

COMPTIA EXPANSION SERIES • EXAM AT0-001 V1

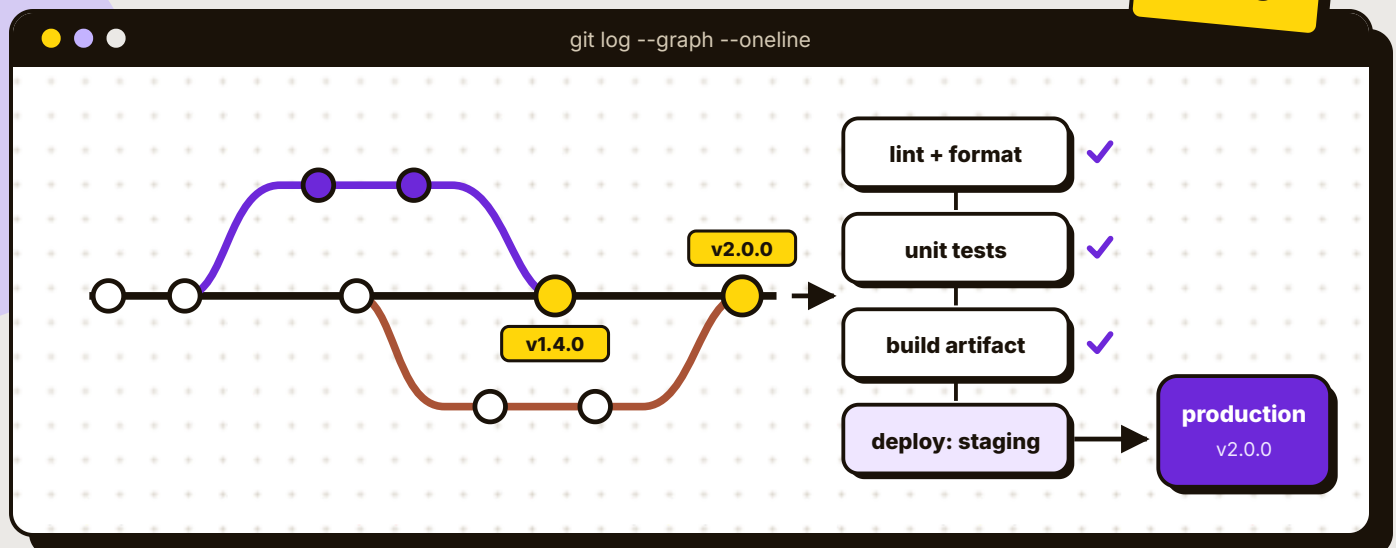
AutoOps+

Course Notes

Domain 3.0: Continuous Integration

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 3.0: Continuous Integration

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

3.0

Continuous Integration 24% of exam · page 1

3.1

Secrets Management

2

3.1

Artifact Management and SBOMs

4

3.1

Task Runners and Environment Provisioning

7

3.2

Pipeline Orchestration

9

3.2

Pipeline Enhancements: Scanning, Containers, and AI

12

3.3

Webhooks, ChatOps, and Pipeline Triggers

15

3.3

Pipeline Definitions: Jenkins, GitLab, and GitHub

19

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.

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

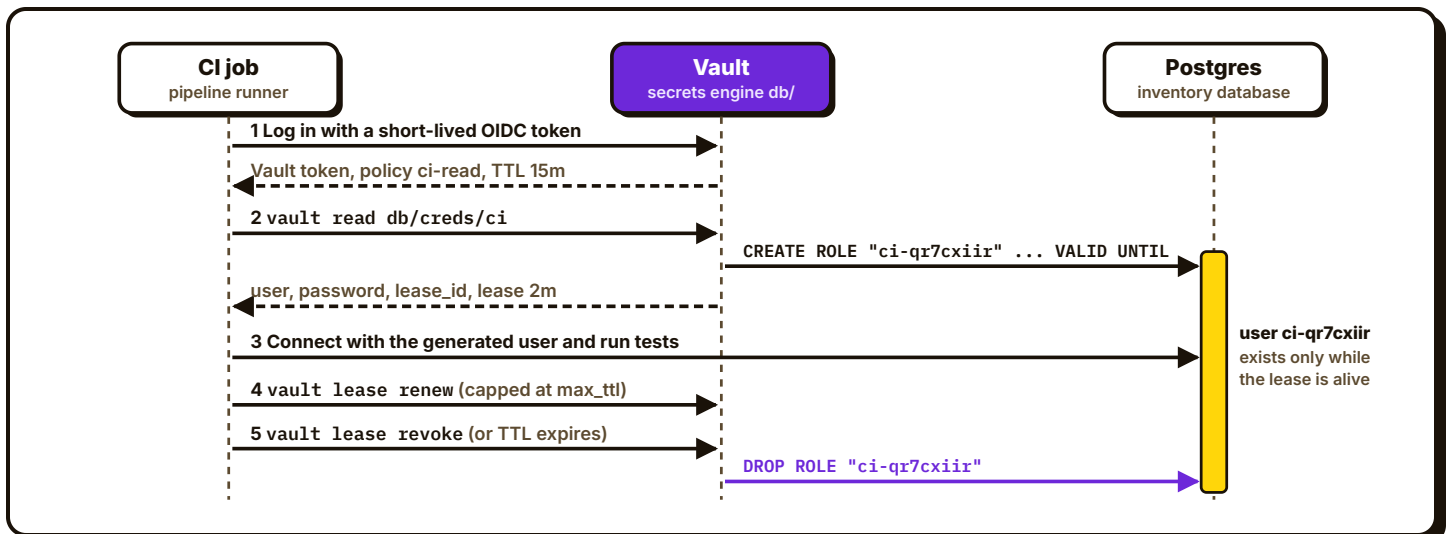


FIGURE A dynamic database secret. The database user exists only while its lease is alive; revoking the lease, or letting it expire, makes Vault drop the user.

- **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
```

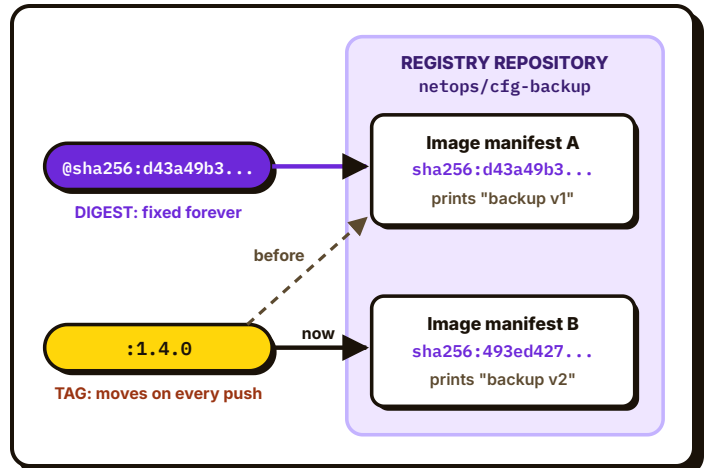


FIGURE Re-pushing 1.4.0 moved the tag to a new image. The digest still names the original, which the registry kept.

- **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 sdx-json=sbom.sdx.json \
  -o cyclonedx-json=sbom.cdx.json
$ jq -r .sdxVersion sbom.sdx.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

- 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
```

TERMINAL

```
$ 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
```

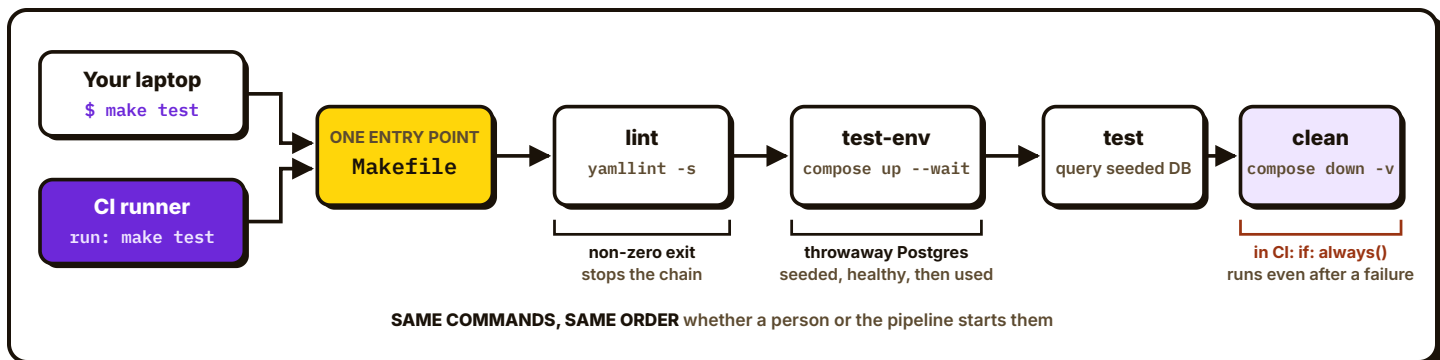


FIGURE `make test` runs lint, test-env, and test in order on any machine. `make clean` is a separate step; in CI, `if: always()` runs it even when a test failed.

- 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

■ 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

TERMINAL

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

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"]
```

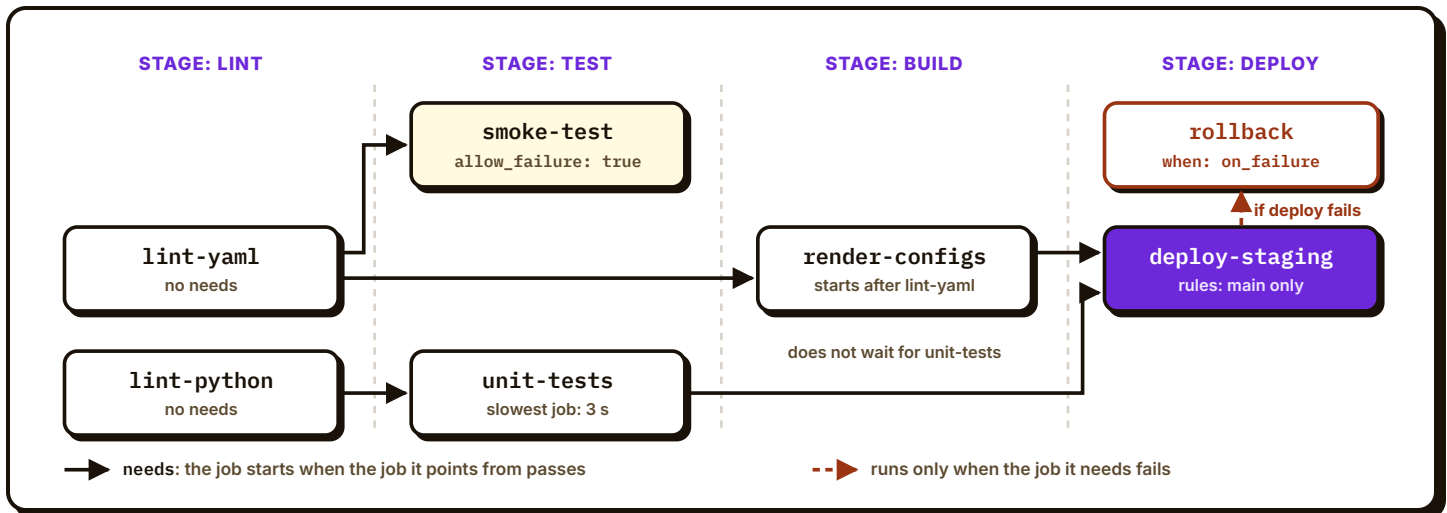


FIGURE The needs graph for this pipeline. `render-configs` is in a later stage than `unit-tests`, yet it starts as soon as `lint-yaml` passes.

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

```
$ 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
```

TERMINAL

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

```
$ git switch -c feature/NET-200-acl
$ 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
```

TERMINAL

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() }`

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)

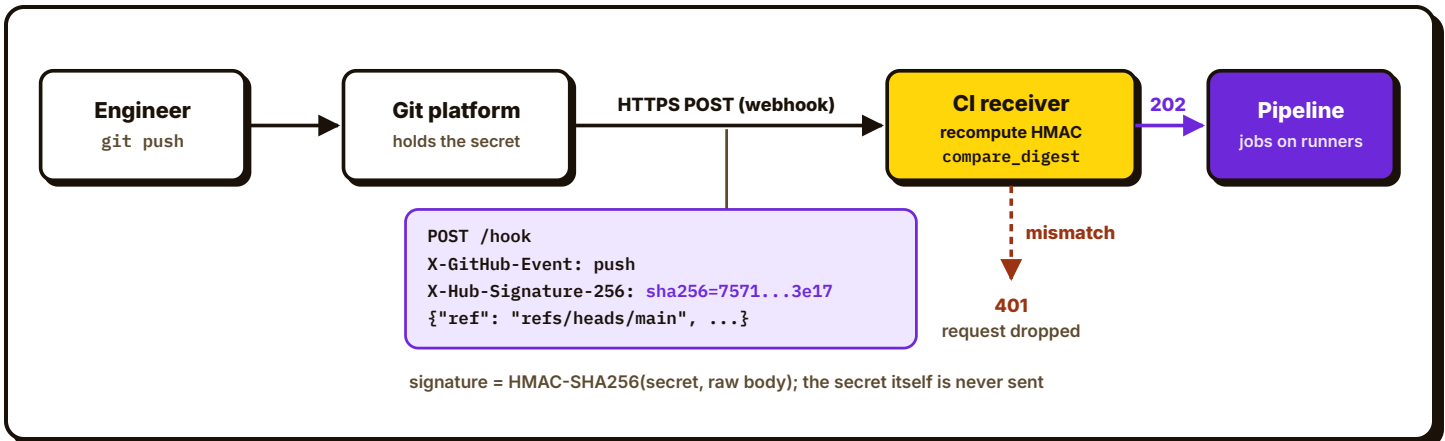


FIGURE The receiver recomputes the signature from the raw body and its own copy of the secret. A forged or altered request fails the comparison and nothing runs.

■ 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/*.yml
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`).