Continuous Integration Continuous Deployment

2026-06-12 · valiant avocado Kranich

Operation Vacation.

The Why

Every deploy is still a human ssh'ing into `prod`, running `git pull`, restarting the service, and wasting a coffee break or feature development capacity. The smallest tweak goes through the same heavyweight path as a real feature, and the only gate between a push and production is the operator's attention.

This block closes that gap. A single bash script on `prod` fetches origin, runs the test suite, and restarts the service when the tests pass. A systemd timer fires the script every thirty seconds, so a green commit on `main` ships without a human touching a terminal. The end of the block introduces feature flags as the runtime switch that lets a service change behaviour without another deploy.

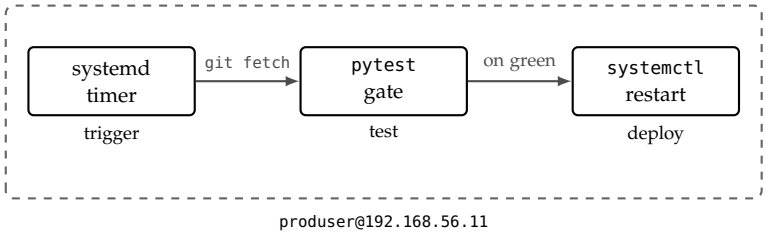

Figure 1: The classroom CI/CD cascade, all of it living on `prod`. The systemd timer is the `trigger` and fires every thirty seconds. The `test` stage fetches origin and runs `pytest` against the new commit. The `deploy` stage restarts the systemd unit, but only when the gate is green. A red test breaks the cascade before the restart.

You wrote a fix. Ran it locally on `dev`, it looks right, you push to `main`, ssh to `prod`, pull, restart, and the service is down. The fix passed your single manual check and shipped a regression that broke the service. No tests ran the existing endpoints, no guard sat between your push and `prod`, and the only person who would have noticed is you.

EVERY MANUAL STEP is a step a (tired) human will skip at some point, and every skipped step is a regression waiting to ship. Automation does not replace careful engineering: it encodes the steps careful engineers already do, so they happen every time, in the same order, at large-scale, without anyone remembering.

Hands On Experience

OUTSIDE THE CLASSROOM. Teams call this practice DevOps. The artefacts are different at scale, with Jenkins, GitLab CI, or GitHub Actions runners replacing the timer, and Kubernetes deployments replacing `systemctl restart`, but the shape is identical: one human push, an automated gate, one observable change to production.

THIS EIGHTH BLOCK introduces automation around the PixelWise pipeline. By the end you will have

- written a smoke test for `classify_batch` that gives `pytest` something to gate on,
- written an auto-deploy script on `prod` that fetches origin, runs `pytest`, and restarts the service on green,
- registered a systemd timer that fires the script every thirty seconds,
- and seen how a feature flag would switch behaviour at runtime without another deploy.

THE PIPELINE IS THE ENGINEERING SURFACE. Once a pipeline runs on every push, the test suite, the deploy script, and the timer become part of how the application is developed. A good pipeline is a good engineering surface: it makes it easy to build fast and reliably. A poor pipeline will grind you to a halt, as your code base gets more complex.

CATCH UP TO `v0.6`. If `dev` or `prod` drifted from the end of Block 7, align both machines with a hard reset to the tag:

```
cd ~/pixelwise
git fetch origin --tags
git reset --hard v0.6
bash setup-server.sh
```

The full set of release tags lives at https://github.com/schutera/pixelwise/tags.

What is CI/CD?

CONTINUOUS INTEGRATION, CI, is the practice of merging every change into a shared branch as soon as it passes an automated gate. The gate is a test suite that rejects broken behaviour, run on a clean machine so the result is reproducible. Green is safe to merge; red does not enter `main`.

CONTINUOUS DEPLOYMENT, CD, extends the chain by one step. The same gate that admits a commit to `main` also ships it to production, with no human between the merge and the restart. The unit of release shrinks from "a quarter of features" to "every green commit on `main`", which makes each release small enough to reason about and small enough to revert.

THREE PIECES shape every CI/CD pipeline, no matter how elaborate the tool around them. A trigger says when to run. A gate says whether to proceed. A deploy step says what to do when the gate is green.

THE INDUSTRY SHAPE. GitHub Actions, Jenkins, and GitLab CI are the cloud-hosted version of the same idea. A push triggers a workflow on a fresh VM the platform provisions for the run. The workflow installs dependencies, runs tests, and on green opens an SSH session to production and ships the code. The configuration lives in YAML, the runner lives somewhere else, and usually you pay for this beyond free tiers.

THE CLASSROOM SHAPE. A single bash script on `prod` does the work. It runs `git fetch origin`, compares the local commit hash against `origin/main`, and exits silently if they match. If `main` has advanced, it pulls, runs `pytest`, and restarts the systemd unit only when the tests pass. A systemd timer fires the script every thirty seconds, so any commit that lands on `main` reaches `prod` before you have switched back to the browser to check.

WHAT THIS IS NOT. This block does not build a GitHub Actions workflow, does not stand up a self-hosted runner, and does not wire a webhook from the forge into the VM. Those are real and worthwhile patterns, and the references in this section's margin notes describe them; the chapter teaches the substance underneath them. In the industry this substance is usually wrapped in a hosted tool, which you will easily adapt to after you understand the shape here.

DEPLOY GATE, NOT CI GATE. The `pytest` step in this chapter runs on `prod` after the push has already merged into `main`. It gates the restart, not the merge: a red test keeps the service on the previous green commit, but `main` already carries the break and any developer pulling `main` pulls the break too. A CI gate runs the same test on a pull request before the merge, so `main` stays green by construction. We omit this step for simplicity and to keep the scope manageable in this lecture block.

BARE-METAL FIRST. The classroom builds the pipeline from scratch on the same machine that hosts the service. The trigger is a systemd timer, the gate is `pytest` on `prod`, the deploy step is one bash script. The substance is identical to a GitHub Actions workflow; the difference is that every part of it is on disk in front of you, debuggable with `journalctl`, and free.

Exercise: The Gate or a Smoke Test

THE GATE NEEDS SOMETHING TO GATE ON. A smoke test proves the wiring works without claiming anything about correctness. The name comes from electronics: turn the power on and see whether smoke comes out. A smoke test that fails means the code does not even run; a smoke test that passes does not mean the code is right, only that it runs, something we can work with.

WHERE TESTS LIVE. The convention in Python is a `tests/` directory at the repository root, with files named `test_*.py` and functions named `test_*`. `pytest` discovers them automatically; no registration, no configuration. The auto-deploy script in the next exercise runs `python -m pytest tests/` from `/opt/pixelwise` and refuses to deploy on any non-zero exit code.

GO DEEPER LATER. The `pytest` documentation at `https://docs.pytest.org/` is the canonical reference. For a book-length treatment, Harry Percival's "Test-Driven Development with Python" is free online at `https://www.obeythetestinggoat.com/`. A second testing block belongs in a follow-on course.

CREATE `tests/test_classifier.py` on `dev` with a smoke test for `classify_batch`. The test builds a batch of two blank images, passes them through the classifier, and asserts the response shape:

```
cd ~/pixelwise
mkdir -p tests
nano tests/test_classifier.py
```

```
# tests/test_classifier.py
import numpy as np
from app.classifier import classify_batch


def test_classify_batch_returns_one_per_image():
    images = np.zeros((2, 28, 28), dtype=np.uint8)
    results = classify_batch(images)
    assert len(results) == 2


def test_classify_batch_result_shape():
    images = np.zeros((1, 28, 28), dtype=np.uint8)
    result = classify_batch(images)[0]
    assert "prediction" in result
    assert "confidence" in result
    assert isinstance(result["confidence"], float)
```

TWO FUNCTIONS, ONE FILE. The first function checks the batch shape: a batch of two images in, two results out. The second function checks the result keys: every result is a dict with `prediction` and `confidence`. Together they prove the function loads, accepts the expected input, and returns the expected shape.

INSTALL `pytest` AND RUN THE TEST LOCALLY ON `dev` before pushing, from the repository root so `pytest` resolves the `app` package

against the on-disk source tree:

```
cd ~/pixelwise
source .venv/bin/activate
pip install pytest
python -m pytest tests/
```

Same gate, different trigger. The same `python -m pytest tests/` invocation that you just did locally and we will gate deployment later is exactly what a CI runner would also execute on every pull request before merging into `main`.

Did it take on `dev`?

`python -m pytest tests/` should print two passes. If `pytest` reports an import error like `ModuleNotFoundError: No module named 'app'`, you ran the command from the wrong directory; `cd` into `~/pixelwise` and try again. If `classify_batch` itself fails to import, the model file is missing on `dev`; rerun `bash setup-server.sh`.

Commit and push the test. The next exercise builds the auto-deploy script that runs the same test on `prod` and gates the restart on it.

Stage and ship.

```
git add tests/
git commit -m "Add smoke
  test for classify_batch"
git push
```

Auto-Deploy on `prod`

THE PIPELINE IS ONE BASH SCRIPT. The script lives in the repository and runs on `prod`. Its body is the trigger, the gate, and the deploy step laid out in order: fetch from origin, compare local against remote, and either exit or pull, install, test, and restart. A handful of operations, one shell file, version controlled like everything else in the project.

ONE SCREEN, ONE HEAD. A pipeline that fits on one screen fits in one head. The deploy script is short enough to read top to bottom, the trigger is a single timer unit, and the gate is one `pytest` call. Everything else is an optimisation.

A COMMIT HASH COMPARISON IS THE TRIGGER. The script asks Git for the local commit hash on `main` and the remote commit hash after a fresh fetch. Equal hashes mean nothing changed since the last run and the script exits cheaply. Different hashes mean a new commit landed upstream and the rest of the script runs. The commit hash is the cheapest test for "is there work to do".

THE GATE IS `pytest`. After the pull, the script runs the test suite against the new commit. A non-zero exit code aborts the script before the restart. The previous service keeps running on the old commit, the operator sees the failure in the system log, and the next run will retry whatever the developer pushes to fix it.

IDEMPOTENCE AND THE TIMER. The script is safe to run every thirty seconds forever. On every run where origin has not moved, it does the cheap fetch and exits. On every run where the tests fail, it leaves the running service untouched. The deploy is the side effect, not the schedule; a timer that fires once per second would do the same work and reach the same state, only louder.

THE TRIGGER IS A SYSTEMD TIMER. A service unit runs the script as a regular user; a timer unit activates the service every thirty seconds. The system survives reboots because the timer is wired into the boot target. Cron would do the same job with one line of a crontab; pick cron when systemd is not available, pick a systemd timer when it is. The substance is the same scheduler running the same script.

PUSHING TO `main` IS SHIPPING. With auto-deploy live, every commit that lands on `main` restarts `prod` within seconds. A solo classroom workflow pushes straight to `main` because the operator is also the reviewer; a team workflow does not. The team shape is four steps:

- `branch` off `main` for the change, so `main` stays untouched while you work.
- `commit` on the branch as often as needed; nothing reaches `prod` yet.
- Open a `PR` so the diff, the CI run, and a reviewer meet before the merge.
- `merge` into `main` only on green CI and an approving review, at which point the pipeline ships it.

With auto-deploy on, the merge is the deploy; the `PR` is the last place to catch a regression cheap.

TWO THINGS THIS DESIGN INTENTIONALLY LACKS. The timer pulls, so there is no push notification from GitHub; a new commit waits for the next tick. The script is local, so there is no central dashboard of recent runs; the system journal is the dashboard. Both gaps trade visibility for simplicity, and the trade is the point: at this stage of the course, the goal is the substance underneath the dashboard, not the dashboard - you usually get that as free lunch anyways, but it will taste better now that you know how it was prepared.

Exercise: Build and Activate the Pipeline

STEP 1. THE TRIGGER. On `dev`, author the systemd service unit and the timer unit. Both files live in the repository and ship to `prod` on the next pull; systemd on `prod` will read them once the provisioner in Step 3 copies them into `/etc/systemd/system/`. The service unit runs the script once, as `produser`, from the working directory `/opt/pixelwise`; the timer activates that service fifteen seconds after boot and then every thirty seconds:

```
cd ~/pixelwise
mkdir -p deploy/systemd
nano deploy/systemd/pixelwise-deploy.service
```

```
[Unit]
Description=PixelWise auto-deploy
After=network-online.target

[Service]
Type=oneshot
User=produser
WorkingDirectory=/opt/pixelwise
ExecStart=/opt/pixelwise/deploy/auto-deploy.sh
```

`Type=oneshot`. A service unit usually models a long-running daemon; the manager keeps the process alive and restarts it on exit. A oneshot unit models a job that runs to completion: systemd waits for the script to exit, records the exit code, and goes back to sleep. That is exactly what an auto-deploy script is.

```
nano deploy/systemd/pixelwise-deploy.timer
```

```
[Unit]
Description=Run PixelWise auto-deploy

[Timer]
OnBootSec=15s
OnUnitActiveSec=30s
Unit=pixelwise-deploy.service

[Install]
WantedBy=timers.target
```

`OnBootSec` vs `OnUnitActiveSec`. `OnBootSec=15s` fires the first run fifteen seconds after boot; `OnUnitActiveSec=30s` fires each subsequent run thirty seconds after the previous one finished. The pair gives you a quick first check after every reboot and a tight cadence in between, so a push to `main` reaches `prod` before you have switched windows. systemd accepts plain second values here, and a tick that does nothing but `git fetch` costs the VM nothing; production deployments tend to be slower because the gate, not the timer, sets the floor.

STEP 2. THE DEPLOY STEP. Still on `dev`, author `deploy/auto-deploy.sh`. The file ships through Git to `prod`, where the timer from Step 1 will execute it; that is why the script's first line is `cd /opt/pixelwise`, the path it will sit at on `prod`, not the `~/pixelwise` you are editing it in. The script is the whole pipeline: fetch origin, compare local commit hash against `origin/main`, conditionally pull, install pinned dependencies, gate on `pytest`, restart the service on green:

```
mkdir -p deploy
nano deploy/auto-deploy.sh
```

```
#!/bin/bash
# deploy/auto-deploy.sh
# Authored on dev, executed on prod by the timer.
# Pull origin/main, gate on pytest, restart
# the service on green.
set -euo pipefail

cd /opt/pixelwise

git fetch origin

LOCAL=$(git rev-parse HEAD)
REMOTE=$(git rev-parse origin/main)
if [ "$LOCAL" = "$REMOTE" ]; then
    exit 0
fi

echo "New commit on main: $REMOTE"
git pull origin main

source .venv/bin/activate
pip install -r requirements.txt > /dev/null

if ! python -m pytest tests/; then
    echo "Tests failed, refusing to deploy."
    exit 1
fi

sudo systemctl restart pixelwise
echo "Deployed: $REMOTE"
```

`set -euo pipefail`. `-e` exits on any non-zero command, `-u` errors on unset variables, `-o pipefail` fails the whole pipeline if any stage fails. A silent failure inside a scheduled deploy is the worst failure mode a pipeline can have.

WHY `sudo systemctl restart`? `produser` owns the code but not the service unit; restarting a system-level unit needs root. `setup-server.sh` drops a one-line sudoers rule into `/etc/sudoers.d/pixelwise` that grants `produser` a passwordless `sudo` entry for exactly this one command and nothing else:

```
produser ALL=(root) NOPASSWD: \
  /usr/bin/systemctl restart
  pixelwise
```

The script can do its one privileged step without holding broad privilege. If the rule is missing, the journal will show a password prompt and `exit 1` on the next deploy; re-running `setup-server.sh` reinstalls it.

Mark the script executable. The executable bit travels with the file in Git, so `prod` gets it on pull without a second `chmod`:

```
chmod +x deploy/auto-deploy.sh
```

STEP 3. ACTIVATE THE PIPELINE. Still on `dev`, teach `setup-server.sh` to install both unit files and enable the timer. Append a prod-only stanza so a fresh `prod` VM picks up the service unit, the timer unit, and the enabled state without any manual steps:

```
nano setup-server.sh
```

```
# Install the auto-deploy systemd timer on prod
if [ -f "$SCRIPT_DIR/deploy/systemd/pixelwise-deploy.timer" ] \
   && command -v systemctl >/dev/null 2>&1 \
   && id produser >/dev/null 2>&1; then
    sudo cp "$SCRIPT_DIR/deploy/systemd/pixelwise-deploy.service" \
        /etc/systemd/system/pixelwise-deploy.service
    sudo cp "$SCRIPT_DIR/deploy/systemd/pixelwise-deploy.timer" \
        /etc/systemd/system/pixelwise-deploy.timer
    sudo systemctl daemon-reload
    sudo systemctl enable --now pixelwise-deploy.timer
fi
```

Stage the script, both unit files, and the updated `setup-server.sh`, then push to `origin`. Switch to `prod` for the last manual deploy the chapter needs. SSH in, pull the new commit, and re-run the provisioner so the timer and the service unit land in `/etc/systemd/system/` and the timer starts ticking. After this run the timer takes over and no later step in the chapter touches `prod` by hand:

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git pull
bash setup-server.sh
```

Watch the pipeline live. On `prod`, in a second shell, tail the journal for the deploy unit. Every run prints either a single `exit 0` line on a no-op fetch, the new commit hash on a real deploy, or the `pytest` output on a red run:

```
journalctl -u pixelwise-deploy -f
```

Push a trivial commit from `dev` to see the end-to-end loop. A whitespace change to the `README` is enough to advance the commit hash on `main`. The next tick of the timer picks it up, runs the tests, restarts the service, and prints the new commit hash into the journal:

```
cd ~/pixelwise
echo "" >> README.md
git commit -am "Trivial commit to trigger auto-deploy"
git push
```

STAGE AND SHIP. `git add deploy/ \ setup-server.sh git commit -m "Add auto-deploy script and timer" git push`

DID THE TIMER INSTALL?

On `prod`:

```
systemctl list-timers | grep pixelwise
```

Should print one line with the next firing time, the previous run, and the timer unit name. If the line is missing, `systemctl status pixelwise-deploy.timer` usually shows the reason: a typo in the unit file, the service unit not found, or the timer not enabled.

READ THE JOURNAL LIKE A LOG. `-u` narrows the journal to one unit. `-f` follows it like `tail -f`, printing every new line as the timer fires. A pipeline you cannot read is a pipeline you cannot trust; `journalctl` is the dashboard the timer comes with for free.

DID THE COMMIT LAND ON `main`?

After the next tick, the journal on `prod` should carry your new hash:

```
journalctl -u pixelwise-deploy -n 5
```

A `Deployed: <hash>` line that matches the hash you just pushed means the pipeline did its job; no such line, or an older hash, is where the chain broke. Of course you can also just check the change in the readme.

Optional Exercise: Notify Discord on Deploy

A WEBHOOK IS AN HTTP POST TO A KNOWN URL. The pipeline runs in silence; the only signal of a deploy is a fresh commit hash in `journalctl`. A Discord webhook makes it audible: one `curl` call appends to a chat channel every time the script restarts the service or refuses to. Create one in the target channel under Integrations, Webhooks, "New Webhook", then "Copy Webhook URL".

ADD THE URL TO `.env` ON `prod` alongside the other runtime secrets, then edit `auto-deploy.sh` on `prod` to source `.env` near the top and to `curl` the webhook in the `pytest` failure branch and after the `systemctl restart`:

THE URL IS THE SECRET. A Discord webhook URL ends in a long opaque token; anyone with the URL can post to the channel. Treat it like a password: keep it off GitHub and out of screenshots. The variable does not land in `.env.example`; each operator pastes their own URL into `.env` on the machine that needs it.

```
# /opt/pixelwise/.env
DISCORD_WEBHOOK_URL="https://discord.com/api/..."

# after `cd /opt/pixelwise`
if [ -f .env ]; then
    set -a; source .env; set +a
fi

# inside the pytest failure branch
curl -sf -H "Content-Type: application/json" \
    -d "{\"content\":\"Deploy red: $REMOTE\"}" \
    "$DISCORD_WEBHOOK_URL" > /dev/null || true

# after `sudo systemctl restart pixelwise`
curl -sf -H "Content-Type: application/json" \
    -d "{\"content\":\"Deployed: $REMOTE\"}" \
    "$DISCORD_WEBHOOK_URL" > /dev/null || true
```

Run the script once by hand to see the next message land in the channel:

```
sudo -u produser bash \
    /opt/pixelwise/deploy/auto-deploy.sh
```

WHY `|| true`? `set -e` aborts the script on any non-zero exit. A flaky Discord call should not roll back a successful deploy or hide a failed test; `|| true` swallows the notification's exit so the rest of the script keeps going. Notifications are useful, not load-bearing.

NOT PART OF `v0.7`. The canonical `auto-deploy.sh` at `v0.7` does not include the notification. Push it to your own `main` if you want to keep it; just know it is your local addition and the reference tag stays clean.

Optional: Feature Flags

A FEATURE FLAG IS A RUNTIME SWITCH. The code path exists in the binary, but a check at the start of the path decides whether to take it. The check reads a boolean from `.env`, an environment variable, or a row in a config table. Flipping the flag changes behaviour without changing code, without a deploy, and without a build.

A DEPLOY SHIPS CODE; A FLAG SHIPS BEHAVIOUR. Decoupling the two is the single most useful primitive in modern service operation: a broken feature turns off in seconds without a rollback, a risky migration runs dark for a day before going live, and a code path can land on `main`, pass CI, sit in production unused, and switch on only when the dashboards are ready to watch what it does. A commented-out path cannot do any of that: it is invisible to the test suite, the type checker, and any reviewer searching for usages, where a flag-gated path is real code that compiles, runs in CI, and lights up the moment the flag flips.

USE CASES IN THE WILD cover more shapes of release than the on/off picture suggests:

- Progressive rollout flips a path on for a fraction of traffic first.
- Dark launch deploys code before the user-visible surface is ready.
- A kill switch keeps a wrong path gated until the fix ships.
- An A/B experiment runs two arms in parallel and keeps the winner.
- A targeted flag fires only for one team's accounts so the new path runs in real traffic before a wider audience sees it.

HOW A FLAG IS READ matters more than what it contains. A flag read once at import time survives until the next restart; a flag read on every request turns on the moment the operator edits `.env` and the next request comes in. Small services read on every request; large services cache the read with a short refresh to keep the hot path fast.

THE SHAPE OF THE FLAG. A simple flag is a boolean. A multi-arm flag is a string from a known set, like `v1` vs `v2`. A percentage flag rolls a feature out to a fraction of traffic. A targeted flag fires for one user, one cohort, one region. The boolean is the floor; the others compose from it.

```
if NEW_UI: render_v2()
if MODEL == "v2":  ...
if random() < 0.1:  ...
if user in COHORT: ...
```

DARK LAUNCH IN DETAIL. The code ships to production, runs against real traffic, and stays invisible to users because the button, page, or endpoint that would expose it is gated off. A new search backend runs on every query, returns its answers into a sink, and logs latency and error rates next to the live backend it shadows. A new pricing engine recomputes every cart in the background and writes its number to a log column the customer never sees. The point is to learn from production load before any user can notice the change: the day the surface goes live the path has already been exercised at full scale, and the team argues from a week of graphs rather than a guess.

OUTSIDE THE CLASSROOM. Production teams use hosted flag systems like LaunchDarkly, Unleash, or Statsig that store flags out of band and push changes to running services in seconds. A small project uses `.env` because it is the lightest implementation of the same pattern; the mechanism is identical.

Tag `v0.7`

THE PIPELINE IS THE ARTEFACT. The end of every block is a tagged, runnable state. Stage every file the block added or changed, commit, push, tag, and push the tag. Future catch-ups reset to this tag and arrive at the same place you are now:

```
cd ~/pixelwise
git add tests/ \
    deploy/auto-deploy.sh \
    deploy/systemd/pixelwise-deploy.service \
    deploy/systemd/pixelwise-deploy.timer \
    setup-server.sh
git commit -m "End of Block 8: auto-deploy pipeline"
git tag -a v0.7 \
    -m "End of Block 8: auto-deploy script and timer"
git push --follow-tags
```

WHAT LANDS IN `v0.7`. `tests/test_classifier.py` with the smoke test; `deploy/auto-deploy.sh` as the pipeline; `deploy/systemd/pixelwise-deploy.service` and `deploy/systemd/pixelwise-deploy.timer` as the trigger; an extended `setup-server.sh` that installs both units. The optional Discord notification and the feature flag theory do not change the repository's tagged state.

THE PIPELINE NOW RUNS WITHOUT YOU. Every commit on `main` is tested. Every green test ships. The service is automated.

WATCH THE PIPELINE LAND ITS OWN TAG. The push that creates `v0.7` is a push to `main`, so the next tick of the timer picks it up. The script pulls the tagged commit on `prod`, runs the tests, restarts the service, and writes the new commit hash into the journal. The first time a chapter ships its own deploy is a moment worth pausing for.

CATCHING UP TO `v0.7`. If a future block finds either machine drifted from this state, align both with a hard reset to the tag and a re-run of the provisioner on `prod`. On `dev` the tag carries the test, the script, and the unit files into the working tree:

```
cd ~/pixelwise
git fetch origin --tags
git reset --hard v0.7
```

On `prod` the same reset is followed by the provisioner so the timer and the deploy service land in `/etc/systemd/` and the next tick of the timer picks the tag up:

```
ssh produser@192.168.56.11
cd /opt/pixelwise
git fetch origin --tags
git reset --hard v0.7
bash setup-server.sh
```

WHY `setup-server.sh` AGAIN. A bare `git reset` restores the unit files inside the repository on `prod`, but systemd reads from `/etc/systemd/system/`. The Block 8 additions to the provisioner copy the timer and the service into that directory, run `systemctl daemon-reload`, and enable the timer. Re-running the provisioner is what closes the gap between repo state and systemd state.

The full set of release tags lives at https://github.com/schutera/pixelwise/tags.

Self-Reflection and Recap

Self-Reflection questions to guide your thoughts:

- A test catches a regression on a feature branch; a user catches it in `prod`. Where in the pipeline is the cheapest place to catch the next one?
- What kinds of regression does the smoke test against `classify_batch` catch, and what kinds does it miss?
- The timer fires every thirty seconds. What property of `auto-deploy.sh` makes it safe to fire on the same commit hash again, and what breaks if that property goes away?
- Where does the classroom auto-deploy script differ from a GitHub Actions workflow that does the same job, and where is the substance identical?
- A feature flag stays `false` for three months and the code path never runs. What makes the flag-gated version safer to leave in the binary than the same code commented out?

Recap of key concepts:

- Continuous Integration runs tests on every change against a known state; Continuous Deployment ships every green commit to `main` without a human in the loop.
- The classroom pipeline is three primitives: a systemd timer as the trigger, `python -m pytest tests/` as the gate, and one bash script as the deploy step.
- The smoke test in `tests/test_classifier.py` doubles as the gate: a non-zero exit from `pytest` stops the deploy before `systemctl restart`.
- `systemctl list-timers` shows the schedule and `journalctl -u pixelwise-deploy` reads its history; swap the timer for a GitHub Actions cron and the script body barely changes.
- Feature flags gate code that lives in the binary, so flipping behaviour is a config edit and a restart instead of a deploy.

Milestone. You now own a service that ships itself. A commit to `main` flows through tests, lands on `prod` without a human in the loop, and tag `v0.7` marks the pipeline itself as the artefact.

You made it Zauberlehrling. Und nun sollen seine Geister Auch nach meinem Willen leben. Seine Wort´ und Werke Merkt ich und den Brauch, Und mit Geistesstärke Tu ich Wunder auch.