

Midpoint Review

Chris Timperley · Monday, October 5, 2026

Administrivia: Midterm I

- **Paper exam:** the full 80-minute class; designed for 60 minutes of work
- **Notes:** up to three double-sided sheets (six sides), printed or handwritten
- **Practice exam:** longer than the real exam, with no target completion time
- **Scope:** real exam will cover seven announced topics
- **Preparation:** explain the key concepts and apply them to an unfamiliar scenario

🎯 Learning Goals

- Explain the key distinctions in each exam topic
- Interpret evidence: what it shows and what it does not
- Recommend an action and justify it with evidence, constraints, and a tradeoff

Containers and Orchestration

A pet must be preserved; cattle can be replaced

A **snowflake server** has unique, undocumented setup that is hard to reproduce.



NAMED VM
apollo

NO · PRESERVE IT

Pet

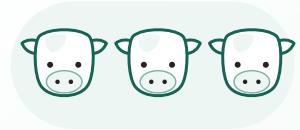
Preserve or repair this specific machine.

It holds something important that no replacement has.

EXAMPLES

- Hand-tuned server
- VM with local files

OR



YES · REPLACE IT

Cattle

Replace it with an equivalent machine.

api-042 api-044

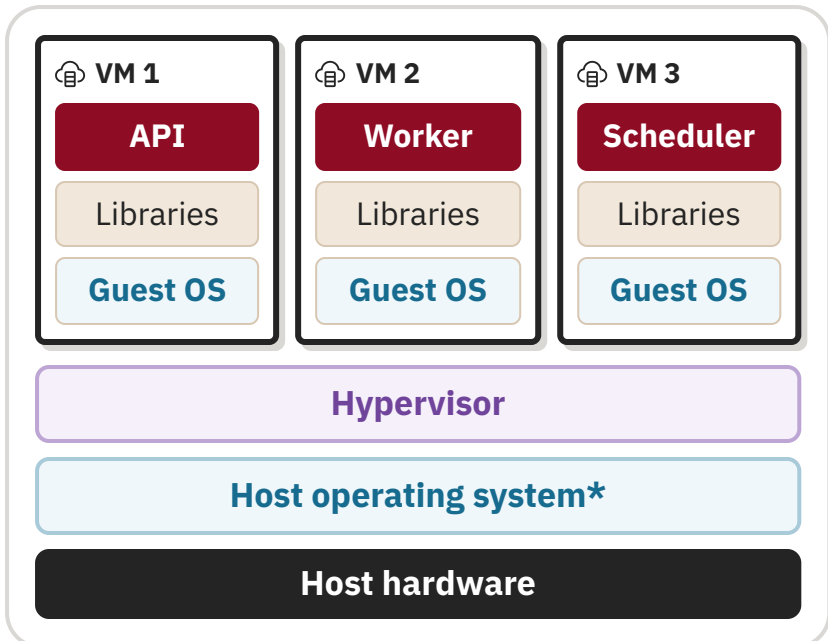
Their setup is repeatable, and important data lives elsewhere.

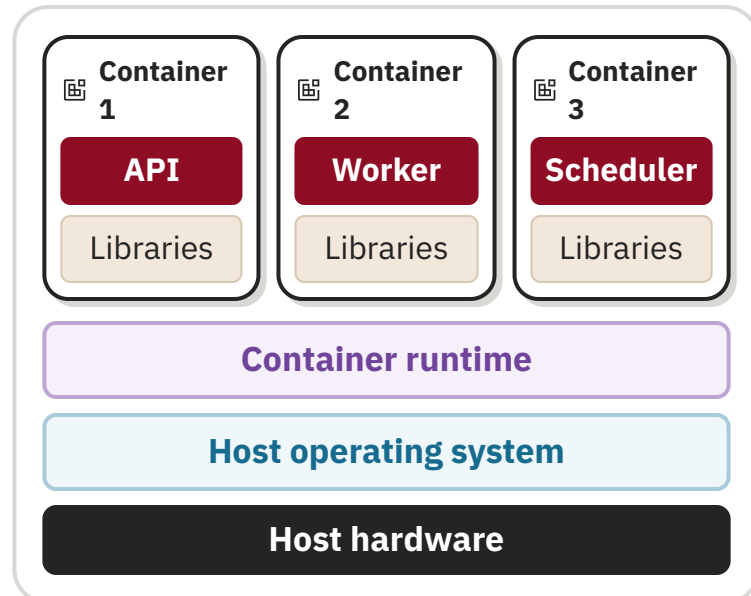
EXAMPLES

- API server from a snapshot
- Temporary server

Safe replacement lets teams treat critical machines as cattle

VMs repeat the operating system; containers share it





Sharing the host kernel lowers container overhead

VIRTUAL MACHINES

Independent guests

Each VM carries and boots a complete guest operating system.

Pros

Strong machine boundary; each VM can run a different guest OS or kernel.

Cons

More memory and storage overhead; another operating system to boot and manage.

CONTAINERS

Shared host kernel

Each container packages userspace while sharing the host kernel.

Pros

Smaller artifacts, fast startup, and denser use of a host.

Cons

The kernel is a shared trust boundary; host compatibility still matters.

A tag labels an image; a digest identifies it

```
$ docker build -t notes-api:stable .
```



YOU CHOOSE

Tag / label

`notes-api:stable`

A readable pointer. Reusing the tag can make it point to different bytes.



CONTENT-DERIVED

Digest

`sha256:7c4a…91e2`

An exact content identity. The same digest always names the same bytes.

Declarative tools converge actual state toward intent



Observe reality

Inspect what is true now

Compare with intent

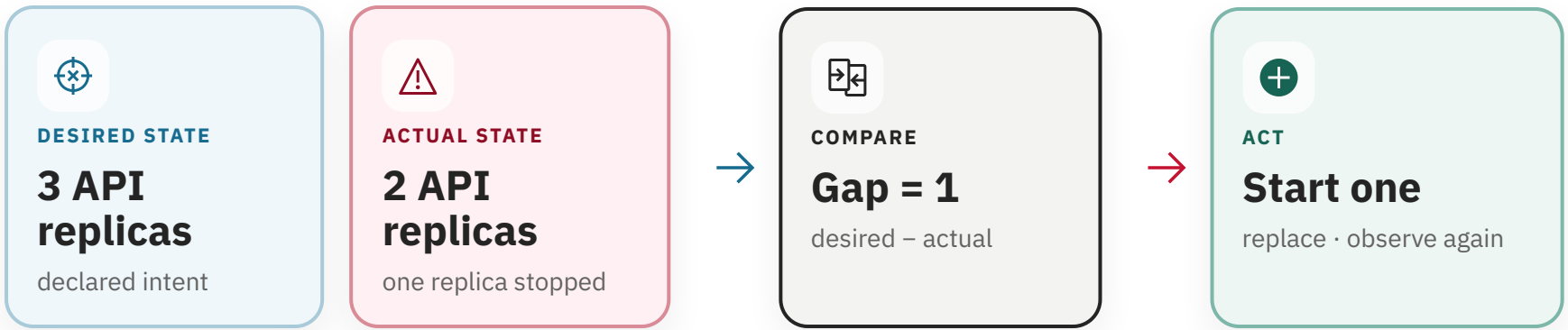
Find **what differs** from the declaration

Change the gaps

Apply **only the changes** that are needed

On the second run...
The existing user is left alone. The service starts only if needed.

Reconciliation closes the desired–actual gap



Recall: the database was replaced successfully

```
services:
  api:
    image: notes-api@sha256:7c4a...
    environment:
      NOTES_DB_URL: postgres://db/notes
  db:
    image: postgres:17
```

The team recreates both containers for every deployment.

The new database starts cleanly. **Every customer note is gone.**

What does successful container replacement prove? What does it leave untested?

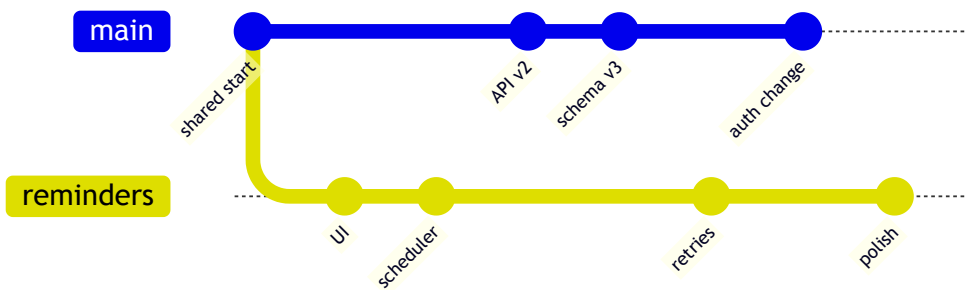
Keep durable data outside the container; test backup and recovery.

Key Concepts: Containers and Orchestration

Drift and recovery	explain snowflakes, pets, and cattle; capture undocumented setup and verify replacements
VMs and containers	compare kernels, isolation, overhead, and compatibility
Images and identity	distinguish recipe, image, container, tag, and digest
Persistence	predict what survives container/host failure: code, configuration, writable layers, and state
Declarative automation	compare imperative steps and desired state; explain idempotence, convergence, and versioned setup
Orchestration	trace reconciliation; compare Compose and cluster scope

Version Control and Collaborative Workflows

A long feature branch accumulates untested differences



Mainline moves

API, schema, and authentication change.

The feature grows

UI, scheduler, retries, and polish accumulate.

The combined state

Never built or tested together. Every change adds more assumptions that may be incompatible.

A semantic conflict can survive a clean merge

A Branch A · `store.ts`
Rename the export

```
export function loadRecord(id) {
  return records.get(id)
}
```

✓ **Callers updated · tests pass**

Git merges cleanly
different files · no overlapping text

B Branch B · `duplicate.ts`
Call the old name

```
import { getRecord } from './store'

export const duplicate = id =>
  copyOf(getRecord(id))
```

✓ **Passes against current main**

Combined build fails: `getRecord` is no longer exported
Semantic conflict: the text merges, but an assumption in the combined behavior is broken.

Small batches turn feedback into progress sooner



LEARN SOONER

Test assumptions early

Get feedback while the change is still easy to adjust.



FIX FASTER

Narrow the search

Inspect fewer changes when something fails.



CHANGE DIRECTION

Stop sooner

Avoid more work on an approach that is not working.



SEE PROGRESS

Finish work often

Frequent completed changes support efficiency and motivation.

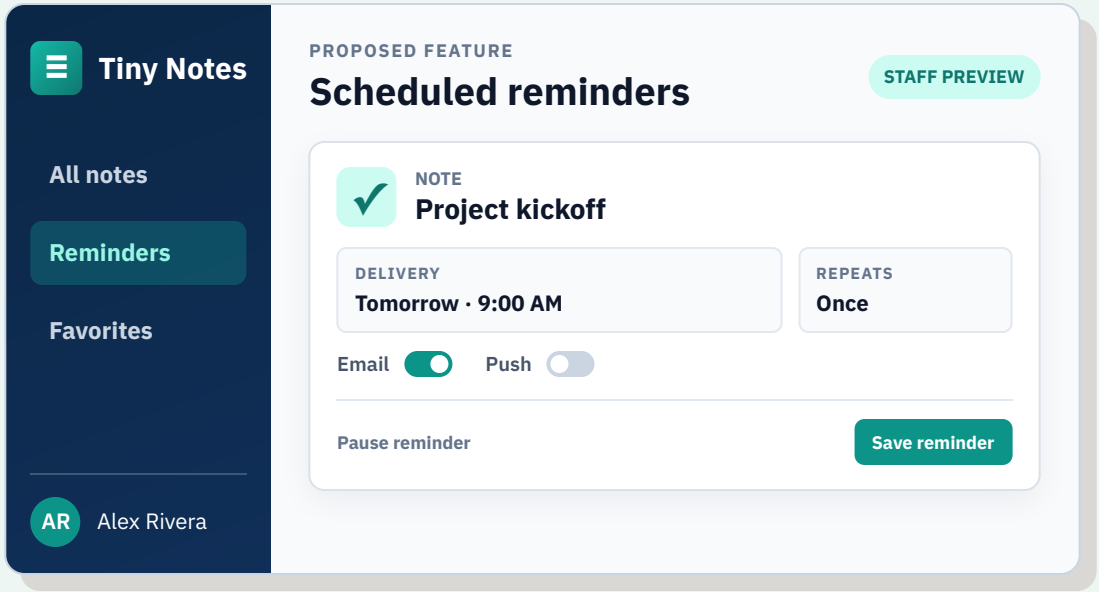
DORA · Working in small batches

Recall Tiny Notes' scheduled reminders

The full feature

- Custom delivery times
- Recurring schedules
- Email and push delivery
- Pause and resume

Estimated effort: **3 weeks**



The first batch can reach mainline before the whole feature is ready.

Feature flags let us merge code before users see it

- **Merge:** Put the code on the mainline while the behavior stays off.
- **Choose who sees it:** Turn it on for staff, a test environment, or part of the user base.
- **Clean up:** Test both states and remove the flag after everyone has the feature.

```
const remindersEnabled = featureFlags.isEnabled(
  'scheduled-reminders',
  currentUser,
)

if (remindersEnabled) {
  showReminderControls()
}
```



LaunchDarkly

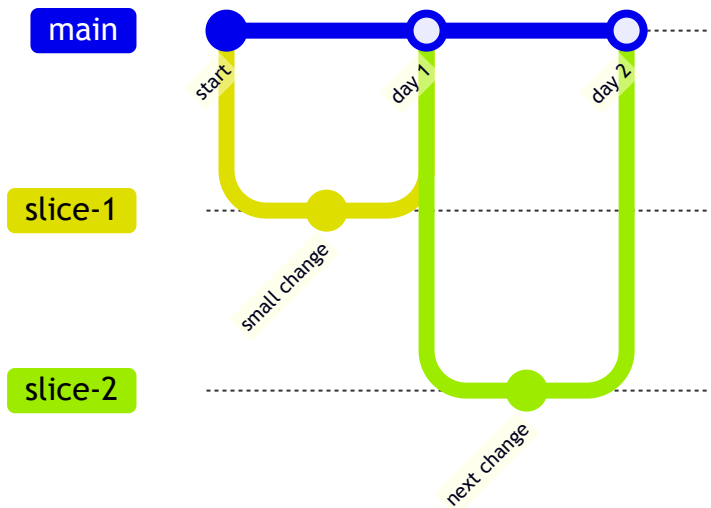


Harness



Optimizely

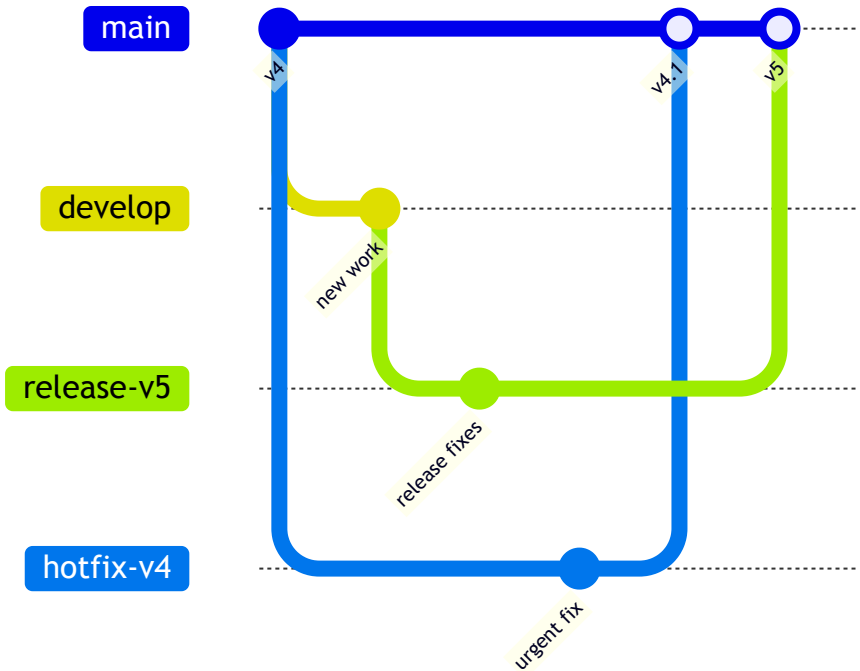
Trunk-based development requires daily integration



- **Merge daily:** each developer integrates at least once per day.
- **Keep branches short:** branches or forks live for less than one day.
- **Keep few branches:** the team maintains fewer than three active branches.

- **Addresses:** branch divergence by limiting how long work stays apart.
- **Policy still decides:** review evidence and release support. A same-day reviewed PR can satisfy both GitHub Flow and trunk-based development.

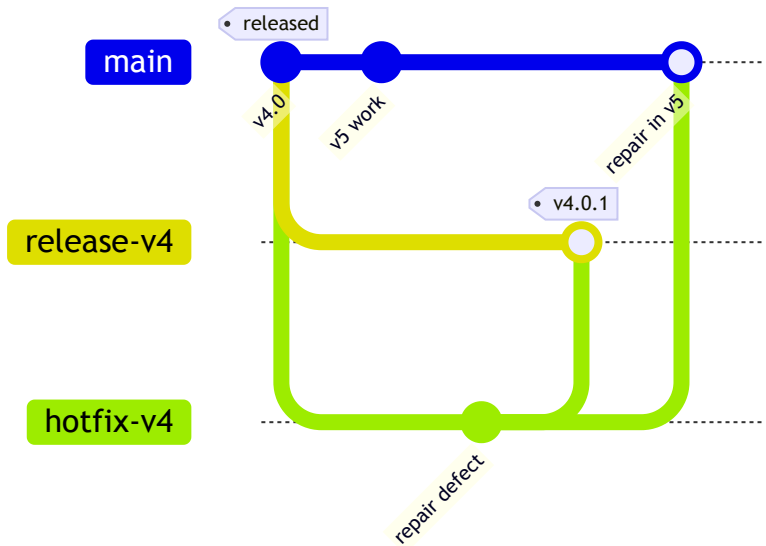
Git-flow adds branches for release support



- `develop`: feature work integrates toward the next version.
- `release-*`: feature scope freezes; only defect fixes and release preparation continue before merging to `main`.
- `hotfix-*`: urgent repairs begin from the released line and propagate forward.

- **Addresses:** planned releases and urgent fixes across supported lines.
- **Policy still decides:** batch limits, review time, required checks, and how fixes propagate between branches.

A hotfix must reach every affected line



1. Branch from the release line that contains the defect.
2. Make and check only the focused repair.
3. Merge the repair into the supported release.
4. Propagate it to every newer affected line.

What does supporting release branches cost?

A monorepo can make a source change atomic

Scheduled reminders touch `web`, `API`, and `worker`.

Monorepo

ATOMIC

ONE COMMIT

`web` + `API` + `worker`

All source changes land together.

Polyrepo

STAGED

SEVERAL COMMITS

1. **Add compatible API**
2. **Update web + worker**
3. **Remove old API**

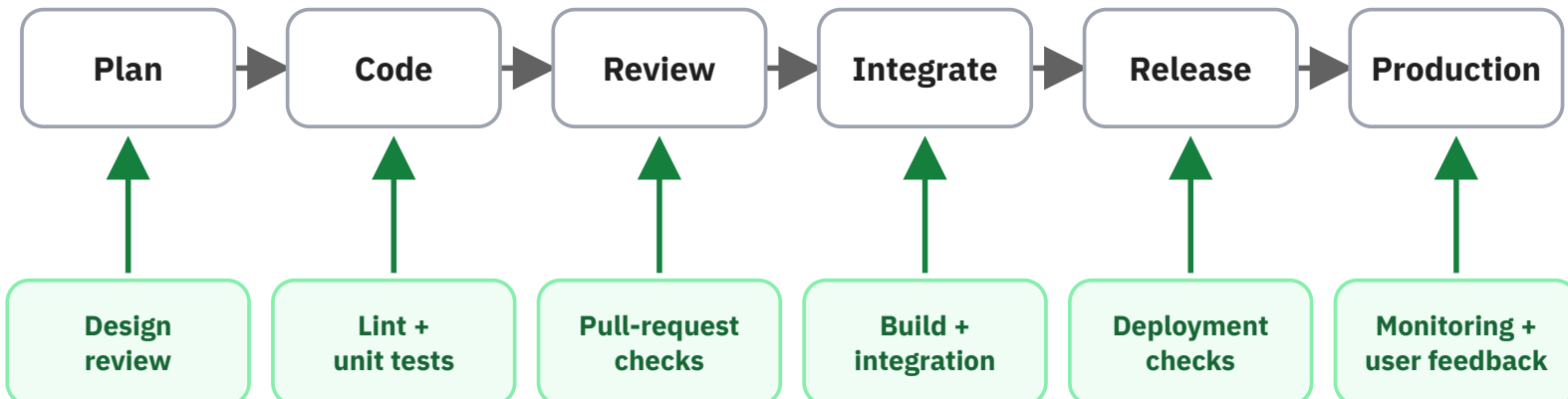
Atomic means one source commit—not one deployment.

Key Concepts: Collaborative Workflows

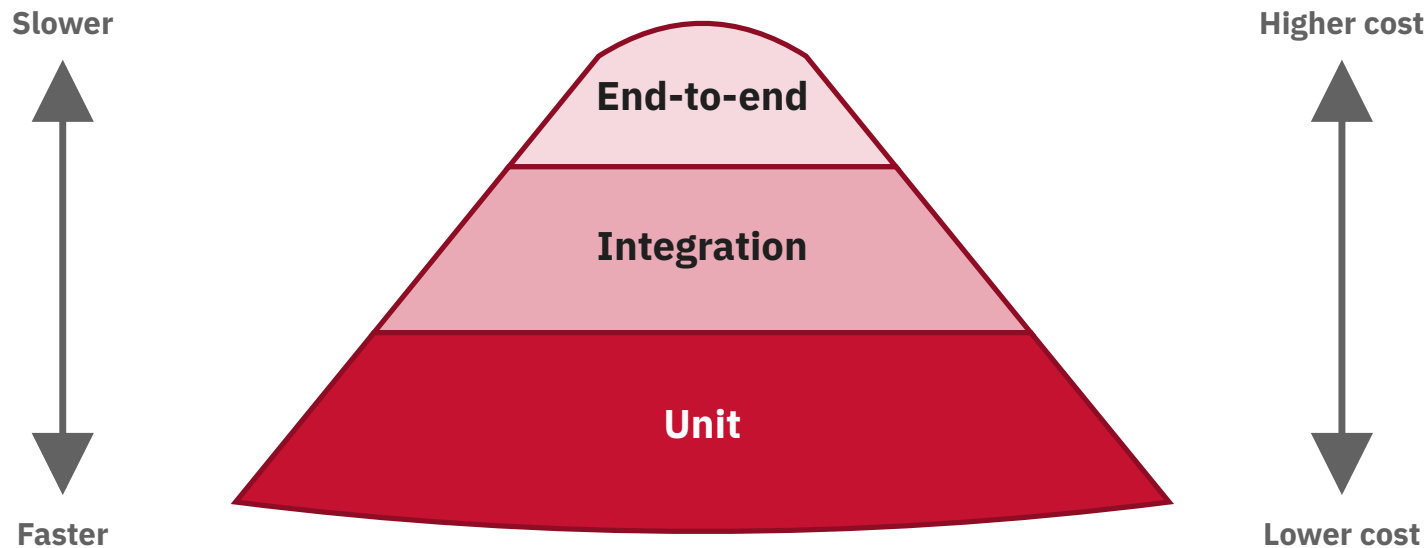
Shared history	distinguish commit, push, review, and merge
Conflicts	explain divergence; distinguish textual and semantic conflicts
Workflows	compare feature branches, GitHub Flow, trunk-based development, and Git-flow
Small batches	plan small, working increments; use flags to separate integration from exposure
Dependent work	explain experimental branches and stacked PRs
Supported lines	distinguish release and hotfix branches; propagate repairs to every affected line
Repositories	compare monorepos and polyrepos; distinguish atomic changes from atomic deployment

Continuous Integration and Testing

Checks turn the delivery lifecycle into feedback loops



The Test Pyramid favors many small tests and a few broad tests



A test double replaces a collaborator during a test

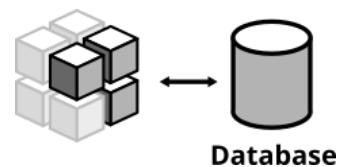
- **Choose its response**, including errors and unusual values.
- **Prevent side effects**, such as database writes or real email.
- **Observe calls** that the unit does not return as output.
- **Make changing values predictable**, such as time and generated IDs.

Keep the real collaborator when it is fast, predictable, and safe.

Integration tests check interactions across a boundary

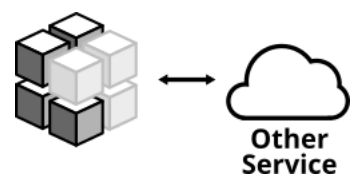
Code is tested at integration points

- Does our REST API work?
- Can we write data to the database and read it back?
- Can we save files to disk and retrieve them?
- Do external API calls work, and do we parse their responses correctly?



Database

PostgreSQL
Testcontainers



HTTP API

MSW
WireMock
Pact

Recall: why not use only end-to-end tests?



Use fast tests for focused feedback; keep broad tests for risks across the full path.

Each static tool checks a different kind of problem



Format

Prettier · Black · `gofmt`



Lint

ESLint · Ruff · Stylelint



Types

TypeScript · mypy



Configuration

JSON Schema · Ajv



Code + data flow

CodeQL · Semgrep



Dependencies

Dependabot · Renovate

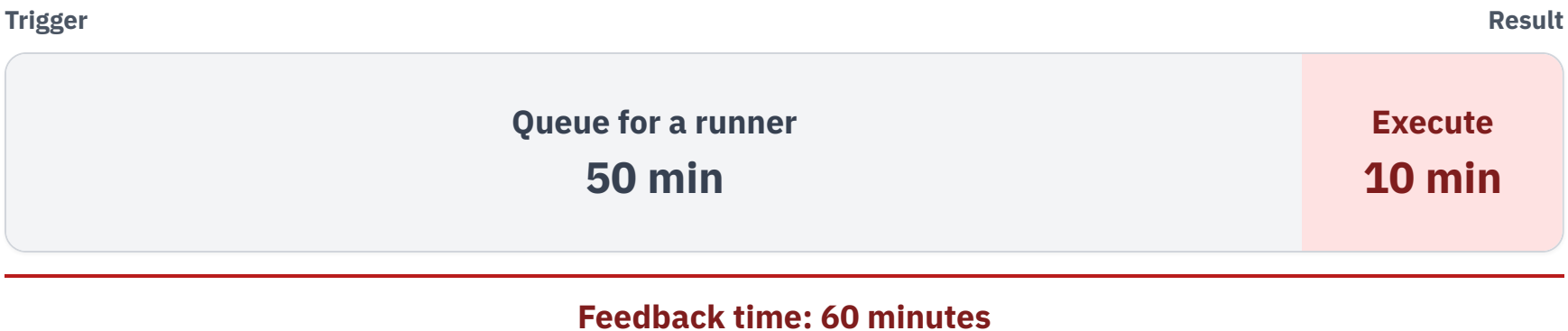
Run fast checks while coding, then run required checks again in CI.

Triggers can come from code, time, people, or other workflows

Trigger	Example
Push	Test a commit pushed to `main`
Pull-request activity	Test when a pull request opens or changes
Scheduled time	Run the full integration suite every night
Manual request	Run diagnostics on demand while investigating a failure
Workflow completion	Publish reports after the test workflow finishes

GitHub Docs · Events that trigger workflows

A ten-minute test suite can still make a developer wait an hour



False failures teach developers to rerun

Same commit

FAIL

→ RERUN

Same commit

PASS

A **flaky test** passes or fails without a relevant code change.

Repeated false failures make “rerun” feel safer than “investigate.”

Only reliable checks should stop a merge

A check should stop a merge only when it:

- protects against a specific risk
- gives the same result for the same code
- has a team or person responsible for maintaining it
- tells developers what to do when it fails

Key Concepts: CI and Testing

CI and shifting left	justify frequent integration, repeatable checks, and fast feedback
Test boundaries	choose unit, integration, or end-to-end tests and state their limits
Test doubles	choose stubs, fakes, spies, or mocks; control external calls, time, and randomness
Static checks	match formatting, lint, type, schema, and dependency checks to risks
Triggers and gates	match PR, mainline, scheduled, and manual checks to risk
Pipeline health	diagnose slow or costly pipelines; choose caching, parallelism, or check selection
Flakes	investigate inconsistent results; quarantine with an owner and a deadline

Secrets, Authentication, and Authorization

Copies can outlive the original leak

Deleting the first copy can leave the same credential in:

- source history and forks
- logs and build artifacts
- caches and image layers
- backups

```
build.log
DEPLOY_TOKEN=...7F3K

artifact/env.txt
DEPLOY_TOKEN=...7F3K

image/config.json
"DEPLOY_TOKEN=...7F3K"
```

Cleanup reduces exposure; revocation ends access.

Valid proof does not grant every permission



Authentication ✓

The token is valid. Identity accepted: ci-build.

Authorization ✕

ci-build cannot delete the production database.

A valid credential does not permit every action.

Scope limits reach; lifetime limits time

SCOPE ↓ · LIFETIME →	LONG LIFETIME	SHORT LIFETIME
BROAD SCOPE	*Persistent, widespread access*	**A · Temporary administrator token** Every Tiny Notes system · 10 minutes
NARROW SCOPE	**B · Read-only token for one repository** One repository · no expiration	**Design target** Narrow scope · short-lived

Durable secret storage follows two common patterns



VERSIONED WITH CONFIGURATION

Encrypted file

SOPS stores ciphertext in Git; age or a cloud KMS controls who can decrypt it.



CENTRALIZED BEHIND AN API

Managed secret store

Google Secret Manager, AWS Secrets Manager, Azure Key Vault, or Vault controls retrieval and lifecycle.

Both patterns protect a durable source.
Decryption or retrieval creates plaintext elsewhere.

Secret settings control delivery, not later use

BEFORE INJECTION

Platform controls delivery

- Repository or environment scope selects eligible jobs
- A review gate can delay access

SECRET INJECTED



AFTER INJECTION

Job code controls use

- The process can read the credential
- The process can send it anywhere the credential is accepted

Run PR checks without publishing access

```yaml
jobs:
  pull-request-checks:
    permissions:
      contents: read
    steps:
      - uses: actions/checkout@v4
      - run: npm ci && npm test
```

1. The repository token has read-only access.
2. PR and dependency code execute in this job.
3. This job receives no publishing or deployment credential.

Publish separately: approved source, protected environment, and narrowly scoped credentials.

Recall: can Tiny Notes close the token incident?

- The deployment token was committed publicly, then removed from the file.
- A production deployment succeeded with the **replacement token**.
- A rollback test succeeded with the **old token**.
- Access logs are missing for the first part of the exposure.

What must happen before we can declare containment and recovery?

Revoke the old token; verify deployment and rollback with the replacement.

Rotation is complete only when the old credential no longer works

The replacement works

- Every inventoried consumer succeeds with it
- Logs show consumers have moved to it

The old credential fails

- Direct checks show the accepting service rejects it
- Logs show no accepted use after retirement

For Tiny Notes:

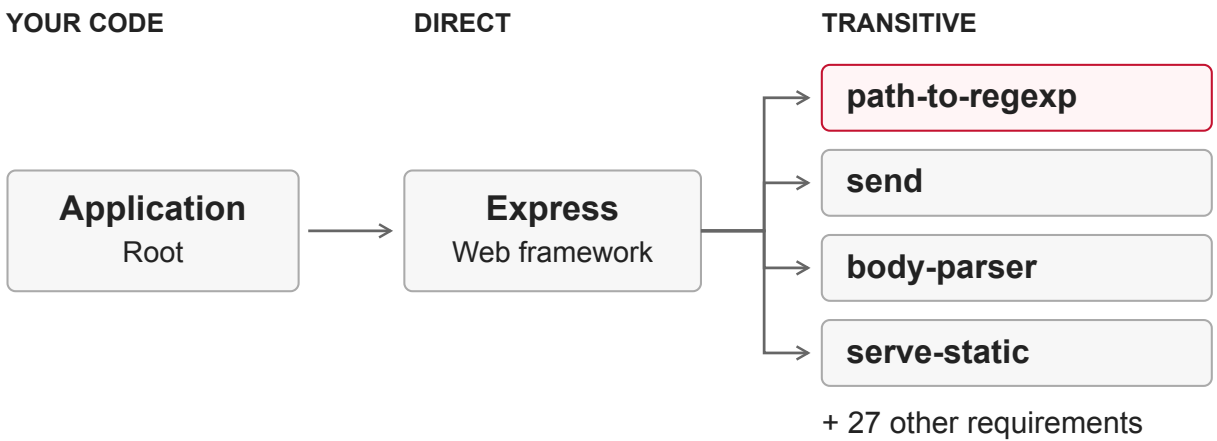
Update stored password → Reload every application instance → **Disable old password in database**

Key Concepts: Secrets and Access

Secrets vs. identifiers	classify values by the access they grant
Access decisions	identify principal and credential; distinguish authentication from authorization
Scope and lifetime	limit permissions and validity; explain the remaining blast radius
Storage and delivery	choose storage, bootstrap access, and runtime delivery
CI hardening	isolate untrusted code; pin actions; minimize permissions
Exposure evidence	track copies and consumers; distinguish exposure from confirmed use
Containment and rotation	revoke leaked access; verify consumers work and old credentials fail

Dependencies and Release Engineering

Your dependencies have dependencies



- **Direct dependency:** a package your project requires (e.g., Express)
- **Transitive dependency:** a package brought in by another dependency (e.g., path-to-regexp through Express)

The lock records direct and transitive versions

Inside the Express repository

Share the lock to repeat the resolver's selections.

Express package.json → `"send": "0.19.0"`

CAPTURED PACKAGE-LOCK.JSON

```
"node_modules/send": {
  "version": "0.19.0",
  "resolved": "…/send-0.19.0.tgz",
  "integrity": "sha512-dW41u5Vf…"
}
```

Selected fields; URL and hash abbreviated.



version

The selected version.



resolved

Registry tarball to fetch.

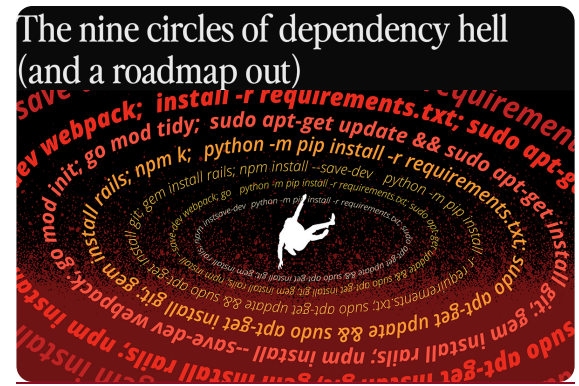


integrity

Hash of the expected bytes.

The nine circles of dependency hell

1. **Limbo:** installed packages ≠ recorded packages
2. **Lust:** adding packages without checking them
3. **Gluttony:** too many dependencies
4. **Greed:** competing package managers
5. **Wrath:** needed package or version unavailable
6. **Heresy:** changing dependency behavior at runtime
7. **Violence:** breaking minor or patch releases
8. **Fraud:** circular dependencies
9. **Treachery:** conflicting shared-library requirements



Matt Rickard · Sourcegraph · 2021

Read the article ↗

SemVer labels compatibility intent

`formatName()` currently returns a string · starting version **2.1.3**

2.1.4

PATCH

Fix unwanted spaces.
Keep the promised API.

`'Ada ' → 'Ada'`

2.2.0

MINOR

Add an optional setting.
Existing calls still work.

`{ order: 'last-first' }`

3.0.0

MAJOR

Return an object.
Callers must adapt.

`{ text: 'Ada Lovelace' }`

Judge compatibility against the promised public API.

Version labels guide us; tests check our application

Version labels can be wrong

- Patch and minor releases can accidentally break working code.
- A version number cannot guarantee compatibility with your application.

Test the behavior you depend on

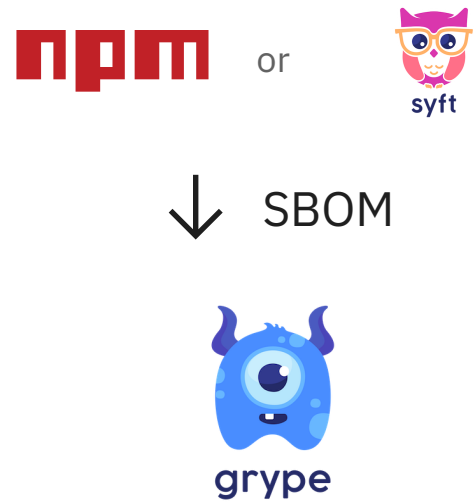
- Run tests that exercise how your application uses the dependency.
- Check expected results—not just whether the application starts.

For our service

Does `GET /notes` still return `200` and `preview` after the update?

Inventory packages, then check for known vulnerabilities

- **Record packages and versions** in a software bill of materials (SBOM).
  - npm: project dependencies
  - Syft: OS + application packages
- **Match against advisories:** Grype
- **New advisory?** Scan again.
  - **New build?** Regenerate the inventory.



Diagnosis helps set repair priority



Affected version
Is our installed version in the advisory's affected range?



Reachability
Can our application run the vulnerable operation?



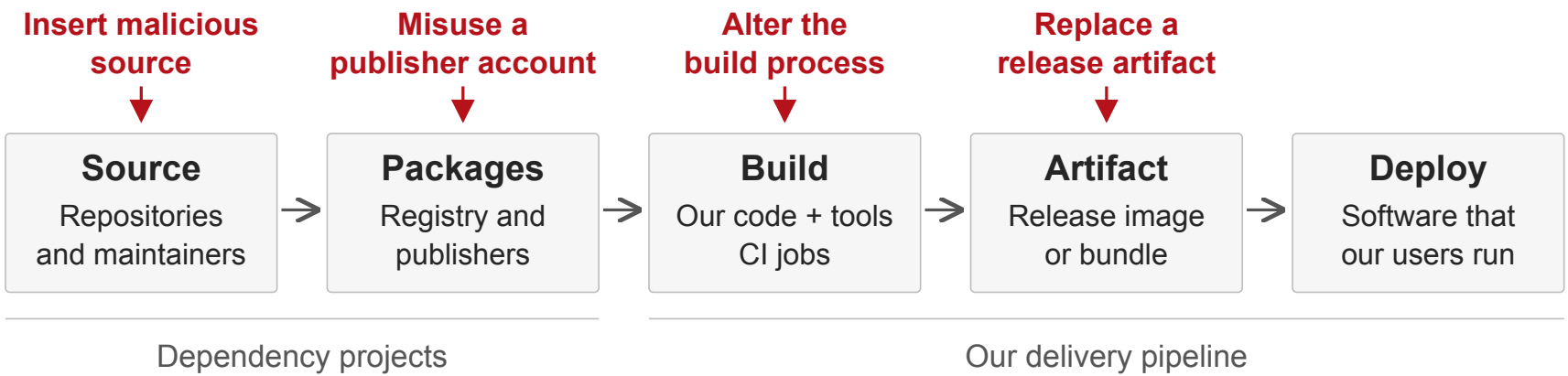
Exposure
Can an attacker supply the input or access needed to trigger it?

Investigate exposure to set urgency; keep the repair scheduled.

Mitigation leaves the flaw in place

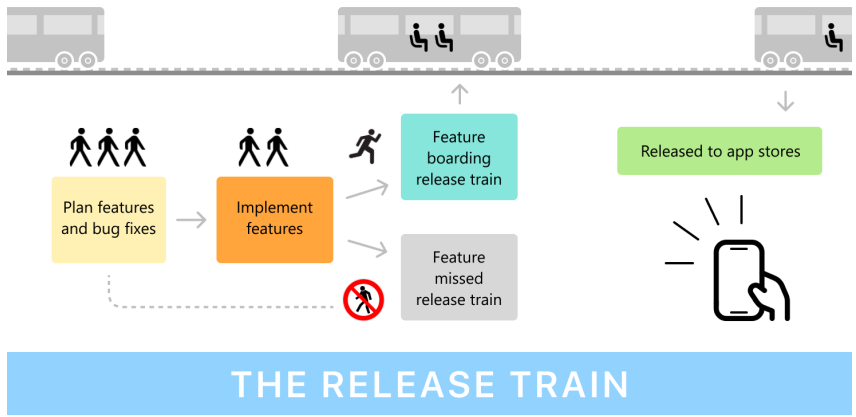
- A **mitigation** limits exploitation or its impact while the vulnerable code remains
  - use it while preparing and testing a repair
  - check that it blocks exploitation and assess its side effects
  - **Example (Express):** restrict route parameters or cap URL length.
- A **patch** changes the code to repair the flaw
  - **keep the patch scheduled:** new routes, configuration, or access can defeat today's mitigation.

Attackers can target any stage of the chain



- **Integrity** checks expected bytes; **provenance** connects an artifact to its source and build.
- Neither proves the code is benign. Tests and scans also have limited coverage.

Ready changes catch the release train



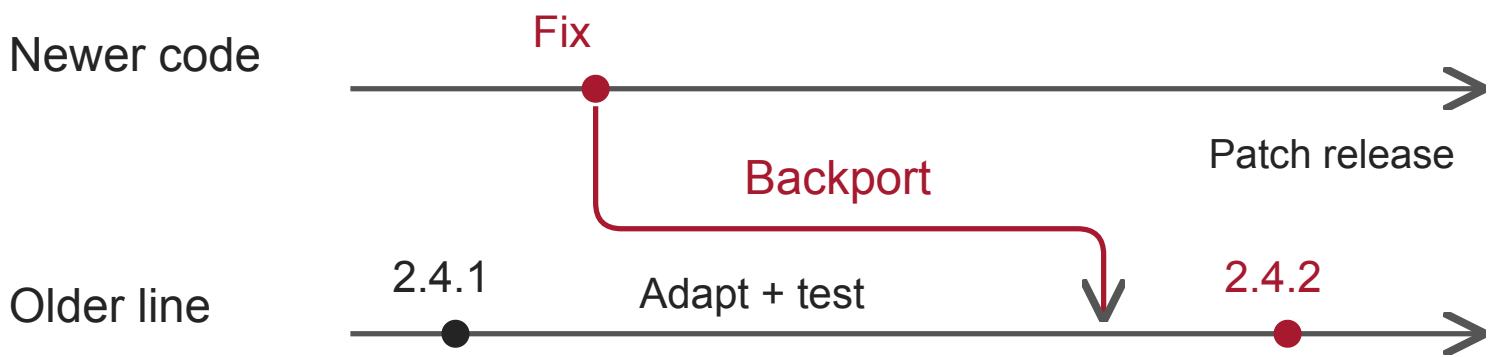
Cadence sets the schedule. A **release train** takes ready changes; late features wait.

Feature freeze: stop adding features to this release; test and fix blockers. Urgent fixes may need an extra release.

Illustration: Runway · the release-train analogy

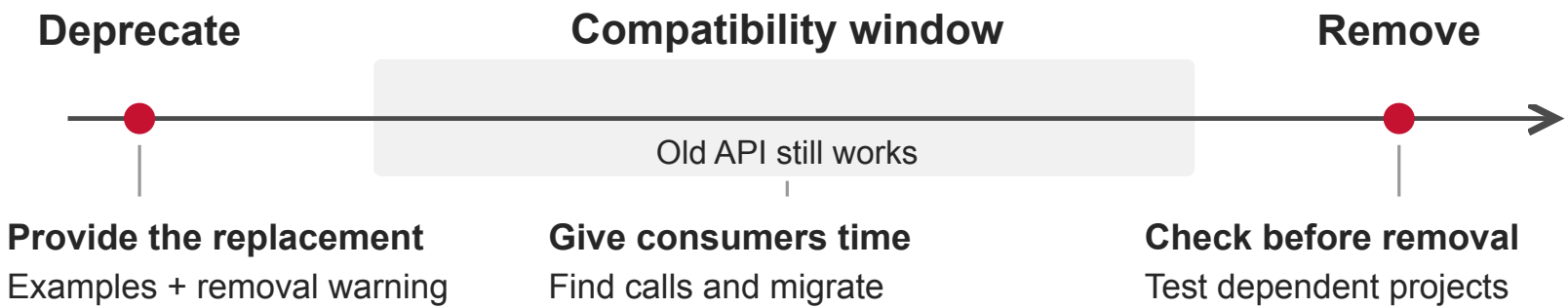
Backports bring fixes to older release lines

A **release line** is a series of versions maintained together.



- **Backport:** adapt and test the fix on the older line, then publish its patch.
- **Support policy:** state covered lines, fixes, and end dates; **EOL** ends support.

Give consumers time to stop using an old API



- **Deprecation:** an interface is being retired, but it still works.
  - Warn developers when they use it (e.g., via compiler or runtime warnings)

Recall: who waits for the Tiny Notes security fix?

The sharing API exposes private documents without sign-in. **Disclosure in 48 hours; logs are inconclusive.**

Line	Support commitment	Fix evidence
3.x	Full support	Checked fix ready
2.x	Security fixes; preserve integrations	Breaking prototype; compatible fix takes 2–4 days
1.x	Support ended	No patch estimate

Temporary protection: disable sharing on 2.x; disconnect 1.x. Both disrupt work.

What can we deploy or publish now? What must administrators do?

The same fix needs different release plans

Case	Constraint	Release response
GitLab	Operator or customer controls the server	Customer upgrade guidance
NumPy	Other products embed the library	Time for downstream preparation
Firefox	Desktop users need stability; organizations need time	Stabilization and support overlap

Key Concepts: Dependencies and Releases

Resolution	trace direct and transitive constraints; distinguish manifests and lockfiles
Compatibility	apply semantic versioning to the public API; explain caller impact
Vulnerability repair	check affected ranges, dependency paths, and reachability
Maintenance	choose update timing, grouping, and checks; plan urgent fixes
Supply-chain trust	trace attack paths; explain the limits of provenance, scans, and tests
Release decisions	publish checked artifacts; plan release trains, freezes, and notes
Support and adoption	apply release-line, LTS, and EOL policies; plan backports and upgrades

Delivery, Environments, and Deployment

Continuous Integration

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

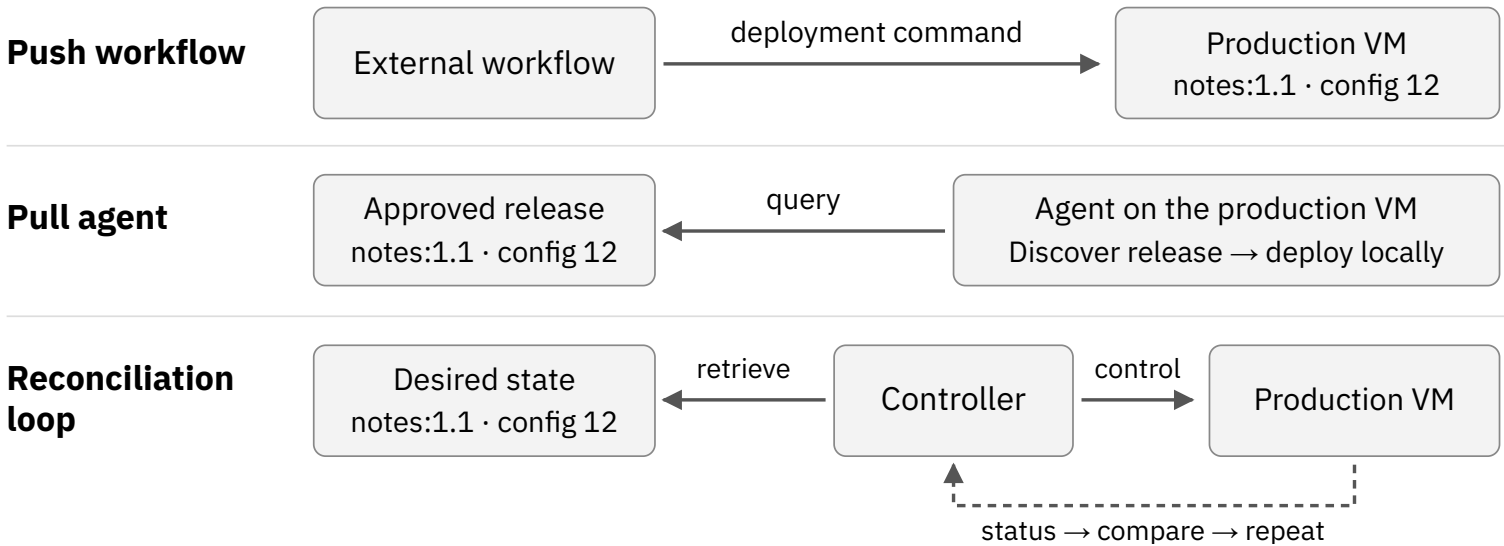
Continuous Delivery

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

Continuous Deployment

Deploy each qualifying change to production automatically.

Three designs can deploy the same release



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

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

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.

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

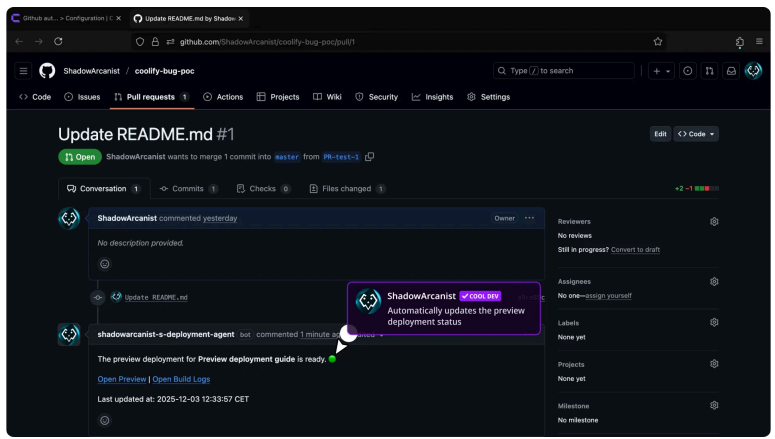


OpenGitOps principles

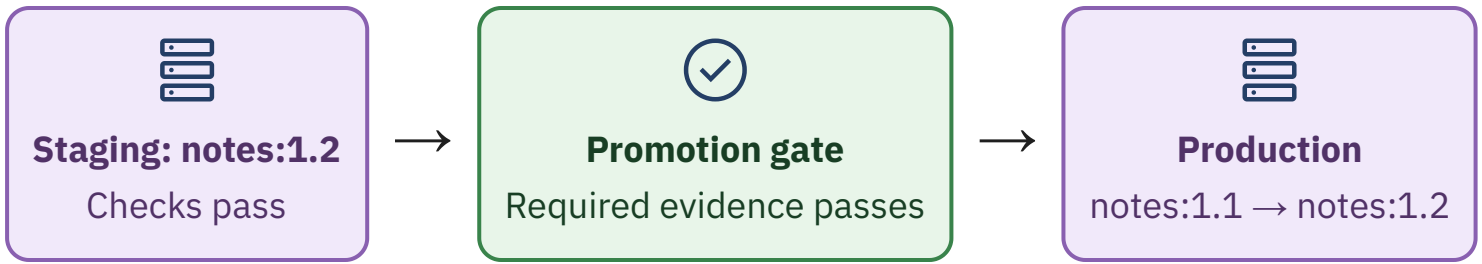
Environments support different stages of delivery

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



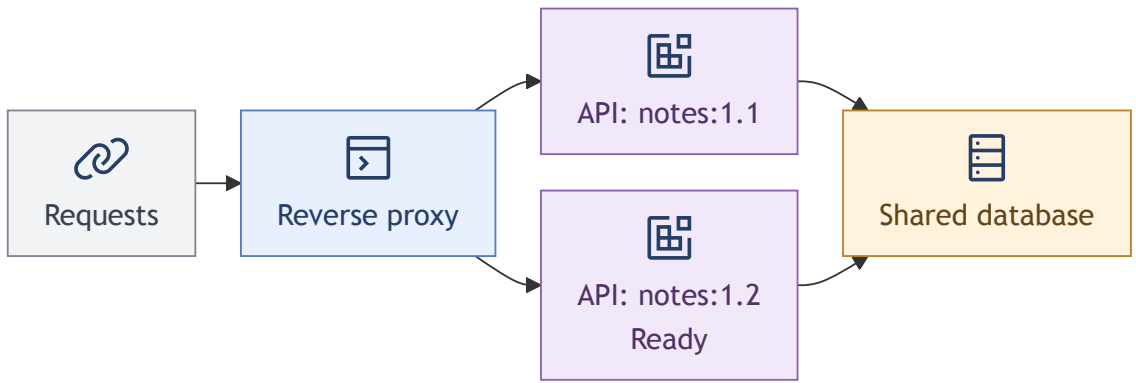
Promotion advances a checked artifact



- 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

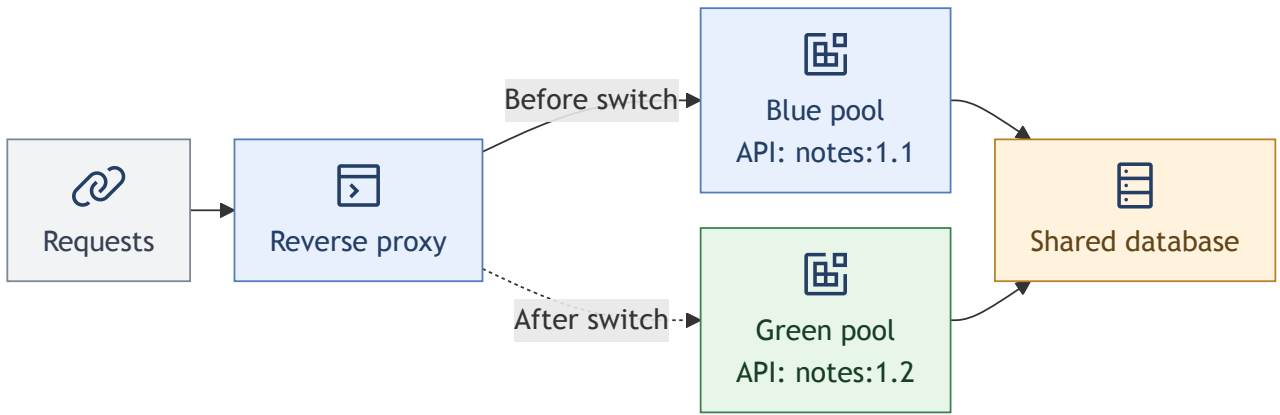
Rolling replacement keeps instances serving

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



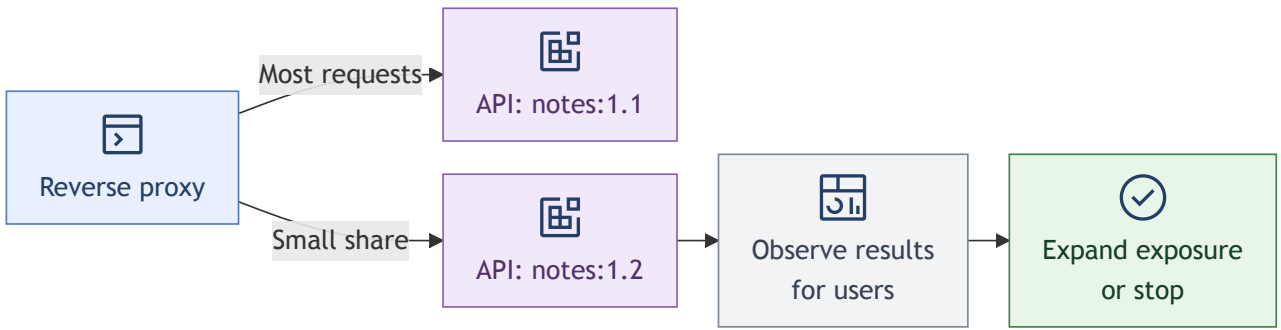
- **Rolling replacement** updates instances in batches: ready `notes:1.2` instances take traffic as `notes:1.1` instances leave. Repeat until all run `notes:1.2`

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

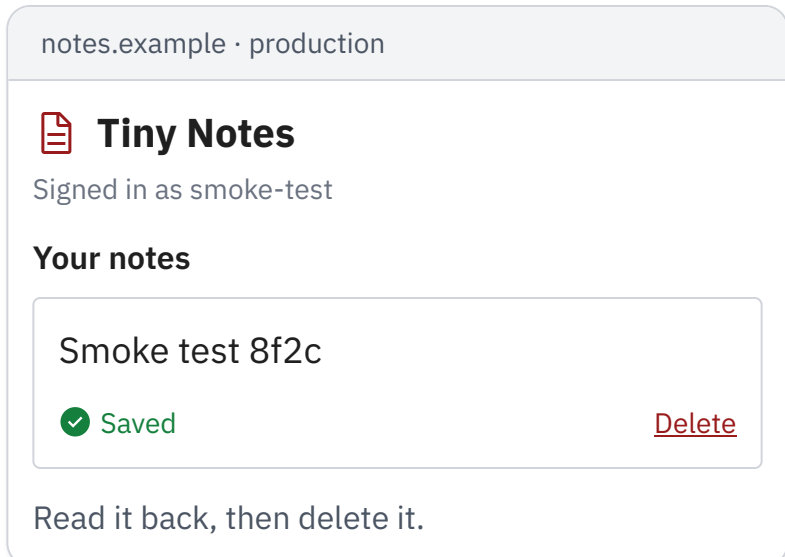


- Canaries send a small share of real requests to `notes:1.2`, observe results, then decide whether to expand
  - increases exposure **based on results**; gradual replacement alone does not.
- **More on this in the second half of the course**

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.

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

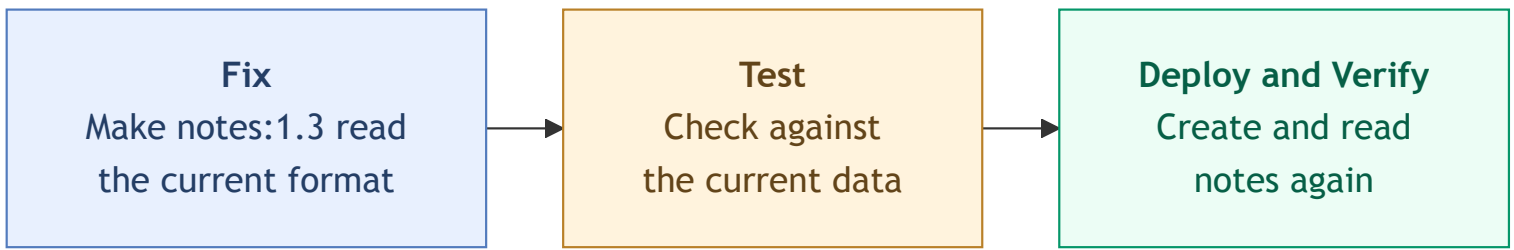
`notes:1.2` wrote a format that `notes:1.1` cannot read.

The old release still fails against the changed data.

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

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**.

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.

Recall: the deployment-design breakout

Scenario	Constraint or risk	One possible decision
Hospital	Shared medication records; new status	Check compatibility, then replace gradually
Satellite	Interrupted uploads; hours offline	Finish upload before switching; recover locally
Game	Active matches; lasting player inventories	Drain matches; check inventory compatibility
Self-checkouts	No inbound connections; active payments	Fetch updates; apply in small batches when idle

What would we still need to establish before using each approach?

Key Concepts: Delivery and Deployment

CI, delivery, deployment	distinguish checking changes, being ready to deploy, and deploying automatically
Push, pull, platforms	trace who initiates, needs access, and reports status
GitOps and reconciliation	predict drift repair; keep emergency changes in desired state
Environments and promotion	isolate data and access; promote the same checked digest
Replacement and exposure	compare in-place, rolling, blue-green, and canary deployments
Verification	distinguish a finished job, a ready instance, and a working user operation
Recovery	contain impact; choose rollback or rollforward; account for persistent state

Monitoring and Observability

The Three Pillars of Telemetry

Logs

Record individual events.

```
14:02:11 ERROR
Save failed
db_timeout
```

Metrics

Measure behavior over time.

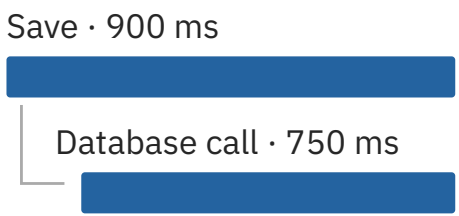
5% failed



of saves in the last minute

Traces

Follow a request across operations.



- Each **signal** gives us a different view of system behavior.

Structured logs make events queryable

```
logger.error("Save failed", {
  operation: "save_note",
  outcome: "failure",
  reason: "db_timeout",
  duration_ms: 900,
});
```

```
{
  "level": "error",
  "message": "Save failed",
  "service": "api",
  "timestamp": "2026-09-28T14:02:11Z",
  "operation": "save_note",
  "outcome": "failure",
  "reason": "db_timeout",
  "duration_ms": 900
}
```

A correlation ID groups one request's events

```
7f3a9c2e  api            Save started
b6d104af  api            Save started
7f3a9c2e  authorization  Access allowed
7f3a9c2e  api            Save failed: db_timeout
```

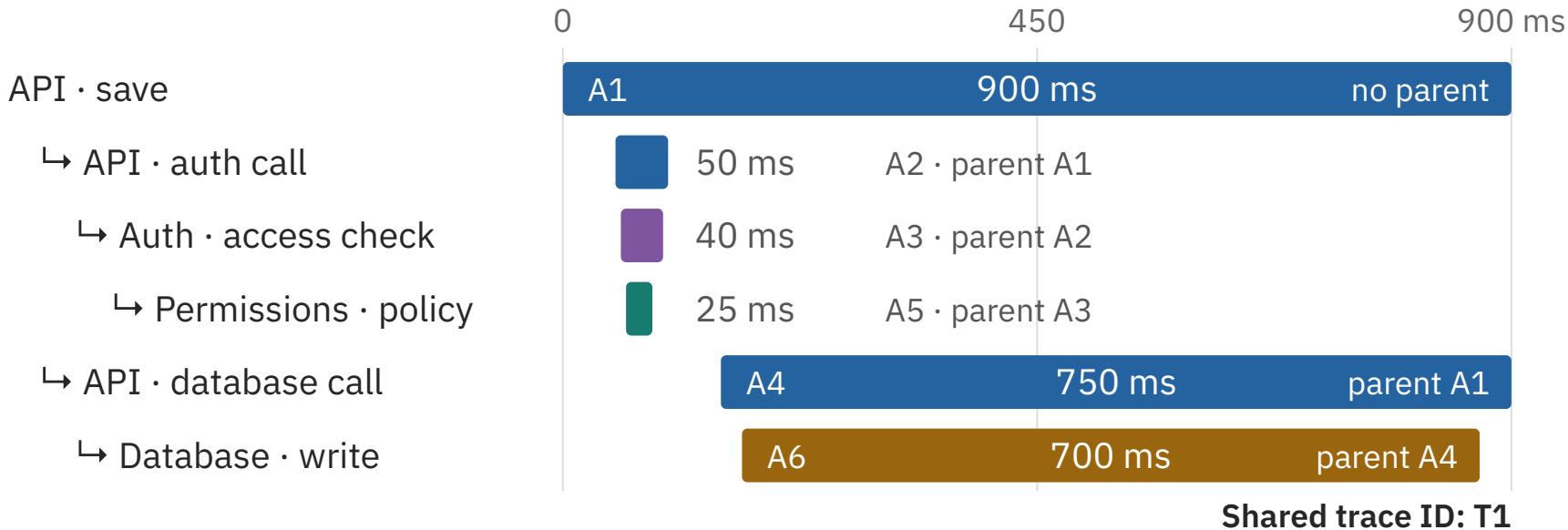
At API entry, once per request:

```
const requestId =
  crypto.randomUUID();
```

Keep the ID in the request's logs and pass it to downstream services.

A **correlation ID** groups one request's events across services.

A trace follows one request



The trace ID groups these operations into one trace.

Span IDs identify operations; parent IDs connect them.

Three categories answer different questions



Business

Are users succeeding?

Streaming: playback starts per second.



Application

Is the software working?

Streaming: playback API errors and request durations.



Infrastructure

Are resources healthy?

Streaming: server CPU, memory, and network utilization.

Choose the metric type to match the question



Counter

Events accumulated so far

Playback starts
Failed API requests



Gauge

A quantity right now

Available memory
Queue length



Histogram

How observations are distributed

Request durations
Response sizes

Example: each question points to a metric

STAKEHOLDER
Upload team

GOAL
Users see their uploaded photos quickly.

STATUS
Are users waiting less than last month?

METRIC
p90 time to first render, by month

DIAGNOSIS
Which stage adds the most delay?

METRIC
Time spent in each stage, per upload

SCOPE
Are some users waiting much longer?

METRIC
p90 wait by region and device type

These metrics are **proxies**: observable evidence about a quality (“quickly”) that we cannot measure directly.

Check that a metric is reliable and valid

RELIABILITY

Is it consistent?

Would the method give the same result under the same conditions?

✗ UNRELIABLE

Timer starts at tap on some phones, at the server on others.

VALIDITY

Is it on target?

Does the method measure the quality we intend to assess?

✗ INVALID

Timer covers processing only, not the user's full wait.

Five questions put a number in context



Target

Is this good enough?



Trend

Is it getting better or worse?



Comparison

Is this unusual?



Segmentation

Where is it concentrated?



Consequence

What does it cost people?

Running example: Friday's mobile uploads had a p90 time until first render of **16 s**.

Recall: choose evidence for a CI decision

Goal: improve CI feedback so developers can act sooner.

1. Choose a decision and a question that would inform it.
2. Define a metric: observations, calculation, population, and time window.
3. Explain how the number could improve while the goal does not.

Example: completed runs get faster after slow runs are canceled.

What evidence would expose the work developers are still waiting for?

DORA measures throughput and instability



SPEED AND RECOVERY

Throughput

How quickly does change move through the system?

Change lead time
Time from commit to production deployment.

Deployment frequency
Deployments per period, or time between them.

Failed deployment recovery time
Time to recover from a deployment requiring immediate intervention.



FAILURE AND REWORK

Instability

How often does change create more work?

Change fail rate
Share of deployments that require immediate intervention.

Deployment rework rate
Share of unplanned deployments caused by a production incident.

Key Concepts: Monitoring and Observability

Monitoring vs. observability	compare checks for known conditions with open-ended investigation
Logs and traces	choose events, fields, and IDs; record outcomes; exclude secrets
Metrics	distinguish business, application, and infrastructure metrics; interpret rates and percentiles
Collection	trace instrumentation, export, and push or scrape; account for sampling
Goal–Question–Metric	define calculation, population, and time window; assess validity and incentives
Dashboards	compare targets and trends; expose stale data and link to evidence
Delivery performance	interpret the five DORA measures

🚀 Today, we learned how to ...

- Explain the key distinctions in each exam topic
- Interpret evidence: what it shows and what it does not
- Recommend an action and justify it with evidence, constraints, and a tradeoff