

Deploying with Confidence

Chris Timperley · Wednesday, September 23, 2026

🎯 Learning Goals

- Justify a deployment strategy using the system's constraints and risks
- Choose checks that show whether users can still use the service after deployment
- Explain when a recovery action could make a failure worse
- Design a deployment plan that addresses replacement, verification, and recovery

Previously: Deploying Changes to Production



Artifact

Deploy the artifact that **passed the checks**.



Authority

Decide **what initiates deployment** and what access it needs.



Environment

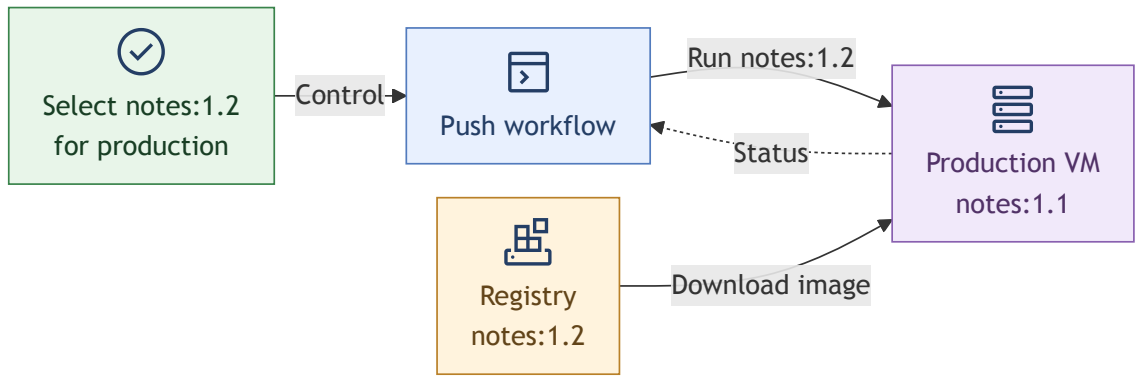
Supply the target environment's **configuration and credentials**.

TODAY | **Replace the running version, verify the result, and recover safely.**

Deployment Strategies

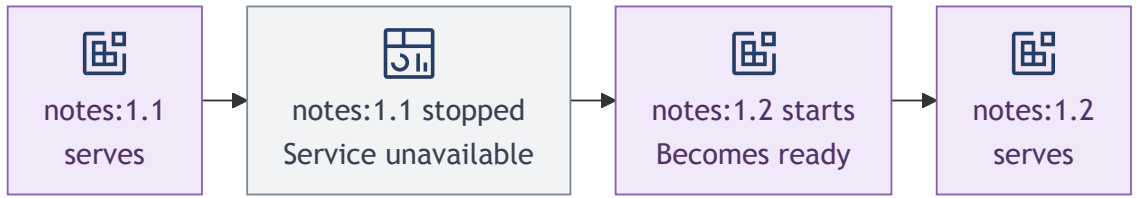
How do we replace the running version?

Initiation and replacement are separate choices



- **Initiation**: external workflow (**push**) or target-side agent (**pull**) starts deployment
- 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**.

In-place replacement interrupts service

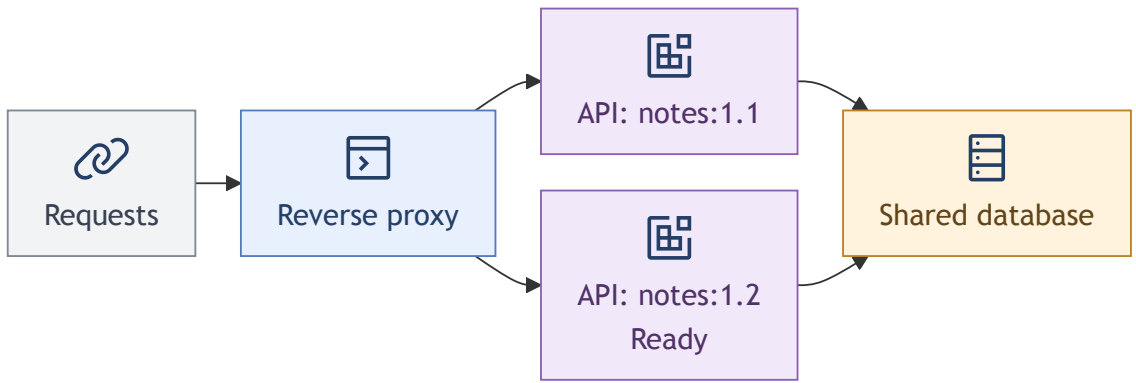


- **In-place replacement:** stop `notes:1.1`, then start `notes:1.2` in the same slot
- During the gap, the proxy has no working API
- No extra instance is needed, but service is interrupted
- If `notes:1.2` fails, redeploy `notes:1.1` and wait for startup

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 ...



Enough capacity

Keep enough instances serving. Add spare capacity or replace fewer instances at once.



Readiness checks

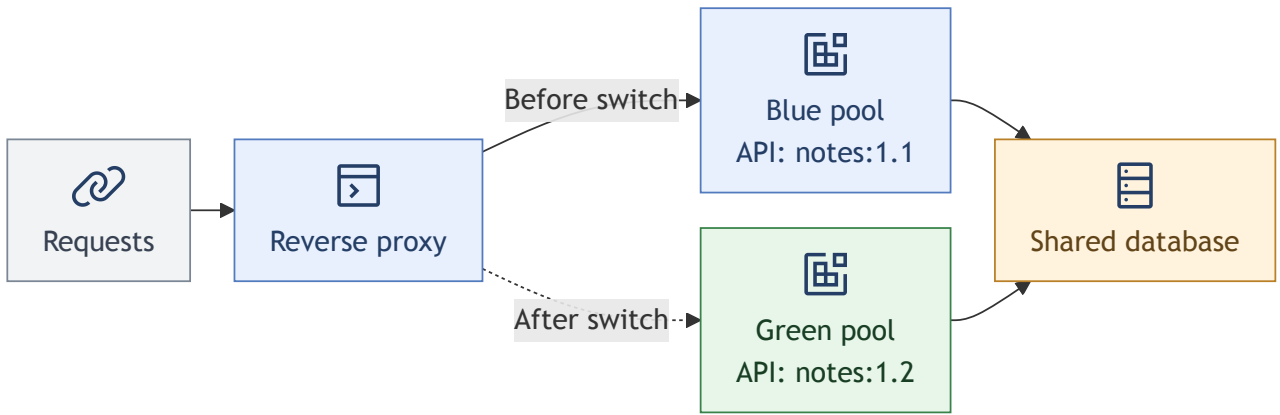
Send requests to new instances only when they can serve them.



Compatible versions

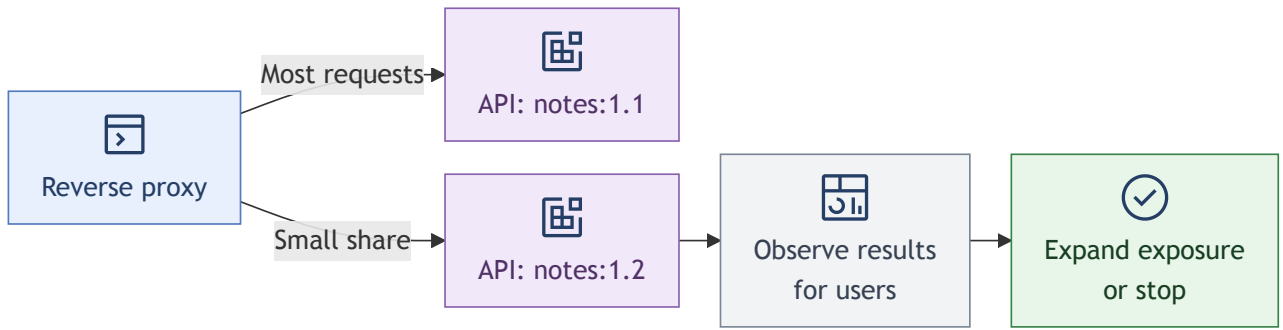
Both versions must work with the shared data during replacement.

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

Canaries test before expanding exposure



- Canaries 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?

1 **One instance · Downtime acceptable · Little spare capacity**

In-place: reuse the slot. Recovery requires redeploying the old version.

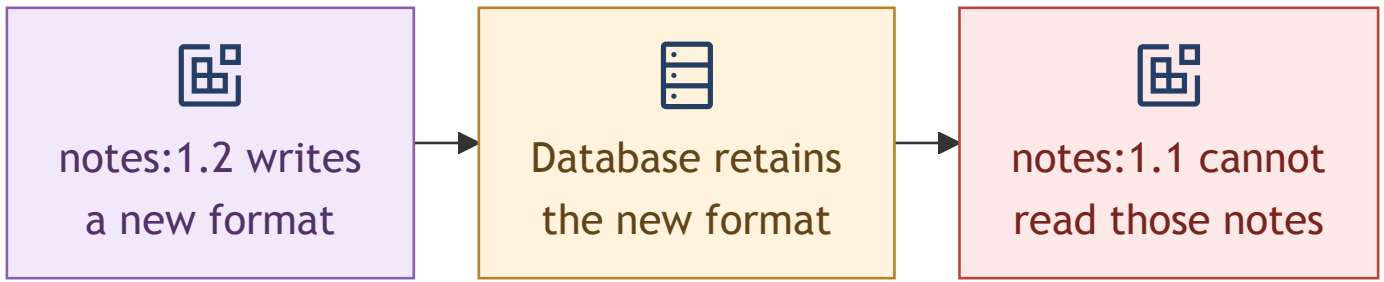
2 **Replicated API · No interruption · Limited spare capacity**

Rolling: keep enough instances serving as batches change. Both versions must work together.

3 **Capacity for two pools · Fast traffic reversal needed**

Blue-green: verify, then switch traffic. Keep the old pool running for quick reversal.

⚠ Beware: Changed data can prevent recovery



Restarting instances or switching traffic does not restore the old data format.

- **In-place:** the old version can fail after restart
- **Rolling:** the old version can fail during overlap or after reversal
- **Blue-green:** the old version can fail after switching traffic back

Verification and Recovery

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

A completed deployment can still fail users

Before deployment

notes.example

Tiny Notes

New note

Review the deployment plan.

Save note ✓ Saved

YOUR NOTES
Review the deployment plan.

After deployment

notes.example

Tiny Notes

New note

Review the deployment plan.

Save note ⊘ Not saved

COULD NOT SAVE YOUR NOTE
Please try again.

Deployment succeeded. Containers are running. Can users still save notes?

How can we detect bad deployments early?

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`: **200 when ready, 503 otherwise**.
- Include `curl` in the image.

Wait for healthy containers

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

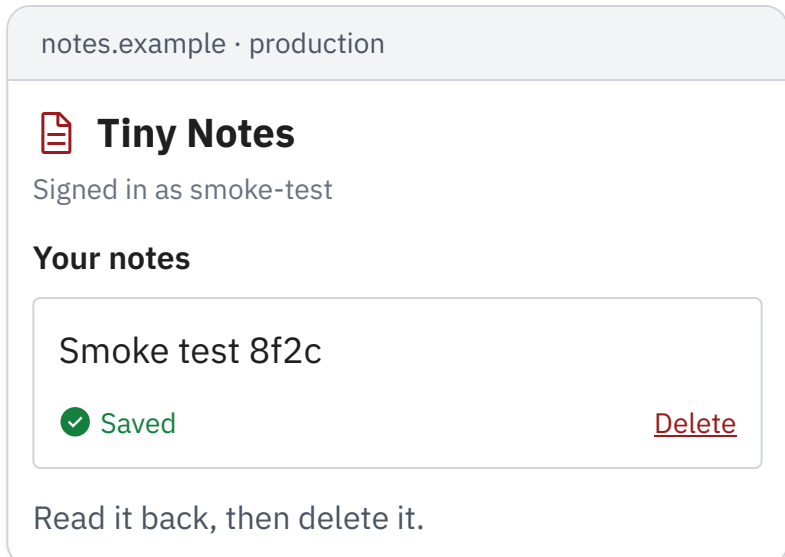
# Inspect their current state
docker compose ps
```

- `--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

Run a smoke test in production

```
page = open(PRODUCTION_URL)
sign_in(page, smoke_test_account)

text = "Smoke test " + unique_id()
note = create_note(page, text)
try:
    reload(page)
    assert read_note(page, note.id) == text
finally:
    delete_note(page, note.id)
```



- After deployment, test a **real user operation against the live service**

Keep checking after the deployment finishes



Observe
Watch failed user operations and response times.

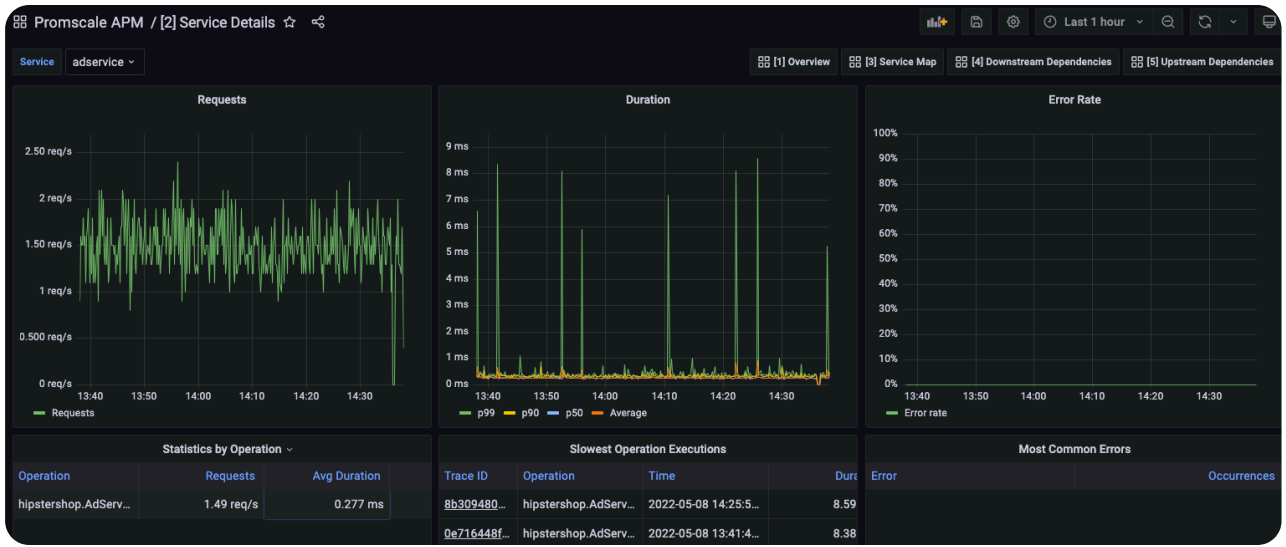


Alert
Notify a responder with the release and investigation details.



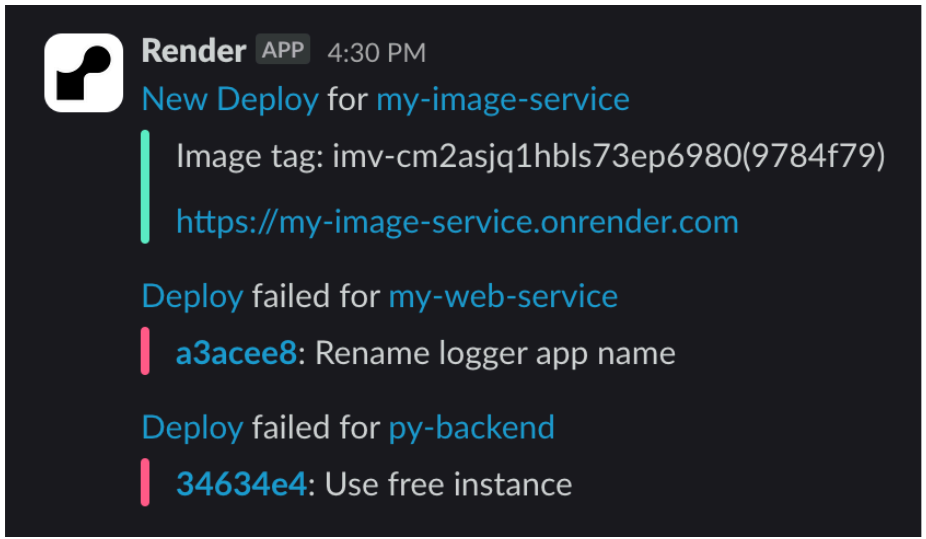
Investigate
Use deployment logs for failed steps; app logs for failed requests.

Monitor user-facing failures after deployment



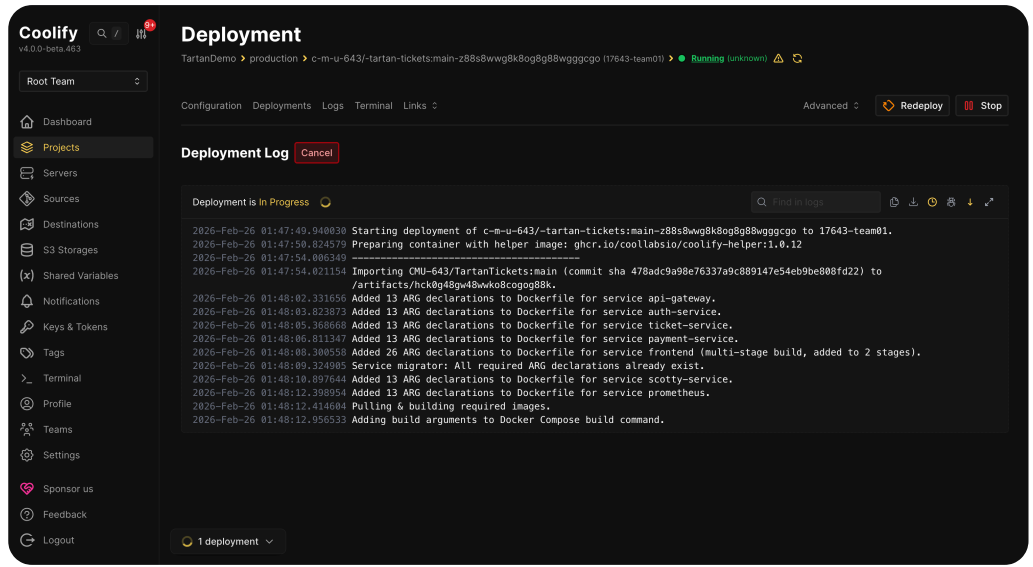
- Watch for **more failed requests** and **longer response times** as real traffic arrives.
- Keep running **create-and-read smoke tests**; failures can appear after deployment.

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

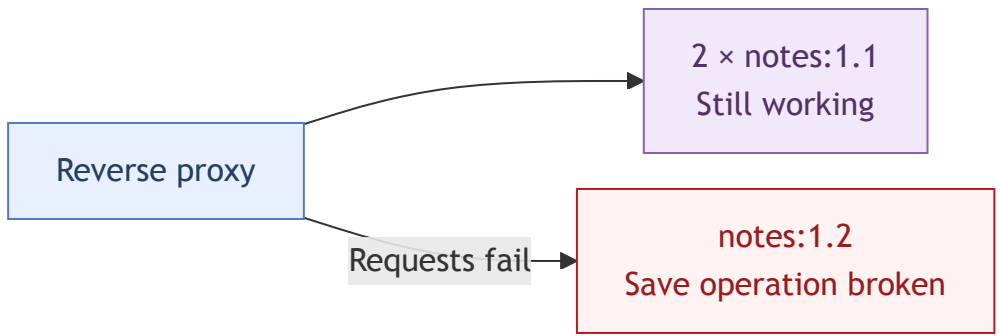


- **Failed deployment:** find the failing step and error in deployment logs
- **Requests failing after startup:** inspect application logs for request errors

When a Release Fails

Rolling: Pause the Rollout

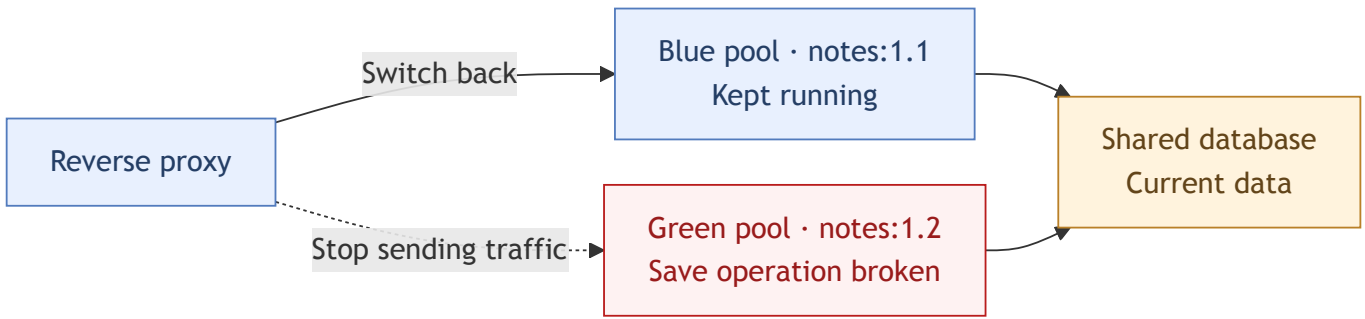
Rolling deployment replaces instances **in batches**. Here, the first replacement fails.



- **Pause the rollout:** keep the remaining healthy `notes:1.1` instances
- **Stop routing to failing instance**, keeping enough healthy capacity to serve users

Blue-Green: switch traffic back to the old pool

Blue-green keeps **two complete pools**. Here, failures appeared after the switch.



- **Switch back only if `notes:1.1` works with the current data.**

  The database stays as it is.

- Pause deployment automation so it does not switch traffic to `notes:1.2` again

Rollback restores an earlier release



Compatible state

A code regression broke `notes:1.2`; data and configuration remain compatible.

Restore `notes:1.1`, then verify it.



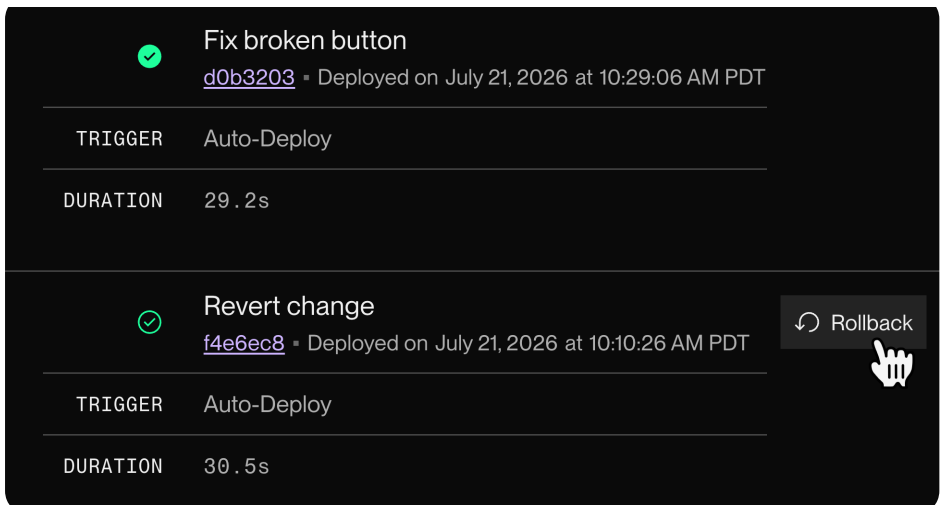
Incompatible state

`notes:1.2` wrote a format that `notes:1.1` cannot read.

The old release still fails against the changed data.

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

Rollback controls depend on your setup



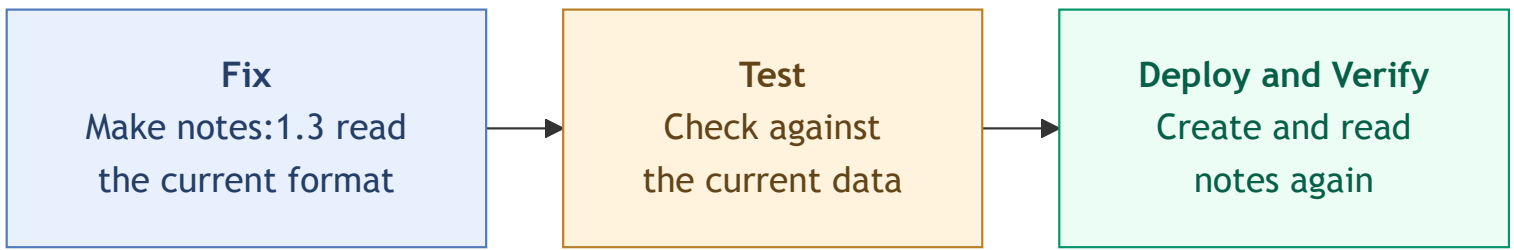
PaaS: Render
Choose a previous successful deploy; click **Rollback** and confirm.

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

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

Rollforward deploys a corrective release

- **Scenario:** `notes:1.2` changed the stored format. Returning to the previous release would leave notes unreadable.



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

Deployment Gone Wrong

Knight Capital (2012)



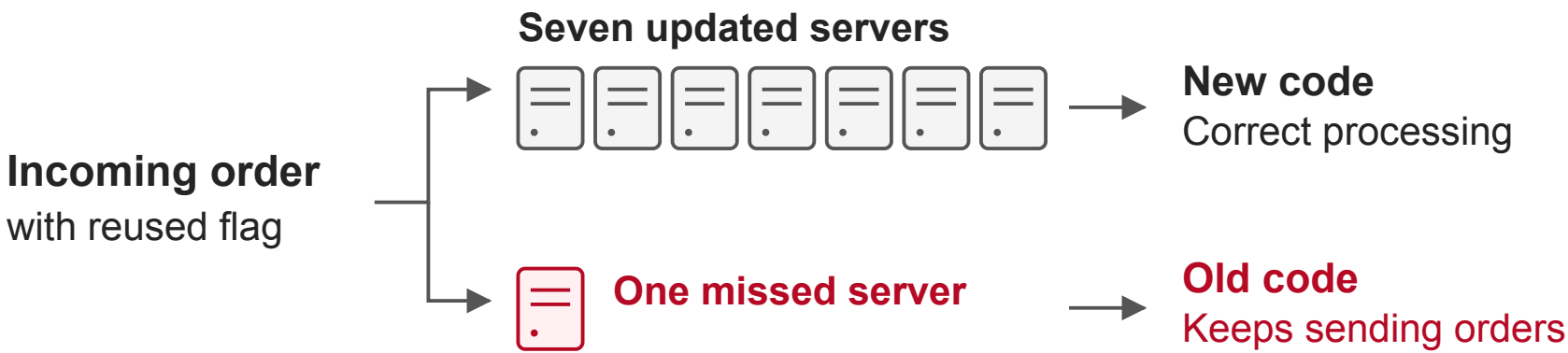
August 3, 2012 · Two days after the incident · Brendan McDermid / Reuters

- Knight's software split customer requests into smaller stock orders.
- On **August 1, 2012**, it kept sending orders beyond the quantities requested.
- The incident lasted about **45 minutes**; losses exceeded **$460 million**.

Restoring the software can't undo trades already executed.

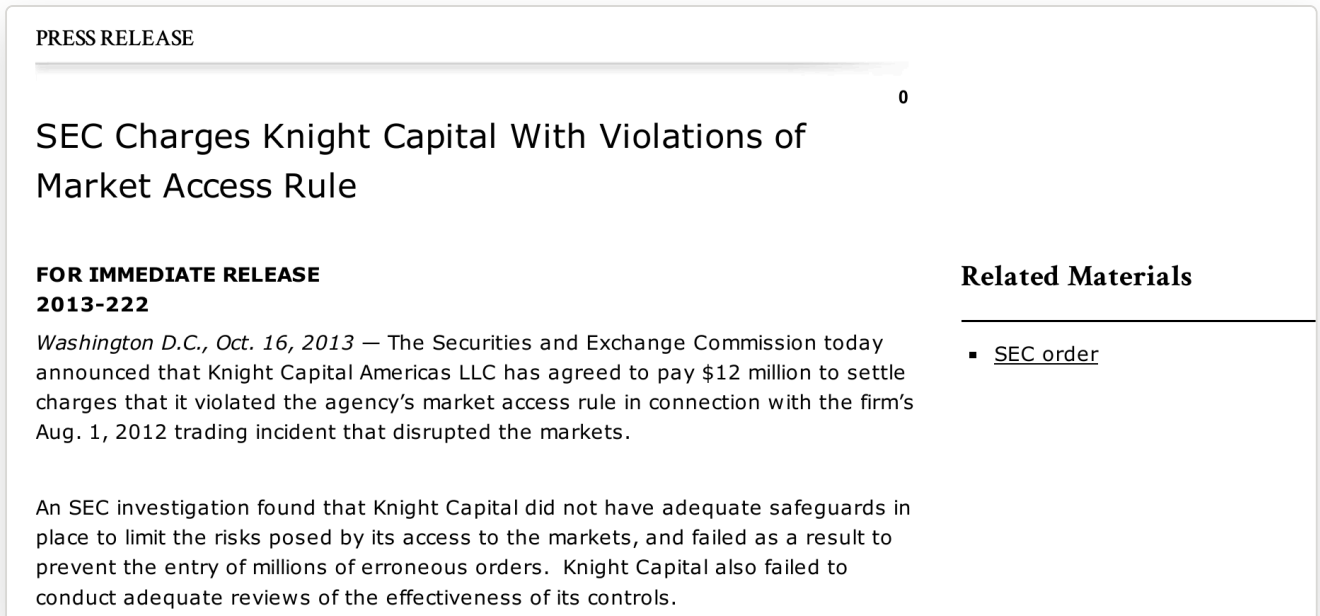
The same flag selected different code

Deployment reached **seven of eight servers**; the omission went unnoticed.



The obsolete “Power Peg” path remained callable. Earlier changes to quantity tracking had left it defective when reactivated.

The SEC found inadequate safeguards



PRESS RELEASE

0

SEC Charges Knight Capital With Violations of Market Access Rule

FOR IMMEDIATE RELEASE
2013-222

Washington D.C., Oct. 16, 2013 — The Securities and Exchange Commission today announced that Knight Capital Americas LLC has agreed to pay $12 million to settle charges that it violated the agency's market access rule in connection with the firm's Aug. 1, 2012 trading incident that disrupted the markets.

An SEC investigation found that Knight Capital did not have adequate safeguards in place to limit the risks posed by its access to the markets, and failed as a result to prevent the entry of millions of erroneous orders. Knight Capital also failed to conduct adequate reviews of the effectiveness of its controls.

Related Materials

- SEC order

SEC · October 16, 2013 · Archived announcement excerpt

The **$12 million settlement penalty** shown here is separate from the **trading loss**.

Knight Capital: What needed to change?

Discuss the **incident brief** in your team of **5–6**.

1. Beyond the missed server, what weaknesses allowed this incident to happen?
2. What changes could have prevented it?
3. If something still went wrong, what safeguards could have limited the damage?

Consider the software, working practices, and incident response.

Deployment Design

Design a deployment for your scenario

- Identify your scenario's **constraints and risks**
- Explain how they shape your plan to **deploy, verify, and recover**
- State your **assumptions** and information needed before proceeding



Hospital

Medication records



Satellite

Image processing



Multiplayer game

Match servers



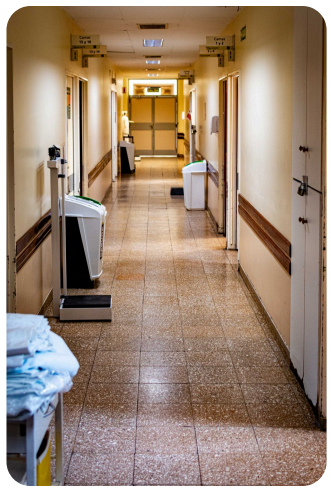
Supermarket

Self-checkouts

A · Hospital Medication Records

Update the system nurses use to read and record medication doses during patient care.

- **Four application servers** share one database
- The update introduces a **new medication status**

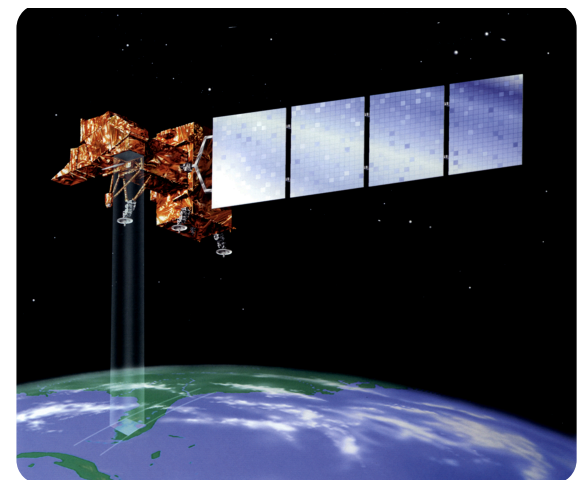


Juan Moccagatta / Pexels

B · Satellite Image Processing

Update a satellite's app that processes collected images. Flight control and communications are separate.

- **Ten-minute contact**, then hours offline. Uploads may be interrupted
- Storage holds **both versions**; only one runs at a time
- A separate program manages the app and can start either stored version



NASA · Landsat 7 illustration

C · Multiplayer Game Servers

Update the servers that run an online game's matches.

- Each server runs one match at a time, lasting up to **30 minutes**
- The update changes how players **earn items**. Player inventories persist across matches and servers.



Alena Darmel / Pexels

D · Supermarket Self-Checkouts

Update payment-response handling across about 500 self-checkouts.

- **Eight checkouts per store**; opening hours vary
- Each checkout runs **one app instance**
- Connections can fail; stores **reject inbound deployment connections**



Hay Kranen · CC BY 4.0 · cropped

Explain one important deployment decision

- What did your team decide?
- What about your scenario led to that decision?
- What would you need to find out before proceeding?

Deployment choices depend on the situation

Scenario	One possible decision	Why it fits the situation
Hospital	Check record compatibility, then replace gradually	Continued access; shared records across versions
Satellite	Finish upload before switching; recover locally	Interrupted uploads; hours without contact
Game	Stop new matches; finish the current match before updating	Updating mid-match could interrupt play
Stores	Fetch updates; apply in small batches when idle	Inbound connections blocked; customers still shopping

What would you still need to find out before using each approach?

🚀 Today, we learned how to ...

- Justify a deployment strategy using the system's constraints and risks
- Choose checks that show whether users can still use the service after deployment
- Explain when a recovery action could make a failure worse
- Design a deployment plan that addresses replacement, verification, and recovery