

COMPTIA EXPANSION SERIES • EXAM AT0-001 V1

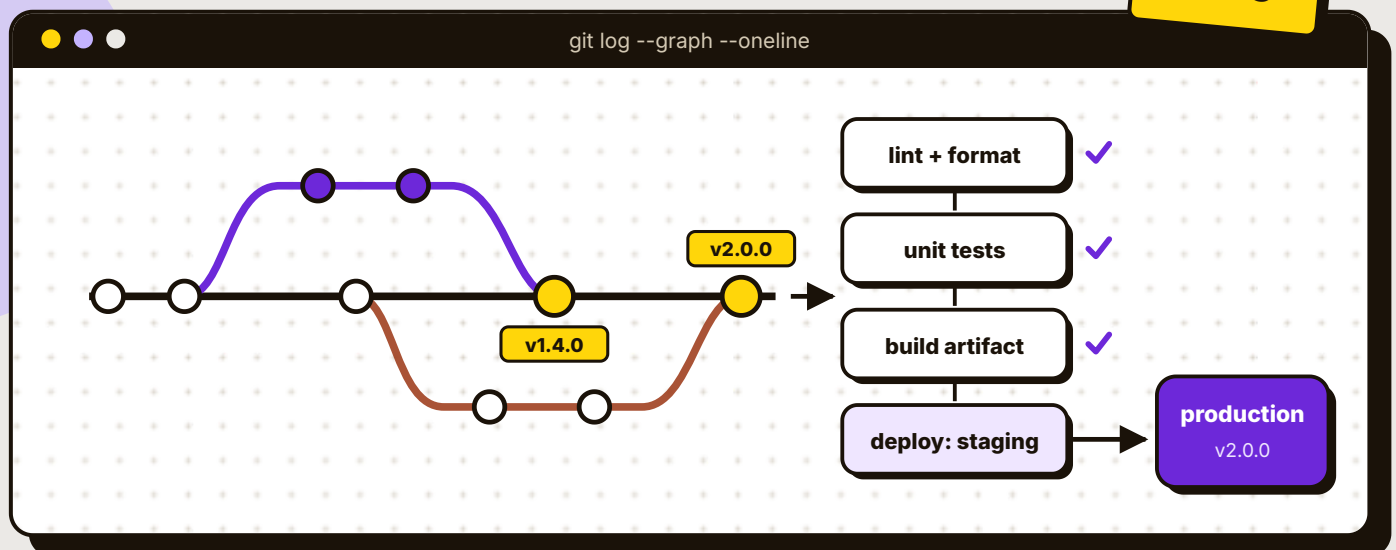
AutoOps+

Course Notes

Domain 1.0: Automation Coding Concepts

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

VERSION
1.0.0



CompTIA AutoOps+ AT0-001 Course Notes: Domain 1.0: Automation Coding Concepts

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

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

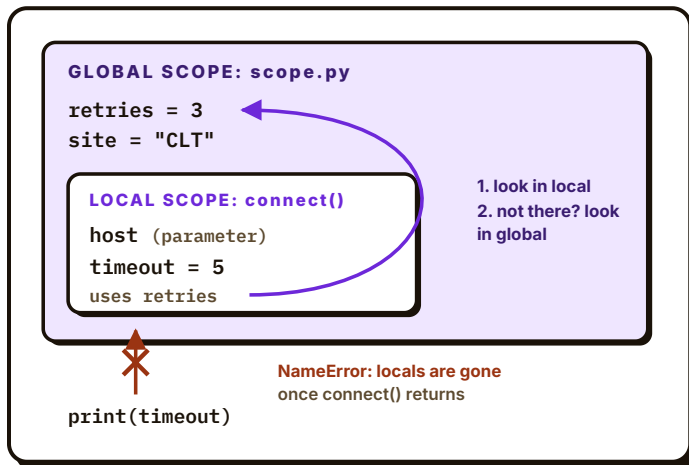


FIGURE A function sees its own locals first, then globals. Nothing outside the function can see its locals.

- 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
vlan      ['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 <code>_</code> ; 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)

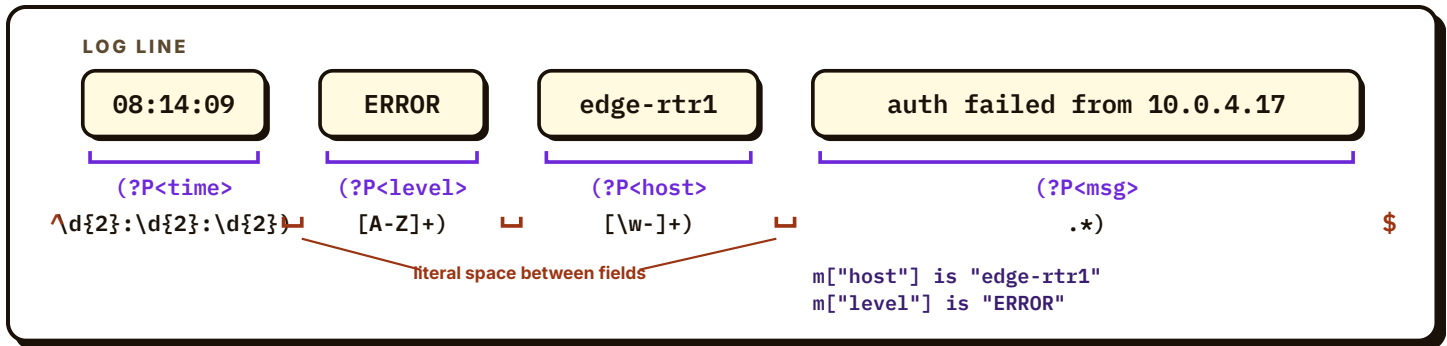


FIGURE Each named group consumes one field. The spaces between groups are literal and anchor the next field.

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)

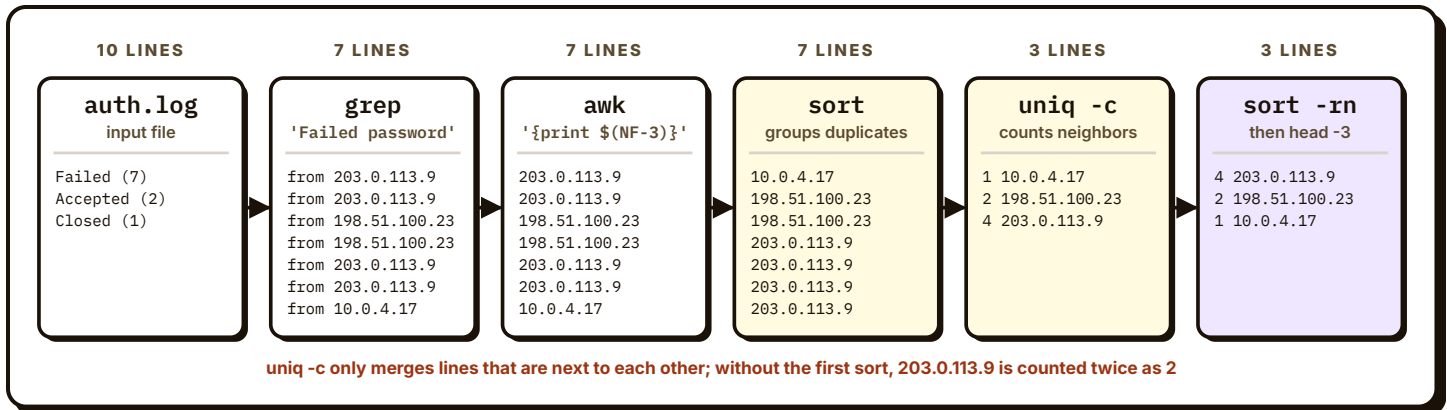


FIGURE Each stage reads the previous stage's output. The line counts on top show the data shrinking from raw log to answer.

- `$(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)

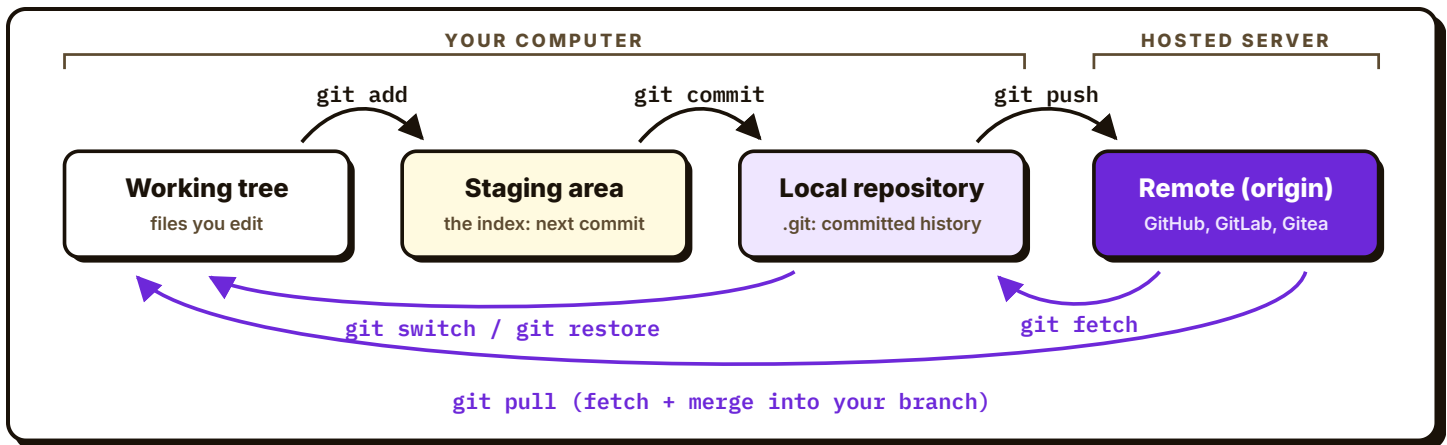


FIGURE Each command moves changes between two areas. Arrows on top send work toward the team; arrows below bring work back.

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

```
$ 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

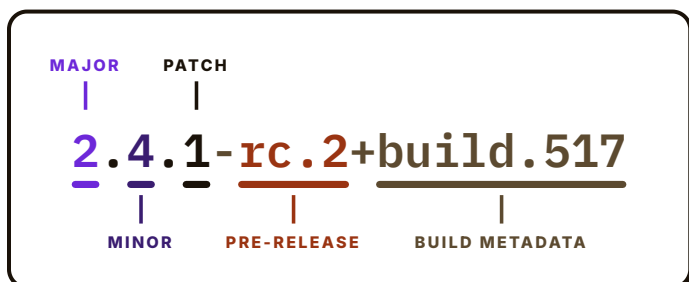


FIGURE Every part has a job. Only MAJOR, MINOR, PATCH, and the pre-release label affect which version is "newer."

- **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.

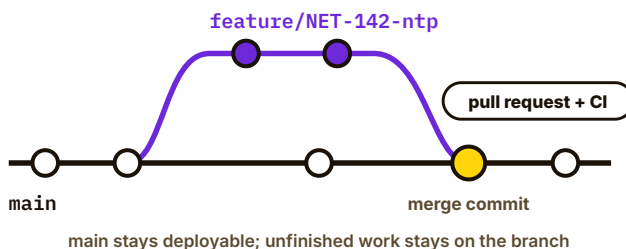
1.2 Branching Strategies

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

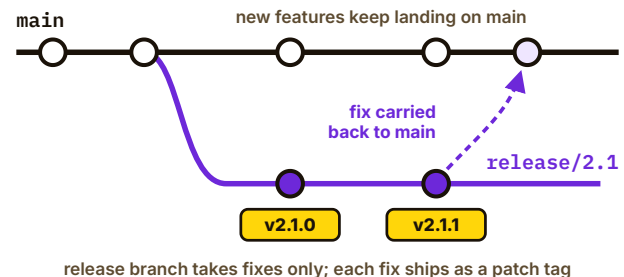


FIGURE Feature branches carry new work into main. Release branches carry a finished version out of main and accept only fixes.

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

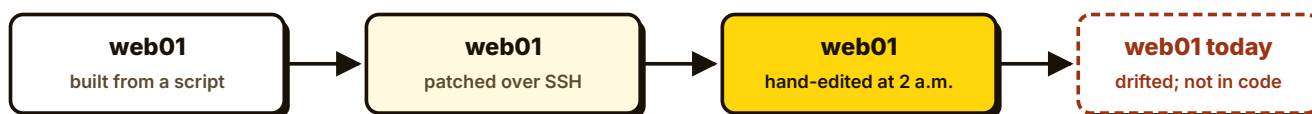
ROLE FOLDER	HOLDS
tasks/	the steps (<code>main.yml</code> runs first)
defaults/	inputs callers may override
vars/	values callers should not change
templates/	Jinja2 files for the <code>template</code> module
files/	static files to copy
handlers/	actions on change, such as a service restart
meta/	dependencies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

MUTABLE: CHANGE THE SAME SERVER IN PLACE



IMMUTABLE: NEVER EDIT, REPLACE FROM A NEW IMAGE

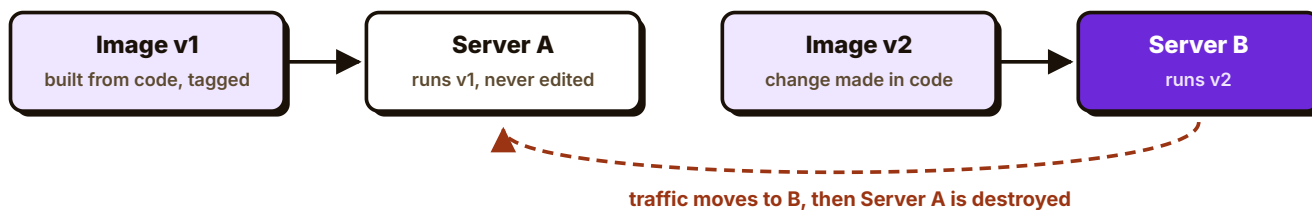


FIGURE A mutable server collects changes nobody recorded. An immutable server is thrown away and replaced by one built from the next versioned image.

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

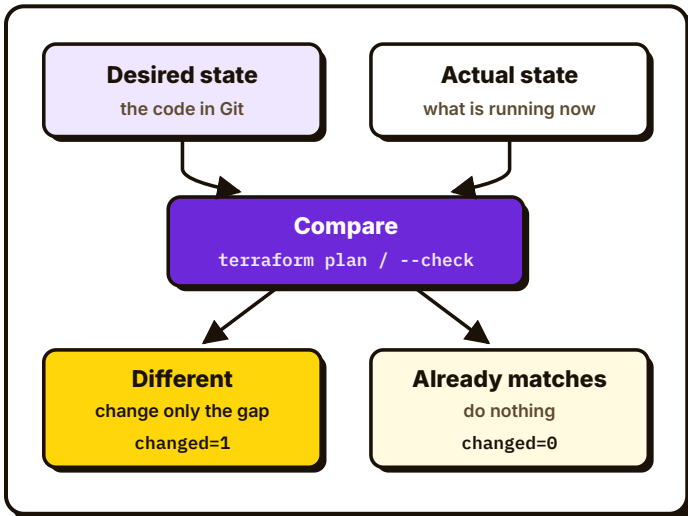


FIGURE Declarative tools compare before they act. When nothing differs, the run changes nothing.

■ 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

TERMINAL

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

- `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

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

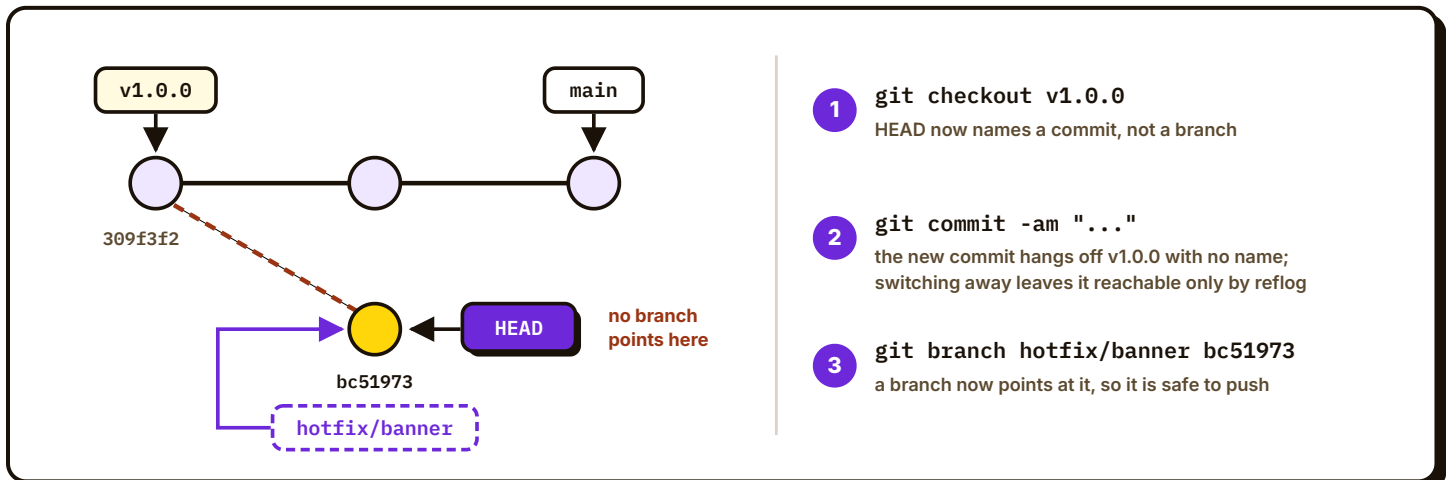


FIGURE A commit made with HEAD detached is attached to nothing. Give it a branch before you switch away, or recover it from the reflog afterwards.

- 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.