

COMPTIA EXPANSION SERIES • EXAM AT0-001 V1

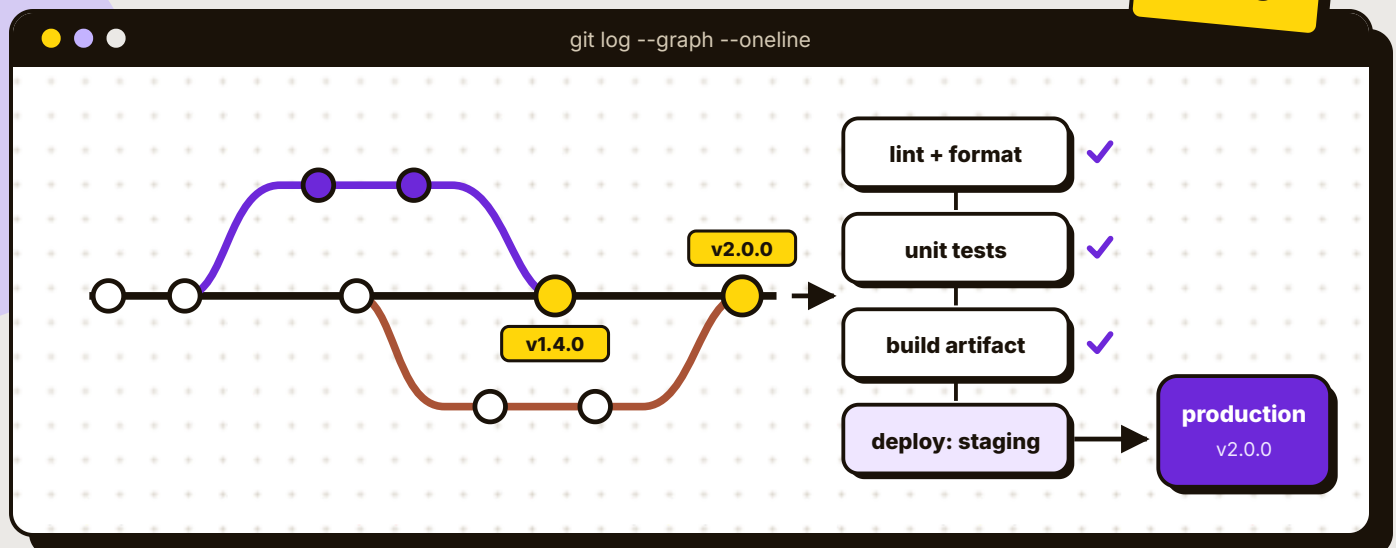
# AutoOps+

# Course Notes

## Domain 4.0: Continuous Delivery

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 4.0: Continuous Delivery**

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

|            |                                                |                      |          |
|------------|------------------------------------------------|----------------------|----------|
| <b>4.0</b> | <b>Continuous Delivery</b>                     | 20% of exam · page 1 |          |
| <b>4.1</b> | Deployment Strategies                          |                      | <b>2</b> |
| <b>4.1</b> | QA Testing, Validation, and Hotfixes           |                      | <b>4</b> |
| <b>4.2</b> | Service Levels: Uptime, SLOs, SLAs, MTTR, MTBF |                      | <b>7</b> |
| <b>4.3</b> | Securing Connections to Providers              |                      | <b>9</b> |

# 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 | <div><div></div></div> | 31% |
| 2.0 System Configuration       | <div><div></div></div> | 25% |
| 3.0 Continuous Integration     | <div><div></div></div> | 24% |
| 4.0 Continuous Delivery        | <div><div></div></div> | 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.

# 04

# Continuous Delivery

**20%** of the exam

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

### From pipeline to production

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

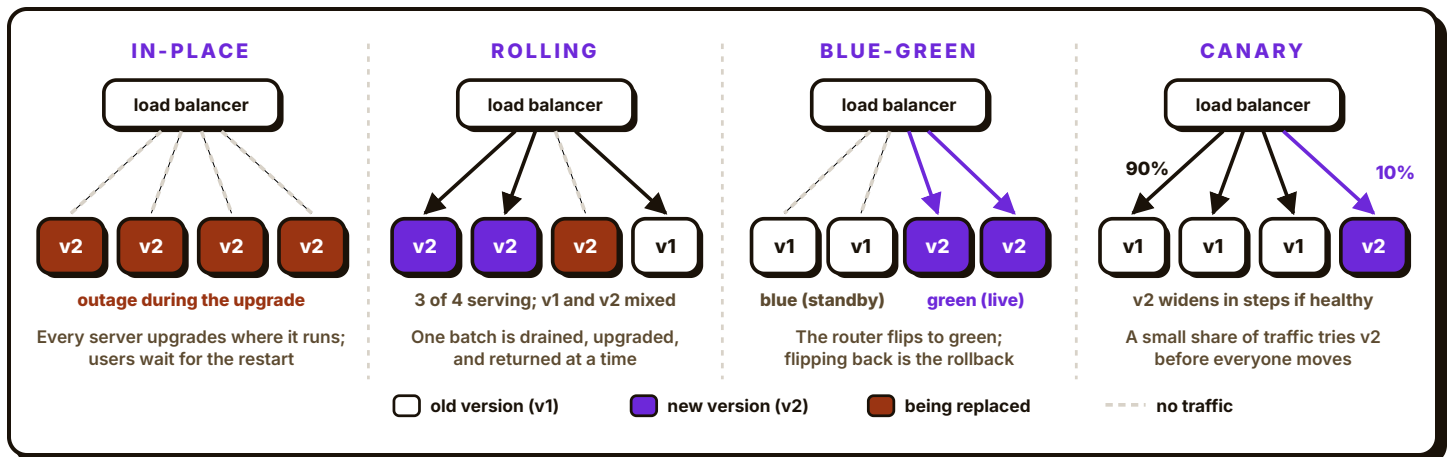


**FIGURE** The same four-server service under each delivery method. Purple boxes run the new version; rust boxes are being replaced.

### In-place and rolling

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

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

#### DEPLOY.YAML (EXCERPT)

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

### Blue-green

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

## 4.1 Deployment Strategies (continued)

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

SWITCH.SH

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

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

TERMINAL

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

### Canary

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

CONF.D/UPSTREAM.CONF

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

TERMINAL

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

### Choosing a method

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

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

## 4.1 Deployment Strategies (continued)

### TRY IT

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

### ON THE EXAM

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

## 4.1 QA Testing, Validation, and Hotfixes

4.0 CONTINUOUS DELIVERY

### Quality assurance in the delivery pipeline

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

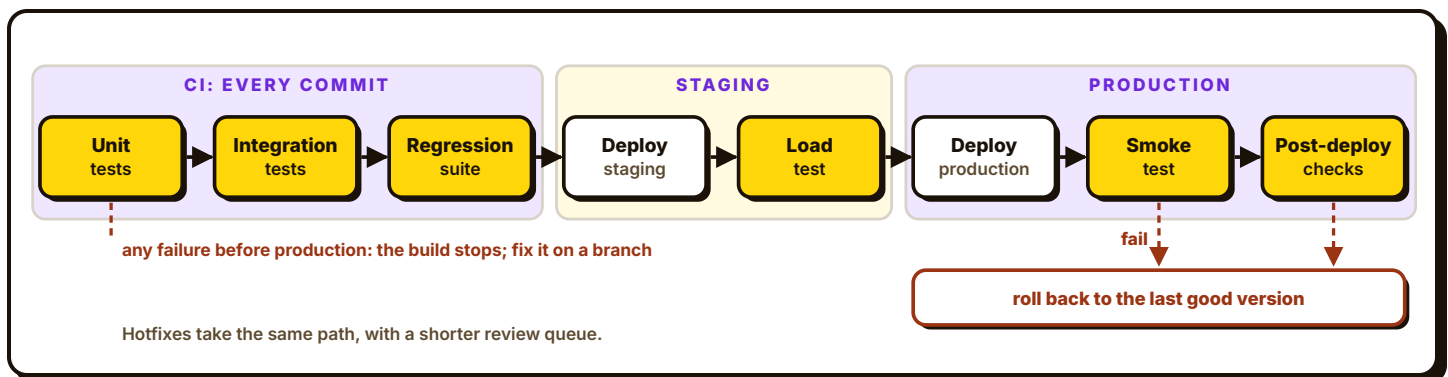


FIGURE Where each check runs. Every yellow box can stop the pipeline; checks after the production deploy trigger a rollback instead.

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

### Integration and regression testing

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

## 4.1 QA Testing, Validation, and Hotfixes (continued)

### TEST\_VLANS.PY

```
from vlans import parse_vlans

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

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

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

### BEFORE AND AFTER THE FIX

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

### Load testing

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

### AB AGAINST THE STAGING CONTAINER

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

### Smoke tests and post-deployment testing

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

### SMOKE.SH

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

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

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

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

## 4.1 QA Testing, Validation, and Hotfixes (continued)

### A GOOD DEPLOY, THEN A BROKEN ONE

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

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

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

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

### Hotfixes

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

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

### TERMINAL

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

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

### ON THE EXAM

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

### Uptime and availability

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

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

- Components in a chain multiply
  - An app at 99.9% that needs a database at 99.9%:  
 $99.9\% \times 99.9\% = 99.8\%$
  - Two redundant copies, either of which can serve:  $1 - (0.001 \times 0.001) = 99.9999\%$
- SLIs, SLOs, and SLAs**
  - Service-level indicator (SLI)**: the measurement
    - Good events divided by valid events: successful requests / all requests, requests under 300 ms / all requests
  - Service-level objective (SLO)**: the internal target for an SLI over a window
    - "99.9% of checkout requests succeed, measured over a rolling 30 days"
    - Set by engineering and product; missing it changes priorities, not invoices
  - Service-level agreement (SLA)**: a contract with customers that names an availability level and a penalty
    - Usually **service credits**: a percentage of the monthly bill refunded
    - Defines what counts: scheduled maintenance and customer-caused outages are usually excluded
- Keep the SLO stricter than the SLA
  - The gap is a warning zone: you miss your own target before you owe anyone money

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

### Error budgets

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

#### BUDGET.PY

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

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

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

TERMINAL

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

### ■ MTTR and MTBF

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

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

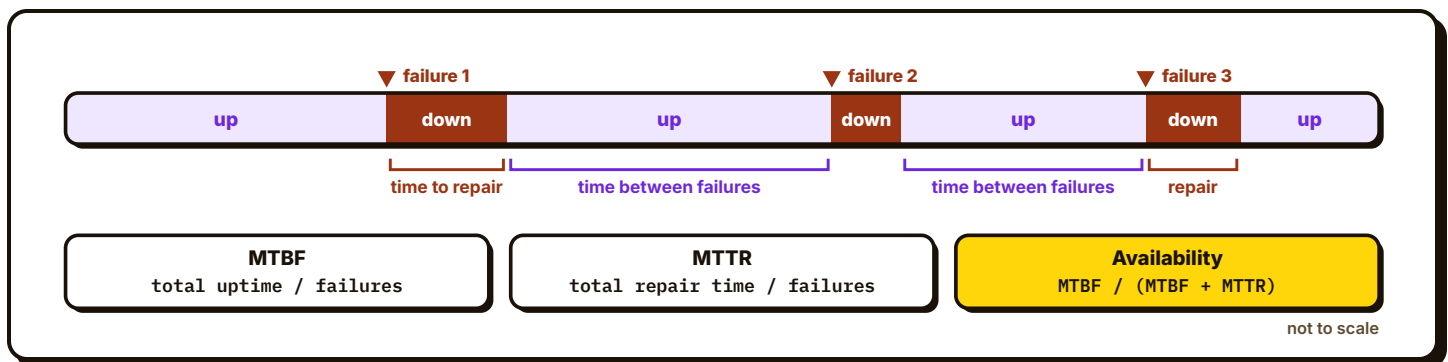


FIGURE MTBF averages the purple stretches between failures; MTTR averages the rust stretches spent restoring service.

MTBF.PY

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

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

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

### ■ Feedback loops

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

TERMINAL

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

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

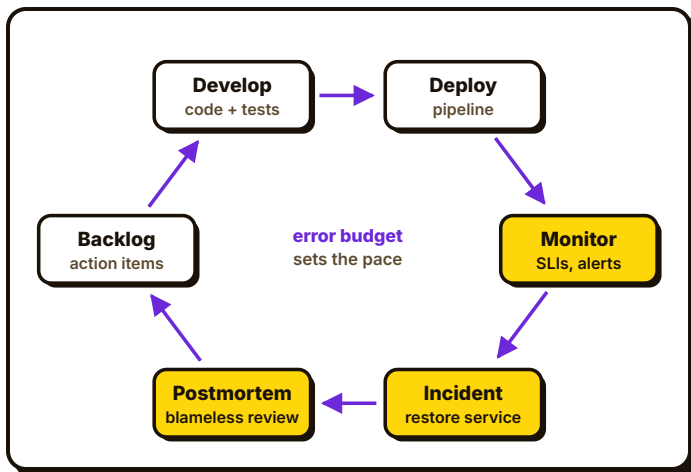


**FIGURE** Production data flows back into the next round of work. Yellow steps are where the loop learns something.

- Monitoring and alerts measure the SLIs and burn rate
- Incidents end in a blameless postmortem
  - Action items go into the backlog with an owner and a due date
  - The bug that caused it becomes a regression test

- Error budget status sets the release pace for the next window
- Delivery metrics close the loop on the pipeline itself
  - The DORA metrics: deployment frequency, lead time for changes, change failure rate, and time to recover from a failed deployment
- User signals count too: support tickets, feature usage, app store reviews

### ON THE EXAM

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

## 4.3 Securing Connections to Providers

4.0 CONTINUOUS DELIVERY

### ■ CLI configuration

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

~/.AWS/CONFIG

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

[profile lab]
region = us-west-2

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

~/.AWS/CREDENTIALS (DUMMY KEYS)

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

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

## 4.3 Securing Connections to Providers (continued)

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

TERMINAL

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

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

### ■ SDK credential chains

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

CHAIN.PY

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

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

OFFLINE RUN WITH DUMMY KEYS

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

## 4.3 Securing Connections to Providers (continued)

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

### ■ IAM and least privilege

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

#### S3-READ.JSON

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

#### PARLIAMENT (IAM POLICY LINTER)

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

### ■ Machine identities and federation

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

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

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

#### DEPLOY.YML (ACTIONLINT: OK)

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

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

### 4.3 Securing Connections to Providers (continued)

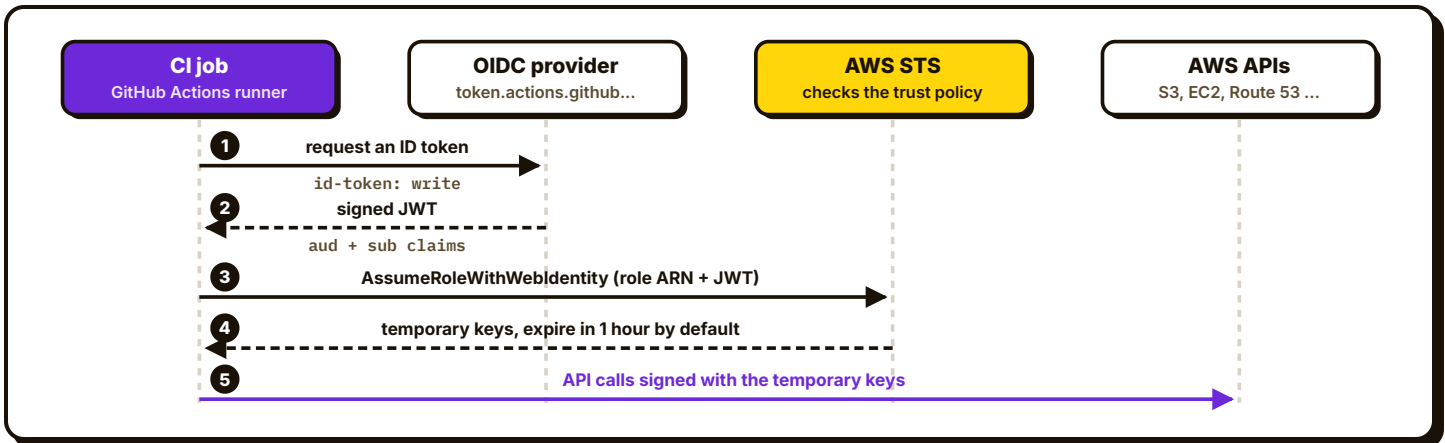


**FIGURE** GitHub Actions to AWS with no stored secret. The trust policy decides which repository, branch, or environment may assume the role.

#### ■ Accessing external APIs

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

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

#### ON THE EXAM

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