Cloud Deployment: From GitLab CI/CD to Google Cloud

Example: `cloud-demo/` · one `git push` → tested, built, and running on Google Cloud

The Lecture in One Sentence

> A developer pushes code to GitLab. GitLab CI/CD automatically **tests** the code, **builds** an image, and **deploys** it to Google Cloud. The same process deploys to **development** and to **production** — with different rules and different configuration.

Prerequisites: Git branches & merge requests · `pytest` · Dockerfiles and images. Everything else is introduced today.

Learning Objectives

1. **Distinguish** continuous integration, delivery, and deployment
2. **Write** a `.gitlab-ci.yml` with jobs, `script`, `stages`, `image`, `artifacts`, `rules` — and **predict** job order
3. **Use** CI/CD variables: predefined, YAML, and UI (masked & protected)
4. **Explain** dev vs. prod **environments**; deploy automatically to one and with approval to the other
5. **Create, start, stop, delete** a Compute Engine VM and run a container on it
6. **Deploy** to Cloud Run and **compare** it with the VM (ops work, cost)
7. **Explain** keyless auth: GitLab **OIDC** + **Workload Identity Federation**
8. **Trace** one `git push` through test → build → dev → prod, and **roll back**

Part 1

Why CI/CD?

"It Works on My Machine" — and Then What?

How you submit an assignment:

1. write code → 2. run locally → 3. commit → 4. push
2. discover it fails on the grader (missing package, other Python, forgotten test)
3. fix, repeat

Now: **10 people, 20 pushes a day, real users.** Every push asks:
Did anyone run the tests? On which machine? Who builds what we ship? Who copies it to the server? Which version is running right now?

> If the answer is "a person, by hand, from their laptop" — every release is a small gamble.

The Idea

> *What if every `git push` automatically ran the tests — on a clean machine — and, if they passed, built and deployed the application?*

That is **CI/CD**: the steps between "code is written" and "code is running for users" move out of people's heads and laptops into a **versioned, reviewable, repeatable file**.



The Vocabulary

Term	Meaning	Who presses the button?
Continuous Integration (CI)	every change is automatically built and tested, many times a day	nobody: a push triggers it
Continuous Delivery	every change that passes CI is packaged and **ready** to release	a **human** approves
Continuous Deployment	every change that passes CI is **released** automatically	nobody
CD	either of the previous two	depends — ask!

> Today's pipeline does **both**: continuous **deployment** to development (every push to `develop` goes live) and continuous **delivery** to production (merge to `main` → deployable image, a person clicks **Run**).

Running Example: `cloud-demo`

Stage of the lecture	Folder	What it adds
First job	`hello-ci/`	one job, one script, one log
Pipelines	`stages-demo/`	stages, parallel jobs, artifacts, a rule
Real app	`cloud-demo/`	FastAPI app, tests, Dockerfile
VM deploy	`cloud-demo/vm/`	Compute Engine instance + startup script
Serverless deploy	`cloud-demo/.gitlab-ci.yml`	Cloud Run dev + prod
Keyless auth	`cloud-demo/gcp/setup.sh`	Workload Identity Federation

`cloud-demo`: Boring on Purpose

Endpoint	Returns	Why it exists
`GET /`	message + which environment it runs in	shows dev vs. prod configuration
`GET /health`	`{"status": "healthy"}`	smoke tests and health checks
`GET /version`	app version, git commit, Cloud Run revision	answers "which code is running?"

> Readable in a minute, so all your attention goes to **the path it travels**. We deploy it **twice**: to a **VM** (we do the ops work) and to **Cloud Run** (Google does). Same image in both places — that's the point.

Part 2

GitLab CI/CD

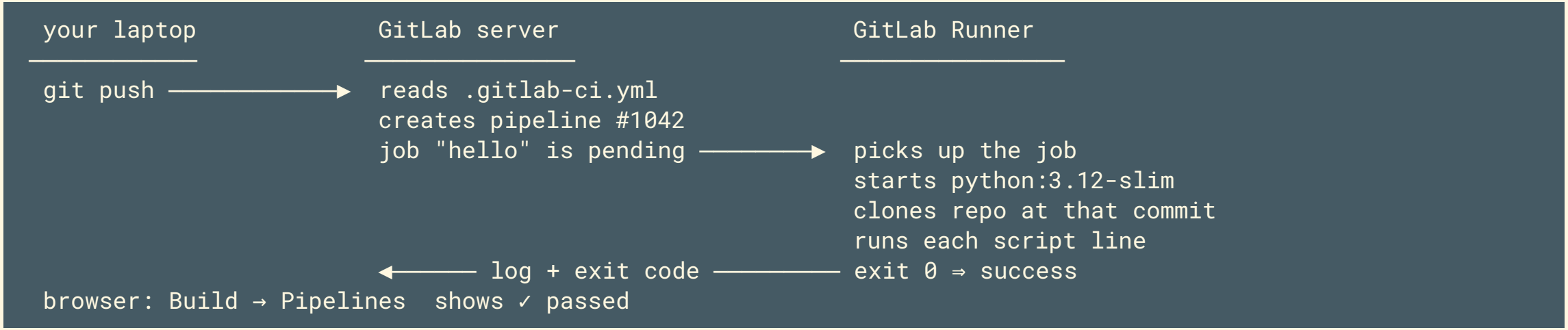
Core Concept 1: Your First Job

> **GitLab CI/CD** — a file named `.gitlab-ci.yml` at the repo root declares **jobs**. On an event (push, MR, schedule, button), GitLab creates a **pipeline** of those jobs and hands each to a **GitLab Runner**, which runs the job's `script` in a fresh environment and reports the log and exit status.

Plain English: write down the commands you'd type by hand, in a file next to your code. GitLab runs them on another computer every time you push.

Intuition: GitLab is the **manager** (reads the to-do list, assigns tasks); the runner is the **worker** (does the task in a clean room, reports "done" or "failed").

Manager and Worker



Worked Example: `hello-ci`

```python
"""The smallest possible program
to run in a GitLab CI/CD job."""
import platform

print("Hello from GitLab CI/CD!")
print(f"Running on Python "
      f"{platform.python_version()} "
      f"({platform.system()})")
```

```yaml
# .gitlab-ci.yml
# No stage: → default `test` stage.
hello:
  image: python:3.12-slim
  script:
    - echo "Starting CI/CD"
    - python --version
    - python hello.py
    - echo "Job completed!"
```

- `hello:` — a **job name** (any top-level key that isn't a reserved keyword)
- `image:` — the Docker image the job runs in: environment is **declared**, not "whatever is installed"
- `script:` — shell commands, in order; **stops at the first non-zero exit**. That exit code is the verdict.

The Job Log DEMO

Push, then **Build → Pipelines** → pipeline → `hello` job:

```
Running with gitlab-runner 17.x (…)
Preparing the "docker" executor
Using docker image python:3.12-slim …
Checking out 3f9a1c2e as detached HEAD (ref is main)...
$ echo "Starting CI/CD"
Starting CI/CD
$ python --version
Python 3.12.7
$ python hello.py
Hello from GitLab CI/CD!
Running on Python 3.12.7 (Linux)
$ echo "Job completed!"
Job completed!
Job succeeded
```

Notice `(Linux)` — even if you pushed from a Mac, the job ran in a Linux container somewhere else.

Three Questions for the Class

1. **What if `hello.py` doesn't exist?**
   `python hello.py` exits 2 → runner stops → `ERROR: Job failed: exit code 2` → pipeline turns red. `echo "Job completed!"` never runs.
2. **What if any command returns non-zero?** Same thing.

> **The whole contract: exit code 0 = success, anything else = failure.** `pytest` exits non-zero on a failing test — so "run tests in CI" is a one-line job.

3. **Does GitLab run these on your laptop?** No. GitLab **orchestrates**; a runner **executes** (GitLab-hosted VMs, university shared runners, or your own).

Common Misconceptions — Jobs

- *"The pipeline runs on the GitLab server."* The server only schedules. No runner → job sits **pending forever** ("stuck pending" ≈ no runner with matching tags).
- *"Files from one job are there in the next."* Every job starts from a **fresh clone in a fresh container**. Use `artifacts` to pass files.
- *"Red warning text means the job failed."* Only the **exit code** matters. A script that prints "ERROR" and exits 0 **passes**.
- *"YAML is forgiving."* It is not. Indentation is structure; a tab is an error. Use **Build → Pipeline editor** to validate.

> Because the procedure lives in the repo, it is **versioned**, **reviewable** (via MRs), and **reproducible** (same commit → same steps → same image).

Core Concept 2: Jobs, Stages, Pipelines

Concept	Definition	Analogy
Job	named unit of work with a script; smallest thing a runner executes	one task on the to-do list
Script	the shell commands a job runs	instructions for that task
Stage	named group of jobs; stages run in listed order	a phase: build, then test, then ship
Pipeline	all jobs created for one event, organized by stages	the whole list for one release attempt
Runner	machine/process that executes jobs	the worker

Rule 1: jobs in the **same stage** may run **in parallel**.
Rule 2: a stage starts only when **every job in the previous stage succeeded**.

Worked Example: `stages-demo`

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

build:
  stage: build
  script:
    - echo "Building application"
    - mkdir -p dist
    - echo "build $CI_COMMIT_SHORT_SHA" > dist/build.txt
  artifacts:            # hand dist/ to later stages
    paths: [dist/]

unit-test:              # same stage as lint →
  stage: test           # runs in parallel
  script:
    - cat dist/build.txt    # artifact from `build`
    - echo "Running unit tests"
```

```yaml
lint:
  stage: test
  script:
    - echo "Running linter"
    # uncomment to watch a failure:
    # - exit 1

deploy:
  stage: deploy
  script:
    - echo "Deploying $CI_COMMIT_SHORT_SHA
       from $CI_COMMIT_BRANCH"
  rules:   # only deploy from main
    - if: $CI_COMMIT_BRANCH ==
          $CI_DEFAULT_BRANCH
```

Predict before you push: job order on `main`? On a branch `feature-x`?

Tracing `stages-demo`

On `main`:

```
time ──────────────────────────────────────────────────────────►
stage build    [ build ✓ ]──── artifacts: dist/ uploaded
stage test                    [ unit-test ✓ ]   (parallel)
                              [ lint      ✓ ]
stage deploy                                    [ deploy ✓ ]
```

On `feature-x`: `deploy` does not exist — its rule is false, so GitLab never creates it. 3 jobs; pipeline ends after `test`.

On `main` with `exit 1` in `lint`:

```
stage build    [ build ✓ ]
stage test                    [ unit-test ✓ ]
                              [ lint      ✗ ]  exit code 1
stage deploy                                     [ deploy ⊘ skipped ]   pipeline: failed
```

→ Nothing ships unless everything before it passed.

Three Keywords from the Example

- `artifacts` — files a job uploads when it finishes; later-stage jobs download them automatically. That's how `unit-test` can `cat dist/build.txt` in a brand-new container. Also used for test reports, coverage, binaries.
- `rules` — decide **whether a job is added to the pipeline at all**, and how (`when: manual`, `never`, `on_success`). First matching `if:` wins; no match → job left out. (Prefer over legacy `only:` / `except:`.)
- `$CI_COMMIT_SHORT_SHA`, `$CI_COMMIT_BRANCH`, `$CI_DEFAULT_BRANCH` — **predefined variables** GitLab injects into every job.

What Triggers a Pipeline?

Event	`$CI_PIPELINE_SOURCE`	Typical use
git push to a branch or tag	`push`	test every change
merge request opened/updated	`merge_request_event`	gate merges on green tests
schedule (cron)	`schedule`	nightly dependency checks
Run pipeline button / API	`web` / `api` / `trigger`	manual re-deploys

Misconceptions:

- *"Stages run alphabetically / jobs in file order."* Stages follow the `stages:` list; **no order within a stage**.
- *"A job with a false rule is skipped."* It's **absent**. ⊘ *skipped* = it exists but an earlier failure blocked it.
- *"More stages = safer."* Every boundary is a wait. Group independent jobs, or use `needs:`.

Core Concept 3: CI/CD Variables

> A **CI/CD variable** is a named value GitLab exports as an environment variable into a job's shell. Three sources: **predefined** by GitLab · **declared in** `.gitlab-ci.yml` · **defined in the UI** (Settings → CI/CD → Variables).

Plain English: how the *same* pipeline file behaves differently depending on *where* it runs and *what* it deploys to — without editing the file.

Intuition: the twelve-factor rule — *configuration lives in the environment, not the code* — applied to the pipeline. The app reads `ENVIRONMENT`; the pipeline reads `GCP_PROJECT_ID`.

Source 1: Predefined Variables

GitLab sets well over a hundred. The ones you'll use constantly:

Variable	Example value	Use
`CI_COMMIT_SHA`	`3f9a1c2e7b…` (40 hex)	exact commit
`CI_COMMIT_SHORT_SHA`	`3f9a1c2e`	image tags, version labels
`CI_COMMIT_BRANCH`	`develop`	rules: which branch?
`CI_DEFAULT_BRANCH`	`main`	rules without hard-coding "main"
`CI_PIPELINE_SOURCE`	`push`	rules: what triggered us?
`CI_PROJECT_PATH`	`my-group/cloud-demo`	identity in OIDC tokens
`CI_PROJECT_DIR`	`/builds/my-group/cloud-demo`	where the repo is checked out
`CI_JOB_NAME`	`deploy_dev`	cache keys, logging

Sources 2 & 3: YAML and the UI

In `.gitlab-ci.yml` — non-secret config that belongs with the code:

```yaml
variables:          # global
  REGION: "us-east1"
  IMAGE_NAME: "cloud-demo"
  IMAGE_TAG: "${CI_COMMIT_SHORT_SHA}"

deploy_dev:
  variables:        # job-level: overrides
    SERVICE: cloud-demo-dev
    ENVIRONMENT: development
  script:
    - echo "$IMAGE_NAME:$IMAGE_TAG
        → $SERVICE in $REGION"
```

In the UI — per-project values, or things that must not be in the repo:

Option	Effect
Masked	shown as `[MASKED]` in logs
Protected	only on protected branches/tags
Env. scope	only for matching `environment`
Expand	expand `$OTHER` in value

Precedence (simplified): manual-run > project/group/instance UI > job YAML > global YAML > predefined. Just avoid defining a name twice.

Part 3

Tests, Images, and Environments

Core Concept 4: The Real Application

```python
import os
from fastapi import FastAPI

APP_VERSION = "1.0"

def create_app() -> FastAPI:
    app = FastAPI(title="cloud-demo",
                  version=APP_VERSION)

    @app.get("/")
    def home():
        return {
          "message": "Hello from Google Cloud!",
          "environment":
              os.getenv("ENVIRONMENT", "local"),
        }

    @app.get("/health")
    def health():
        return {"status": "healthy"}
```

```python
    @app.get("/version")
    def version():
        # "Which code is running?"
        return {
          "version": APP_VERSION,
          "commit":
            os.getenv("GIT_SHA", "unknown"),
          "revision":
            os.getenv("K_REVISION", "n/a"),
        }
    return app

app = create_app()   # uvicorn "app:app"
```

- **All environment-specific values come from env vars** (`ENVIRONMENT`, `GIT_SHA`, `PORT`) — code is identical in dev and prod
- `K_REVISION` is set by Cloud Run · `/version` exists because "what is deployed?" should be one `curl` away

The Tests

```python
@pytest.fixture
def client():
    return TestClient(create_app())

def test_home(client):
    response = client.get("/")
    assert response.status_code == 200
    assert response.json()["message"] == "Hello from Google Cloud!"

def test_home_reports_configured_environment(client, monkeypatch):
    # Same code, different configuration: dev vs. prod in one test.
    monkeypatch.setenv("ENVIRONMENT", "production")
    assert client.get("/").json()["environment"] == "production"

def test_version_reports_commit(client, monkeypatch):
    monkeypatch.setenv("GIT_SHA", "a1b2c3d4")
    body = client.get("/version").json()
    assert body["version"] == APP_VERSION
    assert body["commit"] == "a1b2c3d4"
```

FastAPI's `TestClient` calls the app **in-process** — no server, no network — so the suite runs in well under a second.

The Test Job

```yaml
stages:
  - test

test:
  stage: test
  image: python:3.12-slim
  variables:
    PIP_CACHE_DIR: "${CI_PROJECT_DIR}/.cache/pip"
  cache:              # reuse wheels across pipelines
    key: pip-${CI_JOB_NAME}
    paths: [.cache/pip]
  script:
    - pip install -r requirements-dev.txt
    - pytest -v --junitxml=report.xml
  artifacts:
    when: always      # keep report even on failure
    reports:
      junit: report.xml   # per-test results in UI
```

Why, if you ran tests locally?

- You may forget. CI doesn't.
- A teammate's unrelated change may break your code — CI runs **all** tests on **every** change.
- Your laptop has packages `requirements.txt` forgot. CI has exactly what the file says.
- A failed test **stops deployment** — via stage order.

> `cache` = best-effort, between **pipelines** (speed-up).
> `artifacts` = guaranteed, between **jobs**. Never pass build outputs via cache.

The Image

```dockerfile
FROM python:3.12-slim

ENV PYTHONDONTWRITEBYTECODE=1 \
    PYTHONUNBUFFERED=1 \
    PORT=8080

WORKDIR /app

# Deps first: cached until requirements change
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Then the code (changes often)
COPY app.py .

RUN useradd --create-home appuser
USER appuser              # unprivileged

EXPOSE 8080
# Shell form so $PORT is expanded at start-up
CMD exec uvicorn app:app --host 0.0.0.0 \
    --port ${PORT} --workers 2
```

Thing	Role
`requirements.txt`	runtime deps — **no pytest** (tests don't ship)
image	packaged, immutable; **what we deploy**
container	running instance of an image
registry	where images live so others can pull

Why not `python app.py`? That's `reload=True`: one process watching files — great for dev, wrong for prod. `CMD` runs without reload and with `--workers 2`.

Core Concept 5: Development vs. Production

An **environment** is a distinct deployment target — its own running instance of the app, with its own **configuration, data, access rules, and URL**. Minimal set: **development** (try changes) and **production** (serve real users); many teams add **staging**.

Plain English: same app, different copies, for different audiences.

Misconception to kill first: *"dev and prod are two Git branches."*
Branches hold code; environments run deployed copies. The pipeline *maps* `develop` →development and `main` →production — a choice written in `.gitlab-ci.yml`, not a property of Git.

How They Differ

	Development	Production
Users	the team	real users
Data	test / fake	real
Update frequency	many times a day	deliberate releases
Deploy trigger	**automatic** on push to `develop`	**manual approval** on `main`
Debugging	verbose logs, debug flags OK	debug off, structured logs
Access	loose	restricted: who deploys, who reads data
Failure cost	annoying	users affected, money, trust
Our service	`cloud-demo-dev`	`cloud-demo-prod`

Same Image, Two Configurations

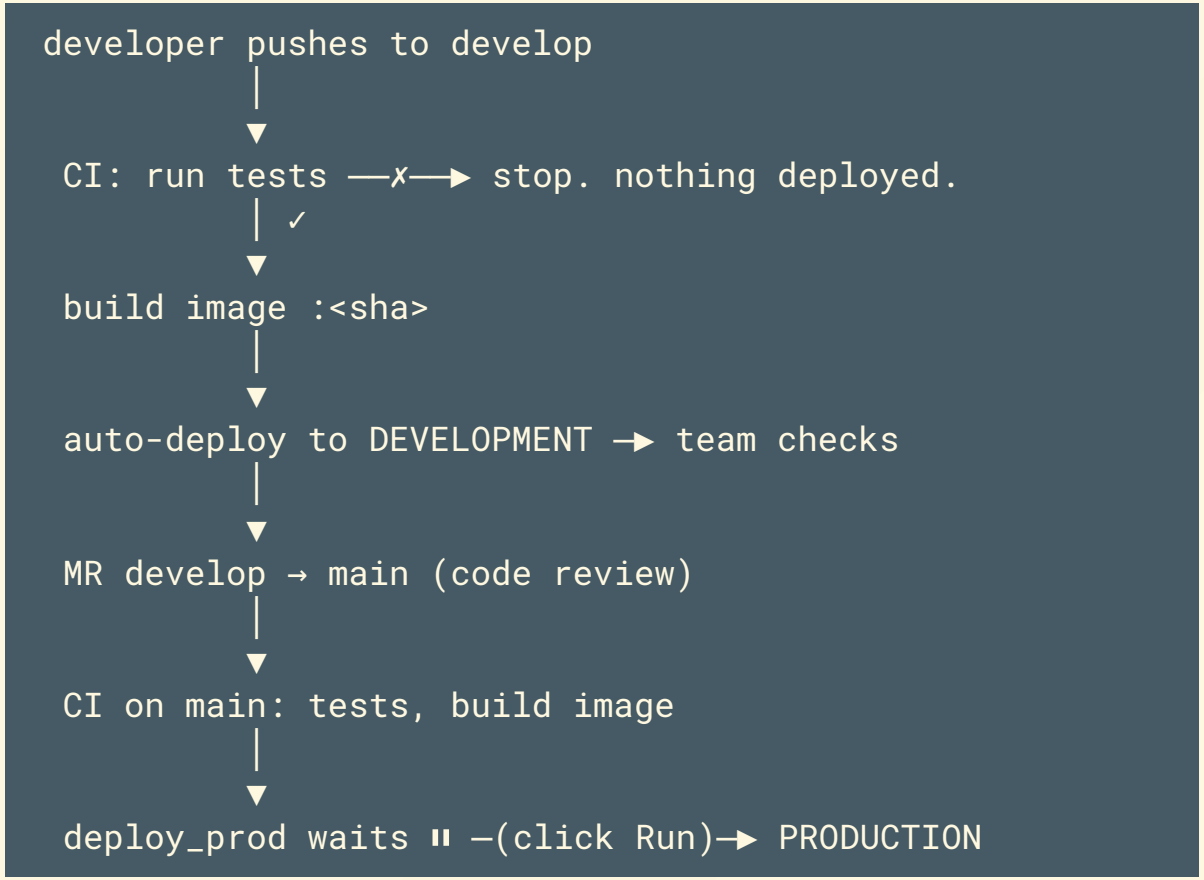
```
gcloud run deploy cloud-demo-dev \
  --image "$IMAGE_URL:3f9a1c2e" --region us-east1 \
  --set-env-vars ENVIRONMENT=development

gcloud run deploy cloud-demo-prod \
  --image "$IMAGE_URL:3f9a1c2e" --region us-east1 \
  --set-env-vars ENVIRONMENT=production
```

```
$ curl https://cloud-demo-dev-123456789.us-east1.run.app/
{"environment":"development","message":"Hello from Google Cloud!"}
$ curl https://cloud-demo-prod-123456789.us-east1.run.app/
{"environment":"production","message":"Hello from Google Cloud!"}
```

Code identical; configuration and target differ. Both use tag `3f9a1c2e`: what you verified in dev is **bit-for-bit** what goes to prod.

The Release Flow



Three principles

1. **Dev deploys can be automatic** — a broken dev env costs a few minutes.
2. **Prod deploys can require approval** — `when: manual` waits for a click; protected branches/environments restrict **who** clicks.
3. **Prod runs a tested, identifiable image** — not "whatever's on someone's laptop."

Part 4

Google Cloud: VMs and Cloud Run

Core Concept 6: Google Cloud Fundamentals

- **Project** — container for everything: resources, permissions, APIs, the bill. Has a **project ID** (`my-class-project`) and **project number** (`123456789012`).
- **Billing account** — pays for a project; even "free" needs one. **Set a budget alert on day one.**
- **Region** (`us-east1`) = geographic area; **zone** (`us-east1-b`) = isolated data center within it. VMs live in a **zone**; Cloud Run & Artifact Registry in a **region**.
- **APIs must be enabled** per project. "API not enabled" is the most common first error.

```
gcloud auth login                          # (laptop only) log in as yourself
gcloud config set project my-class-project
gcloud config set compute/zone us-east1-b
gcloud services enable compute.googleapis.com run.googleapis.com \
    artifactregistry.googleapis.com cloudbuild.googleapis.com
```

Easiest place to run gcloud: **Cloud Shell** (the >_ icon) — free, pre-authenticated, in the browser.

IAM: Who Can Do What

Identity and Access Management answers: *may principal **P** perform action **A** on resource **R**?*

- **Principal** — an identity: a user, a group, or a **service account** (an identity for a *program*, e.g. `gitlab-deployer@my-class-project.iam.gserviceaccount.com`)
- **Role** — named bundle of permissions: `roles/run.admin`, `roles/artifactregistry.writer`, …
- **IAM binding** — grants a role to a principal on a resource

> **Least privilege** — three service accounts:
> the **pipeline** acts as one (deploys) · the **app** runs as one on Cloud Run (needs nothing) · the **VM** runs as one (pulls images, writes logs)

The Services We Use

Service	What it does for us
Artifact Registry	private Docker registry: stores `cloud-demo:<sha>`
Cloud Build	builds a Dockerfile on Google's machines, pushes the image
Compute Engine	virtual machines (Core Concept 7)
Cloud Run	runs containers on demand, HTTPS included (Core Concept 8)
IAM + Workload Identity Federation	lets GitLab act in our project without a key (Core Concept 9)

Create the image repository once:

```
gcloud artifacts repositories create class-docker \
  --repository-format=docker --location=us-east1 \
  --description="CMSC389A class applications"
```

Core Concept 7: Compute Engine

> **Compute Engine** is Google Cloud's **IaaS**: it rents **VM instances** — vCPU + memory (**machine type**), a **boot disk** from an OS **image**, a network interface — in a chosen zone, billed per second while running.

Plain English: a computer in Google's data center that is yours. You get root, an IP address, and **every responsibility**.

Intuition: renting an **empty apartment**. Furnish it any way you like — but you also fix the plumbing.

Anatomy of an Instance

Setting	Our choice	Meaning
name	`cloud-demo-vm`	unique within project + zone
zone	`us-east1-b`	where it physically runs
machine type	`e2-micro`	2 shared vCPUs, 1 GB RAM; free-tier eligible
boot disk image	`debian-12` (`debian-cloud`)	the OS; gcloud preinstalled
network tags	`http-server`	labels firewall rules target
service account	`cloud-demo-vm@…`	identity programs on the VM act as
metadata	`app-image`, `startup-script`	key/value data the VM reads at boot

Create & Start an Instance DEMO

Console

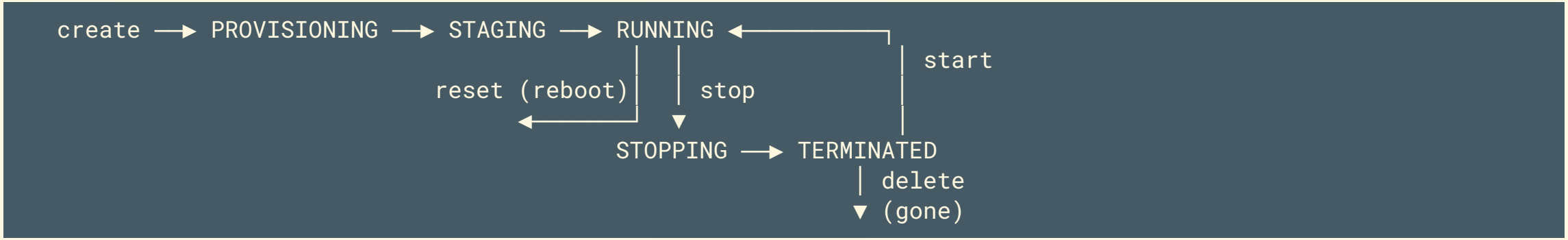
1. Compute Engine → VM instances (enable API)
2. **Create instance**
3. Name `cloud-demo-vm`; `us-east1` / `us-east1-b`
4. Series **E2**, type **e2-micro**
5. Boot disk: **Debian 12**, 10 GB
6. Firewall: ☑ **Allow HTTP traffic**
7. **Create** → status turns green: **RUNNING** (create also starts it)
8. Click **SSH** → browser terminal on your VM

`gcloud`

```
gcloud compute instances create \
  cloud-demo-vm \
  --zone=us-east1-b \
  --machine-type=e2-micro \
  --image-family=debian-12 \
  --image-project=debian-cloud \
  --boot-disk-size=10GB \
  --tags=http-server
```

> Prefer the command: it can be **committed, reviewed, and rerun**. A click-path lives only in your memory.

The Instance Lifecycle




```
gcloud compute instances list
gcloud compute instances stop   cloud-demo-vm --zone=us-east1-b
gcloud compute instances start  cloud-demo-vm --zone=us-east1-b
gcloud compute instances reset  cloud-demo-vm --zone=us-east1-b   # hard reboot
gcloud compute ssh              cloud-demo-vm --zone=us-east1-b
gcloud compute instances delete cloud-demo-vm --zone=us-east1-b
```

> RUNNING pays CPU + memory. Stopped (TERMINATED) still pays **disk** + any static IP. Only **delete** stops all charges. *"I stopped it" ≠ "I deleted it."*

Running `cloud-demo` on the VM: Startup Script

```
#!/usr/bin/env bash
set -euo pipefail
MD="http://metadata.google.internal/computeMetadata/v1/instance/attributes"
md() { curl -fsS -H "Metadata-Flavor: Google" "$MD/$1"; }

if ! command -v docker >/dev/null; then  # first boot
  apt-get update && apt-get install -y docker.io
  systemctl enable --now docker
fi

IMAGE="$(md app-image)"            # from metadata
ENVIRONMENT="$(md environment || echo vm)"
REGISTRY_HOST="${IMAGE%%/*}"
gcloud auth configure-docker "$REGISTRY_HOST" --quiet

docker pull "$IMAGE"
docker rm -f cloud-demo 2>/dev/null || true
docker run -d --name cloud-demo \
  --restart=unless-stopped \
  -p 80:8080 \
  -e ENVIRONMENT="$ENVIRONMENT" \
  -e GIT_SHA="${IMAGE##*:}" "$IMAGE"
```

A **startup script** lives in instance metadata; Google runs it as root **on every boot**.

- The VM learns **which** image from the **metadata server** — the script is generic; metadata is the config
- `-p 80:8080` maps VM port 80 (firewall-open) → uvicorn's 8080
- `--restart=unless-stopped` — **our** supervision; nobody else will restart it

Creating the VM with Identity + Startup Script

```
# The VM's own identity: it only needs to pull images and write logs.
gcloud iam service-accounts create cloud-demo-vm
gcloud projects add-iam-policy-binding "$PROJECT_ID" \
  --member="serviceAccount:cloud-demo-vm@${PROJECT_ID}.iam.gserviceaccount.com" \
  --role=roles/artifactregistry.reader

# Firewall: allow HTTP to VMs tagged http-server.
gcloud compute firewall-rules create allow-http \
  --network=default --direction=INGRESS --action=ALLOW \
  --rules=tcp:80 --source-ranges=0.0.0.0/0 --target-tags=http-server

gcloud compute instances create cloud-demo-vm \
  --zone=us-east1-b --machine-type=e2-micro \
  --image-family=debian-12 --image-project=debian-cloud \
  --tags=http-server \
  --service-account="cloud-demo-vm@${PROJECT_ID}.iam.gserviceaccount.com" \
  --scopes=cloud-platform \
  --metadata=app-image="$IMAGE",environment=vm \
  --metadata-from-file=startup-script=vm/startup.sh
```

From `vm/create-vm.sh` in the project.

Verify, Debug, Redeploy

About a minute later:

```
$ curl http://34.75.112.18/version
{"commit":"3f9a1c2e","revision":"n/a","version":"1.0"}
```

Not answering? Read the **serial console** — the VM's equivalent of a CI job log:

```
gcloud compute instances get-serial-port-output cloud-demo-vm \
  --zone=us-east1-b | grep startup-script
```

Deploy a new version (`vm/deploy-vm.sh`) = point metadata at a new tag and reboot:

```
gcloud compute instances add-metadata cloud-demo-vm --zone=us-east1-b \
  --metadata=app-image="$IMAGE_URL:7c41d0aa"
gcloud compute instances reset cloud-demo-vm --zone=us-east1-b
```

> That works — and users see **~a minute of errors** while the VM reboots. Hold that thought.

What You Now Own

Responsibility	On our VM
OS security patches	you (or configure unattended upgrades)
HTTPS / TLS certificate	you: Caddy or nginx + certbot, plus a domain
Process crashes	Docker's `--restart` policy
VM / zone failure	nothing: the service is **down**
Traffic beyond one e2-micro	resize by hand (downtime) or a managed instance group
Zero-downtime deploys	not with `reset`; needs two VMs + a load balancer
Cost when idle	billed **24/7** whether anyone visits or not

Misconceptions: *"A VM isn't a real deployment"* — it is; many businesses run on a few well-kept VMs. · *"The external IP is permanent"* — ephemeral IPs can change on stop/start. · *"Allow HTTP makes the app reachable"* — it opens port 80; something must **listen** there (hence `-p 80:8080`).

Core Concept 8: Cloud Run

> **Cloud Run** runs a container image as an HTTPS **service**. Each deployment creates an immutable **revision**; Cloud Run starts instances **on demand**, scales out with traffic and **down to zero** when idle, and bills for resources used while handling requests.

Plain English: hand Google an image; get back a URL. No machines to create, patch, or resize.

Intuition: the VM was an apartment. Cloud Run is a **hotel**: bring your suitcase (the image); housekeeping (TLS, restarts, scaling) is included; pay per night you actually sleep there.

The Container Contract

Cloud Run asks very little of your image:

- **Listen** for HTTP on `0.0.0.0` at the port in `$PORT` (default 8080) — our Dockerfile's `CMD` does exactly this
- Be **stateless** — any instance may be stopped any time; two requests from one user may hit different instances. State goes in a **database**, never memory or local files
- **Start reasonably fast** — after scaling to zero, the first request waits for a **cold start**

Cloud Run sets `K_SERVICE`, `K_REVISION`, `K_CONFIGURATION` for you — that's where `/version`'s `revision` comes from.

The Fastest Possible Deploy DEMO

```
gcloud run deploy cloud-demo --source . --region us-east1 --allow-unauthenticated
```

`--source .` uploads the folder → Cloud Build builds the Dockerfile → pushes to Artifact Registry → deploys.

```
✓ Building and deploying new service... Done.
  ✓ Uploading sources...    ✓ Building Container...    ✓ Creating Revision...
  ✓ Routing traffic...      ✓ Setting IAM Policy...
Service [cloud-demo] revision [cloud-demo-00001-x7k] has been deployed and is serving 100 percent of traffic.
Service URL: https://cloud-demo-123456789012.us-east1.run.app
```

```
$ curl https://cloud-demo-123456789012.us-east1.run.app/version
{"commit":"unknown","revision":"cloud-demo-00001-x7k","version":"1.0"}
```

Without asking: **HTTPS certificate, stable URL, automatic restarts, autoscaling.**

`--allow-unauthenticated` makes it public.

Deploying a Pre-Built Image

The pipeline won't use `--source`. It builds the image in its own stage (so the **same** image goes to dev and prod), then deploys by URL:

```
gcloud run deploy cloud-demo-dev \
  --image=us-east1-docker.pkg.dev/my-class-project/class-docker/cloud-demo:3f9a1c2e \
  --region=us-east1 \
  --service-account=cloud-demo-runtime@my-class-project.iam.gserviceaccount.com \
  --set-env-vars=ENVIRONMENT=development,GIT_SHA=3f9a1c2e \
  --allow-unauthenticated
```

Revisions Make Deploys and Rollbacks Safe

Each deploy creates a **new revision**; Cloud Run starts it, waits until it accepts connections, **then** shifts traffic. The old one serves until the switch. (Compare: the VM's "reset and pray.")

```
deploy 7c41d0aa:
  cloud-demo-prod-00004  (3f9a1c2e)  ▓▓▓▓▓▓▓▓▓▓ 100%  ⟶   0%  (kept, idle)
  cloud-demo-prod-00005  (7c41d0aa)  starting… ready ✓   ⟶ 100%
```

Rolling back is a traffic change, not a rebuild:

```
gcloud run revisions list --service=cloud-demo-prod --region=us-east1
gcloud run services update-traffic cloud-demo-prod --region=us-east1 \
  --to-revisions=cloud-demo-prod-00004=100
```

VM vs. Cloud Run, Side by Side

	Compute Engine VM	Cloud Run
You provide	machine config + everything on it	an image
HTTPS	you set it up	included
Crash recovery	Docker restart policy, if you set it	automatic
Scaling	manual (or instance groups)	automatic, **including to zero**
Deploy	metadata + reboot: **downtime**	new revision, then traffic switch
Rollback	redeploy old tag + reboot	move traffic to old revision
Idle cost	full price, 24/7	≈ $0
Freedom	anything: GPUs, daemons, long jobs, any port	HTTP request/response, stateless
Good for	long-running / stateful / unusual	web apps & APIs like `cloud-demo`

Same image in both columns. **The image is the portable part; the platform is the choice.**

Common Misconceptions — Cloud Run

- *"Serverless means no servers."* There are servers; you just don't manage them. What disappears is **capacity planning**.
- *"Cloud Run is always cheaper."* Per CPU-second it's **more** expensive; it wins by charging ~nothing while idle. A service busy 24/7 can cost more than a VM.
- *"I can keep a session dict in memory."* The next request may hit another instance; idle instances are destroyed. **State belongs in a database.**
- *"If Cloud Run manages infra, why Docker?"* The **image is the interface** — it tells Cloud Run exactly what to run, and makes the build portable to a VM, a laptop, or another cloud.

Part 5

Authenticating GitLab to Google Cloud

Core Concept 9: The Problem

The pipeline must run `gcloud builds submit` and `gcloud run deploy` **as someone**.
How does a GitLab job prove it may deploy — without a password in a GitLab file?

The old answer: a **service-account key** — a JSON private key pasted into a CI variable. It works, and it causes a steady stream of real breaches:

- **Never expires** unless someone rotates it (nobody does)
- **Works from anywhere** — leak it once (a log, a fork, a laptop backup) and an attacker deploys to your project
- It's a **secret** you must store, mask, protect, and audit

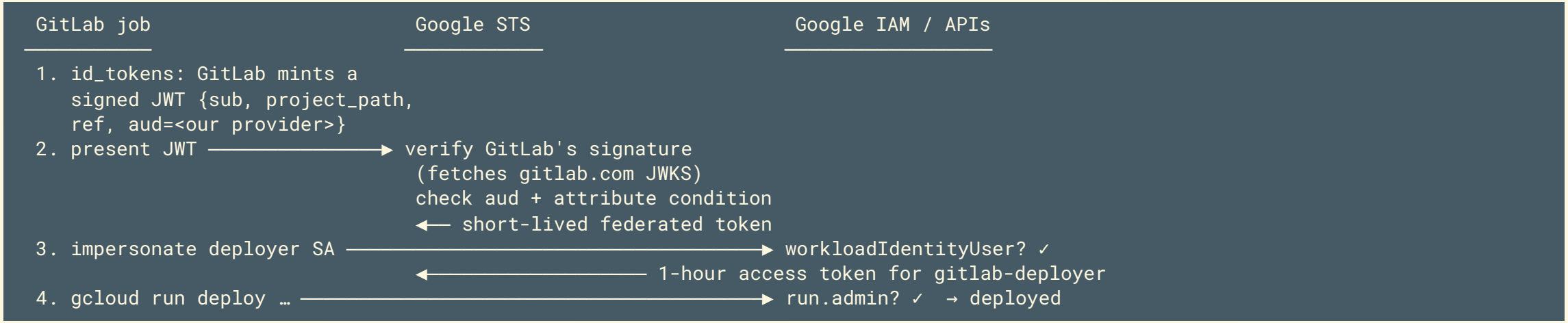
The Modern Answer: OIDC + Workload Identity Federation

> **Workload Identity Federation (WIF)** lets Google Cloud trust identity tokens issued by an external **identity provider** — GitLab, via **OpenID Connect (OIDC)** — and exchange them for **short-lived** Google credentials, subject to conditions on the token's **claims**.

Plain English: GitLab signs a note: *"this is a job from* `my-group/cloud-demo`*, branch* `main`*, pipeline 1042."* Google checks GitLab's signature, checks **your** rule ("only accept notes from `my-group/cloud-demo`"), and hands back a credential that **expires in about an hour**.

Intuition: a key is a **house key** you copy and hand out. WIF is a **front desk** that checks photo ID each time and issues a **day pass**.

The Token Exchange



Nothing long-lived exists anywhere. The JWT (step 1) is valid for minutes, for one job; the credential (step 3) for about an hour.

One-Time Setup: Who May Act as the Deployer

Allow identities from our repo to act as the deployer service account:

```
gcloud iam service-accounts add-iam-policy-binding \
  gitlab-deployer@my-class-project.iam.gserviceaccount.com \
  --role=roles/iam.workloadIdentityUser \
  --member="principalSet://iam.googleapis.com/projects/$PROJECT_NUMBER/locations/global/workloadIdentityPools/gitlab-pool/attribute.project_path/my-group/cloud-demo"
```

Give the deployer **only** what deploying requires:

Role (on the project unless noted)	Why
`roles/run.admin`	deploy services; make them public
`roles/iam.serviceAccountUser` on the runtime SA	assign the app's identity
`roles/cloudbuild.builds.editor`	submit builds
`roles/storage.admin`	upload source to Cloud Build's staging bucket
`roles/artifactregistry.writer`	push images
`roles/viewer`	stream Cloud Build logs into the job log

The Variables — None of Them Secret

Variable	Example
`GCP_PROJECT_ID`	`my-class-project`
`GCP_SERVICE_ACCOUNT`	`gitlab-deployer@my-class-project.iam.gserviceaccount.com`
`GCP_WIF_PROVIDER`	`projects/123456789012/locations/global/workloadIdentityPools/gitlab-pool/providers/gitlab-provider`
`GCP_WIF_AUDIENCE`	`https://iam.googleapis.com/` + the `GCP_WIF_PROVIDER` value
`GCP_RUNTIME_SA`	`cloud-demo-runtime@my-class-project.iam.gserviceaccount.com`

Misconceptions:

- *"If the variables aren't secret, anyone can deploy."* Knowing the provider name is like knowing a bank's address. You need a token **signed by GitLab** whose claims satisfy **your** condition.
- *"The OIDC token is like a password."* It's short-lived and audience-bound — still, don't print it. The template writes it to a file, never the log.
- *"WIF is only for big companies."* It's the **default** recommendation from GitLab and Google: one script to set up.

Part 6

Putting It Together

The Complete Pipeline (1/2)

```yaml
stages: [test, build, deploy_dev, deploy_prod]

variables:
  REGION: "us-east1"
  REPOSITORY: "class-docker"
  IMAGE_NAME: "cloud-demo"
  IMAGE_TAG: "${CI_COMMIT_SHORT_SHA}"    # one immutable tag per commit
  IMAGE_URL: "${REGION}-docker.pkg.dev/${GCP_PROJECT_ID}/${REPOSITORY}/${IMAGE_NAME}"
  PIP_CACHE_DIR: "${CI_PROJECT_DIR}/.cache/pip"

.gcp_auth:            # hidden template: OIDC token → Google credentials (previous slide)
  image: google/cloud-sdk:slim
  id_tokens: { GITLAB_OIDC_TOKEN: { aud: $GCP_WIF_AUDIENCE } }
  before_script: [ ... ]

.deploy_rules:
  rules: &deploy_rules                # YAML anchor: reused by `build`
    - if: $CI_COMMIT_BRANCH == "develop"
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH

test:                                 # 1. runs on every branch; failure stops everything
  stage: test
  image: python:3.12-slim
  cache: { key: pip-${CI_JOB_NAME}, paths: [.cache/pip] }
  script:
    - pip install -r requirements-dev.txt
    - pytest -v --junitxml=report.xml
  artifacts: { when: always, reports: { junit: report.xml } }

build:                                # 2. Cloud Build builds Dockerfile → Artifact Registry
  stage: build
  extends: .gcp_auth
  script:
    - gcloud builds submit --region="$REGION" --tag="${IMAGE_URL}:${IMAGE_TAG}" .
  rules: *deploy_rules                # build only where we will deploy
```

The Complete Pipeline (2/2)

```yaml
deploy_dev:      # 3. automatic from develop
  stage: deploy_dev
  extends: .gcp_auth
  variables:
    SERVICE: cloud-demo-dev
    ENVIRONMENT: development
  script: &deploy_script
    - |
      gcloud run deploy "$SERVICE" \
        --image="${IMAGE_URL}:${IMAGE_TAG}" \
        --region="$REGION" \
        --service-account="$GCP_RUNTIME_SA" \
        --set-env-vars="ENVIRONMENT=${ENVIRONMENT},GIT_SHA=${CI_COMMIT_SHORT_SHA}" \
        --allow-unauthenticated --quiet
    - URL=$(gcloud run services describe "$SERVICE"
        --region="$REGION" --format='value(status.url)')
    - echo "DEPLOY_URL=$URL" > deploy.env
    # Smoke test: deploy only "counts" if it answers
    - curl --fail --silent --show-error "$URL/health"
    - curl --fail --silent --show-error "$URL/version"
  artifacts:
    reports: { dotenv: deploy.env }
  environment: { name: development, url: $DEPLOY_URL }
  rules:
    - if: $CI_COMMIT_BRANCH == "develop"
```

```yaml
deploy_prod:     # 4. same image, same script,
                 #    different configuration
  stage: deploy_prod
  extends: .gcp_auth
  variables:
    SERVICE: cloud-demo-prod
    ENVIRONMENT: production
  script: *deploy_script
  artifacts:
    reports: { dotenv: deploy.env }
  environment: { name: production, url: $DEPLOY_URL }
  resource_group: production   # one at a time
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
      when: manual
```

Dev and prod run **literally the same commands**; only the job `variables` differ. The environment idea, **enforced by syntax**.

Reading Guide

- `.gcp_auth` + `extends` — dot-keys are **hidden** (never run); `extends` merges the template in. Auth written once, reused three times.
- **YAML anchors** `&name` / `*name` — `&deploy_script` labels the dev script; `*deploy_script` pastes it into prod.
- `gcloud builds submit` — builds the Dockerfile on Google's machines. Used instead of Docker-in-Docker, which needs **privileged** runners many shared (university) runners forbid. (`ci-alternatives/build-dind.gitlab-ci.yml` shows DinD.)
- **dotenv artifact** — `DEPLOY_URL=…` in a `dotenv` report becomes a variable for `environment:url`, so Environments links to the real service.
- **Smoke test** — `curl --fail` exits non-zero on HTTP error → job fails. A deploy that doesn't answer `/health` **is a failed deploy**.
- **Why** `test` **everywhere but** `build` **only on** `develop` / `main`**?** Tests are cheap; images for every feature branch cost money for images nobody deploys.

Tracing One Change End to End DEMO

```
git checkout develop
# edit app.py: APP_VERSION = "2.0"
git commit -am "Bump version to 2.0" && git push origin develop
```

Pipeline #1043 on `develop`, commit `7c41d0aa`:

Time	Job	What happens	Visible result
0:00	—	rules create `test`, `build`, `deploy_dev`	graph: 3 jobs
0:05	test	python:3.12-slim, install deps, pytest	12 passed; JUnit report
0:40	build	OIDC → STS → deployer SA; `gcloud builds submit`	`…/cloud-demo:7c41d0aa` in registry
2:10	deploy_dev	new token; `gcloud run deploy cloud-demo-dev`	revision `-00007` serving 100%
2:40	deploy_dev	`curl /health` and `/version`	`{"version":"2.0","commit":"7c41d0aa",…}`
2:41	—	dotenv sets `DEPLOY_URL`	Environments: development @ 7c41d0aa

`deploy_prod` does **not** appear — its rule only matches `main`.

From Develop to Production

1. Team checks the **dev URL**
2. Open a merge request `develop → main` — MR shows the green pipeline and test report
3. Reviewer approves; it merges

Pipeline #1044 on `main`:

```
test ✓ ──▶ build ✓ ──▶ deploy_prod ▶  (pipeline: blocked)
                              │
                  authorized maintainer clicks Run
                              ▼
    cloud-demo-prod: new revision · smoke test ✓ · Environments: production @ new commit
```

What If a Test Fails?

Suppose the change accidentally renamed the `"message"` key:

```
test          ✗  FAILED tests/test_app.py::test_home - KeyError: 'message'
build               (not run)
deploy_dev          (not run)
```

The development service keeps running the **previous** revision. Nothing was built, nothing was deployed, and the developer got an email.

> **The pipeline's most valuable output is the deployment that didn't happen.**

Rolling Back Production

A bug slips through and reaches production. Three ways back, fastest → most principled:

1. **Cloud Run traffic switch** — move 100% to the previous revision (`gcloud run services update-traffic …`). **Seconds.** Doesn't touch Git.
2. **GitLab Rollback** — Operate → Environments → production → **Rollback**. Re-runs the old `deploy_prod` job with the old tag. **No rebuild:** the image is already in Artifact Registry.
3. **Revert the commit** — `git revert` on `main`; the normal pipeline tests, builds, and (after approval) deploys. Slowest, but **history tells the truth**.

> In practice: do **1 or 2** now to stop the bleeding, then **3** so `main` matches production again.

Testing the Pipeline Itself

The pipeline file is **code that controls production**. Test it — it's just pytest over YAML:

```
CI = yaml.safe_load((ROOT / ".gitlab-ci.yml").read_text())

def test_tests_run_before_anything_is_built():
    order = CI["stages"]
    assert order.index("test") < order.index("build") < order.index("deploy_dev")

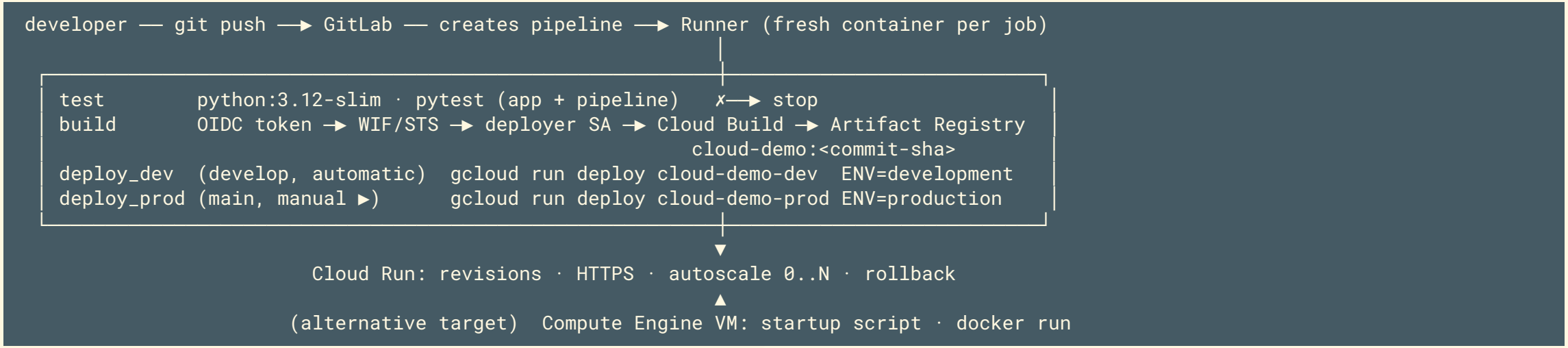
def test_images_are_tagged_with_the_commit_never_latest():
    assert ":latest" not in (ROOT / ".gitlab-ci.yml").read_text()
    assert "CI_COMMIT_SHORT_SHA" in CI["variables"]["IMAGE_TAG"]

def test_production_deploy_is_manual_and_main_only():
    rules = jobs()["deploy_prod"]["rules"]
    assert len(rules) == 1
    assert rules[0]["when"] == "manual"
    assert "CI_DEFAULT_BRANCH" in rules[0]["if"]
```

An MR that "simplifies" prod to deploy automatically turns the pipeline **red before it takes effect**.

> **Policy you care about should be a failing test, not a wiki page.**

The Complete Picture



Narrate it without looking: **Git → GitLab CI/CD → test → build → image → Artifact Registry → Cloud Run → running app.** If you can explain why each arrow exists, you understand the lecture.

The Project: `cloud-demo`

```
cloud-demo/
├── README.md              # setup, deploy, clean-up
├── app.py                 # FastAPI: /, /health, /version
├── requirements.txt       # runtime deps (in image)
├── requirements-dev.txt   # + pytest, pyyaml (CI only)
├── Dockerfile             # python:3.12-slim, non-root
├── .gitlab-ci.yml         # test→build→deploy_dev→deploy_prod
├── tests/
│   ├── test_app.py        # endpoint tests
│   └── test_pipeline.py   # tests over .gitlab-ci.yml
├── gcp/setup.sh           # one-time: APIs, registry, SAs, WIF
├── vm/
│   ├── create-vm.sh       # create + start an instance
│   ├── startup.sh         # Docker, pull image, run
│   └── deploy-vm.sh       # new tag + reboot
└── ci-alternatives/
    ├── build-dind.gitlab-ci.yml
    └── deploy-vm.gitlab-ci.yml
```

Locally (no cloud account):

```
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements-dev.txt
pytest -v          # 12 passed
python app.py      # :8080
docker build -t cloud-demo .
docker run --rm -p 8080:8080 \
  -e ENVIRONMENT=docker cloud-demo
```

Deploy (billing required):

```
export PROJECT_ID=your-project-id
export GITLAB_PROJECT_PATH=your-group/cloud-demo
export GITLAB_ISSUER=https://gitlab.com
bash gcp/setup.sh  # prints 5 CI/CD vars
# paste into Settings → CI/CD → Variables
# protect `develop` and `main`
git checkout -b develop
git push -u origin develop
```

⚠ **Clean up:** delete both Cloud Run services, the VM, and the Artifact Registry repo — a forgotten VM is the classic surprise bill.

Common Mistakes

1. Deploying `:latest` — can't tell what's running, can't roll back to "the previous latest"
2. Deploy jobs **without a rule** — every feature branch deploys over your environment
3. Storing a **service-account JSON key** in a CI variable
4. A WIF provider with **no attribute condition** — trusting every project on GitLab.com
5. Marking deploy variables **protected**, then wondering why they're empty on unprotected `develop`
6. **Rebuilding** the image for production instead of deploying the tested one
7. Listening on a **hard-coded port** or `127.0.0.1` inside the container
8. **Stopping** a VM and assuming the bill stopped; forgetting to delete demo resources
9. Using **cache** to pass build outputs between jobs
10. Treating a green deploy job as proof it works — without a **smoke test**, it only proves `gcloud` exited 0

Key Takeaways

- **CI/CD turns the release procedure into code** — versioned, reviewed, run identically on every push. CI: "is it correct?" CD: "get it running, safely."
- **Job** = script that succeeds iff exit 0 · **stages** order jobs · **pipeline** = all jobs for one event · **runner** executes in fresh containers. `artifacts` pass files; `rules` decide whether a job exists.
- **Variables** configure the pipeline: predefined, YAML (non-secret), UI (protected for deploy creds). **Masked ≠ secret.**
- **Environments** are deployed copies; branches are only *mapped* to them. **Build once, tag with the commit, deploy the same image everywhere**, gate prod behind a manual job.
- A **VM** runs anything and makes you responsible for everything: TLS, restarts, patching, scaling, zero-downtime, a 24/7 bill.
- **Cloud Run** runs the same image with HTTPS, scale-to-zero, revisions, one-command rollback — at the price of statelessness and HTTP-only.
- **Workload Identity Federation**: short-lived OIDC token, trusted only under **your** attribute condition. **No keys, nothing to leak.**