional indices.
- Extract only what is required for the decision.

Phase 3: Converge

All configuration tasks are gated by:

`when: not status is search("REGISTERED")`

```
- name: Reset call-home and remove default entry
ios_config:
parents: call-home
commands:
- no profile CiscoTAC-1
before: no call-home
when: not status is search("REGISTERED")

- name: Configure on-prem call-home
ios_config:
parents: call-home
commands:
- contact-email-addr {{ license_contact }}
- profile "{{ license_profile }}"
- active
- reporting smart-licensing-data
- destination transport-method http
- no destination transport-method email
- destination address http https://{{ ssm }}/Transportgateway/services/DeviceRequestHandler
when: not status is search("REGISTERED")

- name: Certificate revocation check
ios_config:
parents: crypto pki trustpoint SLA-TrustPoint
commands:
- enrollment terminal
- revocation-check none
when: not status is search("REGISTERED")

- name: Register Switch to SSM
ios_command:
commands:
- license smart register idtoken {{ license_token }} Force
when: not status is search("REGISTERED")
```

Phase 4: Validate

```
- name: Diff running-config against startup-config
cisco.ios.ios_config:
diff_against: startup
diff_ignore_lines:
- ntp clock .*
when: not status is search("REGISTERED")
```

Phase 5: Persist

```
- name: Save running to startup when modified
cisco.ios.ios_config:
save_when: modified
when: not status is search("REGISTERED")
```

Safety characteristics

- Changes are gated by validated device state.
- No configuration is applied without prior structure validation.
- Idempotency is preserved.
- Diff and save operations ensure observability and persistence.

Example analysis

- **What this demonstrates:** disciplined application of the brownfield hierarchy (resource module → parser → command).
- **Why it is safe:** state is discovered, structure validated, and convergence is gated.
- **Primary risk:** CLI output format changes across versions.
- **Production hardening:** replace positional extraction with structured parsing when a supported parser becomes available.

4.2 Use case: Configuration (resource modules and filters)

Learning outcomes

After this section, you can:

- structure a brownfield playbook using deterministic execution phases,
- prefer resource modules and structured parsers over raw CLI parsing,
- separate destructive cleanup from intended convergence,
- implement controlled convergence across heterogeneous platforms.

Design rule

In brownfield provisioning, discover state first, validate structure second, remove explicitly third, and only then converge toward intended configuration.

Problem statement

Provision standardized QoS configuration across remote enterprise switches in a brownfield environment where:

- legacy QoS configuration may already exist,
- ACL dependencies may be embedded in class maps,
- different device models require different templates,

- platform capabilities differ (IOS vs IOS-XE).

This is a classical brownfield scenario: assumptions are unsafe.

Execution model

The playbook is divided into five deterministic phases:

1. **Phase 1 — Discover**
2. **Phase 2 — Decide (cleanup planning)**
3. **Phase 3 — Converge**
4. **Phase 4 — Validate**
5. **Phase 5 — Persist**

Ansible playbook for standardized QoS provisioning:

```
---
# ============================================================
# Phase 1 - Discover
# ============================================================

- name: Gather hardware facts
cisco.ios.ios_facts:
gather_subset: hardware

- name: Gather Layer 2 interfaces
cisco.ios.ios_facts:
gather_subset: min
gather_network_resources: l2_interfaces

- name: Gather policy maps
ignore_errors: yes
ansible.utils.cli_parse:
command: "show policy-map"
parser:
name: ansible.netcommon.pyats
os: iosxe
set_fact: pm

- name: Gather class maps
ignore_errors: yes
ansible.utils.cli_parse:
command: "show class-map"
parser:
name: ansible.netcommon.pyats
os: iosxe
set_fact: cm

- name: Validate parsed structures
assert:
that:
- pm is defined
- cm is defined
fail_msg: "Unable to parse QoS structures reliably"


# ============================================================
# Phase 2 - Decide (explicit cleanup)
# ============================================================

- name: Extract ACL references from class maps
set_fact:
acl_data: "{{ cm.class_maps | json_query('*.index.*.match') }}"
```

```
- name: Extract ACL names
set_fact:
access_groups: "{{ acl_data | json_query('[*][0].access_group[0]') }}"

- name: Inspect ACLs to be removed
cisco.ios.ios_command:
commands:
- "show ip access-list {{ item }}"
loop: "{{ access_groups | default([]) }}"

- name: Remove all policy maps
cisco.ios.ios_config:
lines:
- "no policy-map {{ item }}"
loop: "{{ pm.policy_map | default([]) | list }}"
when: pm.policy_map is defined

- name: Remove all class maps
cisco.ios.ios_config:
lines:
- "no class-map {{ item }}"
loop: "{{ cm.class_maps | default([]) | list }}"
when: cm.class_maps is defined


# ============================================================
# Phase 3 - Converge
# ============================================================

- name: Apply Catalyst 3750 QoS template
cisco.ios.ios_config:
src: ../template/qos-3750-master.cfg
match: line
when: ansible_net_model is search("WS-C3750")

- name: Apply generic QoS template
cisco.ios.ios_config:
src: ../template/qos-generic-master.cfg
match: line
when:
- ansible_net_iostype is search("IOS-XE")
or ansible_net_model is search("WS-C3560")

- name: Apply IOS-XE specific QoS template
cisco.ios.ios_config:
src: ../template/qos-iosxe-master.cfg
match: line
when: ansible_net_iostype is search("IOS-XE")

- name: Attach PM-MARKING to IOS-XE access interfaces
cisco.ios.ios_config:
parents: interface {{ item.name }}
lines:
- service-policy input PM-MARKING
loop: "{{ ansible_network_resources.l2_interfaces }}"
when:
- item.access is defined
- ansible_net_iostype is search("IOS-XE")

- name: Attach PM-QUEUEING to IOS-XE trunk interfaces
cisco.ios.ios_config:
parents: interface {{ item.name }}
lines:
- service-policy output PM-QUEUEING
```

```
loop: "{{ ansible_network_resources.l2_interfaces }}"
when:
- item.trunk is defined
- ansible_net_iostype is search("IOS-XE")

- name: Apply 3750 access interface configuration
cisco.ios.ios_config:
parents: interface {{ item.name }}
lines:
- srr-queue bandwidth share 10 10 60 20
- srr-queue bandwidth shape 10 0 0 0
- priority-queue out
- service-policy input PM-MARKING
loop: "{{ ansible_network_resources.l2_interfaces }}"
when:
- item.mode is defined
- item.mode == "access"
- ansible_net_model is search("WS-C3750")

- name: Apply 3750 trunk interface configuration
cisco.ios.ios_config:
parents: interface {{ item.name }}
lines:
- srr-queue bandwidth share 10 10 60 20
- srr-queue bandwidth shape 10 0 0 0
- queue-set 2
- priority-queue out
- mls qos trust dscp
loop: "{{ ansible_network_resources.l2_interfaces }}"
when:
- item.mode is defined
- item.mode == "trunk"
- ansible_net_model is search("WS-C3750")


# ============================================================
# Phase 4 - Validate
# ============================================================

- name: Diff running-config against startup-config
cisco.ios.ios_config:
diff_against: startup
diff_ignore_lines:
- ntp clock .*


# ============================================================
# Phase 5 - Persist
# ============================================================

- name: Save running configuration if modified
cisco.ios.ios_config:
save_when: modified
```

Safety characteristics

- Structured parsers are preferred over raw CLI parsing.
- Destructive cleanup is explicit and isolated from convergence.
- Platform-specific logic is guarded and deterministic.
- Validation precedes persistence.
- Final diff ensures observability.

Example analysis

- **What this demonstrates:** a full brownfield execution lifecycle using structured discovery and controlled convergence.
- **Why it is safe:** destructive actions are preceded by state discovery and structure validation.
- **Primary risk:** parser variance or incomplete data structures.
- **Mitigation:** assert on parser output and introduce automated validation gates before production rollout.

Key takeaways

- Brownfield provisioning behaves as a controlled state machine.
- Discovery and explicit cleanup precede convergence.
- Deterministic phase separation reduces operational risk in heterogeneous environments.

4.3 Use case: Reporting

Learning outcomes

After this section, you can:

- run a reconnaissance playbook to discover the Ansible facts data model,
- extract a stable set of fields for reporting (guarded against missing keys),
- generate a deterministic CSV artifact per device,
- apply the unified phase model: Discover → Decide → Converge → Validate.

Design rule

In reporting workflows, first discover the facts model, then converge toward a deterministic artifact generated from a guarded, explicit schema.

Problem statement

Generate a CSV report that lists, per access interface with an end host, the connected interface and key interface attributes. This is a **read-only** workflow: it does not modify device state.

Inputs and outputs

Inputs:

- Inventory group: `sw_ios`
- Facts: `ansible_facts.net_neighbors`, `ansible_facts.net_interfaces`

Outputs:

- Per-device CSV artifact: `~/reports/{{ inventory_hostname }}.csv`

Phase 1 — Discover

Reconnaissance is used temporarily to understand the returned data structures. Once the schema is understood, debug tasks can be removed.

Reconnaissance `main.yml`:

```
---
- name: Gather switch interface information (reconnaissance)
cisco.ios.ios_facts:
```

```
available_network_resources: yes
gather_network_resources:
- l2_interfaces
- lldp_interfaces
- vlans

- name: Display variable data type
ansible.builtin.debug:
msg:
- "Data type of 'ansible_facts' is: {{ ansible_facts | type_debug }}"

- name: Print available facts
ansible.builtin.debug:
var: ansible_facts

- name: Print neighbors structure
ansible.builtin.debug:
var: ansible_facts.net_neighbors

- name: Print interfaces structure
ansible.builtin.debug:
var: ansible_facts.net_interfaces
```

Key takeaways

- Reconnaissance establishes the facts schema (keys, nesting, optional fields).
- Once identified, lock the report to a stable column definition.

Phase 2 — Decide

Define explicit decision criteria:

- Treat neighbors without a `platform` key as end hosts.
- Only process interfaces present in both neighbor and interface facts.
- Guard all key access with `is defined` or `default(")`.

```
- name: Gather switch interface information (reporting)
cisco.ios.ios_facts:
gather_subset: min

- name: Assert required facts are present
ansible.builtin.assert:
that:
- ansible_facts.net_neighbors is defined
- ansible_facts.net_interfaces is defined
fail_msg: "Required facts missing: net_neighbors and/or net_interfaces"
```

Phase 3 — Converge (artifact generation)

```
- name: Ensure report directory exists (local)
ansible.builtin.file:
path: "{{ lookup('env','HOME') }}/reports"
state: directory
mode: "0755"
delegate_to: localhost
run_once: true

- name: Build per-device report lines
ansible.builtin.set_fact:
report_lines: "{{ report_lines | default([]) + [ lookup('template','neighbor-report.j2') ] }}"
loop: "{{ ansible_facts.net_neighbors | dict2items }}"
```

```
loop_control:
loop_var: outer
when:
- outer.value is defined
- outer.value | length > 0
- outer.value[0].platform is not defined
- ansible_facts.net_interfaces[outer.key] is defined

- name: Write per-device CSV (local)
ansible.builtin.copy:
dest: "{{ lookup('env','HOME') }}/reports/{{ inventory_hostname }}.csv"
content: |
hostname,local_interface,remote_port,ipv4,bandwidth,operstatus,mediatype,description
{% for line in report_lines | default([]) %}
{{ line }}
{% endfor %}
mode: "0644"
delegate_to: localhost
```

Template: `neighbor-report.j2`

```
{{ inventory_hostname }},
{{ outer.key }},
{{ outer.value[0].port | default('') }},
{{ ansible_facts.net_interfaces[outer.key].ipv4 | default('') }},
{{ ansible_facts.net_interfaces[outer.key].bandwidth | default('') }},
{{ ansible_facts.net_interfaces[outer.key].operstatus | default('') }},
{{ ansible_facts.net_interfaces[outer.key].mediatype | default('') }},
"{{ ansible_facts.net_interfaces[outer.key].description | default('') }}"
```

Phase 4 — Validate

Validation occurs implicitly through:

- assertions on required facts,
- guarded key access using `default(")`,
- deterministic schema (fixed header row + fixed column order).

Example analysis

- **What this demonstrates:** reporting as a deterministic artifact pipeline following the unified lifecycle.
- **Why it is safe:** devices are only read; file writes occur locally.
- **Why it is robust:** schema is explicit and guarded; missing optional fields do not break rendering.
- **Didactic value:** the lifecycle model applies even to read-only automation.

Key takeaways

- Reporting is convergence toward an artifact, not toward device state.
- Guard every assumption about fact structure.
- Make schemas explicit and deterministic.

4.4 Use case: Templating

Learning outcomes

After this section, you can:

- explain why templating is an *offline* and low-risk automation primitive,
- structure variables so templates remain readable and maintainable,
- implement a deterministic render → review → apply workflow,
- anticipate and mitigate schema drift and template sprawl.

Design rule

Templates are strongest when variables are structured, validated, and version-controlled alongside the templates.

Problem statement

Generate complete device configurations (or reusable building blocks) from a standardized template library using a data-driven inventory of devices and profiles. Rendering happens *offline*: output can be inspected and validated before any change reaches the network.

Templating is therefore a low-risk primitive when properly structured.

Phase 1 — Discover

Discovery in templating is not about device state but about the *data model*:

- What variables exist?
- Which are mandatory?
- How are profiles mapped to template families?
- What assumptions do templates make?

The device list forms the foundational schema:

```
devices:
- hostname: egl-post-config
template: egl-c
profile: post

- hostname: redacted
template: egl-c
profile: mk-ls-initial
mgmt_ip: redacted
link1_remote_switch: redacted
link1_remote_port: te1/0/1
link2_remote_switch: redacted
link2_remote_port: te1/0/1
```

Design rule

Treat the device list as a *render contract*: it defines what templates are allowed to assume.

Discovery here establishes the input schema before any rendering occurs.

Phase 2 — Decide

Decide which templates and profiles apply to each device and validate that required keys exist.

Minimal validation example:

```
- name: Validate render contract
ansible.builtin.assert:
that:
- item.hostname is defined
- item.template is defined
- item.profile is defined
loop: "{{ devices }}"
```

Decision criteria include:

- Template family selection (`egl-c`)
- Profile selection (`post`, `mk-ls-initial`, etc.)
- Optional per-device overrides

At this point, assumptions are explicit and validated.

Phase 3 — Converge (offline rendering)

Rendering is convergence toward a deterministic artifact.

```
---
- name: Generate configuration files (offline rendering)
ansible.builtin.template:
src: "{{ item.template }}/{{ item.profile }}.j2"
dest: "~/out-basket/{{ item.hostname }}_{{ item.profile }}.cfg"
mode: "0644"
loop: "{{ devices }}"
```

This produces candidate configuration files without modifying device state.

Key takeaways

- Prefer `loop` over `with_items`.
- Determinism depends on stable input variables.

Phase 4 — Validate

Validation in templating means reviewing rendered output before applying it.

This may include:

- Diffing against previous rendered baselines.
- Comparing against intended-state repositories.
- Reviewing changes through peer review or CAB processes.

Didactic hardening practices:

- **Golden outputs:** store representative rendered baselines and diff in CI.
- **Schema validation:** assert required keys before rendering.
- **Template boundaries:** compose smaller templates using `{% include %}`.
- **Review discipline:** treat rendered diffs as the change artifact.

Validation ensures that convergence is correct before any deployment occurs.

Phase 5 — Persist (optional apply)

Applying rendered configurations to devices should occur in a separate, controlled workflow.

This preserves separation of concerns:

- Rendering remains low risk and repeatable.
- Application is gated and auditable.
- Change control processes remain intact.

The classical didactic pattern:

$$\text{Render} \rightarrow \text{Review} \rightarrow \text{Apply}$$

is therefore an instance of the unified lifecycle:

$$\text{Discover} \rightarrow \text{Decide} \rightarrow \text{Converge} \rightarrow \text{Validate} \rightarrow \text{Persist}$$

Example analysis

- **What this demonstrates:** scalable configuration generation via a data-driven render contract.
- **Why it is safe:** rendering is offline; validation precedes application.
- **Primary risks:** missing variables, inconsistent defaults, uncontrolled template growth.
- **Production hardening:** CI validation, golden baselines, explicit input schema enforcement.

Key takeaways

- Templating is convergence toward an artifact, not directly toward device state.
- The stability of the data model determines the stability of the rendered output.
- Separation of render and apply reduces operational risk.

4.5 Use case: Device upgrade

Learning outcomes

After this section, you can:

- model device upgrades as deterministic state machines,
- implement integrity and boot-variable validation gates,
- separate image preparation from reload orchestration,
- harden upgrade workflows for enterprise environments.

Design rule

Upgrades are controlled state machines: never copy images, modify boot variables, or reload without explicit validation gates.

Problem statement

Upgrade enterprise access switches to a specific IOS version while ensuring:

- only supported device models are affected,
- sufficient flash space exists,
- image integrity is verified,
- the boot variable is correct,
- reload is controlled and verified,
- post-reload compliance is validated.

This is a high-risk workflow. Deterministic gating is mandatory.

Upgrade state machine

1. Is the device model supported? If not, stop.
2. Is the device already compliant? If yes, stop.
3. Is the target image present on flash? If no, copy it.
4. Is the MD5 checksum valid? If no, stop.
5. Is the boot variable correct? If no, update it.
6. Is reload required? If yes, reload in a controlled manner.
7. After reload, re-verify version compliance.

Phase 1 — Discover

```
---
- hosts: enterprise_l2
gather_facts: false
connection: local

vars:
models:
"WS-C2960XR-48FPD-I":
ios_version: "15.2(7)E0a"
ios_path: "IOS/"
ios_archive: "c2960x-universalk9-tar.152-7.E0a.tar"
ios_binary: "c2960x-universalk9-mz.152-7.E0a.bin"
ios_md5: "07195a8c7866ac2a91c64b716465c516"
ios_size_kb: 37489
server: "1.1.1.1"
protocol: "http"

tasks:

- name: Gather hardware facts
cisco.ios.ios_facts:
gather_subset: hardware

- name: Assert device model is supported
assert:
that:
- models[ansible_net_model] is defined
fail_msg: "Unsupported device model"

- name: Check current boot variable
cisco.ios.ios_command:
commands:
- "show boot | include BOOT"
register: bootvar

- name: Check if image is present on flash
cisco.ios.ios_command:
commands:
- "show flash: | include {{ models[ansible_net_model]['ios_archive'] }}"
register: dir_flash
```

Phase 2 — Decide

```
- name: Stop if already compliant
meta: end_play
when:
- ansible_net_version == models[ansible_net_model]['ios_version']

- name: Ensure sufficient flash space if copy required
assert:
that:
```

```
- ansible_net_filesystems_info['flash:']['spacefree_kb']
> models[ansible_net_model]['ios_size_kb']
fail_msg: "Insufficient flash space"
when:
- models[ansible_net_model]['ios_archive']
not in dir_flash.stdout[0]
```

Phase 3 — Converge

```
- name: Copy image if not present
cisco.ios.ios_command:
commands:
- command: >
copy {{ models[ansible_net_model]['protocol'] }}://
{{ models[ansible_net_model]['server'] }}/
{{ models[ansible_net_model]['ios_path'] }}
{{ models[ansible_net_model]['ios_archive'] }}
flash:/
prompt: "Destination filename \\[{{ models[ansible_net_model]['ios_archive'] }}\\]?"
answer: "\r"
vars:
ansible_command_timeout: 1800
when:
- models[ansible_net_model]['ios_archive']
not in dir_flash.stdout[0]

- name: Set boot variable if incorrect
cisco.ios.ios_config:
lines:
- "boot system flash:{{ models[ansible_net_model]['ios_binary'] }}"
when:
- models[ansible_net_model]['ios_binary']
not in bootvar.stdout[0]

- name: Reload device if upgrade required
cisco.ios.ios_command:
commands:
- command: reload
prompt: "Proceed with reload? \\[confirm\\]"
answer: "\r"
when:
- ansible_net_version != models[ansible_net_model]['ios_version']
```

Phase 4 — Validate

```
- name: Wait for device to return
wait_for:
host: "{{ ansible_host }}"
port: 22
delay: 180
timeout: 900
delegate_to: localhost

- name: Gather facts after reload
cisco.ios.ios_facts:
gather_subset: hardware

- name: Verify MD5 checksum
cisco.ios.ios_command:
commands:
- command: >
verify /md5 flash:
{{ models[ansible_net_model]['ios_archive'] }}
```

```
register: md5_result
vars:
ansible_command_timeout: 600

- name: Assert MD5 integrity
assert:
that:
- models[ansible_net_model]['ios_md5']
in md5_result.stdout[0]
fail_msg: "MD5 checksum mismatch"

- name: Assert version compliance
assert:
that:
- ansible_net_version
== models[ansible_net_model]['ios_version']
fail_msg: "Upgrade verification failed"
```

Phase 5 — Persist

```
- name: Save running configuration if modified
cisco.ios.ios_config:
save_when: modified
```

Safety characteristics

- Unsupported platforms are excluded during discovery.
- Compliance short-circuits unnecessary upgrades.
- Flash capacity is validated before copy.
- MD5 integrity prevents corrupt image activation.
- Boot variable correctness is enforced before reload.
- Post-reload validation confirms successful convergence.

Example analysis

- **What this demonstrates:** upgrade as a deterministic state machine following the full lifecycle.
- **Why it is safe:** each destructive step is gated by explicit validation.
- **Primary risks:** flash capacity errors, MD5 mismatch, reload timing variance.
- **Production hardening:** maintenance window gating, centralized logging, rollback image retention.

Key takeaways

- Upgrades are state transitions, not task sequences.
- Validation gates must precede every destructive action.
- Post-reload verification is mandatory.

4.6 Use case: Compliance

Learning outcomes

After this section, you can:

- express compliance as a deterministic comparison against intended state,
- perform audit-only compliance checks,
- extend compliance into controlled convergence,
- harden compliance workflows for enterprise environments.

Design rule

Compliance follows the deterministic lifecycle: discover observed state, decide drift via explicit comparison, converge only when required, validate results, and persist audit evidence.

Problem statement

Verify that device configuration matches company standards.

The intended configuration is stored in version-controlled templates. The device configuration represents the observed state.

Compliance = **Intended State − Observed State**

Phase 1 — Discover

Retrieve the current running configuration from the device.

In practice, this is performed implicitly by the `ios_config` module when used with `diff_against: intended`.

Phase 2 — Decide

Perform a deterministic comparison between observed state and intended configuration.

Minimal audit-only compliance

The simplest compliance check requires one task:

```
---
- name: Perform compliance check against master configuration
cisco.ios.ios_config:
diff_against: intended
intended_config: "{{ lookup('file', '../template/compliance-master.cfg') }}"
```

This produces a structured diff without modifying device state.

Key takeaways

- The intended configuration must be version-controlled.
- The diff output is the decision artifact.
- No convergence occurs unless explicitly requested.

Phase 3 — Converge (optional)

If drift should be corrected automatically, shift from audit mode to enforcement mode:

```
- name: Enforce compliance against intended configuration
cisco.ios.ios_config:
src: "../template/{{ compliance_template }}"
match: line
when:
- ansible_net_model is defined
- compliance_template is defined
```

Design rule

Separate audit mode and enforce mode explicitly. Do not mix both implicitly in a single task.

Phase 4 — Validate

Validation confirms whether:

- the device is compliant after enforcement, or
- drift remains and requires further investigation.

Re-running the diff task verifies convergence.

Phase 5 — Persist

Persist audit evidence and ensure traceability.

Guarded compliance by device type

In heterogeneous environments, compliance templates differ per platform or role:

```
- name: Perform compliance check (device-type specific)
cisco.ios.ios_config:
diff_against: intended
intended_config: "{{ lookup('file', '../template/' ~ compliance_template) }}"
when:
- ansible_net_model is defined
- compliance_template is defined
```

Example variable definition:

```
compliance_template: "compliance-{{ ansible_net_model }}.cfg"
```

This ensures:

- compliance intent matches platform capability,
- unsupported devices are not evaluated incorrectly,
- audit results remain deterministic.

Production hardening considerations

- Store diffs centrally for audit trails.
- Separate read-only audit playbooks from enforcement playbooks.
- Guard by device role, model, and site.
- Validate templates in CI before deployment.
- Avoid enforcing compliance during maintenance windows without explicit approval.

Example analysis

- **What this demonstrates:** compliance as a deterministic lifecycle aligned with all other automation workflows.
- **Why it is safe:** audit mode does not alter state; enforcement is explicit and controlled.
- **Primary risk:** intended template not aligned with device capability.
- **Mitigation:** platform-aware template selection and pre-validation in CI.

Key takeaways

- Compliance is intent comparison, not pattern matching.
- Deterministic diffs enable both auditing and controlled convergence.
- Production-grade compliance requires version control, device scoping, and explicit enforcement boundaries.

Chapter 5

Common anti-patterns in network automation

Common anti-patterns

- Parsing text with ad-hoc regex when structured data exists.
- Making changes without discovering state first in brownfield environments.
- Omitting post-change validation (diffs, assertions, compliance checks).
- Performing reloads/upgrades without boot verification and integrity checks.
- Mixing intent (desired state) and imperative steps without explicit boundaries.

Chapter 6

Synthesis

Across all use cases in this guide, a single execution grammar appears repeatedly:

$$\text{Discover} \rightarrow \text{Decide} \rightarrow \text{Converge} \rightarrow \text{Validate} \rightarrow \text{Persist}$$

This sequence is not stylistic. It is structural.

It applies to:

- Configuration (last-resort modules),
- Configuration (resource modules and structured parsers),
- Reporting (artifact convergence),
- Templating (offline render contracts),
- Device upgrades (state machines),
- Compliance (intent comparison and controlled enforcement).

Each use case is an instance of the same deterministic lifecycle.

The Unified Lifecycle

Phase 1 — Discover Make implicit state explicit. Gather device facts, retrieve running configuration, or inspect input schemas. No assumptions are permitted.

Phase 2 — Decide Convert raw state into validated decision variables. Assert structure before trust. Diff observed state against intended state. Determine whether action is required.

Phase 3 — Converge Apply bounded change — or converge toward a deterministic artifact. Convergence may modify device state, generate reports, render templates, or prepare upgrade images.

Phase 4 — Validate Confirm that intended and observed state now align. Re-run diffs. Verify integrity checks. Confirm version compliance. Ensure artifact correctness.

Phase 5 — Persist Store configuration, publish artifacts, log audit evidence, or separate enforcement workflows from audit workflows. Persistence ensures durability and traceability.

Brownfield Discipline

In enterprise environments, most automation is brownfield. Therefore, module selection follows a strict hierarchy:

1. Prefer network resource modules.
2. Use structured parsers when necessary.
3. Use command modules only as a validated last resort.

This hierarchy minimizes ambiguity and reduces operational risk.

Automation quality is determined less by syntax and more by how explicitly decisions are bounded.

State Machines, Not Scripts

Configuration is controlled convergence. Upgrades are guarded state transitions. Compliance is intent comparison. Templating is artifact convergence. Reporting is schema-driven extraction.

In every case:

- State is discovered before action.
- Structure is validated before trust.
- Destructive operations are gated.
- Verification follows convergence.
- Evidence is persisted.

Automation fails when phases collapse into one another. It succeeds when phases remain explicit.

Production-Grade Characteristics

Enterprise-ready automation exhibits the following properties:

- Deterministic behavior
- Idempotency
- Explicit validation gates
- Observable diffs and artifacts
- Version-controlled intent
- Clear separation of audit and enforcement

Tools do not create reliability. Structured decision boundaries do.

Final Principle

Automation is not about executing commands faster.

It is about reducing uncertainty.

When the lifecycle remains explicit and the decision hierarchy is respected, automation becomes a controlled system rather than a collection of scripts.

That discipline—not syntax—is what makes network automation reliable.

Part III

Intent-Driven Systems

Chapter 7

Next-step Project: Intent-Driven Fabric Automation

Learning outcomes

After this section, you can:

- explain why intent-driven automation requires a normalized internal data model,
- distinguish Intent (inputs) from Derived Model (computed truth) and Renderers (outputs),
- structure a cross-platform VXLAN/EVPN workflow for NX-OS and IOS-XE,
- describe a repeatable lifecycle: Discover → Decide → Produce → Validate.

Design rule

Treat intent as a contract: validate inputs strictly, compile them into a derived model, and render outputs deterministically from that derived model.

Motivation and scope

The v1.0 guide shows robust playbook patterns per use case. The next level is to systematize structured network automation by introducing a reusable **intent compiler**:

- Inputs are expressed as intent (site/fabric parameters).
- A compiler produces a normalized **derived model** (meaningful key:value hierarchy).
- Renderers produce configuration, documentation, and reports from the same derived model.

This section proposes a project that demonstrates the pattern on a concrete domain: **VXLAN/EVPN fabric configuration for NX-OS and IOS-XE**.

Target outcome

v1.1 (project milestone):

- A validated intent schema for VXLAN/EVPN fabrics.
- A derived model that normalizes platform differences.
- Deterministic configuration rendering using best-practice templates.
- Deterministic reporting output (e.g., Excel) generated from the same derived model.

v1.2+ (extension):

- State discovery + compliance: compile *observed state* into the same derived model shape.

- Drift detection and controlled convergence.
- Consolidated multi-device documentation artifacts.

Architecture: Intent → Derived Model → Outputs

Core components:

- **Intent schema (input contract):** declarative YAML (validated).
- **Compiler (Python):** transforms intent into a derived model.
- **Renderers:** Jinja2 templates per platform and role (leaf/spine/superspine).
- **Drivers (Ansible integration):** run the compiler and apply rendered configs.
- **Artifacts:** configuration files + documentation + Excel reporting.

Design rule

The derived model is the single internal truth. Templates and reports consume the derived model, not raw intent.

Unified lifecycle phases

This project uses the same didactic lifecycle as the use cases in this guide:

1. **Discover:** (optional at first) gather platform facts and capabilities; later gather observed state.
2. **Decide:** validate intent and compile derived model; compute per-device roles and parameters.
3. **Produce:** render device configurations and reporting artifacts deterministically.
4. **Validate:** schema validation, template rendering checks, diff/compliance gates before apply.

Intent schema (inputs)

A VXLAN/EVPN module needs sufficient intent to compile a full fabric configuration.

Example intent categories:

- **Fabric topology:** clos-3 vs clos-5, device roles (leaf/spine/superspine).
- **Platforms:** NX-OS vs IOS-XE, device model families for role mapping.
- **Addressing intent:** loopback subnet pools, underlay addressing strategy, anycast gateway intent.
- **Tenant/VLAN intent:** VLANs, names/descriptions, VNIs, tenant grouping.
- **Traffic profile:** campus north-south vs data center east-west (policy and design choices).

Derived model (normalized key:value hierarchy)

The derived model should normalize cross-platform differences into a stable structure, e.g.:

- `fabric.topology` (clos-3/clos-5)
- `devices[hostname].role` (leaf/spine/superspine)
- `devices[hostname].platform` (nxos/iosxe)
- `devices[hostname].interfaces.underlay[]` (peer links, addressing, MTU, etc.)
- `tenants[].vlans[]` (id, name, vni, gateway, policy)
- `features.evpn / features.vxlan` (capability flags + parameters)

Design rule

If a renderer needs a value, it must exist in the derived model. If it does not exist, compilation should fail fast.

Implementation milestones

1. **Schema:** define a minimal intent schema; validate with strict rules.
2. **Compiler:** generate derived model for one topology (clos-3) and one platform (NX-OS).
3. **Render:** produce deterministic configs from templates (per role).
4. **Ansible integration:** apply rendered configs via a role (then migrate to a custom module if desired).
5. **Reporting:** generate an Excel report from the derived model (inventory, VLAN/VNI mapping, interface plan).
6. **Cross-platform:** extend derived model + templates to IOS-XE.
7. **Validation gates:** add diff/compliance and "no-op if compliant" behavior.

Success criteria and risks

Success means:

- deterministic output (same intent → same configs/reports),
- explicit failure on invalid or incomplete intent,
- safe application boundaries (diffs, checks, and guarded apply),
- cross-platform capability variance handled in the derived model, not in ad-hoc playbook logic.

Primary risks:

- platform feature variance and syntax drift (NX-OS vs IOS-XE),
- hidden assumptions in templates,
- schema drift as the intent grows.

Key takeaways

- The derived model is the keystone: it enables config rendering and reporting from one truth.
- Ansible is the execution plane; Python is the compilation plane.
- Cross-platform automation is achieved by normalization, not by piling conditional logic into playbooks.