

Safe Evolution of APIs and Data

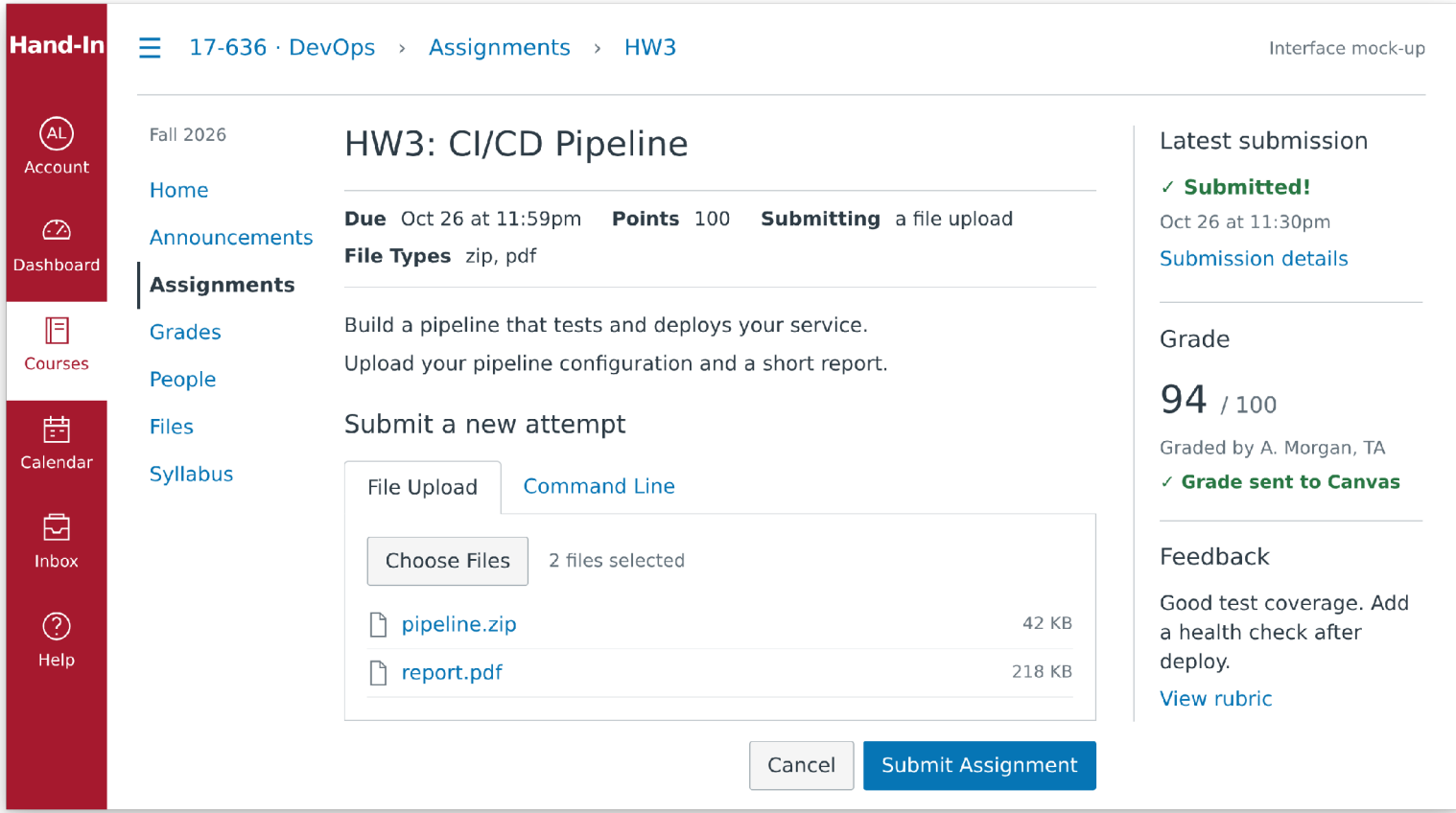
Chris Timperley • Monday, October 26, 2026

Administrivia

- **Homework 4B:** Candidate will be supplied on a PR later today
  - traffic should already be hitting your server
  - new events will start appearing
- No lab this week
- Guest lecture from **Michael Hilton** on Wednesday

🎯 Learning Goals

- Diagnose which consumers or stored data, including hidden ones, a change would break, and whether the break is structural or semantic
- Plan a transition with safe intermediate states, a source of truth in each, and the evidence required before each step
- Identify where rollback stops being enough, and choose to roll forward, re-derive, restore, or compensate
- Compare strategies for retiring an old contract and justify one for the consumers that remain


Hand-In
Account
Dashboard
Courses
Calendar
Inbox
Help
17-636 · DevOps › Assignments › HW3
Interface mock-up
Fall 2026
Home
Announcements
Assignments
Grades
People
Files
Syllabus
HW3: CI/CD Pipeline
Due Oct 26 at 11:59pm Points 100 Submitting a file upload
File Types zip, pdf
Build a pipeline that tests and deploys your service.
Upload your pipeline configuration and a short report.
Submit a new attempt
File Upload
Command Line
Choose Files
2 files selected
pipeline.zip
42 KB
report.pdf
218 KB
Cancel
Submit Assignment
Latest submission
✓ Submitted!
Oct 26 at 11:30pm
Submission details
Grade
94 / 100
Graded by A. Morgan, TA
✓ Grade sent to Canvas
Feedback
Good test coverage. Add a health check after deploy.
View rubric

You run Hand-in, a course-submission service



Submit

Students upload from the **web UI or the handin CLI**

Grade

Staff grade each submission

Sync

The **LMS grade sync** sends grades to a learning management system, such as Canvas

- Every submission belongs to exactly one `student_id`.
- All of these use Hand-in's HTTP API; any course account can create an API token.
- **12 instances share 1 database**; rolling deploys replace **2 at a time** over about **10 minutes**.

Instructors have requested support for group submissions

One submission should be shared by several students. Existing courses, clients, and past submissions must keep working.

Finding What a Change Breaks

Who depends on the old representation?

The PR replaces `student_id` with `members`, and every check passes

Open

Pull request #214

Add group submissions

group-submissions → main

Author

The IDE updated every caller, and the database migration is included. It ships Friday.

Types and tests passed

`GET /submissions/812`

Before

```
{ "id": 812, "student_id": "ana" }
```

After

```
{ "id": 812, "members": ["ana"] }
```

Who breaks on Monday, how, and what would you need to find out?

Debrief: The checks saw only callers in the codebase

Consumer	How it uses `student_id`	Old versions last
Web UI	Deployed with the service	Same release
`handin` CLI	Requires it in each listed item; sends it on submit; ignores unknown fields	Weeks, on laptops
LMS grade sync	Identifies whose LMS gradebook entry to update	Until its separate deployment
TA script	Counts submissions per assignment	Unknown, untracked
Old instances	Read and write the old schema	Minutes, each rollout
Stored rows	Two million, each with one `student_id`	Until converted

Check compatibility in both directions

Backward compatible

Old producer → **New reader**

A **new** version still accepts what **old** versions produce: old clients' requests and old stored data.

⊗ The new server rejects the old CLI's `student_id` submission.

Forward compatible

New producer → **Old reader**

An **old** version still handles what **new** versions produce: new responses and new stored data.

⊗ The old CLI crashes reading a response without `student_id`.

- During any rollout, old and new versions coexist, so a change usually needs **both**



KEY CONCEPT

Tolerant reader

A consumer that reads only the fields it needs and ignores anything it does not recognize.

The provider can **add** fields while preserving the fields and meanings the reader needs. For **one-student submissions**, Hand-in keeps `student_id` alongside `members`. The old CLI reads `student_id` and ignores `members`; the new CLI reads `members`.

Martin Fowler · Tolerant Reader

Changing an API Without Breaking Clients

How can old and new API contracts work while clients migrate?

Add groups while old CLIs remain installed

- You control when the backend, web UI, and LMS sync are updated. Students choose when to upgrade their CLIs.
- An API's **contract** is what clients rely on: its fields and what they mean. Old requests and responses use `student_id`, meaning the submission's **sole owner**.

Define what old and updated clients get, how Hand-in tells them apart, and your rollout order.

Debrief: Keep both contracts available while clients move

Decision	One workable design
Existing clients	Keep `student_id` for individual submissions; no version header
Updated clients	Use `members` for individuals and groups; send `API-Version: 2`
Old request involving a group	An explicit upgrade error under the pilot's support policy
Release order	Serve both → update UI/sync → enable pilot groups
Remove the old API	A later decision based on remaining use and a retirement policy

Check behavior as well as response shape

Structural break

A request or response no longer has the **structure** its consumer expects: a required field is missing or has a different type. It can **fail loudly**.

A1: the old CLI crashes without `student_id`.

Semantic break

The response has the expected fields, but their **meaning changes**. A shape check can pass while the result is **silently wrong**.

For a group, returning Ana as `student_id` quietly changes “sole owner” to “uploader.”

Version the API so clients migrate separately

- **API versioning:** serve both contracts; let each request select one

Version 1: no header

```
{ "id": 812, "student_id": "ana" }
```

Individual submissions use `student_id`.

Version 2: `API-Version: 2`

Individual

```
{ "id": 812,
  "members": ["ana"] }
```

Group

```
{ "id": 813,
  "members": ["ana","ben","chen"] }
```



KEY CONCEPT

Parallel change

Replace an interface in three phases: expand, migrate, then contract. Keep both forms available while consumers migrate, then retire the old interface.

Hand-in: **expand** by serving both API versions; **migrate** the UI, sync, and CLIs to `members`; **contract** by retiring the old API. Each intermediate state must work for the consumers still supported there.

Danilo Sato · Parallel Change

Migrating Data on a Live System

What makes each step safe, and how do you know?

GitHub's ORM still queried a dropped column

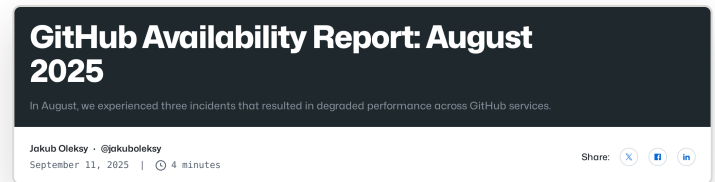
ORM model still lists the column

↓

Generated SQL still names it

↓

Database rejects the query



GitHub Blog · September 11, 2025

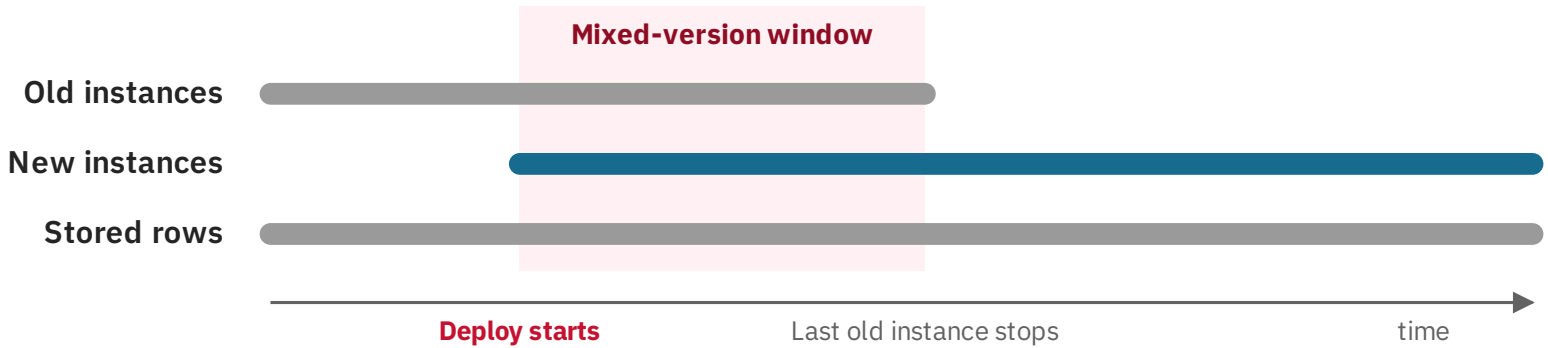
The **object-relational mapper (ORM)** still generated queries naming the removed columns.

- **Aug 5:** ≈4% of web/API traffic failed at peak.
- **Aug 27:** up to 77% of Copilot requests failed.

GitHub temporarily blocked all column drops.

A schema's consumers include infrastructure nobody lists.

Old and new instances share one database during every rollout



A deployment replaces **two of 12 instances at a time** over about **ten minutes**. Old and new versions share the database throughout.

Pramod Sadalage & Martin Fowler · Evolutionary Database Design

A colleague's plan moves the data in five steps: is it safe?

1. Add the `submission_members` table.
2. Backfill membership rows for every past submission.
3. Deploy code that reads and writes only `submission_members`.
4. Enable group submissions for the pilot course.
5. Drop the `student_id` column.

- **Two million** past submissions: a backfill takes hours.
- Students keep submitting, including during deployments.
- Hand-in cannot go offline.
- Groups start in the **pilot course**; a group has no `student_id`.

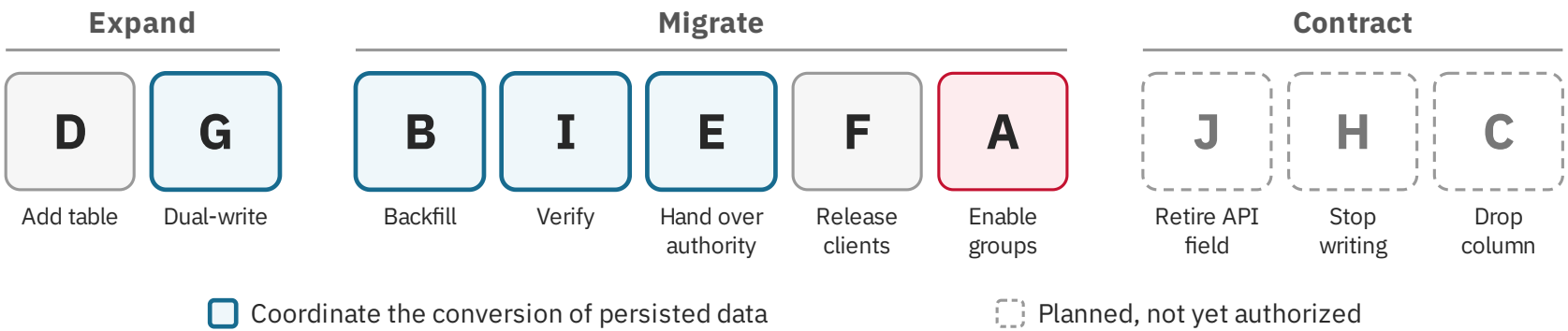
Is each step safe? Explain the risk or missing precondition.

Keep submissions working throughout

- Repair the plan on worksheet **3b**, keeping your Activity 2 API contracts working.
- **Source of truth:** the authoritative copy, which determines the correct value; the other copy is maintained from it.
- **Authority flag:** G deploys code that supports either copy as the source of truth; E switches between them without redeploying.

If the migration pauses after any step, can students keep submitting?

Debrief: The plan breaks because the data keeps changing while it runs

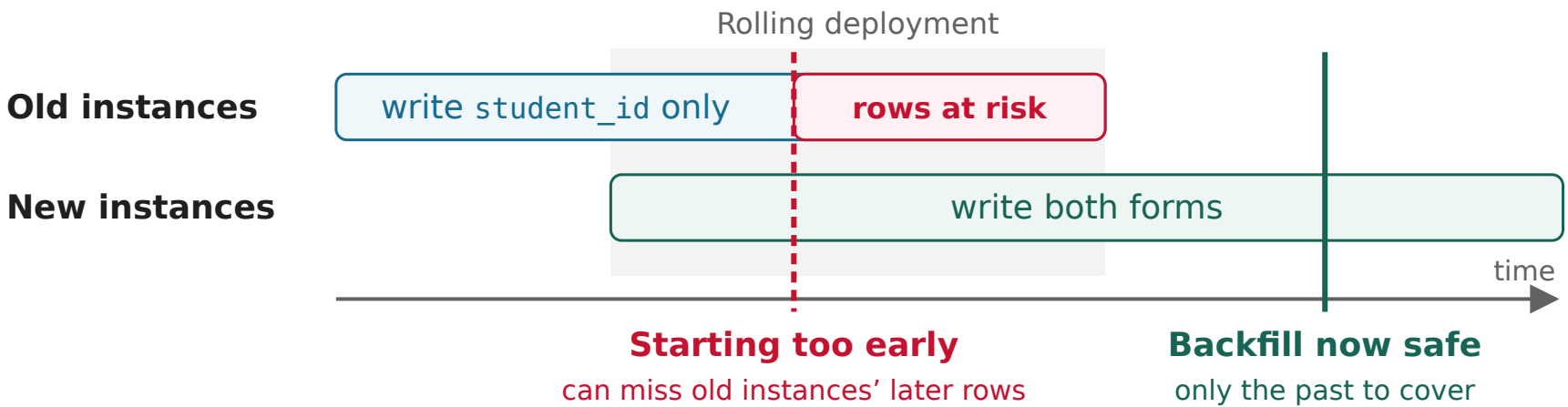


Expand–contract: add support for the new form, migrate data and consumers, then remove the old form once its consumers have moved.

Migrating stored data adds these responsibilities

Step	What it means	Why data needs it
Dual-write	Maintain both representations for ongoing writes	Prevent new gaps while converting existing data
Backfill	Convert existing rows to the new form, in resumable batches	Old rows outlive every deployment
Verify	Check an invariant that relates the two forms	Filled-in is not the same as correct
Hand over authority	Make the new form the source of truth for every writer	A disagreement needs exactly one answer

Write both forms everywhere before backfilling the past



A **backfill** converts existing data. First maintain both forms for ongoing writes; then convert the past in **small batches that can pause, resume, and rerun** without overwriting newer data.

Give readers and writers one source of truth

The **source of truth** (the authoritative copy) is the one that wins when two copies disagree; the other is derived from it and can be rebuilt.

Hand-in stage	Authoritative copy	The other copy
Writing both forms	`student_id`	Membership rows, derived from it
After the switch	Membership rows	`student_id`, derived for individuals
Groups enabled	Membership rows	`student_id`, empty for groups

Hand-in deploys both modes, then flips a flag. Groups wait until **all 12 writers** switch.

Enable new data only when every writer supports it.

Verify migrations with invariants

An **invariant** is a condition every valid state must satisfy. Verify that migrated data preserves the facts and relationships the system requires.

Hand-in before groups: each submission has exactly one member matching `student_id`.

Check presence

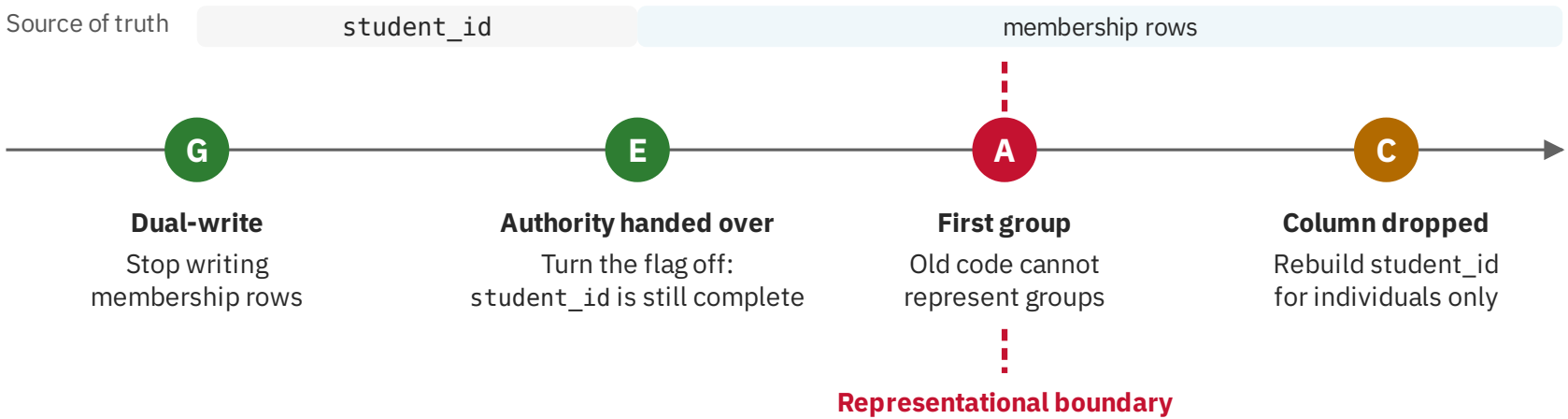
```sql
SELECT count(*) FROM submissions s
WHERE NOT EXISTS (
  SELECT 1 FROM submission_members m
  WHERE m.submission_id = s.id);
-- 0: none missing
```

A row naming Ben passes this check even if the original owner was Ana.

Check the required relationship

```sql
SELECT s.id FROM submissions s
LEFT JOIN submission_members m
  ON m.submission_id = s.id
GROUP BY s.id, s.student_id
HAVING count(m.student_id) <> 1
    OR max(m.student_id)
       <> s.student_id;
-- IDs that violate the invariant
```

Debrief: Check what recovery can preserve



Rollback requires old code and schema that can handle all current data.



KEY CONCEPT

Representational boundary

The first moment new data cannot be represented by the old model. Old code can no longer handle all current data correctly.

Hand-in crosses this boundary at its **first group submission**.

When a step fails beyond this boundary, how should we recover?

Recovering Partway Through a Migration

When a step goes wrong partway through, how do you recover?

Contain the failure, then choose a recovery

For your incident: identify checks, contain the failure, choose a recovery, and explain why.

Recovery	What it means
Roll back	Return to the previous version of the code or flag setting
Roll forward	Deploy a corrective change while continuing from the current state
Re-derive	Rebuild a derived copy from the source of truth
Restore	Bring back lost data from a backup
Compensate	Correct effects that have left the system, and tell the people affected

What you find

Incident 3 · one member cannot find the submission

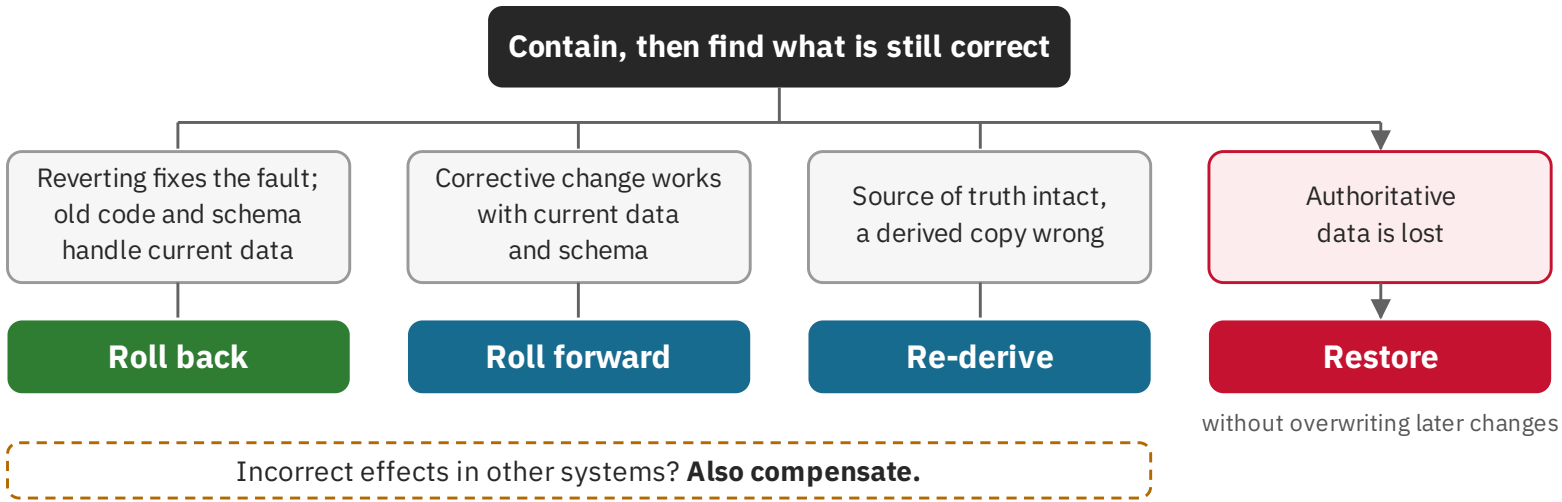
- Every member of every group is present in `submission_members`.
- The list's new query joins members through course enrollment.
- The members who cannot find their submission all enrolled after the add/drop deadline.

Incident 5 · membership rows deleted at 09:00

- Nightly backup: 02:00. A separate point-in-time restore to 08:59 is available.
- The deletion log contains row IDs, not row contents.
- Between 02:00 and 09:00: 37 new submissions and 4 group membership changes.

Match recovery to what remains correct

A forward fix may change code or data. More than one recovery option may apply.



Code rollback, data repair, and correction of external effects are separate decisions. One incident may require all three.

Debrief: Each recovery depends on what is still correct

Card	Where it happened	Recovery
1	After E: authority handed over	Turn the flag off
2	After B: backfill done	Re-derive from `student_id`
3	After A: groups enabled	Roll forward a fixed query
4	After A: groups enabled	Roll forward, and compensate in the LMS
5	After C: column dropped	Restore deleted rows from a separate copy

Cards 1 and 2 can use `student_id` because it is **complete, correct, and kept current**. After groups are enabled, the retained column represents only individual submissions.

Retiring the Old Contract

How and when should `student_id` go?

Keeping `student_id` slows every change

Groups work in the pilot. The LMS sync and current CLIs read `members`. The team wants `student_id` gone **before next academic year**.

What we know

- Contract tests from the CLI and LMS sync no longer mention `student_id`.
- About **3%** of requests come from old CLI versions.
- About **2%** call `?student_id=` with API tokens nobody recognizes.
- CLI telemetry reports which versions are active.

Migration is not retirement

- Old CLIs still need an upgrade path.
- Unknown scripts need an owner and a support policy.
- Passing contract tests does not establish that all consumers have moved.

Every kind of evidence misses something

Evidence	What it establishes	What it misses
Consumer contract tests	Published consumer expectations still pass	Unknown consumers and untested behavior
API request logs	Who calls which endpoints and parameters	Which response fields a client actually reads
Client version telemetry	Which client versions are active	Installed but dormant clients
Database query logs	Observed SQL that uses the column	Jobs that have not run; other environments

Ian Robinson · Consumer-Driven Contracts

Pick a retirement strategy per consumer

Strategy	What it means	Fits consumers that...
Deprecation window	Announce removal, warn users, then require a minimum version	You can reach and measure
Brownouts	Switch the old contract off for short, announced periods before a deadline	You know only by their traffic
Versioned endpoint	Keep the old version running for a stated support window	Move slowly
Permanent adapter	Translate the old contract to the new one indefinitely	Cost more to find than to keep

Make every change a sequence of compatible states

Find the consumers
Hidden ones too; check backward and forward compatibility

Keep every state compatible
Parallel change; keep both contracts while consumers migrate

Migrate data in safe steps
Dual-write, backfill, verify invariants, hand over authority

Plan recovery past each boundary
Roll back, roll forward, re-derive, restore, compensate

Retire the old form on evidence
Deprecation windows, brownouts, versions, or adapters

🚀 Today, we learned how to ...

- Diagnose which consumers or stored data, including hidden ones, a change would break, and whether the break is structural or semantic
- Plan a transition with safe intermediate states, a source of truth in each, and the evidence required before each step
- Identify where rollback stops being enough, and choose to roll forward, re-derive, restore, or compensate
- Compare strategies for retiring an old contract and justify one for the consumers that remain