

Delivery and Environments

Chris Timperley · Monday, September 21, 2026

Administrivia

- **Homework 2 (Build Fast CI and Prevent Secret Leaks)**
  - Checkpoint B was due last night
- **Homework 3 (Build and Operate a CD Pipeline)**
  - design your pipeline → implement → demonstrate
  - we will **provide access to VMs** this week; be careful with storing credentials
  - we will **give access to your meta-repo**
- **Lab: Komodo**
  - learn how to do GitOps-like deployments

🎯 Learning Goals

- Trace changes from source to production, distinguishing continuous integration, continuous delivery, and continuous deployment
- Justify a delivery approach based on permissions, connectivity, operational effort, and visibility into results
- Explain how reconciliation and GitOps handle drift and emergency changes
- Plan promotion across isolated environments, preserving checked artifacts while varying configuration, secrets, data, and dependencies
- Verify application health and choose deployment and recovery strategies based on availability, capacity, and data compatibility

Continuous Delivery and Deployment

How does a repository change become a running release?

Continuous Integration

Integrate changes frequently; build and test them for fast feedback.

Continuous Delivery

Keep changes deployable; a human can approve release to production.

Continuous Deployment

Deploy each qualifying change to production automatically.

TinyNotes runs with Docker Compose

```yaml
# production/compose.yaml
services:
  api:
    image: ghcr.io/example/notes:1.0
    ports:
      - "127.0.0.1:3000:3000"
    environment:
      LOG_LEVEL: info
      DATABASE_URL: ${DATABASE_URL:?required}
  db:
    image: postgres:17
    environment:
      POSTGRES_DB: notes
      POSTGRES_USER: notes
      POSTGRES_PASSWORD: ${DB_PASSWORD:?required}
    volumes:
      - notes-data:/var/lib/postgresql/data
volumes:
  notes-data:
```

Image → Production currently selects `notes:1.0`.

Ports → The VM's reverse proxy reaches the API at `127.0.0.1:3000`.

Environment → Set logging and the database connection at runtime. The VM supplies secret values.

Storage → `notes-data` keeps database files when the database container is replaced.

A new release for TinyNotes is created (v1.1)

example / **notes**

Releases / 1.1

TinyNotes 1.1

Latest

`1.1` Tagged application commit

What's changed

Fix note validation and improve API error messages.

Source code (zip) Source code (tar.gz)

A release tag triggers an image build

Images are automatically created and uploaded on every release.

1. Runs tests
2. Builds the image on the CI runner
3. Uploads the image to a **registry** (e.g., **GitHub Container Registry**, DockerHub, Quay.io, Amazon ECR)

```yaml
# .github/workflows/image.yml
on: { push: { tags: ['[0-9]*'] } }
permissions: { contents: read, packages: write }
jobs:
  image:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: ./ci/test.sh
      - uses: docker/login-action@v4
        with:
          registry: ghcr.io
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
      - run: |
          IMAGE="ghcr.io/example/notes"
          IMAGE="$IMAGE:$GITHUB_REF_NAME"
          docker build -t "$IMAGE" .
          docker push "$IMAGE"
```

A new image does not update production



Registry (e.g., GHCR)

ghcr.io/example/notes

`1.0` · available

`1.1` · just uploaded ✓



Production VM

Still running notes:1.0
Real users reach this API through a reverse proxy.

The registry stores the image. A deployment must download and run it.

How should we update production?

One approach is to update the VM by hand:



01
Login
Connect to the production VM.

02
Edit
Change the image version and settings.

03
Pull
Download the selected image.

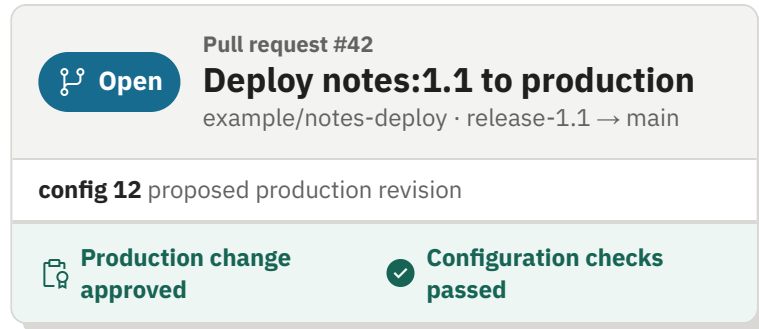
04
Restart
Run the API with the new configuration.

How would we review or reproduce this change?

Review changes in a deployment repository

A **deployment repository** (or **meta-repo**) stores production configuration in Git

- **Record:** keep the image version and settings together in a commit.
- **Review:** approve the production change through a pull request.
- **Check:** require automated configuration checks before merging.




```
 # production/compose.yaml
-    image: ghcr.io/example/notes:1.0
+    image: ghcr.io/example/notes:1.1
     environment:
-      LOG_LEVEL: info
+      LOG_LEVEL: debug
```

Apply the approved image and configuration



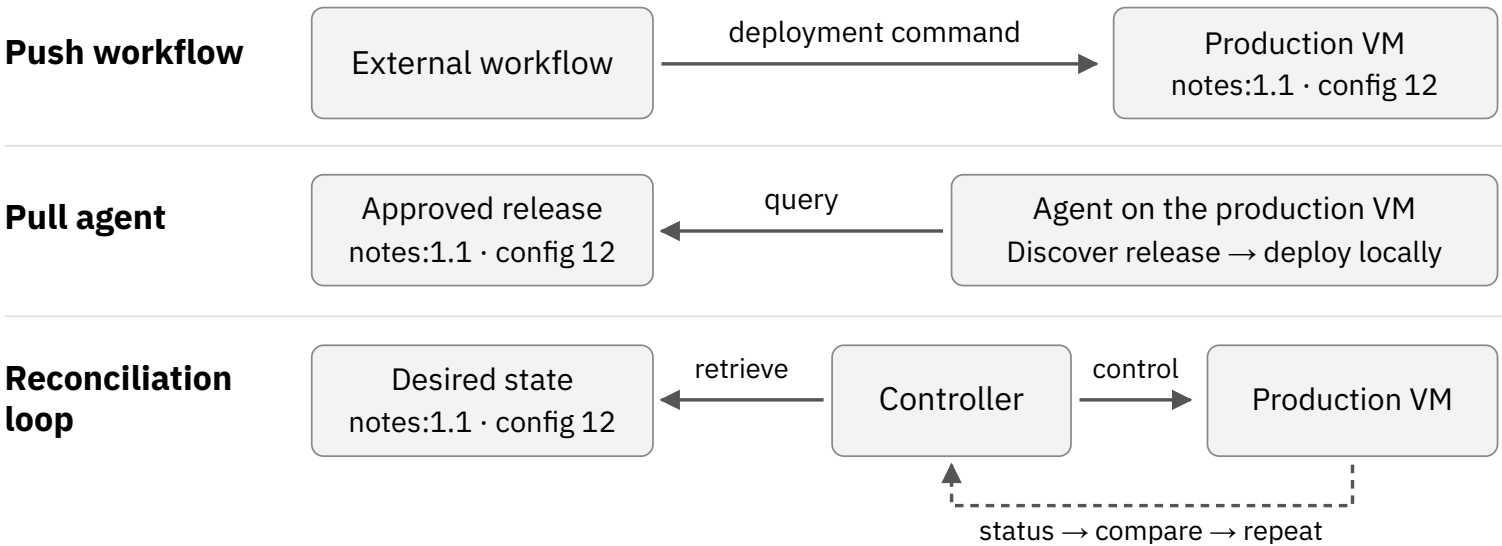
01
Approve
Merge the production PR: notes:1.1 + config 12.

02
Apply config
Deliver updated config to the VM; supply its secrets.

03
Download
The VM pulls notes:1.1 from GHCR.

04
Run
Replace the API using the selected image and config.

Three designs can deploy the same release



- All three download the image from the registry and apply production configuration

Push and Pull Deployment

Who starts the deployment, and who carries it out?

We could manually deploy changes

```
set -e
cd /opt/tiny-notes
git pull --ff-only origin production

docker compose pull api
docker compose up -d --no-build --no-deps api

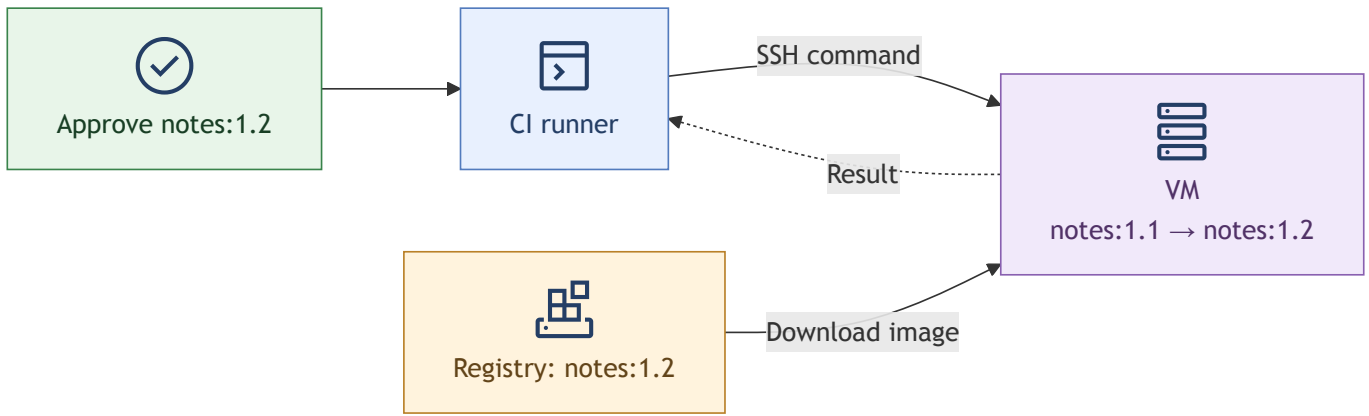
docker compose ps api
docker compose logs --tail 30 api
```

1. **Fetch configuration from Git**
2. **Download the selected notes:1.2 image**
3. **Replace the notes:1.1 API container**
4. **Inspect status and recent logs**

- The `production` branch selects the approved image in `compose.yaml`
- Secrets stay on the VM.

What could go wrong with this approach?

A push workflow initiates the VM change



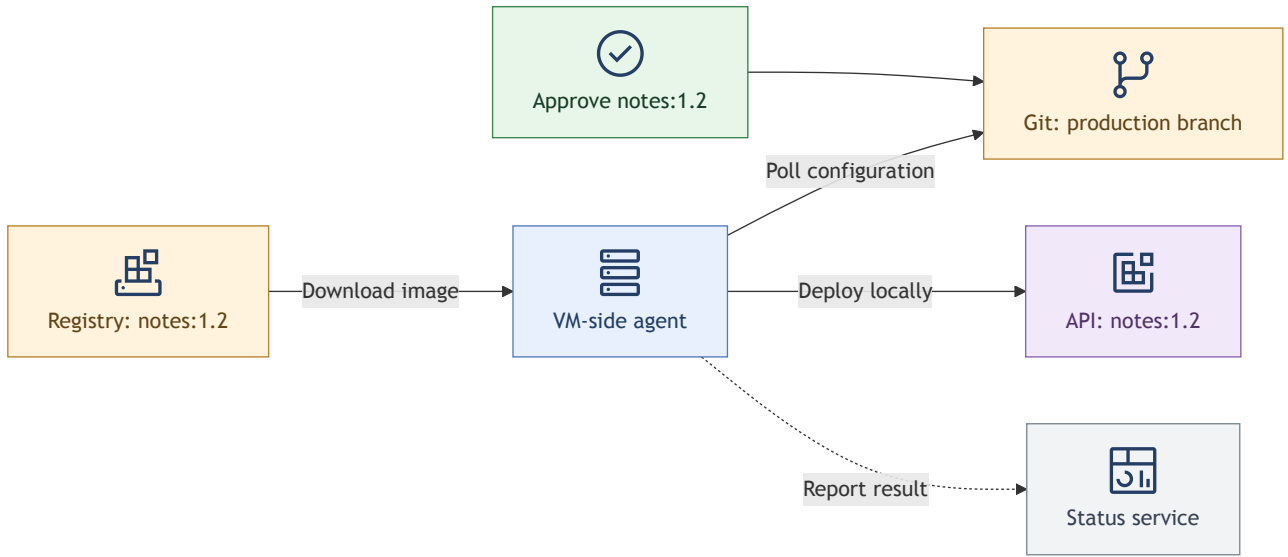
- The **runner** starts the deployment. The **VM** downloads and runs the selected image
- The runner needs SSH access to the VM
- The VM needs read access to Git and the image registry

The runner sends the commands over SSH

```yaml
# .github/workflows/deploy.yaml
on: workflow_dispatch
jobs:
  deploy:
    runs-on: [self-hosted, deployment]
    environment: production
    concurrency: notes-production
    steps:
      - name: Deploy approved configuration
        run: |
          ssh notes-prod 'bash -se' <<'REMOTE'
          cd /opt/tiny-notes
          git pull --ff-only origin production
          docker compose pull api
          docker compose up -d --no-build --no-deps api
          docker compose ps api
          REMOTE
```

- Require review for `production`.
- Set up `notes-prod`: SSH key, verified host key, and `BatchMode yes`.
- Protect the deployment repository's `production` branch.

A pull agent discovers approved releases



- The **VM-side agent** decides when to start a local deployment.
  - e.g., cron, systemd, Ansible, Komodo, ArgoCD, Flux, ...

A timer can turn our script into a pull agent

We run the same script on the VM

```
#!/bin/sh
# /usr/local/bin/pull-notes
set -e
cd /opt/tiny-notes
git pull --ff-only origin production
docker compose pull api
docker compose up -d --no-build --no-deps api
```

systemd supplies the schedule

```
# notes-deploy.service
[Service]
Type=oneshot
User=deploy
ExecStart=/usr/local/bin/pull-notes

# notes-deploy.timer
[Timer]
OnBootSec=2min
OnUnitInactiveSec=2min
```

- The VM checks again two minutes after each run finishes

Fetch a playbook and apply it on the VM

The timer runs this command

```
ansible-pull \
  -U https://github.com/ORG/notes-deploy.git \
  -C production \
  -d /opt/tiny-notes \
  -i localhost, local.yml
```

Fetch the `production` branch, then run `local.yml` on this VM.

The playbook applies Compose

```
# local.yml in notes-deploy
- hosts: localhost
  connection: local
  gather_facts: false
  tasks:
    - name: Apply the selected API image
      community.docker.docker_compose_v2:
        project_src: /opt/tiny-notes
        services: [api]
        dependencies: false
        pull: always
        build: never
        state: present
```

The shell script is enough here. Ansible becomes useful when deployment also manages **packages, user accounts, directories, and configuration files**.

	Push workflow	Pull agent
Who discovers the release?	External workflow	VM-side agent
Who initiates the VM change?	Runner sends a deployment command	Agent starts a local deployment
Who downloads the image?	VM, following the command	VM, following the agent's decision

Note: A CI runner that executes `docker compose pull` remotely still uses **push deployment**.

Deployment Platforms bring the controls together

Our script runs commands. A team also needs a place to:



Start deployments
Choose the application, release, and target server.



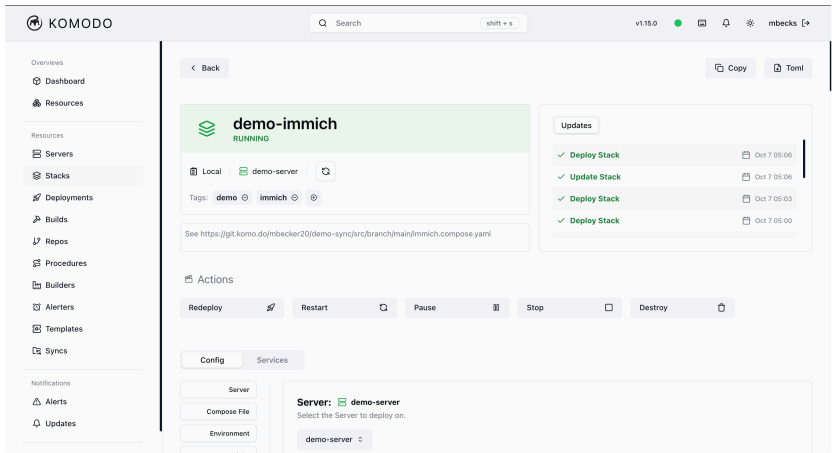
Manage access
Configure the credentials used to reach servers and registries.

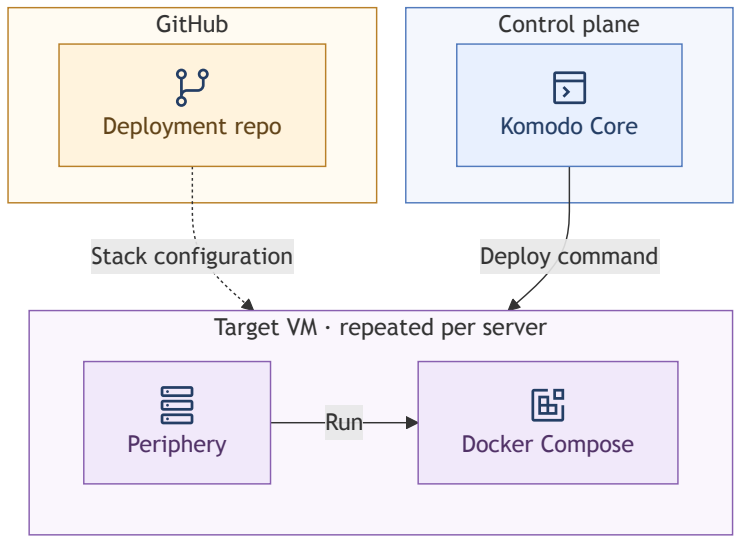


See results
Inspect deployment status and application logs.

Komodo and Coolify provide this interface for servers your team operates.

Komodo manages stacks across many servers

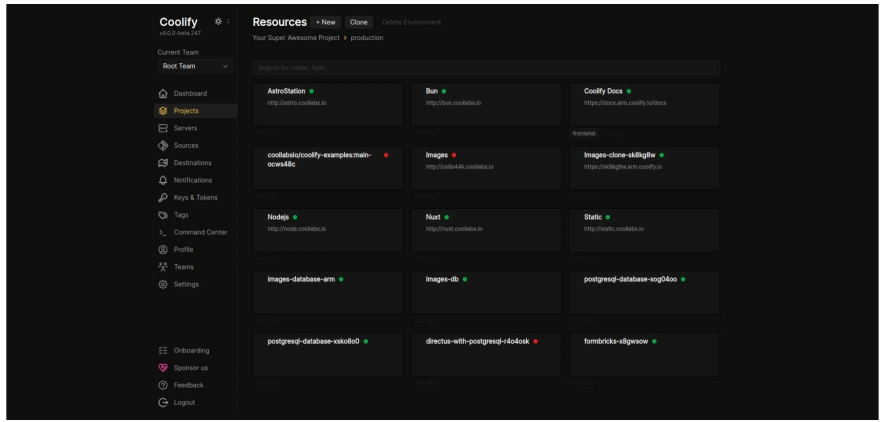


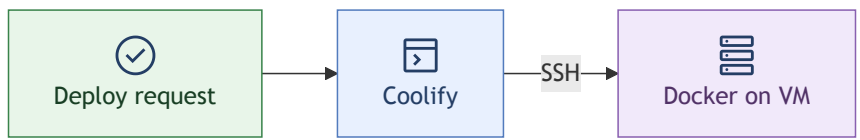


- **Core:** one dashboard for servers, stacks, and deployments.
- **Periphery:** executes Core's commands on each VM.

Coolify runs deployment commands over SSH

- Configure applications and target servers in the dashboard.
- Coolify connects to the VM over SSH and runs Docker commands.
- For Tiny Notes, select the CI-built **notes:1.2** image.





PaaS delegates infrastructure operations

Platform as a service (PaaS)

- The provider operates deployment and runtime infrastructure
- Your team selects releases, manages application settings and secrets, and checks the application
- Less infrastructure work can justify platform costs and constraints



Render



Heroku



Railway

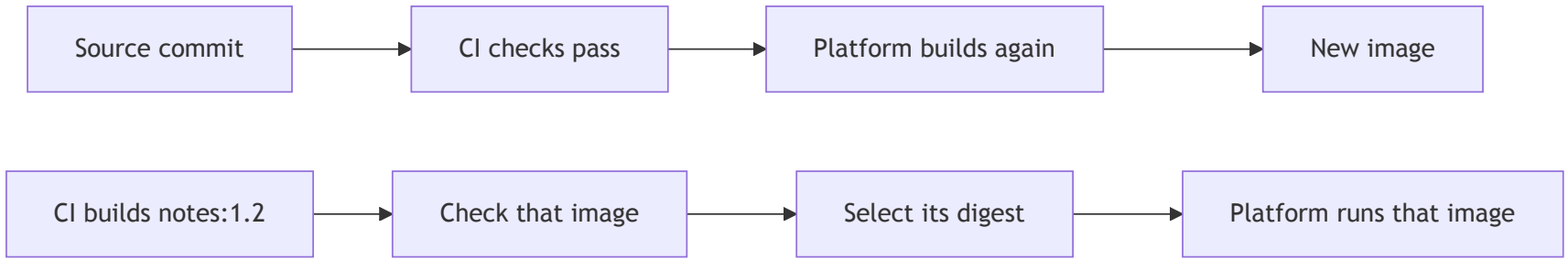


Google Cloud Run

When is running your own deployment infrastructure worth the effort?



Deploy the image that passed your checks



- A source commit identifies code. An **image digest** identifies the built artifact.
- For Tiny Notes, use the platform's **prebuilt-image** option and select the checked digest; configure secrets separately.

Can we safely run the deployment again?

The SSH connection drops before the runner receives the result. **Did the update finish?**

Idempotency: once the intended state is reached, applying the operation again makes no further change.

Operation	First attempt	Retry
Ensure notes:1.2 is running	Replace notes:1.1 if needed	Leave notes:1.2 running if already correct
Unconditionally restart the API	Restart the API	Restart and interrupt service again

⚠ The retry must inspect current state. A script or playbook does not automatically make every command or migration safe to repeat.

Push vs. Pull

Concern	Push workflow	Pull agent
Connectivity	Runner reaches VM	VM reaches Git and registry
Permissions	Runner can command VM	Agent can deploy locally
Operation	Maintain workflow; collect job results	Maintain schedule; publish results and last check time

How would you know your deployment automation had stopped working?



Which approach fits this VM?

- The VM accepts **no inbound deployment connections**. It can reach Git, the registry, and a status service.
- **Which approach fits, and how would you learn whether deployment succeeded?**



A reachable VM can use a push workflow

Setting: one Tiny Notes VM, reachable over SSH from the deployment runner

Choice: push can reuse the runner and SSH connection

- Keep the deployment key protected
- Collect command results and check the API
- No VM-side polling schedule to maintain

An outbound-only VM can use a pull agent

Setting: the VM can reach Git and the registry, but the runner cannot initiate a connection to it

Choice: pull lets the VM retrieve approved configuration and deploy locally

- Maintain the agent and its schedule
- Prevent overlapping deployment runs
- Publish results and last check time so silence is visible

Recap: drift

APPROVED CONFIGURATION

Timeout: 30 s

Recorded in Git.

LIVE CONFIGURATION

Timeout: 60 s

An operator edited the VM.



No new release was selected. **What would detect this change?**

Intended system ≠ running system

A completed run can leave drift undetected



1 · DEPLOYMENT MATCHES

30 s = 30 s

Git and the VM both use the approved timeout.



2 · OPERATOR EDITS VM

30 s → 60 s

The live timeout changes. Git still specifies 30 s.



3 · MISMATCH PERSISTS

30 s ≠ 60 s

No new release is selected. The VM still differs from Git.

A finished push job stops checking. A pull agent that acts **only when Git changes** also misses this edit.

Detecting drift requires comparing live configuration with intended configuration repeatedly.

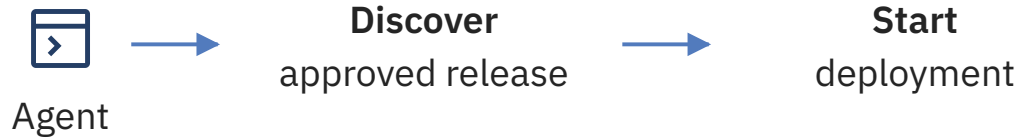
Reconciliation and GitOps

What keeps the running system aligned with its declared state?

A pull agent can also reconcile

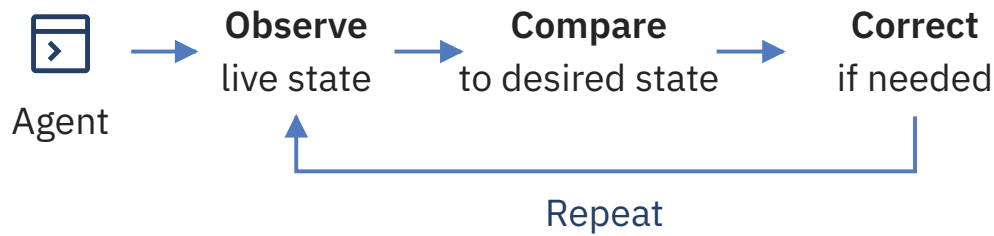
Pull deployment

Who discovers the approved release and starts deployment?



Reconciliation

What differs from desired state, and what needs correcting?



- Our timer and `ansible-pull` examples **reapply configuration even when Git is unchanged**. They can repair differences that their tasks detect and manage.

Desired state is configuration we can inspect

```yaml
# compose.yaml — API fragment
services:
  api:
    image: ghcr.io/example/notes:1.2
    environment:
      REQUEST_TIMEOUT_SECONDS: "30"
```

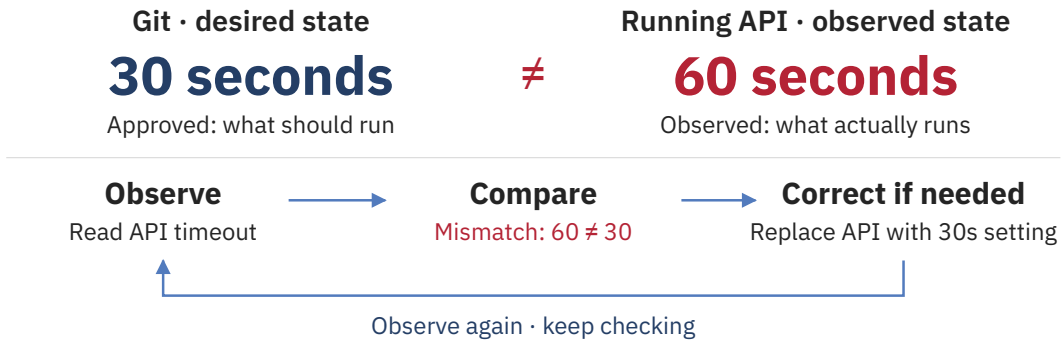
Maintain this result

Selected API image

30-second timeout

- `docker compose up -d` applies the declaration **when invoked**. It does not keep comparing this file with the container afterward.

Reconciliation repairs drift even when Git is unchanged



- Someone recreates the API container with a **60-second timeout**
- A controller repairs **only what it observes and manages**
- The same comparison can find an approved image update that has not yet deployed

GitOps Principles

v1.0.0

1 Declarative

A system managed by GitOps must have its desired state expressed declaratively.

2 Versioned and Immutable

Desired state is stored in a way that enforces immutability, versioning and retains a complete version history.

3 Pulled Automatically

Software agents automatically pull the desired state declarations from the source.

4 Continuously Reconciled

Software agents continuously observe actual system state and attempt to apply the desired state.

OpenGitOps principles

Tools apply Git-managed configuration at different scopes



Argo CD

Kubernetes

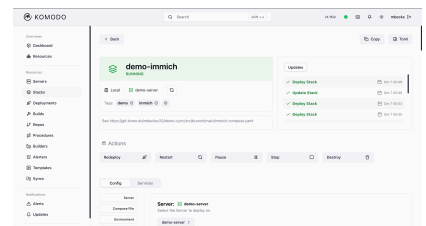
Compare and sync application resources with Git.



Flux

Kubernetes

Continuously reconcile declared resources.



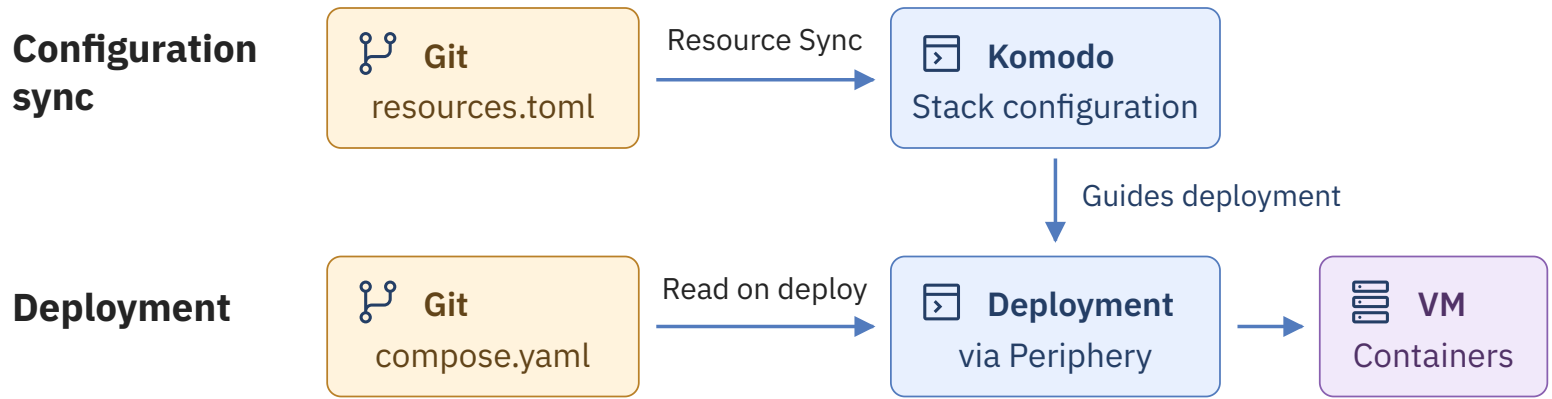
Komodo

Servers and Compose stacks

Sync platform configuration from Git and coordinate deployments.

Argo CD · Flux · Komodo

Komodo sync and container repair are separate



Resource Sync: compare Git declarations with Komodo's Stack configuration.

Deployment: apply the Stack's `compose.yaml` to containers through Periphery.

Git-backed Stacks can redeploy on pushes. **That alone does not establish repair of later live edits.**

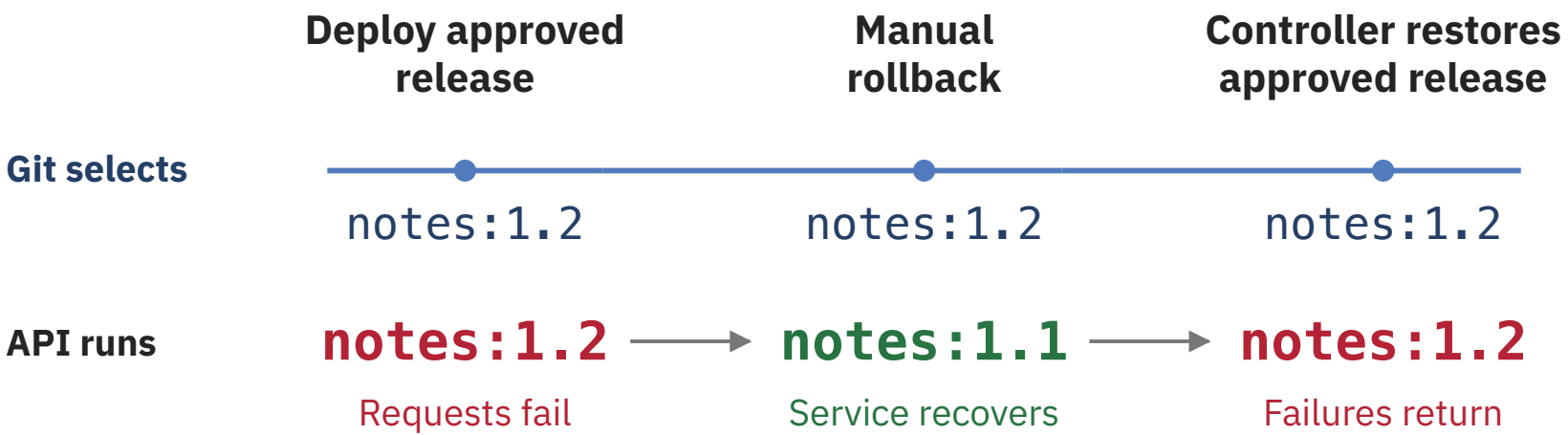
Will a later live edit be detected and corrected without a new Git revision?

Komodo: Sync Resources · Compose

When might automatically correcting a server's configuration make things worse?



⚠ Drift repair can undo an emergency rollback



- **Roll back in Git.** Otherwise, the controller restores the version Git still declares.

Use reconciliation when drift is costly



One VM with little drift

- Another controller to maintain and troubleshoot.
- Emergency changes may be undone unless you update Git or pause repair.



Many servers with recurring drift

- Repair managed settings without waiting for the next deployment.
- Keep servers consistent with less manual checking and correction.

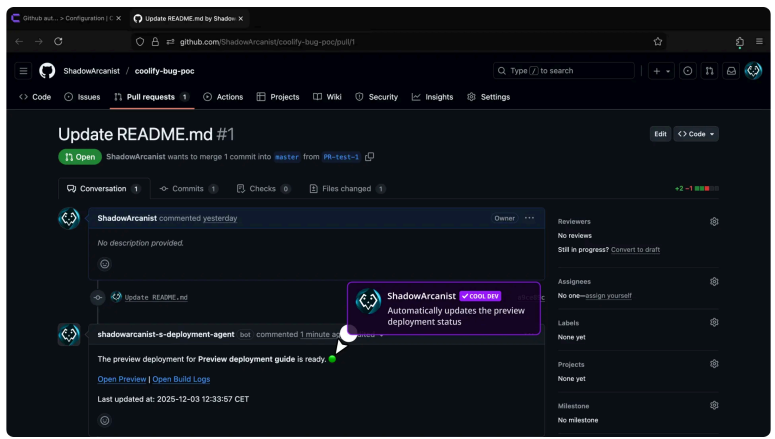
Environments and Promotion

What should we learn before moving a release into production?

Environments support different stages of delivery

An **environment** combines running software with configuration, credentials, data, and connected services.

- **Local:** get fast feedback while coding
- **Preview:** try one proposed change before integration
- **Staging:** rehearse with relevant dependencies
- **Production:** serve real users under actual operating conditions



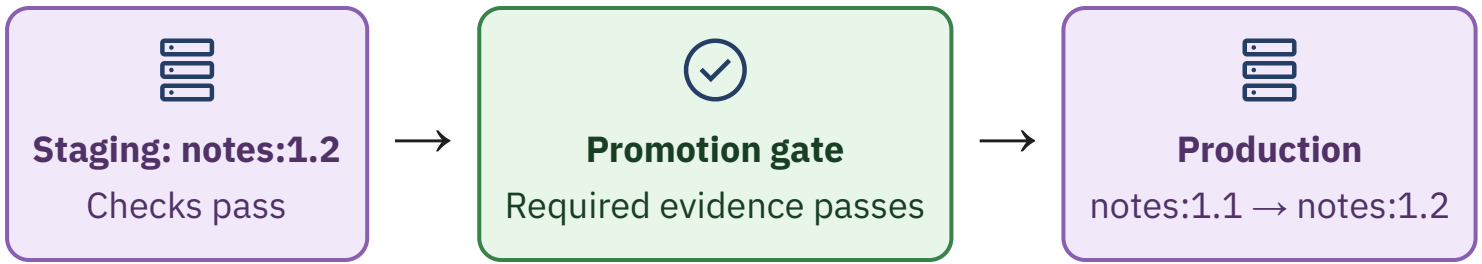
⚠ A separate URL does not isolate data



Isolation limits which resources an environment can change.

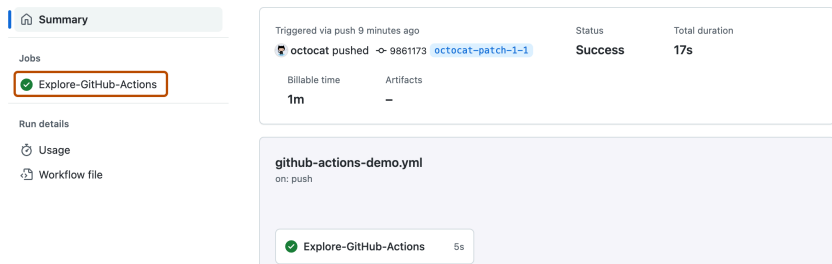
- Separate data and credentials (e.g., database); deny preview access to production.
- Use test service endpoints to avoid real notifications or other side effects.
- Preserve the behavior needed by checks while separating state.

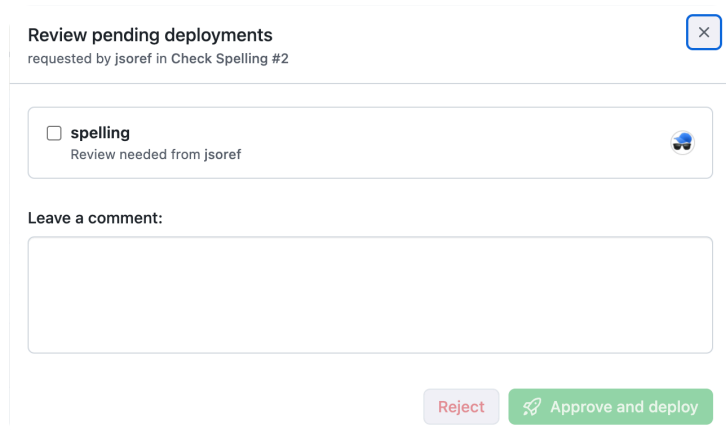
Promotion advances a checked artifact



- Preserve the checked **image digest** when selecting the next release
- Supply the target environment's **configuration and secret references**
- Use a **promotion gate** to check evidence before approving the next environment

Promotion gates can be automated or manual





Automated checks

- Tests pass for the selected image
- Settings and permissions checks pass

Human review

- Is the user impact acceptable?
- Can we recover if deployment fails?

Which production changes would you deploy automatically? Which would need human approval—and why?



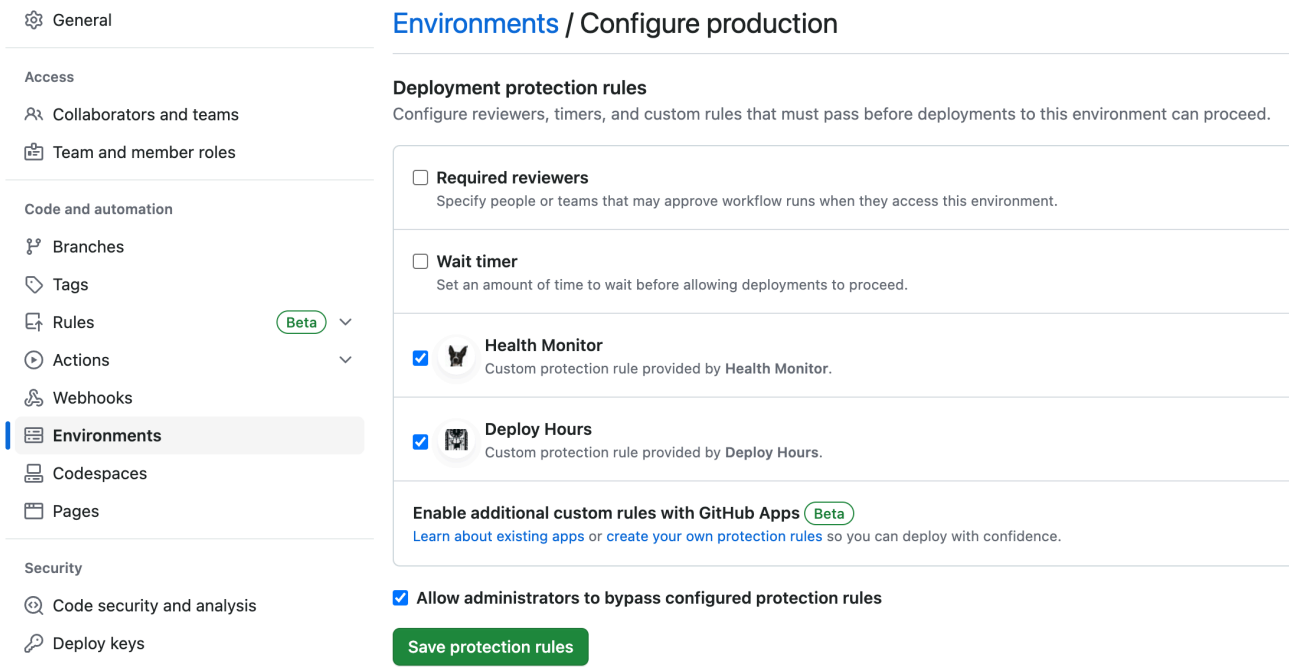
Tip: Use GitHub environments to control deployment

```yaml
# .github/workflows/deploy.yaml
on: workflow_dispatch
jobs:
  deploy:
    runs-on: [self-hosted, deployment]
    environment: production
    concurrency: notes-production
    steps:
      - name: Deploy and verify
        env:
          TARGET: ${{ vars.DEPLOY_TARGET }}
          CHECK_TOKEN: ${{ secrets.CHECK_TOKEN }}
        run: |
          ssh "$TARGET" /opt/notes/bin/deploy-approved
          /opt/notes/bin/check-api
```

- `environment`: target and protection rules.
- `vars`: SSH target setting.
- `secrets`: API check credential.

`deploy-approved`: earlier Git/Compose commands. `check-api`: API test using `CHECK_TOKEN`.

Tip: Configure the environment's rules



GitHub Changelog · April 2023

How do we store configuration for different environments?

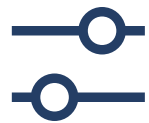
Choose where environment settings live



Git

envs/production.yaml

Review configuration changes in pull requests.



Deployment Platform

GitHub environment variables

Settings live outside the code's history.



Configuration Service

AWS AppConfig

Update settings independently of the app.

These are not mutually exclusive choices.

Give each environment its own configuration

Organize by...	Example	When it helps
Files	`staging.yaml`, `production.yaml`	One configuration file per environment.
Directories	`staging/`, `production/`	Several files per environment.
Branches	`staging`, `production` branches	Separate histories; merges must preserve environment differences.

Start with files or directories on one branch.

Tip: Share defaults; override differences

Shared defaults (`compose.yaml`)

```yaml
services:
  api:
    environment:
      LOG_LEVEL: info
      REQUEST_TIMEOUT_SECONDS: "30"
```

Overrides (`envs/staging.yaml`)

```yaml
services:
  api:
    environment:
      LOG_LEVEL: debug
```

Staging uses **debug logging** and keeps the **30-second timeout**.

```
docker compose -f compose.yaml -f envs/staging.yaml up -d
```

Configuration changes are release changes

The image stays the same, but requests can now run twice as long.

Release input	Before	After
Image digest	`sha256:8f2c…`	`sha256:8f2c…`
Configuration revision	`prod-config-27`	`prod-config-28`
Request timeout	30 seconds	60 seconds

Record the **image digest + configuration version** for each deployment

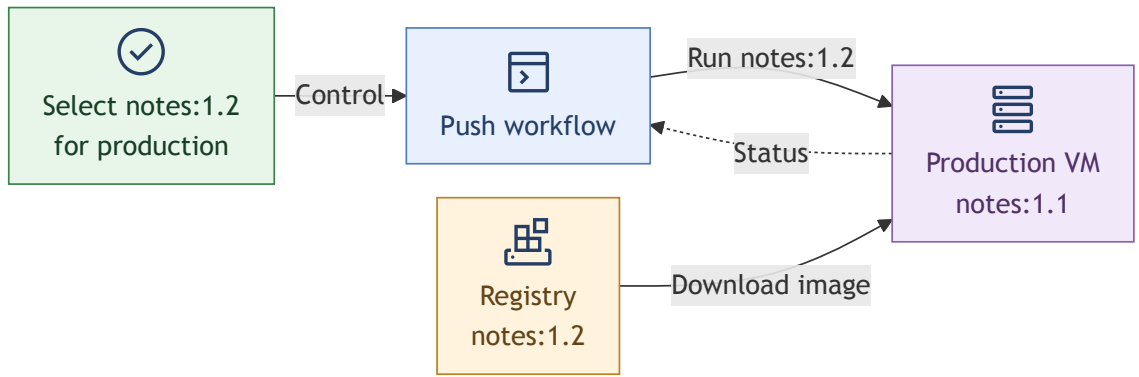
Should changing a setting go through the same review as changing code?



Deployment Strategies

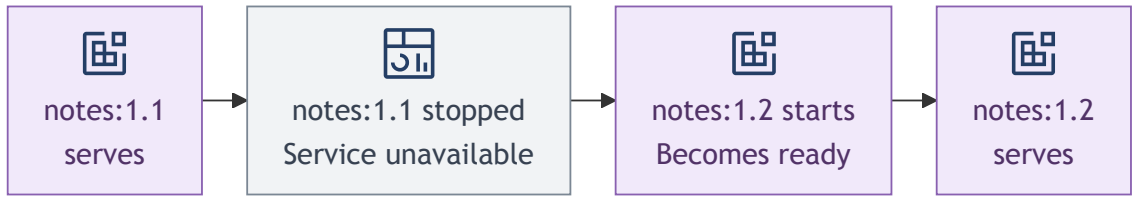
How do we replace the running version?

Initiation and replacement are separate choices



- The **replacement strategy** determines when `notes:1.1` stops, when `notes:1.2` starts, and where requests go
- Compare **interruption, spare capacity, version compatibility, and recovery speed**. Reconcilers face these choices too

In-place replacement interrupts service

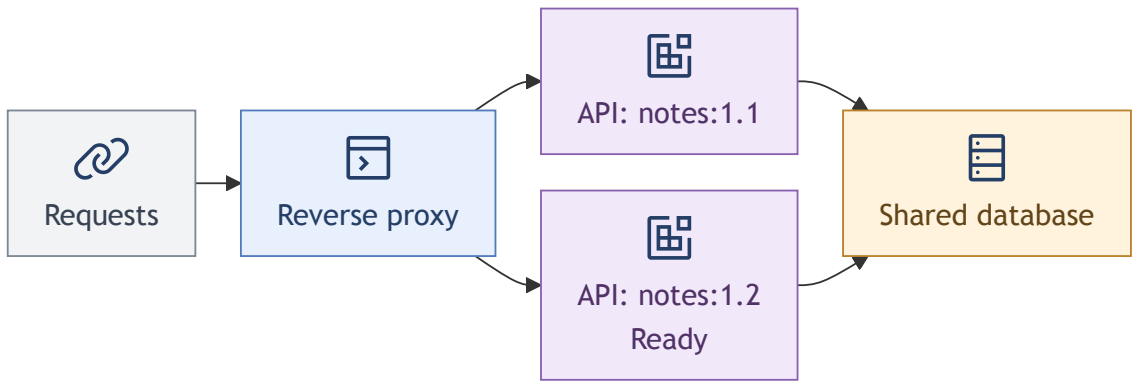


- **In-place replacement:** stop `notes:1.1`, then start `notes:1.2` in the same slot
- During the gap, the proxy has no working API
- No extra instance is needed, but service is interrupted
- If `notes:1.2` fails, redeploy `notes:1.1` and wait for startup

Suited to small internal services with limited capacity and reliable redeployment

Rolling replacement keeps instances serving

Tiny Notes now has **multiple API instances** sharing one database.



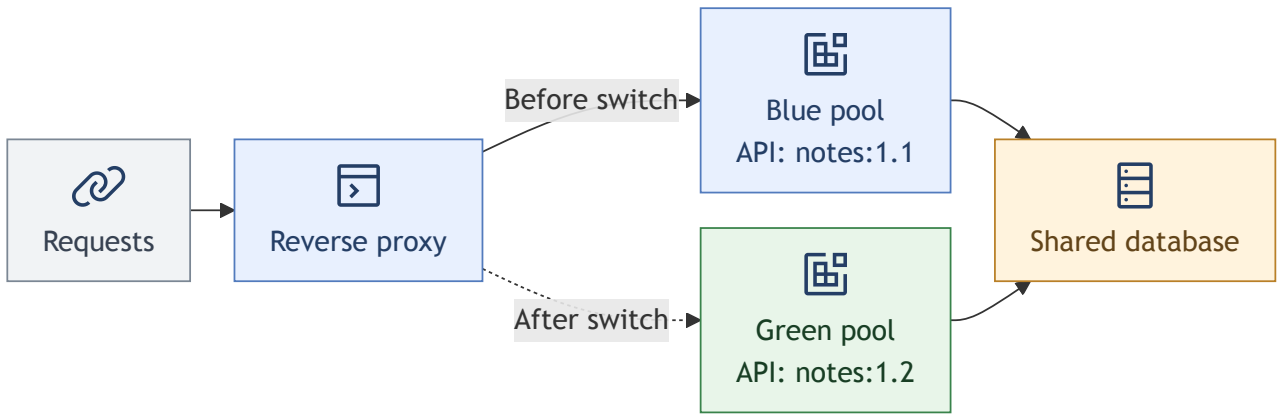
- **Rolling replacement** updates instances in batches: ready `notes:1.2` instances take traffic as `notes:1.1` instances leave. Repeat until all run `notes:1.2`

Rolling replacement needs capacity and safe overlap

Requirement	Why it matters during replacement
Enough serving capacity	Removing a `notes:1.1` instance can overload those left. Use spare capacity or limit the batch size.
Meaningful readiness checks	Sending traffic to `notes:1.2` too early can cause failures while the rollout appears to progress.
Compatible `notes:1.1` and `notes:1.2` behavior	Both versions access the same notes and database during the overlap.

Rolling replacement can avoid interruption when these conditions hold. Reversing it takes time: replace `notes:1.2` instances with `notes:1.1` and verify them again.

Blue-green verifies before switching traffic

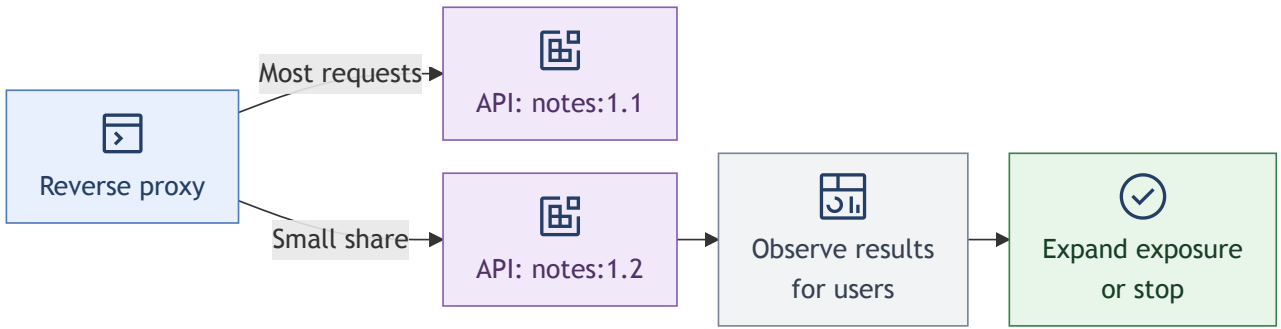


- Verify `notes:1.2` is ready while `notes:1.1` serves users, then switch traffic
- Keep both pools for quick reversal; let active requests finish
- ⚠️ **Database changes remain.** `notes:1.1` must work with the updated data

Would you always choose a deployment strategy with no downtime? Why or why not?



Canaries test before expanding exposure



- Canary send a small share of real requests to `notes:1.2`, observe results, then decide whether to expand
  - increases exposure **based on results**; gradual replacement alone does not.
- **More on this in the second half of the course**

What's the right replacement strategy?

Tiny Notes constraints	Choice and reasoning
One instance; an interruption is acceptable; little spare capacity	**In-place:** reuse the slot. Accept the service gap and the time needed to redeploy `notes:1.1` if `notes:1.2` fails.
Replicated API; service must continue; limited spare capacity	**Rolling:** replace small batches while enough ready instances carry the load. `notes:1.1` and `notes:1.2` must coexist safely.
Capacity for duplicate pools; rapid traffic reversal matters	**Blue-green:** verify `notes:1.2` alongside `notes:1.1`, then switch traffic. Retaining `notes:1.1` avoids waiting to restart it during reversal.

⚠ Each choice still needs health evidence and a recovery path that works with the current database.

⚠ Beware: Changed data can prevent recovery

Suppose `notes:1.2` writes notes in a format that `notes:1.1` cannot read. Those notes remain after `notes:1.2` stops.

Strategy	Why returning to `notes:1.1` can still fail
In-place	Redeployed `notes:1.1` encounters the new data
Rolling	`notes:1.1` can fail during overlap or after reversing the rollout
Blue-green	Switching traffic back sends requests to `notes:1.1` against the changed database

A running old version is not proof of a working recovery path. Verify that the previous release can still serve users with the state `notes:1.2` leaves behind.

Verification and Recovery

How do we know the release works, and what if it does not?

What would convince you that a deployment succeeded?



A completed deployment can still fail users



Deployment succeeded

Containers are running.



Users cannot save notes

The user operation fails.

How do we detect the failure quickly and restore service?

Add a health check to Docker Compose

```yaml
# Add to production/compose.yaml
services:
  api:
    image: ghcr.io/example/notes:1.2
    healthcheck:
      test:
        - CMD
        - curl
        - -fsS
        - http://localhost:3000/health
      interval: 10s
      timeout: 2s
      retries: 3
      start_period: 20s
```

1. **test: call /health; exit 0 means success.**
2. **interval: check every 10 seconds.**
3. **timeout: stop a check after 2 seconds.**
4. **retries: 3 consecutive failures → unhealthy.**
5. **start_period: 20-second startup grace.**

- Implement `/health` in the API and include `curl` in its image.
  - return HTTP 200 when ready, 503 otherwise.

Wait for healthy containers

```
# Start services and wait up to 60 seconds
docker compose up --wait --wait-timeout 60

# Inspect their current state
docker compose ps
```

- `--wait` requires **healthy** when a check exists; otherwise, only **running**
- Health status: **starting**, **healthy**, or **unhealthy**. Checks continue after startup
- Fail the deployment job if the wait fails

Compose up · Docker health status

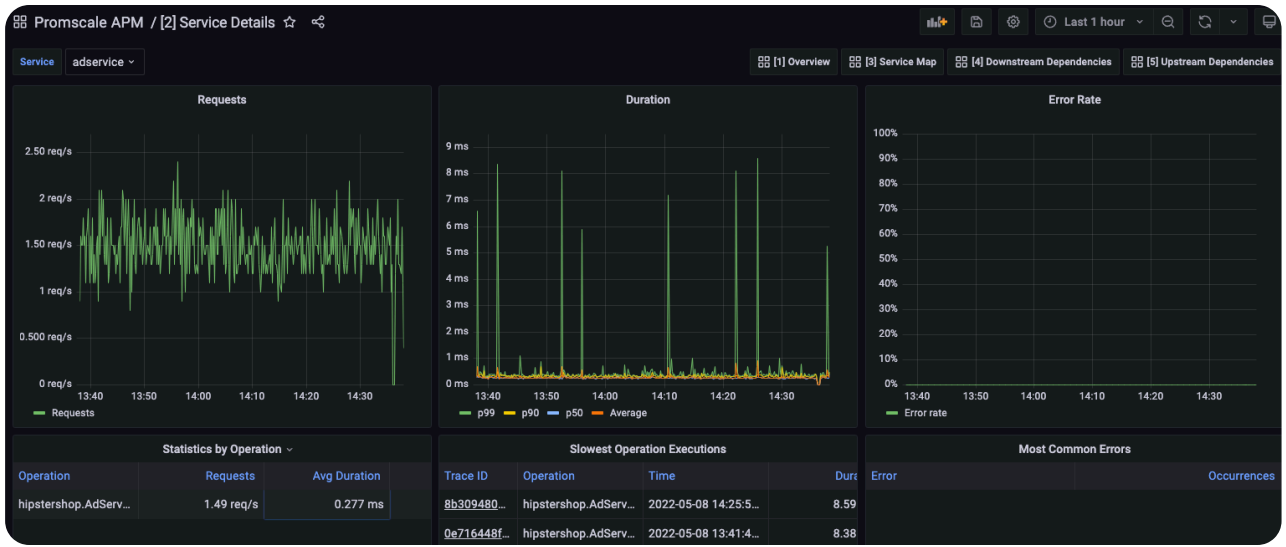
Test a user operation after deployment

`notes:1.2` passes readiness, but creating and reading a note fails.

Check	What it tells us
Deployment command succeeded	The deployment operation completed
Readiness check passed	The instance passed its configured traffic check
Create → read via public API	A user operation works through the proxy, API, and database

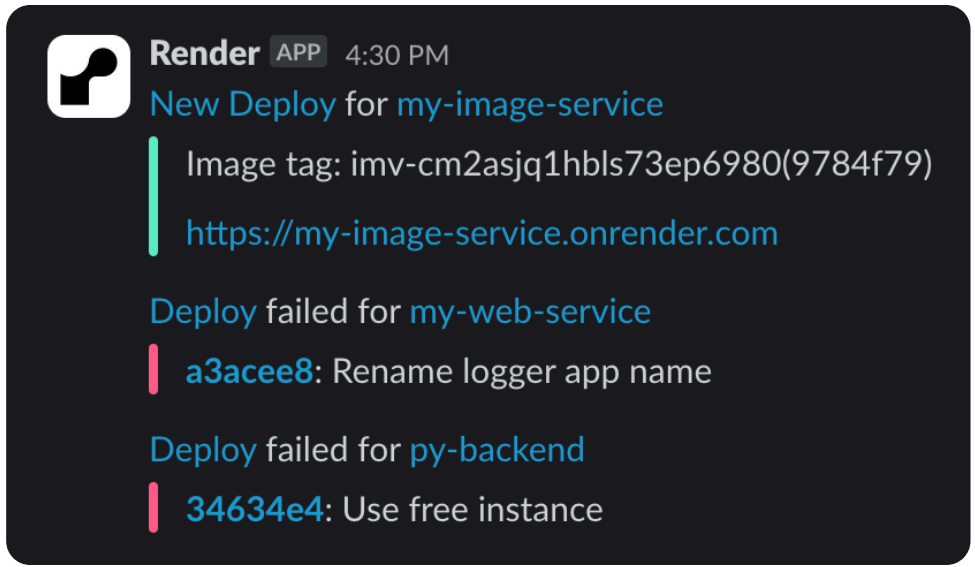
Run a production smoke test: create and read a dedicated test note through the public API, then delete it.

Monitor user-facing failures after deployment



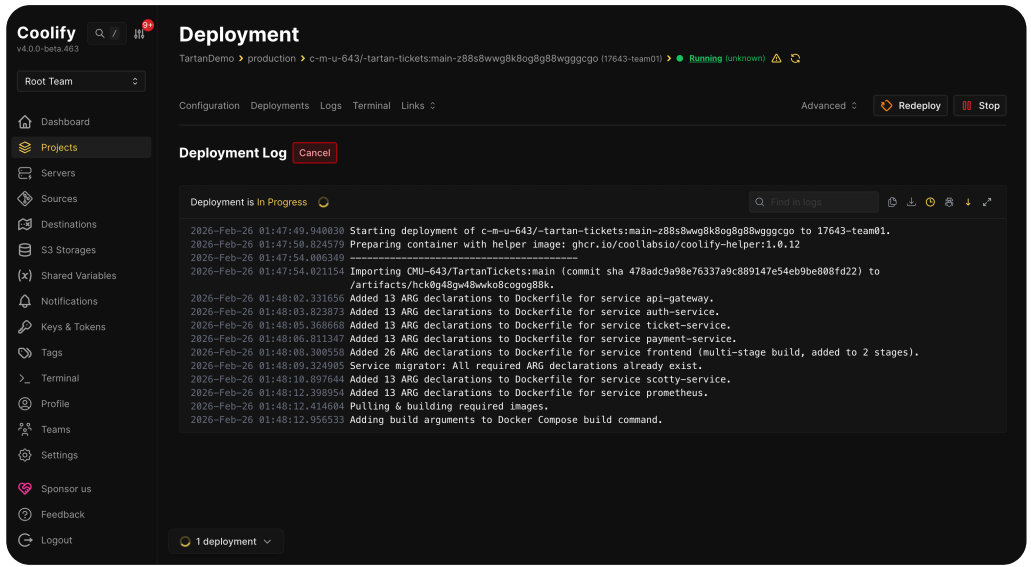
- Watch for **more failed requests** and **longer response times** as real traffic arrives.
- Keep running **create-and-read smoke tests**; failures can appear after deployment.

Route failure alerts to someone who can respond



- Send alerts to a monitored channel with a responsible responder.
- Include the affected service and a link to investigation details.

Use logs to investigate failures



- **Failed deployment:** find the failing step and error in deployment logs
- **Requests failing after startup:** inspect application logs for request errors

🔒 Containment limits user impact while we choose a recovery

- **Rolling:** pause the rollout before replacing more healthy `notes:1.1` instances
- **Blue-green:** switch the proxy back to `notes:1.1` if it still works with the current data
- Keep traffic on healthy instances only when they have enough capacity to serve it

Pause deployment automation as needed so it does not continue the failing rollout.

Would you automatically roll back every failed deployment?

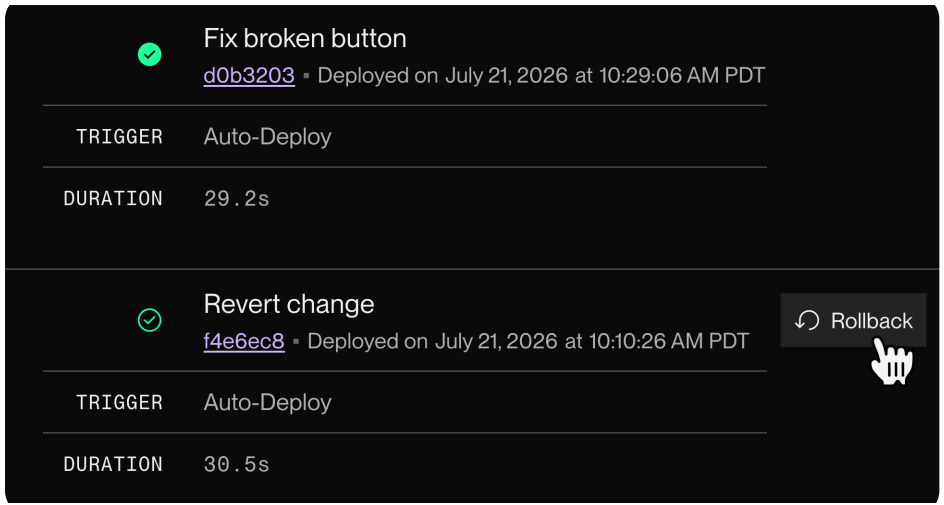


⏮ Rollback restores an earlier release

Failure after deploying `notes:1.2`	**Can returning to `notes:1.1` help?**
Code regression; data and configuration remain compatible	Yes: restore `notes:1.1` with compatible configuration, then verify
Stored notes now use a format that `notes:1.1` cannot read	No: the old version still encounters the changed data

⚠️ **Changing the application version does not undo changes to stored data.**

Rollback controls depend on your setup



Render documentation: rollback control

PaaS — Render
Choose a previous successful deploy; click **Rollback** and confirm.

VM — Docker Compose
Restore the previous image and compatible configuration; rerun deployment.

GitOps
Change the version in Git; the controller applies the desired state.

Update the deployment configuration so automation preserves the recovered version.

⏭ Rollforward deploys a corrective release

`notes:1.2` changed the stored format. Returning to `notes:1.1` would leave notes unreadable.

1. **Fix:** prepare `notes:1.3` to read the current format correctly.
2. **Check:** test the fix against the current data before deploying.
3. **Deploy and verify:** confirm that users can create and read notes again.

Even when rollback is possible, rollforward may be safer. Consider **the size of the fix** and **the time to deploy it**.

🚀 Today, we learned how to ...

- Trace changes from source to production, distinguishing continuous integration, continuous delivery, and continuous deployment
- Justify a delivery approach based on permissions, connectivity, operational effort, and visibility into results
- Explain how reconciliation and GitOps handle drift and emergency changes
- Plan promotion across isolated environments, preserving checked artifacts while varying configuration, secrets, data, and dependencies
- Verify application health and choose deployment and recovery strategies based on availability, capacity, and data compatibility