

Automation coding, system configuration, continuous integration, and continuous delivery, one exam objective at a time.



CompTIA AutoOps+ AT0-001 Course Notes: Domain 2.0: System Configuration

Written by Frazier A. Smith, Ed.D.

Central Piedmont Community College, Charlotte, North Carolina

<https://vfrazier.app>

Version 1.0.0, October 2026

License

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License (CC BY-NC-SA 4.0). You may copy, adapt, and share it for noncommercial purposes with attribution, provided adaptations carry the same license. <https://creativecommons.org/licenses/by-nc-sa/4.0/>

Source files, figures, and the scripts that test every command in these notes: <https://github.com/frazier-at-cpcc/autoops-course-notes>

Trademark Acknowledgments

CompTIA and AutoOps+ are trademarks or registered trademarks of CompTIA, Inc. These notes are an independent study resource and are not produced, endorsed, or affiliated with CompTIA. All other product names and trademarks are the property of their respective owners.

The objective numbers and titles referenced throughout follow the CompTIA AutoOps+ AT0-001 V1 exam objectives. Download the official objectives from CompTIA and use them as your checklist; they are the final word on what the exam covers.

Warning and Disclaimer

These notes are provided to help you prepare for the CompTIA AutoOps+ certification exam. They may contain errors. Use them as a study companion, not as the sole source of information. The author accepts no liability for any loss or damage arising from their use.

Contents

2.0

System Configuration 25% of exam · page 1

2.1

Drift Detection and State Management

2

2.1

Workload Configuration: Certificates and ACLs

4

2.1

Virtual Network Infrastructure as Code

7

2.2

Remote and Local Automation

10

2.2

Environments, Migrations, and Provisioning

12

2.2

Event-Driven Automation

15

2.3

REST APIs: Methods, Headers, and Status Codes

17

2.3

Working with REST APIs: curl, wget, and Postman

21

2.4

Troubleshooting System Configuration

24

2.4

Configuration File Syntax: INI, YAML, and JSON

28

How to Use These Notes

60

QUESTIONS, MAX

60

MINUTES

600

PASSING (100–900)

V1

ATO-001, JUNE 2026

The exam mixes multiple-choice and performance-based questions. These notes follow the four exam domains in order, and every topic is labeled with the objective number it covers, so you can line them up against the official objectives, your course modules, and your lab work.

1.0 Automation Coding Concepts		31%
2.0 System Configuration		25%
3.0 Continuous Integration		24%
4.0 Continuous Delivery		20%

Reading the page

Highlighted terms are the vocabulary the exam expects you to recognize on sight. If a highlighted term is unfamiliar a week before your exam, go back to that page.

On the exam boxes describe how a topic tends to show up in a question: the scenario wording, the distractor answers, and the distinction being tested.

Try it boxes are short exercises you can run on your own machine or in a lab VM with the tools the box names. Do them. AutoOps+ is a hands-on exam, and reading about a command is not the same as running it.

Every command was run

The commands, code samples, and terminal output in these notes come from real runs, not from memory. The source repository keeps a test script for each topic that runs them again, so they can be rechecked whenever a tool releases a new version.

02

System Configuration

25% of the exam

-
- 2.1** Given a scenario, use configuration management techniques to deploy a solution
-
- 2.2** Compare and contrast basic approaches to automation
-
- 2.3** Given a scenario, interact with RESTful systems to perform create, read, update, and delete (CRUD) operations
-
- 2.4** Given a scenario, troubleshoot common issues with system configuration
- 

■ Desired state and drift

- Configuration management tools describe the **desired state**: what the switch, server, or cloud network should look like
 - The tool compares that description to the **current state** and changes only what differs
- Drift** is any gap between the code and what is really running
 - An engineer fixes an outage by hand at 2 a.m. and never updates the repo
 - A console click, a vendor support session, a script outside the pipeline
 - A failed or half-finished apply
- Drift means the code is no longer the truth
 - The next apply silently undoes the hand fix, and a rebuild from code no longer matches production
- Drift detection and remediation** is the loop: compare often, report the gap, then put code and reality back in agreement

■ Terraform state

- Terraform keeps a **state file** (`terraform.tfstate`): a JSON map from each resource address in the code to the real object it created
 - `aws_vpc.lab` in code maps to `vpc-0a1b2c...` in AWS
 - Records every attribute Terraform knows, plus dependencies and outputs
- State is how Terraform knows what it owns
 - Without it, Terraform would try to create everything again
 - `terraform state list` and `terraform state show <address>` read it; never hand-edit the JSON

TERMINAL

```
$ terraform state list
local_file.motd
$ jq '{version, serial, lineage}' terraform.tfstate
{
  "version": 4,
  "serial": 1,
  "lineage": "9023d652-1ee4-6b76-e92d-3c3d40c3c323"
}
```

- `serial` goes up on every write; `lineage` identifies this state's history, so Terraform refuses to overwrite it with an unrelated state

- State holds **secrets in plain text**: database passwords, private keys generated by the `tls` provider
 - Never commit it: add `*.tfstate` and `*.tfstate.*` to `.gitignore`
 - Store it encrypted, with access limited to the pipeline and a few admins

■ Remote backends and locking

- A **backend** decides where state lives
 - Default **local** backend: a file on one laptop; fine for a lab, wrong for a team
 - Remote backends**: Amazon S3, Azure Blob Storage (`azurerm`), Google Cloud Storage (`gcs`), HCP Terraform, Consul, PostgreSQL (`pg`)
- A remote backend gives one shared copy, encryption at rest, versioning for rollback, and access control
- State locking** stops two runs from writing at once
 - The first **plan** or **apply** takes the lock; anyone else waits or fails
 - S3: `use_lockfile = true` writes a `.tflock` object beside the state (Terraform 1.10 and later)
 - The older DynamoDB lock table (`dynamodb_table`) is deprecated

BACKEND.TF

```
terraform {
  backend "s3" {
    bucket      = "netops-tfstate"
    key         = "core/motd.tfstate"
    region      = "us-east-1"
    encrypt     = true
    use_lockfile = true
  }
}
```

- Settings that differ per environment can stay out of the file: `terraform init -backend-config=...` (partial configuration)
- While one apply runs, a second engineer's plan is refused

SECOND ENGINEER, DURING AN APPLY

```
$ terraform plan
Error: Error acquiring the state lock
Lock Info:
  Path:      netops-tfstate/core/motd.tfstate
  Operation: OperationTypeApply
  Who:       root@vm
```

- Wait for the other run; never reach for `-lock=false`

2.1 Drift Detection and State Management (continued)

- A crashed run can leave a stale lock: confirm nobody is running, then `terraform force-unlock <LOCK_ID>`

■ Detecting drift

- `terraform plan` refreshes state from the real infrastructure, then compares it to the code

- Detailed exit codes:** `-detailed-exitcode` turns the answer into something a pipeline can act on
 - `0`: no changes; `2`: changes pending (drift or new code); `1`: error

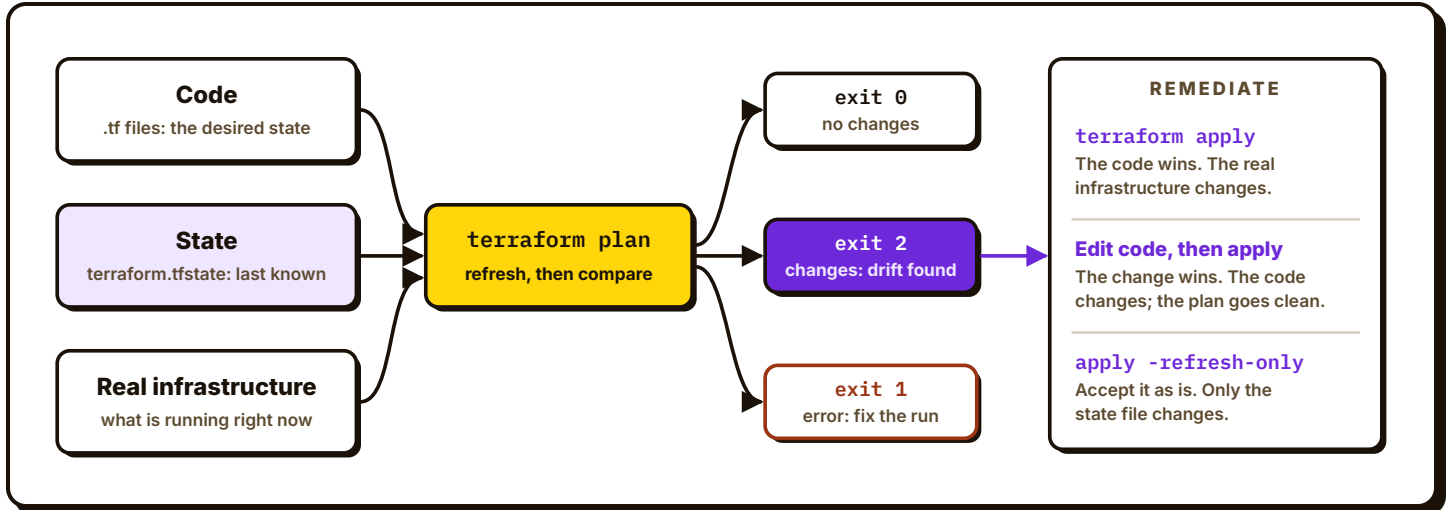


FIGURE Plan reads the real infrastructure, compares it to the code through state, and reports with an exit code. Each remediation moves a different side.

- Real drift: someone edits a file that Terraform manages

```
TERMINAL
$ echo "Free Wi-Fi password: guest123" > motd.txt
$ terraform plan -detailed-exitcode
# local_file.motd will be created
Plan: 1 to add, 0 to change, 0 to destroy.
$ echo $?
2
```

- The `local` provider treats changed content as a missing file, so the fix shows as a create; a cloud provider would usually show `~ update in-place`
- Schedule it: a nightly pipeline job runs the plan and alerts (or opens a ticket) on exit code 2
 - Some platforms do this for you: HCP Terraform health assessments, Spacelift, env0

■ Remediating drift

- First decide which side is right: the code or the change someone made

SITUATION	REMEDIATION	COMMAND
The hand change was wrong	Re-apply the code; it overwrites the drift	<code>terraform apply</code>
The change was right	Copy it into the code, review, then apply (plan shows no changes)	<code>edit .tf</code> , <code>terraform apply</code>
Accept reality into state without changing anything	Refresh-only run updates state to match the real objects	<code>terraform apply -refresh-only</code>
Someone built a resource outside Terraform	Bring it under management	<code>import</code> block, or <code>terraform import</code>
Terraform should stop managing a resource	Remove it from state and leave it running	<code>removed</code> block, or <code>terraform state rm</code>

- `terraform plan -refresh-only` lists "Objects have changed outside of Terraform" and proposes no fixes
 - Replaces the deprecated `terraform refresh`, which updated state without review
- `import` blocks (Terraform 1.5 and later) put imports in code review; `terraform plan -generate-config-out=generated.tf` writes a starting configuration

2.1 Drift Detection and State Management (continued)

- Prevent repeat drift: remove console write access, make the pipeline the only path to production

■ Configuration drift with Ansible

- Ansible has no state file; it checks the target every run and acts only if the target differs (idempotent modules)
- `--check` is a dry run: report what would change, change nothing
- `--diff` shows the exact lines; together they are a drift report

NTP.YML

```
- name: Enforce the NTP baseline
  hosts: localhost
  connection: local
  gather_facts: false
  tasks:
    - name: Chrony points at the site NTP servers
      ansible.builtin.copy:
        dest: chrony.conf
        mode: "0644"
        content: |
          server 10.0.0.1 iburst
          server 10.0.0.2 iburst
          makestep 1.0 3
```

AFTER A HAND EDIT

```
$ ansible-playbook ntp.yml --check --diff
TASK [Chrony points at the site NTP servers] ***
--- before: chrony.conf
+++ after: chrony.conf
@@ -1,3 +1,3 @@
  server 10.0.0.1 iburst
- server 192.0.2.99 iburst
+ server 10.0.0.2 iburst
  makestep 1.0 3

changed: [localhost]
localhost : ok=1    changed=1
```

- `changed=1` in check mode means drift; the file is still untouched
 - The exit code is still 0, so a pipeline must read the recap (or a JSON callback), unlike Terraform's exit code 2
- Remediate by running the playbook normally
- Agent-based tools correct drift on a schedule: a Puppet agent run every 30 minutes by default

ON THE EXAM

- Exit codes for `terraform plan -detailed-exitcode`: 0 means no changes, 2 means changes (drift), 1 means an error. A scenario about "alert when production no longer matches code" wants this flag in a scheduled job.
- State locking prevents two concurrent applies from corrupting state. Remote state in a shared, encrypted, locked backend is the answer for teams; committing `terraform.tfstate` to Git is always a wrong answer.
- Drift remediation depends on which side is correct: re-apply to undo the change, update the code to keep it, refresh-only to accept it into state, import to adopt an unmanaged resource.

2.1 Workload Configuration: Certificates and ACLs

2.0 SYSTEM CONFIGURATION

■ Workload configuration

- **Workload configuration** is the settings on the workload itself: which certificate a server presents, who may read its key, what traffic a router permits
- Manage it like infrastructure: in Git, applied by a tool, checked for drift
 - A certificate renewed by hand on one node, or an ACL line typed in during an outage, is drift
- The objective names two: **certificates** and **access control lists (ACLs)**

■ Certificate building blocks

- A **certificate authority (CA)** signs certificates; clients trust whatever chains back to a CA they already trust
 - Public CAs for internet-facing names; an internal CA (or HashiCorp Vault PKI) for lab and private names
- **Chain of trust**: server (leaf) certificate, signed by an intermediate, signed by the root
 - The server sends its own certificate plus the intermediates; the client already holds the root

2.1 Workload Configuration: Certificates and ACLs (continued)

- **Certificate signing request (CSR)**: the public key and the requested names, sent to the CA; the private key never leaves the server
- **Subject Alternative Name (SAN)**: the DNS names and IPs the certificate is valid for
 - Clients check the SAN, not the CN; a missing SAN fails validation
- Validity dates: **notBefore** and **notAfter**
 - Public TLS certificates: 200 days maximum since March 15, 2026; 100 days from March 2027; 47 days from March 2029

■ Creating a CA and server certificate

LAB CA AND SERVER CERTIFICATE

```
# 1. Root CA: key and self-signed cert, 10 years
openssl req -x509 -newkey ec \
  -pkeyopt ec_paramgen_curve:P-256 \
  -nodes -keyout ca.key -out ca.crt -days 3650 \
  -subj "/CN=Lab Root CA"

# 2. Server key and CSR with a SAN
openssl req -newkey ec \
  -pkeyopt ec_paramgen_curve:P-256 \
  -nodes -keyout web.key -out web.csr \
  -subj "/CN=web01.lab.example" \
  -addext "subjectAltName=DNS:web01.lab.example"

# 3. The CA signs the CSR: 90-day certificate
openssl x509 -req -in web.csr -CA ca.crt \
  -CAkey ca.key -CAcreateserial -days 90 \
  -copy_extensions copy -out web.crt
```

- Inspect before you deploy

TERMINAL

```
$ openssl verify -CAfile ca.crt web.crt
web.crt: OK
$ openssl x509 -in web.crt -noout -enddate
notAfter=Jan 17 13:02:54 2027 GMT
$ openssl x509 -in web.crt -noout -checkend 2592000
Certificate will not expire
$ openssl x509 -in web.crt -noout -checkend 8640000
Certificate will expire
```

- **-checkend <seconds>** exits 0 if still valid that far ahead, 1 if not; 2592000 seconds is 30 days, so a renewal job acts on exit code 1
- **openssl verify -verify_hostname db01.lab.example** fails with "hostname mismatch": the SAN does not cover that name

■ Deployment and renewal automation

- Certificates as code: the Terraform **tls** provider builds the same CA and server certificate (**tls_private_key**, **tls_self_signed_cert**, **tls_cert_request**, **tls_locally_signed_cert**); the **local.*** values point at the CA key, CA certificate, and CSR

WEB.TF (EXCERPT)

```
resource "tls_locally_signed_cert" "web" {
  cert_request_pem = local.web_csr
  ca_private_key_pem = local.ca_key
  ca_cert_pem      = local.ca_cert

  validity_period_hours = 2160 # 90 days
  early_renewal_hours   = 720  # 30 days left
  allowed_uses = [
    "digital_signature",
    "server_auth",
  ]
}
```

TERMINAL

```
$ terraform output web_ready_for_renewal
false
```

- When 30 days remain, **ready_for_renewal** turns true and the next **terraform apply** replaces the certificate
- The generated private keys are stored in the state file in plain text; this is a lab pattern, and state must be protected

TOOL	WHERE	HOW RENEWAL HAPPENS
certbot and other ACME clients	Linux web servers	timer runs certbot renew , then reloads the service
cert-manager	Kubernetes	controller renews Certificate resources into Secrets
AWS Certificate Manager	load balancers, CloudFront	managed renewal of ACM-issued certificates
Vault PKI	internal services	short-lived certificates issued on demand

- **ACME** (RFC 8555) is the protocol Let's Encrypt uses: prove control of the name, receive a certificate, repeat before expiry

■ File ACLs

- Mode bits allow one owner, one group, and everyone else; a file **ACL** adds entries for more users or groups without changing ownership
 - Scenario: the **www-data** group must read one private key that **root** owns with mode **0600**

2.1 Workload Configuration: Certificates and ACLs (continued)

TERMINAL

```
$ chmod 600 tls/web.key
$ setfacl -m g:www-data:r tls/web.key
$ getfacl -p tls/web.key
# file: tls/web.key
user::rw-
group:---
group:www-data:r--
mask::r--
other:---
```

- `ls -l` shows a `+` after the mode (`-rw-r-----+`) when an ACL exists
- The **mask** caps what named users and groups can get; `chmod` on the group bits changes the mask

- `setfacl -d -m g:www-data:r tls` sets a **default ACL**: new files in `tls/` inherit the entry
- `setfacl -x g:www-data` removes one entry, `setfacl -b` all of them; as code: Ansible `ansible.posix.acl`; Windows NTFS ACLs with `icacls`, `Set-Acl`, or `ansible.windows.win_acl`

Network ACLs as code

- A network ACL is an ordered rule list that permits or denies traffic by address, protocol, and port
 - Evaluated top down; the **first match** wins
 - Anything not matched hits the **implicit deny** at the end

FILTER	STATEFUL?	RULES	DEFAULT
Router ACL (Cisco IOS)	No: return traffic needs its own rule	permit and deny, numbered, first match wins	implicit deny
AWS network ACL (subnet)	Stateless : allow ephemeral ports 1024-65535 back	allow and deny, lowest rule number first	default NACL allows all; custom NACL denies all
AWS security group (interface)	Stateful : replies are allowed automatically	allow only, all rules evaluated	deny all inbound

- Keep the rules as data, render the device syntax from a template

ACL_VARS.YML

```
acl_name: WEB-IN
web: host 10.20.1.10
rules:
  - seq: 10
    action: permit
    src: any
    port: 443
  - seq: 20
    action: permit
    src: host 10.99.0.5
    port: 22
```

ACL.J2

```
ip access-list extended {{ acl_name }}
{% for r in rules %}
  {{ r.seq }} {{ r.action }} tcp {{ r.src }}
  {{- " " ~ web ~ " eq " ~ r.port }}
{% endfor %}
1000 deny ip any any log
```

- An `ansible.builtin.template` task writes `WEB-IN.cfg` (`{{- }}` joins the two template lines); the explicit last rule logs what the implicit deny would drop silently
- Add rule 30 (HTTP) to the data, then preview the change before it reaches the router

TERMINAL

```
$ ansible-playbook acl.yml --check --diff
@@ -1,4 +1,5 @@
ip access-list extended WEB-IN
  10 permit tcp any host 10.20.1.10 eq 443
  20 permit tcp host 10.99.0.5 host 10.20.1.10 eq 22
+ 30 permit tcp any host 10.20.1.10 eq 80
  1000 deny ip any any log
```

- Push with the `cisco.ios.ios_acls` module; **state: overridden** also deletes lines not in the data, which fixes ACL drift

ON THE EXAM

- Expiry questions: `openssl x509 -checkend` or `-enddate` finds certificates about to expire; the fix in a scenario is automated renewal (ACME, cert-manager, ACM, or a pipeline job), not a calendar reminder. A browser error on one hostname usually means that name is missing from the SAN.
- Give one more group access to one file without changing its owner or group: `setfacl`, not `chmod 777` or `chown`.
- Network ACLs are ordered and end in an implicit deny; a new permit placed after a broader deny never matches. NACLs are stateless; security groups are stateful.

2.1 Virtual Network Infrastructure as Code

2.0 SYSTEM CONFIGURATION

■ Networks defined in code

- **Virtual network infrastructure configuration**: subnets, routers, route tables, load balancers, firewalls, and DNS declared in files, not clicked together in a console
 - Reviewed, versioned, and drift-checked like any code; a second region is the same code with new variables

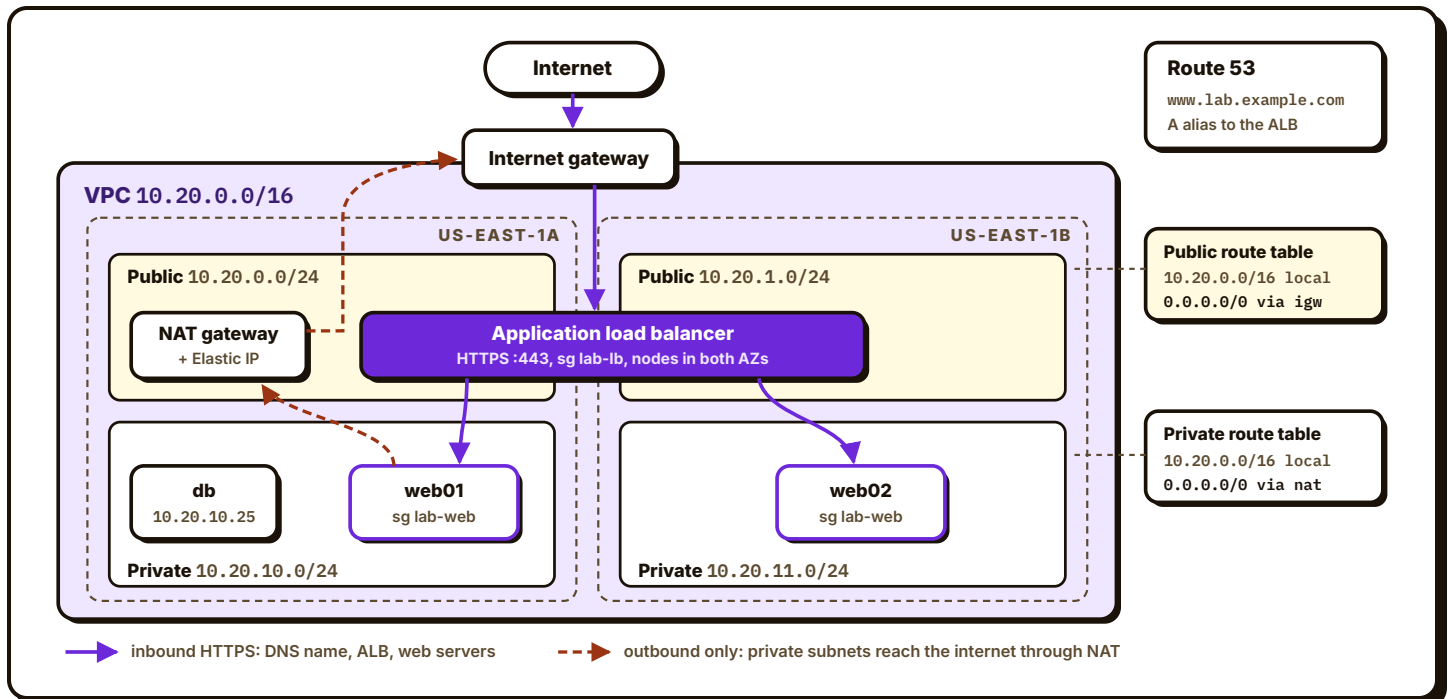


FIGURE The example in this topic: two availability zones, public subnets for the load balancer and NAT gateway, private subnets for servers, one route table per tier, and a DNS alias in front.

■ Subnets

- A **VPC** (virtual private cloud) gets one CIDR block; **subnets** carve it up, one subnet per availability zone per tier
 - **Public subnet**: its route table sends **0.0.0.0/0** to an internet gateway
 - **Private subnet**: no route from the internet; outbound only, through NAT
- AWS keeps 5 addresses in every subnet (the first four and the last), so a **/24** has 251 usable

TERRAFORM CONSOLE

```
> cidrsubnet("10.20.0.0/16", 8, 1)
"10.20.1.0/24"
> cidrsubnet("10.20.0.0/16", 8, 10)
"10.20.10.0/24"
> cidrhost("10.20.10.0/24", 25)
"10.20.10.25"
```

2.1 Virtual Network Infrastructure as Code (continued)

NETWORK.TF (EXCERPT)

```
locals {
  azs      = ["us-east-1a", "us-east-1b"]
  public   = ["10.20.0.0/24", "10.20.1.0/24"]
  private  = ["10.20.10.0/24", "10.20.11.0/24"]
}

resource "aws_subnet" "public" {
  count          = 2
  vpc_id         = aws_vpc.lab.id
  availability_zone = local.azs[count.index]
  cidr_block     = local.public[count.index]

  map_public_ip_on_launch = true
}
```

■ Routers and route tables

- Every VPC has an implicit **router**; you never deploy it, you program it with **route tables**
 - Each table starts with a **local** route for the VPC CIDR
 - Each subnet is associated with exactly one route table
- Gateways are the next hops
 - **Internet gateway**: two-way internet access for public subnets
 - **NAT gateway**: sits in a public subnet with an Elastic IP; lets private subnets start outbound connections (patches, APIs) while blocking inbound ones

ROUTING.TF (EXCERPT)

```
resource "aws_route_table" "private" {
  vpc_id = aws_vpc.lab.id
  route {
    cidr_block     = "0.0.0.0/0"
    nat_gateway_id = aws_nat_gateway.nat.id
  }
}

resource "aws_route_table_association" "public" {
  count          = 2
  subnet_id     = aws_subnet.public[count.index].id
  route_table_id = aws_route_table.public.id
}
```

■ Firewalls and security groups

- A **security group** is a stateful firewall attached to an interface (instance, load balancer, database)
 - Allow rules only; anything not allowed is dropped; replies return automatically
 - A rule can name another security group as its source instead of an address range

- The network ACL on the subnet is the stateless second layer (see the ACL table in the workload configuration topic)

FIREWALL.TF (EXCERPT)

```
resource "aws_security_group" "web" {
  name = "lab-web"
  vpc_id = aws_vpc.lab.id
  ingress {
    protocol = "tcp"
    from_port = 80
    to_port = 80
    security_groups = [aws_security_group.lb.id]
  }
}
```

- Only the load balancer can reach the web servers on port 80; no address ranges to keep current
- Large rule sets: one **aws_vpc_security_group_ingress_rule** resource per rule

■ Load balancers

- A **load balancer** spreads traffic across healthy targets and gives clients one stable name
 - **Application load balancer** (layer 7): HTTP and HTTPS, path and host routing, TLS termination
 - **Network load balancer** (layer 4): TCP, UDP, and TLS at very high rates, static IPs
- Three parts in code
 - **aws_lb**: **load_balancer_type** = "application", **subnets** = **aws_subnet.public[*].id** (a splat: every public subnet ID)
 - **aws_lb_target_group**: the servers plus a **health check** (**path** = "/healthz"); unhealthy targets get no traffic
 - **aws_lb_listener**: port 443 with the certificate (**certificate_arn**), forwarding to the target group

■ DNS records as code

- **Public hosted zone**: names on the internet; **private hosted zone**: names that resolve only inside the attached VPCs
- An **alias record** points a name at an AWS resource; it follows the load balancer's changing IPs and works at the zone apex

2.1 Virtual Network Infrastructure as Code (continued)

DNS.TF (EXCERPT)

```
resource "aws_route53_record" "www" {
  zone_id = aws_route53_zone.public.zone_id
  name    = "www.lab.example.com"
  type    = "A"
  alias {
    name            = aws_lb.web.dns_name
    zone_id         = aws_lb.web.zone_id
    evaluate_target_health = true
  }
}
```

- The internal record is a plain A record:
`db.lab.internal`, TTL 300, `10.20.10.25`
- The same building blocks exist in every cloud; only the resource names change

OBJECTIVE TERM	AWS (TERRAFORM RESOURCE)	AZURE (TERRAFORM RESOURCE)
Virtual network	<code>aws_vpc</code>	<code>azurearm_virtual_network</code>
Subnets	<code>aws_subnet</code>	<code>azurearm_subnet</code>
Routers	implicit VPC router, <code>aws_internet_gateway</code> , <code>aws_nat_gateway</code>	system routes, <code>azurearm_nat_gateway</code>
Route tables	<code>aws_route_table</code> , <code>aws_route_table_association</code>	<code>azurearm_route_table</code>
Load balancers	<code>aws_lb</code> (application or network), <code>aws_lb_target_group</code> , <code>aws_lb_listener</code>	<code>azurearm_lb</code> , <code>azurearm_application_gateway</code>
Firewalls	<code>aws_security_group</code> , <code>aws_network_acl</code>	<code>azurearm_network_security_group</code> , <code>azurearm_firewall</code>
DNS	<code>aws_route53_zone</code> , <code>aws_route53_record</code>	<code>azurearm_dns_zone</code> , <code>azurearm_private_dns_zone</code>

■ Checking it without an AWS account

- `terraform validate` checks syntax, types, and references; it needs the provider but no credentials
- `terraform plan` must call the AWS API, so without credentials it stops

TERMINAL

```
$ terraform init -input=false
$ terraform validate
Success! The configuration is valid.
$ terraform plan -var-file=lab.tfvars
Error: No valid credential sources found
```

- Validate also catches broken references:
`aws_subnet.public.id` on a counted resource fails with `Error: Missing resource instance key`; the fix is an index such as `[0]`

- Run `terraform fmt -check` and `terraform validate` in CI on every pull request; save `plan` and `apply` for the stage that holds cloud credentials

ON THE EXAM

- Match the term to the job: a route table decides the next hop, a security group filters traffic to an interface, a NAT gateway gives private subnets outbound-only internet access, an internet gateway makes a subnet public.
- A scenario where instances in a private subnet cannot download updates points at the private route table: it is missing `0.0.0.0/0` to a NAT gateway. Adding an internet gateway route would make the subnet public instead.
- `terraform validate` needs no cloud credentials; `plan` and `apply` do.

2.2 Remote and Local Automation

2.0 SYSTEM CONFIGURATION

Where the code runs

- **Local** automation runs on the machine it changes
 - A script on a server, a scheduled task on a workstation, a playbook with `connection: local`
- **Remote** automation runs from a control node and reaches targets over the network
 - One controller, hundreds of switches and servers
 - Needs a transport, credentials, and a firewall path to every target
- Three ways to reach a remote target

APPROACH	TRANSPORT AND PORT	NEEDS ON THE TARGET
Secure Shell (SSH)	TCP 22, key-based login	<code>sshd</code> ; Python for most Ansible modules
Windows Remote Management (WinRM)	TCP 5985 (HTTP) or 5986 (HTTPS)	WinRM listener and PowerShell
Agent-based	Agent calls the server, such as Puppet on TCP 8140	Installed agent and its certificate
Local	None	The script and its interpreter

Remote over SSH

- The default remote channel for Linux, Unix, and network gear
 - Encrypted, with **key-based authentication**; no password typed
- `BatchMode yes` makes `ssh` fail instead of prompting
 - A script waiting on a password prompt hangs the pipeline
- A per-project `ssh_config` keeps hosts, ports, and keys out of the command line

SSH_CONFIG

```
Host lab-srv1
  HostName 127.0.0.1
  Port 18250
  User netops
  IdentityFile ./id_ed25519
  BatchMode yes
  StrictHostKeyChecking accept-new
  UserKnownHostsFile ./known_hosts
```

TERMINAL

```
$ ssh -F ssh_config lab-srv1 'uname -n; id -un'
lab-srv1
netops
```

- Ansible is **agentless**: it rides the same SSH connection
 - Copies a small module to the target, runs it with Python, deletes it
 - Network devices use `network_cli` instead: the module runs on the controller and sends CLI commands over SSH

INVENTORY.INI

```
[linux]
lab-srv1 ansible_host=127.0.0.1 ansible_port=18250

[linux:vars]
ansible_user=netops
ansible_connection=ssh
ansible_python_interpreter=/usr/bin/python3
```

TERMINAL

```
$ ansible linux -i inventory.ini -m ping
lab-srv1 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
$ ansible linux -i inventory.ini -a "uname -n"
lab-srv1 | CHANGED | rc=0 >>
lab-srv1
$ ansible linux -i inventory.ini -m ping -vvv
<127.0.0.1> ESTABLISH SSH CONNECTION FOR USER:
netops
PUT ../tmpm30q4kv1 TO ../AnsiballZ_ping.py
```

- `-m ping` is not ICMP
 - It proves the login works and the target has a usable Python

2.2 Remote and Local Automation (continued)

Remote over WinRM

- WinRM is Microsoft's implementation of the WS-Management standard
 - SOAP messages over HTTP; PowerShell Remoting rides on it
 - 5985** is HTTP, **5986** is HTTPS
 - On 5985, Kerberos or NTLM still encrypts the message body; Basic auth does not, so use 5986 with a certificate
- Turn it on with `Enable-PSRemoting` (or `winrm quickconfig`)
- PowerShell Remoting**: `Invoke-Command -ComputerName win-dc1` runs a script block remotely; `Enter-PSSession` opens an interactive shell
- Ansible: `ansible_connection: winrm` (or `psrp`), plus the `ansible.windows` modules
- Current Windows ships an OpenSSH server, so SSH to Windows is also an option

WINDOWS.YML

```
windows:
  hosts:
    win-dc1:
      ansible_host: win-dc1.corp.example.com
  vars:
    ansible_connection: winrm
    ansible_port: 5986
    ansible_winrm_scheme: https
    ansible_winrm_transport: kerberos
```

- Kerberos needs the host name, not an IP address, and a domain-joined target
- No Windows host here: the test only checks this inventory with `ansible-inventory`

Agent-based automation

- An **agent** is a service installed on every managed node
 - It connects out to a server, fetches its configuration, applies it, and reports back
 - No inbound port on the node; works behind NAT

TOOL	AGENT	TALKS TO	DEFAULT RHYTHM
Puppet	<code>puppet agent</code>	Puppet server, TCP 8140	Every 30 minutes
Chef Infra	<code>chef-client</code>	Chef Infra Server, HTTPS	Every 30 minutes as a service
Salt	<code>salt-minion</code>	Salt master, TCP 4505 and 4506	Persistent link; master sends jobs at once
AWS Systems Manager	<code>amazon-ssm-agent</code>	SSM endpoints, HTTPS 443	Polls for commands

- Costs: install and patch the agent, manage its certificates, CPU and memory on every node
- Gains: continuous **drift** correction, no credentials for every node on one controller
- Agentless costs the opposite: inbound access and stored credentials, but nothing to install

TERMINAL

```
$ ansible localhost -c local -m ping
localhost | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
```

Local: command-line scripts

- A **command-line script** runs where it lives: Bash, PowerShell, or Python
 - Shebang** line (`#!/usr/bin/env bash`), `chmod +x`, exit code 0 for success
 - Started by hand, by `cron` or a systemd timer, or by Task Scheduler on Windows
- Remote tools often deliver local scripts
 - Ansible's `script` module copies a script to the target and runs it there
- Ansible can also run locally: `-c local` skips SSH entirely
 - Used for cloud and controller APIs, where the "target" is an HTTPS endpoint

Foreground and background

- Foreground**: the shell waits and the job owns the terminal
 - Ctrl+C stops it; Ctrl+Z suspends it
- Background**: `&` returns the prompt at once
 - The shell prints a job number and a PID
- `jobs` lists them; `fg %n` and `bg %n` move them; `kill %n` ends them
 - `kill -STOP %1` below does what Ctrl+Z does at the keyboard

2.2 Remote and Local Automation (continued)

TERMINAL

```
$ ./poll-switches.sh &
[1] 17895
$ jobs
[1]+  Running                  ./poll-switches.sh &
$ kill -STOP %1
[1]+  Stopped                  ./poll-switches.sh
$ bg %1
[1]+ ./poll-switches.sh &
$ sleep 1 &
[2] 18038
$ fg %2
sleep 1
$ kill %1
[1]+  Terminated              ./poll-switches.sh
$ nohup ./poll-switches.sh >poll.log 2>&1 &
[1] 18079
$ disown %1
$ jobs
```

- Background jobs die with the terminal unless protected
 - **nohup** ignores the hangup signal sent at logout
 - **disown** removes the job from the shell's table
 - Anything that must always run belongs in a systemd service, not a shell job
- PowerShell: **Start-Job**, **Get-Job**, **Receive-Job**
 - **Invoke-Command -AsJob** runs remote work in the background
 - PowerShell 7 also accepts **&** at the end of a pipeline

BACKUP-JOB.PS1

```
Start-Job -Name backup -ScriptBlock {
    Start-Sleep -Seconds 2
    "Saved 14 configs"
} | Out-Null
Get-Job | Format-Table Id, Name, State
Receive-Job -Name backup -Wait -AutoRemoveJob
```

TERMINAL

```
$ pwsh -File backup-job.ps1
Id Name      State
-- --
1 backup Running

Saved 14 configs
```

ON THE EXAM

- Match the port to the protocol: SSH is 22, WinRM is 5985 for HTTP and 5986 for HTTPS. "Manage Windows hosts remotely with no agent" points to WinRM (or SSH on newer Windows).
- Agent-based means software on every node that checks in with a server; agentless (Ansible) means the controller connects in over SSH or WinRM. A node behind NAT with no inbound access suits an agent.
- A long job that must survive logout needs **nohup** or a service; **&** alone only frees the prompt.

TRY IT

Start **sleep 300 &** and run **jobs -l**. Bring it forward with **fg**, suspend it with **Ctrl+Z**, resume it with **bg**, and end it with **kill %1**, running **jobs -l** after each step to watch the state change while the PID stays the same.

2.2 Environments, Migrations, and Provisioning

2.0 SYSTEM CONFIGURATION

■ Two ways to write the automation

- **Imperative**: you write the steps and their order
 - Bash and PowerShell scripts, database migrations, a runbook turned into code
 - The script does exactly what it says, even when the step is no longer needed
- **Declarative**: you write the end state; the tool works out the steps
 - Ansible tasks, Terraform, Kubernetes manifests, Compose files
 - Run it again and only the differences change
- Objective 1.3 covers the IaC theory (state, idempotency); here the question is which to reach for
 - Imperative: one-time changes, strict ordering, glue between tools
 - Declarative: configuration that must stay in a known state
- Most real automation mixes the two
 - A playbook is an ordered list (imperative) of declarative tasks

2.2 Environments, Migrations, and Provisioning (continued)

Script writing

- **Script writing** is the imperative approach done carefully
 - Fail fast: `set -euo pipefail` stops on an error, an unset variable, or a broken pipe
 - Check before you change, so a second run is harmless
 - Print what changed, and exit 0 only on success; cron, CI, and Ansible trust the **exit code**

BOOTSTRAP-NODE.SH

```
#!/usr/bin/env bash
# Imperative: steps run in the order written
set -euo pipefail
conf="${NODE_ROOT:-}/etc/chrony.sources"
mkdir -p "$(dirname "$conf")"
if grep -qs '^server 10.0.20.5' "$conf"; then
    echo "ntp: already set, no change"
else
    echo "server 10.0.20.5 iburst" >> "$conf"
    echo "ntp: added 10.0.20.5"
fi
```

TERMINAL

```
$ ./bootstrap-node.sh
ntp: added 10.0.20.5
$ ./bootstrap-node.sh
ntp: already set, no change
```

- Every guard is code to write and test; the declarative **copy** task in `local.yml` (below) needs no `if`

Migrations

- A schema **migration** is a small, versioned script that changes a database one step
 - Applied in order, each once; never edit one that has run, add a new one
 - An **up** section and, where possible, a **down** to roll back
 - The tool records what ran in a table (`schema_migrations`)
 - Flyway, Liquibase, Alembic, dbmate, golang-migrate, Django, Rails
- Migrations are imperative; **state-based** tools (Atlas, SSDT) diff a desired schema against the live one, the declarative style

20261015090000_ADD_SITE.SQL

```
-- migrate:up
ALTER TABLE devices ADD COLUMN site TEXT
    NOT NULL DEFAULT 'CLT1';

-- migrate:down
ALTER TABLE devices DROP COLUMN site;
```

TERMINAL

```
$ export DATABASE_URL=sqlite:db/net.db
$ dbmate status
[X] 20261001090000_create_devices.sql
[X] 20261008090000_add_mgmt_ip.sql
[ ] 20261015090000_add_site.sql

Applied: 2
Pending: 1
$ dbmate up
Applying: 20261015090000_add_site.sql
Applied: 20261015090000_add_site.sql in 1.368997ms
$ sqlite3 db/net.db 'SELECT * FROM
schema_migrations'
20261001090000
20261008090000
20261015090000
```

- **dbmate rollback** runs the newest **down** section; timestamped names keep two branches from claiming the same version
- Infrastructure has migrations too: Terraform **moved** blocks relocate resources in state without rebuilding them

Push and pull provisioning

- **Push**: a control node sends the change when you run it
 - **ansible-playbook** over SSH or WinRM, Salt's master, **terraform apply**
 - Immediate and ordered: drain, patch, reboot, rejoin
 - Needs a path and credentials to every target; an offline node misses the run
- **Pull**: each node asks for its configuration on a schedule
 - Puppet and Chef agents, **ansible-pull** from cron, **cloud-init** at first boot, GitOps (Argo CD, Flux)
 - Scales out; outbound-only; drift corrected at the next check-in
 - Changes land within one interval; ordering across nodes is harder

2.2 Environments, Migrations, and Provisioning (continued)

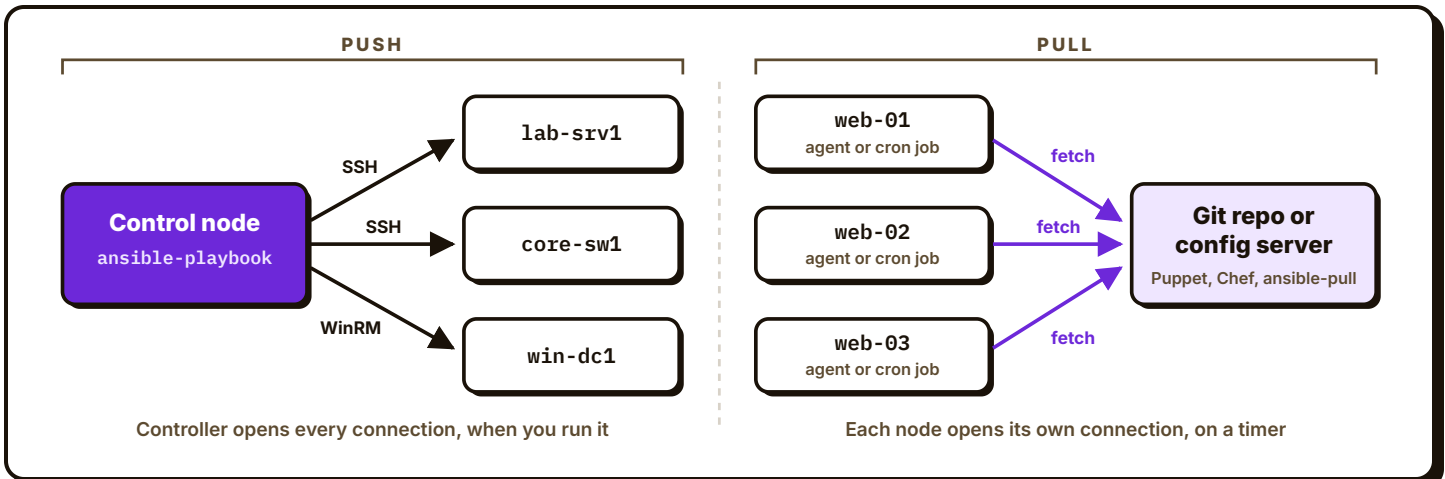


FIGURE Push: the control node starts the run and connects out to every target. Pull: each node starts its own run and fetches its configuration from a central source.

LOCAL.YML (IN THE GIT REPO)

```
- name: Baseline this node
  hosts: localhost
  connection: local
  gather_facts: false
  vars:
    root: "{{ lookup('env', 'NODE_ROOT') }}"
  tasks:
    - name: Declare the NTP sources file
      ansible.builtin.copy:
        dest: "{{ root }}/etc/chrony.sources"
        content: "server 10.0.20.5 iburst\n"
        mode: "0644"
```

TERMINAL (ON THE NODE)

```
$ REPO=file://$PWD/node-config.git
$ ansible-pull -U "$REPO" -i localhost, local.yml
Starting Ansible Pull at 2026-10-19 08:54:48
PLAY [Baseline this node] *****
TASK [Declare the NTP sources file] *****
changed: [localhost]
PLAY RECAP *****
localhost : ok=1 changed=1
$ ansible-pull -o -U "$REPO" -i localhost, local.yml
Repository has not changed, quitting
```

- In production **REPO** is an HTTPS URL and cron runs this every 15 to 30 minutes; **-o** (**--only-if-changed**) skips the play when Git has nothing new

■ Ephemeral and persistent environments

- **Ephemeral environment**: built from code on demand, thrown away after
 - A review app per pull request, CI service containers, a lab that **terraform destroy** removes at 6 p.m.
 - Proves the code builds everything from nothing: a fresh database runs all three migrations in order

- **Persistent environment**: production, staging
 - Holds real data; changed in place, one pending migration at a time

	EPHEMERAL	PERSISTENT
Lives for	Minutes to days	Months to years
Data	Seeded, disposable	Real, backed up
Changed by	Rebuild from code	Change in place
Drift	None	Detect and correct
Cost	Only while running	Always on

ON THE EXAM

- "Nodes check in with a server and apply their configuration on a schedule" is pull (agent-based). "One command configures every server now" is push.
- Ephemeral environments are built per test or pull request and destroyed after; persistent ones keep data. Clean test runs and lower cost point to ephemeral.
- Migrations are ordered, versioned scripts tracked in a table. Fix a bad one with a new migration or a rollback, not by editing it.

TRY IT

Write two dbmate migrations for **sqlite:lab.db** and run **dbmate up** and **dbmate status**. Add a third, run **dbmate up** again, then point it at **sqlite:fresh.db** and compare.

From polling to events

- Polling: a cron job asks "anything new?" every five minutes
 - Wasted runs when nothing changed; up to five minutes late when something did
- Event-driven: the system that changed announces it, and automation reacts in seconds
 - An **event** is a record that something happened: type, source, time, data
- **Cloud-based events** come in three building blocks: notifications, queues, and functions

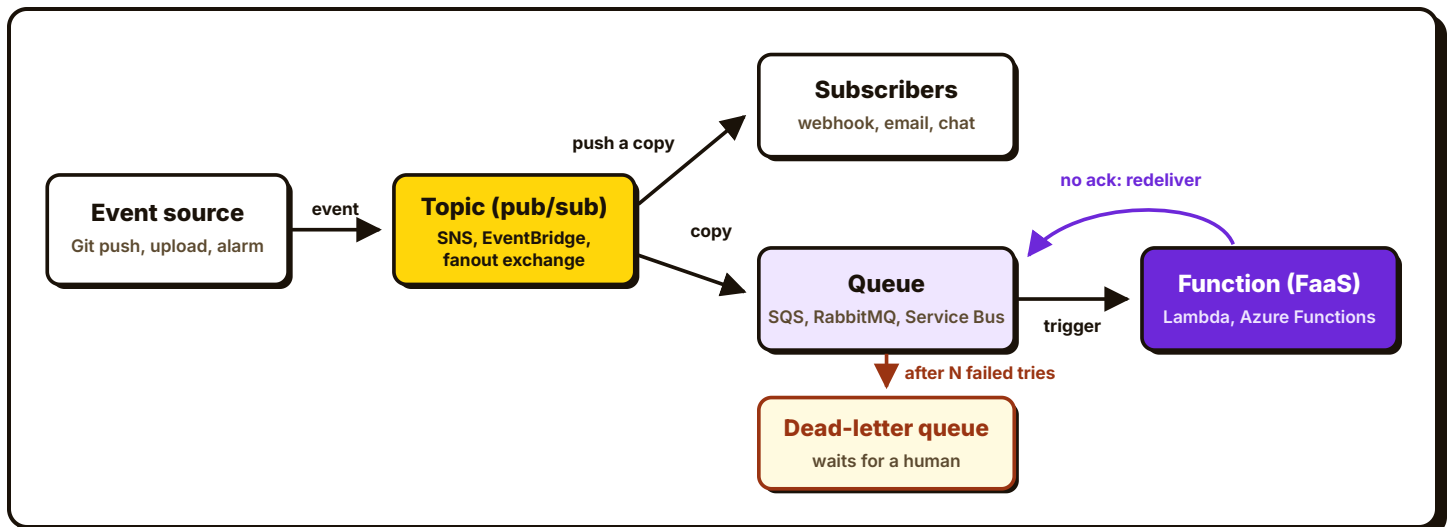


FIGURE A topic pushes a copy of each event to every subscriber. A queue holds work until one consumer finishes it; repeated failures move the message to a dead-letter queue.

Notification-based events

- **Notification-based events** push a message to whoever is subscribed at that moment
 - **Publish/subscribe (pub/sub)**: one publisher, many subscribers, each gets its own copy
 - The publisher does not know or wait for the subscribers
- Examples: webhooks (GitHub POSTs to the CI server on every push), AWS SNS and EventBridge, Azure Event Grid, S3 "object created" events
- No subscriber, no delivery: a topic is not storage
 - Below, the first event fires before the **audit** and **backup** queues subscribe, and it is simply dropped

TERMINAL

```

$ python3 events.py
notify: dropped, nobody subscribed
notify: delivered
  audit received core-sw1 v43
  backup received core-sw1 v43
  
```

Messaging queues

- A **messaging queue** stores each message until one consumer handles it
 - Point-to-point: each message is processed once, by one worker
 - Producers and consumers run at their own speed; the queue absorbs bursts
- **Competing consumers**: add workers to drain the queue faster
- The consumer **acknowledges** (acks) a message only after the work succeeds
 - No ack (crash, timeout) and the broker hands the message out again
 - SQS hides an in-flight message for its visibility timeout (30 seconds by default)
- Examples: AWS SQS, RabbitMQ, Azure Service Bus, Redis streams
- **Fan-out**: subscribe several queues to one topic (SNS to SQS) for a durable copy per team

2.2 Event-Driven Automation (continued)

	PUB/SUB TOPIC	QUEUE
Who gets a message	Every current subscriber	Exactly one consumer
Stored until handled	No	Yes, until acked or it expires
Typical use	Tell many systems something happened	Hand a unit of work to a pool of workers
Cloud examples	SNS, EventBridge, Event Grid	SQS, Service Bus, Storage Queues

■ At-least-once delivery and dead letters

- Most queues promise **at-least-once delivery**
 - Never lost once accepted, but it can arrive twice (the worker finished, then crashed before the ack)
 - At-most-once can lose messages; exactly-once holds only within limits, such as SQS FIFO deduplication over five minutes
- So every consumer must be an **idempotent consumer**
 - Record each message ID (Redis, a unique database key) and skip ones already done
 - Or make the action safe to repeat: upsert instead of insert
- A **dead-letter queue (DLQ)** catches a message that keeps failing
 - After N deliveries (SQS `maxReceiveCount`, RabbitMQ `x-delivery-limit`) it moves aside, so one poison message cannot block the queue
 - Alert on DLQ depth; a human fixes and replays, or discards

EVENTS.PY (EXCERPT)

```
DLQ = "jobs.backup.dlq"
args = {
    "x-queue-type": "quorum",
    "x-delivery-limit": 2,
    "x-dead-letter-exchange": "",
    "x-dead-letter-routing-key": DLQ,
}
ch.queue_declare(DLQ, durable=True)
ch.queue_declare("jobs.backup", durable=True,
                 arguments=args)

def handle(m, p, body):
    mid = p.message_id
    if mid in seen:
        return f"{mid}: duplicate, skipped"
    seen.add(mid)
    return f"{mid}: backed up {body.decode()}"
```

TERMINAL (CONTINUED)

```
worker-a backed up core-sw1
worker-b backed up core-sw2
worker-a backed up edge-rtr1
worker-b backed up edge-rtr2
try 1 edge-rtr9: crashed
try 2 edge-rtr9: crashed (redelivered)
try 3 edge-rtr9: crashed (redelivered)
dead-lettered edge-rtr9 (delivery_limit)
job-2001: backed up core-sw1
job-2001: duplicate, skipped
$ rabbitmqctl -q list_queues name type messages
name      type      messages
jobs.backup      quorum    0
jobs.backup.dlq  classic  1
```

- Two workers split four jobs; `edge-rtr9` failed three times and went to the DLQ; the retried `job-2001` ran once

■ Function as a service (FaaS)

- **Function as a service (FaaS)**: upload a function; the platform runs it when an event arrives
 - No server to patch; scales out per event and down to zero; billed per request and millisecond
- Triggers: HTTP request, queue message, topic, schedule, file upload, database change
- Platforms: AWS Lambda, Azure Functions, Google Cloud Run functions; self-hosted OpenFaaS or Knative
- Short and stateless by design
 - Lambda stops at 15 minutes; nothing local survives between calls; the first call pays a **cold start**
- Automation uses: tag new VMs, close a security group opened to `0.0.0.0/0`, back up a switch when its config changes

2.2 Event-Driven Automation (continued)

HANDLER.PY (EXCERPT)

```
def handler(event, context):
    """Called once per batch of queue messages."""
    failed = []
    for record in event["Records"]:
        job = json.loads(record["body"])
        try:
            backup(job["host"])
        except ValueError:
            failed.append(record["messageId"])
    # Only the failed messages go back to the queue
    return {"batchItemFailures": [
        {"itemIdentifier": i} for i in failed]}
```

TERMINAL

```
$ python3 invoke.py event.json
{"batchItemFailures": [{"itemIdentifier": "m-2"}]}
```

- `invoke.py` feeds the handler a two-message, SQS-shaped batch; no cloud account is involved
- With an SQS trigger and `ReportBatchItemFailures` on, Lambda returns only `m-2` to the queue; without it, the whole batch is retried

ON THE EXAM

- "Every subscribed system must be told" is pub/sub (SNS, Event Grid, webhooks). "Each job must be processed once, even if workers are offline for an hour" is a queue (SQS, Service Bus).
- At-least-once delivery means duplicates are possible, so the answer is an idempotent consumer. Messages that fail again and again go to a dead-letter queue.
- FaaS is event-triggered, short-running, and stateless; a two-hour batch job is the distractor that does not fit.

TRY IT

Start RabbitMQ in Docker (`rabbitmq:4-management-alpine` , port 5672), install `pika` , and publish five messages to a queue. Consume with two workers and watch the broker split the work, then stop one worker before it acks and see the message come back with `redelivered=True` .

2.3 REST APIs: Methods, Headers, and Status Codes

2.0 SYSTEM CONFIGURATION

What makes an API RESTful

- **REST** (Representational State Transfer) is an architectural style for web APIs
 - Not a protocol or a product: plain HTTP or HTTPS, usually with JSON bodies
 - A **RESTful** API follows its constraints
- Everything is a **resource** with its own URI
 - Collection: `/api/v1/vlans` ; one item: `/api/v1/vlans/30`
 - Nouns in the path, the action in the method: **DELETE** `/vlans/30` , not `/deleteVlan?id=30`
- **Stateless**: each request carries everything the server needs
 - The token rides in a header on every call; there is no session to lose
 - Any node behind a load balancer can answer any request

- The other constraints
 - Client-server split, cacheable responses, a uniform interface
 - Layered system: proxies and gateways in the path are invisible to the client
- An **endpoint** is a base URL plus a resource path
 - The version usually sits in the base URL: `/api/v1/`
 - A query string filters a collection: `/api/v1/vlans?name=voice`
- Where REST shows up in network automation
 - NetBox or Nautobot as the source of truth, the Meraki Dashboard API, Catalyst Center, cloud provider APIs, GitHub and GitLab
 - The Ansible `uri` module and most Terraform providers make these same calls underneath

Methods map to CRUD

2.3 REST APIs: Methods, Headers, and Status Codes (continued)

CRUD	METHOD	EXAMPLE REQUEST	TYPICAL SUCCESS RESPONSE
Create	POST	POST /api/v1/vlans with a JSON body	201 Created plus a Location header
Read	GET	GET /api/v1/vlans/30	200 OK with the resource in the body
Update (replace)	PUT	PUT /api/v1/vlans/30 with the complete object	200 OK, or 204 No Content
Update (modify)	PATCH	PATCH /api/v1/vlans/30 with only the changed fields	200 OK, or 204 No Content
Delete	DELETE	DELETE /api/v1/vlans/30	204 No Content (some APIs send 200)

- **PUT replaces** the whole resource
 - Any field you leave out is removed or reset to its default
 - Some APIs also let **PUT** create a resource at a URI the client picks
- **PATCH modifies** only the fields you send; the rest stay as they were
- From the lab API in the next topic: VLAN 30 has a **name** and a **description**
 - PUT {"name": "voip"} returns {"id": 30, "name": "voip"}: the description is gone
 - PATCH {"description": "Desk phones"} returns all three fields
- **Safe** methods only read: **GET**, **HEAD**, **OPTIONS**
- **Idempotent** methods leave the server in the same state no matter how many times you send them
 - The safe methods, plus **PUT** and **DELETE**
 - A second **DELETE** returns **404**, but the VLAN is still gone: same state, different response code
 - **POST** is neither: send it twice and you get two resources or a **409 Conflict**
 - **PATCH** is not guaranteed idempotent (a patch that appends to a list adds twice)
- Why automation cares: a script or pipeline can safely retry an idempotent call after a timeout

Headers

- **Request headers** describe the client and the body it sends; **response headers** describe the result
- **Name: value** lines between the first line and the body
 - Header names are case-insensitive
 - A blank line ends the headers; the body follows

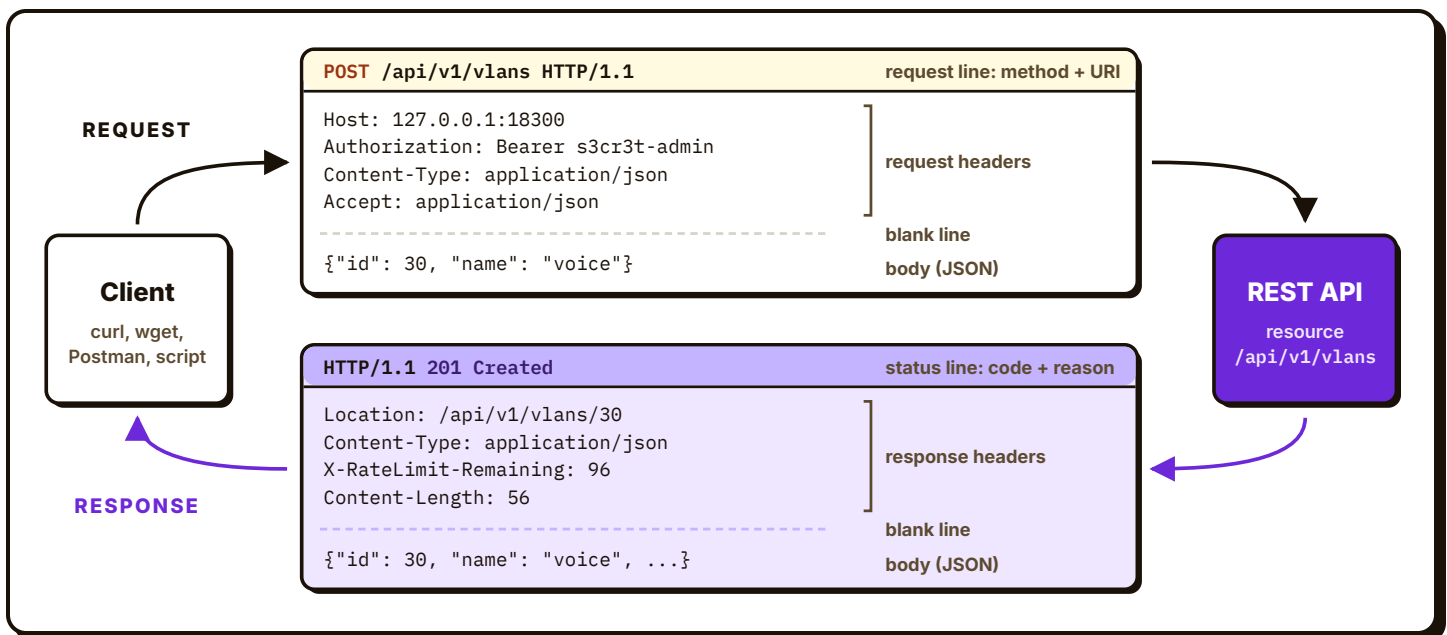


FIGURE One request and one response. Every HTTP message has the same four parts: a first line, headers, a blank line, and an optional body.

2.3 REST APIs: Methods, Headers, and Status Codes (continued)

HEADER	SENT BY	PURPOSE
Authorization	client	Credentials: Bearer <token>, or Basic with user:pass in base64 (encoded, not encrypted)
X-API-Key	client	A vendor-specific API key header; the name varies by product
Content-Type	both	Format of the body in this message, such as application/json
Accept	client	Formats the client is willing to receive
Location	server	URI of a newly created resource (201) or of a redirect target (3XX)
ETag	server	Version tag of the resource; the client sends it back in If-None-Match (cache check) or If-Match (only update if unchanged)
WWW-Authenticate	server	Sent with 401 ; names the authentication scheme the server expects
Retry-After	server	Seconds to wait after a 429 or 503
X-RateLimit-Limit, X-RateLimit-Remaining	server	Your quota and the calls left in this window (a common convention; exact names vary by vendor)

- **curl -v** prints the request headers (>) and the response headers (<)

```
$ API=http://127.0.0.1:18300/api/v1
$ AUTH="Authorization: Bearer s3cr3t-admin"
$ curl -sv -H "$AUTH" $API/vlans/10
> GET /api/v1/vlans/10 HTTP/1.1
> Host: 127.0.0.1:18300
> User-Agent: curl/8.5.0
> Accept: */*
> Authorization: Bearer s3cr3t-admin
>
< HTTP/1.1 200 OK
< X-RateLimit-Limit: 100
< X-RateLimit-Remaining: 99
< ETag: "bc2bc3803223"
< Content-Type: application/json
< Content-Length: 61
```

- Send the ETag back and an unchanged resource costs no body

```
$ ET='If-None-Match: "bc2bc3803223"'
$ curl -si -H "$AUTH" -H "$ET" $API/vlans/10
HTTP/1.1 304 Not Modified
ETag: "bc2bc3803223"
Content-Length: 0
```

Response codes

- A **response code** (status code) is three digits plus a reason phrase on the first line of every response
 - The first digit is the **class**: **2XX**, **3XX**, **4XX**, **5XX** ; **1XX** (informational) rarely matters to scripts
 - Test the number, never the phrase: HTTP/2 does not send phrases at all

2.3 REST APIs: Methods, Headers, and Status Codes (continued)

CODE	NAME	WHAT IT TELLS YOUR SCRIPT
2XX	Success	The request worked
200	OK	Result is in the body
201	Created	New resource; its URI is in <code>Location</code>
204	No Content	Worked, nothing to return (typical for <code>DELETE</code>)
3XX	Redirection	Look somewhere else
301	Moved Permanently	Resource has a new URI for good; update the script (<code>curl -L</code> follows it)
302	Found	Temporary redirect, often to a login page; keep using the old URI
304	Not Modified	Your cached copy (by <code>ETag</code>) is current; no body
4XX	Client error	Fix the request; resending it unchanged will fail again
400	Bad Request	Malformed request, such as invalid JSON
401	Unauthorized	Not authenticated: token missing, wrong, or expired
403	Forbidden	Authenticated, but this identity is not allowed to do this
404	Not Found	Wrong URI, or the resource does not exist
409	Conflict	Clashes with current state: duplicate ID, edit collision
422	Unprocessable Content	Valid JSON, but a value fails validation (older name: Unprocessable Entity)
429	Too Many Requests	Rate limit hit; wait for <code>Retry-After</code>
5XX	Server error	The request may be fine; the service is not
500	Internal Server Error	Unhandled error inside the application
502	Bad Gateway	A proxy or load balancer got no valid answer from the backend
503	Service Unavailable	Overloaded or in maintenance; often sends <code>Retry-After</code>
504	Gateway Timeout	A proxy gave up waiting for the backend

- `401` versus `403` , from the lab API

```

$ curl -si $API/vlans
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Bearer realm="vlan-api"

{"error": "missing or invalid token"}
$ RO="Authorization: Bearer s3cr3t-viewer"
$ curl -si -X DELETE -H "$RO" $API/vlans/10
HTTP/1.1 403 Forbidden

{"error": "token is read-only"}
```

```

$ OLD=http://127.0.0.1:18300/api/vlans
$ curl -si -H "$AUTH" $OLD
HTTP/1.1 301 Moved Permanently
Location: /api/v1/vlans
$ curl -sL -H "$AUTH" $OLD | jq length
2
```

- A `301` from the old, unversioned path

2.3 REST APIs: Methods, Headers, and Status Codes (continued)

ON THE EXAM

- **401** means the server does not know who you are (fix the credential); **403** means it knows and says no (fix the permission). A scenario with a valid token that cannot delete is **403**.
- "Change one field without sending the whole object" is **PATCH**; "replace the object" is **PUT**; "create" is **POST**, answered by **201** and a **Location** header.
- A **4XX** points at the request, a **5XX** at the service. **502** and **504** come from the proxy in front of the application, so look at the backend behind it.

2.3 Working with REST APIs: curl, wget, and Postman

2.0 SYSTEM CONFIGURATION

■ The lab API

- Every example on these pages ran against a small VLAN inventory API
 - Python standard library only, started by `tests/2.3-rest-api.sh` on port 18300
 - **GET** and **POST** on `/api/v1/vlans`; **GET**, **PUT**, **PATCH**, **DELETE** on `/api/v1/vlans/{id}`
 - Bearer tokens: `s3cr3t-admin` can write, `s3cr3t-viewer` can only read
- Put the parts you repeat in shell variables
 - Shorter commands, and the token is typed once

TERMINAL

```
$ API=http://127.0.0.1:18300/api/v1
$ AUTH="Authorization: Bearer s3cr3t-admin"
$ CT="Content-Type: application/json"
```

■ curl options for API work

- **curl** sends one request, prints the response body, and exits
 - Installed nearly everywhere, including Windows 10 and later as `curl.exe`
 - The default method is **GET**

OPTION	WHAT IT DOES
<code>-X PATCH</code>	Sets the method
<code>-H "Name: value"</code>	Adds a request header (repeatable)
<code>-d '{...}'</code> , <code>-d @file</code>	Sends a body; implies POST
<code>--json '{...}'</code>	Body plus JSON headers (curl 7.82+)
<code>-i</code>	Prints response headers too
<code>-v</code>	Shows both sides' headers and TLS
<code>-s</code> , <code>-S</code>	No progress meter; still show errors
<code>-f</code>	Exit code 22 on HTTP 400 or higher
<code>-o file</code>	Saves the body to a file
<code>-w '%{http_code}'</code>	Prints the status code
<code>-L</code>	Follows 3XX redirects
<code>-u user:pass</code>	Basic authentication
<code>-k</code>	Skips certificate checks: lab only

- `-d` alone sends **Content-Type: application/x-www-form-urlencoded**
 - A JSON API answers that with **415 Unsupported Media Type**; always add the **Content-Type** header (or use `--json`)

2.3 Working with REST APIs: curl, wget, and Postman (continued)

A full CRUD session with curl

VLAN30.JSON

```
{
  "id": 30,
  "name": "voice",
  "description": "IP phones"
}
```

- **Create** with **POST**, and three ways the create can be refused (headers trimmed)

TERMINAL

```
$ curl -si -H "$AUTH" -H "$CT" -d @vlan30.json $API/vlans
HTTP/1.1 201 Created
Location: /api/v1/vlans/30
Content-Type: application/json

{"id": 30, "name": "voice", "description": "IP phones"}
$ curl -si -H "$AUTH" -H "$CT" -d @vlan30.json $API/vlans
HTTP/1.1 409 Conflict

{"error": "VLAN 30 already exists"}
$ curl -si -H "$AUTH" -d @vlan30.json $API/vlans
HTTP/1.1 415 Unsupported Media Type

{"error": "send Content-Type: application/json"}
$ curl -si -H "$AUTH" -H "$CT" -d '{"id": 5000, "name": "lab"}' $API/vlans
HTTP/1.1 422 Unprocessable Content

{"error": "validation failed", "fields": {"id": "must be 1-4094"}}
```

- **Read, update, and delete**

TERMINAL

```
$ curl -s -H "$AUTH" $API/vlans | jq -r '.[].name'
users
servers
voice
$ curl -s -X PUT -H "$AUTH" -H "$CT" -d '{"name": "voip"}' $API/vlans/30
{"id": 30, "name": "voip"}
$ curl -s -X PATCH -H "$AUTH" -H "$CT" -d '{"description": "Desk phones"}' $API/vlans/30
{"id": 30, "name": "voip", "description": "Desk phones"}
$ curl -si -X DELETE -H "$AUTH" $API/vlans/30
HTTP/1.1 204 No Content
$ curl -si -X DELETE -H "$AUTH" $API/vlans/30
HTTP/1.1 404 Not Found
```

- The **PUT** dropped the description; the **PATCH** added one back and kept the name
- Inside the **-d** string, JSON needs double quotes, so wrap the whole string in single quotes for the shell

curl in scripts

- By default curl exits **0** for any HTTP response, even a **500**
 - **-f** (**--fail**) is the fix
 - It turns **4XX** and **5XX** into exit code **22**, so **set -e** and CI jobs notice

TERMINAL

```
$ curl -sSf -H "$AUTH" $API/vlans/30
curl: (22) The requested URL returned error: 404
$ curl -sf -H "$AUTH" $API/vlans/30 || echo
"exit=$?"
exit=22
```

- To branch on the exact code, capture it with **-w** and discard the body

2.3 Working with REST APIs: curl, wget, and Postman (continued)

VLAN-EXISTS.SH

```
#!/usr/bin/env bash
# Exit 0 if VLAN $1 exists, 1 if not, 2 on anything
else
code=$(curl -s -o /dev/null -w '%{http_code}' \
-H "$AUTH" "$API/vlans/$1")
case $code in
200) echo "VLAN $1 present" ;;
404) echo "VLAN $1 missing"; exit 1 ;;
*) echo "unexpected HTTP $code"; exit 2 ;;
esac
```

TERMINAL

```
$ ./vlan-exists.sh 10
VLAN 10 present
$ ./vlan-exists.sh 30; echo "exit=$?"
VLAN 30 missing
exit=1
$ AUTH="Authorization: Bearer typo" ./vlan-exists.sh
10
unexpected HTTP 401
```

- **jq** slices the JSON that comes back

TERMINAL

```
$ curl -s -H "$AUTH" $API/vlans -o all.json
$ jq -r '.[0] | "\(.id) \(.name)"' all.json
10 users
20 servers
31 voice2
$ jq 'map(select(.id >= 20)) | length' all.json
2
```

wget

- **wget** is a downloader first
 - Saves the body to a file named after the URL; **-O-** writes to the terminal instead
 - Follows redirects on its own (curl needs **-L**)
 - Exit code **8** means the server answered with an error status
- Same API, wget flags

TERMINAL

```
$ wget -qO- --header "$AUTH" $API/vlans/10
{"id": 10, "name": "users", "description": "Staff desktops"}
$ wget -qO- --header "$AUTH" --header "$CT" --post-data '{"id": 60, "name": "cameras"}' $API/vlans
{"id": 60, "name": "cameras"}
$ wget -qO- --method=PATCH --header "$AUTH" --header "$CT" --body-data '{"name": "cctv"}' $API/vlans/60
{"id": 60, "name": "cctv"}
$ wget -qS -O /dev/null --method=DELETE --header "$AUTH" $API/vlans/60
HTTP/1.1 204 No Content
$ wget -q --header "$AUTH" $API/vlans/60; echo "exit=$?"
exit=8
```

- **--post-data** sends **POST**; any other method needs **--method** plus **--body-data**
- **-S** prints the response headers (to stderr)
- Reach for curl when you are working an API; wget fits "fetch this file" jobs such as pulling an image or a config template
- An **environment** is a named set of **variables** for one target: lab, staging, production
 - Requests use **{{baseUrl}}** and **{{token}}**; pick another environment and the same collection hits another system
 - Scopes from broadest to narrowest: global, collection, environment, data (a CSV or JSON file feeding a run), local; the narrowest one wins

Postman and Newman

- **Postman** is a graphical API client: build a request, send it, save it, share it
- A **collection** is a saved, ordered set of requests
 - Each request can carry scripts written in JavaScript against the **pm** object
 - Pre-request scripts run first (build a timestamp, fetch a token); post-response scripts hold the tests
 - Set auth once on the collection; requests inherit it
 - Exports to JSON (collection format v2.1), so it can live in Git

LAB.JSON

```
{
  "name": "lab",
  "values": [
    { "key": "baseUrl",
      "value": "http://127.0.0.1:18300/api/v1" },
    { "key": "token", "value": "s3cr3t-admin" }
  ]
}
```

2.3 Working with REST APIs: curl, wget, and Postman (continued)

CREATE VLAN: POST-RESPONSE SCRIPT

```
pm.test('status is 201', () =>
  pm.response.to.have.status(201));
pm.test('Location header set', () =>
  pm.response.to.have.header('Location'));
```

- **Newman** runs an exported collection from the command line (`npm install -g newman`)
 - Exit code `0` when every test passes, `1` when any fails, so a CI job can gate on it
 - Postman also publishes its own runner, the Postman CLI (`postman collection run`)

TERMINAL

```
$ newman run vlans.json -e lab.json --disable-unicode
POST http://127.0.0.1:18300/api/v1/vlans [201 Created, 245B, 50ms]
Pass status is 201
Pass Location header set
GET http://127.0.0.1:18300/api/v1/vlans/70 [200 OK, 234B, 4ms]
Pass name is printers
PATCH http://127.0.0.1:18300/api/v1/vlans/70 [200 OK, 209B, 46ms]
Pass status is 200
DELETE http://127.0.0.1:18300/api/v1/vlans/70 [204 No Content, 156B, 2ms]
Pass status is 204
| assertions | 5 | 0 |
$ newman run vlans.json -e ro.json --disable-unicode --color off
POST http://127.0.0.1:18300/api/v1/vlans [403 Forbidden, 220B, 26ms]
1. status is 201
1. AssertionError status is 201
   expected response to have status code 201 but got 403
```

- Same collection, environment with the read-only token: every write fails and the run exits `1`

ON THE EXAM

- Match flags to jobs: `-X` sets the method, `-H` adds a header, `-d` sends a body, `-i` shows response headers, `-L` follows redirects, `-k` ignores certificate errors (a red flag in any production answer).
- Postman environment variables let one collection target lab or production unchanged. Newman, or the Postman CLI, runs that collection in a pipeline and fails the job on a failed test.
- A JSON body sent without `Content-Type: application/json` is a common distractor; expect `415` or `400` from the API.

TRY IT

The lab API is the Python heredoc inside `tests/2.3-rest-api.sh`. Save it as `api.py`, start it with `python3 api.py 18300`, and from a second terminal create VLAN 99 with `POST`, change only its name with `PATCH`, then `PUT` a body with just a name and see which field disappears.

2.4 Troubleshooting System Configuration

2.0 SYSTEM CONFIGURATION

■ Work the layers in order

- An API call, an agent check-in, and a playbook task that calls a controller all fail in the same places
 - Read the exact error first; the **curl exit code** or HTTP status usually names the layer
 - `curl -v` shows how far the call got: resolved, connected, TLS done, request sent
 - Fix the lowest broken layer first
 - A TLS error means DNS and TCP already worked; an HTTP `401` means TLS already worked

2.4 Troubleshooting System Configuration (continued)

A CALL CLEARS EACH LAYER IN ORDER. FIX THE FIRST ONE THAT FAILS.

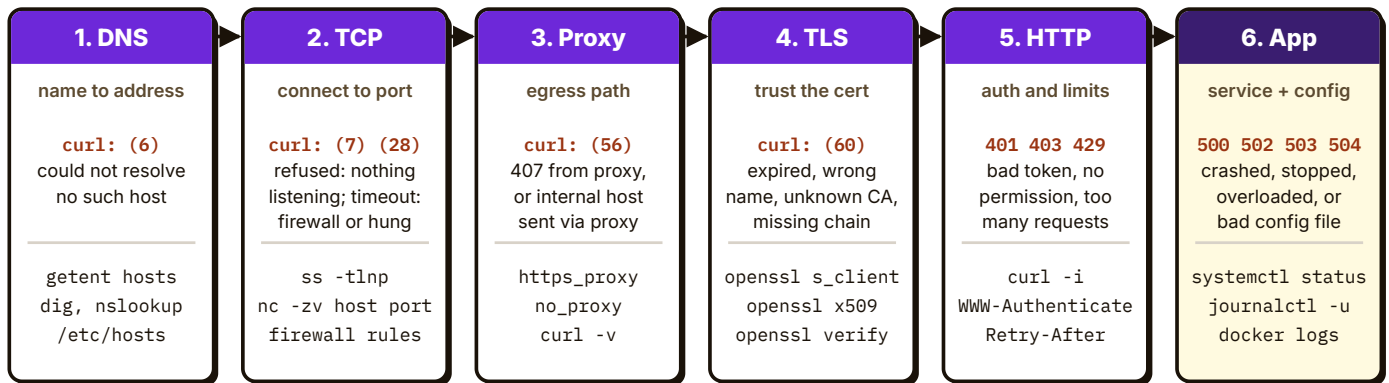


FIGURE Each layer depends on the one to its left. The curl exit code or HTTP status tells you which layer stopped the call.

API communication failures

- Four **API communication failures**: a DNS miss, **connection refused**, a **timeout**, and a **proxy** that wants credentials (curl exit codes in parentheses)

TERMINAL

```
$ curl -sS http://inv.lab.invalid:18300/
curl: (6) Could not resolve host: inv.lab.invalid
$ getent hosts inv.lab.invalid; echo "exit=$?"
exit=2
$ curl -sS http://127.0.0.1:18319/
curl: (7) Failed to connect to 127.0.0.1 port 18319 after 0 ms: Couldn't connect to server
$ curl -sS -m 3 http://127.0.0.1:18318/
curl: (28) Operation timed out after 3002 milliseconds with 0 bytes received
$ https_proxy=http://127.0.0.1:18317 curl -sS https://api.example.com/
curl: (56) CONNECT tunnel failed, response 407
```

SYMPTOM	LIKELY CAUSE	CHECK OR FIX
Could not resolve host	Typo in the hostname, missing DNS record, wrong resolver	<code>getent hosts</code> , <code>dig</code> ; fix the record or the URL in the config
Connection refused	Host is up, nothing listens on that port: service down, wrong port, bound to 127.0.0.1 only	<code>ss -tlnp</code> and <code>systemctl status</code> on the server
Timeout	Firewall or security group dropping packets, bad route, or a hung application	<code>nc -zv host port</code> ; firewall rules; set <code>--connect-timeout</code> and <code>-m</code> in scripts
407, CONNECT tunnel failed	The proxy wants credentials, or an internal host is being sent through the proxy	<code>https_proxy</code> , <code>no_proxy</code> ; <code>curl -v</code> shows the CONNECT
401, 403, 429	Bad or expired token; missing permission; rate limit (objective 2.3)	<code>curl -i</code> ; read <code>WWW-Authenticate</code> and <code>Retry-After</code> ; only 429 improves by waiting

Agent communication issues

- Agent**-based tools run a daemon on every managed node that talks to a central server
 - Zabbix: server polls the agent on 10050; the agent sends active checks to the server on 10051
 - Puppet agent to Puppet Server on 8140; Salt minion to master on 4505 and 4506
 - Ansible is agentless (SSH, WinRM), so its "agent" problems are SSH or WinRM problems
- Two Zabbix agent 2 containers: one with a server name that does not resolve, one whose server is not listening

2.4 Troubleshooting System Configuration (continued)

TERMINAL

```
$ docker logs ao-e-zbx 2>&1 | grep -m2 -E "cannot connect|started to fail"
2026/10/19 13:00:50.004532 [101] cannot connect to [zbx.lab.invalid:10051]: dial tcp: lookup zbx.lab.invalid on
8.8.8.8:53: no such host
2026/10/19 13:00:50.004543 [101] active check configuration update from host [edge-sw-mon] started to fail
$ docker logs ao-e-zbx 2>&1 | grep -m1 "cannot connect"
2026/10/19 13:00:50.002120 [101] cannot connect to [127.0.0.1:10051]: dial tcp :0->127.0.0.1:10051: connect:
connection refused
```

- The agent keeps running and the server shows the host as unavailable; only the agent's log has the cause
- Checklist
 - Server address and port in the agent config:
ServerActive= in `zabbix_agent2.conf`, `server =` in `puppet.conf`
 - DNS and firewall rules as seen from the agent, in the direction the connection flows
 - Agent hostname matches the host defined on the server
 - Certificates or PSK match (Puppet agents need a certificate signed by the server's CA); NTP running, since clock skew breaks certificates
 - Restart the agent after editing its config

■ Application failures

- A unit that crashes on start: systemd restarts it, then gives up

TERMINAL

```
$ systemctl status invsync --no-pager
x invsync.service - Inventory sync agent
   Loaded: loaded (/etc/systemd/system/invsync.service; enabled; preset: enabled)
   Active: failed (Result: exit-code) since Mon 2026-10-19 13:01:01 UTC; 1s ago
   Process: 193 ExecStart=/usr/bin/python3 /opt/invsync/agent.py (code=exited, status=1/FAILURE)
$ journalctl -u invsync -o cat --no-pager | grep -E "Error|repeated" | tail -2
json.decoder.JSONDecodeError: Expecting property name enclosed in double quotes: line 4 column 1 (char 62)
invsync.service: Start request repeated too quickly.
$ systemctl show invsync -p NRestarts -p ExecMainStatus
NRestarts=5
ExecMainStatus=1
```

- The cause was a trailing comma in `/etc/invsync/agent.json` (next topic)
- **Start request repeated too quickly**: by default, five failed starts within 10 seconds stops the restarts
 - After the fix: `systemctl reset-failed invsync`, then `systemctl restart invsync`
 - After editing the unit file itself: `systemctl daemon-reload`
- **journalctl** flags: `-u unit`, `-n 50` last lines, `-f` follow, `-b` this boot, `--since "10 min ago"`, `-p err` errors only

- Active is not the same as working: with valid JSON the service ran, and its log showed the next layer down

TERMINAL

```
$ systemctl is-active invsync
active
$ journalctl -u invsync -n 1 --no-pager -o cat
cannot reach http://inv.lab.invalid:8080: <urlopen
error [Errno -2] Name or service not known>
```

- Containers: the **exit code** and the logs survive after the container stops

2.4 Troubleshooting System Configuration (continued)

TERMINAL

```
$ docker ps -a --filter name=ao-e-web --format "{{.Names}}: {{.Status}}"
ao-e-web: Exited (1) Less than a second ago
$ docker logs ao-e-web 2>&1 | tail -1
nginx: [emerg] unknown directive "proxy_pas" in /etc/nginx/conf.d/default.conf:4
$ docker run --rm alpine:3.20 inventory-sync 2>&1 | grep -o "exec:.*"
exec: "inventory-sync": executable file not found in $PATH
$ docker run --rm alpine:3.20 inventory-sync 2>/dev/null; echo "exit=$?"
exit=127
$ docker inspect -f "{{.State.ExitCode}} OOMKilled={{.State.OOMKilled}}" ao-e-kill
137 OOMKilled=false
```

- Exit codes to know: **1** general error, **126** found but not executable, **127** command not found, **137** = 128 + 9 (SIGKILL: OOM killer or `docker kill`), **143** = 128 + 15 (SIGTERM)
- Test a config before you load it: `nginx -t`, `apachectl configtest`, `sshd -t`, `named-checkconf`; the nginx typo above fails `nginx -t` without taking the site down
- 5XX** from a reverse proxy: the proxy is fine, the backend is not

TERMINAL

```
$ P=http://127.0.0.1:18310
$ curl -s -o /dev/null -w "%{http_code}\n" $P/down/502
$ curl -s -o /dev/null -w "%{http_code}\n" $P/slow/504
```

- nginx's error log names the cause: `connect() failed (111: Connection refused)` for the **502**, `upstream timed out (110: Operation timed out)` for the **504**

■ Certificate issues

- Four local `openssl s_server` instances show the common **certificate issues**: `root.pem` is the lab root CA
 - 18320 is correct (leaf plus intermediate), 18321 sends only the leaf, 18322 is expired, 18323 is self-signed
 - The first curl below does not pass `--cacert`, so the lab CA is untrusted
- Read the certificate first, then let curl and `openssl s_client` try the servers

TERMINAL

```
$ openssl x509 -in api.pem -noout -subject -issuer -enddate -ext subjectAltName
subject=CN = api.lab.test
issuer=CN = Lab Issuing CA
notAfter=Jan 17 13:01:14 2027 GMT
X509v3 Subject Alternative Name:
    DNS:api.lab.test, IP Address:127.0.0.1
$ curl -sS https://127.0.0.1:18320/
curl: (60) SSL certificate problem: unable to get local issuer certificate
$ curl -sS https://127.0.0.1:18323/
curl: (60) SSL certificate problem: self-signed certificate
$ curl -sS --cacert root.pem https://localhost:18320/
curl: (60) SSL: no alternative certificate subject name matches target host name 'localhost'
$ curl -sS --cacert root.pem https://127.0.0.1:18322/
curl: (60) SSL certificate problem: certificate has expired
$ curl -sS --cacert root.pem https://127.0.0.1:18321/
curl: (60) SSL certificate problem: unable to get local issuer certificate
$ openssl s_client -connect 127.0.0.1:18321 -CAfile root.pem -brief </dev/null 2>&1 | grep -E "depth|error"
depth=0 CN = api.lab.test
verify error:num=20:unable to get local issuer certificate
verify error:num=21:unable to verify the first certificate
$ openssl s_client -connect 127.0.0.1:18320 -CAfile root.pem -showcerts </dev/null 2>/dev/null | grep -E "^ [0-9] s:|^Verify return"
0 s:CN = api.lab.test
1 s:CN = Lab Issuing CA
Verify return code: 0 (ok)
```

2.4 Troubleshooting System Configuration (continued)

- Port 18321 is the **incomplete chain** trap: curl says "unable to get local issuer" even with the right CA, because the server sent only its own certificate
 - `s_client` adds error 21, and a healthy server lists two certificates (leaf, then intermediate)

PROBLEM	WHAT THE CLIENT REPORTS	FIX
Expired	<code>certificate has expired</code> (OpenSSL error 10)	Renew; automate renewal (ACME, cert-manager); monitor with <code>openssl x509 -checkend</code>
Hostname mismatch	<code>no alternative certificate subject name matches target host name</code>	Use a name listed in the SAN, or reissue with the right SAN (clients ignore the CN)
Untrusted CA	<code>unable to get local issuer certificate</code> (20), <code>self-signed certificate</code> (18)	Add the CA to the client's trust store: <code>--cacert</code> , <code>update-ca-certificates</code> , <code>REQUESTS_CA_BUNDLE</code>
Incomplete chain	<code>curl: error 20; s_client : error 21 unable to verify the first certificate</code>	Configure the server with the full chain: leaf plus intermediates
Wrong clock	<code>certificate is not yet valid</code> (9), or a false expiry	Fix NTP on the client

- Never fix a certificate error with `curl -k`, `verify=False`, or `validate_certs: false` in automation; it removes the protection TLS exists for

ON THE EXAM

- Map the message to the layer: "could not resolve" is DNS, "connection refused" is a port with no listener, a timeout is usually a firewall, `curl: (60)` is a certificate, `401` / `403` is credentials or permissions.
- An agent that is running but whose host shows as unavailable is a communication problem: check the server address in the agent config, DNS, and the firewall in the direction the agent connects.
- Same "unable to get local issuer certificate" error, two causes: the client does not trust the CA (fix the client trust store) or the server did not send its intermediate (fix the server chain).

2.4 Configuration File Syntax: INI, YAML, and JSON

2.0 SYSTEM CONFIGURATION

Parse before you deploy

- A config file with one bad character can stop a service, fail a playbook, or push half a change
 - The parser stops at the first error and reports a line and column
 - That position is where the parser gave up, which is often the line after the real mistake
- Validate** every config file in the pipeline, before anything restarts
 - One small checker covers all three formats used below

2.4 Configuration File Syntax: INI, YAML, and JSON (continued)

CHECK-CONFIG.PY

```
#!/usr/bin/env python3
"""Parse INI, YAML, or JSON files; report the
error."""
import configparser, json, sys
import yaml # PyYAML

status = 0
for path in sys.argv[1:]:
    try:
        with open(path) as f:
            if path.endswith(".ini"):
                cp = configparser.ConfigParser()
                cp.read_file(f)
                # reading each value expands %
                (name)s references
                [dict(cp[s]) for s in cp.sections()]
            elif path.endswith((".yaml", ".yml")):
                yaml.safe_load(f)
            else:
                json.load(f)
        print(f"{path}: OK")
    except Exception as e:
        print(f"{path}: {type(e).__name__}: {e}")
        status = 1
sys.exit(status)
```

- Format-specific tools: **yamllint** for YAML, **jq empty** or **python3 -m json.tool** for JSON, and the application's own test mode (**nginx -t**, **ansible-playbook --syntax-check**)

INI

- Sections** in brackets, then **key = value** (or **key: value**) lines
 - Comments start with **;** or **#** on their own line
 - Flat: sections cannot nest
 - Every value is a string; the program converts it (**getint**, **getboolean**)

- Seen in **ansible.cfg**, Ansible's INI-style inventory, **.gitconfig**, **php.ini**, and systemd unit files (a close cousin)
- An inline comment is not a comment to **configparser** by default
 - timeout = 10 ; seconds** stores the string **10 ; seconds**, and **getint** then fails

AGENT.INI

```
; inventory sync agent
[api]
url = https://inv.lab.test:8443
token = s3cr3t
timeout = 10

[logging]
level = info
file = /var/log/invsync/%(level)s.log
```

- %(level)s** is **interpolation**: Python's **configparser** swaps in another key's value
 - A literal **%** must be written **%%**
- Four broken files and what **configparser** says about each

TERMINAL

```
$ cat noeq.ini
[api]
url = https://inv.lab.test:8443
timeout = 10
retries 3
```

TERMINAL

```
$ ./check-config.py nohdr.ini noeq.ini dup.ini pct.ini
nohdr.ini: MissingSectionHeaderError: File contains no section headers.
file: 'nohdr.ini', line: 1
'url = https://inv.lab.test:8443\n'
noeq.ini: ParsingError: Source contains parsing errors: 'noeq.ini'
[line 4]: 'retries 3\n'
dup.ini: DuplicateOptionError: While reading from 'dup.ini' [line 4]: option 'timeout' in section 'api' already exists
pct.ini: InterpolationSyntaxError: '%' must be followed by '%' or '(', found: '%word'
```

- nohdr.ini**: a key before the first **[section]**
- noeq.ini**: a line with no **=** or **:**
- dup.ini**: **timeout** set twice in one section (strict mode, the default)
- pct.ini**: a password containing **%**; write **P@ss%word**

YAML

- Structure comes from **indentation**: spaces only, never tabs
 - Two spaces per level is the convention; it just has to be consistent
- key: value** needs a space after the colon

2.4 Configuration File Syntax: INI, YAML, and JSON (continued)

- A list item starts with `-`
- Quote strings that contain `:`, start with a special character (`{`, `[`, `*`, `&`, `!`, `%`, `@`), or look like another type
- `---` marks the start of a document; `#` starts a comment
- Ansible playbooks and inventories, Kubernetes manifests, Docker Compose, GitHub Actions and GitLab CI pipelines

VLANS.YML

```
---
site: charlotte-dc1
vlangs:
  - id: 10
    name: users
  - id: 20
    name: servers
    description: "Rack 4: east wing"
```

TERMINAL

```
$ cat -A tab.yml | sed -n 4p
^Iname: users$
```

- `cat -A` shows a tab as `^I`; most editors hide it

TERMINAL

```
$ ./check-config.py tab.yml colon.yml indent.yml nospace.yml
tab.yml: ScannerError: while scanning for the next token
found character '\t' that cannot start any token
  in "tab.yml", line 4, column 1
colon.yml: ScannerError: mapping values are not allowed here
  in "colon.yml", line 4, column 24
indent.yml: ParserError: while parsing a block collection
  in "indent.yml", line 3, column 3
expected <block end>, but found '<block mapping start>'
  in "indent.yml", line 6, column 4
nospace.yml: ScannerError: mapping values are not allowed here
  in "nospace.yml", line 4, column 12
```

- `colon.yml`: `description: Rack 4: east wing` without quotes; the second `:` looks like another key
- `indent.yml`: `name:` indented three spaces under a list item whose keys sit at four
- `nospace.yml`: `hostname:core-sw1` has no space after the colon, so it is a plain string, not a key; the error lands on the next line
- Valid YAML can still be wrong: PyYAML follows YAML 1.1 type rules

SITE.YML

```
---
site:
  country: NO
  file_mode: 0644
  os_version: 7.10
  lldp: on
  vlan:
    name: users
    name: staff
```

TERMINAL

```
$ python3 -c "import yaml; print(yaml.safe_load(open('site.yml')))"
{'site': {'country': False, 'file_mode': 420, 'os_version': 7.1, 'lldp': True, 'vlan': {'name': 'staff'}}}
$ yamllint site.yml
site.yml
 3:12      warning  truthy value should be one of [false, true]  (truthy)
 6:9       warning  truthy value should be one of [false, true]  (truthy)
 9:5       error    duplication of key "name" in mapping  (key-duplicates)
```

- Norway became `False`, the file mode became octal `420`, version `7.10` became `7.1`, and the first `name` silently vanished
 - Fix: quote them (`"NO"`, `"0644"`, `"7.10"`) and run `yamllint` in CI, which catches what the parser accepts
 - Ansible loads YAML with PyYAML, which is why its docs tell you to quote file modes: `mode: "0644"`

2.4 Configuration File Syntax: INI, YAML, and JSON (continued)

JSON

- Objects `{}` of `"key": value` pairs and arrays `[]`, separated by commas
- Strict rules, and each breaks a parse
 - Keys and strings in **double quotes** only
 - No **trailing comma** after the last item
 - No comments
 - Lowercase `true`, `false`, `null`
- REST API bodies, `package.json`, AWS IAM policies, Terraform `.tf.json`, VS Code settings

MISTAKE	EXAMPLE	JQ REPORTS	PYTHON3 -M JSON.TOOL REPORTS
Trailing comma	<code>{"id": 10, "name": "users"}, then]</code>	Expected another array element at line 4, column 3	Illegal trailing comma before end of array: line 3 column 32 (char 46)
Single quotes	<code>'id': 10</code>	Invalid numeric literal at line 2, column 7	Expecting property name enclosed in double quotes: line 2 column 3 (char 4)
Comment	<code>// core VLAN</code>	Invalid numeric literal at line 2, column 5	Expecting property name enclosed in double quotes: line 2 column 3 (char 4)
Missing comma	<code>"id": 10 then "name": ...</code>	Expected separator between values at line 3, column 8	Expecting ',' delimiter: line 3 column 3 (char 15)
Capital boolean	<code>"voice": False</code>	Invalid numeric literal at line 4, column 0	Expecting value: line 3 column 12 (char 25)

- The two tools disagree on positions; neither always points at the bad character
- The trailing-comma message is new in Python 3.13; older versions say **Expecting value** or **Expecting property name**
- `jq empty` prints nothing and exits `0` on valid JSON, so it works as a CI gate

TERMINAL

```
$ jq empty vlans.json && echo valid
valid
```

- JSON is valid YAML (YAML 1.2 is a superset), so a YAML parser reads a JSON file; the reverse does not work

Side by side

	INI	YAML	JSON
Structure	Sections and <code>key = value</code>	Indentation, <code>key: value</code> , <code>- item</code>	Braces, brackets, commas
Nesting	One level (sections)	Unlimited	Unlimited
Data types	Strings only	Strings, numbers, booleans, null, lists, maps (inferred)	Strings, numbers, <code>true</code> / <code>false</code> , <code>null</code> , arrays, objects
Comments	<code>;</code> or <code>#</code>	<code>#</code>	None
Whitespace matters	No	Yes: indentation is the structure	No
Classic mistake	Duplicate key, missing section	Tab, bad indent, unquoted <code>:</code>	Trailing comma, single quotes
Check with	<code>configparser</code> , the app's own loader	<code>yamllint</code> , <code>yaml.safe_load</code>	<code>jq empty</code> , <code>python3 -m json.tool</code>

ON THE EXAM

- Expect a short file and a question about why it fails: a tab or uneven indent in YAML, a trailing comma or single quotes in JSON, a key outside any section in INI.
- "Which format allows comments?" INI and YAML

TRY IT

Save `site.yml` from this page, run `yamllint` on it, quote the values it warns about, delete the duplicate key, and run it again until it is clean.

2.4 Configuration File Syntax: INI, YAML, and JSON (continued)

yes, JSON no. "Which uses indentation for structure?" YAML.

- A YAML file that parses but gives the wrong value (`NO` to false, `0644` to 420) is fixed by quoting; a linter such as `yamllint` flags it.