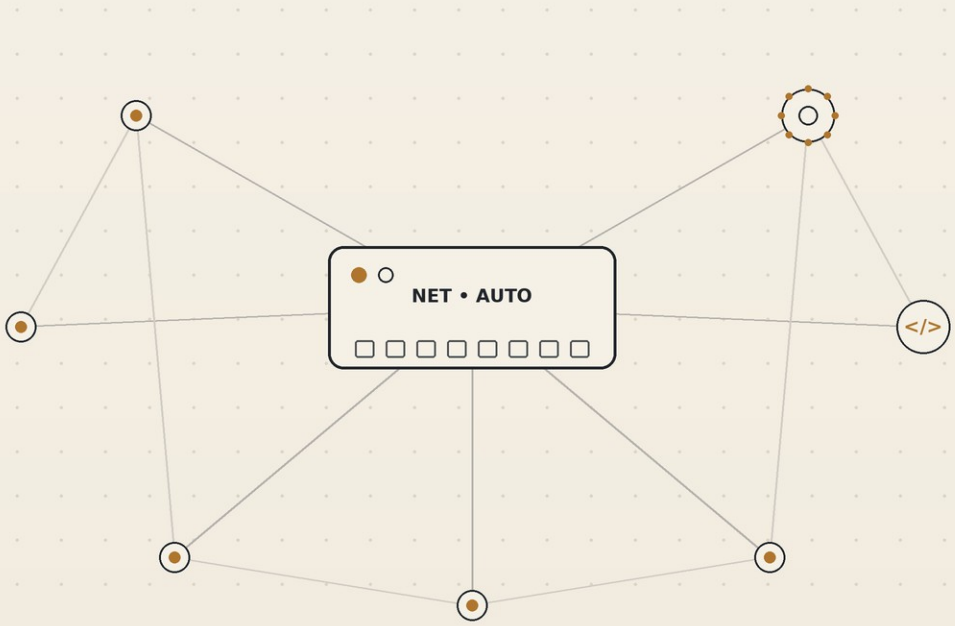
CCNP & CCIE AUTOMATION



AUTOCOR 350-901

Complete Learning Guide

Designing, Deploying and Managing
Network Automation Systems v2.0

DOMAINS 1.0 – 4.0 • 240 PRACTICE QUESTIONS

Jozef Baroš

AUTHOR

About This Book

How to use this guide

This guide is a complete, self-contained preparation resource for the **Cisco AUTOCOR 350-901** exam — *Designing, Deploying and Managing Network Automation Systems v2.0*, associated with the **CCNP** and **CCIE Automation** certifications. It covers all four exam domains in depth, in one volume.

The book is organized into four chapters, one per exam domain. Every sub-section follows the same rhythm: a beginner-friendly explanation, worked examples with real, runnable code, a **Summary**, a set of **Key Takeaways**, and a **Knowledge Check** of ten Cisco-style multiple-choice questions with fully explained answers — **240 practice questions** in total.

Read each domain in order, work through the code on a lab or in **CML**, and use the Knowledge Checks to confirm mastery before moving on. The **Official Cisco Exam Blueprint** follows this page so you can map every topic you study back to the exam objectives.

Core technologies covered: **Ansible**, **Terraform**, **RESTCONF/YANG**, **Python** (Netmiko, NAPALM, ncclient), **Git**, **GitLab CI/CD**, **Cisco Modeling Labs**, **Docker Compose**, **NetBox/Nautobot**, **model-driven telemetry**, **pyATS**, **TLS/PKI**, secure coding, and **AI/LLM** automation with **MCP** and **FastMCP**.

Contents

Official Cisco Exam Blueprint

Designing, Deploying and Managing Network Automation Systems v2.0 (350-901)

Exam Description. AUTOCOR 350-901 is a **120-minute** exam associated with the **CCNP** and **CCIE Automation** certifications. It certifies a candidate's knowledge of network automation systems development and design, including infrastructure as code, operations, and AI in automation. Technologies covered include **Cisco IOS XE, Cisco ACI, Cisco Meraki, Cisco Catalyst Center, Cisco SD-WAN, Cisco Identity Services Engine, and Webex Messaging**.

The following topics are general guidelines for the content likely to be included on the exam. Other related topics may also appear on any specific delivery of the exam. The four domains and their weightings are:

30% 1.0 Network Automation

1.1 Construct a network automation solution with Ansible to manage configurations such as VLANs, OSPF, asset management, interface settings, and ACLs

1.2 Construct a network automation solution with Terraform to manage configurations such as VLANs, OSPF, asset management, interface settings, and ACLs

1.3 Construct a network automation solution with RESTCONF (RFC 8040), given the YANG model, to manage configurations such as VLANs, OSPF, asset management, interface settings, and ACLs

1.4 Construct a network automation solution with Python to manage configurations such as VLANs, OSPF, asset management, interface settings, and ACLs

1.5 Select the network automation approach to achieve technical and business requirements considering options such as infrastructure as code framework, low code/no code, and custom applications

1.6 Construct a network automation solution that consumes REST APIs including extended API attributes (such as pagination, complex authentication workflows, and rate limiting), error handling, and persistent authentication

30% 2.0 Infrastructure as Code

2.1 Use version control operations with Git

- Merge a branch including squash and conflict resolution
- git cherry-pick
- git reset
- git checkout
- git revert

2.2 Diagnose a GitLab CE CI/CD pipeline failure such as missing dependency, incompatible versions of components, and failed tests

2.3 Construct a GitLab CE CI/CD pipeline to deploy a network automation solution including stages for: build, prevalidation, deploy, and post-validation

2.4 Construct a network simulation with Cisco Modeling Labs (CML) to test the network automation solution

2.5 Interpret a Docker Compose file including services, networks, volumes, and links

2.6 Integrate source of truth into a network automation solution

2.7 Construct a YAML or JSON representation of a network configuration given a YANG-based data model

20% 3.0 Operations

3.1 Describe architectural components of model-driven telemetry

3.2 Implement a logging strategy for a network automation solution targeting destinations such as syslog or webhooks

3.3 Diagnose problems with network automation given logs and output related to an event

3.4 Implement change validation for a network automation solution using pyATS CLI tools

3.5 Describe the process to obtain and deploy CA-signed TLS certificates

3.6 Implement secure coding practices into a network automation solution to meet input validation, authentication, and secret management requirements

20% 4.0 AI in Automation

4.1 Describe the benefits and risks of AI-assisted code development for network automation such as data privacy, IP ownership, and code validation

4.2 Interpret the security risks in a given AI-based network automation solution

4.3 Construct an MCP server to provide network information to an AI-agent using Python FastMCP

4.4 Construct a conversational agent that leverages LLMs for network automation

4.5 Evaluate the accuracy of AI recommendations on a network automation solution

1.0 Network Automation — Domain Overview

Network automation is the practice of using software — instead of an engineer typing commands into a terminal — to configure, validate, and operate network devices. On the AUTOCOR 350-901 exam, Domain 1.0 is worth **30% of your score**, more than any other domain, so it deserves the most preparation time.

The Cisco blueprint expects you to build the *same* network changes — VLANs, OSPF, interface settings, ACLs, and asset/inventory management — using **four different tools**: Ansible, Terraform, RESTCONF, and Python. This is deliberate. The exam wants you to understand that there is no single "right" tool; each has a sweet spot, and a real automation engineer chooses based on technical and business requirements (which is exactly what sub-section 1.5 tests).

Two mental models will carry you through the whole domain. The first is **imperative vs. declarative**. Imperative automation describes the *steps* ("log in, enter config mode, type these commands"). Declarative automation describes the *desired end state* ("VLAN 10 named SALES should exist") and lets the tool work out the steps. The second is **idempotency**: running the same automation twice should leave the device in the same state and not pile up duplicate or conflicting configuration. Almost every exam question is, at heart, testing whether you understand these two ideas.

> **Exam strategy**
>
> When a question gives you a scenario, first identify the tool, then ask "is this describing the desired state or the steps?" and "would running this twice be safe?" Those two questions resolve a large share of Domain 1.0 items.

Here is how the six sub-sections relate to one another. Sections 1.1 to 1.4 each teach one tool end-to-end. Section 1.5 steps back and asks you to *choose* between them for a given requirement. Section 1.6 zooms into the hardest part of real API automation — authentication, pagination, rate limiting, and error handling — because almost every modern Cisco controller (Catalyst Center, Meraki, ISE, SD-WAN vManage) is driven through a REST API.

1.1 Network Automation with Ansible

Ansible is an open-source automation engine maintained by Red Hat. It is the most common first step into network automation because it is **agentless** (nothing is installed on the routers and switches) and because automation is written in **YAML**, which is far easier to read than a programming language. You describe what you want in a file called a **playbook**, run it from a **control node** (your laptop or a server), and Ansible connects to each device and makes the change.

How Ansible connects to network devices

On servers, Ansible normally connects over SSH and runs Python on the remote machine. Network devices usually cannot run Python locally, so Ansible uses special **connection plugins** that run the module *on the control node* and only push CLI or API calls to the device. The three you must know are:

- `network_cli` — screen-scrapes the device CLI over SSH. Used for traditional IOS, IOS XE, NX-OS, and IOS configuration.
- `netconf` — speaks NETCONF/XML to YANG-capable devices.
- `httpapi` — speaks to REST/HTTP-based platforms (for example NX-OS NX-API or Meraki Dashboard).

You select the plugin with the variable `ansible_connection` and tell Ansible which OS family it is talking to with `ansible_network_os` (for IOS XE this is `cisco.ios.ios`).

Inventory: telling Ansible what to manage

The **inventory** lists your devices and groups them. Variables can be attached per host or per group. A small INI-style inventory looks like this:

inventory.ini — two IOS XE devices in a group

```
# inventory.ini
[ios_xe]
csr1   ansible_host=192.0.2.11
csr2   ansible_host=192.0.2.12

[ios_xe:vars]
ansible_connection=ansible.netcommon.network_cli
ansible_network_os=cisco.ios.ios
ansible_user=automation
ansible_password={{ vault_device_password }}
```

Notice the password is a **variable**, not a clear-text secret. Real secrets live in an encrypted **Ansible Vault** file (covered below). The exam likes to test that you never hard-code passwords in playbooks or inventory.

Collections and resource modules

Modern Ansible ships networking content in **collections**. For Cisco IOS XE the key collection is `cisco.ios` and the shared plumbing lives in `ansible.netcommon`. Within a collection, the most important tools are the **resource modules** — purpose-built, fully idempotent modules for a

single feature, such as `ios_vlans`, `ios_l2_interfaces`, `ios_ospfv2`, `ios_interfaces`, and `ios_acls`. There is also a generic `ios_config` module that pushes raw CLI when no resource module exists.

> **Resource module states**
>
> Every resource module accepts a `state` key that controls how it reconciles configuration: **merged** (add/update only — the safe default), **replaced** (make this resource match exactly, per-item), **overridden** (make the whole feature match exactly — removes anything not listed), **deleted** (remove), **gathered** (read current state as structured data), and **rendered/parsed** (convert between structured data and CLI offline). Knowing what **overridden** does — it deletes config you did not mention — is a classic exam trap.

Example 1 — Managing VLANs

This playbook ensures three VLANs exist on every device in the `ios_xe` group. Because `state` is `merged`, it adds these VLANs without touching others, and re-running it changes nothing (idempotent).

vlans.yml

```
---
- name: Configure access VLANs
  hosts: ios_xe
  gather_facts: false
  tasks:
    - name: Ensure VLANs are present
      cisco.ios.ios_vlans:
        config:
          - name: SALES
            vlan_id: 10
          - name: VOICE
            vlan_id: 20
          - name: MGMT
            vlan_id: 99
        state: merged
```

Example 2 — Interface settings and an L2 access port

interfaces.yml

```
---
- name: Configure interfaces
  hosts: ios_xe
  gather_facts: false
  tasks:
    - name: Set description and enable interface
      cisco.ios.ios_interfaces:
        config:
          - name: GigabitEthernet2
            description: Uplink to core
```

```
            enabled: true
        state: merged

    - name: Make Gi3 an access port in VLAN 10
      cisco.ios.ios_l2_interfaces:
        config:
          - name: GigabitEthernet3
            access:
              vlan: 10
        state: merged
```

Example 3 — OSPF

ospf.yml

```
---
- name: Configure OSPF
  hosts: ios_xe
  gather_facts: false
  tasks:
    - name: Enable OSPF process 1, area 0
      cisco.ios.ios_ospfv2:
        config:
          processes:
            - process_id: 1
              router_id: 1.1.1.1
              areas:
                - area_id: '0'
              networks:
                - address: 10.0.0.0
                  wildcard_bits: 0.0.0.255
                  area: '0'
        state: merged
```

Example 4 — An extended ACL

acls.yml

```
---
- name: Configure ACLs
  hosts: ios_xe
  gather_facts: false
  tasks:
    - name: Permit web, deny the rest
      cisco.ios.ios_acls:
        config:
          - afi: ipv4
            acls:
              - name: WEB_FILTER
                acl_type: extended
                aces:
```

```
                  - sequence: 10
                    grant: permit
                    protocol: tcp
                    source:
                      any: true
                    destination:
                      any: true
                      port_protocol:
                        eq: www
        state: merged
```

Asset management with facts

"Asset management" on the blueprint means collecting inventory data — model, serial number, software version, interfaces — as structured data you can store or report on. Resource modules in `gathered` state, or the `ios_facts` module, return facts you can save:

facts.yml

```
---
- name: Collect device inventory
  hosts: ios_xe
  gather_facts: false
  tasks:
    - name: Gather IOS facts
      cisco.ios.ios_facts:
        gather_subset: all

    - name: Save inventory to a file per host
      ansible.builtin.copy:
        content: "{{ ansible_net_serialnum }},{{ ansible_net_version }},
{{ ansible_net_model }}"
        dest: "inventory/{{ inventory_hostname }}.csv"
      delegate_to: localhost
```

Variables, templates, and secrets

Reusable playbooks separate **data** from **logic**. Put per-group data in `group_vars/` and per-host data in `host_vars/`, then reference it. For repetitive config you can render a **Jinja2** template. For secrets, encrypt a vars file with `ansible-vault encrypt secrets.yml` and supply the password at run time with `--ask-vault-pass`. The encrypted file is safe to commit to Git.

Idempotency and dry runs

Run any playbook with `--check` (dry run) and `--diff` (show the exact changes) before applying. A correct resource-module playbook reports `changed=0` on the second run — proof that it is idempotent.

Summary

Ansible is an **agentless**, YAML-driven automation engine. You write **playbooks** that run from a **control node** and push changes to devices using a connection plugin (`network_cli`, `netconf`, or `httpapi`).

Cisco IOS XE content lives in the `cisco.ios` collection. **Resource modules** (`ios_vlans`, `ios_ospfv2`, `ios_l2_interfaces`, `ios_acls`, `ios_interfaces`) are idempotent and use a `state` key (merged, replaced, overridden, deleted, gathered).

Keep secrets in **Ansible Vault**, separate data into `group_vars`/`host_vars`, and verify safety with `--check` and `--diff`.

Key Takeaways

- Ansible is **agentless** and uses **SSH/API from the control node** — nothing is installed on the device.
- Playbooks are **YAML**; the unit of work is a **task** calling a **module**.
- Prefer **resource modules** over raw `ios_config` — they are idempotent and return structured state.
- The `state` key is critical: **merged** adds, **overridden** removes anything not listed.
- Never hard-code secrets — use **Ansible Vault**; verify changes with **--check / --diff**.

Knowledge Check — 1.1 Ansible

Q1. Why is Ansible described as "agentless" for network automation?

A. It does not require a license to run on Cisco devices
B. It automatically discovers devices without an inventory
C. No software agent is installed on the managed devices; modules run from the control node and push CLI/API calls
D. It runs entirely in the cloud with no local installation

Correct answer: C. Agentless means there is no persistent Ansible process on the routers/switches. The control node executes the module logic and connects over SSH (network_cli), NETCONF, or HTTP-API to apply changes. Licensing, auto-discovery, and cloud hosting are unrelated to the term.

Q2. An engineer must add VLAN 50 to a switch without removing any existing VLANs. Which value of `state` should be used with `cisco.ios.ios_vlans`?

A. merged
B. overridden
C. replaced
D. deleted

Correct answer: A. `merged` adds or updates only the items you specify and leaves everything else intact. `overridden` would delete every VLAN not listed, `replaced` reconciles per-item but is unnecessary here, and `deleted` removes configuration.

Q3. Which connection plugin is appropriate for managing a traditional Cisco IOS XE device over its CLI?

- **A.** ansible.netcommon.httpapi
- **B.** ansible.builtin.ssh
- **C.** ansible.netcommon.netconf
- **D.** ansible.netcommon.network_cli

Correct answer: D. `network_cli` screen-scrapes the device CLI over SSH and is the standard choice for IOS/IOS XE CLI automation. `httpapi` is for REST platforms, `netconf` for NETCONF/YANG, and `ssh` is the generic Linux connection, not a network plugin.

Q4. A playbook is run twice. After the second run Ansible reports `changed=0`. What does this demonstrate?

- **A.** The playbook failed silently
- **B.** The playbook is idempotent — the desired state already matches
- **C.** The device rejected the configuration
- **D.** Ansible cached the result and skipped the device

Correct answer: B. `changed=0` on a repeat run means the device already matches the declared state, so no change was needed. This is the definition of idempotency and a key reason to prefer resource modules.

Q5. Where should device passwords be stored so they can be safely committed to a Git repository?

- **A.** In an Ansible Vault-encrypted variables file
- **B.** As plain text in the inventory file
- **C.** In a comment inside the playbook
- **D.** In the `ansible.cfg` defaults section

Correct answer: A. **Ansible Vault** encrypts the file at rest; the password is supplied at run time. The encrypted file can be committed safely. Plain text in inventory, comments, or `ansible.cfg` would expose the secret.

Q6. What is the effect of running `state: overridden` with the `ios_acls` module while supplying only ACL WEB_FILTER?

- **A.** It merges WEB_FILTER with existing ACLs
- **B.** It deletes only WEB_FILTER
- **C.** It makes the IPv4 ACL configuration match exactly, removing any ACL not listed
- **D.** It returns the current ACLs as structured data

Correct answer: C. `overridden` forces the entire feature (here, IPv4 ACLs) to match the supplied configuration, **removing any ACL you did not include**. This is powerful but dangerous, and a frequent exam trap versus `merged`.

Q7. Which flags let you preview the exact changes a playbook would make without applying them?

- **A.** `--list-tasks` and `--syntax-check`
- **B.** `--check` and `--diff`
- **C.** `--verbose` and `--step`
- **D.** `--start-at-task` and `--tags`

Correct answer: B. `--check` performs a dry run and `--diff` shows the line-by-line difference that would be applied. The other flags affect listing, verbosity, or task selection, not change preview.

Q8. For "asset management" — collecting model, serial, and software version as structured data — which approach fits best?

- **A.** Run `ios_config` with a show command
- **B.** Parse the running-config manually with regex in YAML
- **C.** Use the `httpapi` plugin against the CLI
- **D.** Use `ios_facts` or a resource module in `gathered` state

Correct answer: D. `ios_facts` and `gathered` state return normalized, structured data (serial, version, model, interfaces) that you can store or report on. Raw `ios_config` output is unstructured text.

Q9. Which variable tells Ansible which network OS it is automating, enabling the correct module behavior?

- **A.** `ansible_connection`
- **B.** `ansible_host`
- **C.** `ansible_python_interpreter`
- **D.** `ansible_network_os`

Correct answer: D. `ansible_network_os` (e.g., `cisco.ios.ios`) selects the platform-specific behavior. `ansible_connection` picks the plugin, `ansible_host` is the address, and `ansible_python_interpreter` is unrelated to network OS.

Q10. An engineer wants to reuse the same VLAN definitions across many switch groups while keeping the data separate from the task logic. What is the recommended structure?

- **A.** Hard-code the VLANs in each playbook
- **B.** Place the VLAN data in `group_vars` and reference it from a single playbook
- **C.** Create one playbook per switch
- **D.** Store the VLANs in the inventory hostnames

Correct answer: B. Putting data in `group_vars` (or `host_vars`) and referencing it from one parameterized playbook separates **data from logic**, the standard Ansible pattern for reuse and scale.

1.2 Network Automation with Terraform

Terraform (by HashiCorp) is an **Infrastructure-as-Code** tool built around a **declarative** model and a concept Ansible does not emphasize: **state**. You describe the desired infrastructure in HashiCorp Configuration Language (**HCL**), and Terraform compares that description to a stored **state file** that records what it believes already exists. From that comparison it builds a **plan** — the minimal set of create, update, or delete actions needed to reach the desired state — and then **applies** it.

> **Terraform vs. Ansible (exam favourite)**
>
> Ansible is primarily **procedural/push**: it runs tasks in order each time. Terraform is **declarative with state**: it tracks reality in a state file and computes the difference (drift). Terraform shines at *lifecycle* management — create, change, and destroy resources — especially on controller-based platforms (ACI, Meraki, Catalyst Center, SD-WAN). Ansible shines at *configuration push* to many CLI devices.

Providers: how Terraform talks to Cisco

Terraform itself knows nothing about networking. **Providers** are plugins that translate HCL resources into API calls. Cisco publishes providers under the `CiscoDevNet` namespace, including `CiscoDevNet/iosxe` (which uses RESTCONF under the hood), `CiscoDevNet/aci`, `CiscoDevNet/nxos`, `CiscoDevNet/meraki`, `CiscoDevNet/sdwan`, `CiscoDevNet/ise`, and `CiscoDevNet/catalystcenter`. You declare which providers you need in a `required_providers` block.

providers.tf — declaring and configuring the IOS XE provider

```
terraform {
  required_providers {
    iosxe = {
      source  = "CiscoDevNet/iosxe"
      version = "~> 0.5"
    }
  }
}

provider "iosxe" {
  url      = var.iosxe_url      # https://192.0.2.11
  username = var.iosxe_user
  password = var.iosxe_password
  insecure = true               # lab only; use valid certs in production
}
```

Variables and the core workflow

Inputs are declared with `variable` blocks and supplied via a `terraform.tfvars` file, environment variables, or the CLI. The everyday workflow is three commands:

- `terraform init` — download providers and initialize the working directory and backend.

- `terraform plan` — show what *would* change (the diff between desired state and recorded state). Nothing is applied.
- `terraform apply` — execute the plan after confirmation. `terraform destroy` removes managed resources.

variables.tf + terraform.tfvars

```
# variables.tf
variable "iosxe_url"      { type = string }
variable "iosxe_user"     { type = string }
variable "iosxe_password" { type = string, sensitive = true }

# terraform.tfvars  (never commit real secrets; use a secrets manager or TF_VAR_*)
iosxe_url      = "https://192.0.2.11"
iosxe_user     = "automation"
```

Example 1 — VLAN as a resource

A **resource** block declares one managed object. Terraform records its identity in state, so if someone deletes the VLAN out-of-band, the next `plan` shows drift and offers to recreate it.

vlans.tf

```
resource "iosxe_vlan" "sales" {
  vlan_id = 10
  name    = "SALES"
}

resource "iosxe_vlan" "voice" {
  vlan_id = 20
  name    = "VOICE"
}
```

Example 2 — Interface and OSPF

interfaces_ospf.tf (resource names vary by provider version)

```
resource "iosxe_interface_ethernet" "uplink" {
  type        = "GigabitEthernet"
  name        = "2"
  description = "Uplink to core"
  shutdown    = false
}

resource "iosxe_ospf" "core" {
  process_id = 1
  router_id  = "1.1.1.1"
}
```

FREE SAMPLE

Enjoying the preview?

You have just read the first 15 pages of the AUTOCOR 350-901 Complete Learning Guide — the all-in-one study guide for the Cisco CCNP & CCIE Automation core exam (350-901, v2.0 blueprint).

- **Complete coverage** — all four exam domains, start to finish
- **240 practice questions** — Cisco-style, with fully explained answers
- **Real, runnable code** — Ansible, Terraform, Python, pyATS, FastMCP & more
- **Reference built in** — full exam blueprint + glossary of key terms
- **145 pages** — one polished, up-to-date volume

Get the complete guide

and walk into the exam fully prepared.

by Jozef Baroš