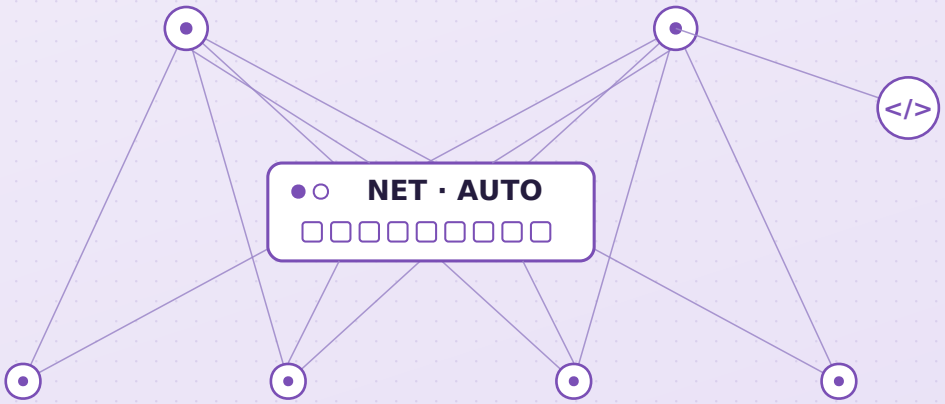
DATA CENTER · AUTOMATION



DCNAUTO
300-635

Complete Learning Guide

Automating Cisco Data Center
Networking Solutions v2.0

An Independent, Unofficial Study Guide

Jozef Baroš
AUTHOR

DATA CENTER · AUTOMATION

DCNAUTO 300-635

Complete Learning Guide

Automating Cisco Data Center Networking Solutions v2.0

An Independent, Unofficial Study Guide

Jozef Baroš

v1.0 · July 2026

Copyright, Trademarks and Disclaimer

DCNAUTO 300-635 v2.0 — Complete Learning Guide

Copyright © 2026 Jozef Baroš. All rights reserved. Version 1.0, July 2026.

No part of this publication may be reproduced, distributed or transmitted in any form or by any means without the prior written permission of the author, except for brief quotations in a review, and except that the code listings in this book may be freely used, adapted and deployed in your own work without attribution or restriction.

This book is not affiliated with Cisco

> ⚠️ **Common Pitfall — Read this page before you read anything else**
>
> **This is an independent, unofficial study guide.** It is not authorised, endorsed, sponsored, reviewed or approved by Cisco Systems, Inc. or by any of its affiliates.
>
> **Cisco, the Cisco logo, CCNA, CCNP, CCIE, NX-OS, Nexus, ACI, Nexus Dashboard, Nexus Hyperfabric, DevNet, pyATS and Genie are trademarks or registered trademarks of Cisco Systems, Inc.** All other trademarks are the property of their respective owners.
>
> **Every trademark in this book is used nominatively** — that is, solely to identify the products and the certification examination the book discusses, as permitted by law. No Cisco logo, trade dress or copyrighted artwork appears anywhere in this book. Every diagram in it is an original work created for this book.
>
> **This book contains no examination content.** It is not a braindump. It contains no real, remembered, leaked or reconstructed examination questions. Every practice question in it was written from the published examination blueprint, to teach the topic that blueprint names. Attempting to obtain or use real examination content violates the Cisco Certifications and Confidentiality Agreement, and it is a good way to lose a certification you worked hard for.

About the code and the configuration

Every command, playbook, template, plan and script in this book was written to be correct and instructive. None of it was written to be run, unread, against your production fabric.

- **Test everything in a lab first.** Section 4.1 exists partly to tell you how, and Cisco's DevNet Sandbox is free.
- **Several examples are deliberately insecure, and say so.** `verify=False`, `hostkey_verify=False`, `insecure = true` and inline passwords appear because they make

an example short enough to read. Each is labelled. Section 5.2 discusses at some length what happens when that label is ignored — including by a language model that learned from books like this one.

- **Products move.** Nexus Dashboard API paths, collection versions, provider arguments and NX-OS behaviour change between releases. Where this book gives a path or a version, it is illustrative. The authority is the documentation for the release in front of you.

No guarantee

This book was written to teach the subject matter of the DCNAUTO 300-635 v2.0 examination thoroughly and honestly. It cannot and does not guarantee that you will pass, because no book can: the examination is Cisco's, the blueprint may be revised, and the outcome depends on your preparation, your experience and the day. The author and publisher accept no liability for any loss or damage arising from the use of this book or the code within it.

Errata and contact

If you find an error — a wrong parameter, a stale path, a question whose answer you can argue with — the author would genuinely like to know. Corrections make the next version better for the next reader, and technical books are never finished, only published.

Contents

How to Use This Book

This book has one organising principle: **it follows the published examination blueprint exactly.** Five sections, mapping one-to-one onto the five domains. Twenty-six sub-sections, mapping one-to-one onto the blueprint's twenty-six topics, in its order, using its wording. When Cisco writes *"Construct an Ansible playbook with controller and device collections"*, there is a sub-section called that, and it is about exactly that.

That is not decoration. It means that when you have finished a sub-section, you can tick a line on the blueprint and know precisely what you have covered — and, more usefully, precisely what you have not.

What is in every sub-section

Element	What it is for
Prose	The explanation. Written to be read once, not skimmed four times.
An original diagram	One per topic. Drawn for this book, not borrowed.
Runnable code	Real playbooks, templates, plans and scripts — not fragments.
Did you know?	The detail that makes the topic click, usually the one nobody mentions.
Exam Tip	What the examination is actually likely to ask, and how it phrases it.
Common Pitfall	The mistake you are otherwise going to make. Written from experience.
From the Field	Why this matters at 3am, when it stops being a syllabus item.
Summary + Key Takeaways	A dense recap, then six lines you could write on a card.
5 Knowledge Checks	Exam-style questions with a full explanation of *why*.

The questions

There are **130 knowledge-check questions** — five at the end of every sub-section — and a **60-question practice exam** in Appendix B, timed at 90 minutes to match the real thing. That is 190 questions.

Every one of them comes with an explanation that says why the right answer is right **and why the plausible wrong ones are wrong**, because the second half is where the learning is. If you only ever read the explanations for questions you got wrong, you are using half the book.

> ✅ **Exam Tip — Do not memorise the answer letters**
>
> The correct answers are deliberately spread evenly across A, B, C and D, and they are shuffled. If you find yourself remembering that a question's answer was 'C', you have learned nothing that will survive contact with the examination. Cover the options, answer the question from the stem alone, and then read all four explanations.

Three ways to read it

If you are...	Do this
Starting from scratch	Front to back. The sections build on each other deliberately — Section 3's `clid()` and Section 3.6's `cli_show` fail for the same reason, and Section 5.3's approval gate is Section 2.4's `terraform plan` wearing a different hat. Reading in order is how you notice.
Experienced, and revising	Read each sub-section's **Summary** and **Key Takeaways** first. If both feel obvious, do the five questions. If you score 5/5, move on. If not, read the sub-section — you have found a gap you did not know about, which is the entire point of doing it this way.
A week out	Appendix B under exam conditions. 90 minutes, no notes, no pausing. Then map every wrong answer back to its sub-section, and read only those. A wrong answer is a diagnostic, not a verdict.

Type the code

Four of the five domains use **Construct**, **Implement** or **Configure** as their verb. Those are not reading verbs. There is no way to fake fluency with `state: replaced`, `terraform import`, `sample-interval 0` or `dohost` — you either recognise them instantly under time pressure or you do not, and the difference is made by having typed them.

You do not need hardware. **Cisco's DevNet Sandbox is free** and includes NX-OS, ACI and Nexus Dashboard environments; **Cisco Modeling Labs** builds a fabric you can break; **Containerlab** starts a topology in seconds. Section 4.1 covers all of it. There is no excuse left, and there has not been for some years.

> 🌍 **From the Field — The honest study plan**
>
> Six weeks, one sub-section per sitting, in blueprint order. Read it, type the code, do the five questions the same day. At the end of each week, redo the previous week's questions cold — spaced repetition is not a trick, it is the only thing that reliably moves knowledge from recognised to known. In week six, sit Appendix B, then reread only what you got wrong. That is roughly forty hours, it works, and it is

considerably less glamorous than any shortcut you have been offered.

The Examination Blueprint

DCNAUTO 300-635 v2.0 — Automating Cisco Data Center Networking Solutions. A 90-minute examination, associated with both the **CCNP Data Center** and the **CCNP Automation** certifications. The technologies in scope are Cisco NX-OS, Nexus Dashboard, ACI in network-centric mode, and Nexus Hyperfabric.

The table below summarises what each topic covers, **in this book's words**, alongside the section that treats it. The domain numbering and the percentage weights are Cisco's and are reproduced as published; the descriptions are the author's own summaries and are not Cisco's wording.

> ⚠️ **Common Pitfall — Read the official blueprint, not this table**
>
> This summary exists so you can navigate the book — it is not a substitute for the source. **Cisco publishes the authoritative topic list at cisco.com, and that document is the only one that governs what appears on your examination.** It is free, it is short, and it is revised without notice. Read it before you book, and read it again the week before you sit. Cisco's own caveat applies and is worth taking literally: the published topics are general guidelines, and other related topics may appear on any given delivery.

#	What the topic covers	Section
1.0	**Network Automation Foundation — 15%**	**Section 1**
1.1	The three YANG model families, and when each one applies	1.1
1.2	ACI's object model in network-centric mode, and how its pieces relate	1.2
1.3	What a DPU is, and the problem it exists to solve	1.3
1.4	The four management interfaces — transports, ports and roles	1.4
1.5	Deriving a gRPC payload from a YANG module, and the tooling for it	1.5
2.0	**Infrastructure as Code — 25%**	**Section 2**
2.1	What IaC means, and what GitOps adds on top of it	2.1
2.2	Templating: iteration, branching, whitespace control and filters	2.2
2.3	Playbooks that drive a controller and individual switches	2.3
2.4	Plans that drive a controller and individual switches	2.4
2.5	Diagnosing failures in both tools, layer by layer	2.5

#	What the topic covers	Section
3.0	**Network Element Programmability — 25%**	**Section 3**
3.1	Reading and writing configuration over NETCONF from Python	3.1
3.2	Provisioning a switch that has never been configured	3.2
3.3	Running scripts on the switch itself, in Bash and in Python	3.3
3.4	How the controller models reusable configuration and applies it	3.4
3.5	Authoring those templates yourself	3.5
3.6	What the switch's HTTP APIs offer	3.6
4.0	**Operations — 25%**	**Section 4**
4.1	Where simulation fits in operations, and what it can prove	4.1
4.2	Proving a change did only what you intended, from the command line	4.2
4.3	The parts a streaming-telemetry pipeline is assembled from	4.3
4.4	Setting up telemetry subscriptions, in both directions	4.4
4.5	Wiring automation to an authoritative database of intent	4.5
4.6	Collecting health from switch and controller in one script	4.6
4.7	Following a packet through a Linux host's virtual networking	4.7
5.0	**AI in Automation — 10%**	**Section 5**
5.1	Using AI tooling to write automation code	5.1
5.2	Assessing the risks of an AI-based automation design	5.2
5.3	Connecting agents to devices, controllers and platforms	5.3

Reading the weights

Sections 2, 3 and 4 are **seventy-five percent of the examination between them**, and they are the three that ask you to construct, implement, configure and troubleshoot rather than describe. Section 1 is fifteen percent of foundation you cannot skip, because the rest of the book assumes it. Section 5 is ten percent and three Describes — it is the cheapest ten percent in the examination, and it takes an afternoon.

If you are budgeting time and something has to give, it is not Section 4. It is the largest domain by topic count, it is the one people underprepare because it feels like operations rather than automation, and it is a quarter of the marks.

Section 1 — Network Automation Foundation

Fifteen percent of the exam sits here, and it is the fifteen percent that decides the other eighty-five. Every topic in the rest of this book — Jinja2 templates, Ansible playbooks, Terraform plans, telemetry subscriptions, AI agents that call APIs — is ultimately a program writing a **structured document** and handing it to a **transport** so that a **device** can act on it. Section 1 is where you learn what that document looks like, who defines its shape, and which transports will carry it.

If you come from a CLI background, the mental shift is this: you are no longer typing commands, you are **submitting data against a schema**. The schema is YANG. The transports are NETCONF and gRPC. The controller-side abstraction is the ACI object tree. The new silicon in the path is the DPU. Get these four straight and the exam's "construct" topics become mechanical.

Sub-section	Focus	Verb the exam uses
1.1	OpenConfig, IETF and native YANG models	Describe
1.2	ACI network-centric mode: EPG, BD, contract, VRF	Describe
1.3	DPUs in data center network switches	Describe
1.4	NETCONF, gNMI, gRPC and gNOI	Describe
1.5	Building a gRPC payload from YANG (YANG Suite, pyang)	Construct

> ✅ **Exam Tip — Watch the verb, budget your effort accordingly**
>
> Four of the five topics here say **Describe** — those questions test whether you can pick the right definition or spot the false statement in a list. Topic 1.5 says **Construct**, which in Cisco's blueprint language means you may be shown a payload, a path, or a `pyang` invocation and asked what it does, what is wrong with it, or which option produces the required result. Read the describe topics for precision of vocabulary; drill 1.5 by actually typing the commands.

> 🌍 **From the Field — Why this section is not filler**
>
> In production the failures are almost never in the clever part. They are in the boring part: the wrong `origin` on a gNMI path, a native leaf that has no OpenConfig equivalent, a bridge domain configured for unicast routing when the design assumed L2-only. Engineers who skip the foundation write

automation that works on the lab switch and detonates on the fabric.

1.1 YANG Data Models: OpenConfig, IETF and Native

Why a model exists at all

For thirty years the network's management interface was a **command-line interface** — text designed to be read by a human sitting at a terminal. Automation against a CLI means **screen-scraping**: send a string, receive a wall of text, and use pattern matching to guess where the useful part is. It works until the vendor adds a column to `show interface status`, at which point every script in the estate breaks silently, which is the worst way for anything to break.

YANG ("Yet Another Next Generation", standardised in RFC 7950) is the answer. YANG is a **data modelling language**: it does not move data and it does not configure anything. It describes, formally and machine-readably, what data a device may hold — which nodes exist, what type each one is, which are writable, which are read-only, which depend on which. If a database schema tells you what a table is allowed to contain, a YANG module tells you what a switch is allowed to contain.

That distinction is the single most examinable idea in this sub-section, so state it to yourself in one line: **YANG is the schema; NETCONF, RESTCONF and gNMI are the protocols that carry data conforming to it.** They are orthogonal. You can model data you never transport, and a protocol can carry a model it has never heard of.

Figure 1.1 — Anatomy of a YANG module, and the three model flavours

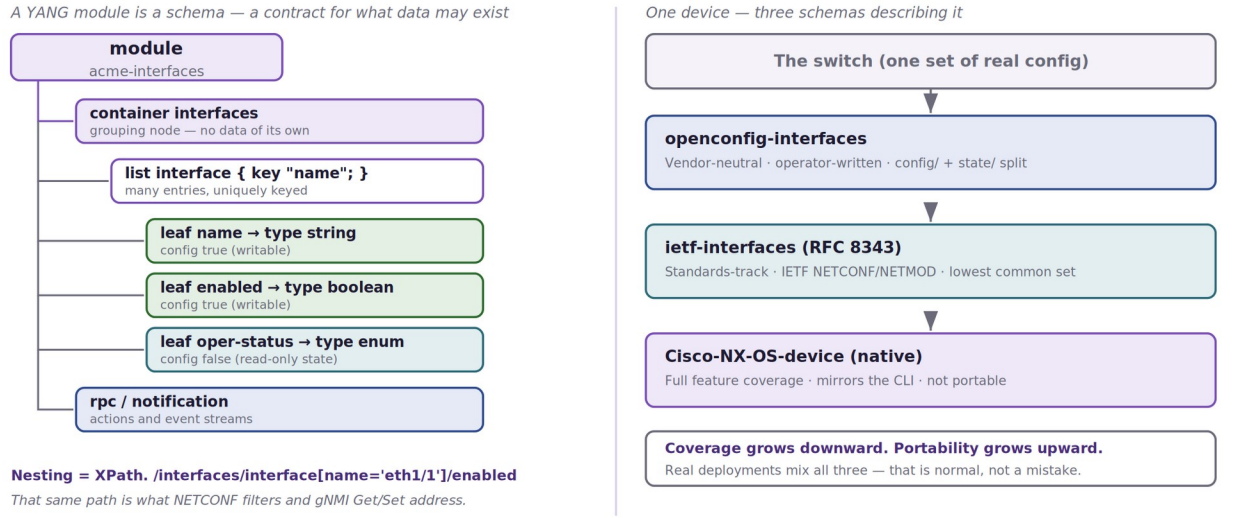


Figure 1.1 — The nodes a YANG module is built from (left), and the three families of module that describe the very same switch (right).

The nodes a module is built from

A YANG module is a tree. Roughly a dozen statement types build it, and the exam expects you to recognise them on sight rather than write them from memory.

Statement	What it is	Mental model
module	The top-level container of a schema, with its own namespace and prefix	A file / a database
container	A grouping node that holds other nodes but no value of its own	A folder
list	Zero or more entries, each uniquely identified by its key	A table
key	The leaf (or leaves) that make a list entry unique	A primary key
leaf	A single node holding exactly one value of a given type	A cell
leaf-list	A leaf that may hold multiple values of the same type	An array of scalars
grouping / uses	A reusable block of nodes, and the statement that instantiates it	A function and its call
augment	Adds nodes into a tree defined by someone else's module	A subclass / an extension
deviation	Declares that this device does NOT behave as the model says	An honest asterisk
rpc / action	An operation the device exposes, with input and output	An API endpoint
notification	An event the device can emit asynchronously	A webhook

Cutting across all of them is one keyword that matters more than the rest: **`config`**. A node declared `config true` is **configuration** — something you set, something that survives a reload, something that appears in `get-config`. A node declared `config false` is **operational state** — counters, oper-status, neighbour tables. It is read-only, it is not in `get-config`, and no amount of well-formed XML will let you write to it.

A minimal, complete YANG module — read it as a schema, not as configuration

```
module acme-interfaces {
  yang-version 1.1;
  namespace "urn:acme:params:xml:ns:yang:acme-interfaces";
  prefix acme-if;

  organization "ACME Networks";
  revision 2026-01-15 { description "Initial revision."; }
```

```
  container interfaces {
    description "Top-level container for all interfaces.";

    list interface {
      key "name";                       // <- makes each entry unique
      description "One physical or logical interface.";

      leaf name        { type string; }            // config true (default)
      leaf description { type string { length "0..240"; } }
      leaf enabled     { type boolean; default true; }
      leaf mtu         { type uint16 { range "576..9216"; } }

      leaf oper-status {
        config false;                   // <- OPERATIONAL STATE, read-only
        type enumeration { enum up; enum down; enum testing; }
      }
      leaf-list tags   { type string; }            // many values, one leaf
    }
  }

  notification link-change {            // an event, not a value
    leaf if-name    { type leafref { path "/interfaces/interface/name"; } }
    leaf new-status { type string; }
  }
}
```

Look at what the schema has already bought you before a single packet moves. `mtu` cannot be set to 70000 — the range constraint rejects it. `enabled` cannot be set to the string `"yes"` — it is a boolean. `oper-status` cannot be written at all. A client library that loads this module can catch those errors **on your laptop**, in milliseconds, instead of discovering them on a production leaf at 02:00.

> 💡 **Did you know? — The tree is the path, and the path is the API**
>
> Nesting in YANG is not decoration — it defines addressability. The `enabled` leaf above lives at `/interfaces/interface[name='eth1/1']/enabled`. That XPath is exactly what a NETCONF subtree filter selects, exactly what a gNMI `Get` requests, and exactly what a telemetry subscription streams. Learn to read a YANG tree and you have simultaneously learned to read every path in this book.

Three families describing one switch

Here is the fact that confuses newcomers: a single Nexus switch is described by **several** YANG models at once, and they overlap. This is not a bug. Three constituencies wanted three different things.

IETF models — the standards-track baseline

Published as RFCs by IETF working groups. `ietf-interfaces` (RFC 8343), `ietf-routing` (RFC 8349), `ietf-ip`, `ietf-system`. They are the product of consensus, which means they are stable, well specified, and deliberately **minimal** — they model what every vendor could agree every device does. That makes them portable and it makes them shallow. IETF models also carry the historic quirk of keeping configuration and state in the same tree, distinguished only by the `config` keyword.

OpenConfig models — written by the people who run networks

OpenConfig is an **operator-led** consortium — the modules were driven by large network operators who were tired of writing a different automation stack per vendor. `openconfig-interfaces`, `openconfig-bgp`, `openconfig-platform`. Two design traits matter for the exam. First, they are **vendor-neutral by intent**: the same path should work on hardware from different manufacturers. Second, they use an explicit **`config/` and `state/` container split** — what you asked for lives under `config`, what is actually happening lives under `state`, and comparing the two is how you detect drift without inference.

Native models — everything the box can actually do

Vendor-authored, mirroring the device's own feature set. On Nexus this is `Cisco-NX-OS-device.yang` — a single enormous module generated from the platform's internal object model, covering essentially the whole CLI. Native models are **complete** and they are **not portable**. If the feature exists on the box, the native model can configure it; if you move to another vendor, every native path you wrote is worthless.

	IETF	OpenConfig	Native (Cisco-NX-OS-device)
Authored by	IETF working groups	Operator consortium + vendors	Cisco
Published as	RFCs	GitHub, openconfig/public	With the NX-OS image
Portability	High	High	None
Feature coverage	Lowest — the common subset	Middle — what operators asked for	Complete — the whole box
Config vs state	Same tree, `config` keyword	Explicit `config/` and `state/`	Mixed, model-dependent
Pace of change	Slow (consensus)	Fast (semver releases)	Tracks the NX-OS release
Use it when	You need a standards answer	Multi-vendor automation	The feature exists nowhere else

> ⚠️ **Common Pitfall — "Vendor-neutral" is a design goal, not a guarantee**
>
> Two vendors can both claim `openconfig-bgp` support and still disagree about default values, which optional containers exist, or what a `deviation` statement quietly removed. Never assume a path that works on one platform works on another because the model name matches. Ask the device: run a `Capabilities` RPC and check the advertised revision date, then check the platform's deviation modules. Portability is something you verify, not something you inherit.

Reading a model with pyang

`pyang` is the standard open-source YANG toolchain — a validator and translator. In practice you use it for one thing above all: rendering a 40,000-line module as a readable tree so you can find the path you need. You will meet it again in 1.5; here, just learn to read its output.

Rendering a module as a tree, and narrowing to one branch

```
$ pip install pyang
$ pyang -f tree acme-interfaces.yang

module: acme-interfaces
  +--rw interfaces
     +--rw interface* [name]
        +--rw name           string
        +--rw description?   string
        +--rw enabled?       boolean
        +--rw mtu?           uint16
        +--ro oper-status    enumeration
        +--rw tags*          string
  notifications:
    +---n link-change
       +--ro if-name?      -> /interfaces/interface/name
       +--ro new-status?   string

# Real models import other models. Point pyang at the whole directory:
$ pyang -f tree --path ./yang-modules openconfig-interfaces.yang

# 40,000 lines is unreadable. Cut to the branch you care about:
$ pyang -f tree --tree-path /interfaces/interface/config \
        --path ./yang-modules openconfig-interfaces.yang
```

The symbols carry all the meaning, and they are examinable:

- **`rw`** — read-write. This node is configuration (`config true`).
- **`ro`** — read-only. Operational state (`config false`). You cannot write it.
- **`*`** — a list or leaf-list. `interface* [name]` means many entries, keyed on `name`.
- **`?`** — optional. The node may be absent.
- **`->`** — a **leafref**: this leaf's value must exist at the referenced path.
- **`+---n`** / **`+---x`** — a notification / an action (RPC).

> 🌍 **From the Field — Where the modules actually come from**
>
> On NX-OS the models ship in the image — `bootflash:/yang/` — and the safest habit is to pull them off the exact box you are automating rather than from GitHub, because the revision on your switch is the revision that matters. For OpenConfig, `github.com/openconfig/public` is authoritative. Pin the revision date in your repo alongside your playbooks; a model upgrade is a breaking API change and deserves the same review as a code change.

> ✅ **Exam Tip — The four sentences worth memorising verbatim**
>
> **YANG models data; NETCONF, RESTCONF and gNMI transport it.** A question that treats YANG as a protocol is testing exactly this.
>
> **`config false` = operational state = read-only = absent from `get-config`.**
>
> **OpenConfig = operator-driven, vendor-neutral, explicit `config/` and `state/` split.**
>
> **Native = complete coverage, zero portability.** It exists because the standard models do not cover everything, and it is the correct answer whenever the question mentions a platform-specific feature.

Summary

YANG is a data modelling language — a schema for what a device may contain — and nothing more; the protocols that carry YANG-modelled data are separate concerns. A module is a tree of containers, lists (uniquely identified by their `key`), and leaves, with `config true` marking writable configuration and `config false` marking read-only operational state. That nesting defines XPath-style paths, which are the addresses every protocol in this book uses. The same switch is described by three families of module at once: IETF models are standards-track, portable and minimal; OpenConfig models are operator-driven, vendor-neutral and split `config/` from `state/`; native models such as `Cisco-NX-OS-device` cover every feature the box has but are worthless on anyone else's

hardware. `pyang -f tree` renders any of them as a readable tree, where `rw` means configuration, `ro` means state, `*` marks a list and `?` marks an optional node.

Key Takeaways

YANG = schema. NETCONF / gNMI = transport. They are orthogonal; do not conflate them.

`config true` → writable, appears in `get-config`. `config false` → read-only operational state.

Three families: IETF (standard, minimal), OpenConfig (operator-driven, neutral, `config/`+`state/`), native (complete, non-portable).

A YANG tree is an XPath tree — the nesting is literally the address you will filter, get, set and subscribe on.

`pyang -f tree --path <dir>` is how you read a real model; `--tree-path` cuts it to one branch.

Portability is verified, not assumed — check the advertised revision and any deviations before you trust a shared path.

Knowledge Check — 1.1 YANG Data Models

Q1. Which statement most accurately describes the relationship between YANG and NETCONF?

A. YANG is a transport protocol optimised for XML, and NETCONF is the schema language that describes the payloads it carries

B. YANG and NETCONF are two names for the same specification, used interchangeably in RFC 6241

C. YANG defines the structure and constraints of the data; NETCONF is a protocol that transports and operates on data conforming to that structure

D. NETCONF generates YANG modules dynamically at runtime by introspecting the device's running configuration

Correct answer: C. YANG (RFC 7950) is a data modelling language — a schema. NETCONF (RFC 6241) is a protocol that carries and manipulates data conforming to such schemas. The two are deliberately separable, which is exactly why the same YANG module can be served over NETCONF, RESTCONF and gNMI. Any answer that treats YANG as a protocol, or NETCONF as a modelling language, inverts the relationship.

Q2. In a YANG module, what is the purpose of the `key` statement inside a `list`?

A. It names the leaf (or leaves) whose value uniquely identifies each entry in the list

B. It encrypts the list contents when they are transmitted over the management session

C. It marks the list as read-only operational state that cannot be modified by a client

D. It sets the maximum number of entries the list is permitted to contain

Correct answer: A. `key` is the list's primary key — it names the leaf or leaves that make each entry unique, and those key leaves are what you place inside the square brackets of a path, e.g. `/interfaces/interface[name='eth1/1']`. Read-only is `config false`; entry limits are `min-elements` / `max-elements`; encryption is a transport concern and has nothing to do with YANG.

Q3. A leaf is declared with `config false`. Which of the following is true?

A. It represents configuration that is applied only after an explicit commit to the startup datastore

B. It is a mandatory leaf that must be supplied in every edit-config payload targeting its parent

C. It is configuration that is hidden from unauthenticated users but writable by an administrator

D. It represents operational state: it is read-only and does not appear in the output of a `get-config` operation

Correct answer: D. `config false` marks a node as operational state — counters, oper-status, learned neighbours. It is read-only and is excluded from `get-config`, which by definition returns configuration only. To retrieve it over NETCONF you use `<get>`, which returns configuration and state together. Access control is a separate mechanism (NACM), and mandatory-ness is the `mandatory` statement.

Q4. Which model family is characterised by being authored primarily by network operators, aiming for vendor neutrality, and using an explicit split between `config/` and `state/` containers?

A. IETF standards-track models such as ietf-interfaces (RFC 8343)

B. OpenConfig

C. Native models such as Cisco-NX-OS-device

D. IEEE 802.1 management models

Correct answer: B. OpenConfig came out of an operator-led consortium whose explicit goal was a vendor-neutral automation surface, and its signature structural trait is the paired `config/` (intended) and `state/` (actual) containers — which makes drift detection a simple comparison rather than an inference. IETF models are standards-track and keep both in one tree distinguished by the `config` keyword; native models are vendor-specific and non-portable.

Q5. An organisation has standardised on OpenConfig for multi-vendor automation. Why would its engineers still write some tasks against `Cisco-NX-OS-device` native paths?

A. Because native models cover the platform's complete feature set, including features that no standard or OpenConfig model represents

B. Because native models are the only ones that can be transported over gNMI; OpenConfig requires NETCONF

C. Because native models are portable across vendors while OpenConfig models are locked to a single platform

D. Because native models are validated by the IETF and therefore carry stronger correctness guarantees

Correct answer: A. The native model is generated from the platform's own object model, so it covers essentially everything the box can do — including features that standard bodies and OpenConfig have never modelled. That completeness is the entire reason to accept its total lack of portability. gNMI can carry any model (you select it with `origin`), portability is precisely what native models lack, and native models are vendor-published, not IETF-ratified.

1.2 ACI Network-Centric Mode: EPG, Bridge Domain, Contract and VRF

In ACI, everything is an object

Cisco **ACI** (Application Centric Infrastructure) is a controller-driven fabric. The controller is the **APIC**, and the fabric's entire configuration — every tenant, every VLAN, every port binding — lives in one enormous tree of objects called the **Management Information Tree (MIT)**. There is no per-switch config file to edit. You create, modify and delete objects, and the APIC renders them into hardware.

This matters for automation because the API is a direct projection of that tree. Three vocabulary items unlock it:

- **Class** — the object's type, written as a two-part name: a package prefix plus a class name. `fvTenant` is the `fv` (fabric virtualisation) package's `Tenant` class. `vzBrCP` is the `vz` (virtualisation zones — policy) package's `BrCP`, a binary contract profile.
- **RN (Relative Name)** — the object's name within its parent, prefixed by its class's naming convention. A tenant called PROD has RN `tn-PROD`; a bridge domain called BD-10 has RN `BD-BD-10`.
- **DN (Distinguished Name)** — the full path from the root, `uni`, down to the object: the concatenation of every RN on the way. This is the object's unique address and, in effect, its URL.

DNs are hierarchical addresses — read them right to left to see the containment

```
uni/tn-PROD                                  # the tenant PROD
uni/tn-PROD/ctx-PROD-VRF                     # a VRF inside it
uni/tn-PROD/BD-BD-10                         # a bridge domain
uni/tn-PROD/BD-BD-10/subnet-[10.0.10.1/24]   # its gateway subnet
uni/tn-PROD/ap-NETWORK/epg-EPG-10            # an EPG inside an app profile
uni/tn-PROD/brc-WEB-TO-DB                    # a contract

# Everything is addressable over REST. The DN becomes the URL:
POST https://apic/api/mo/uni/tn-PROD.json
GET  https://apic/api/mo/uni/tn-PROD/BD-BD-10.json?query-target=children

# ...and every object of a class is queryable fabric-wide:
GET  https://apic/api/class/fvAEPg.json
```

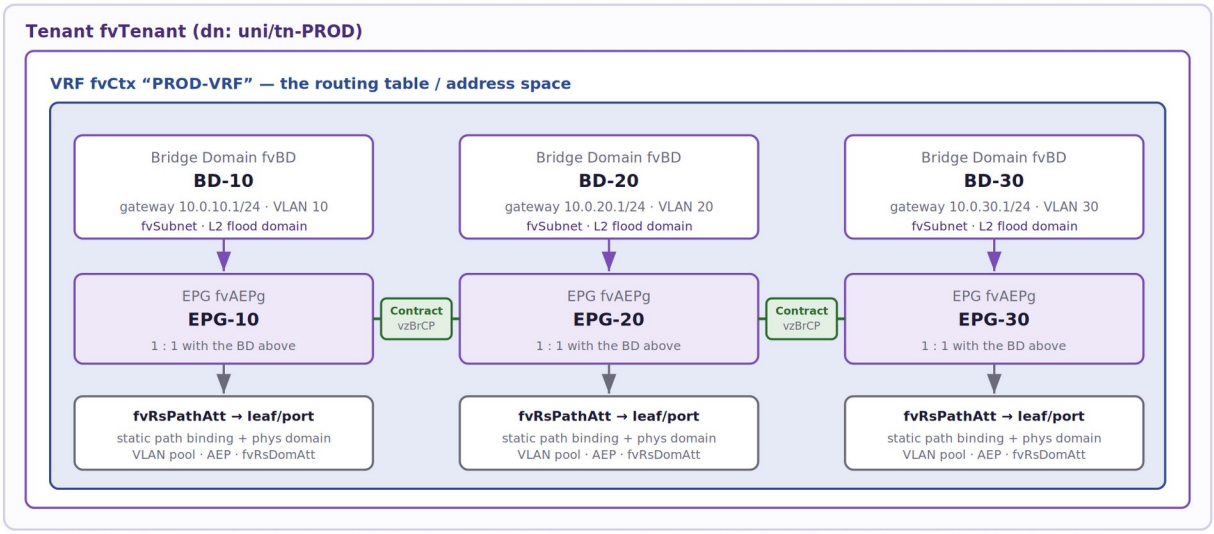
> 💡 **Did you know? — `uni` is short for “universe”**
>
> Every configurable DN in ACI starts at `uni` — the policy universe, the root of the configurable tree. Its sibling root, `topology`, holds the physical inventory: `topology/pod-1/node-101/sys/phys-[eth1/5]`. When you see a DN beginning with `topology` you are reading hardware state; when it

> begins with `uni` you are reading intent. The APIC's job is to make the second true of the first.

The five objects the exam actually asks about

Figure 1.2 — ACI network-centric mode: the object hierarchy you automate

Network-centric = one VLAN maps to one Bridge Domain and one EPG. The fabric behaves like a big traditional switch.



Shortcut most network-centric designs take: leave the VRF unenforced, or attach a permit-any contract to vzAny — then policy lives where it always did, on the firewall.

Figure 1.2 — The containment hierarchy in a network-centric design. Each VLAN gets exactly one bridge domain and exactly one EPG.

Object	Class	What it is	Nearest traditional equivalent
Tenant	fvTenant	The top-level administrative and policy boundary	A VDC / a customer / an environment
VRF	fvCtx	A routing table and address space; the unit of L3 isolation	A VRF — identical concept
Bridge Domain	fvBD	An L2 flood domain; where the gateway lives	A VLAN's broadcast domain + SVI
Subnet	fvSubnet	The anycast gateway address, configured under a BD	An SVI's `ip address`
EPG	fvAEPg	A group of endpoints that share a policy identity	The set of ports in a VLAN
Contract	vzBrCP	The policy permitting traffic between EPGs	An ACL applied between VLANs
App Profile	fvAp	A folder that holds EPGs	A naming convention

The relationship that trips people up is **VRF ↔ bridge domain ↔ EPG**. A VRF holds many bridge domains. A bridge domain is an L2 flood domain and it is where the **gateway** (`fvSubnet`) is configured — not on the EPG. A bridge domain can hold many EPGs. An EPG is a **policy identity**: a bag of endpoints to which contracts are applied. Endpoints inside one EPG talk freely; endpoints in different EPGs need a contract, unless the VRF is unenforced.

> ⚠️ **Common Pitfall — The gateway lives on the bridge domain, not the EPG**
>
> This is the most reliably missed fact in the whole ACI blueprint. `fvSubnet` is a child of `fvBD`. An EPG has no IP address. You *can* place a subnet under an EPG for specific route-leaking designs, but the default — and the answer to the exam question — is the bridge domain. If a payload puts the gateway under `fvAEPg`, that is the wrong option.

Network-centric versus application-centric

ACI's name promises **application-centric** policy: model your application's tiers as EPGs, write contracts describing which tier may talk to which, and let physical topology stop mattering. It is genuinely powerful and almost nobody deploys it on day one, because it demands that you already know exactly how your applications communicate — and almost nobody does.

So the overwhelmingly common starting design is **network-centric mode**: use ACI as a very good, very automatable traditional fabric. The blueprint calls this out by name, and the definition is mechanical.

	Network-centric	Application-centric
VLAN ↔ BD ↔ EPG	Strict 1 : 1 : 1	Many EPGs may share one BD
What defines an EPG	The VLAN / the subnet	The application tier or role
Segmentation policy	Usually unenforced VRF, or a permit-any contract on vzAny	Explicit contracts between tiers
Where firewalling happens	On the existing external firewall	In the fabric itself
Migration effort	Low — it mirrors what you already have	High — requires application flow discovery
Typical use	Brownfield migration, day one	The target state, after discovery

So a network-centric fabric is one where **each VLAN becomes one bridge domain containing exactly one EPG**, and the policy model is deliberately flattened so the fabric behaves like a large, well-instrumented traditional switch. That 1:1:1 mapping is the definition, and it is what an exam question will hinge on.

> 🌍 **From the Field — Flattening the policy without cheating**
>
> There are two ways to make a network-centric fabric permit everything, and they are not equivalent. Setting the VRF's policy control enforcement to **unenforced** turns contract checking off for the whole VRF — blunt, effective, and it means you get zero policy telemetry. Attaching a **permit-any contract to `vzAny`** (the implicit 'every EPG in this VRF' object) keeps enforcement on but permits everything — slightly more work, and it leaves you a place to start tightening later, one contract at a time, without a flag day. Most teams that intend to reach application-centric mode eventually choose the second.

Wiring an EPG to a physical port

An EPG is a logical construct; something must connect it to copper. Four objects do that, and they are the ones that break first-time deployments:

1. **VLAN pool** (`fvnsVlanInstP`) — the range of VLAN IDs available for encapsulation, either static (you assign) or dynamic (the APIC assigns, for VMM integration).
2. **Domain** (`physDomP` for bare metal, `vmmDomP` for a hypervisor) — binds a VLAN pool to a scope of use.
3. **AEP** (`infraAttEntityP`, Attachable Access Entity Profile) — ties one or more domains to the interface policy groups, i.e. to the actual leaf ports.
4. **Static path binding** (`fvRsPathAtt`) — the EPG's own statement of "I appear on leaf 101, port eth1/5, with encapsulation VLAN 10".

Creating the whole network-centric stack in one atomic REST call

```
POST https://apic/api/mo/uni.json

{
  "fvTenant": {
    "attributes": { "name": "PROD", "dn": "uni/tn-PROD" },
    "children": [

      { "fvCtx": {                                  // VRF
          "attributes": { "name": "PROD-VRF",
                          "pcEnfPref": "unenforced" } } },
```

```
      { "fvBD": {                                    // Bridge Domain
          "attributes": { "name": "BD-10",
                          "arpFlood": "yes",
                          "unicastRoute": "yes",
                          "unkMacUcastAct": "proxy" },
          "children": [
            { "fvRsCtx": {                           // BD -> VRF relation
                "attributes": { "tnFvCtxName": "PROD-VRF" } } },
            { "fvSubnet": {                          // THE GATEWAY. On the BD.
                "attributes": { "ip": "10.0.10.1/24",
                                "scope": "private" } } }
          ] } },

      { "fvAp": {                                    // Application Profile
          "attributes": { "name": "NETWORK" },
          "children": [
            { "fvAEPg": {                            // EPG
                "attributes": { "name": "EPG-10" },
                "children": [
                  { "fvRsBd": {                      // EPG -> BD relation
                      "attributes": { "tnFvBDName": "BD-10" } } },
                  { "fvRsDomAtt": {                  // EPG -> physical domain
                      "attributes": { "tDn": "uni/phys-PHYS-DOM" } } },
                  { "fvRsPathAtt": {                 // EPG -> leaf port + VLAN
                      "attributes": {
                        "tDn": "topology/pod-1/paths-101/pathep-[eth1/5]",
                        "encap": "vlan-10",
                        "mode": "regular" } } }
                ] } }
          ] } }
    ]
  }
}
```

Notice the pattern, because it generalises to everything you will do against the APIC. **Containment** is expressed by nesting objects in `children`. **Association** is expressed by an `fvRs*` object — a *relation*, whose `tnF...Name` or `tDn` attribute names the target. `fvRsCtx` is the BD's relation to a VRF. `fvRsBd` is the EPG's relation to a BD. `fvRsPathAtt` is the EPG's relation to a port. Once you can spot relation objects by their `Rs` infix, unfamiliar ACI payloads stop being intimidating.

> ✅ **Exam Tip — Four facts, four likely questions**

`fvCtx` is the VRF. The class name does not contain the letters V, R or F, and that is exactly why it gets asked.

The gateway (`fvSubnet`) is a child of the bridge domain (`fvBD`).

Network-centric mode = one VLAN, one BD, one EPG — a strict 1:1:1 mapping, with policy usually flattened via an unenforced VRF or a permit-any `vzAny` contract.

Contracts are directional by role: an EPG *provides* a contract, another EPG *consumes* it. Provider is the server side; consumer is the client side.

Summary

ACI is a controller-driven fabric whose entire configuration is a tree of objects (the MIT) hosted by the APIC, where every object has a class (`fvTenant`, `fvBD`), a relative name, and a distinguished name that doubles as its REST URL. The containment chain the exam cares about runs Tenant → VRF (`fvCtx`) → Bridge Domain (`fvBD`, which is where the `fvSubnet` gateway lives) → EPG (`fvAEPg`, a policy identity, not a subnet), with contracts (`vzBrCP`) permitting traffic between EPGs on a provider/consumer basis. Network-centric mode maps one VLAN to exactly one bridge domain and exactly one EPG, and typically flattens policy by leaving the VRF unenforced or attaching a permit-any contract to `vzAny` — producing a fabric that behaves like a traditional switched network while remaining fully API-driven. Connecting an EPG to hardware requires a VLAN pool, a domain, an AEP and a static path binding, and in JSON, containment appears as nested `children` while associations appear as `fvRs*` relation objects.

Key Takeaways

`fvTenant` → `fvCtx` (VRF) → `fvBD` → `fvAEPg` (EPG) — memorise this chain and its class names.

`fvSubnet` (the gateway) hangs off the bridge domain, never off the EPG.

Network-centric = 1 VLAN : 1 BD : 1 EPG, policy flattened via unenforced VRF or permit-any on `vzAny`.

A contract (`vzBrCP`) is provided by one EPG and consumed by another — provider = server side.

The DN is the URL: `uni/tn-PROD/BD-BD-10` → `POST /api/mo/uni/tn-PROD/BD-BD-10.json`.

`children` = containment; `fvRs*` = association. Spotting the `Rs` infix decodes any ACI payload.

Knowledge Check — 1.2 ACI Network-Centric Mode

Q1. Which ACI object provides the Layer 3 routing table and address space — the equivalent of a traditional VRF?

A. fvBD

B. fvAEPg

C. vzBrCP

D. fvCtx

Correct answer: D. `fvCtx` is ACI's VRF: the class name is a historical artefact of the internal object model ('context'), which is precisely why it is a favourite exam target. `fvBD` is the bridge domain (an L2 flood domain that lives inside a VRF), `fvAEPg` is the EPG (a policy grouping of endpoints), and `vzBrCP` is a contract.

Q2. In a standard network-centric design, where is the default gateway for the endpoints in VLAN 10 configured?

A. As an `fvSubnet` object under the EPG (`fvAEPg`) associated with VLAN 10

B. As an `fvSubnet` object under the bridge domain (`fvBD`) associated with VLAN 10

C. As an IP address attribute directly on the `fvCtx` VRF object

D. As a static path binding (`fvRsPathAtt`) on the leaf switch interface

Correct answer: B. `fvSubnet` is a child of `fvBD` — the bridge domain is the L2 flood domain and the anycast gateway belongs to it. The EPG is a policy identity with no address of its own; placing a subnet under an EPG is a specialised route-leaking pattern, not the default. `fvCtx` holds no addresses, and `fvRsPathAtt` binds an EPG to a port and encapsulation, not a gateway.

Q3. Which characteristic best identifies a fabric deployed in network-centric mode?

A. Each VLAN is mapped to exactly one bridge domain containing exactly one EPG

B. Each application tier is modelled as its own EPG, with explicit contracts between tiers

C. All bridge domains are configured with unicast routing disabled and no subnets

D. Every EPG must provide at least one contract to at least one other EPG

Correct answer: A. The strict 1 VLAN : 1 BD : 1 EPG mapping is the definition of network-centric mode — the fabric reproduces the traditional VLAN model while remaining API-driven. Modelling application tiers as EPGs with inter-tier contracts is application-centric mode, the opposite design. Disabling unicast routing is a legitimate L2-only choice but not definitional, and network-centric designs usually reduce contracts rather than mandate them.

Q4. What is the function of a contract (`vzBrCP`) in the ACI policy model?

A. It reserves a range of VLAN identifiers that a physical domain may use for encapsulation

B. It associates a bridge domain with the VRF that supplies its routing table

C. It defines which traffic is permitted between EPGs, with one EPG providing it and another consuming it

D. It binds an EPG to a specific leaf switch, port and encapsulation VLAN

Correct answer: C. A contract is the policy construct governing inter-EPG traffic, and it is directional by role: the provider offers the service (the server side), the consumer initiates toward it (the client side). VLAN reservation is a VLAN pool (`fvnsVlanInstP`); BD-to-VRF association is the `fvRsCtx` relation; port binding is `fvRsPathAtt`.

Q5. An automation script must add a bridge domain named BD-20 to the tenant PROD via the APIC REST API. Which distinguished name identifies the new object?

- **A.** uni/BD-BD-20/tn-PROD
- **B.** topology/pod-1/tn-PROD/BD-BD-20
- **C.** uni/tn-PROD/ctx-BD-20
- **D.** uni/tn-PROD/BD-BD-20

Correct answer: D. A DN is the concatenation of relative names from the configurable root `uni` downward, so a bridge domain inside tenant PROD is `uni/tn-PROD/BD-BD-20` — the `BD-` prefix is the class's naming convention, followed by the object's name. Reversing the containment is invalid; `topology` is the root of the physical inventory rather than the policy tree; and `ctx-` is the VRF's prefix, not the bridge domain's.

END OF SAMPLE

DCNAUTO 300-635

Complete Learning Guide

You have read topics 1.1 and 1.2. The full book continues with:

- The remaining 24 topics — all five exam domains
- 26 original diagrams, drawn for this book
- 130 knowledge checks, five per sub-section
- A 60-question practice exam, timed to 90 minutes
- 13 labs, every one runnable free
- A 69-term glossary built around exam distinctions

303 pages · 26 diagrams · 190 questions

Read the rest at

leanpub.com/dcnauto

An independent, unofficial study guide. Not endorsed by Cisco Systems, Inc.