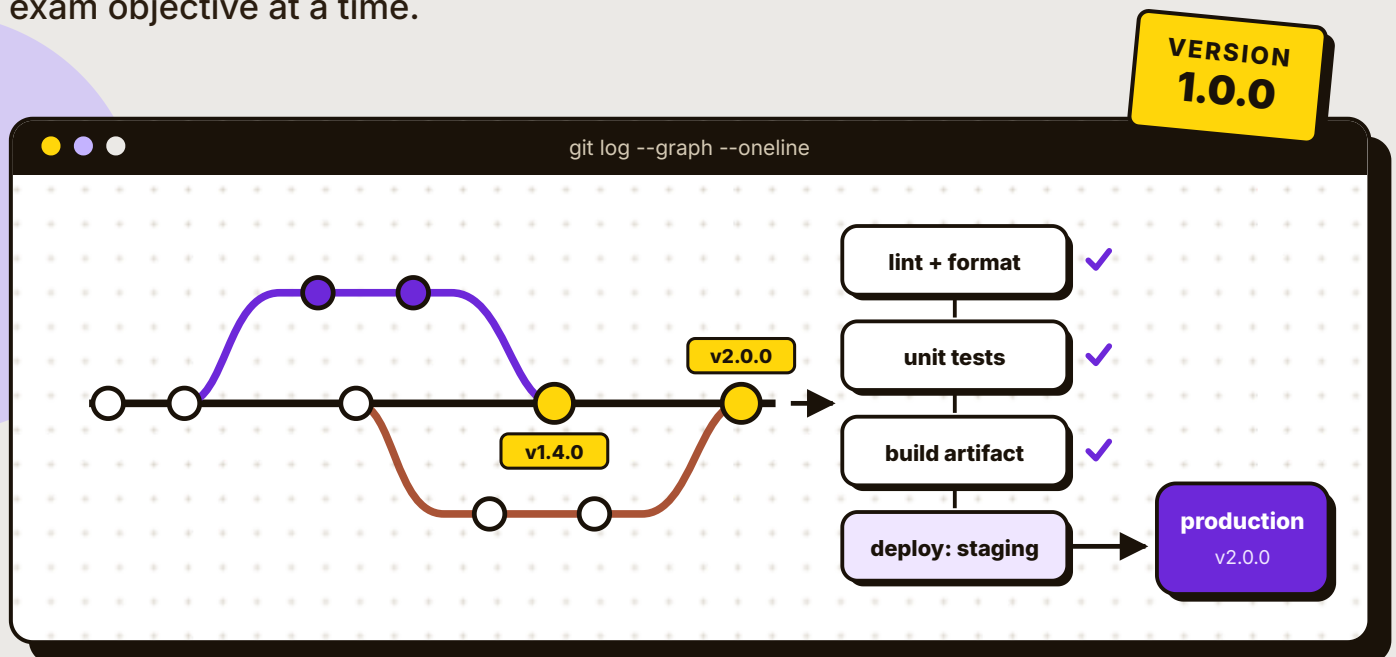


COMPTIA EXPANSION SERIES • EXAM AT0-001 V1

AutoOps+ Course Notes

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



CompTIA AutoOps+ AT0-001 Course Notes

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

1.0 Automation Coding Concepts 31% of exam · page 1

1.1	Variables, Conditionals, and Functions	2
1.1	Data Types: Primitives, JSON, and YAML	4
1.1	Regular Expressions and Application Logs	7
1.1	Command-Line Text Tools	9
1.1	Dependency Management	12
1.2	Git Fundamentals	15
1.2	Working with Remotes	17
1.2	Semantic Versioning	18
1.2	Branching Strategies	20
1.2	The Commit Life Cycle	22
1.3	Infrastructure as Code Principles	24
1.3	Declarative, Imperative, and Idempotent	26
1.4	Syntax and Runtime Errors	28
1.4	Troubleshooting Git	31

2.0 System Configuration 25% of exam · page 34

2.1	Drift Detection and State Management	35
2.1	Workload Configuration: Certificates and ACLs	37
2.1	Virtual Network Infrastructure as Code	40
2.2	Remote and Local Automation	43
2.2	Environments, Migrations, and Provisioning	45
2.2	Event-Driven Automation	48
2.3	REST APIs: Methods, Headers, and Status Codes	50
2.3	Working with REST APIs: curl, wget, and Postman	54
2.4	Troubleshooting System Configuration	57
2.4	Configuration File Syntax: INI, YAML, and JSON	61

3.0 Continuous Integration 24% of exam · page 65

3.1	Secrets Management	66
3.1	Artifact Management and SBOMs	68
3.1	Task Runners and Environment Provisioning	71
3.2	Pipeline Orchestration	73
3.2	Pipeline Enhancements: Scanning, Containers, and AI	76
3.3	Webhooks, ChatOps, and Pipeline Triggers	79
3.3	Pipeline Definitions: Jenkins, GitLab, and GitHub	83

4.0 Continuous Delivery 20% of exam · page 87

4.1	Deployment Strategies	88
-----	-----------------------	----

4.1	QA Testing, Validation, and Hotfixes	90
4.2	Service Levels: Uptime, SLOs, SLAs, MTTR, MTBF	93
4.3	Securing Connections to Providers	95

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.

01

Automation Coding Concepts

31% of the exam

-
- 1.1** Given a scenario, use code to support automation
 - 1.2** Given a scenario, use source control techniques to version code
 - 1.3** Explain concepts related to infrastructure as code (IaC)
 - 1.4** Given a scenario, troubleshoot common issues with the code life cycle
-

Variables and scope

- A **variable** is a name bound to a value: `hostname`, `vlan_id`, `retries`
 - Change the value in one place; every line that uses it follows
- Syntax: Python `vlan = 20`; PowerShell `$vlan = 20`; Bash `vlan=20` (no spaces around `=`), read as `"$vlan"`
 - Bash stores everything as a string; math needs `=$((vlan + 1))`
- Scope** decides which parts of a script can see a variable
 - Global**: defined at the top level of the script; readable from every function
 - Local**: created inside a function; gone when the function returns
 - A name not found locally is looked up in the global scope

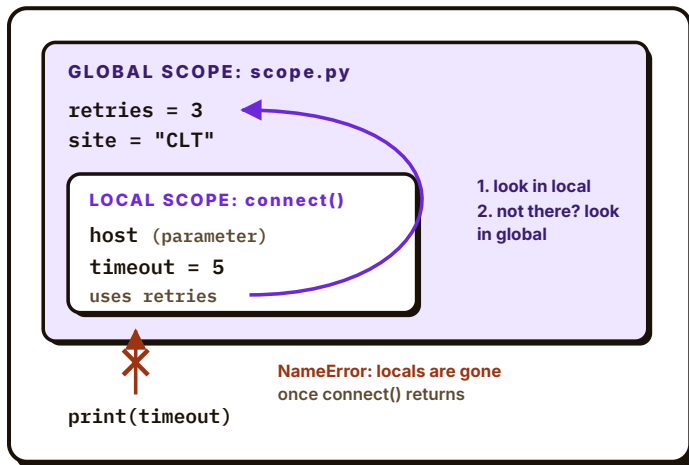


FIGURE A function sees its own locals first, then globals. Nothing outside the function can see its locals.

- Python: assigning inside a function creates a local
 - To change a global, declare `global retries` first; without it, `retries += 1` fails with `UnboundLocalError`

- Bash: the opposite default
 - Every variable is global, even one set inside a function
 - Declare `local` so a function cannot overwrite the caller's values
- PowerShell: functions get a child scope; write back with `$script:name` or `$global:name`

```
SCOPE.PY

retries = 3                                # global

def connect(host):
    timeout = 5                            # local to connect()
    return (f"{host} retries={retries} "
            f"timeout={timeout}")

def bump():
    global retries                          # rebind the global
    retries += 1

print(connect("core-sw1"))
bump()
print(connect("core-sw1"))
print(timeout)                             # not visible here
```

```
TERMINAL

$ python3 scope.py
core-sw1 retries=3 timeout=5
core-sw1 retries=4 timeout=5
NameError: name 'timeout' is not defined
```

Conditional statements

- Conditional statements** run a block only when a test is true
 - `if`, optional `elif` (Python, Bash) or `elseif` (PowerShell), optional `else`
 - Multi-way: `case` in Bash, `match` in Python 3.10+, `switch` in PowerShell
- In Bash the condition is a command's **exit status**
 - `0` means true (success); anything else means false
 - `if grep -q shutdown sw1.cfg; then` branches on whether grep matched
- Python truthiness: `0`, `""`, `[]`, `{}`, and `None` count as false

1.1 Variables, Conditionals, and Functions (continued)

TEST	PYTHON	BASH	POWERSHELL
Numbers equal	<code>a == b</code>	<code>[[\$a -eq \$b]]</code>	<code>\$a -eq \$b</code>
Strings equal	<code>a == b</code>	<code>[[\$a == "\$b"]]</code>	<code>\$a -eq \$b</code> (ignores case); <code>-ceq</code> matches case
Greater than	<code>a > b</code>	<code>[[\$a -gt \$b]]</code>	<code>\$a -gt \$b</code>
And / or / not	<code>and</code> , <code>or</code> , <code>not</code>	<code>&&</code> , <code> </code> , <code>!</code>	<code>-and</code> , <code>-or</code> , <code>-not</code>
File exists	<code>os.path.exists(p)</code>	<code>[[-f \$p]]</code>	<code>Test-Path \$p</code>

Iterations

- An **iteration** (loop) repeats a block: one block, 400 switches
- A **for loop** walks a known collection: hosts, VLANs, files, lines of output
- A **while loop** repeats until a condition changes: poll a service, retry a login
 - Always cap the attempts so a dead host cannot hang the pipeline
- break** leaves the loop; **continue** skips to the next item
- Bash `while read -r line; do ...; done < hosts.txt` reads a file line by line

CHECK_HOSTS.SH

```
#!/usr/bin/env bash
# Report which inventory hosts answer on a TCP port
PORT=${1:-22}          # parameter, default 22
HOSTS=(127.0.0.1 127.0.0.2 192.0.2.10)
up=0                   # global counter

probe() {
    # $1 = host, $2 = port
    local host=$1 port=$2
    timeout 1 bash -c "</dev/tcp/$host/$port" \
        2>/dev/null
}

for h in "${HOSTS[@]"; do
    if probe "$h" "$PORT"; then
        echo "$h:$PORT up"
        up=$((up + 1))
    else
        echo "$h:$PORT down"
    fi
done
echo "$up of ${#HOSTS[@]} hosts up"
```

TERMINAL

```
$ ./check_hosts.sh 18100
127.0.0.1:18100 up
127.0.0.2:18100 down
192.0.2.10:18100 down
1 of 3 hosts up
```

Parameters and functions

- A **function** is a named, reusable block: write the logic once, call it for every device
- Parameters** are the inputs a function declares; **arguments** are the values passed in a call
 - Positional: matched by order (`probe "$h" "$PORT"`)
 - Keyword or named: matched by name (`attempts=2, -Vlan 20`)
 - A default value makes a parameter optional
- Return values
 - Python and PowerShell return objects to the caller
 - A Bash function returns only an exit status (0 to 255); print data with `echo` and capture it with `${...}`
- Scripts take parameters too: Bash `$1`, `"$@"`, `$#`; Python `sys.argv` or `argparse`; PowerShell a `param()` block

WAIT_FOR.PY

```
import socket
import time

def wait_for(host, port, attempts=5, delay=0.5):
    n = 0
    while n < attempts:
        n += 1
        try:
            socket.create_connection(
                (host, port), timeout=1).close()
            return n          # attempts used
        except OSError:
            time.sleep(delay)
    return None              # never came up

print(wait_for("127.0.0.1", 18100))
print(wait_for("127.0.0.1", 18101, attempts=2))
```

TERMINAL

```
$ python3 wait_for.py
1
None
```

1.1 Variables, Conditionals, and Functions (continued)

- PowerShell parameters validate themselves before the body runs
 - `[Parameter(Mandatory)]`, `[ValidateRange()]`, `[ValidateSet()]`

SUBNET.PS1

```
function Get-Subnet {
    param(
        [Parameter(Mandatory)][string]$Site,
        [ValidateRange(1, 4094)][int]$Vlan = 10
    )
    "$Site VLAN $Vlan -> 10.$Vlan.0.0/24"
}
foreach ($s in 'CLT', 'RDU') { Get-Subnet -Site $s }
Get-Subnet -Site CLT -Vlan 20
Get-Subnet -Site CLT -Vlan 5000
```

TERMINAL

```
$ pwsh -NoProfile -File subnet.ps1
CLT VLAN 10 -> 10.10.0.0/24
RDU VLAN 10 -> 10.10.0.0/24
CLT VLAN 20 -> 10.20.0.0/24
Get-Subnet: subnet.ps1:10
| Cannot validate argument on parameter 'Vlan'.
The 5000 argument is greater than the maximum
allowed range of 4094.
```

ON THE EXAM

- Expect a short script and a question about its output. Trace scope: a Bash variable set in a function without `local` changes the global; a Python assignment in a function creates a new local unless `global` is declared.
- In Bash, `if` tests an exit status, and `0` is success. A distractor will treat `0` as false.
- Parameters are declared, arguments are passed; a `for` loop walks a list, a `while` loop runs until a condition changes.

TRY IT

Remove `local` from `probe()` in `check_hosts.sh`, add `echo "$host"` after the loop, and run it. The last host probed leaks into the global scope.

1.1 Data Types: Primitives, JSON, and YAML

1.0 AUTOMATION CODING CONCEPTS

Primitive data types

- A data type tells the language what a value is and what you can do with it
- Primitive** types hold a single value
 - String**: text, such as a hostname or an interface name
 - Integer**: whole numbers, such as a VLAN ID or a port
 - Float**: numbers with a fraction, such as a CPU load of `0.73`
 - Boolean**: true or false, such as `stp_enabled`
 - Null: "no value," such as an interface with no uplink
- Collections hold many values
 - List (array): ordered values, such as `[10, 20, 30]`
 - Dictionary (map, object): key and value pairs, such as `{"hostname": "core-sw1"}`

TYPE	PYTHON	JSON	YAML
String	<code>"core-sw1"</code> (<code>str</code>)	<code>"core-sw1"</code>	<code>core-sw1</code> or <code>"core-sw1"</code>
Integer	<code>9216</code> (<code>int</code>)	<code>9216</code>	<code>9216</code>
Float	<code>0.73</code> (<code>float</code>)	<code>0.73</code>	<code>0.73</code>
Boolean	<code>True</code> / <code>False</code>	<code>true</code> / <code>false</code>	<code>true</code> / <code>false</code>
Null	<code>None</code>	<code>null</code>	<code>null</code> or <code>~</code>
List	<code>[10, 20]</code>	<code>[10, 20]</code>	<code>- 10</code> on its own line, or <code>[10, 20]</code>
Dictionary	<code>{"mtu": 9216}</code>	<code>{"mtu": 9216}</code>	<code>mtu: 9216</code>

1.1 Data Types: Primitives, JSON, and YAML (continued)

- Python is dynamically but strongly typed: mixing types needs a conversion such as `int()` or `str()`
- Bash has no real types; every value is a string until `$(())` treats it as a number

TERMINAL

```
$ python3 -c 'print("10" + 1)'
TypeError: can only concatenate str (not "int") to str
$ python3 -c 'print(int("10") + 1)'
11
```

JavaScript Object Notation (JSON)

- JSON** is the format most REST APIs send and receive
 - Also `package.json`, Terraform state, cloud CLI output
- Objects in braces `{}`, arrays in brackets `[]`, `"key": value` pairs separated by commas
- The rules are strict
 - Keys and strings use **double quotes** only
 - No trailing commas** after the last item
 - No comments**
 - `true`, `false`, and `null` are lowercase
- Whitespace and indentation do not matter

SWITCH.JSON

```
{
  "hostname": "core-sw1",
  "mgmt_ip": "10.0.0.11",
  "vlans": [10, 20, 30],
  "stp_enabled": true,
  "uplink": null,
  "mtu": 9216
}
```

- One stray comma breaks the whole document

TERMINAL

```
$ echo '{"vlan": 10,}' | jq .
jq: parse error: Expected another key-value pair at
line 1, column 13
```

YAML Ain't Markup Language (YAML)

- YAML** is the format people write by hand
 - Ansible playbooks, Kubernetes manifests, Docker Compose, CI pipelines
- Structure comes from **indentation**: spaces only, never tabs
- `key: value` makes a mapping; `- item` makes a list entry; `#` starts a comment
- Strings usually need no quotes

- `---` starts a new document; `|` keeps line breaks in a block of text (a login banner)
- YAML 1.2 is a superset of JSON: a YAML parser can read a JSON file

SWITCH.YAML

```
# core switch facts
hostname: core-sw1
mgmt_ip: 10.0.0.11
vlans:
  - 10
  - 20
  - 30
stp_enabled: true
uplink: null
mtu: 9216
```

The same data in both formats

- `switch.json` and `switch.yaml` above load into the **same Python dictionary**
 - JSON for machine-to-machine traffic; YAML for files people edit and review

LOAD.PY

```
import json
import yaml # PyYAML: pip install PyYAML

with open("switch.json") as f:
    from_json = json.load(f)
with open("switch.yaml") as f:
    from_yaml = yaml.safe_load(f)

print("same data:", from_json == from_yaml)
for key, value in from_yaml.items():
    print(f"{key}:<12> {type(value).__name__}")
```

TERMINAL

```
$ python3 load.py
same data: True
hostname      str
mgmt_ip       str
vlans         list
stp_enabled   bool
uplink        NoneType
mtu           int
```

- `mgmt_ip` is a string; neither format has an IP address type
- Use `yaml.safe_load()` (the **safe loader**); the full loader can build arbitrary Python objects from untrusted input
- `yq -o json switch.yaml` (Go `yq`) converts YAML to JSON for `jq`

1.1 Data Types: Primitives, JSON, and YAML (continued)

■ YAML gotchas

- `show.py` prints each key with the value and type PyYAML produced

```
import sys
import yaml
```

SHOW.PY

```
with open(sys.argv[1]) as f:
    data = yaml.safe_load(f)
for key, value in data.items():
    kind = type(value).__name__
    print(f"{key:<10} {value!r} ({kind})")
```

- Wrong indentation can change the structure with no error: one extra space merges two list items

```
vlan:
- 10
- 20
```

VLANS.YAML

```
$ python3 show.py vlans.yaml
vlans      ['10 - 20'] (list)
```

TERMINAL

- A tab used for indentation fails outright: **found character '\t' that cannot start any token**
- Unquoted values are **type-guessed** by the parser
 - YAML 1.1 parsers turn `yes`, `no`, `on`, `off` into booleans
 - The **Norway problem**: country code `NO` becomes `false`
 - `1.10` is a float, which prints as `1.1`
 - YAML 1.1 reads `22:22` as base-60 arithmetic
- PyYAML (6.0.3, the parser Ansible uses) follows YAML 1.1

```
country: NO
vlan_name: off
version: 1.10
ssh_map: 22:22
quoted: "NO"
```

SITE.YAML

TERMINAL

```
$ python3 show.py site.yaml
country      False (bool)
vlan_name    False (bool)
version      1.1 (float)
ssh_map      1342 (int)
quoted       'NO' (str)
```

- YAML 1.2 parsers such as `ruamel.yaml` keep `NO`, `off`, and `22:22` as strings
 - `1.10` is still a float in every version
- The fix is the same everywhere: **quote anything that must stay a string**
 - Country codes, version numbers, port mappings, leading-zero IDs like `"0042"`
- `yamllint` warns on `yes / no / on / off` (its **truthy** rule) and bad indentation

ON THE EXAM

- Given a JSON snippet that fails to parse, look for a trailing comma, single quotes, a comment, or capitalized `True`. Those are Python habits that JSON rejects.
- Given YAML that "loads the wrong value," suspect indentation first, then an unquoted value that the parser converted to a boolean or number.
- Know which primitive each value becomes: `"10"` is a string, `10` is an integer, `true` is a Boolean, `null` is null.

TRY IT

Add `"country": "NO"` to `switch.json` and `country: NO` to `switch.yaml`, then rerun `load.py`. It prints `same data: False`. Quote the YAML value and run it again.

1.1 Regular Expressions and Application Logs

1.0 AUTOMATION CODING CONCEPTS

Application logs

- **Application logs** are the record a program writes about what it did
 - First stop when a playbook, pipeline job, or service misbehaves; automation reads them too (count failures, alert on a pattern)
- A useful line carries a UTC timestamp, a **log level**, the host or component, the message, and a job or request ID
- Where logs go
 - Containers: stdout and stderr, read with **docker logs**
 - systemd services: the journal; **journalctl -u nginx -p err** shows only errors
 - Files under **/var/log/**, rotated by **logrotate**; central platforms (syslog, Elastic, Loki, Splunk) collect from every host
- Never log secrets: passwords, API tokens, session cookies

Log levels

- A **log level** ranks how serious a message is
- The configured level is a **threshold**: that level and everything more severe is written; anything less severe is dropped
 - Production usually runs at INFO or WARNING
 - Switch to DEBUG while troubleshooting, then switch back (DEBUG is noisy and may expose data)

SYSLOG CODE	SYSLOG SEVERITY	PYTHON LOGGING
0	Emergency	
1	Alert	
2	Critical	CRITICAL (50)
3	Error	ERROR (40)
4	Warning	WARNING (30)
5	Notice	
6	Informational	INFO (20)
7	Debug	DEBUG (10)

- Syslog: a **lower number is more severe**; Python: a higher number is more severe
- Python's **logging** writes to stderr by default

JOB.PY

```
import logging
import os

logging.basicConfig(
    level=os.environ.get("LOG_LEVEL", "INFO"),
    format="%(levelname)-8s %(name)s: %(message)s",
)
log = logging.getLogger("backup")
log.debug("opening SSH to core-sw1")
log.info("core-sw1 saved 412 lines")
log.warning("dist-sw2 slow reply 4.8s")
log.error("edge-rtr1 auth failed")
```

TERMINAL

```
$ python3 job.py
INFO      backup: core-sw1 saved 412 lines
WARNING   backup: dist-sw2 slow reply 4.8s
ERROR     backup: edge-rtr1 auth failed
$ LOG_LEVEL=ERROR python3 job.py
ERROR     backup: edge-rtr1 auth failed
$ LOG_LEVEL=DEBUG python3 job.py 2>&1 | head -1
DEBUG     backup: opening SSH to core-sw1
```

Structured vs. unstructured logs

- **Unstructured logs** are free text written for people
 - Easy to read; hard to parse reliably
 - Pulling out a field takes a regular expression, and a format change breaks it
- **Structured logs** write every event as fields, usually one JSON object per line (JSON Lines)
 - Machines filter by field name; no pattern guessing
 - Log platforms index the fields for search and dashboards

BACKUP.LOG (UNSTRUCTURED)

```
08:14:02 INFO core-sw1 saved 412 lines
08:14:05 WARNING dist-sw2 slow reply 4.8s
08:14:09 ERROR edge-rtr1 auth failed from 10.0.4.17
08:14:11 INFO dist-sw3 saved 388 lines
08:14:20 ERROR edge-rtr2 timeout after 30s
08:14:26 ERROR edge-rtr1 auth failed from 10.0.4.17
```

BACKUP.JSONL (STRUCTURED)

```
{"level": "info", "host": "core-sw1", "lines": 412}
{"level": "error", "host": "edge-rtr1", "err": "auth"}
{"level": "error", "host": "edge-rtr2", "err": "timeout"}
```

1.1 Regular Expressions and Application Logs (continued)

TERMINAL

```
$ jq -r 'select(.level=="error").host' backup.jsonl
edge-rtr1
edge-rtr2
```

Regular expression building blocks

- A **regular expression (regex)** is a pattern that describes text
 - Search (does a line match?), extract (pull out the IP), validate (is this a hostname?), replace

PATTERN	MATCHES	EXAMPLE
.	any single character	Gi0.1 matches Gi0/1 and Gi0.1
\d \w \s	a digit; a letter, digit, or _; whitespace	\d+ matches 412
[A-Z] [^]	one character from a set; any character not in it	[0-9a-f] is one hex digit
^ \$	start of line; end of line	^interface
* + ?	previous item 0 or more times; 1 or more; 0 or 1	vlan? matches vlan and vlans
{3} {1,3}	exactly 3; between 1 and 3	\d{1,3} is one IPv4 octet
()	group, and capture what matched	(\d+) lines captures 412
a b	either side	ERROR CRITICAL
\. \b	a literal dot; a word boundary	10\.0\.4\.17

- * and + are **greedy**: they take as much text as possible
 - Add ? for the shortest match: ".*?" stops at the first closing quote
- Escape characters that have a meaning (. * + ? () [] { } | \ ^ \$) when you mean them literally
 - An unescaped . in 10.0.4.17 also matches 10a0b4c17
- Anchor patterns with ^ and \$ when validating whole values

Extracting fields from log lines

- Capture groups** turn one unstructured line into named fields
 - Python (?P<name>...) makes a **named group**; read it back as m["name"]

PARSE_LOG.PY

```
import re
from collections import Counter

LINE = re.compile(
    r"^(?P<time>\d{2}:\d{2}:\d{2}) "
    r"(?P<level>[A-Z]+) "
    r"(?P<host>[\w-]+) "
    r"(?P<msg>.*)$"
)

levels = Counter()
with open("backup.log") as f:
    for line in f:
        m = LINE.match(line.rstrip("\n"))
        if not m:
            continue # skip other formats
        levels[m["level"]] += 1
        if m["level"] == "ERROR":
            print(f'{m["host"]}:<10} {m["msg"]}')
print(dict(levels))
```

TERMINAL

```
$ python3 parse_log.py
edge-rtr1 auth failed from 10.0.4.17
edge-rtr2 timeout after 30s
edge-rtr1 auth failed from 10.0.4.17
{'INFO': 2, 'WARNING': 1, 'ERROR': 3}
```

1.1 Regular Expressions and Application Logs (continued)

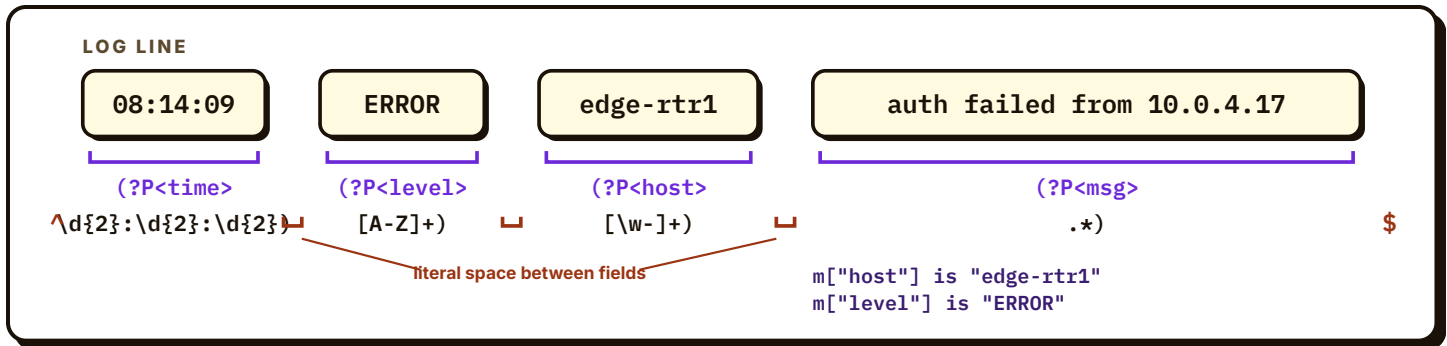


FIGURE Each named group consumes one field. The spaces between groups are literal and anchor the next field.

Regex flavors on the command line

- Tools speak different dialects
 - Basic (BRE):** `grep`, `sed` by default; `+ ? | ( ) { }` need a backslash to be special
 - Extended (ERE):** `grep -E`, `sed -E`, `awk`; those characters work bare
 - Perl-compatible (PCRE):** `grep -P`; adds `\d`, lazy `*?`, lookarounds; Python `re` and PowerShell `-match` use similar Perl-style syntax
- `\d` is not part of BRE or ERE
 - GNU `grep -E '\d'` matches the letter `d`, so the count below is 0

TERMINAL

```
$ grep -c ERROR backup.log
3
$ grep -cE '^d' backup.log
0
$ grep -cP '^d' backup.log
6
$ grep -oE '([0-9]{1,3}\.){3}[0-9]{1,3}' backup.log
10.0.4.17
10.0.4.17
```

ON THE EXAM

- Expect to pick the regex that matches a sample line or extracts one field. Check anchors, escaped dots, and quantifiers: `\d{1,3}` versus `\d+` versus `\d*`.
- Know the severity order. Syslog 0 (Emergency) is the most severe and 7 (Debug) the least; setting a level shows that level and everything more severe.
- Structured (JSON) logs are filtered by field with a tool like `jq`; unstructured text needs a regex.

TRY IT

Add a line `08:15:00 CRITICAL core-sw1 PSU 2 failed to backup.log` and rerun `parse_log.py`. Change the script so CRITICAL lines print too.

1.1 Command-Line Text Tools

1.0 AUTOMATION CODING CONCEPTS

Why text tools still matter

- Configs, `show` output, logs, and API replies are all text; small **command-line tools** each do one job and chain together with **pipes**
- The files used below

1.1 Command-Line Text Tools (continued)

SW1.CFG

```
hostname core-sw1
!
ntp server 10.0.0.5
snmp-server community public R0
!
interface Gi0/1
  description uplink to dist-sw2
!
interface Gi0/2
  description spare
  shutdown
!
```

INV.CSV

```
name,site,ip
core-sw1,CLT,10.0.0.11
dist-sw2,CLT,10.0.0.12
edge-rtr1,RDU,10.1.0.1
```

TERMINAL

```
$ awk -F, 'NR > 1 {print $3, $1}' inv.csv
10.0.0.11 core-sw1
10.0.0.12 dist-sw2
10.1.0.1 edge-rtr1
$ awk -F, '$2 == "CLT" {n++} END {print n}' inv.csv
2
```

INV.JSON

```
[
  {
    "name": "core-sw1",
    "site": "CLT",
    "status": "active",
    "vlans": [10, 20]
  },
  {
    "name": "dist-sw2",
    "site": "CLT",
    "status": "offline",
    "vlans": [10]
  },
  {
    "name": "edge-rtr1",
    "site": "RDU",
    "status": "active",
    "vlans": []
  }
]
```

grep: find lines

- **grep** prints lines that match a pattern
 - **-i** ignore case; **-v** invert (non-matching lines); **-c** count; **-n** line numbers
 - **-E** extended regex; **-o** only the matching part; **-r** search a folder; **-l** file names only
 - **-A 2 / -B 2**: lines after / before each match, for context
- **Exit status** 0 if anything matched, 1 if not
 - **grep -q** prints nothing and is made for **if** tests

TERMINAL

```
$ grep -n shutdown sw1.cfg
11: shutdown
$ grep -c '^interface' sw1.cfg
2
```

awk: work with columns

- **awk** splits each line into **fields**: **\$1**, **\$2**, ..., **\$NF** (the last)
 - Splits on whitespace by default; **-F**, splits on commas
- A program is **pattern { action }**: the action runs only on matching lines
 - **NR** is the line number, so **NR > 1** skips a header row
 - **END { }** runs after the last line (totals)

sed: edit a stream

- **sed** edits text line by line as it flows through
 - **s/old/new/** substitutes the first match on each line; add **g** for every match
 - **/pattern/d** deletes matching lines
 - **-n '/start/,/end/p'** prints only a range of lines
- Output goes to the screen; the file is untouched unless you add **-i**
 - **-i.bak** edits **in place** and keeps a backup copy
 - Escape the dots in IP addresses, or **10.0.0.5** also matches **10a0b0c5**

TERMINAL

```
$ sed -n '/^interface Gi0\|2/,/^!/p' sw1.cfg
interface Gi0/2
  description spare
  shutdown
!
$ sed -i.bak 's/10\.0\.0\.5/10.0.0.6/' sw1.cfg
$ diff sw1.cfg.bak sw1.cfg
3c3
< ntp server 10.0.0.5
---
> ntp server 10.0.0.6
```

jq: query JSON

- **jq** filters and reshapes JSON, the format of nearly every REST API reply
 - **.** the whole document; **.name** one key; **.[]** each array item; **|** chains steps
 - **select(condition)** keeps only matching items
 - **-r raw output** (no quotes) for Bash loops; **-c** compact output
 - Many APIs wrap the list in a key: NetBox replies use **.results[]**

1.1 Command-Line Text Tools (continued)

TERMINAL

```
$ jq length inv.json
3
$ jq -r '.[.] | select(.site=="CLT") | .name'
inv.json
core-sw1
dist-sw2
$ jq -c '.[.].vlans[] | unique' inv.json
[10,20]
```

PowerShell cmdlets

- A **cmdlet** is a PowerShell command named **Verb-Noun**:
Get-Content, **Select-String**, **Where-Object**,
ConvertFrom-Json
- PowerShell pipes **objects**, not text
 - ConvertFrom-Json** returns objects with properties;
no field counting
 - \$_** is the current object inside a script block
- PowerShell 7 (**powershell**) runs on Windows, Linux, and macOS

TASK	LINUX TOOL	POWERSHELL CMDLET
Read a file	<code>cat file</code>	<code>Get-Content file</code>
Find matching lines	<code>grep pattern file</code>	<code>Select-String -Pattern p -Path file</code>
Keep matching rows	<code>awk '\$4 == "up"'</code>	<code>Where-Object { \$_.Status -eq 'up' }</code>
Pick columns or properties	<code>awk '{print \$1}'</code>	<code>Select-Object Name</code>
Replace text	<code>sed 's/a/b/'</code>	<code>\$text -replace 'a', 'b'</code>
Parse JSON	<code>jq</code>	<code>ConvertFrom-Json</code> / <code>ConvertTo-Json</code>
Count, group	<code>wc -l</code> , <code>sort uniq -c</code>	<code>Measure-Object</code> , <code>Group-Object</code>

REPORT.PS1

```
Select-String -Path ./sw1.cfg -Pattern 'shutdown' |
  ForEach-Object { "line $($_.LineNumber)" }

$devices = Get-Content ./inv.json -Raw |
  ConvertFrom-Json
$devices |
  Where-Object { $_.status -ne 'active' } |
  ForEach-Object { $_.name }
```

TERMINAL

```
$ powershell -NoProfile -File report.ps1
line 11
dist-sw2
```

TERMINAL

```
$ head -1 auth.log
Oct 19 08:01:12 bastion sshd[811]: Failed password
for admin from 203.0.113.9 port 50122 ssh2
```

TOP_IPS.SH

```
#!/usr/bin/env bash
set -o pipefail
grep 'Failed password' auth.log |
  awk '{print $(NF-3)}' |
  sort |
  uniq -c |
  sort -rn |
  head -3
```

TERMINAL

```
$ ./top_ips.sh
4 203.0.113.9
2 198.51.100.23
1 10.0.4.17
```

Pipelines

- The **pipe** | sends one command's standard output into the next command's standard input
- Example: which addresses are failing SSH logins on a bastion host?

1.1 Command-Line Text Tools (continued)

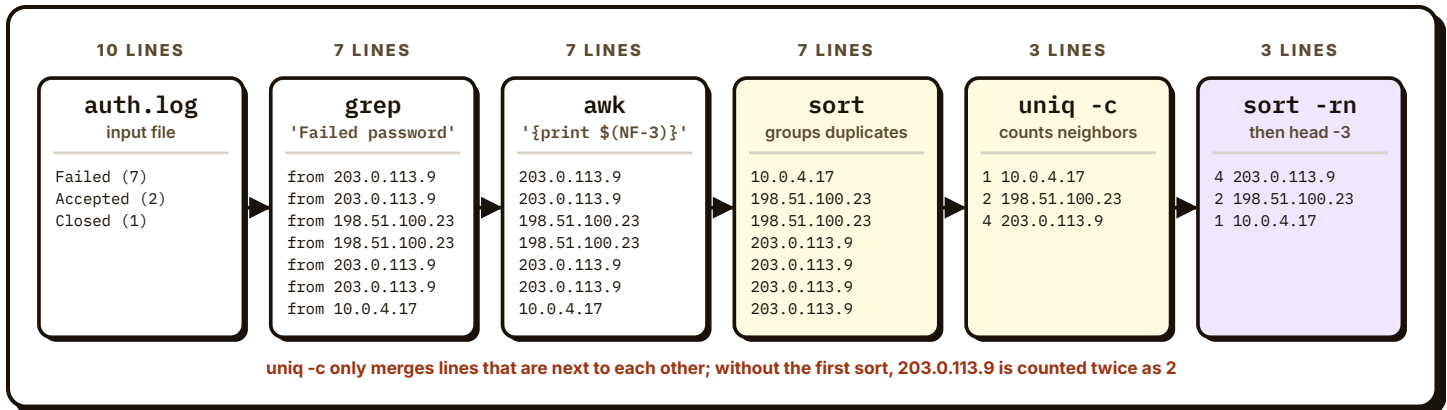


FIGURE Each stage reads the previous stage's output. The line counts on top show the data shrinking from raw log to answer.

- `$(NF-3)` counts back from the end, so it still finds the address when the line says `for invalid user bob`
- `uniq` only merges **adjacent** duplicates: always `sort` first
- By default a pipeline's exit status is the last command's
 - `set -o pipefail` makes the pipeline fail if any stage fails, so a missing log file is not hidden by a successful `sort`

ON THE EXAM

- Match the tool to the job: `grep` finds lines, `awk` pulls columns, `sed` substitutes or deletes, `jq` queries JSON, and the PowerShell cmdlets do the same work on objects.
- Expect to complete or read a pipeline. `sort | uniq -c | sort -rn` counts and ranks; leaving out the first `sort` gives wrong counts.
- `sed -i` changes the file; without it `sed` only prints. A scenario that "ran `sed` but the file did not change" is missing `-i`.

TRY IT

Add a fourth device, `fw1`, with `"status": "planned"` to `inv.json`. Write one `jq` filter and one `Where-Object` line that list every device that is not active.

1.1 Dependency Management

1.0 AUTOMATION CODING CONCEPTS

Libraries and packages

- A **library** is reusable code your script imports instead of writing itself
 - `Jinja2` renders templates, `PyYAML` parses YAML, `netmiko` drives SSH to switches, `requests` calls REST APIs
- A **package** is the distributable bundle that delivers a library
 - Code plus metadata: name, version, and the packages it needs
 - Python packages come from PyPI as wheels; Node packages from npm; Ansible collections from Galaxy
 - Operating systems have packages too: `apt` and `dnf` install `.deb` and `.rpm` files
- A package manager downloads, installs, upgrades, and removes packages: `pip`, `npm`, `apt`, `ansible-galaxy`

1.1 Dependency Management (continued)

- **Dependencies** are the packages your code needs
 - Direct: the ones you asked for
 - **Transitive**: the ones your dependencies need (Jinja2 pulls in MarkupSafe)
- **Dependency management** keeps that set declared, versioned, and reproducible
 - "Works on my laptop" usually means a different version of something installed somewhere else

■ requirements.txt

- The plain-text list of packages a Python project needs, one per line
 - Install everything with `pip install -r requirements.txt`
- **Pinning** fixes the exact version so every install is identical

LINE	INSTALLS
<code>Jinja2==3.1.6</code>	exactly 3.1.6 (pinned)
<code>Jinja2>=3.1,<4</code>	newest 3.x at or above 3.1
<code>Jinja2~3.1.2</code>	3.1.2 or newer, still 3.1.x
<code>Jinja2</code>	whatever is newest today

- Pin for automation that runs unattended (pipelines, scheduled jobs)
 - An unpinned dependency can change between Monday's run and Tuesday's
 - Update pins on purpose, test, then commit the new file
- `pip freeze` lists every installed package, transitive ones included, in pinned form
- `--require-hashes` adds a checksum per package so a tampered download fails

REQUIREMENTS.TXT

```
# runtime libraries for render.py
Jinja2==3.1.6
PyYAML==6.0.3
```

- `render.py` turns the `switch.yaml` from Data Types into switch config lines

RENDER.PY

```
import sys
import yaml
from jinja2 import Template

TEMPLATE = Template(
    "hostname {{ hostname }}\n"
    "{% for v in vlans %}vlan {{ v }}\n{% endfor %}"
)

with open(sys.argv[1]) as f:
    data = yaml.safe_load(f)
print(TEMPLATE.render(**data), end="")
```

■ Virtual environments

- A **virtual environment** (venv) is a private copy of Python with its own `site-packages` folder
 - One per project; projects stop fighting over versions
 - Installs need no root and never touch the system Python that OS tools rely on
- Debian 12+ and Ubuntu 23.04+ mark the system Python as externally managed
 - A bare `pip install` there fails with **error: externally-managed-environment**
- Keep `.venv/` out of Git (`.gitignore`); commit `requirements.txt` instead
 - Anyone can rebuild the same environment from the file

TERMINAL

```
$ python3 -m venv .venv
$ . .venv/bin/activate
$ pip install -q -r requirements.txt
$ pip freeze
Jinja2==3.1.6
MarkupSafe==3.0.4
PyYAML==6.0.3
$ python render.py switch.yaml
hostname core-sw1
vlan 10
vlan 20
vlan 30
```

- `deactivate` returns to the system Python
- MarkupSafe appeared without being asked for: a transitive dependency
 - It is not pinned, so a later install may bring a newer MarkupSafe
 - To pin it too, write `pip freeze > requirements.txt`, or use a lock tool (`pip-tools`, `uv`, Poetry)

1.1 Dependency Management (continued)

Dependencies in a Dockerfile

- A **Dockerfile** builds an image that carries the code, the runtime, and every dependency
 - The same image runs on a laptop, in CI, and in production
- Order instructions from least to most frequently changed
 - Each instruction becomes a cached **layer**; a change rebuilds that layer and everything after it
 - To keep the **build cache** useful, copy **requirements.txt** and install it before copying the code, so a code edit does not reinstall packages

```
FROM python:3.13-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY render.py switch.yaml ./
ENTRYPOINT ["python", "render.py"]
CMD ["switch.yaml"]
```

DOCKERFILE

REBUILD AFTER EDITING render.py

FROM python:3.13-slim

CACHED

WORKDIR /app

CACHED

COPY requirements.txt .

CACHED

RUN pip install ...

CACHED

COPY render.py switch.yaml ./

RUNS

ENTRYPOINT, CMD

settings,
no layer

FIGURE Editing render.py invalidates only the layers from its COPY down. The pip install layer comes from cache.

TERMINAL

```
$ docker build -q -t cfg-render:1.0 .
sha256:07bb04430c43fe146b60bbd9944bda41...
$ docker run --rm cfg-render:1.0
hostname core-sw1
vlan 10
vlan 20
vlan 30
```

- After editing only **render.py**, a rebuild with **--progress=plain** prints **CACHED** for **WORKDIR**, **COPY requirements.txt**, and **RUN pip install**; only the last **COPY** runs again
 - Changing a pin in **requirements.txt** rebuilds the install layer, as it should
- Pin the base image tag (**python:3.13-slim**), never **latest**
- no-cache-dir** keeps pip's download cache out of the image
- OS packages: one **RUN** that chains **apt-get update** and **apt-get install -y --no-install-recommends**, so the package list is never stale

Other ecosystems

- Same idea everywhere: a **manifest** names what you need; a **lock file** records exactly what was installed

ECOSYSTEM	YOU DECLARE	EXACT VERSIONS RECORDED IN	INSTALL
Python	requirements.txt or pyproject.toml	pinned requirements.txt , uv.lock , poetry.lock	pip install -r requirements.txt
Node.js	package.json	package-lock.json	npm ci
Ansible	collections/requirements.yml	versions in the same file	ansible-galaxy install -r
Terraform	required_providers block	.terraform.lock.hcl	terraform init
Go	go.mod	go.sum	go mod download

1.1 Dependency Management (continued)

- Commit lock files to Git so CI installs the same versions you tested
- Scan dependencies for known vulnerabilities: `pip-audit`, `npm audit`, Dependabot

ON THE EXAM

- A scenario where a script works on one machine and fails on another after "nothing changed" points to an unpinned dependency or a missing virtual environment. The fix is a pinned `requirements.txt` (or lock file) installed in a venv or container.
- Know the vocabulary: a library is code you import; a package is how it is distributed; a transitive dependency is one your dependency needs.
- In a Dockerfile, copying `requirements.txt` and running `pip install` before copying the code keeps the install layer cached.

TRY IT

Change the pin to `Jinja2==3.1.5`, rebuild with `--progress=plain`, and note which layers rebuild. Then change only `render.py` and compare.

1.2 Git Fundamentals

1.0 AUTOMATION CODING CONCEPTS

■ Why automation code lives in Git

- Every change is recorded
 - Who made it, when, and the message explaining why
 - Roll back a bad change in seconds instead of rebuilding by hand
- Automation code is production code
 - Scripts, Ansible playbooks, Terraform files, pipeline YAML
 - A typo in a playbook can take down every switch it touches
- Git is a **distributed version control system**
 - Every clone holds the complete history
 - Work offline, sync with the team later
 - No single copy is the only copy
- The repository is also the trigger
 - Pushing a commit is what starts most CI pipelines (Domain 3)

■ Three local areas, one remote

- **Working tree**
 - The files you see and edit in the project folder
- **Staging area** (also called the **index**)
 - The changes you have selected for the next commit
 - Lets you commit one fix while other edits stay uncommitted
- **Local repository**
 - The `.git` folder: every committed snapshot
- **Remote repository**
 - A shared copy on a server such as GitHub, GitLab, or a self-hosted Gitea
 - Your commits stay private to your machine until you push them

1.2 Git Fundamentals (continued)

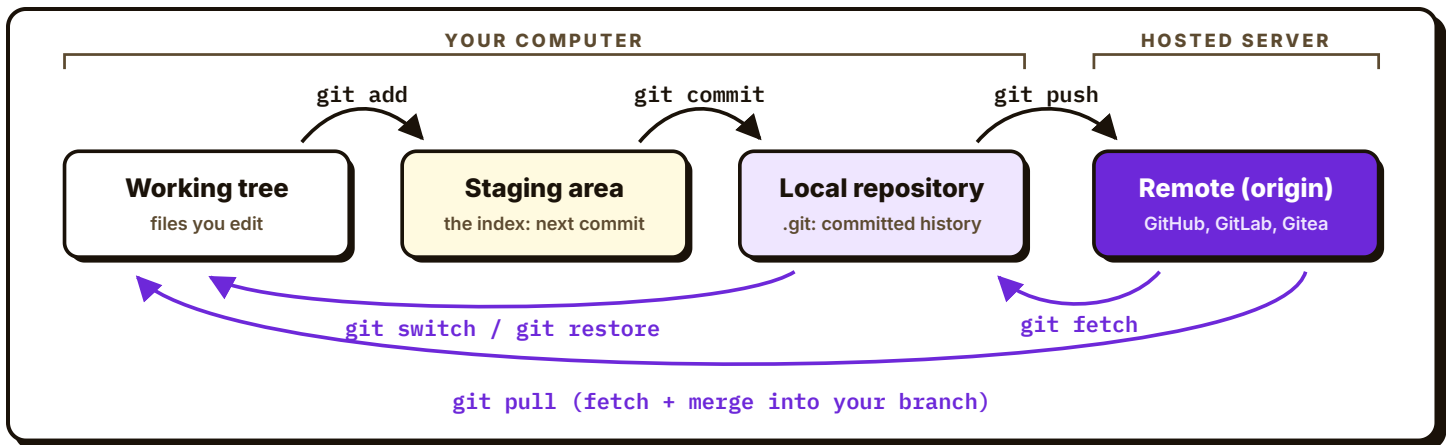


FIGURE Each command moves changes between two areas. Arrows on top send work toward the team; arrows below bring work back.

git config

- Settings live at three levels
 - system**: every user on the machine (`/etc/gitconfig`)
 - global**: your user account (`~/.gitconfig`)
 - local**: one repository (`.git/config`), the default when no flag is given
 - The most specific level wins: **local overrides global, global overrides system**
- Set your identity before your first commit
 - Name and email are written into every commit you make

TERMINAL

```
$ git config --global user.name "Ada Admin"
$ git config --global user.email "ada@example.com"
$ git config --global init.defaultBranch main
$ git config --list --show-origin
file:/home/ada/.gitconfig user.name=Ada Admin
```

- show-origin** prints the file each setting came from
 - The fastest way to find out why a setting is not what you expect
- Settings you will meet later in these notes
 - core.hooksPath**: a shared folder of hooks
 - pull.rebase**: rebase instead of merge on pull
 - versionsort.suffix**: correct ordering of pre-release tags

git add

- Copies changes from the working tree into the staging area
 - git add switch.cfg**: one file
 - git add .**: everything under the current folder
 - git add -p**: walk through each change and choose
- git status** shows what is where

TERMINAL

```
$ git status --short
?? banner.txt
?? switch.cfg
$ git add switch.cfg
$ git status --short
A switch.cfg
?? banner.txt
```

- ??** is untracked; **A** is staged as a new file; **M** is modified
- .gitignore** lists files Git should never stage
 - `.env`, `*.pem`, `.terraform/`, `terraform.tfstate`, build output
 - Keeps secrets and generated files out of history

git commit

- Saves the staged snapshot to the local repository with a message
- Every commit gets an ID: a **SHA hash** of its content and metadata
 - 40 hex characters, usually shown as the first 7

TERMINAL

```
$ git commit -m "feat: add core switch baseline"
[main (root-commit) ffe321e] feat: add core switch baseline
1 file changed, 1 insertion(+)
$ git log --oneline
1e5e6c6 feat: add login banner
ffe321e feat: add core switch baseline
```

- git commit -am "msg"** stages tracked files and commits in one step
 - Skips untracked files; new files still need **git add**
- See changes before you commit them
 - git diff**: working tree versus staging
 - git diff --staged**: staging versus the last commit

1.2 Git Fundamentals (continued)

ON THE EXAM

- Know which two areas each command moves changes between: `add` is working tree to staging, `commit` is staging to the local repository, `push` is local to remote.
- A commit that has not been pushed does not exist for teammates or for the pipeline. If a scenario says "the pipeline did not run," check whether the change was pushed.

TRY IT

Make a folder, run `git init`, create two files, and stage only one. Run `git status --short` before and after `git add` and match each two-letter code to the figure above.

1.2 Working with Remotes

1.0 AUTOMATION CODING CONCEPTS

■ Remotes

- A **remote** is a short name for another copy of the repository
 - `origin` is the name Git gives the remote you cloned from
 - A repository can have several: `origin`, `upstream`, `backup`
- `git remote -v` lists remotes and their URLs
- `git remote add origin <url>` connects an existing local repository
- Remote URLs use HTTPS or SSH
 - HTTPS: personal access token for authentication
 - SSH: key pair; `git@github.com:org/repo.git`
- **Remote-tracking branches**
 - `origin/main` is your local, read-only record of where `main` was on the remote the last time you talked to it
 - Updated by `fetch`, `pull`, and `push`; never edited directly

■ git push

- Sends your local commits to a branch on the remote
- First push of a new branch: set the upstream
 - `git push -u origin main`
 - After that, plain `git push` and `git pull` know where to go
- Tags are not pushed by default
 - `git push origin --tags`, or push one tag: `git push origin v1.2.0`
- A push is **rejected** when the remote has commits you do not

TERMINAL

```
$ git push
! [rejected]          main -> main (fetch first)
error: failed to push some refs to
'github.com:acme/netconfig.git'
hint: Updates were rejected because the remote
hint: contains work that you do not have locally.
```

- The fix is to bring those commits down first, then push again
 - Never answer a rejection with `git push --force` on a shared branch; it discards your teammates' work

■ git fetch

- Downloads new commits and updates the `origin/*` branches
- Does **not** change your working tree or your local branches
 - Always safe; nothing you are working on moves
- Use it to look before you integrate

TERMINAL

```
$ git fetch origin
1e5e6c6..271e551 main -> origin/main
$ git status -sb
## main...origin/main [ahead 1, behind 1]
$ git log --oneline main..origin/main
271e551 feat: add VLAN 20
```

- **ahead 1**: you have a commit the remote does not
- **behind 1**: the remote has a commit you do not
- `main..origin/main` means "commits on origin/main that are not on main"

1.2 Working with Remotes (continued)

git pull

- **git pull = git fetch + integrate**
 - Integrate means merge by default, so you may get a merge commit
 - `git pull --rebase`, or set `pull.rebase true`, replays your commits on top for a straight-line history

- Pull changes your working tree
 - If you and a teammate edited the same lines, the pull stops with a **merge conflict** (troubleshooting is objective 1.4)
- Pull before you start work and before you push

COMMAND	DIRECTION	CHANGES YOUR FILES?	CAN CONFLICT?
<code>git fetch</code>	remote to local tracking branches	No	No
<code>git pull</code>	remote to your current branch	Yes	Yes
<code>git push</code>	your branch to remote	No	Rejected if behind
<code>git clone</code>	remote to a new local repository	Creates them	No

ON THE EXAM

- "Update the local view of the remote without modifying any working files" is `git fetch`. "Get the latest changes into my branch" is `git pull`.
- A rejected push is fixed by pulling (or fetching and merging) and pushing again, not by forcing. Force pushing a shared branch rewrites history other people already have.

TRY IT WITH TWO CLONES

Create a bare repository with `git init --bare origin.git`, clone it twice, push a commit from one clone, then run `git fetch` and `git status -sb` in the other. Commit something there too before pulling and you will see the rejected push for yourself.

1.2 Semantic Versioning

1.0 AUTOMATION CODING CONCEPTS

Version numbers that carry meaning

- **Semantic versioning** (SemVer): **MAJOR.MINOR.PATCH**
 - Defined at semver.org; the current specification is 2.0.0
 - The number tells consumers whether an upgrade is safe before they read a single release note
- Applies to anything other people depend on
 - Ansible roles and collections, Terraform modules, internal Python packages, container images, APIs

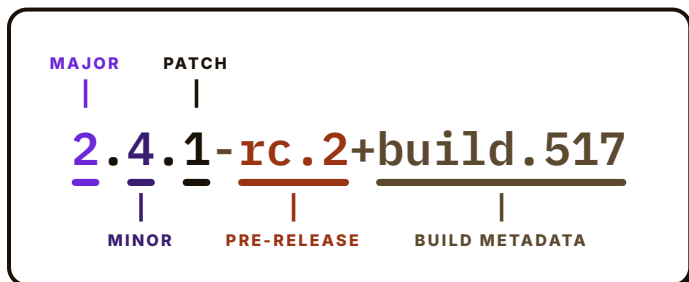


FIGURE Every part has a job. Only MAJOR, MINOR, PATCH, and the pre-release label affect which version is "newer."

- **MAJOR**: an incompatible, breaking change
 - A playbook variable is renamed, an API field is removed, a default changes behavior
- **MINOR**: new functionality that is backward compatible
 - A new optional variable with a sensible default
- **PATCH**: a backward-compatible bug fix
- Bumping one number resets everything to its right
 - **1.4.2** to **1.4.3** (fix), to **1.5.0** (feature), to **2.0.0** (break)
- **0.y.z** is initial development
 - Anything may change at any time; the public interface is not stable yet
- Once a version is released, its contents never change
 - Fixing it means releasing a new version

Major vs. minor releases

- A **major release** asks consumers to do work
 - Read the upgrade notes, change their code or inventory, test
 - Schedule it; announce it; support the previous major for a while

1.2 Semantic Versioning (continued)

- A **minor release** is safe to take
 - New capabilities appear; nothing existing breaks
- Patch releases should be taken quickly
 - Especially security fixes

CHANGE TO A TERRAFORM MODULE	BUMP
Rename a required input variable	Major
Change a default that alters existing resources	Major
Add an optional input with a default	Minor
Add a new output	Minor
Fix a wrong tag value on a resource	Patch
Correct a typo in the README only	None, or patch

■ Pre-release versions

- A hyphen and identifiers after the patch number
 - `2.0.0-alpha`, `2.0.0-beta.2`, `2.0.0-rc.1` (release candidate)
- A pre-release ranks **below** its final release
 - `1.9.0` < `2.0.0-alpha` < `2.0.0-beta.2` < `2.0.0-rc.1` < `2.0.0`
- Signals "not yet stable": deploy to test and staging, not production
- **Build metadata** follows a plus sign: `2.0.0+build.517`
 - Records the build or commit; ignored when comparing versions

■ Filtering techniques

- **Version constraints** filter which versions a tool may install
 - Written in `package.json`, `requirements.txt`, Ansible `requirements.yml`, Terraform `required_providers`

CONSTRAINT	ACCEPTS	REJECTS
<code>1.2.3</code>	exactly 1.2.3	everything else
<code>^1.2.3</code>	<code>>=1.2.3 <2.0.0</code>	2.0.0 and up
<code>~1.2.3</code>	<code>>=1.2.3 <1.3.0</code>	1.3.0 and up
<code>1.x</code>	any 1.y.z	2.0.0 and up
<code>>=1.4.0 <2.0.0</code>	that range	outside it
Terraform <code>~> 1.2</code>	<code>>=1.2 <2.0</code>	2.0 and up
Terraform <code>~> 1.2.0</code>	<code>>=1.2.0 <1.3.0</code>	1.3.0 and up

- Caret (`^`) means "take new minors and patches"; tilde (`~`) means "patches only"

- Pre-releases are excluded unless you ask for them
 - `^1.2.3` does not match `1.3.0-beta.1`
- `^0.2.3` stays inside `0.2.x`, because in 0.y.z every minor may break

NPM SEMVER CLI

```
$ npx semver -r '^1.2.3' 1.2.9 1.9.0 2.0.0 1.3.0-beta.1
1.2.9
1.9.0
$ npx semver -i premajor --preid rc 1.4.2
2.0.0-rc.0
```

■ Filtering versions in Git

- Releases are marked with **tags**
 - `git tag -a v1.0.0 -m "First stable release"` creates an annotated tag (author, date, message)
 - `git tag v1.0.0` creates a lightweight tag (just a pointer)
- `git tag -l 'v1.*'`: list tags matching a pattern
- `git describe --tags --abbrev=0`: the most recent tag reachable from HEAD
- Sort by version, not alphabetically
 - Alphabetical order puts `v1.10.0` before `v1.2.0`

TERMINAL

```
$ git tag -l --sort=-v:refname
v2.0.0-rc.1
v2.0.0
v1.10.0
$ git config versionsort.suffix -
$ git tag -l --sort=-v:refname
v2.0.0
v2.0.0-rc.1
v1.10.0
```

- Without `versionsort.suffix`, Git ranks `v2.0.0-rc.1` above `v2.0.0`
 - `sort -V` has the same blind spot
- Stable releases only: `git tag -l | grep -E '^v[0-9]+\.[0-9]+\.[0-9]+$'`

ON THE EXAM

- Given a description of a change, pick the bump: breaking is major, new compatible feature is minor, fix is patch. Watch for "removed" or "renamed" in the scenario; that is a major release.
- `2.0.0-rc.1` is older than `2.0.0`. A distractor will treat the extra characters as "newer."
- Caret allows minors; tilde allows only patches.

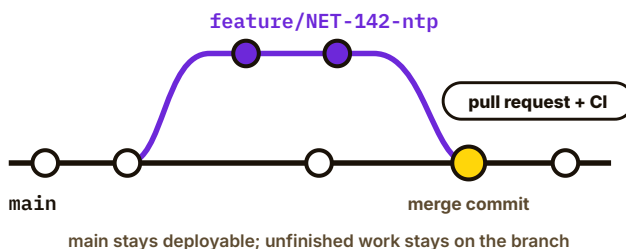
1.2 Branching Strategies

1.0 AUTOMATION CODING CONCEPTS

Branches

- A **branch** is a movable pointer to a commit
 - Creating one is instant and costs almost nothing
 - Each branch is a separate line of work
- main** is the default branch
 - Treat it as always deployable
 - Most teams protect it: no direct pushes, changes arrive by review
- The everyday commands
 - `git switch -c feature/x`: create a branch and move to it
 - `git switch main`: go back
 - `git merge feature/x`: bring that work into the current branch
 - `git branch -d feature/x`: delete it once merged
- A **branching strategy** is the team's agreement on which branches exist, what goes on each, and how work moves between them

FEATURE BRANCHING



RELEASE BRANCHING

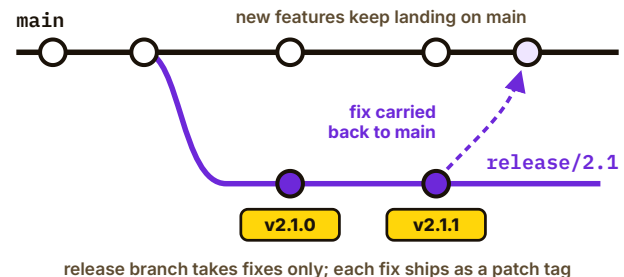


FIGURE Feature branches carry new work into main. Release branches carry a finished version out of main and accept only fixes.

Feature branching

- One branch per feature, bug, or ticket
 - Created from **main**, merged back into **main**
- Work arrives through a **pull request** (GitLab calls it a **merge request**)
 - Teammates review the change
 - The CI pipeline runs lint and tests on the branch before merge
 - Branch protection can require both to pass
- Keeps unfinished work out of **main**
 - Abandon an experiment by deleting the branch
- Keep branches short-lived
 - The longer a branch lives, the further it drifts and the worse the merge conflicts
- Trunk-based development** is feature branching taken to its limit
 - Branches last hours or a day; everyone merges to **main** constantly
 - Unfinished features hide behind **feature flags**

TERMINAL

```
$ git switch -c feature/NET-142-ntp-servers
$ git commit -am "feat(ntp): add NTP servers"
$ git push -u origin feature/NET-142-ntp-servers
```

Release branching

- Cut a **release branch** from **main** when a version is ready to stabilize
 - release/2.1** or **release/v2.1**
- Only fixes go on the release branch
 - New features keep landing on **main** for the next version
- Tag each release from the branch
 - v2.1.0**, then **v2.1.1** for the first fix
- Every fix is carried back to **main**
 - Merge the release branch, or `git cherry-pick -x <commit>`
 - Otherwise the bug comes back in 2.2
- Lets you support several versions in production at once
 - Customers on 2.1 get patches while 3.0 is being built

1.2 Branching Strategies (continued)

- **GitFlow** is a heavier model built on the same idea
 - Long-lived `main` and `develop` branches plus feature, release, and hotfix branches

STRATEGY	BRANCHES LIVE FOR	BEST FIT	WATCH OUT FOR
Feature branching	days	most teams; review before merge	branches that drift for weeks
Trunk-based	hours	mature CI, frequent deploys	needs feature flags and fast tests
Release branching	a version's support life	several versions in production	fixes not carried back to main
GitFlow	develop and main forever	scheduled releases	ceremony and merge overhead

■ Branch naming conventions

- Prefix by purpose
 - `feature/`, `fix/` or `bugfix/`, `hotfix/`, `release/`, `chore/`, `docs/`
- Include the ticket ID so the work links back to the tracker
 - `feature/NET-142-ntp-servers`
- Lowercase words joined by hyphens; no spaces
- Consistent names let automation act on patterns
 - Run the deploy job only on `release/*`
 - Apply branch protection rules to `main` and `release/*`
 - `git branch --list 'release/*'` finds every release line

ON THE EXAM

- "Maintain version 1 for existing customers while building version 2" points to release branching.
- "Isolate a new feature until it is reviewed and tested" points to feature branching with a pull request.
- A naming convention exists so tools and people can filter branches by pattern.

1.2 The Commit Life Cycle

1.0 AUTOMATION CODING CONCEPTS

From edit to remote

- Between editing a file and the change reaching the team, Git gives you places to run checks automatically
 - Checks that run early are cheap
 - A lint error caught before commit costs seconds
 - The same error caught by the pipeline costs a pipeline run and a context switch



FIGURE Client-side hooks give fast feedback. Anything that must never be skipped is enforced again on the server or in CI.

Hooks

- A **hook** is a script Git runs automatically at a point in the life cycle
 - Stored in `.git/hooks/`, named for the event, and executable
 - A **non-zero exit code** stops the action
- Client-side hooks

HOOK	RUNS	TYPICAL JOB
pre-commit	before the commit is created	format, lint, scan for secrets
prepare-commit-msg	before the editor opens	insert a ticket ID or template
commit-msg	after the message is written	enforce the message format
post-commit	after the commit	notifications
pre-push	before the push sends anything	run quick tests

- Server-side hooks: `pre-receive`, `update`, `post-receive`
 - Run on the Git server; the client cannot skip them
- Client hooks are **not enforced**
 - `git commit --no-verify` skips `pre-commit` and `commit-msg`
 - Repeat any check that matters in the CI pipeline
- `.git/hooks/` is not part of the repository
 - Share hooks with a tracked folder: `git config core.hooksPath .githubhooks`
 - Or use the **pre-commit framework**: one tracked `.pre-commit-config.yaml`, then `pre-commit install`

`.GIT/HOOKS/COMMIT-MSG`

```
#!/bin/sh
# Enforce Conventional Commits: type(optional
scope): summary,
# with the whole first line at 72 characters or
fewer
subject=$(head -1 "$1")
pattern='^(feat|fix|docs|chore|refactor|test|ci)(\
([a-z0-9-]+))?!?: .+$'
if ! printf '%s\n' "$subject" | grep -Eq "$pattern"
|| [ ${#subject} -gt 72 ]; then
    echo "commit-msg: use 'type(scope): summary'" >&2
    exit 1
fi
```

Formatting

- A **formatter** rewrites code into one consistent style
 - Python: `black`, `ruff format`; YAML, JSON, Markdown: `prettier`; Terraform: `terraform fmt`; shell: `shfmt`
- Ends style arguments in code review
- Keeps diffs about real changes, not whitespace
- Two modes
 - Fix mode rewrites files (on your machine, in `pre-commit`)
 - Check mode** only reports and fails (in CI): `black --check`, `terraform fmt -check`

Linting

- A **linter** reads code without running it and flags likely bugs and risky patterns
 - Python: `ruff`, `flake8`, `pylint`; shell: `shellcheck`; YAML: `yamllint`; Ansible: `ansible-lint`; Terraform: `tflint`

1.2 The Commit Life Cycle (continued)

- Catches unused imports, undefined variables, unquoted shell variables, invalid YAML
- Formatting changes how code looks; linting finds problems in what it does
- A formatter hook reports **Failed** when it rewrites a file
 - Nothing is broken: review the change, `git add` it, and commit again

PRE-COMMIT RUN --ALL-FILES

```
$ pre-commit run --all-files
trim trailing whitespace.....Passed
check yaml.....Passed
detect private key.....Failed
ruff.....Failed
ruff-format.....Failed
```

.PRE-COMMIT-CONFIG.YAML

```
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v5.0.0
  hooks:
  - id: trailing-whitespace
  - id: check-yaml
  - id: detect-private-key
- repo: https://github.com/astral-sh/ruff-pre-commit
  rev: v0.6.9
  hooks:
  - id: ruff          # lint
  - id: ruff-format   # format
```

■ Detecting sensitive data

- Once a secret is committed, it is in history
 - A later commit that deletes the line does not remove it from earlier commits
 - Treat a pushed secret as compromised: **rotate the credential** first, then clean history
- Prevent it at the earliest point
 - `.gitignore` for `.env`, `*.pem`, `terraform.tfstate`
 - A `pre-commit` scanner: `gitleaks`, `git-secrets`, `detect-secrets`, `trufflehog`
 - Scanners match known key formats (an AWS access key ID starts with `AKIA`) and high-entropy strings

- Enforce it on the server too
 - Platform secret scanning and push protection reject the push

.GIT/HOOKS/PRE-COMMIT

```
#!/bin/sh
# Block commits that stage something shaped like an
# AWS access key ID
if git diff --cached -U0 | grep -E '^\\+.AKIA[0-9A-Z]{16}' >/dev/null; then
  echo "pre-commit: possible AWS access key in
  staged changes" >&2
  exit 1
fi
```

■ Comments

- The **commit message** is the comment on a change
 - Future you reads it at 2 a.m. during an outage
- First line: a summary in the imperative mood, about 50 characters, 72 at most
 - "fix(dns): correct TTL on internal zone", not "fixed stuff"
- Blank line, then a body explaining why
- **Conventional Commits**: `type(scope): summary`
 - Types: `feat`, `fix`, `docs`, `chore`, `refactor`, `test`, `ci`
 - **!** after the type, or a **BREAKING CHANGE:** footer, marks a breaking change
 - Tools read the history and pick the next version: `fix` is a patch, `feat` is a minor, breaking is a major
- Code comments explain why, not what
 - Never leave a credential in a commented-out line

ON THE EXAM

- Hooks are scripts in `.git/hooks/`, and a non-zero exit blocks the action. `pre-commit` is where formatting, linting, and secret scanning run; `commit-msg` checks the message.
- Client-side hooks can be bypassed with `--no-verify`, so mandatory checks belong in CI or a server-side hook.
- If a secret reached the remote, the first step is revoking or rotating it, not rewriting history.

What infrastructure as code means

- **Infrastructure as code (IaC)**: servers, networks, and services defined in text files that a tool reads and carries out
 - Replaces clicking through a cloud console or typing commands into each switch
 - The files live in Git (objective 1.2): reviewed in pull requests, tagged, rolled back
 - One set of files builds dev, test, and production, so the environments match
- Two jobs, often split between two tools
 - **Provisioning**: create the infrastructure itself (VMs, VPCs, subnets, DNS records, load balancers)
 - **Configuration management**: set up what runs inside it (packages, config files, services, users)
- A common pairing: Terraform builds the VMs, Ansible configures them

TOOL	MAIN JOB	HOW YOU WRITE IT
Terraform, OpenTofu	provisioning, any provider	HCL; declarative; tracks a state file
AWS CloudFormation, Azure Bicep	provisioning, one cloud	YAML or JSON, Bicep; declarative
Pulumi	provisioning	Python, TypeScript, Go, C#; desired-state engine
Ansible	configuration (and some provisioning)	YAML playbooks; agentless over SSH or WinRM
Puppet, Chef, Salt	configuration	agent on each node; Puppet DSL, Ruby, YAML
Packer	machine images	HCL; bakes a VM or cloud image

Configuration drift

- **Configuration drift**: the running system no longer matches its definition
 - A fix typed over SSH during an outage and never written down
 - A setting changed in the console "just to test"
 - Updates applied to some servers and not others
- Drift produces **snowflake servers**
 - Each one is unique; nobody can rebuild it; everyone is afraid to reboot it
- Drift is the main reason to adopt IaC
 - When the code is the only way changes happen, the code is always the truth
- Detect drift by running the tool in compare mode
 - `terraform plan` prints "Objects have changed outside of Terraform"
 - `terraform plan -refresh-only` reports drift without proposing changes
 - `ansible-playbook --check --diff` shows what a run would change (example in the next topic)
- Schedule it: a nightly pipeline job runs `terraform plan -detailed-exitcode`
 - Exit 0 is no changes, 1 is an error, 2 means a difference exists
- Fix it by applying the code again, or, if the manual change was right, by adding it to the code

Reusability

- Write a definition once, use it many times with different inputs
 - Forty branch offices, one switch template; a fix reaches every site
- Terraform **module**: a folder of `.tf` files with input variables and outputs
 - The calling (root) module uses it in a `module` block and passes values
 - Source can be a local path, a Git URL pinned to a tag (`?ref=v1.4.0`), or a registry address with a **version** constraint

```

MODULES/SWITCH_CONFIG/MAIN.TF

resource "local_file" "config" {
  filename = "out/${var.hostname}.cfg"
  content = templatefile(
    "${path.module}/switch.tftpl",
    { hostname = var.hostname, vlans = var.vlans }
  )
}

```

1.3 Infrastructure as Code Principles (continued)

MAIN.TF (TWO CALLS, ONE MODULE)

```
module "sw1" {  
  source = "./modules/switch_config"  
  hostname = "access-sw1"  
  vlans = { "10" = "users", "20" = "voice" }  
}  
  
module "sw2" {  
  source = "./modules/switch_config"  
  hostname = "access-sw2"  
  vlans = { "10" = "users", "30" = "printers" }  
}
```

TERMINAL

```
$ terraform apply -auto-approve  
Apply complete! Resources: 2 added, 0 changed, 0  
destroyed.  
$ ls out  
access-sw1.cfg  
access-sw2.cfg  
$ cat out/access-sw1.cfg  
hostname access-sw1  
vlan 10  
  name users  
vlan 20  
  name voice
```

- Ansible **role**: a standard folder layout that bundles tasks with their defaults, templates, and handlers
 - `ansible-galaxy role init roles/resolver` creates the skeleton
 - A play applies the role and overrides only what differs, such as `resolver_servers` for the Dallas site
 - Share roles through Ansible Galaxy or a private Git repository

ROLE FOLDER	HOLDS
tasks/	the steps (<code>main.yml</code> runs first)
defaults/	inputs callers may override
vars/	values callers should not change
templates/	Jinja2 files for the <code>template</code> module
files/	static files to copy
handlers/	actions on change, such as a service restart
meta/	dependencies on other roles

- Pin the version of every shared module or role; an unpinned upgrade changes every site at once

■ Immutability

- **Mutable infrastructure**: a server is built once, then patched, upgraded, and edited in place for years
- **Immutable infrastructure**: a running server is never modified
 - To change it, build a new one from a new version and replace the old one
 - Nothing is edited after deploy, so there is nothing to drift

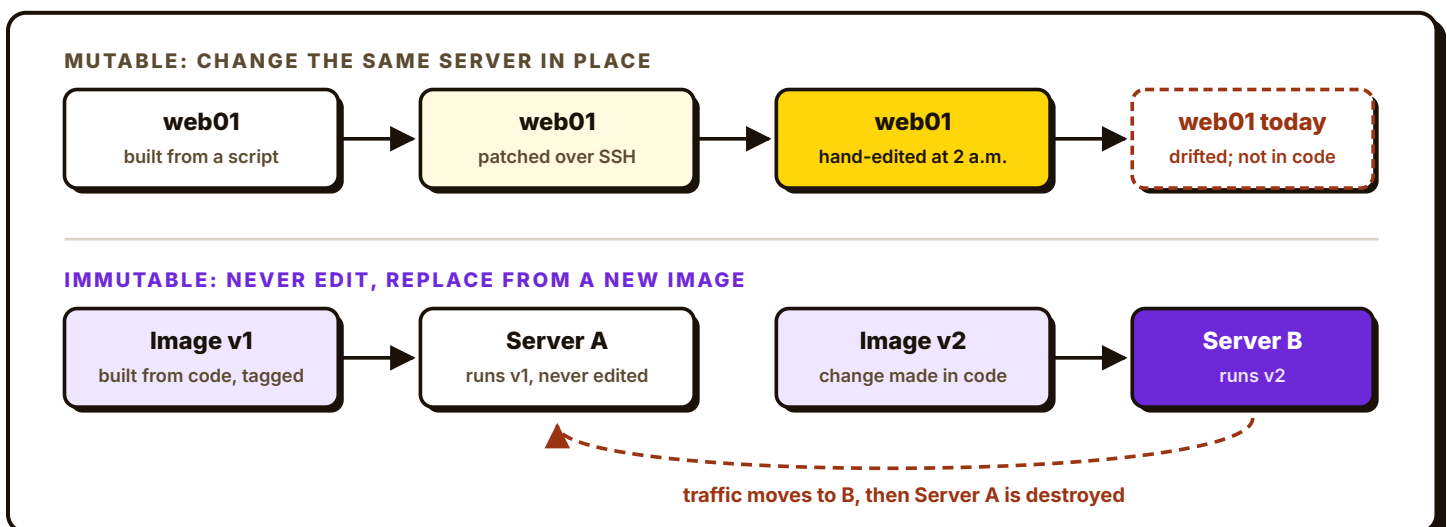


FIGURE A mutable server collects changes nobody recorded. An immutable server is thrown away and replaced by one built from the next versioned image.

1.3 Infrastructure as Code Principles (continued)

- How teams make servers immutable
 - Golden image**: Packer bakes the OS, agents, and app into a VM template or cloud image, versioned like code
 - Containers: each release is a new image tag; a running container is replaced, not patched
 - Terraform replaces a resource when an argument cannot change in place:

TERMINAL

```
$ terraform plan
# module.sw2.local_file.config must be replaced
-/+ resource "local_file" "config" {
  ~ content           = <<-EOT # forces
  replacement
Plan: 1 to add, 0 to change, 1 to destroy.
```

- `-/+` means destroy, then create the replacement
 - With `lifecycle { create_before_destroy = true }` the symbol is `+/-`: the new one comes up first
 - `terraform apply -replace=<address>` forces a replacement on purpose
- Pros: identical servers; rollback is redeploying the previous image
- Costs: build and deploy must be automated; data and logs must live off the server

- Shorthand: **pets versus cattle**. Pets are nursed back to health; cattle are replaced.

ON THE EXAM

- "A server was changed by hand and no longer matches the template" is configuration drift. The fix is to reapply the code, or to codify the change, not to document it in a wiki.
- "Rebuild from a new image instead of patching the running server" is immutability. "Use the same definition for 40 sites with different values" is reusability: modules in Terraform, roles in Ansible.
- Copying a whole playbook or `.tf` folder per site is the distractor for reusability: every copy must then be fixed separately.

TRY IT

Build the `switch_config` module above, then add a third `module` block for `access-sw3`. Run `terraform plan` and confirm it plans exactly 1 to add: the two existing switches are untouched.

1.3 Declarative, Imperative, and Idempotent

1.0 AUTOMATION CODING CONCEPTS

Imperative: say how

- Imperative** (also called **procedural**) code lists the steps to carry out, in order
 - Bash, PowerShell, Python, a runbook of CLI commands
 - Each step runs every time, whether or not its work is already done
- Full control, but you own every problem: checking current state, ordering, error handling

ADD-DNS.SH

```
#!/usr/bin/env bash
# Imperative: steps, run in order, every time
set -euo pipefail
mkdir -p etc
echo "nameserver 10.0.0.53" >> etc/resolv.conf
```

- The tool compares the **desired state** (your code) with the **actual state** (what is running) and changes only the difference
 - Terraform orders the work itself, from the references between resources
- Ansible is a hybrid: each module is declarative (`state: present`, `content:`), but tasks still run top to bottom
- Declarative vs. imperative comes down to what versus how

	IMPERATIVE	DECLARATIVE
You write	the steps	the end state
Order	as written	chosen by the tool
Run twice	repeats every step	nothing to do
Hand edit	not noticed	detected, reverted

Declarative: say what

- Declarative** code describes the end state; the tool works out the steps
 - Terraform HCL, CloudFormation, Kubernetes manifests, Ansible modules

1.3 Declarative, Imperative, and Idempotent (continued)

SITE.YML

```
- name: Baseline resolver config
hosts: localhost
connection: local
gather_facts: false
vars:
  etc: "{{ playbook_dir }}/etc"
tasks:
  - name: Config directory exists
    ansible.builtin.file:
      path: "{{ etc }}"
      state: directory
      mode: "0755"

  - name: Resolver points at internal DNS
    ansible.builtin.copy:
      dest: "{{ etc }}/resolv.conf"
      content: "nameserver 10.0.0.53\n"
      mode: "0644"
```

TERMINAL

```
$ ./add-dns.sh; ./add-dns.sh
$ cat etc/resolv.conf
nameserver 10.0.0.53
nameserver 10.0.0.53
```

- The playbook is: run 1 ends **ok=2 changed=2**, run 2 only checks
 - **changed**: the task had to act; **ok**: already correct

ANSIBLE-PLAYBOOK SITE.YML (SECOND RUN)

```
$ ansible-playbook site.yml
TASK [Config directory exists] *****
ok: [localhost]

TASK [Resolver points at internal DNS] *****
ok: [localhost]

PLAY RECAP *****
localhost : ok=2    changed=0
```

- Terraform's second **apply** of the module from the previous topic

TERMINAL

```
$ terraform apply -auto-approve
No changes. Your infrastructure matches the
configuration.
Apply complete! Resources: 0 added, 0 changed, 0
destroyed.
```

■ Catching drift with check mode

- Someone edits **resolv.conf** by hand to point at a public resolver
- **--check** is Ansible's **check mode**, a **dry run**: it predicts changes without making them; **--diff** shows the lines

TERMINAL

```
$ ansible-playbook site.yml --check --diff
TASK [Resolver points at internal DNS] *****
--- before: etc/resolv.conf
+++ after: etc/resolv.conf
@@ -1 +1 @@
-nameserver 8.8.8.8
+nameserver 10.0.0.53

changed: [localhost]
```

- Check mode leaves the file alone; a normal run puts it back. **changed=0** on a scheduled check means no drift

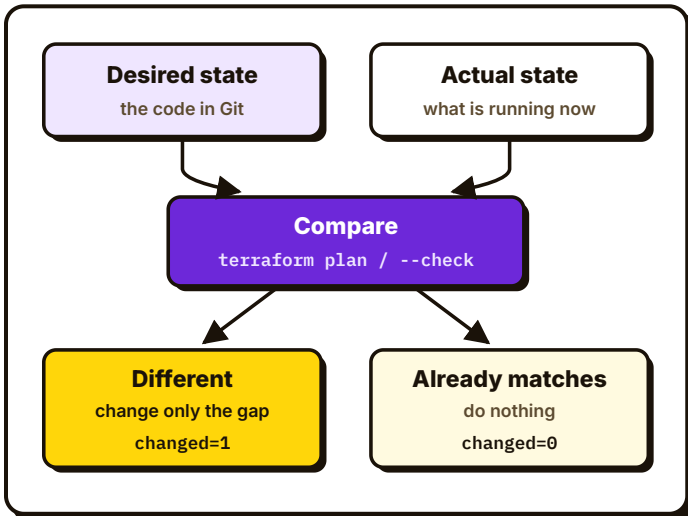


FIGURE Declarative tools compare before they act. When nothing differs, the run changes nothing.

■ Idempotency

- **Idempotent**: running an operation once or ten times leaves the system in the same state
 - The second run finds nothing to do and reports no changes
- Why automation depends on it
 - Pipelines rerun jobs; a half-finished run can be retried safely
 - A scheduled run doubles as drift correction
- The imperative script is **not idempotent**

1.3 Declarative, Imperative, and Idempotent (continued)

Writing idempotent code yourself

- Imperative scripts can be idempotent; you add the **guard checks**
 - Look before you act; prefer `mkdir -p`, `ln -sf`, and writing a whole file over appending with `>>`

ADD-DNS-SAFE.SH

```
#!/usr/bin/env bash
# Check the current state before acting
set -euo pipefail
mkdir -p etc
line="nameserver 10.0.0.53"
grep -qxF "$line" etc/resolv.conf 2>/dev/null ||
    echo "$line" >> etc/resolv.conf
```

- In Ansible, prefer a module over `command` or `shell`
 - Ansible cannot know what a command did, so it reports **changed** on every run
 - Guard commands with `creates:` or `removes:` (skip if a file exists or is absent), or set `changed_when:`

CMD.YML (TASKS)

```
- name: Append with shell
  ansible.builtin.shell: echo "vlan 10" >> vlans.txt

- name: Generate an API token once
  ansible.builtin.command:
    cmd: openssl rand -hex -out api.token 16
    creates: api.token
```

ANSIBLE-PLAYBOOK CMD.YML (SECOND RUN)

```
TASK [Append with shell] *****
changed: [localhost]

TASK [Generate an API token once] *****
ok: [localhost]
```

- The same word applies to APIs (objective 2.3)
 - `GET`, `PUT`, and `DELETE` are idempotent by definition; `POST` is not, so a retried `POST` can create a duplicate

ON THE EXAM

- "Running the playbook again makes no further changes" or "safe to run repeatedly" means idempotent. "Describe the desired end state and let the tool decide the steps" means declarative.
- Ansible `shell` and `command` tasks report changed every run unless guarded with `creates`, `removes`, or `changed_when`. Replacing them with a purpose-built module is the usual fix.
- Do not assume imperative means non-idempotent: a script that checks state before acting can be idempotent too.

TRY IT

Run `site.yml` twice and compare the **PLAY RECAP** lines. Then change the address in `etc/resolv.conf` by hand and run it with `--check --diff` to watch it detect the drift without fixing it.

1.4 Syntax and Runtime Errors

1.0 AUTOMATION CODING CONCEPTS

Two kinds of failure

- Syntax error**: the code breaks the grammar of its language, so the parser rejects the file before anything runs
 - Missing colon, unclosed brace or quote, bad YAML indentation
 - Not even line 1 executes
- Runtime error**: the code is valid, but something fails while it executes
 - Host unreachable, file missing, permission denied, key not in the data
 - Everything before the failing line already ran, so a target can be left half-configured
- Undefined variable errors** land in either group, depending on the tool
 - Python and Ansible find them only when that line runs
 - Terraform, `shellcheck`, and Python linters catch them before anything runs
- Rule for pipelines: run the cheap syntax checks first, before anything touches a device

1.4 Syntax and Runtime Errors (continued)

LANGUAGE	CHECK WITHOUT RUNNING	LINTER THAT ALSO FINDS UNDEFINED NAMES
Python	<code>python3 -m py_compile app.py</code>	<code>ruff check</code> , <code>pylint</code> , <code>flake8</code>
Bash	<code>bash -n script.sh</code>	<code>shellcheck</code>
PowerShell	parsed when the script loads	<code>Invoke-ScriptAnalyzer</code> (PSScriptAnalyzer)
Terraform	<code>terraform validate</code> , <code>terraform fmt -check</code>	<code>tflint</code>
Ansible	<code>ansible-playbook --syntax-check</code>	<code>ansible-lint</code>
YAML, JSON	<code>yamllint</code> , <code>jq . file.json</code>	

Python: syntax errors and tracebacks

- A syntax error points at the line and column where the parser gave up
 - The real mistake can be on that line or just before it

```
$ python3 vlans.py
File "vlans.py", line 4
    for vid in vlan_ids
                ^
SyntaxError: expected ':'
```

- An **undefined variable** in Python is a `NameError`, raised only when that line runs
 - Here the loop variable is `vlan_id` but line 4 says `vlanid`
 - `py_compile` passes; the script prints `rendering`, then fails
 - Python 3.10 and later suggest the closest name

```
$ python3 render.py
rendering
Traceback (most recent call last):
  File "render.py", line 9, in <module>
    print(render({10: "users"}))
    ~~~~~^~~~~~
  File "render.py", line 4, in render
    lines.append(f"vlan {vlanid}")
                ^^^^^
NameError: name 'vlanid' is not defined. Did you mean: 'vlan_id'?
```

- Reading a **traceback**
 - Start at the last line: the exception type and message
 - Frames run from the outermost call down to the failure ("most recent call last"); the lowest frame in your own code is usually the bug

Runtime errors

- The same code works against one device and fails against another; the environment is the cause
- A push to a device whose service is down

TERMINAL

```
$ python3 push.py
Traceback (most recent call last):
  File "push.py", line 8, in <module>
    push("127.0.0.1", 18159, "vlan 10\n")
  File "push.py", line 4, in push
    with socket.create_connection(
         ^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/lib/python3.13/socket.py", line 868
    raise exceptions[0]
ConnectionRefusedError: [Errno 111] Connection
refused
```

- Expect it and handle it: **exception handling** with `try / except` `ConnectionRefusedError`, a one-line message, and a non-zero **exit code** (`sys.exit(2)`)
 - The pipeline still fails, but the log says why: `push failed: [Errno 111] Connection refused`

ERROR	USUAL CAUSE
<code>ConnectionRefusedError</code>	service down or wrong port
<code>TimeoutError</code>	firewall dropping traffic, wrong IP
<code>FileNotFoundError</code>	wrong path or working directory
<code>PermissionError</code>	file mode, missing <code>sudo</code>
<code>KeyError</code>	field missing from a dict or API reply
<code>TypeError: 'NoneType'...</code>	a lookup returned nothing

Bash: unset variables and set -u

- Bash expands an unset variable to an empty string and keeps going
 - `rm -rf "$STAGE_DIR/*"` with `STAGE_DIR` unset becomes `rm -rf /*`
- set -u** (part of `set -euo pipefail`) turns that into an **unbound variable** runtime error that stops the script

1.4 Syntax and Runtime Errors (continued)

BANNER.SH

```
#!/usr/bin/env bash
set -euo pipefail
site="CLT-01"
echo "site: ${site}"
echo "contact: ${site_contact}"
echo "done"
```

TERMINAL

```
$ bash banner.sh
site: CLT-01
banner.sh: line 5: site_contact: unbound variable
```

- Without `-u` the same script prints `contact:` with nothing after it, then `done`, and exits 0
- `shellcheck` warns before you run it: "SC2154 (warning): site_contact is referenced but not assigned."
- Make intent explicit: `${site_contact:-noc@example.com}` for a default, `${site_contact:?must be set}` to require it
- `bash -n` is the syntax check: an `if` with no `fi` reports `syntax error: unexpected end of file`

■ Terraform: validate before plan

- Referencing a variable that was never declared is an **undeclared input variable**, caught by `terraform validate`

TERMINAL

```
$ terraform validate

Error: Reference to undeclared input variable

   on main.tf line 7, in resource "local_file"
   "motd":
    7:   content = "Welcome to ${var.host_name}\n"

An input variable with the name "host_name" has not
been declared. Did you
mean "hostname"?
```

- A declared variable with no value is a missing **required variable** ("Error: No value for required variable" at `plan`)
 - Supply it with `-var`, a `.tfvars` file, `TF_VAR_hostname`, or a `default`
- A missing `}` is a syntax error: "Error: Unclosed configuration block"

- `validate` proves the code is well formed, not that it will work
 - A `local_file` path under `/proc` passes `validate` and `plan`, then `apply` fails at run time with "Error: Create local file error ... mkdir /proc/motd: no such file or directory"

■ Ansible: undefined variables at run time

- `--syntax-check` parses the YAML and play structure only; **templated values are not evaluated** until the task runs, so this passes the check and fails on the target

MOTD.YML (TASK)

```
- name: Write the banner
  ansible.builtin.copy:
    dest: "{{ playbook_dir }}/motd"
    content: "{{ site }} {{ site_contact }}\n"
    mode: "0644"
```

ANSIBLE-PLAYBOOK MOTD.YML (TRIMMED)

```
TASK [Write the banner] *****
[ERROR]: Task failed: ...
<<< caused by >>>

Error while resolving value for 'content':
'site_contact' is undefined
Origin: motd.yml:11:18
11         content: "{{ site }} {{ site_contact
   }}\n"
                                ^ column 18

fatal: [localhost]: FAILED! => ...
```

- Fixes
 - Define it where it belongs: `play vars`, `group_vars/`, `host_vars/`, role `defaults/`, or `-e`
 - Fall back with the **default filter**: `{{ site_contact | default('noc@example.com') }}`
 - Fail early with a clear message: `{{ site_contact | mandatory }}`

ON THE EXAM

- "The script fails before any output appears" points to a syntax error. "It worked on the lab switch but fails on the production switch" points to a runtime error caused by the environment.
- Know where each tool reports an undefined variable: Python raises `NameError` and Ansible reports `'x' is undefined` at run time; Bash prints `unbound variable` only with `set -u`; Terraform fails `validate` with "Reference to undeclared input variable."

■ Merge conflicts

- Three **Git errors** come up again and again: merge conflicts, authentication issues, and detached HEAD
- A **merge conflict** happens when two branches change the same lines, or one edits a file the other deleted
 - Git merges different lines on its own; it stops only when it cannot choose
- Any command that combines histories can hit one: `merge`, `pull`, `rebase`, `cherry-pick`, `stash pop`
- Here `main` and `feature/ntp` both changed the NTP server line (branching basics are in objective 1.2)

TERMINAL

```
$ git merge feature/ntp
Auto-merging switch.cfg
CONFLICT (content): Merge conflict in switch.cfg
Automatic merge failed; fix conflicts and then
commit the result.
$ git status --short
UU switch.cfg
```

- UU** (**both modified** in long `git status`) marks an unmerged file
- Git writes both versions into the file between **conflict markers**

SWITCH.CFG (DURING THE CONFLICT)

```
hostname core-sw1
<<<<<< HEAD
ntp server 10.0.0.3
=====
ntp server 10.0.0.2
>>>>>> feature/ntp
logging host 10.0.0.50
```

MARKER

STARTS

<<<<<< HEAD	your side: the branch you are on
<base>	the original text (zdiff3 style only)
=====	the incoming side
>>>>>> feature/ntp	end of the incoming side

■ Resolving a conflict

- Edit the file to the final content you want: one side, the other, or a mix
 - Delete every marker line; `git commit` will not stop you from committing them

- `git add` the file to mark it **resolved**, then finish
 - Merge: `git commit` (the message is prefilled)
 - Rebase: `git rebase --continue`
- Not ready? `git merge --abort` (or `git rebase --abort`) puts everything back
- Keep one whole side of a file with **ours or theirs**: `git checkout --ours switch.cfg` or `--theirs`
 - During a rebase the names swap: "ours" is the branch you are rebasing onto
- `git config --global merge.conflictStyle zdiff3` adds a `|||||| 309f3f2` section holding the original line (`ntp server 10.0.0.1`), so you can see what both sides changed
- Before committing, let Git find leftover markers: `git diff --check` prints `switch.cfg:2: leftover conflict marker` and exits non-zero
- Fewer conflicts: short-lived branches, pull `main` often, run the formatter so whitespace never conflicts

■ Authentication issues

- Two transports, two kinds of credential
 - HTTPS: username plus a **personal access token** (PAT) used as the password
 - SSH: a **key pair**; the public key is registered on your account, the private key never leaves your machine
- GitHub stopped accepting account passwords for Git in August 2021
 - Using one is rejected with "Invalid username or token. Password authentication is not supported for Git operations."
- Lab output below comes from a private Gitea server over plain HTTP; Git prints the same failures over HTTPS (Gitea adds `remote: Verify`)
- A stored token that has expired or been revoked

TERMINAL

```
$ git pull
remote: Verify
fatal: Authentication failed for
'http://git.corp.example:3000/ada/netconfig.git/'
```

- Fix: create a new token, then replace the cached one
 - The **credential helper** caches the old token; `git credential reject` (or Windows Credential Manager, macOS Keychain) removes the stale entry and the next command asks again

1.4 Troubleshooting Git (continued)

- A pipeline job with no credential cannot prompt, so it fails at once
 - Git reports "could not read Username for '...': terminal prompts disabled"
 - Give the job a token from the CI secret store, or a read-only deploy key
- SSH, with a key that is not on the account
 - `ssh -T` tests authentication without Git in the way; `ssh -vT` lists each key offered
 - Once the public key is on the account, the same test succeeds; GitHub answers "Hi *username*! You've successfully authenticated, but GitHub does not provide shell access."
 - `git remote -v` shows which transport a clone uses; `git remote set-url origin git@...` switches it

TERMINAL

```
$ ssh -T git@git.corp.example
git@git.corp.example: Permission denied (publickey).
```

MESSAGE (TRANSPORT)	LIKELY CAUSE	FIX
Authentication failed for '<url>' (HTTPS)	wrong, expired, or revoked token; password used	new token; clear the cached credential
could not read Username ... terminal prompts disabled (HTTPS)	no credential and no terminal (CI job)	token from the CI secret store
Repository not found, or HTTP 403 (HTTPS)	token scope or repository access missing	grant access or a token with write scope
Permission denied (publickey). (SSH)	key not on the account, wrong key, empty agent	add the <code>.pub</code> key; <code>ssh-add</code> ; check with <code>ssh -vT</code>
Host key verification failed. (SSH)	server key unknown or changed	verify the fingerprint; a surprise change can mean interception
UNPROTECTED PRIVATE KEY FILE! (SSH)	private key readable by others	<code>chmod 600 ~/.ssh/id_ed25519</code>

■ Detached HEAD

- Normally `HEAD` names a branch, and the branch names a commit

- **Detached HEAD:** `HEAD` points straight at a commit, with no branch in between

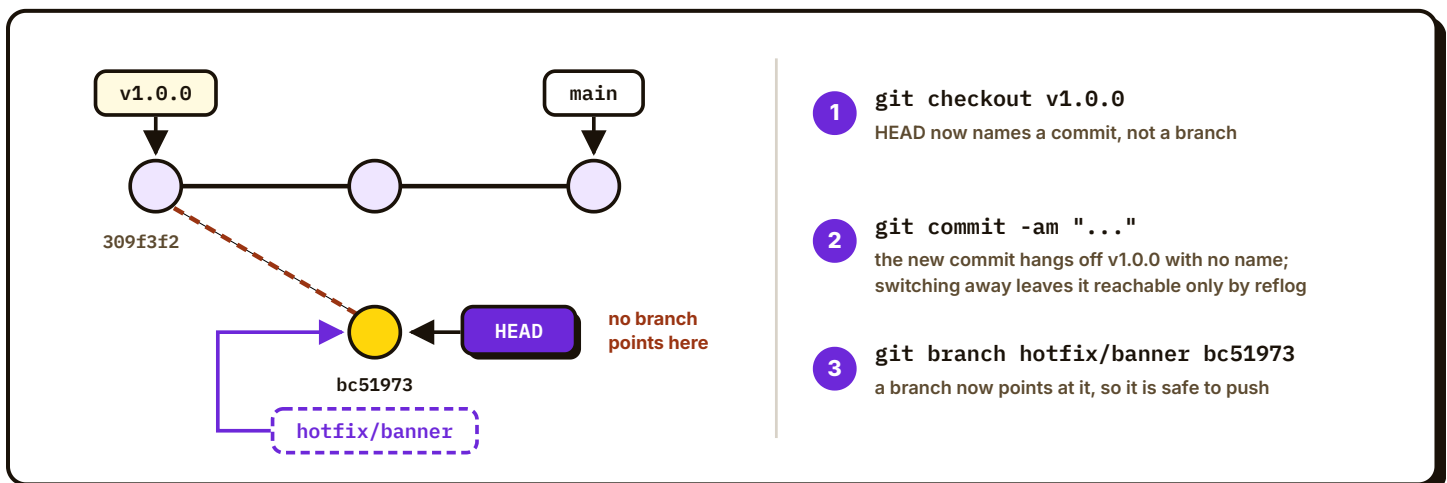


FIGURE A commit made with HEAD detached is attached to nothing. Give it a branch before you switch away, or recover it from the reflog afterwards.

- How you get there
 - `git checkout v1.0.0` or `git checkout <commit>` to inspect an old version
 - `git switch --detach`, an interrupted rebase, `git bisect`
 - CI runners check out the exact commit ID, so jobs usually run detached

1.4 Troubleshooting Git (continued)

TERMINAL

```
$ git checkout v1.0.0
Note: switching to 'v1.0.0'.

You are in 'detached HEAD' state.
HEAD is now at 309f3f2 feat: baseline config
$ git status
HEAD detached at v1.0.0
```

- Looking around is harmless; trouble starts when you commit
 - The commit exists, but no branch points at it
 - `git push` fails: **fatal: You are not currently on a branch.**
- Recover it with the **reflog**, the local record of everywhere **HEAD** has been

TERMINAL

```
$ git commit -am "fix: add login banner"
[detached HEAD bc51973] fix: add login banner
$ git switch main
Warning: you are leaving 1 commit behind, not
connected to
any of your branches:

    bc51973 fix: add login banner
$ git reflog -3
bc51973 HEAD@{1}: commit: fix: add login banner
309f3f2 HEAD@{2}: checkout: moving from main to
v1.0.0
$ git branch hotfix/banner bc51973
$ git push -u origin hotfix/banner
```

- Better: before committing while detached, run `git switch -c hotfix/ntp` right where you are
- The reflog is local only; by default entries for unreachable commits expire after 30 days, then garbage collection deletes the commits

ON THE EXAM

- "Git reports **CONFLICT (content)**" means: edit the file, remove the markers, `git add`, commit. "Start over" is `git merge --abort`.
- "Authentication failed" on an HTTPS remote points to the token (expired, revoked, missing scope, or a password used instead). "Permission denied (publickey)" points to the SSH key. Match the message to the transport.
- "Commits made after checking out a tag are missing" is detached HEAD. The answer is `git reflog` to find the commit ID, then `git branch <name> <id>` or `git switch -c`.

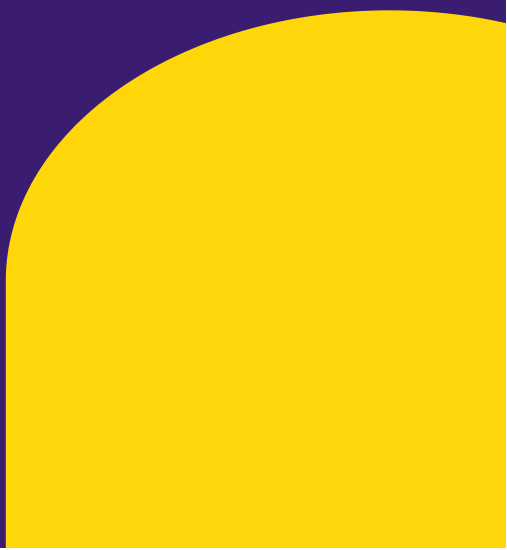
TRY IT

Clone any repository, run `git checkout HEAD~1`, and make a commit. Switch back to your branch, read the warning, then use `git reflog` to find the commit and save it on a new branch.

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

```

$ 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 (`azurearm`), 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

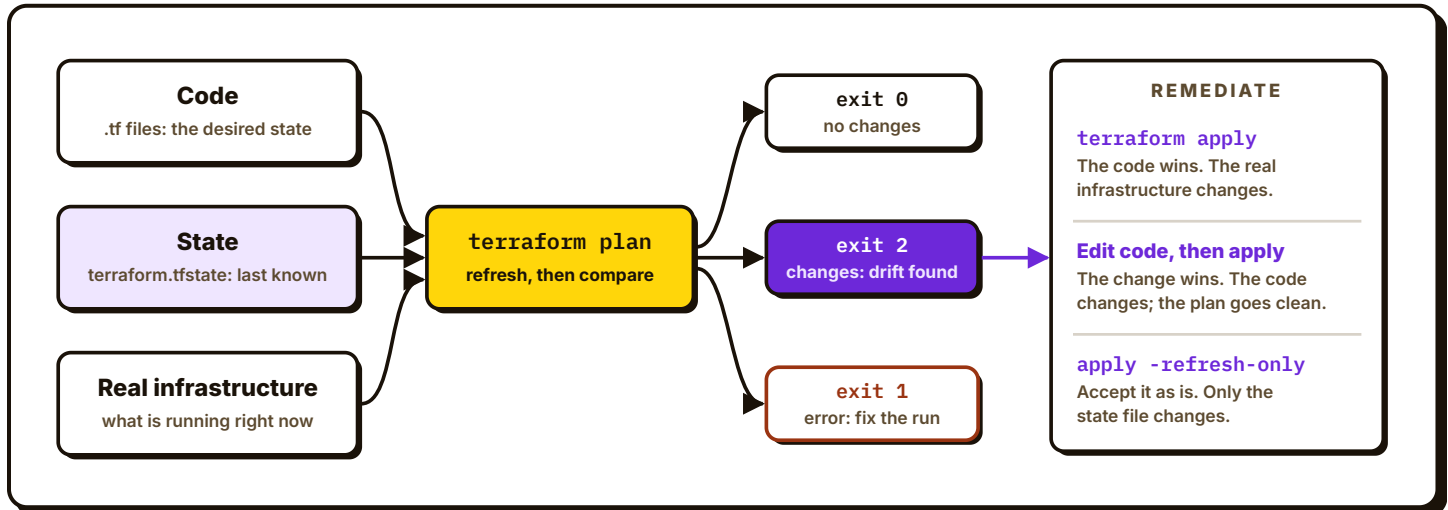


FIGURE Plan reads the real infrastructure, compares it to the code through state, and reports with an exit code. Each remediation moves a different side.

- 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.

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

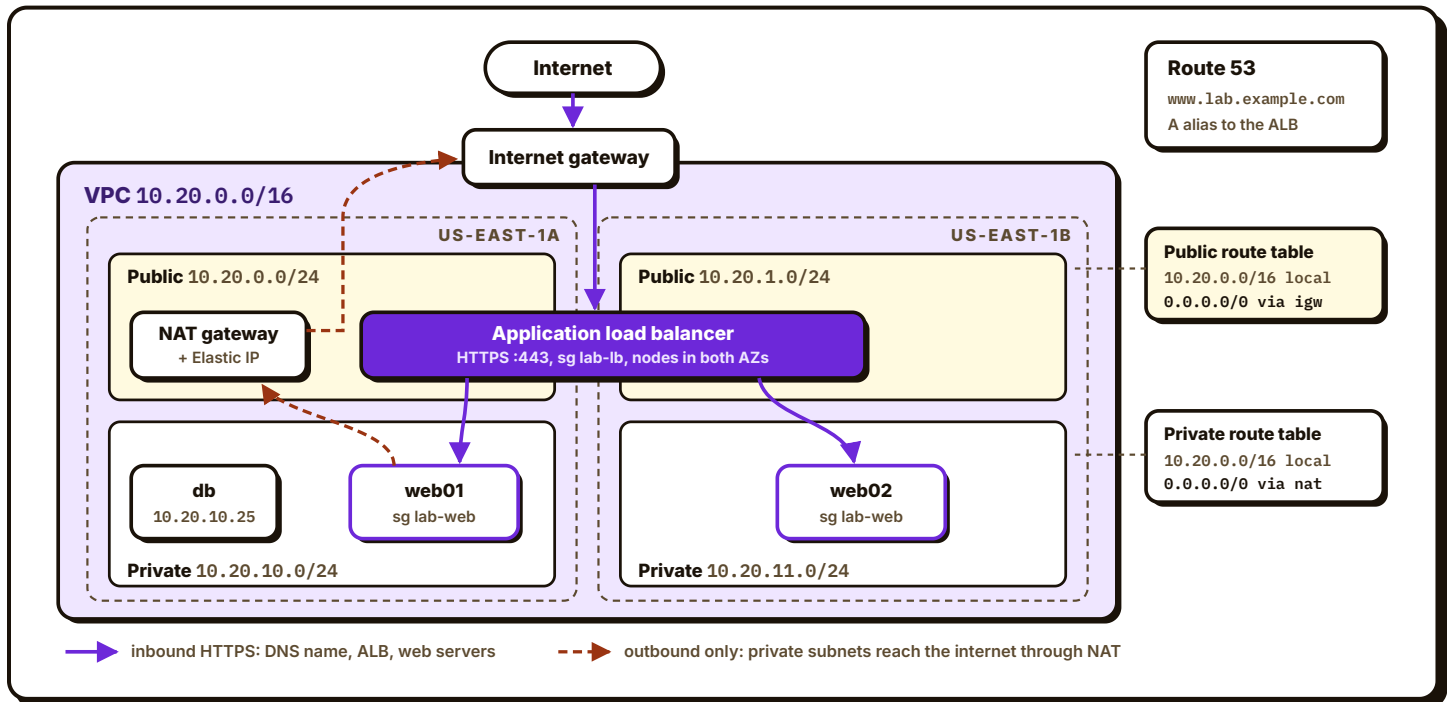


FIGURE The example in this topic: two availability zones, public subnets for the load balancer and NAT gateway, private subnets for servers, one route table per tier, and a DNS alias in front.

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.private.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)

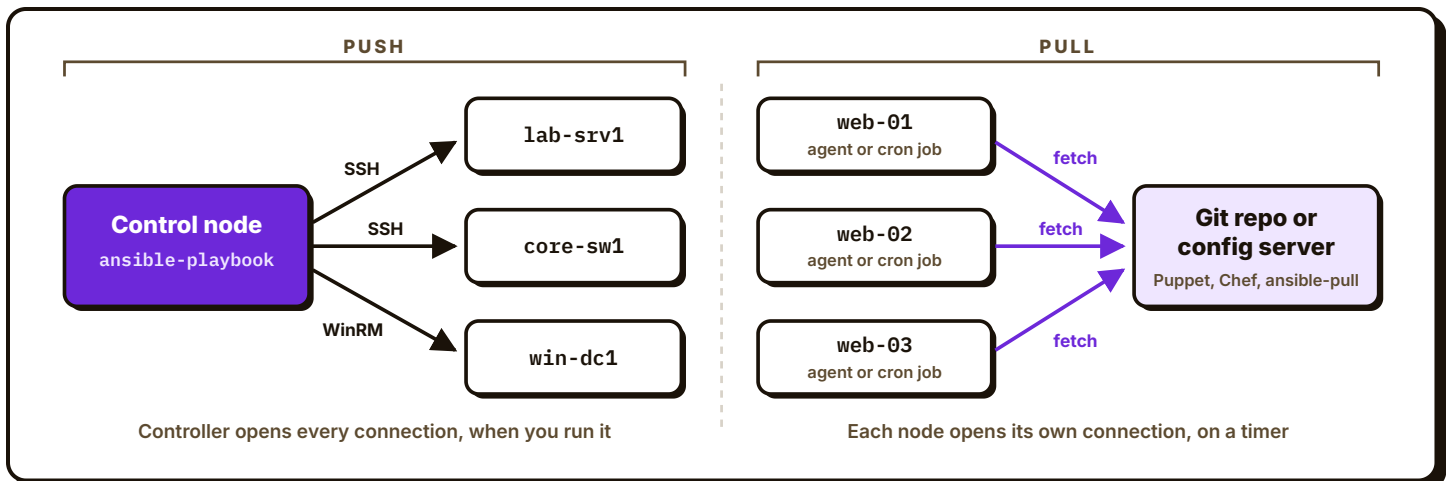


FIGURE Push: the control node starts the run and connects out to every target. Pull: each node starts its own run and fetches its configuration from a central source.

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

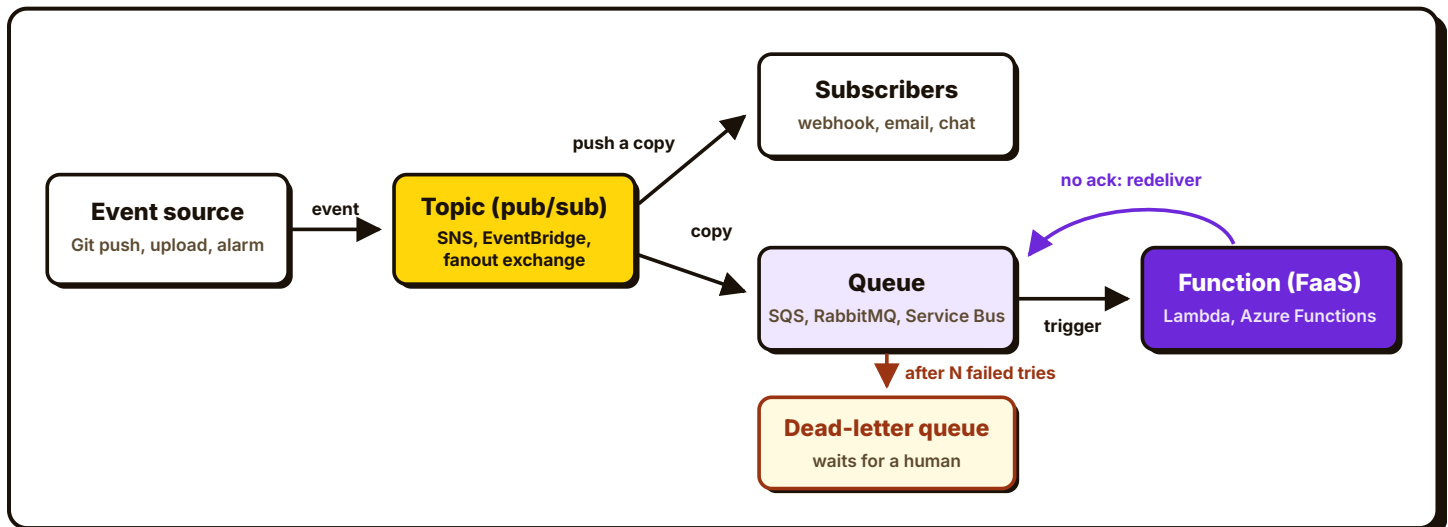


FIGURE A topic pushes a copy of each event to every subscriber. A queue holds work until one consumer finishes it; repeated failures move the message to a dead-letter queue.

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

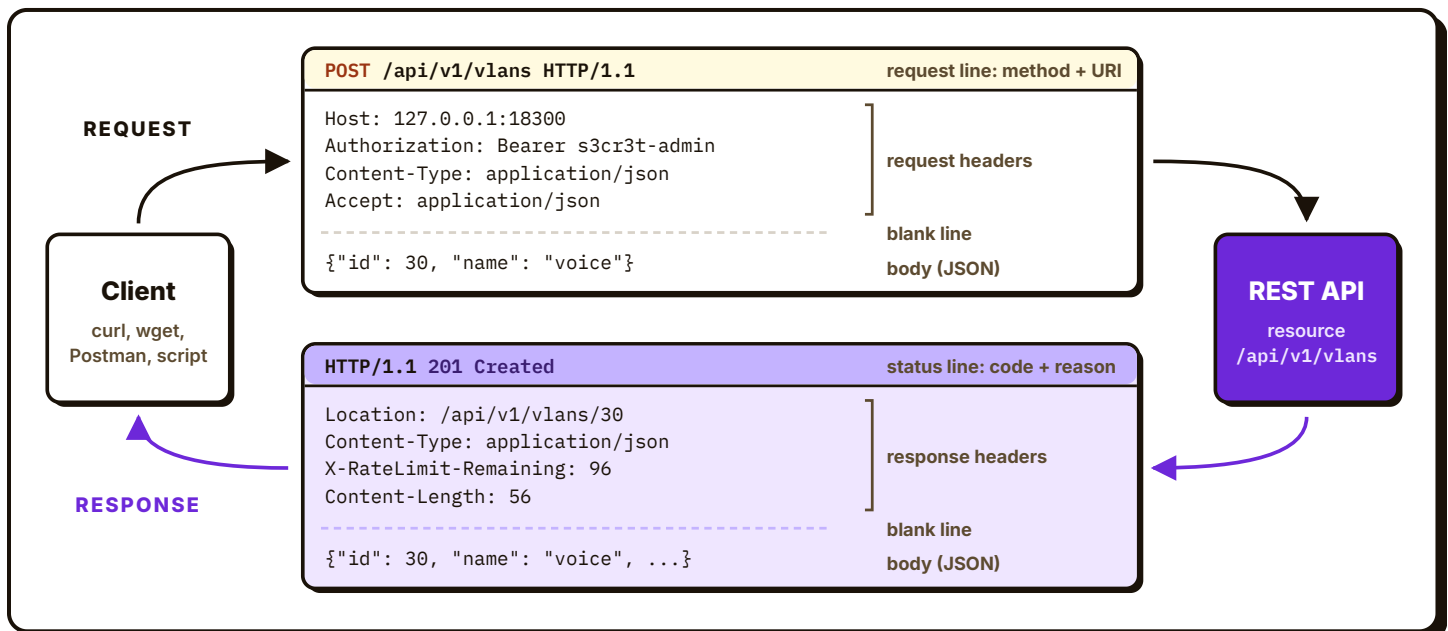


FIGURE One request and one response. Every HTTP message has the same four parts: a first line, headers, a blank line, and an optional body.

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"}
```

TERMINAL

```
$ 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
```

TERMINAL

- 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.

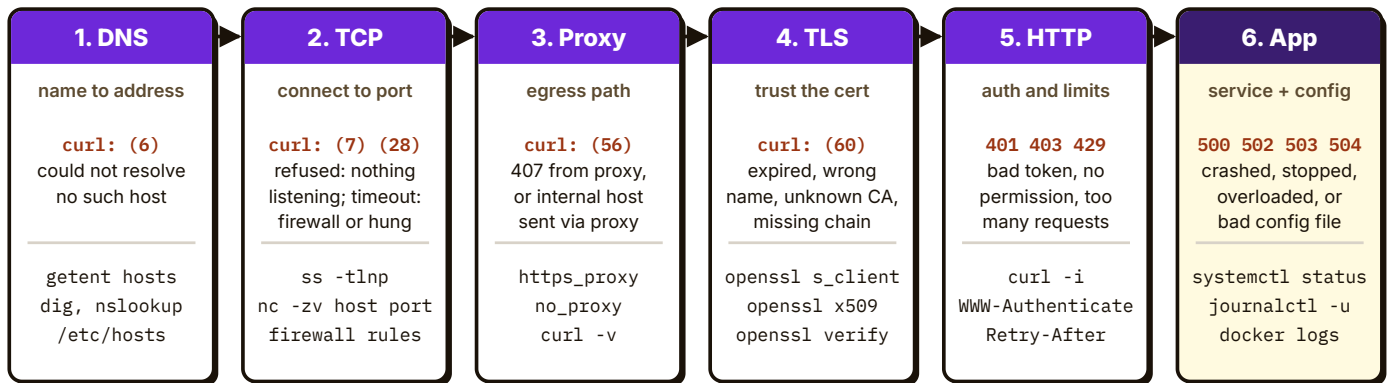


FIGURE Each layer depends on the one to its left. The curl exit code or HTTP status tells you which layer stopped the call.

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  truthful value should be one of [false, true] (truthy)
6:9       warning  truthful 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.

03

Continuous Integration

24% of the exam

-
- 3.1** Explain environmental factors related to CI management
 - 3.2** Explain workflow management concepts in CI
 - 3.3** Given a scenario, analyze configurations to manage basic automation pipelines
-

Secrets in the pipeline

- A **secret** is any value that grants access
 - API tokens, database passwords, SSH and TLS private keys, cloud access keys, SNMP community strings
- Secret sprawl**: one password pasted into CI variables, `.env` files, playbook vars, and a wiki page
 - Nobody knows every copy, so nobody can rotate it safely
- CI platforms **mask secrets** in job logs
 - GitHub Actions prints registered secrets as `***`; `::add-mask::VALUE` masks a value created during the run
 - GitLab: masked variables are hidden in logs; protected variables reach only protected branches and tags
 - Masking is cosmetic: a step can still base64 the value or send it over the network

Encrypted key vaults

- An **encrypted key vault** stores secrets encrypted at rest and releases them only to authenticated callers that a **policy** allows
 - Every read is in the audit log: "who read the switch password last Tuesday?"

VAULT	NOTES
HashiCorp Vault	self-hosted or HCP; KV, dynamic secrets, PKI
AWS Secrets Manager	KMS-encrypted; rotation by Lambda or managed rotation
Azure Key Vault	secrets, keys, certificates; RBAC; soft delete
Google Secret Manager	versioned; rotation reminders via Pub/Sub

- A dev server (`vault server -dev`) is unsealed and in-memory: lab only

VAULT 2.2 DEV SERVER

```
$ vault kv put -mount=secret ci/netbox \
  api_token=nb-7f3a9c
==== Secret Path ====
secret/data/ci/netbox
...
version            1
$ vault kv get -mount=secret -field=api_token \
  ci/netbox
nb-7f3a9c
```

Tokens, leases, and expiration

- Every Vault login returns a **token** with a **TTL** (time to live) and a policy

CI-READ POLICY

```
path "secret/data/ci/*" {
  capabilities = ["read"]
}
```

TERMINAL

```
$ vault token create -policy=ci-read -ttl=15s
Key          Value
---          -
token        hvs.CAESIFZ-NewiH1dhcsq4...
token_duration 15s
token_policies ["ci-read" "default"]
$ export VAULT_TOKEN=$CI_TOKEN # 15s later
$ vault kv get -mount=secret ci/netbox
Code: 403. Errors:
* 2 errors occurred:
  * permission denied
  * invalid token
```

- While it was valid, the same token could read but got **permission denied** on a write
- Lease management and expiration**
 - A **lease** is Vault's record of a secret it handed out: ID, duration, renewable or not
 - Renewing extends a lease but never past the role's **max_ttl**; Vault caps the request and warns
 - When a lease expires, Vault deletes the secret itself
- Size the TTL to the job: a 10-minute pipeline needs a credential for 10 minutes, not 90 days

Dynamic secrets and revocation

- A **dynamic secret** is created on demand, unique to one caller, and destroyed when its lease ends
 - Vault's database engine runs `CREATE ROLE` in Postgres for each request; this `ci` role has `default_ttl=2m` and `max_ttl=10m`
 - The read also returns a **lease_id** (trimmed below); save it

3.1 Secrets Management (continued)

TERMINAL

```
$ vault read db/creds/ci
lease_duration      2m
lease_renewable     true
password            rmnxoE6-NQf3ew1dX42T
username            ci-qr7cxiir
$ LEASE=db/creds/ci/B2tPc050C5LKh93jJMLDW6KJ
$ vault lease lookup $LEASE
expire_time        2026-10-19T12:57:53.343306968Z
ttl                1m58s
$ vault lease renew -increment=5m $LEASE
lease_duration      5m
$ vault lease revoke $LEASE
All revocation operations queued successfully!
```

- **Revocation** ends a secret before its TTL; here `ci-qr7cxiir` vanished from Postgres
 - Every credential a role issued: `vault lease revoke -prefix db/creds/ci` (the breach button)
 - `vault token revoke` also revokes the token's children and their leases

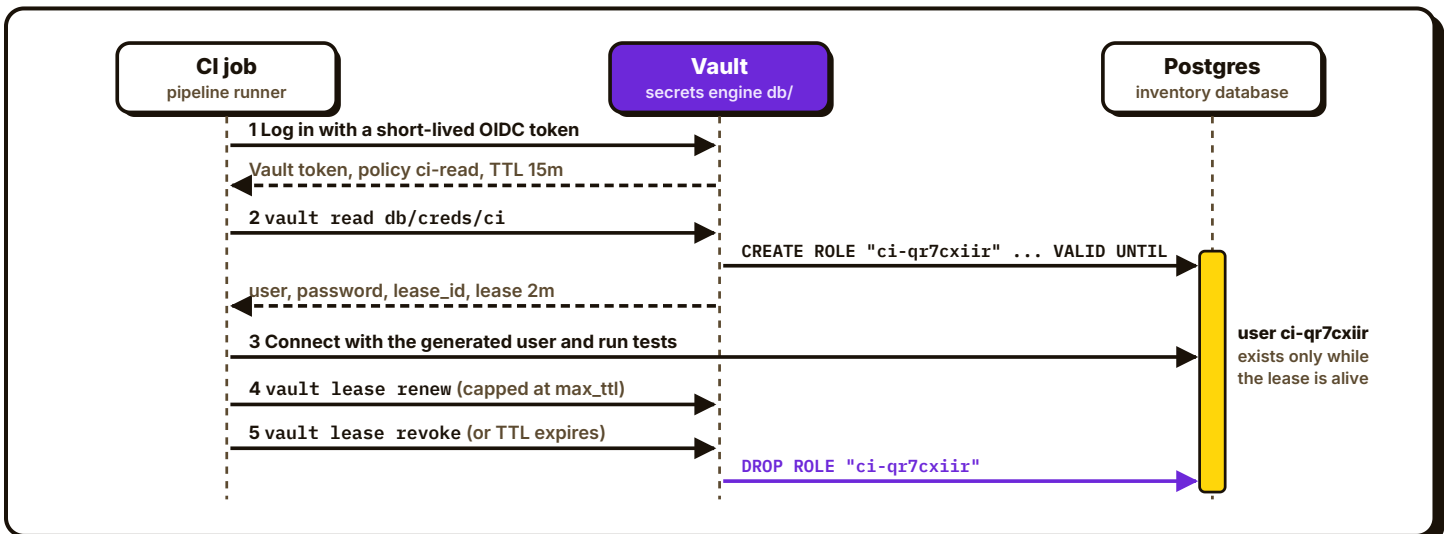


FIGURE A dynamic database secret. The database user exists only while its lease is alive; revoking the lease, or letting it expire, makes Vault drop the user.

- **Dynamic secret rotation:** credentials change so often that a leaked one soon stops working
 - Dynamic secrets rotate by design; static ones need a **rotation schedule** (Secrets Manager rotation, Key Vault expiry dates, Vault static roles)
 - `vault write -f db/rotate-root/inventory` gave the Postgres admin a password only Vault knows; the old one stopped working

	STATIC SECRET	DYNAMIC SECRET
Created	once, by a person	per request, by the vault
Shared	by every job	unique to one job
Lifetime	until someone rotates it	the lease TTL
After a leak	find and rotate everywhere	revoke the lease

Tokenization

- **Tokenization** swaps a sensitive value for a random stand-in (a token) with no mathematical link to the original
 - The original stays in a protected token vault; only authorized systems can **detokenize**
 - Encryption differs: anyone holding the key can reverse ciphertext
- Lets pipelines use realistic data without real data
 - Test fixtures carry tokenized card numbers, so CI logs and test databases never hold real ones; fewer systems in PCI DSS scope
- Offered by Vault Enterprise (Transform engine), payment providers, and cloud DLP services
 - Not the same as a Vault or API token: those are credentials, not stand-ins for data

3.1 Secrets Management (continued)

Zero Trust for pipelines

- **Zero Trust** (NIST SP 800-207): no trust based on network location; verify every request, least privilege, assume breach
- Applied to CI
 - No long-lived cloud keys stored in CI variables
 - The job proves its identity with a short-lived **OIDC** token; Vault checks the repository and branch claims before issuing anything
 - Per-job, short-TTL credentials on ephemeral runners; everything logged and revocable

.GITHUB/WORKFLOWS/DEPLOY-ACL.YML

```
name: deploy-acl
on: push
jobs:
  deploy:
    runs-on: ubuntu-latest
    permissions:
      id-token: write # request an OIDC token
      contents: read
    steps:
      - uses: actions/checkout@v6
      - uses: hashicorp/vault-action@v3
        with:
          url: https://vault.example.net
          method: jwt
          role: netops-deploy
          secrets: |
            secret/data/ci/netbox api_token | TOKEN
      - run: ./push-acl.sh # TOKEN shows as ***
```

- No secret is stored in the repository or CI settings; **vault-action** masks each value it fetches

ON THE EXAM

- Match the term to the behavior: a lease that the vault cleans up at its TTL is lease management and expiration; ending it early is revocation; a credential created per job is a dynamic secret.
- Masked CI variables hide values in logs but do not stop a malicious step from stealing them. Removing stored credentials from CI points to OIDC federation with a vault (Zero Trust).
- Tokenization replaces data with a meaningless stand-in and keeps the original in a vault; encryption transforms data with a key. A distractor will treat them as the same.

3.1 Artifact Management and SBOMs

3.0 CONTINUOUS INTEGRATION

Artifacts and where they live

- An **artifact** is anything a pipeline builds and keeps
 - Container images, Python wheels, npm packages, Ansible collection tarballs, Helm charts, Terraform modules, signed firmware bundles, test reports
- Build once, then promote the same artifact through test, staging, and production
 - Rebuilding for each stage means production runs something that was never tested
- An **artifact registry** stores artifacts with names, versions, and access control
 - Pipelines push to it; deploy jobs and servers pull from it

- An **external artifact repository** is one hosted outside your build system, by a vendor or a separate team
 - Also used as a **proxy** or cache for public registries, so builds survive a Docker Hub rate limit or a deleted upstream package

REPOSITORY	HOLDS
Docker Hub	public and private images
GitHub Packages (GHCR)	images, npm, Maven, NuGet
Sonatype Nexus	most formats, plus proxies
JFrog Artifactory	most formats, plus proxies
AWS ECR, Azure ACR, Google GAR	images in that cloud

3.1 Artifact Management and SBOMs (continued)

Authentication and authorization

- **Authentication**: who are you?
 - A password, a scoped access token, or a short-lived OIDC token from the CI platform
 - HTTP **401 Unauthorized** means the registry does not know who you are
- **Authorization**: what may you do?
 - Pull, push, delete, or administer, per repository or project
 - HTTP **403 Forbidden** means it knows who you are and the answer is no
- The lab registry is **registry:2** (CNCF Distribution) with **htpasswd** users

```
$ REG=localhost:18352/netops/cfg-backup
$ curl -s -o /dev/null -w '%{http_code}\n' \
  http://localhost:18352/v2/_catalog
401
$ docker push $REG:1.4.0
...
push access denied, ... no basic auth credentials
$ echo 'Pipeline!' | docker login localhost:18352 \
  -u ci-bot --password-stdin
WARNING! Your credentials are stored unencrypted in
'/tmp/tmp.1fG6coP3mo/docker/config.json'.
...
Login Succeeded
```

- That warning matters on shared runners
 - Configure a **credential helper** (**docker-credential-pass**, **osxkeychain**, **wincrd**, or a cloud helper), and **docker logout** at the end of the job
- Plain **registry:2** stops at authentication
 - In the test, a second account named **auditor** pushed a new tag without any trouble
 - Harbor, Nexus, Artifactory, and the cloud registries add roles: a CI robot account that can push to one project, and pull-only accounts for servers

Versioning: tags and digests

- A **tag** is a human-friendly name that points at an image: **1.4.0**, **1.4**, **latest**
 - Tags are **mutable** by default: pushing the same tag again moves it
- A **digest** is the SHA-256 hash of the image manifest: **sha256:d43a...**
 - Change one byte of content and the digest changes, so a digest always means the same image
- Version with semantic versioning tags; deploy by digest

TERMINAL

```
$ V1=$(docker push $REG:1.4.0 \
  | awk '/digest:/{print $3}')
$ echo $V1
sha256:d43a49b3bbb5b75b35ef09deb88ed2...
```

- Edit **backup.sh** to print **backup v2**, rebuild, and push the same tag

TERMINAL

```
$ docker push $REG:1.4.0 | awk '/digest:/{print $3}'
sha256:493ed42735d69488582c190b7240e3...
$ docker run --rm $REG:1.4.0
backup v2
$ docker run --rm $REG@$V1
backup v1
```

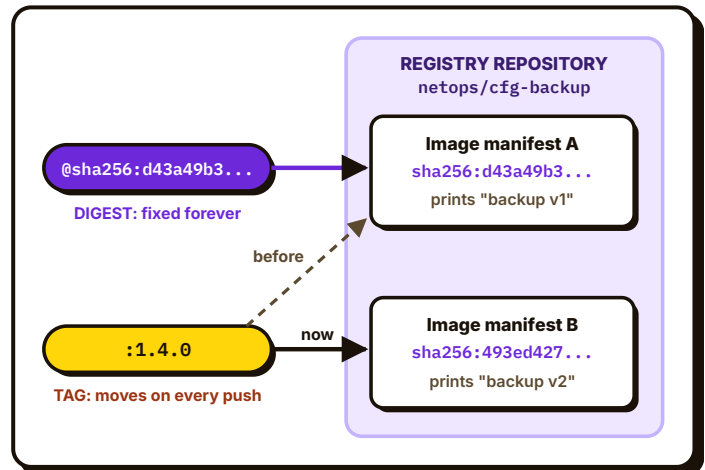


FIGURE Re-pushing 1.4.0 moved the tag to a new image. The digest still names the original, which the registry kept.

- **latest** is only a tag; nothing makes it the newest image
- Record the digest the build produced and pass it to the deploy job; Kubernetes manifests and Compose files accept **image@sha256:...**

Immutability

- **Immutability**: once a version is published, its contents never change
 - Same rule as SemVer: a fix is a new version, never a re-push
- Turn it on in the registry
 - Amazon ECR: tag mutability set to **IMMUTABLE**
 - Harbor: tag immutability rules per project
 - Artifactory and Nexus: release repositories that refuse to redeploy an existing version
- A digest is immutable by definition; an immutable tag makes the tag as trustworthy as the digest

3.1 Artifact Management and SBOMs (continued)

- Pair with **signing**: `cosign sign` attaches a signature to the digest, and the deploy side verifies it before pulling

■ Software bill of materials (SBOM)

- A **software bill of materials (SBOM)** is the ingredient list of an artifact
 - Every package inside it: name, version, supplier, license, and a package URL (`purl`) or CPE that scanners can match
- Two standard formats
 - SPDX** (Linux Foundation, ISO/IEC 5962): strong on licenses
 - CycloneDX** (OWASP): strong on security and dependency data
- Why it matters
 - A new CVE lands (think Log4Shell): search your SBOMs instead of opening every image
 - License review before shipping; US federal buyers ask for SBOMs (Executive Order 14028)
- syft** (Anchore) reads an image, directory, or archive and writes either format

- `backup.sh` is not listed: an SBOM lists packages a tool can identify (apk, deb, rpm, pip, npm, Go modules, Java JARs), not loose scripts
- Scan by digest so the SBOM describes exactly the image you deploy
- Publish the SBOM with the artifact: `docker buildx build --sbom=true` attaches one to the image, and `grype sbom:sbom.cdx.json` checks it for known vulnerabilities

ON THE EXAM

- A tag can move; a digest cannot. A scenario where "the same version" behaves differently on two servers points to a re-pushed mutable tag; the fix is immutable tags or deploying by digest.
- 401 is an authentication problem (no or bad credentials); 403 is an authorization problem (valid user, missing permission).
- SPDX and CycloneDX are SBOM formats. An SBOM lists components and versions; it does not by itself say which ones are vulnerable; that takes a scanner.

SYFT 1.54

```
$ export SYFT_REGISTRY_INSECURE_USE_HTTP=true
$ syft scan -q "registry:$REG@$V1" -o table
NAME                VERSION    TYPE
alpine-baselayout    3.6.5-r0  apk
alpine-keys          2.4-r1    apk
busybox              1.36.1-r31 apk
ca-certificates-bundle 20260413-r0 apk
...
$ syft scan -q "registry:$REG@$V1" \
  -o spdx-json=sbom.spdx.json \
  -o cyclonedx-json=sbom.cdx.json
$ jq -r .spdxVersion sbom.spdx.json
SPDX-2.3
$ jq '.components[]
  | select(.name=="libssl3")
  | {name, version, type,
    license: .licenses[0].license.id}' \
  sbom.cdx.json
{
  "name": "libssl3",
  "version": "3.3.7-r0",
  "type": "library",
  "license": "Apache-2.0"
}
```

What a task runner does

- A **task runner** gives each repeatable job a short name
 - `make test`, `task lint`, `npm run build`, `invoke deploy`
 - Each task is a list of shell commands, plus its **dependencies**: tasks that must run first
- Why automation teams use one
 - One command replaces ten copied from a wiki
 - The CI file stays short: it calls tasks instead of repeating their commands
 - Laptop and runner run the same steps, so "works on my machine" bugs show up locally
- Not a CI server: the pipeline decides when to run; the task runner decides what runs

RUNNER	TASK FILE	RUN A TASK
GNU Make	Makefile	<code>make test</code>
Task (go-task)	Taskfile.yml	<code>task test</code>
npm scripts	package.json "scripts"	<code>npm run test</code> (or <code>npm test</code>)
Invoke (Python)	tasks.py with @task functions	<code>invoke test</code>
just	justfile	<code>just test</code>

Make

- The oldest runner, installed on almost every Linux image
- A **target** names a task; its **prerequisites** run first; its **recipe** lines run in a shell
 - Recipe lines must start with a real tab; spaces give **missing separator**
 - .PHONY** marks targets that are task names, not files to build
 - @** at the start of a line hides the command echo
- Each recipe line is checked: a **non-zero exit** stops make, and make itself exits `2`

MAKEFILE

```
COMPOSE := docker compose -f compose.test.yml
PSQL := $(COMPOSE) exec -T db psql -U ci -tA
.PHONY: lint test-env test clean

lint:
    yamllint -s inventory/

test-env:
    $(COMPOSE) up -d --wait

test: lint test-env
    @$(PSQL) -c "SELECT count(*) FROM devices" \
    inventory | grep -qx 3
    @echo "inventory: 3 devices, tests passed"

clean:
    $(COMPOSE) down -v
```

- `make -n test` prints every command it would run, without running any of them
- Environment provisioning**
 - Environment provisioning**: create the services a test needs, just before the test
 - A database, a mock API, a DNS server, a containerized network OS image
 - Throwaway and identical every run: built from files in the repository, seeded with known data, destroyed afterward
 - No shared "test database" that another job changed yesterday
 - Docker Compose** is the usual tool for a few services

COMPOSE.TEST.YML

```
services:
  db:
    image: postgres:17-alpine
    environment:
      POSTGRES_USER: ci
      POSTGRES_PASSWORD: throwaway-test-only
      POSTGRES_DB: inventory
    ports:
      - "127.0.0.1:18353:5432"
    volumes:
      - ./seed:/docker-entrypoint-initdb.d:ro
    healthcheck:
      # -h 127.0.0.1: TCP is up only after seeding
      test: pg_isready -h 127.0.0.1 -U ci
      interval: 2s
      retries: 15
```

3.1 Task Runners and Environment Provisioning (continued)

- `up -d --wait` returns only when every service is **healthy**
 - Without a health check, tests start while Postgres is still booting and fail at random
 - While it runs the `seed/` SQL, Postgres listens only on its local socket, so a TCP check waits for seeding
- The password guards a container that lives seconds; real credentials still come from a vault

```
$ make test
yamllint -s inventory/
docker compose -f compose.test.yml up -d --wait
Container ao-f-inventory-db-1 Started
Container ao-f-inventory-db-1 Waiting
Container ao-f-inventory-db-1 Healthy
inventory: 3 devices, tests passed
$ make clean
docker compose -f compose.test.yml down -v
Container ao-f-inventory-db-1 Removed
Network ao-f-inventory_default Removed
```

TERMINAL

- A second `make test` reuses the running container, so the laptop edit-test loop stays fast
- `down -v` also deletes the data volume; the next run starts from the seed

One entry point, local and CI

- The pipeline calls the same **entry point** a person would

.GITHUB/WORKFLOWS/INVENTORY.YML

```
name: inventory
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - run: make test
      - if: always() # tear down even on failure
        run: make clean
```

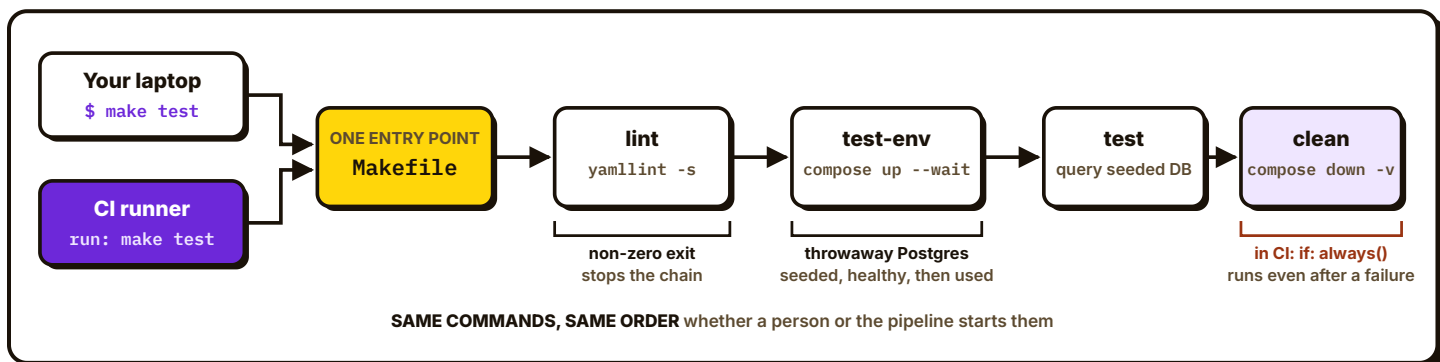


FIGURE `make test` runs lint, test-env, and test in order on any machine. `make clean` is a separate step; in CI, `if: always()` runs it even when a test failed.

- Moving to GitLab CI or Jenkins changes a few lines of YAML, not the test logic
- A failing task fails the step, because the **exit code** passes straight through

```
$ make lint
yamllint -s inventory/
inventory/broken.yml
...duplication of key "ansible_host" in mapping...
make: *** [Makefile:6: lint] Error 1
$ echo $?
2
```

TERMINAL

Task and other runners

- **Task (go-task)** reads YAML, runs on Linux, macOS, and Windows, and ships as one binary
 - `desc:` documents each task; `task --list` prints them
 - `deps:` run in parallel before the task; `sources:` and `generates:` skip work that is already up to date

3.1 Task Runners and Environment Provisioning (continued)

TASKFILE.YML

```
version: '3'
vars:
  COMPOSE: docker compose -f compose.test.yml
tasks:
  test-env:
    desc: Start a throwaway Postgres
    cmds:
      - '{{.COMPOSE}} up -d --wait'
  test:
    desc: Lint, provision, and test
    deps: [test-env]
    cmds:
      - yamllint -s inventory/
      - >-
        '{{.COMPOSE}} exec -T db psql -U ci -tA
        -c "SELECT count(*) FROM devices"
        inventory | grep -qx 3
  clean:
    desc: Remove the test environment
    cmds:
      - '{{.COMPOSE}} down -v'
```

TASK 3.54

```
$ task --list
task: Available tasks for this project:
* clean:      Remove the test environment
* test:       Lint, provision, and test
* test-env:   Start a throwaway Postgres
```

- **npm scripts** live in `package.json`; `pretest` and `posttest` scripts run automatically around `test`
- **Invoke** suits Python teams: tasks are decorated functions, so they can loop over inventory or call libraries before running commands with `c.run()`

ON THE EXAM

- A task runner (make, Task, npm scripts, Invoke) defines named, repeatable steps. A scenario where CI and developers run tests differently is solved by one task entry point both of them call.
- Environment provisioning creates the services a test needs (often `docker compose up`) before the tests and removes them after. Watch for answers that share one long-lived test environment between jobs.
- A cleanup step must run even after a failure: `if: always()` in GitHub Actions, `after_script` in GitLab CI, `post { always { } }` in Jenkins.

3.2 Pipeline Orchestration

3.0 CONTINUOUS INTEGRATION

■ Pipelines, stages, jobs, and steps

- A **pipeline** is the full automated run that a trigger starts
- A **stage** is a named phase: `lint`, `test`, `build`, `deploy`
- A **job** is one unit of work inside a stage
 - Gets its own runner (GitLab) or runner machine (GitHub), starts from a clean workspace
 - A job passes or fails as a whole
- A **step** is one command or action inside a job
 - Steps in a job share the same workspace and run in order
- **Orchestration techniques** are the rules that decide
 - Which jobs run, in what order, and how many at once
 - What happens when one of them fails

- Same ideas, different words
 - Jenkins: a pipeline has `stages`, each stage has `steps`; one agent and one workspace unless you say otherwise
 - GitHub Actions: a workflow has `jobs`, each job has `steps`; no stage keyword

■ Parallel vs. sequential

- **Sequential**: each piece waits for the one before it
 - Stages always run in order; stage 2 starts only after every job in stage 1 passes
 - Safe and easy to read, but the pipeline is as slow as the sum of its parts
- **Parallel**: independent jobs run at the same time
 - Jobs in the same stage run in parallel when runners are free
 - Lint the YAML and lint the Python at once; neither needs the other

3.2 Pipeline Orchestration (continued)

- A **matrix** fans one job out into many parallel copies
 - GitHub: `strategy.matrix`; GitLab: `parallel:` `matrix:`; Jenkins: a `matrix` or `parallel` block
 - One test job per site, OS, or Python version
- Matrix controls
 - `fail-fast: false` keeps the other copies running when one fails (GitHub defaults to `true`)
 - `max-parallel` caps how many run at once, so you do not hit 40 switches at the same moment
- Parallel is capped by runner capacity: ten jobs on two runners still run two at a time

Step dependencies

- Stage order alone is coarse: one slow test job holds back every job in the next stage
- `needs:` declares the real **dependencies** between jobs
 - A job starts the moment the jobs it needs succeed, whatever stage it is in
 - The jobs and their `needs` form a **directed acyclic graph (DAG)**: arrows point one way and never loop back
 - `needs: []` starts a job immediately, with no dependencies
- GitHub uses the same keyword: `needs: build` or `needs: [lint, unit]`

- Jenkins runs stages in order; use a `parallel` block for branches of work

.GITLAB-CI.YML (PART 1)

```
stages: [lint, test, build, deploy]

lint-yaml:
  stage: lint
  script: [echo "inventory ok"]

lint-python:
  stage: lint
  script: [echo "ruff ok"]

unit-tests:
  stage: test
  needs: [lint-python]
  script: [sleep 3, echo "14 passed"]

smoke-test:
  stage: test
  needs: [lint-yaml]
  allow_failure: true
  script: [exit 1]

render-configs:
  stage: build
  needs: [lint-yaml]
  script: [echo "rendered 12 configs"]
```

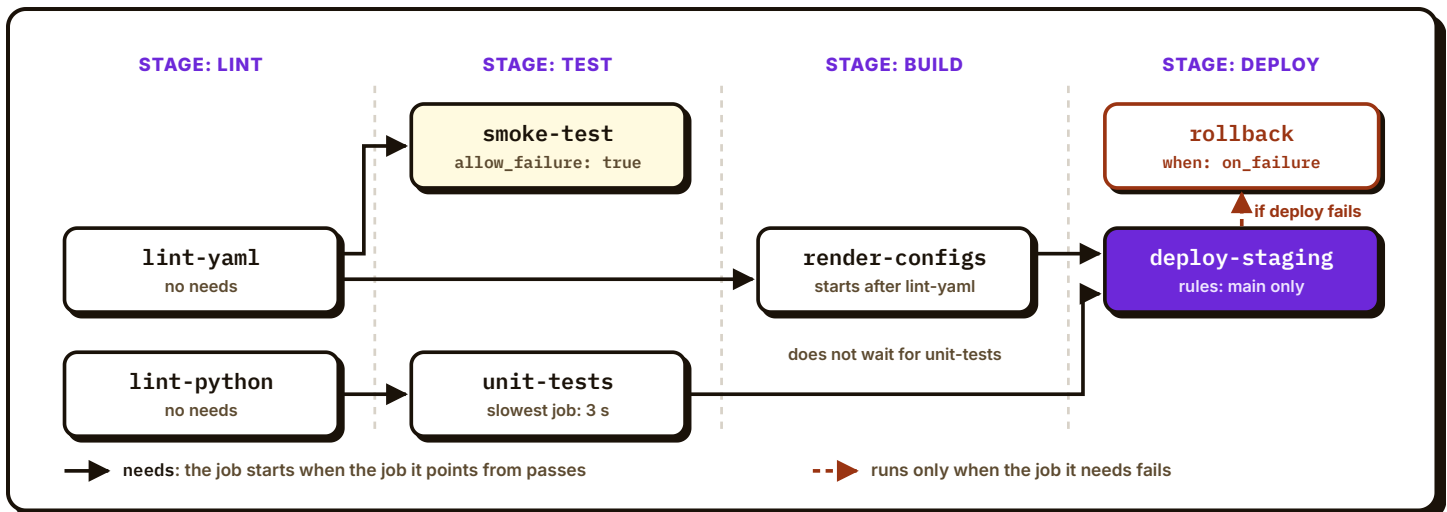


FIGURE The needs graph for this pipeline. render-configs is in a later stage than unit-tests, yet it starts as soon as lint-yaml passes.

3.2 Pipeline Orchestration (continued)

- Run it locally with `gitlab-ci-local`: `render-configs` (build stage) finishes while `unit-tests` (test stage) is still running

TERMINAL

```
$ gitlab-ci-local
lint-yaml      starting shell (lint)
lint-python    starting shell (lint)
lint-yaml      finished in 48 ms
lint-python    finished in 50 ms
smoke-test     starting shell (test)
render-configs starting shell (build)
unit-tests     starting shell (test)
smoke-test     finished in 53 ms  WARN 1
render-configs finished in 56 ms
unit-tests     finished in 3.04 s
deploy-staging starting shell (deploy)
deploy-staging finished in 19 ms
```

Conditional execution

- Conditional execution** decides whether a job runs at all for this pipeline
- GitLab: `rules`: evaluated top to bottom; the first match wins
 - `if`: tests variables: `$CI_COMMIT_BRANCH == "main"`
 - `changes`: runs only when matching files changed:
`changes: [inventory/**/*]`
 - Each rule can set `when`: (`on_success`, `manual`, `on_failure`, `never`); no match means the job is left out
- GitHub: `if`: on a job or step
 - `if: github.ref == 'refs/heads/main'`
 - Status functions**: `success()` (the default), `failure()`, `always()`, `cancelled()`
- Jenkins: a `when` block on a stage: `when { branch 'main' }, when { buildingTag() }`

- On a feature branch, the deploy rules evaluate to `never`

TERMINAL

```
$ git switch -c feature/NET-200-ac1
$ gitlab-ci-local --list-json \
  | jq -r '.[ ] | "\(.name) \(.when)"'
lint-yaml on_success
lint-python on_success
unit-tests on_success
smoke-test on_success
render-configs on_success
deploy-staging never
rollback never
```

Failure handling

- Default behavior: a failed job fails the pipeline and later jobs that depend on it do not run
- Tolerate a known-flaky or advisory job
 - GitLab `allow_failure: true`: the job shows a warning (`WARN` above) and the pipeline continues
 - GitHub `continue-on-error: true` on a step or a job
- Retries** for failures that are not your code's fault
 - GitLab `retry: 2`, or `retry: {max: 2, when: runner_system_failure}`
 - Jenkins `retry(3) { ... }`; GitHub has no `retry` keyword (re-run the job, or retry inside the script)
 - Never retry a failing test until it passes; that hides a real bug
- Timeouts** stop a hung job from holding a runner for hours
 - GitHub's default `timeout-minutes` is 360; set your own
- Cleanup and notification jobs run on failure or always (table below)

NEED	GITLAB CI	GITHUB ACTIONS	JENKINS (DECLARATIVE)
Keep going after a failure	<code>allow_failure: true</code>	<code>continue-on-error: true</code>	<code>catchError</code> / <code>unstable</code>
Retry	<code>retry: 2</code>	re-run job	<code>retry(3)</code>
Time limit	<code>timeout: 10m</code>	<code>timeout-minutes: 10</code>	<code>timeout(...)</code> in <code>options</code>
Run only after a failure	<code>when: on_failure</code>	<code>if: failure()</code>	<code>post { failure { } }</code>
Always run (cleanup)	<code>when: always</code>	<code>if: always()</code>	<code>post { always { } }</code>

Automated rollbacks

- An **automated rollback** returns production to the **last known good** state without a person typing commands at 2 a.m.

- The pipeline detects the failure
 - The deploy command exits non-zero, or a post-deploy health check or smoke test fails
- The pipeline reacts with a job that runs only on failure

3.2 Pipeline Orchestration (continued)

.GITLAB-CI.YML (PART 2)

```
deploy-staging:
  stage: deploy
  needs: [unit-tests, render-configs]
  timeout: 10m
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
  script:
    - echo "pushing configs"
    - test "$BREAK_DEPLOY" != "yes"

rollback:
  stage: deploy
  needs: [deploy-staging]
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
      when: on_failure
  script: [echo "restoring last good configs"]
```

TERMINAL

```
$ gitlab-ci-local --variable BREAK_DEPLOY=yes
deploy-staging $ test "$BREAK_DEPLOY" != "yes"
deploy-staging finished in 12 ms FAIL 1
rollback      > restoring last good configs
PASS rollback
FAIL deploy-staging
```

- The same pattern in GitHub Actions: a `rollback` job with `needs: deploy` and `if: failure()`
 - With `act`, a failed deploy shows `deploy failure` then `rollback success`

- Rollback needs something to roll back to
 - Versioned, immutable artifacts: redeploy build `v1.3.2`, never "rebuild the old code"
 - Kubernetes: `kubectl rollout undo`; Helm 4: `helm upgrade --rollback-on-failure` (Helm 3 called it `-atomic`)
 - Blue/green: switch traffic back to the old environment
 - Network devices: Junos `commit confirmed 5` reverts unless confirmed; Cisco `configure replace` to an archived config
- Database schema changes are the hard case: make migrations backward compatible so old code still runs

ON THE EXAM

- "Run independent jobs at the same time to shorten the pipeline" is parallel execution; "start a job as soon as the specific jobs it depends on finish" is `needs` (a DAG), not stage order.
- Match the keyword to the behavior:
 - `allow_failure` / `continue-on-error` tolerate a failure, `retry` repeats on transient errors, `timeout` limits a hung job, and `when: on_failure` / `if: failure()` / `post { failure }` run rollback or alert jobs.
- A scenario where a job never ran on a feature branch usually points to a `rules`, `if`, or `when` condition, not a broken script.

TRY IT

Copy both parts of the `.gitlab-ci.yml` into a new Git repository, run `gitlab-ci-local`, then add `needs: []` to `render-configs` and run it again. Watch when it starts compared with `lint-yaml`.

3.2 Pipeline Enhancements: Scanning, Containers, and AI

3.0 CONTINUOUS
INTEGRATION

■ Checks that make a pipeline worth trusting

- A pipeline that only builds and deploys ships bugs and vulnerabilities faster

3.2 Pipeline Enhancements: Scanning, Containers, and AI (continued)

- **Pipeline-enhancing concepts** add automated checks between commit and deploy
 - **Security scanning**: code, dependencies, images, secrets
 - Code quality scanning: style, complexity, duplication, test coverage
 - **Containerization**: every job runs in a known, pinned environment
 - AI-based log analysis: help reading a failed job's output
- Each scanner becomes a **gate**: a non-zero exit code fails the job and blocks the merge or deploy
- Run the cheap checks first; a 5-second linter should not wait behind a 10-minute image build

SCAN	WHAT IT READS	FINDS	EXAMPLE TOOLS
SAST (static application security testing)	your source code	injection, unsafe calls, hardcoded passwords	Bandit, Semgrep, CodeQL
SCA / dependency scanning	lock and requirements files	libraries with known CVEs, license issues	Trivy, Gype, pip-audit, Dependabot
Container image scanning	OS packages and libraries inside an image	CVEs in the base image and installed packages	Trivy, Gype, Clair
Secret scanning	files and Git history	API keys, tokens, private keys	gitleaks, TruffleHog, push protection
IaC scanning	Terraform, Kubernetes, Dockerfiles	misconfigurations such as public buckets	Checkov, Trivy config, tfsec
DAST (dynamic testing)	the running application	issues visible from outside	OWASP ZAP

■ Static and dependency scanning

- **SAST** reads source code without running it, like a linter with a security rule set
- Bandit (Python) on a backup script: every finding has a test ID and a severity

BACKUP.PY

```
import subprocess

PASSWORD = "Sw1tch!Adm1n"

def backup(host):
    cmd = f"scp admin@{host}:running-config ."
    subprocess.run(cmd, shell=True)
```

TERMINAL

```
$ bandit -q -f custom \
  --msg-template '{line}:{test_id}:{severity}' \
  backup.py
1:B404:LOW
3:B105:LOW
8:B602:HIG
$ echo $?
1
```

- **B602**: `subprocess` with `shell=True`; a host name like `x; rm -rf /` becomes a command

- **B105**: hardcoded password; **B404**: importing `subprocess` at all (informational)
- **Software composition analysis (SCA)** checks the libraries you depend on against vulnerability databases
 - Most of a modern application is other people's code
 - Also produces a software bill of materials (**SBOM**) in CycloneDX or SPDX format

TERMINAL

```
$ cat requirements.txt
requests==2.19.1
PyYAML==5.3
$ trivy fs -q --scanners vuln \
  --severity HIGH,CRITICAL --format json . \
  | jq -r '.Results[].Vulnerabilities[] |
  [.PkgName, .InstalledVersion, .VulnerabilityID,
  .Severity, .FixedVersion] | join(" ")'
PyYAML 5.3 CVE-2020-14343 CRITICAL 5.4
PyYAML 5.3 CVE-2020-1747 CRITICAL 5.3.1
requests 2.19.1 CVE-2018-18074 HIGH 2.20.0
```

- The fix column is the action: raise the pin to that version or later, then rerun

3.2 Pipeline Enhancements: Scanning, Containers, and AI (continued)

Container image and secret scanning

- **Container image scanning**: an image carries a whole OS userland, and every package in it can have CVEs
 - Scan after the build and before the push to the registry
 - Rebuild on a schedule; a clean image today has new CVEs next month
- `--exit-code 1` turns findings into a failing gate; `--severity` decides what counts

TERMINAL

```
$ trivy image -q --severity CRITICAL --exit-code 1 \
  --format json alpine:3.14.0 > scan.json
$ echo $?
1
$ jq -r '.Results[].Vulnerabilities[] |
  [.PkgName, .VulnerabilityID, .FixedVersion]
  | join(" ")' scan.json
apk-tools CVE-2021-36159 2.12.6-r0
libcrypto1.1 CVE-2021-3711 1.1.1l-r0
libssl1.1 CVE-2021-3711 1.1.1l-r0
zlib CVE-2022-37434 1.2.12-r2
```

- Fix by moving to a newer base image tag, not by patching inside a running container
- **Secret scanning** looks for credentials by known key formats and by entropy
 - In the pipeline as a backstop to the `pre-commit` hook (objective 1.2)
 - `gitleaks git` scans every commit in history; `gitleaks dir` scans files

TERMINAL

```
$ gitleaks dir --no-banner --no-color -v .
Finding:      AKIAQX4ZL2RBWN3T6YUM
RuleID:       aws-access-token
File:         deploy.env
Line:         1
9:11AM WRN leaks found: 1
```

Code quality scanning

- **Code quality scanning** measures maintainability, not security
 - Linters and formatters in check mode: `ruff check`, `yamllint`, `ansible-lint`, `terraform fmt -check`
 - Complexity (deeply nested functions), duplicated code, **test coverage**

- SonarQube (and SonarQube Cloud) collect these into one report per branch
 - A **quality profile** is the rule set for each language
 - A **quality gate** is the pass or fail condition, for example: coverage on new code at least 80%, no new critical issues, duplication under 3%
 - The pipeline step waits for the gate result and fails the job when the gate fails
- Gates usually judge new code only, so a legacy repository can adopt one without first fixing ten years of history
- GitLab shows Code Quality findings in the merge request; GitHub shows code scanning alerts on the pull request

Containerization in the pipeline

- Each job runs inside a container image named in the pipeline file
 - GitLab `image:`; GitHub `container:` on a job or `docker://` for a step; Jenkins `agent { docker { image '...' } }`
- Why containerize build steps
 - **Reproducible**: the same tool versions on every runner and on your laptop
 - **Isolated**: one job cannot leave files or packages behind for the next
 - No hand-installed tools on runners; changing a version is a one-line change reviewed in Git
- Pin the tag (`trivy:0.57.1`, not `latest`), or the digest (`@sha256:...`) for full immutability
- A scan job that runs the scanner in its own image

.GITLAB-CI.YML

```
stages: [build, scan]

container-scan:
  stage: scan
  image:
    name: aquasec/trivy:0.57.1
    entrypoint: [""]
  variables:
    IMAGE: $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
  script:
    - trivy image --exit-code 1
      --severity CRITICAL "$IMAGE"
```

- `entrypoint: [""]` clears the image's own entrypoint so the runner can start a shell
- Building images inside CI: Docker-in-Docker (a privileged `docker:dind` service), or daemonless and rootless builders such as Buildah or rootless BuildKit
- `services`: start helper containers (a database, a mock API) next to the job

3.2 Pipeline Enhancements: Scanning, Containers, and AI (continued)

AI-based log analysis

- A failed job can leave thousands of log lines; the real error is often one line near the middle
- Artificial intelligence (AI)-based log analysis** uses a language model or anomaly detection to
 - Summarize why the job failed and point to the lines that matter
 - Group repeated failures across runs (the same flaky test every Tuesday)
 - Suggest a likely fix: a missing dependency, a renamed variable, an expired token
- Built into platforms: GitLab Duo Root Cause Analysis on a failed job, Copilot explaining a failed GitHub Actions job; log platforms add anomaly detection
- Limits

CONCERN	WHAT TO DO
Confident but wrong answers	check the cited log line yourself
Long logs truncated	send the failing step, not the whole run
Secrets in logs	mask CI variables before logs leave
Different answer each run	treat it as a hint, not evidence

- Human in the loop**: the model suggests, an engineer decides
 - Never let a model's suggestion change production or retry a deploy by itself
 - Masked CI/CD variables (GitLab prints `[MASKED]`) and `secrets.*` (GitHub prints `***`) keep credentials out of the text being analyzed

ON THE EXAM

- Match the scan to the artifact: source code is SAST, dependencies are SCA, a built image is container scanning, keys in files or history are secret scanning, and a running app is DAST.
- A quality gate fails the pipeline when a threshold is not met (coverage, new critical issues). A "pin the toolchain so every runner behaves the same" scenario points to running the job in a container image.
- AI-based log analysis speeds up diagnosis, but its output is a suggestion that a person verifies; watch for answers that let it act on production unreviewed.

TRY IT

Install Bandit with `pip install bandit`, save `backup.py`, and run the scan. Change the call to `subprocess.run(["scp", f"admin@{host}:running-config", "."])` and run it again. `B602` disappears because there is no shell to inject into; Bandit still lists `B603` as a low-severity reminder to validate the input.

3.3 Webhooks, ChatOps, and Pipeline Triggers

3.0 CONTINUOUS INTEGRATION

Hooks and triggers

- A **pipeline trigger** is the event that starts a pipeline
 - Push, merge request, tag, schedule, a button, an API call, a chat command, another pipeline
- A **webhook** is an HTTP POST the Git platform sends to a URL you register when an event happens
 - Push-based: the CI server hears about the commit within seconds
 - The alternative is **polling** (Jenkins `pollSCM`): ask the repository every few minutes; slower and wasteful
- GitHub Actions and GitLab CI run inside the platform; external CI (Jenkins, a deploy tool, a chat bot) needs a real webhook
- Webhook request: headers name the event, the body is JSON describing it
 - GitHub: `X-GitHub-Event: push`, `X-GitHub-Delivery` (unique ID), `X-Hub-Signature-256`
 - GitLab: `X-Gitlab-Event: Push Hook`; `X-Gitlab-Token` (shared secret in plain text) or, from GitLab 19.0, an HMAC `webhook-signature`

3.3 Webhooks, ChatOps, and Pipeline Triggers (continued)

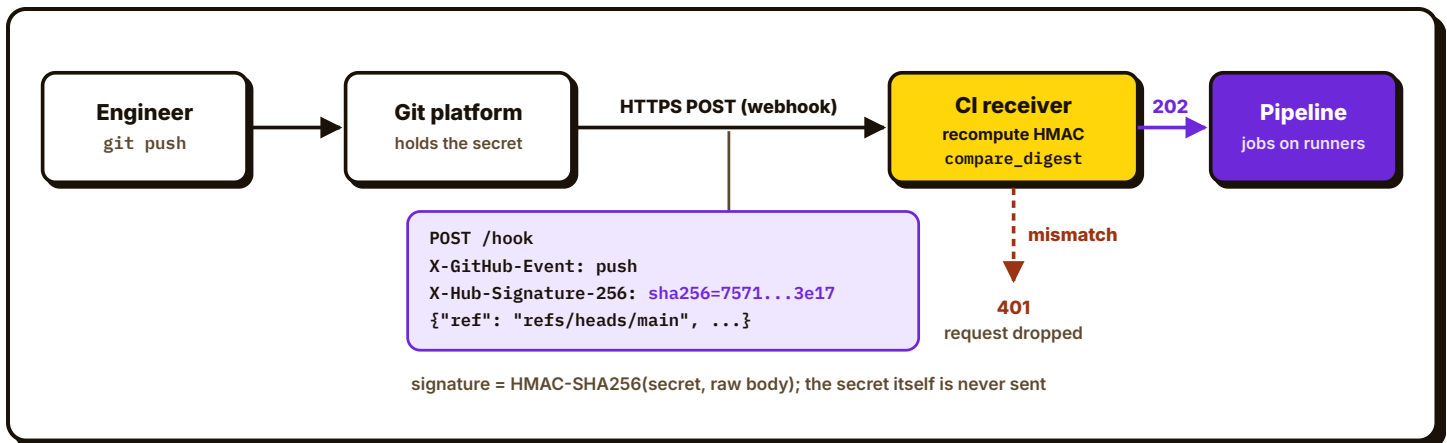


FIGURE The receiver recomputes the signature from the raw body and its own copy of the secret. A forged or altered request fails the comparison and nothing runs.

■ Verifying a signed webhook

- Anyone who finds the URL can POST to it, so the receiver must prove the request came from the platform
- GitHub signs the raw body with **HMAC-SHA256** using a secret you set on the webhook
 - Header: **X-Hub-Signature-256: sha256=<hex digest>**
 - The receiver computes the same HMAC and compares with a **constant-time comparison** (`hmac.compare_digest`) so timing does not leak the answer

HOOK.PY

```
import hashlib
import hmac
import os
from http.server import BaseHTTPRequestHandler
from http.server import HTTPServer

SECRET = os.environ["WEBHOOK_SECRET"].encode()

class Hook(BaseHTTPRequestHandler):
    def do_POST(self):
        size = int(self.headers["Content-Length"])
        body = self.rfile.read(size)
        mac = hmac.new(SECRET, body, hashlib.sha256)
        want = "sha256=" + mac.hexdigest()
        got = self.headers.get(
            "X-Hub-Signature-256", ""
        )
        ok = hmac.compare_digest(want, got)
        # 202: queue the pipeline; 401: drop it
        self.send_response(202 if ok else 401)
        self.end_headers()

server = HTTPServer(("127.0.0.1", 18400), Hook)
server.serve_forever()
```

TERMINAL

```
$ export WEBHOOK_SECRET='lab-only-secret'
$ python3 hook.py &
$ BODY='{ "ref": "refs/heads/main" }'
$ SIG=$(printf '%s' "$BODY" | openssl dgst \
    -sha256 -hmac "$WEBHOOK_SECRET" -r | cut -c1-64)
$ post() { curl -s -o /dev/null -w '%{http_code}\n' \
    -H "X-Hub-Signature-256: sha256=$SIG" \
    -d "$1" http://127.0.0.1:18400/; }
$ post "$BODY"
202
$ post '{ "ref": "refs/heads/evil" }'
401
```

- Same signature, one changed byte in the body: rejected
- Also: answer fast (queue the work, return **202**), use HTTPS, and ignore repeated delivery IDs

■ ChatOps

- **ChatOps (chat-based operations)** runs operations from a chat channel: a command in Slack, Teams, or Mattermost starts a pipeline job, and the result is posted back
 - The channel becomes a shared, searchable log of who ran what and what happened
 - Suited to routine, low-risk actions: show VLANs, check a deploy status
- GitLab ChatOps: `/gitlab netops/configs run show-vlans core-sw1`
 - Runs the named job from the default branch with `CI_PIPELINE_SOURCE` set to `chat`
 - Arguments arrive in `CHAT_INPUT`; also `CHAT_CHANNEL` and `CHAT_USER_ID`
 - Needs the Developer role or higher on the project
- Other platforms: a bot (Hubot, Errbot, a Slack app) that calls the CI API, often with an approval step

3.3 Webhooks, ChatOps, and Pipeline Triggers (continued)

- Treat chat commands like any other access: least privilege, audit log, no secrets pasted into the channel

.GITLAB-CI.YML (CHAT JOB)

```
show-vlans:
  rules:
    - if: $CI_PIPELINE_SOURCE == "chat"
  script:
    - echo "VLANs on $CHAT_INPUT are 10 20 30"
```

TERMINAL

```
$ gitlab-ci-local --variable CI_PIPELINE_SOURCE=chat \
  --variable CHAT_INPUT=core-sw1 show-vlans
show-vlans > VLANs on core-sw1 are 10 20 30
```

■ Push, manual, and scheduled triggers

- **Push trigger**: every push (or every push to listed branches, tags, or paths) starts the pipeline
- **Manual trigger**: a person starts it, often with inputs; used for production deploys and one-off jobs
- **Scheduled trigger**: a **cron** expression starts it at set times; nightly backups, drift checks, image rescans

TRIGGER	GITHUB ACTIONS	GITLAB CI	JENKINS
Push	on: push with branches, tags, paths	default; filter with rules	webhook, or pollSCM('H/5 * * * *')
Merge or pull request	on: pull_request	\$CI_PIPELINE_SOURCE == "merge_request_event"	multibranch CHANGE_ID builds
Manual	on: workflow_dispatch with inputs	when: manual job; Run pipeline button	Build with Parameters; input step pauses
Scheduled	on: schedule with cron (UTC)	pipeline schedule in the UI; \$CI_PIPELINE_SOURCE == "schedule"	triggers { cron('H 2 * * *') }

- Cron fields: minute hour day-of-month month day-of-week
 - 17 6 * * 1-5 : 06:17 Monday to Friday; */15 * * * *: every 15 minutes; 0 2 1 * * : 02:00 on the first of the month
 - Jenkins H picks a hashed minute so 50 jobs do not all start at :00
- GitHub schedules run on the default branch in UTC, no more often than every 5 minutes, and can start late when runners are busy

.GITHUB/WORKFLOWS/NETOPS.YML

```
name: netops
on:
  push:
    branches: [main]
    tags: ['v*']
  schedule:
    - cron: '17 6 * * 1-5'
  workflow_dispatch:
    inputs:
      target:
        type: choice
        options: [staging, production]
        default: staging
jobs:
  info:
    runs-on: ubuntu-latest
    steps:
      - run: |
          echo "event=${{ github.event_name }}"
          echo "ref_name=${{ github.ref_name }}"
          echo "ref_type=${{ github.ref_type }}"
          echo "sha=${{ github.sha }}"
          echo "target=${{ inputs.target }}"
```

- **actionlint** catches a bad cron before GitHub silently ignores the schedule

3.3 Webhooks, ChatOps, and Pipeline Triggers (continued)

TERMINAL

```
$ actionlint bad.yml
bad.yml:7:13: invalid CRON format "0 25 * * *" in
schedule event: end of range (25) above maximum
(23): 25 [events]
```

Git variables: tag, branch, commit

- Every run exposes **Git variables** naming the **ref** (branch or tag) and the commit that started it; jobs use these to name artifacts, tag images, and decide whether to deploy

VALUE	GITHUB ACTIONS	GITLAB CI	JENKINS (MULTIBRANCH)
Branch	<code>github.ref_name</code> when <code>github.ref_type</code> is <code>branch</code>	<code>CI_COMMIT_BRANCH</code> (not set in tag or merge request pipelines)	<code>BRANCH_NAME</code>
Tag	<code>github.ref_name</code> when <code>github.ref_type</code> is <code>tag</code>	<code>CI_COMMIT_TAG</code> (set only in tag pipelines)	<code>TAG_NAME</code> (tag builds only)
Commit	<code>github.sha</code> / <code>\$GITHUB_SHA</code>	<code>CI_COMMIT_SHA</code> , <code>CI_COMMIT_SHORT_SHA</code>	<code>GIT_COMMIT</code> (Git plugin)
Branch or tag	<code>github.ref</code> (<code>refs/heads/main</code>)	<code>CI_COMMIT_REF_NAME</code>	<code>BRANCH_NAME</code> (holds the tag name in a tag build)

- The workflow above, run with `act` for a push of tag `v1.4.0` and for a manual run

TERMINAL

```
$ act push -e tag.json --json \
  | jq -j 'select(.raw_output) | .msg'
event=push
ref_name=v1.4.0
ref_type=tag
sha=5772eca0b2b1196ba48b82018561690b38f5ca84
target=
$ act workflow_dispatch -e dispatch.json --json \
  | jq -j 'select(.raw_output) | .msg'
event=workflow_dispatch
ref_name=main
ref_type=branch
target=production
```

- GitLab, a job that echoes `CI_PIPELINE_SOURCE` , `CI_COMMIT_BRANCH` , `CI_COMMIT_TAG` , and `CI_COMMIT_SHA` , run for a push to `main` and for tag `v1.4.0`

TERMINAL

```
$ gitlab-ci-local vars
vars > source=push
vars > branch=main
vars > tag=
vars > sha=448f0309d1aeb724b1da80315c08703adfd23bc4
$ gitlab-ci-local --variable CI_COMMIT_TAG=v1.4.0 \
  --unset-variable CI_COMMIT_BRANCH vars
vars > branch=
vars > tag=v1.4.0
```

- Jenkins multibranch build of tag `v1.4.0` : the `Release` stage runs `sh 'echo "release $TAG_NAME"'` and prints `release v1.4.0` ; the `main` build prints the full `GIT_COMMIT`
- A rule that tests `CI_COMMIT_BRANCH == "main"` never matches in a tag pipeline; release jobs test `CI_COMMIT_TAG` instead

ON THE EXAM

- "Notify the external build server immediately when code is pushed" is a webhook; "check the repository every five minutes" is polling. A webhook receiver validates `X-Hub-Signature-256` (HMAC-SHA256 of the body with a shared secret).
- Know the trigger keywords: `on: push` , `workflow_dispatch` and `when: manual` (manual), `schedule` with cron (scheduled), and ChatOps for commands typed in chat.
- Read the variable for the ref type: a job that should run only for releases tests the tag (`CI_COMMIT_TAG` , `github.ref_type == 'tag'` , `TAG_NAME`), and it will be empty on a branch build.

■ Pipeline as code

- **Pipeline as code**: a **pipeline definition** is a file in the repository that describes the whole pipeline
 - Versioned with the code it builds; changed through a pull request; rolled back with `git revert`
 - The CI system reads the file from the commit being built, so each branch can carry its own pipeline changes
- Three formats to read on the exam: a **Jenkinsfile** (Groovy-based DSL), **.gitlab-ci.yml** (YAML), and a GitHub Actions workflow (YAML in `.github/workflows/`)
- Below, the same pipeline in each format
 - Build: render a switch config into `dist/`
 - Test: check the rendered file
 - Deploy: only on `main`; Release: only for a tag
 - A 10-minute time limit on every format

■ Jenkinsfile

- **Declarative pipeline**: a fixed structure inside `pipeline { }`, checked before it runs
- **Scripted pipeline**: plain Groovy inside `node { }`; flexible, but errors appear only at run time

JENKINSFILE

```

pipeline {
    agent any
    options { timeout(time: 10, unit: 'MINUTES') }
    stages {
        stage('Build') {
            steps {
                sh 'mkdir -p dist'
                sh 'echo "vlan 10" > dist/core-sw1.cfg'
            }
        }
        stage('Test') {
            steps {
                sh 'grep -q "vlan 10" dist/core-sw1.cfg'
            }
        }
        stage('Deploy') {
            when { branch 'main' }
            steps {
                sh 'echo "deploy $GIT_COMMIT"'
            }
        }
        stage('Release') {
            when { buildingTag() }
            steps {
                sh 'echo "release $TAG_NAME"'
            }
        }
    }
    post {
        failure { echo 'notify #netops' }
    }
}

```

- Anatomy
 - **agent**: where it runs (`any`, a `label`, or a `docker` image)
 - **stages** / **stage** / **steps**: the work, in order; stages share one workspace, so `dist/` is still there for Test
 - **when**: the condition for a stage; **options**: timeout, retry, build retention
 - **post**: **always**, **success**, **failure** blocks after the run; also **environment**, **parameters**, **triggers**
- A **multibranch pipeline** job finds every branch and tag with a **Jenkinsfile** and sets **BRANCH_NAME**, **TAG_NAME**, and **CHANGE_ID**
- Validate before committing with the declarative linter (also `jenkins-cli declarative-linter`)

3.3 Pipeline Definitions: Jenkins, GitLab, and GitHub (continued)

TERMINAL

```
$ J=http://127.0.0.1:18410
$ curl -s -X POST -F "jenkinsfile=<Jenkinsfile" \
  $J/pipeline-model-converter/validate
Jenkinsfile successfully validated.
```

- Leave the `steps { }` wrapper out of the Test stage and the same call fails with: Expected one of "steps", "stages", or "parallel" for stage "Test"
- The multibranch run for `main`, trimmed to the last two stages

JENKINS CONSOLE, NETOPS/MAIN

```
[Pipeline] { (Deploy)
deploy 9a2956fa7d0012ddb1bb04952d79bfe2ac36f42d
[Pipeline] { (Release)
Stage "Release" skipped due to when conditional
Finished: SUCCESS
```

■ .gitlab-ci.yml

- One file at the repository root; every top-level key that is not a reserved **keyword** is a **job**

.GITLAB-CI.YML

```
stages: [build, test, deploy]

default:
  image: alpine:3.20
  timeout: 10m

build:
  stage: build
  script:
    - mkdir -p dist
    - echo "vlan 10" > dist/core-sw1.cfg
  artifacts:
    paths: [dist/]

test:
  stage: test
  script:
    - grep -q "vlan 10" dist/core-sw1.cfg

deploy:
  stage: deploy
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
  script:
    - echo "deploy $CI_COMMIT_SHA"

release:
  stage: deploy
  rules:
    - if: $CI_COMMIT_TAG
  script:
    - echo "release $CI_COMMIT_TAG"
```

- Anatomy
 - `stages`: the order; `default`: settings every job inherits
 - Each job: `stage`, `script` (required except in trigger jobs), `image`, `rules`, `needs`, `variables`
 - `artifacts`: files handed to later jobs; each job starts in a fresh container, so without this `test` has no `dist/`
 - `include`: pulls in shared templates from other files or projects
- Validate with the pipeline editor or CI Lint in GitLab; run locally with `gitlab-ci-local`
 - `--force-shell-executor` runs each job in your shell instead of pulling its `image`; `--shell-isolation` gives each job its own copy, so artifacts really move

TERMINAL

```
$ gitlab-ci-local --force-shell-executor \
  --shell-isolation
build    exported artifacts in 118 ms
test     imported artifacts in 80 ms
deploy   imported artifacts in 87 ms
deploy   > deploy
e897e39de70c708577d13b96b5bb20e38dbfc169
PASS    build
PASS    test
PASS    deploy
```

■ GitHub Actions workflow

- Any number of YAML files in `.github/workflows/`; the objectives call this one `github-actions.yml`
- A **workflow** file: `on:` lists the triggers; `jobs:` run in parallel unless `needs:` orders them

3.3 Pipeline Definitions: Jenkins, GitLab, and GitHub (continued)

.GITHUB/WORKFLOWS/GITHUB-ACTIONS.YML

```
name: netops-ci
on:
  push:
    branches: [main]
    tags: ['v*']
    pull_request:

jobs:
  build:
    runs-on: ubuntu-latest
    timeout-minutes: 10
    steps:
      - uses: actions/checkout@v7
      - name: Build
        run: |
          mkdir -p dist
          echo "vlan 10" > dist/core-sw1.cfg
      - name: Test
        run: grep -q "vlan 10" dist/core-sw1.cfg

  deploy:
    needs: build
    if: github.ref == 'refs/heads/main'
    runs-on: ubuntu-latest
    steps:
      - run: echo "deploy $GITHUB_SHA"

  release:
    needs: build
    if: github.ref_type == 'tag'
    runs-on: ubuntu-latest
    steps:
      - run: echo "release $GITHUB_REF_NAME"
```

- Anatomy
 - **runs-on**: the runner label; **steps**: either **uses:** (a reusable **action**) or **run:** (shell)
 - Build and Test are steps in one job, so they share a workspace; separate jobs would need **actions/upload-artifact** and **download-artifact**
 - Pass event data through environment variables (**\$GITHUB_REF_NAME**) rather than pasting **{{ }}** expressions into scripts; a branch name is user input
 - **permissions**: limits what the job's **GITHUB_TOKEN** can do
- Validate with **actionlint**; run locally with **act** (it maps **ubuntu-latest** to a Docker image)

TERMINAL

```
$ actionlint && echo "actionlint: no problems"
actionlint: no problems
$ act push -e tag.json --json > run.json
$ jq -r 'select(.jobResult)'
| "\(.jobID) \(.jobResult)" run.json
build success
release success
```

	JENKINSFILE	.GITLAB-CI.YML	GITHUB ACTIONS
Language	Groovy-based DSL	YAML	YAML
Location	repository root (any path in job config)	repository root	.github/workflows/*.yaml
Unit of work	stage with steps	job (top-level key)	job with steps
Ordering	stages in sequence; parallel block	stages plus needs	parallel unless needs
Condition	when { branch 'main' }	rules: - if:	if: expression
Files between steps	shared workspace	artifacts	upload / download artifact actions
Runs on	agent	runner, image , tags	runs-on , container
After failure	post { failure { } }	when: on_failure	if: failure()
Check it locally	declarative linter	CI Lint, gitlab-ci-local	actionlint , act

3.3 Pipeline Definitions: Jenkins, GitLab, and GitHub (continued)

ON THE EXAM

- Identify the platform from the file: `pipeline { agent ... stages { stage(...) { steps ... } } }` is a declarative Jenkinsfile; top-level job names with `script:` and `stages:` is `.gitlab-ci.yml`; `on:`, `jobs:`, `runs-on:`, and `steps:` with `uses:` / `run:` is a GitHub Actions workflow.
- Scenario questions often hinge on one line: a job that skips on a tag (`if: $CI_COMMIT_BRANCH == "main"`), a missing `artifacts` entry so the next job cannot find the file, or `needs` that forces an order.
- A declarative Jenkinsfile must have `agent` and `stages`, and every `stage` needs `steps` (or nested `stages` / `parallel`).

04

Continuous Delivery

20% of the exam

-
- 4.1** Given a scenario, implement techniques of continuous delivery (CD)
 - 4.2** Explain concepts related to application service levels
 - 4.3** Compare and contrast different methods to secure a connection to providers
-

From pipeline to production

- **Continuous delivery** (CD): every change that passes the pipeline is ready to release
 - A person approves the production step; **continuous deployment** removes that approval
- A **delivery method** decides how the new version replaces the old one
 - How many users see it first, whether anything goes down, how fast you can undo it
- Four methods to know: in-place, rolling, blue-green, canary
 - All four can be automated: AWS CodeDeploy, Kubernetes, Argo Rollouts, Ansible, a load balancer API

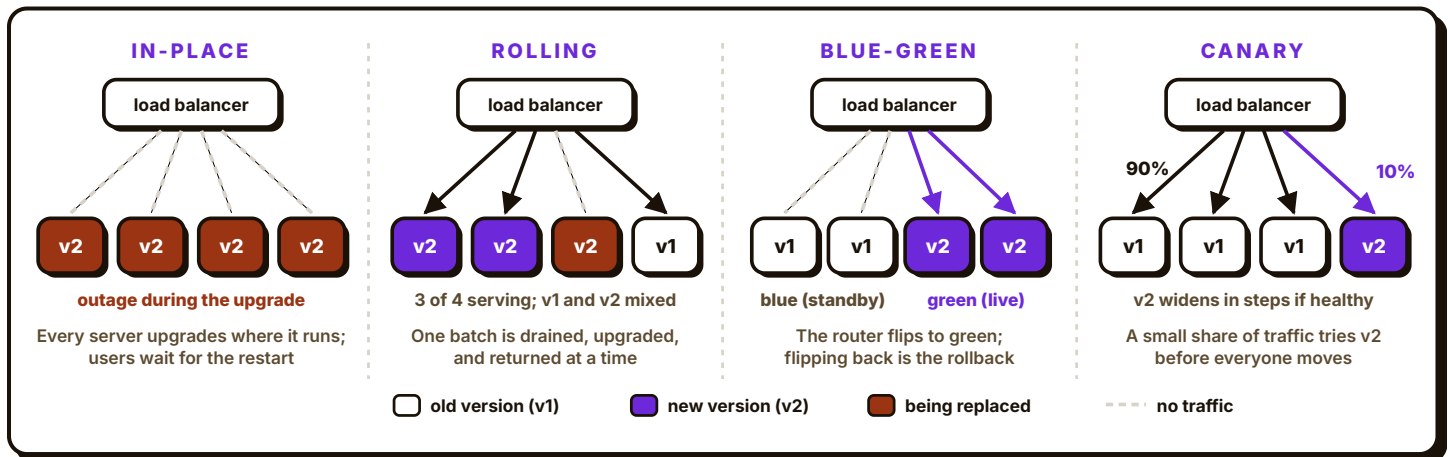


FIGURE The same four-server service under each delivery method. Purple boxes run the new version; rust boxes are being replaced.

In-place and rolling

- **In-place** deployment: stop the app on the existing servers, install the new version, start it again
 - No new infrastructure; the cheapest method
 - Users see an outage while the service restarts
 - Rollback means running the old deployment again, which takes as long as the bad one did
 - Kubernetes calls this the **Recreate** strategy
- **Rolling** deployment: replace instances in batches
 - The load balancer drains a batch, the batch upgrades, passes its health check, and rejoins
 - No full outage, but capacity drops by one batch at a time
 - Old and new versions serve traffic together until the last batch finishes, so both must work with the same database schema and API
 - Rollback is another rolling deployment, in the other direction

- Kubernetes **Deployment** objects roll by default
 - **maxUnavailable**: how many pods may be down during the update
 - **maxSurge**: how many extra pods may be created above **replicas**
 - Both default to 25%
 - A **readinessProbe** keeps traffic away from a pod until it answers

DEPLOY.YAML (EXCERPT)

```
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1 # at most 1 pod down
      maxSurge: 1 # at most 1 extra pod
```

Blue-green

- Two identical production environments
 - **Blue** serves users; **green** sits idle with the new version installed
- Test green with production configuration while nobody uses it

4.1 Deployment Strategies (continued)

- Cut over by repointing the router: load balancer target group, reverse proxy upstream, or DNS record
- **Rollback** is the same switch in reverse, in seconds
 - Keep blue running until green has proved itself
- Cost: double the capacity while both run
- Shared database is the hard part
 - Schema changes must work for blue and green until blue is retired
- DNS cutovers are slowed by caching; clients keep the old address until the TTL expires

SWITCH.SH

```
#!/bin/sh
# Usage: ./switch.sh blue/green
# $PROXY is the proxy container's name
set -e
cat > conf.d/upstream.conf <<CONF
upstream app {
    server $1:80;
}
CONF
docker exec "$PROXY" nginx -t -q
docker exec "$PROXY" nginx -s reload
```

- The demo: an nginx proxy on port 18450 in front of two app containers that answer **v1 blue** and **v2 green**
 - **traffic.sh** sends a request every 10 ms and prints the body and HTTP status
- **nginx -s reload** starts new workers and lets old ones finish their requests, so no request fails

TERMINAL

```
$ ./traffic.sh 300 > log.txt &
$ ./switch.sh green
$ wait; sort log.txt | uniq -c
    209 v1 blue 200
     91 v2 green 200
$ ./switch.sh blue
$ curl -s http://127.0.0.1:18450/
v1 blue
```

■ Canary

- Send a small share of real traffic to the new version first
 - Named for the canary that warned coal miners of bad air
- Watch the canary's error rate and latency against the old version's
 - Healthy: raise the share in steps (10%, 25%, 50%, 100%)
 - Unhealthy: set its share to zero; only that slice of users ever saw the problem
- Needs good monitoring and automation to judge each step
 - Argo Rollouts and Flagger automate the analysis in Kubernetes
 - AWS CodeDeploy has predefined canary and linear traffic-shifting configurations for Lambda and ECS
- Split by weight, or by user group (employees first, one region first)
- A **feature flag** hides a feature inside code that is already deployed; a canary limits who reaches the new deployment

CONF.D/UPSTREAM.CONF

```
upstream app {
    zone app 64k;
    server blue:80 weight=9;
    server green:80 weight=1;
}
```

TERMINAL

```
$ ./traffic.sh 100 | sort | uniq -c
    90 v1 blue 200
    10 v2 green 200
```

■ Choosing a method

METHOD	DOWNTIME	ROLLBACK	EXTRA COST
In-place	yes, while the app restarts	slow: redeploy the old version	none
Rolling	none; capacity dips one batch at a time	slow: roll back batch by batch	little (surge instances)
Blue-green	none; one switch moves everyone	fast: switch back	a full second environment
Canary	none; a small share sees v2 first	fast: send the share back to v1	little (the canary instances)

- Mixed versions in production at once: rolling and canary
 - Design changes so both versions can run side by side
- Smallest blast radius: canary
- Fastest, simplest rollback: blue-green
- Lowest cost and simplest tooling: in-place, if a maintenance window is acceptable

4.1 Deployment Strategies (continued)

TRY IT

Build the blue-green demo: two `nginx:1.27-alpine` containers on one Docker network, each with a `return 200 "v1 blue";` style location, and a third nginx with the upstream file above. Run `switch.sh` while a curl loop is running and confirm every line of the log ends in `200`.

ON THE EXAM

- "Release to 5% of users and expand if metrics stay healthy" is canary. "Two identical environments and a router switch" is blue-green.
- "Instant rollback" points to blue-green; "lowest cost, downtime acceptable" points to in-place.
- Rolling and canary both run two versions at once; a question about schema or API compatibility during deployment is pointing at them.

4.1 QA Testing, Validation, and Hotfixes

4.0 CONTINUOUS DELIVERY

Quality assurance in the delivery pipeline

- **Quality assurance (QA) testing** proves a build is fit to release before users depend on it
- Each kind of test is a **gate**: a failure stops the pipeline
 - Cheap, fast tests run first on every commit; slow, expensive ones run later on fewer builds
- After the deploy, **validation** confirms the release works where it runs; **remediation** is what you do when it does not

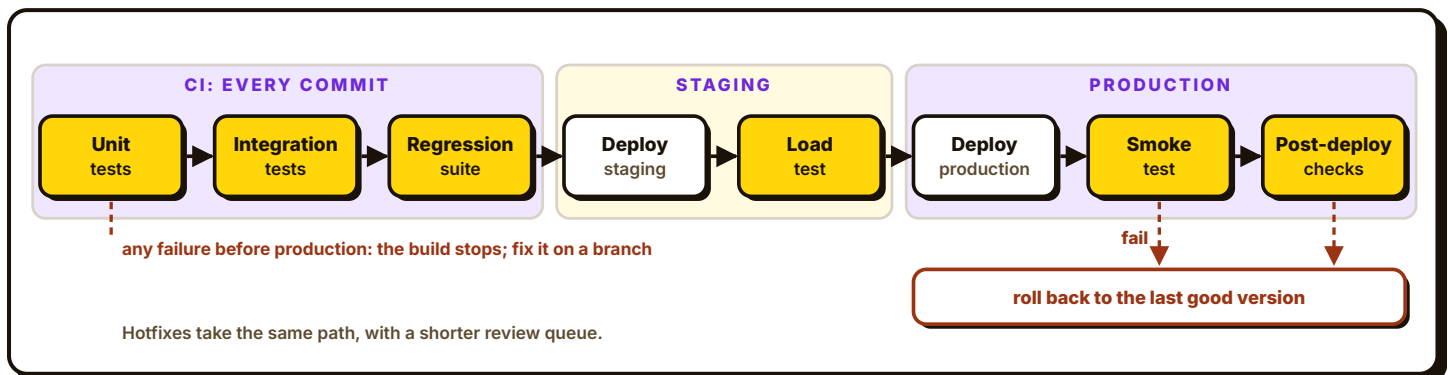


FIGURE Where each check runs. Every yellow box can stop the pipeline; checks after the production deploy trigger a rollback instead.

TEST	QUESTION IT ANSWERS
Unit	does this function work alone?
Integration	do the parts work together?
Regression	did an old bug come back?
Load	does it hold up at expected traffic?
Smoke	is the new deploy alive at all?
Post-deployment	is production healthy now?

Integration and regression testing

- **Integration testing** runs components together with their real dependencies
 - The app against a real PostgreSQL, Redis, or message queue instead of a mock
 - CI starts the dependencies as containers: GitLab `services:`, GitHub Actions service containers, `docker compose up`
 - An Ansible role tested against a throwaway VM or container (Molecule)
- **Regression testing** reruns the existing suite after every change
 - When a bug is fixed, a test that reproduces it joins the suite
 - The bug can never return quietly: if it does, that test fails

4.1 QA Testing, Validation, and Hotfixes (continued)

TEST_VLANS.PY

```
from vlans import parse_vlans

def test_single():
    assert parse_vlans("10") == [10]

def test_range():
    assert parse_vlans("20-22") == [20, 21, 22]

# NET-311: a trailing comma crashed the parser
def test_trailing_comma():
    assert parse_vlans("10,20,") == [10, 20]
```

BEFORE AND AFTER THE FIX

```
$ pytest -q --tb=no -rN
..F [100%]
1 failed, 2 passed in 0.01s
$ pytest -q --tb=no -rN
... [100%]
3 passed in 0.01s
```

Load testing

- **Load testing** sends a realistic volume of requests and measures the response
 - Throughput (requests per second), **latency percentiles** (p95, p99), error rate
 - Averages hide the slow tail; the p99 is what your unluckiest users feel
- Related tests
 - Stress: push past expected load to find the breaking point
 - Soak: hold normal load for hours to expose memory leaks
 - Spike: jump from idle to peak at once
- Run against staging that matches production size, never against production without a plan
- Tools: **ab** (ApacheBench), **hey**, **k6**, JMeter, Locust
 - k6 **thresholds** such as **p(95)<200** fail the run, which fails the pipeline stage
- In the **ab** output below, the last block is the latency percentiles in milliseconds

AB AGAINST THE STAGING CONTAINER

```
$ ab -n 2000 -c 20 http://127.0.0.1:18451/
Concurrency Level:      20
Time taken for tests:    0.476 seconds
Complete requests:      2000
Failed requests:         0
Requests per second:    4201.19 [#/sec] (mean)
Time per request:       4.761 [ms] (mean)
    50%      4
    90%      8
    95%     11
    99%     18
   100%     34 (longest request)
```

Smoke tests and post-deployment testing

- **Smoke testing**: a short check run right after a deploy
 - Does the service answer, is the right version live, is it fast enough?
 - Seconds, not minutes; it does not prove every feature
- The exit code is the gate: **0** passes, anything else fails the pipeline step

SMOKE.SH

```
#!/bin/sh
# Usage: ./smoke.sh BASE_URL EXPECTED_VERSION
base=$1 want=$2 rc=0
ok() { echo "PASS $1"; }
bad() { echo "FAIL $1"; rc=1; }

code=$(curl -s -o /dev/null -m 5 \
  -w '%{http_code}' "$base/healthz")
if [ "$code" = 200 ]; then ok "healthz 200"
else bad "healthz returned $code"; fi

ver=$(curl -s -m 5 "$base/version")
if [ "$ver" = "$want" ]; then ok "version $ver"
else bad "version $ver, want $want"; fi

t=$(curl -s -o /dev/null -m 5 \
  -w '%{time_total}' "$base/")
if awk "BEGIN{exit !($t < 0.5)}"; then
  ok "home page in ${t}s"
else bad "home page took ${t}s"; fi
exit $rc
```

4.1 QA Testing, Validation, and Hotfixes (continued)

A GOOD DEPLOY, THEN A BROKEN ONE

```
$ ./smoke.sh http://127.0.0.1:18451 2.1.1
PASS healthz 200
PASS version 2.1.1
PASS home page in 0.000952s
$ echo $?
0
$ ./smoke.sh http://127.0.0.1:18452 2.1.1
FAIL healthz returned 503
FAIL version 2.1.0, want 2.1.1
PASS home page in 0.001103s
$ echo $?
1
```

- **Post-deployment testing** keeps checking production after the smoke test passes
 - Synthetic monitoring: scripted user journeys every minute from several regions
 - Compare error rate and latency with the previous version
 - A failed check triggers **automatic rollback** or pages the on-call engineer

.GITHUB/WORKFLOWS/DEPLOY.YML (ACTIONLINT: OK)

```
steps:
  - uses: actions/checkout@v6
  - name: Deploy to the idle color
    run: ./switch.sh green
  - name: Smoke test (gate)
    run: ./smoke.sh "$APP_URL" "${TAG#v}"
    env:
      APP_URL: https://app.example.com
      TAG: ${github.ref_name}
  - name: Roll back on failure
    if: failure()
    run: ./switch.sh blue
```

Hotfixes

- A **hotfix** is an urgent fix to the version running in production
 - Outage, security hole, data corruption: it cannot wait for the next planned release

- Branch from the release that is live, not from **main**
 - **main** already holds unreleased features you do not want to ship tonight
 - This is the release branch model from objective 1.2: the fix lands on **release/2.1** and ships as patch **v2.1.1**
- Same pipeline, shorter queue
 - Expedited review, but never skip the tests and smoke gate; a rushed fix is how one outage becomes two
- Carry the fix back to **main**, or the next release brings the bug back

TERMINAL

```
$ git switch -c hotfix/NET-207 release/2.1
Switched to a new branch 'hotfix/NET-207'
$ sed -i 's/ttl 3$/ttl 300/' zone.conf
$ git commit -qam "fix(dns): restore TTL"
$ git switch -q release/2.1
$ git merge --no-ff --no-edit hotfix/NET-207
Merge made by the 'ort' strategy.
zone.conf | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
$ git tag -a v2.1.1 -m "Hotfix: zone TTL"
$ git switch -q main
$ git cherry-pick -x hotfix/NET-207
[main f0b3af7] fix(dns): restore TTL
1 file changed, 1 insertion(+), 1 deletion(-)
```

- Remediation options, fastest first: roll back, turn off a feature flag, ship a hotfix
 - Roll back first if you can; fix forward when rollback is impossible (a database migration already ran)

ON THE EXAM

- "Verify the critical functions right after deployment" is a smoke test. "A fixed bug reappeared" calls for regression testing. "Will it handle Black Friday traffic" is load testing.
- Integration testing checks components together; a distractor will describe unit testing of one function in isolation.
- A hotfix branches from the production release, not from **main**, and must be merged back.

Uptime and availability

- Uptime**: the share of a time window when the service works for its users
 - Written as a percentage, often in "nines": 99.9% is three nines
- Availability** = uptime / (uptime + downtime)
- Each extra nine cuts the allowed downtime by a factor of ten
 - And usually multiplies the cost: redundancy, automation, on-call staff
- The table below assumes a 30-day month and a 365-day year

UPTIME	PER DAY	PER MONTH	PER YEAR
99%	14m 24s	7h 12m	3d 15h 36m
99.5%	7m 12s	3h 36m	1d 19h 48m
99.9%	1m 26s	43m 12s	8h 45m 36s
99.95%	43s	21m 36s	4h 22m 48s
99.99%	9s	4m 19s	52m 34s
99.999%	1s	26s	5m 15s

- Components in a chain multiply
 - An app at 99.9% that needs a database at 99.9%:
 $99.9\% \times 99.9\% = 99.8\%$
 - Two redundant copies, either of which can serve: $1 - (0.001 \times 0.001) = 99.9999\%$

SLIs, SLOs, and SLAs

- Service-level indicator (SLI)**: the measurement
 - Good events divided by valid events: successful requests / all requests, requests under 300 ms / all requests
- Service-level objective (SLO)**: the internal target for an SLI over a window
 - "99.9% of checkout requests succeed, measured over a rolling 30 days"
 - Set by engineering and product; missing it changes priorities, not invoices
- Service-level agreement (SLA)**: a contract with customers that names an availability level and a penalty
 - Usually **service credits**: a percentage of the monthly bill refunded
 - Defines what counts: scheduled maintenance and customer-caused outages are usually excluded
- Keep the SLO stricter than the SLA
 - The gap is a warning zone: you miss your own target before you owe anyone money

TERM	WHAT IT IS	EXAMPLE	WHO HOLDS YOU TO IT
SLI	a measured ratio	999,412 of 1,000,000 requests returned 2xx	dashboards and alerts
SLO	target for the SLI	99.9% success over 30 days	your own team
SLA	contract with consequences	99.5% monthly or a 10% credit	customers and lawyers

Error budgets

- Error budget** = 1 - SLO
 - A 99.9% SLO over 5 million requests allows 5,000 failed requests
 - Over a 30-day month as time: 43 minutes 12 seconds
- Budget left: ship features, take deployment risks
- Budget spent: freeze risky releases and spend the time on reliability
 - Agree on this policy before the outage, not during it

BUDGET.PY

```
slo = 0.999 # 99.9% of requests
good, total = 4_996_210, 5_000_000

sli = good / total
allowed = (1 - slo) * total # failures allowed
used = (total - good) / allowed
print(f"SLI {sli:.4%}")
print(f"SLO met {sli >= slo}")
print(f"budget {allowed:,.0f} bad requests")
print(f"budget used {used:.1%}")
```

4.2 Service Levels: Uptime, SLOs, SLAs, MTTR, MTBF (continued)

TERMINAL

```
$ python3 budget.py
SLI          99.9242%
SLO met      True
budget       5,000 bad requests
budget used   75.8%
```

■ MTTR and MTBF

- **Mean time to repair (MTTR)**: average time from failure until service is restored
 - MTTR = total downtime / number of failures
 - Includes detecting, diagnosing, fixing, and verifying; mean time to detect (MTTD) is often tracked on its own
 - The R is also read as recover, restore, or resolve; check which one a report means
 - Lower it with good alerts, runbooks, automated rollback, and practiced on-call staff

- **Mean time between failures (MTBF)**: average working time between one failure and the next
 - $MTBF = \text{total uptime} / \text{number of failures}$
 - Higher is better; raise it with testing, redundancy, and removing known causes
- Mean time to failure (MTTF) is for parts that are replaced, not repaired: disks, fans, optics
- Tie them together: $\text{availability} = MTBF / (MTBF + MTTR)$
 - Cutting MTTR in half helps as much as doubling MTBF

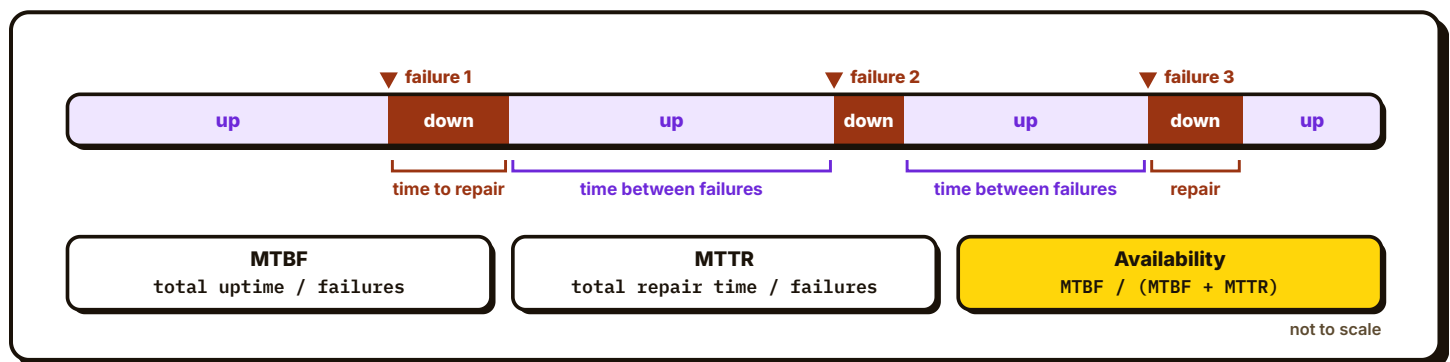


FIGURE MTBF averages the purple stretches between failures; MTTR averages the rust stretches spent restoring service.

MTBF.PY

```
window_h = 30 * 24          # 720 hours
outages_min = [42, 18, 30]  # three incidents
n = len(outages_min)

mttr_h = sum(outages_min) / n / 60
uptime_h = window_h - sum(outages_min) / 60
mtbf_h = uptime_h / n
avail = mtbf_h / (mtbf_h + mttr_h)
print(f"MTTR      {mttr_h * 60:.0f} min")
print(f"MTBF      {mtbf_h:.1f} h")
print(f"availability {avail:.3%}")
print(f"check       {uptime_h / window_h:.3%}")
```

- Worked by hand: 90 minutes down over 3 failures is a 30-minute MTTR
 - 720 h minus 1.5 h down = 718.5 h up; $718.5 / 3 = 239.5$ h MTBF
 - $239.5 / (239.5 + 0.5) = 99.792\%$, the same as $718.5 / 720$
- 99.792% misses a 99.9% SLO: the error budget was 43.2 minutes and the month used 90

■ Feedback loops

- A **feedback loop** carries what production teaches back into development

TERMINAL

```
$ python3 mtbf.py
MTTR      30 min
MTBF      239.5 h
availability 99.792%
check     99.792%
```

4.2 Service Levels: Uptime, SLOs, SLAs, MTTR, MTBF (continued)

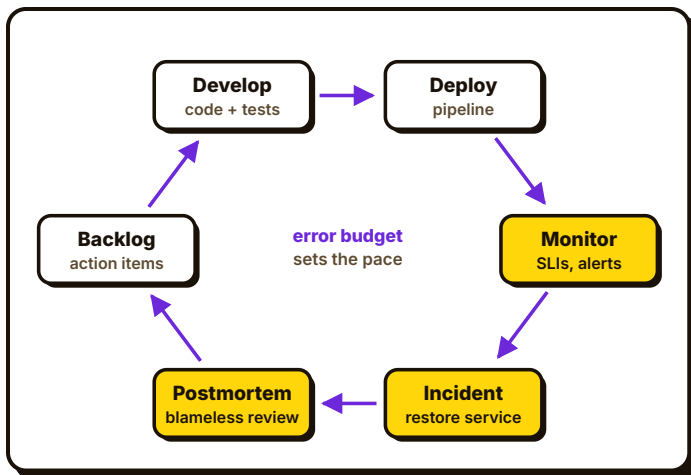


FIGURE Production data flows back into the next round of work. Yellow steps are where the loop learns something.

- Error budget status sets the release pace for the next window
 - Delivery metrics close the loop on the pipeline itself
 - The DORA metrics: deployment frequency, lead time for changes, change failure rate, and time to recover from a failed deployment
 - User signals count too: support tickets, feature usage, app store reviews
- Monitoring and alerts measure the SLIs and burn rate
 - Incidents end in a blameless postmortem
 - Action items go into the backlog with an owner and a due date
 - The bug that caused it becomes a regression test

ON THE EXAM

- SLA is the external contract with penalties; SLO is the internal target; SLI is the measurement. A distractor will swap SLA and SLO.
- MTTR measures how fast you recover, MTBF how often you fail. "Reduce the time to restore service" means lowering MTTR.
- Expect to convert a percentage into allowed downtime: 99.9% is about 43 minutes per month and 8.76 hours per year.

4.3 Securing Connections to Providers

4.0 CONTINUOUS DELIVERY

■ CLI configuration

- Automation reaches a cloud provider through its **command-line interface (CLI)**, a **software development kit (SDK)**, or the raw API
 - Same question for all three: where does the credential come from, how long does it live, what may it do?
 - Worst: long-lived keys in scripts. Best: short-lived credentials issued to a **machine identity**
- **aws configure** writes two plain-text files
 - `~/.aws/credentials`: secrets, one section per profile
 - `~/.aws/config`: region, output, roles, SSO settings
 - Section names differ: `[lab]` in credentials, `[profile lab]` in config
- Pick a profile with `--profile lab` or `AWS_PROFILE=lab`
- Environment variables: `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_SESSION_TOKEN`, `AWS_REGION`
 - `AWS_CONFIG_FILE` and `AWS_SHARED_CREDENTIALS_FILE` move the files, which keeps tests and CI isolated

`~/.AWS/CONFIG`

```
[default]
region = us-east-1

[profile lab]
region = us-west-2

[profile deployer]
role_arn = arn:aws:iam::111122223333:role/deploy
source_profile = lab
```

`~/.AWS/CREDENTIALS (DUMMY KEYS)`

```
[default]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = dummy-secret-1

[lab]
aws_access_key_id = AKIAI44QH8DHBEXAMPLE
aws_secret_access_key = dummy-secret-2
```

4.3 Securing Connections to Providers (continued)

- **Precedence**, highest first: command-line options, environment variables, role and SSO settings in the profile, `~/.aws/credentials`, `~/.aws/config`, container role, EC2 instance profile
 - `aws configure list` shows which source won for each value
 - For people, `aws configure sso` plus `aws sso login` gives short-lived credentials and leaves no secret on disk

TERMINAL

```
$ AWS_REGION=eu-west-1 aws configure list --profile lab
NAME      : VALUE                                : TYPE      : LOCATION
profile   : lab                                      : manual    : --profile
access_key : *****MPLE                          : shared-credentials-file :
secret_key : *****et-2                          : shared-credentials-file :
region    : eu-west-1                             : env       : ['AWS_REGION', 'AWS_DEFAULT_REGION']
```

TASK	AWS CLI	AZURE CLI	GOOGLE CLOUD CLI
Person signs in	<code>aws sso login</code>	<code>az login</code>	<code>gcloud auth login</code>
Settings folder	<code>~/.aws/</code> (<code>AWS_CONFIG_FILE</code>)	<code>~/.azure/</code> (<code>AZURE_CONFIG_DIR</code>)	<code>~/.config/gcloud/</code> (<code>CLOUDSDK_CONFIG</code>)
Named settings	profiles: <code>--profile</code> , <code>AWS_PROFILE</code>	subscriptions: <code>az account set --subscription</code>	configurations: <code>gcloud config configurations activate</code>
Machine signs in	instance profile or role	<code>az login --identity</code> or <code>--service-principal</code>	attached service account, <code>gcloud auth activate-service-account</code>
SDK default chain	env, profiles, container or instance role	<code>DefaultAzureCredential</code> : env, workload identity, managed identity, CLI sign-in	Application Default Credentials: <code>GOOGLE_APPLICATION_CREDENTIALS</code> , gcloud ADC file, metadata server

■ SDK credential chains

- An SDK walks a **credential provider chain** and uses the first source that answers
 - Same code on a laptop (profile), in CI (federated token), and on a server (instance role)
 - Never put keys in the client constructor; they end up in Git
- boto3 checks code arguments, then environment variables, then profiles, then container and instance roles
 - An explicit `profile_name` beats keys in the environment; `AWS_PROFILE` does not

CHAIN.PY

```
# chain.py: which provider supplied credentials?
import boto3

for profile in (None, "lab", "deployer"):
    s = boto3.Session(profile_name=profile)
    c = s.get_credentials()
    print(f"{profile or '(none)':9} {c.method}")
```

OFFLINE RUN WITH DUMMY KEYS

```
$ python3 chain.py
(none)    shared-credentials-file
lab       shared-credentials-file
deployer  assume-role
$ export AWS_ACCESS_KEY_ID=AKIAI44QH8DHBEXAMPLE
$ export AWS_SECRET_ACCESS_KEY=dummy-secret-3
$ python3 chain.py
(none)    env
lab       shared-credentials-file
deployer  assume-role
```

4.3 Securing Connections to Providers (continued)

- `assume-role` is lazy: boto3 calls STS only when the keys are first used

■ IAM and least privilege

- **Identity and Access Management (IAM)**: who may call which API on which resource, under which conditions
- **Least privilege**: only the actions and resources the job needs
 - A job that reads switch backups needs `s3:GetObject` on one prefix, not `s3:*` on `*`
 - One identity per job, so one leaked credential opens one door
- Lint policies in the pipeline: `parliament` flags `s3:*` on `*`, and an object action aimed at a bucket ARN

S3-READ.JSON

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "ReadBackups", "Effect": "Allow",
      "Action": "s3:GetObject",
      "Resource": "arn:aws:s3:::net-backups/cfg/*",
    },
    {
      "Sid": "ListThatPrefix", "Effect": "Allow",
      "Action": "s3:ListBucket",
      "Resource": "arn:aws:s3:::net-backups",
      "Condition": { "StringLike": {
        "s3:prefix": "cfg/*"
      } }
    }
  ]
}
```

PARLIAMENT (IAM POLICY LINTER)

```
$ parliament --minimal < too-broad.json
RESOURCE_STAR
$ parliament --minimal < wrong-arn.json
RESOURCE_MISMATCH
$ parliament --minimal < s3-read.json; echo $?
0
```

■ Machine identities and federation

- A **machine identity** belongs to a workload, not a person
 - The platform issues short-lived credentials and rotates them; nothing is stored on disk

PLATFORM	MACHINE IDENTITY
AWS EC2	instance profile (an IAM role) via IMDSv2
AWS ECS, EKS	task role; EKS Pod Identity or IRSA
Azure	managed identity, system- or user-assigned
Google Cloud	attached service account
CI, other clouds	workload identity federation

- Federation: the outside workload presents an **OpenID Connect (OIDC)** token instead of a stored key
 - AWS: IAM OIDC provider plus a role trust policy;
 - Azure: federated credential; Google Cloud: workload identity pool

DEPLOY.YML (ACTIONLINT: OK)

```
permissions:
  id-token: write # lets the job request a JWT
  contents: read
jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production
    steps:
      - uses: aws-actions/configure-aws-credentials@v6
        with:
          role-to-assume: ${vars.DEPLOY_ROLE_ARN}
          aws-region: us-east-1
      - run: aws sts get-caller-identity
```

- The role's **trust policy** allows `sts:AssumeRoleWithWebIdentity` only when the token's claims match
 - `aud` (audience): `sts.amazonaws.com`
 - `sub` (subject): `repo:cpcc-netops/site:environment:production`
 - Leave out the `sub` check and any GitHub repository could assume the role

4.3 Securing Connections to Providers (continued)

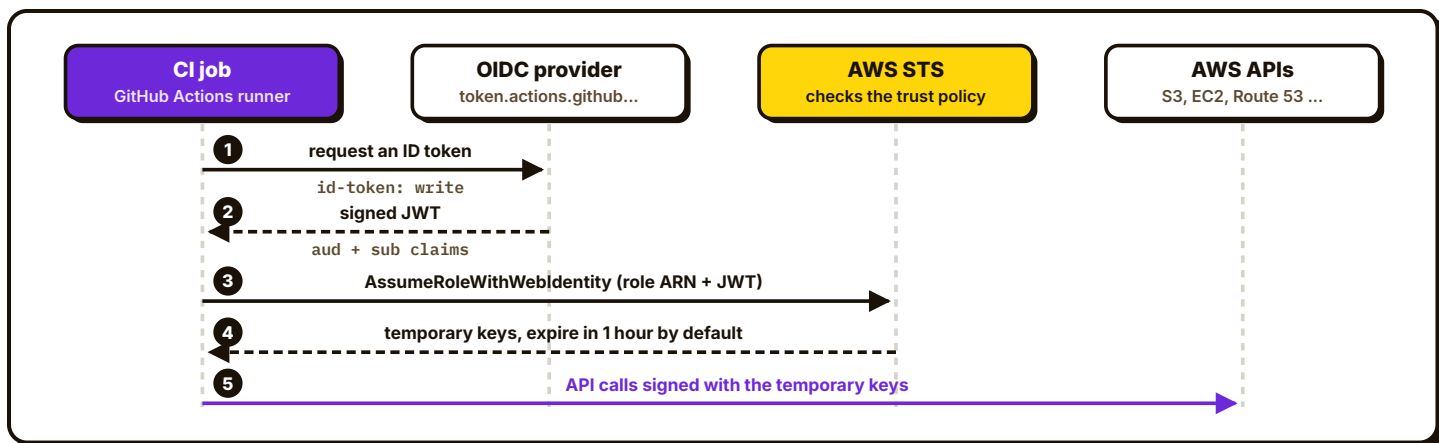


FIGURE GitHub Actions to AWS with no stored secret. The trust policy decides which repository, branch, or environment may assume the role.

■ Accessing external APIs

	API KEY	OAUTH 2.0 CLIENT CREDENTIALS
Sent with each call	the same static key, in a header such as <code>X-API-Key</code>	a short-lived access token in <code>Authorization: Bearer</code>
How it is obtained	copied from the provider's console	client ID and secret posted to the token endpoint with <code>grant_type=client_credentials</code>
Lifetime and scope	until revoked; often all-or-nothing	minutes to an hour; scopes requested per token

- Keep either in a secrets manager (Vault, AWS Secrets Manager, Azure Key Vault), never in the repository
- **Key rotation**: replace credentials on a schedule, and at once after any exposure
 - An AWS IAM user may hold two access keys, so rotation has no gap: create the new key, deploy it, deactivate the old one, then delete it
 - `GetAccessKeyLastUsed` finds keys nobody uses any more
 - Machine identities and short-lived tokens remove most rotation work

ON THE EXAM

- "No long-lived credentials in the pipeline" points to OIDC federation (workload identity) or a machine identity, not a better-protected access key.
- CLI precedence: command-line options beat environment variables, which beat the shared files; instance and container roles are the fallback.
- An API key is static until revoked; OAuth client credentials trade a client secret for short-lived, scoped tokens.