COMPANION GUIDE

From E2E to Contract Tests: A Practical Migration Path Beyond Microservices

A real migration from 20 full-stack E2E tests to layered contract coverage — including the failures that exposed why the change mattered.

Elena Kulgavaya · Lead QA Engineer, Source2Sea

Creator of Surety (contract testing framework) · qaexplained.com

Contents

Section 1 — The Starting Point: A Full E2E Suite 3

Section 2 — The Migration: From E2E to Contracts 11

Step 1 — Rewrite in Python 12

Step 2 — Isolate the Frontend 13

Step 3 — Prove the Backend Contract 15

Step 4 — Turn Frontend Requests Into Contracts 17

Step 5 — Move Feedback Into CI 18

Step 6 — Remove the E2E Safety Net 20

Section 3 — Examples in Practice 21

Example 1 — Renaming a Field 21

Example 2 — Renaming an Enum Value 25

Afterword — Where Else This Applies 29

SECTION 1

The Starting Point: A Full E2E Suite

Before any migration work began, talkvote's behavior was verified entirely through Playwright + TypeScript end-to-end tests. Eight spec files, twenty tests total, all running against the full application deployed somewhere real — with whatever pre-set data happens to be sitting in that environment, data the test suite can barely control or reset. This is the baseline the rest of this guide migrates away from.

> **WHY THIS MATTERS**
>
> Every example later in this guide refers back to these eight files. Keep them in mind as you read the migration steps — each step re-implements a slice of what's tested here, using a different tool for a different job.

> **AT A GLANCE**
>
> 8 spec files · 20 tests · 100% system-level · 0% mocked

The eight files

1. **`01-talks-list.spec.ts`**
   Confirms the seed list renders with at least eight talks, that each card shows a title, speaker, and vote count, and that the "Submit a talk" button navigates to the submission form. *(3 tests)*

2. **`02-filtering.spec.ts`**
   Filters the list by track, by level, and by status, checking every visible card carries the matching badge — plus a fourth test for the empty-state message when a filter combination matches nothing. *(4 tests)*

3. **`03-sorting.spec.ts`**
   Switches the sort control to "votes" and checks that the visible vote-count badges come back in descending order. *(1 test)*

4. **`04-submit-talk.spec.ts`**
   Fills out the submission form and confirms the redirect to the new talk's detail page, then — in a second, independent test — submits again and confirms the new talk shows up back on the list. *(2 tests)*

5. **`05-submit-validation.spec.ts`**
   Confirms an empty submission is blocked from navigating away, and that a minimal-but-valid submission (short title, short abstract) succeeds. *(2 tests)*

6. **`06-voting.spec.ts`**
   Casts a vote and checks the count increments and the button label changes, that the button becomes disabled after voting, and that the "Already voted" state survives a full page reload. *(3 tests)*

7. **`07-admin.spec.ts`**
   Confirms the admin queue lists submitted talks, that approving one removes it from the queue, and — the longest test here — that an approved talk shows up filtered as "approved" back on the main list. *(3 tests)*

8. **`08-talk-detail.spec.ts`**
   Reads the title and speaker off the list, then checks they match on the detail page. A second test checks the back link. *(2 tests)*

01-talks-list.spec.ts

Confirms the seed list renders with at least eight talks, that each card shows a title, speaker, and vote count, and that the “Submit a talk” button navigates to the submission form.

e2e/01-talks-list.spec.ts — PLAYWRIGHT · TYPESCRIPT

```ts
import { test, expect } from "@playwright/test";

test.describe("Talks list", () => {
  test("displays talks list with seed data", async ({ page }) => {
    await page.goto("./");
    await expect(page.locator("h1")).toContainText("TalkVote");
    await page.waitForSelector("article");
    const cards = page.locator("article");
    expect(await cards.count()).toBeGreaterThanOrEqual(8);
  });

  test("shows talk title, speaker name and vote count on each card", async ({ page }) => {
    await page.goto("./");
    const firstCard = page.locator("article").first();
    await expect(firstCard.locator("h2")).not.toBeEmpty();
    await expect(firstCard).toContainText("by");
    await expect(firstCard).toContainText("👍");
  });

  test("navigate to submit page via button", async ({ page }) => {
    await page.goto("./");
    await page.click("text=Submit a talk");
    await expect(page).toHaveURL(/\/submit/);
  });
});
```

LOOK CLOSER

Three tests to describe a home page — and every one of them still needs a full browser and a running backend just to check it.

02-filtering.spec.ts

Filters the list by track, by level, and by status, checking every visible card carries the matching badge — plus a fourth test for the empty-state message when a filter combination matches nothing.

e2e/02-filtering.spec.ts — PLAYWRIGHT · TYPESCRIPT

```ts
import { test, expect } from "@playwright/test";

const waitForFilter = (page: Parameters<typeof test>[1] extends (args: { page: infer P }) => unknown
? P : never) =>
  page.waitForResponse(r => r.url().includes("/api/talks") && r.status() === 200);

test.describe("Filtering", () => {
  test("filters by track and shows only matching talks", async ({ page }) => {
    await page.goto("./");
    await page.waitForSelector("article");
    const totalCount = await page.locator("article").count();

    await Promise.all([
      waitForFilter(page),
      page.selectOption('[data-testid="filter-track"]', "qa"),
    ]);
```

code continues on next page

▲ continued — e2e/02-filtering.spec.ts

```
    const filteredCount = await page.locator("article").count();
    expect(filteredCount).toBeGreaterThan(0);
    expect(filteredCount).toBeLessThan(totalCount);

    // Every visible card should show the QA track badge
    const cards = page.locator("article");
    const count = await cards.count();
    for (let i = 0; i < count; i++) {
      await expect(cards.nth(i)).toContainText("qa");
    }
  });

  test("filters by level", async ({ page }) => {
    await page.goto("./");
    await page.waitForSelector("article");

    await Promise.all([
      waitForFilter(page),
      page.selectOption('[data-testid="filter-level"]', "beginner"),
    ]);

    const cards = page.locator("article");
    const count = await cards.count();
    expect(count).toBeGreaterThan(0);
    for (let i = 0; i < count; i++) {
      await expect(cards.nth(i)).toContainText("beginner");
    }
  });

  test("filters by status", async ({ page }) => {
    await page.goto("./");
    await page.waitForSelector("article");

    await Promise.all([
      waitForFilter(page),
      page.selectOption('[data-testid="filter-status"]', "approved"),
    ]);

    const cards = page.locator("article");
    const count = await cards.count();
    expect(count).toBeGreaterThan(0);
    for (let i = 0; i < count; i++) {
      await expect(cards.nth(i)).toContainText("approved");
    }
  });

  test("shows no results message when filters match nothing", async ({ page }) => {
    await page.goto("./");
    await page.waitForSelector("article");

    await Promise.all([
      waitForFilter(page),
      page.selectOption('[data-testid="filter-track"]', "devops"),
    ]);
    await Promise.all([
      waitForFilter(page),
      page.selectOption('[data-testid="filter-level"]', "beginner"),
    ]);

    // devops + beginner has no seed data
    await expect(page.locator("text=No talks found")).toBeVisible();
  });
});
```

LOOK CLOSER

The waitForFilter helper is there because filtering triggers a real network round trip — the test has to wait for that response before the page actually re-renders with the filtered set. Skip it, and the assertion runs against the old, stale list.

03-sorting.spec.ts

Switches the sort control to “votes” and checks that the visible vote-count badges come back in descending order.

e2e/03-sorting.spec.ts — PLAYWRIGHT · TYPESCRIPT

```ts
import { test, expect } from "@playwright/test";

test.describe("Sorting", () => {
  test("sorted by votes shows highest-voted talk first", async ({ page }) => {
    await page.goto("./");
    await page.waitForSelector("article");

    // Default sort is by votes (highest first)
    await page.selectOption('[data-testid="sort-select"]', "votes");
    await page.waitForTimeout(300);

    // Collect vote counts from visible cards
    const voteBadges = page.locator("article span[style*='font-weight: 700']");
    const texts = await voteBadges.allTextContents();
    const votes = texts.map(Number).filter((n) => !isNaN(n));

    // Votes should be in descending order
    for (let i = 0; i < votes.length - 1; i++) {
      expect(votes[i]).toBeGreaterThanOrEqual(votes[i + 1]);
    }
  });
});
```

> **LOOK CLOSER**
>
> A single, narrow assertion — run at the full price of a browser test anyway.

04-submit-talk.spec.ts

Fills out the submission form and confirms the redirect to the new talk's detail page, then — in a second, independent test — submits again and confirms the new talk shows up back on the list.

e2e/04-submit-talk.spec.ts — PLAYWRIGHT · TYPESCRIPT

```ts
import { test, expect } from "@playwright/test";

test.describe("Submit talk", () => {
  test("submits a valid talk and redirects to detail page", async ({ page }) => {
    await page.goto("./#/submit");
    await expect(page.locator("h1")).toContainText("Submit a Talk");

    await page.fill('[data-testid="input-title"]', "E2E Test Talk");
    await page.fill('[data-testid="input-speakerName"]', "Test Speaker");
    await page.fill('[data-testid="input-abstract"]', "This is an abstract for the E2E test talk.");
    await page.selectOption('[data-testid="input-track"]', "qa");
    await page.selectOption('[data-testid="input-level"]', "intermediate");

    await page.click('[data-testid="submit-button"]');

    // Should redirect to detail page of the new talk
    await expect(page).toHaveURL(/\/talks\/.+/);
    await expect(page.locator("h1")).toContainText("E2E Test Talk");
  });

  test("new talk appears on the list with status submitted", async ({ page }) => {
    await page.goto("./#/submit");
```

code continues on next page ▾

continued — e2e/04-submit-talk.spec.ts

```
    const uniqueTitle = `Auto Talk ${Date.now()}`;
    await page.fill('[data-testid="input-title"]', uniqueTitle);
    await page.fill('[data-testid="input-speakerName"]', "Auto Speaker");
    await page.fill('[data-testid="input-abstract"]', "Abstract for automated submission test.");
    await page.selectOption('[data-testid="input-track"]', "backend");
    await page.selectOption('[data-testid="input-level"]', "advanced");

    await page.click('[data-testid="submit-button"]');
    await page.waitForURL(/\/talks\/.+/);

    // Go back to list
    await page.goto("./");
    await page.waitForSelector("article");

    await expect(page.locator(`text=${uniqueTitle}`)).toBeVisible();
  });
});
```

LOOK CLOSER

The second test re-does the entire submit flow just to check the talk appears on the list afterward. That's real duplication of setup across tests, purely because there's no cheaper way to get the system into the right state.

05-submit-validation.spec.ts

Confirms an empty submission is blocked from navigating away, and that a minimal-but-valid submission (short title, short abstract) succeeds.

e2e/05-submit-validation.spec.ts — PLAYWRIGHT · TYPESCRIPT

```
import { test, expect } from "@playwright/test";

test.describe("Submit validation", () => {
  test("shows browser validation when required fields are empty", async ({ page }) => {
    await page.goto("./#/submit");
    // Click submit without filling any fields
    await page.click('[data-testid="submit-button"]');
    // Title input should have browser required validation triggered
    // (We check that no navigation happened)
    await expect(page).toHaveURL(/\/submit/);
  });

  test("clears field error when user corrects the value", async ({ page }) => {
    await page.goto("./#/submit");

    // Fill all required fields except abstract to trigger a short submission
    await page.fill('[data-testid="input-title"]', "T");
    await page.fill('[data-testid="input-speakerName"]', "S");
    // Don't fill abstract to rely on HTML required
    await page.fill('[data-testid="input-abstract"]', "Short abstract");
    await page.selectOption('[data-testid="input-track"]', "qa");
    await page.selectOption('[data-testid="input-level"]', "beginner");

    // Submit should work (all fields minimal but valid)
    await page.click('[data-testid="submit-button"]');
    await expect(page).toHaveURL(/\/talks\/.+/);
  });
});
```

LOOK CLOSER

This whole suite is really about the browser's built-in required-field validation — arguably not something that needs a real backend round trip to verify at all.

06-voting.spec.ts

Casts a vote and checks the count increments and the button label changes, that the button becomes disabled after voting, and that the “Already voted” state survives a full page reload.

e2e/06-voting.spec.ts — PLAYWRIGHT · TYPESCRIPT

```ts
import { test, expect } from "@playwright/test";

test.describe("Voting", () => {
  test("vote button increments vote count", async ({ page }) => {
    await page.goto("./");
    await page.waitForSelector("article");
    // Open the first talk
    await page.locator("article").first().locator("a[href*='/talks/']").last().click();
    await page.waitForURL(/\/talks\/.+/);

    const countEl = page.locator('[data-testid="vote-count"]');
    const before = parseInt(await countEl.textContent() || "0");

    await page.click('[data-testid="vote-button"]');

    // After voting button should change and count should increase
    await expect(page.locator('[data-testid="vote-button"]')).toContainText("Already voted");
    const after = parseInt(await countEl.textContent() || "0");
    expect(after).toBe(before + 1);
  });

  test("vote button is disabled after voting (within same session)", async ({ page }) => {
    await page.goto("./");
    await page.waitForSelector("article");

    await page.locator("article").first().locator("a[href*='/talks/']").last().click();
    await page.waitForURL(/\/talks\/.+/);

    await page.click('[data-testid="vote-button"]');
    await expect(page.locator('[data-testid="vote-button"]')).toBeDisabled();
  });

  test("reload page still shows Already voted state", async ({ page }) => {
    await page.goto("./");
    await page.waitForSelector("article");

    await page.locator("article").first().locator("a[href*='/talks/']").last().click();
    await page.waitForURL(/\/talks\/.+/);
    const url = page.url();

    await page.click('[data-testid="vote-button"]');
    await expect(page.locator('[data-testid="vote-button"]')).toContainText("Already voted");

    // Reload — localStorage persists
    await page.goto(url);
    await expect(page.locator('[data-testid="vote-button"]')).toContainText("Already voted");
  });
});
```

> **LOOK CLOSER**
>
> That third test — reload and recheck — exists because the voting state lives in the browser's localStorage. Worth protecting, but it's really testing what the browser remembers, not what the server returns.

07-admin.spec.ts

Confirms the admin queue lists submitted talks, that approving one removes it from the queue, and — the longest test here — that an approved talk shows up filtered as “approved” back on the main list.

e2e/07-admin.spec.ts — PLAYWRIGHT · TYPESCRIPT

```ts
import { test, expect } from "@playwright/test";

test.describe("Admin panel", () => {
  test("admin page shows submitted talks", async ({ page }) => {
    await page.goto("./#/admin");
    await expect(page.locator("h1")).toContainText("Admin");

    // Should have at least one submitted talk from seed data
    const rows = page.locator('[data-testid^="admin-talk-"]');
    await expect(rows).not.toHaveCount(0);
  });

  test("approving a talk removes it from admin queue", async ({ page }) => {
    await page.goto("./#/admin");
    await page.waitForSelector('[data-testid^="admin-talk-"]');

    const beforeCount = await page.locator('[data-testid^="admin-talk-"]').count();
    await page.locator('[data-testid^="approve-"]').first().click();

    // Wait for removal
    await page.waitForTimeout(500);
    const afterCount = await page.locator('[data-testid^="admin-talk-"]').count();
    expect(afterCount).toBe(beforeCount - 1);
  });

  test("approved talk appears as approved in the main list", async ({ page }) => {
    // Submit a new talk to have a fresh one
    await page.goto("./#/submit");
    const title = `Admin Approve Test ${Date.now()}`;
    await page.fill('[data-testid="input-title"]', title);
    await page.fill('[data-testid="input-speakerName"]', "Admin Tester");
    await page.fill('[data-testid="input-abstract"]', "Testing admin approval flow end to end.");
    await page.selectOption('[data-testid="input-track"]', "devops");
    await page.selectOption('[data-testid="input-level"]', "intermediate");
    await page.click('[data-testid="submit-button"]');
    await page.waitForURL(/\/talks\/.+/);

    // Go to admin, approve it
    await page.goto("./#/admin");
    await page.waitForSelector(`text=${title}`);
    // Click Approve on the row containing this talk's title
    const talkRow = page.locator('[data-testid^="admin-talk-"]', { hasText: title });
    await talkRow.locator('[data-testid^="approve-"]').click();
    await page.waitForTimeout(300);

    // Check that talk is no longer in admin queue
    await expect(page.locator(`text=${title}`)).not.toBeVisible();

    // Go to main list, filter by approved, check it's there
    await page.goto("./");
    await page.waitForSelector("article");
    await page.selectOption('[data-testid="filter-status"]', "approved");
    await page.waitForTimeout(300);
    await expect(page.locator(`text=${title}`)).toBeVisible();
  });
});
```

LOOK CLOSER

The third test alone touches submission, the admin queue, approval, navigation, and filtering — five different pieces of the system chained into one assertion. When it fails, the failure tells you almost nothing about which piece broke.

08-talk-detail.spec.ts

Reads the title and speaker off the list, then checks they match on the detail page. A second test checks the back link.

e2e/08-talk-detail.spec.ts — PLAYWRIGHT · TYPESCRIPT

```ts
import { test, expect } from "@playwright/test";

test.describe("Talk detail page", () => {
  test("shows correct title, speaker, abstract, and vote count", async ({ page }) => {
    await page.goto("./");
    await page.waitForSelector("article");

    const firstCard = page.locator("article").first();
    const titleText = await firstCard.locator("h2").textContent();
    const speakerText = await firstCard.locator("p").first().textContent();
    await firstCard.locator("a[href*='/talks/']").last().click();
    await page.waitForURL(/\/talks\/.+/);

    await expect(page.locator("h1")).toContainText(titleText!.trim());
    await expect(page.locator("p").first()).toContainText(speakerText!.trim().replace("by ", ""));
    await expect(page.locator('[data-testid="vote-count"]')).toBeVisible();
  });

  test("back link returns to list", async ({ page }) => {
    await page.goto("./");
    await page.waitForSelector("article");
    await page.locator("article").first().locator("a[href*='/talks/']").last().click();
    await page.waitForURL(/\/talks\/.+/);

    await page.click("text=← Back to list");
    await expect(page).toHaveURL(/\/$/);
  });
});
```

> **LOOK CLOSER**
>
> Same pattern as the list test: read values from one screen, assert they reappear on another. It's checking that the list and the detail page agree with each other, as much as it's checking either one works on its own.

What this baseline actually gives you

- **System-level coverage, twenty times over.** Every one of these 20 tests drives a real browser against a real frontend build, which talks to a real backend, which talks to a real (or seeded) database. Nothing here is mocked.
- **Full stack must be running for all twenty.** To execute even the narrowest check — one sort-order assertion, one validation check — the entire application has to be built and deployed first. There's no cheaper tier to fall back to.
- **Real duplication, real coupling.** Several tests re-run a full submit or navigate flow just to reach the state they want — so when one breaks, the failure often points at the setup, not the actual bug.
- **Confidence, at a cost.** These tests genuinely prove the system works end to end. The cost shows up in runtime, flakiness, and how far the feedback travels before a break surfaces — the subject of the next section.

SECTION 2

The Migration: From E2E to Contracts

Section 1 left us with eight files and twenty tests, all leaning on the full, deployed app. Turning that into a contract-tested suite took five real pull requests — plus a sixth, quieter step that's more of an ongoing habit than a single PR. Each one changes exactly one thing at a time.

WHAT THIS MIGRATION IS ACTUALLY ARGUING

This isn't an argument for zero E2E tests. It's an argument for not asking E2E to prove things a lower layer can prove faster and more precisely — and for keeping E2E for the cross-system behavior that genuinely needs it.

1. **Rewrite in Python**
   Port the same 20 tests over one-for-one. Nothing clever here — just proving the suite survives the language change before anything harder happens.
2. **Isolate the frontend**
   Move the frontend's tests off the real backend and onto a mock, driven by one shared contract. The original E2E suite keeps running right alongside it — nothing gets deleted yet.
3. **Prove the backend contract**
   Write tests that hit the real, running backend directly, to check it actually keeps its side of that same contract.
4. **Turn frontend requests into contracts**
   Check the other half of the story — not just what the frontend shows you, but what it actually sends over the wire.
5. **Move feedback into CI**
   Get all five pipelines running in CI, each one spinning up only the infrastructure it actually needs.
6. ✓ **Remove the E2E safety net**
   Retire the old coverage once its replacement has proven itself, one test at a time.

WHY FIVE STEPS, NOT ONE REWRITE

Bundling a language change, a testing-paradigm change, and a new set of infrastructure dependencies into a single PR means that if something breaks, there are three candidate causes to untangle. Each step below changes exactly one of those things at a time — which is also why Step 1 is deliberately the least interesting step in this guide.

HOW MUCH TO CONVERT AT ONCE

talkvote moves its whole suite through each step at once — reasonable with only twenty tests. On a larger suite, start smaller: one test or one file, carried through every step first.

1 Rewrite in Python

SAME SUITE, SAME COVERAGE, NEW LANGUAGE

For this migration, browser-tool choice wasn't the architectural problem — Playwright versus Selenium is a solved question either way. The real complexity shows up once tests need to reach past the frontend: reading a database directly, draining a queue, processing a Kafka event, mocking Elasticsearch. Porting to Python first — while the suite is still just browser automation, still low-risk, still a 1:1 match with the TypeScript original — decouples "what language is this suite in" from "what does this suite eventually need to reach into."

One idiom, translated

TypeScript's `Promise.all` race between a network wait and the action that triggers it becomes Python's `with` block — same synchronization idea, different language's native shape for it.

02-filtering.spec.ts — TYPESCRIPT

```ts
await Promise.all([
  page.waitForResponse(r =>
    r.url().includes("/api/talks") && r.status() === 200),
  page.selectOption('[data-testid="filter-track"]', "qa"),
]);
```

test_02_filtering.py — PYTHON

```python
with page.expect_response(
    lambda r: "/api/talks" in r.url and r.status == 200
):
    page.select_option('[data-testid="filter-track"]', "qa")
```

LOOK CLOSER

This step is supposed to be boring. Regex literals become `re.compile()`, template literals become f-strings, `describe` blocks collapse into flat functions grouped by file — all mechanical. The goal is proving the suite's shape survives the language change intact: same 8 files, same 20 tests, same coverage. Nothing here should be clever.

2 Isolate the Frontend

REPLACE BE CALLS WITH MOCKS · PR #2 ↗

talkvote's frontend tests move off the real Node/Express backend onto a mock instead — every mocked response built from one shared contract file, `api.py`'s `GetTalks`, `Talk`, and friends, not hand-written JSON. The original Playwright E2E suite from Section 1 is left running exactly as it was — nothing is deleted yet. The two suites coexist on purpose, so the new one can prove itself before the old one goes anywhere.

How this actually happens

In practice: take the existing Playwright suite and swap the real backend for a mock. Build the frontend normally, but point it at a mock server instead of the real API — MockServer here, though Pretender or any HTTP-level mock works the same way — then run each test and watch what comes through. Whatever request the mock receives is the request that test needs answered to keep passing.

That's naturally a one-test-at-a-time process on a large suite, but with only twenty tests here, it was faster to run it as one bulk pass across the whole thing rather than file by file.

The data behind each mocked response doesn't need much thought at this stage: hardcode it to start, and only reach for randomized values once a test proves it doesn't care about the specifics. The clicks, selectors, and navigation are worth reusing directly from the original suite — copying them is a fine fallback if reuse turns out awkward.

> **ONE RULE MATTERS MORE THAN THE REST**
>
> The original E2E suite stays exactly where it is until the new one has proven itself. This step isn't a replacement yet — it's a parallel path, run side by side until there's real confidence to retire the old one.

What one converted test looks like

The result, for a single test from `02-filtering.spec.ts` — same behavior, same assertions, but the backend is gone:

e2e/02-filtering.py (Step 2) — PYTHON · SURETY

```python
def test_filters_by_track_and_shows_only_matching_talks(page: Page):
    GetTalks.reply(reset=True, times=2, body=GetTalksResponse().with_values([
        Talk() for _ in range(8)
    ]))
    page.goto("./")
    page.wait_for_selector("article")
    total_count = page.locator("article").count()
```

code continues on next page ▾

continued — e2e/02-filtering.py (Step 2)

```
    with page.expect_response(
        lambda r: "/api/talks" in r.url and r.status == 200
    ):
        GetTalks.reply(reset=True, body=GetTalksResponse().with_values([
            Talk().with_values({Talk.Track.name: 'qa'})
        ]), params={'track': 'qa'})
        page.select_option('[data-testid="filter-track"]', "qa")
    filtered_count = page.locator("article").count()
    assert filtered_count > 0
    assert filtered_count < total_count
    cards = page.locator("article")
    for i in range(cards.count()):
        expect(cards.nth(i)).to_contain_text("qa")
```

LOOK CLOSER

`params={'track': 'qa'}` isn't decoration. It exists to tell MockServer which registered reply should answer a given request when more than one reply could plausibly match the same endpoint at once. Where a test only ever has one active mock in play, there's nothing to disambiguate — so it's correctly left off. It's a routing detail, not a rule every mock has to follow.

One fake talk, two consistent mocks

Submitting a talk and then loading its detail page are two separate contract calls — `SubmitTalk` and `GetTalk` — but in the mocked test they're built from one shared object:

e2e/04-submit-talk.py (Step 2) — PYTHON · SURETY

```
talk = Talk().with_values({Talk.Title.name: "E2E Test Talk"})
SubmitTalk.reply(body=SubmitTalkResponse().with_values(talk.value))
GetTalk.reply(
    path_params=TalkPathParams().with_values({
        TalkPathParams.Id.name: talk.Id.value
    }),
    body=GetTalkResponse().with_values(talk.value),
    times=2
)
```

Both mocked responses come from the same `talk.value` — so the POST response and the follow-up GET are guaranteed to agree with each other structurally, not by discipline.

AT A GLANCE — THE CI JOB

The Node/Express backend never runs in this pipeline. MockServer in a Docker container, the real frontend build, and the Python test runner — that's the entire job. No database, no seed data, no backend build step.

3 Prove the Backend Contract

PROVIDER TESTS — NO BROWSER AT ALL · PR #3 ↗

Step 2 proved the frontend renders correctly when fed a reasonable-looking response. Step 3 proves that response is actually true — real HTTP calls against the live, running backend, verified against the exact same shared contract.

One shared contract, refined in place

Writing tests against the real backend surfaced a few things the frontend-mock side hadn't needed yet — and because it's the same `api.py` on both sides, the refinement flows back automatically:

- **Track** went from a plain `String` to a proper `Enum` (Frontend / Backend / QA / DevOps).
- **Status**'s third value changed from `Pending` to `Submitted` — a genuine correction, matching what every real submission actually returns.
- A new **ErrorResponse** shape and a test-only **Clear** endpoint were added, both needed only once tests started asserting on 404s, 409s, and a known-empty starting state.

> LOOK CLOSER
>
> This is the strong version of "one shared contract." Because api.py isn't duplicated between the frontend-mock side and the backend-provider side, a correction made while writing provider tests didn't need to be manually copied anywhere — the consumer side just inherits it.

The actual test: build the input, send it, check the output

Step 2's mock never proved the backend would really respond this way — it just assumed a shape and moved on. This is where that assumption gets tested for real:

test_submit_talk.py — PYTHON · SURETY

```python
def test_submit_talk():
    talk = Talk().with_values({
        Talk.Track.name: Track.QA,
        Talk.Level.name: Level.Beginner,
    })
    SubmitTalk(
        req_body=SubmitTalkRequest().with_values(talk.value)
    ).request().verify()
```

The same `Talk` object Step 2 used to build its mocked response now builds a real request body instead. `.request()` sends it to the live backend; `.verify()` checks what comes back against the contract, not against whatever the mock happened to be told to say.

Verifying what nobody can predict exactly

The single clearest illustration of what "verify against the contract" means, as opposed to naive JSON equality — from `calls.py`'s `SubmitTalk.verify()`, the method that call just made:

calls.py — SubmitTalk.verify() — PYTHON · SURETY

```python
def verify(self, talks=None):
    super().verify_response(
        error_code=201,
        resp_body=api.SubmitTalkResponse().with_values(
            self.req_body.value
        ).with_values({
            api.SubmitTalkResponse.CreatedAt.name: datetime.now(
                timezone.utc
            ).strftime(Pattern.DATETIME_DELIM_T_WITH_ZONE_PRECISED),
            api.SubmitTalkResponse.Status.name: api.Status.Submitted,
            api.SubmitTalkResponse.Votes.name: 0,
        }),
        rules={
            api.SubmitTalkResponse.Id.name: is_valid_uuid,
            api.SubmitTalkResponse.CreatedAt.name:
                precise_dates_equal_with_delta,
        }
    )
```

Expects **201**, expects the response to echo back whatever was submitted plus the server's own defaults — then supplies `rules` for the two fields nobody could possibly predict: `Id` only needs to be *a valid UUID*, not a specific one; `CreatedAt` only needs to be *close to now*, not an exact string match.

Proving a rejection, not just an acceptance

The same build-input-then-verify-output shape extends naturally to checking the backend correctly *refuses* something, not only that it accepts what it should:

test_vote_twice_forbidden.py — PYTHON · SURETY

```python
def test_vote_twice_forbidden():
    talk = Talk()
    SubmitTalk(
        req_body=SubmitTalkRequest().with_values(talk.value)
    ).request()
    path = TalkPathParams().with_values(
        {TalkPathParams.Id.name: talk.Id.value})
    voter = {'X-Voter-Id': [fake.uuid4()]}
    Vote(path_params=path, headers=voter).request().verify()
    Vote(path_params=path, headers=voter).request().verify(
        error_code=409)
```

Same voter id, sent twice. The first vote succeeds; the second gets a **409** — reproducing that as a real person would mean two sessions or a cleared cookie. Here it's three lines, no browser involved.

New coverage, not translated coverage: the original E2E suite never exercised this — it only ever cast one vote per run. This test checks the real contract, not what the browser showed.

4 Turn Frontend Requests Into Contracts

NOT JUST WHAT IT RENDERS — WHAT IT SENDS · PR #4 ↗

Step 3 proved what a valid request looks like — build the input, send it for real, verify what comes back. Step 4 turns that same question on the frontend: does the request it actually sends match that structure, or does the app just render the right screen because the mock was told what to say?

test_04_submit_talk.py (Step 4) — PYTHON · SURETY

```
def test_submit_request(page: Page, talk, submit):
    api.SubmitTalk.verify_called(
        expected=api.SubmitTalkRequest().with_values({
            api.SubmitTalkRequest.Title.name: talk.Title.value,
            api.SubmitTalkRequest.Abstract.name: talk.Abstract.value,
            api.SubmitTalkRequest.SpeakerName.name:
                talk.SpeakerName.value,
            api.SubmitTalkRequest.Track.name: talk.Track.value,
            api.SubmitTalkRequest.Level.name: talk.Level.value,
        })
    )
```

The expected shape here — `SubmitTalkRequest` — is the exact type Step 3 built and sent to the real backend. Step 3 proved the backend accepts and correctly handles that shape; this proves the frontend's real outgoing request actually matches it, not just that the app renders the right detail page once it's told the answer.

When there's no fixed value to check against

The same idea still has to hold up when the request contains something generated fresh each time. Voting sends a random client-side voter id — there's nothing fixed to compare, only a shape to check:

test_06_voting.py (Step 4) — PYTHON · SURETY

```
def test_vote_request(load_talk, vote):
    api.Vote.verify_called(
        path_params=api.TalkPathParams().with_values({
            api.TalkPathParams.Id.name: load_talk.Id.value
        }),
        headers={'X-Voter-Id': [fake.uuid4()]},
        header_rules={'X-Voter-Id': [is_valid_uuid]},
    )
```

`header_rules` does for outgoing requests exactly what Step 3's `rules` did for responses — the frontend generates a fresh random voter id before every vote, so there's no exact value to assert, only that whatever it sent is shaped like a valid UUID.

The complete contract, in two tests: Step 3 proves the *server* accepts a properly-shaped request. Step 4 proves the *client* actually sends one, not a request that merely produces the right screen.

5 Move Feedback Into CI

FIVE WORKFLOWS, FIVE DIFFERENT FOOTPRINTS · PR #5 ↗

For the first time, there's something concrete to compare instead of just asserting it. Five workflows now coexist, each spun up with only the infrastructure it personally needs:

WORKFLOW	TRIGGERS ON (PR)	SPINS UP	BROWSER
`e2e.yml`	tests/e2e/**	nothing — points at a deployed env	Yes
`e2e-python.yml`	tests/e2e/python/**	nothing — points at a deployed env	Yes
`contract-frontend.yml`	frontend_tests/**, api.py, src/frontend/**	MockServer + real FE build	Yes
`contract-backend.yml`	backend_tests/**, api.py, src/backend/**	the real backend, in-job	No
`deploy-frontend.yml`	src/frontend/** (push only)	builds & ships to GitHub Pages	No

> **WHY THE E2E PIPELINES CAN'T GATE A PULL REQUEST**
>
> This isn't a prioritization choice — it's a structural limit. `e2e.yml` and `e2e-python.yml` both need a real, deployed frontend and backend to point at, and deployment only happens on push to *main*, after merge. There's simply nothing running yet for a pull-request branch to test against. The contract pipelines don't have that problem: each one spins up everything it needs — a mock server, or the real backend directly — inside the job itself, so they can gate a PR that E2E structurally cannot.

Why each pipeline triggers the way it does

E2E only makes sense to run in two situations: when the E2E test files themselves change, or once a new version of the app is actually live — which is exactly what a push to *main* means. Triggering E2E on every backend or frontend source change would mean testing a version of the app that isn't deployed yet.

Contract tests don't have that constraint — each one spins up whatever it needs inside the job itself, so they can run on a branch, before anything is deployed at all. But they answer to something E2E doesn't: the contract itself. Any test standing in as a consumer of `api.py` needs to re-run whenever that shared file changes, to prove it still agrees with the new version — which is why `api.py` sits in both contract pipelines' trigger paths, not just one.

The chain, one link at a time

1. A change lands in `src/backend/**` → `contract-backend.yml` runs, because backend source sits in its trigger path.
2. If the backend now behaves differently than `api.py` describes, that test fails immediately, naming the exact drift.
3. Fixing it means updating `api.py` to match reality.
4. Updating `api.py` is itself a trigger — for both pipelines. `contract-frontend.yml` now runs too, checking whether the frontend still agrees with the newly-updated contract.

> **NOT HYPOTHETICAL**
>
> This is exactly the sequence both examples in Section 3 walk through: the backend changes, the backend contract test catches it first, the shared contract gets updated — and only then does the frontend contract test get its turn to catch what it needs to catch.

Remove the E2E Safety Net

COVERAGE MOVES DOWN A LAYER, ONE TEST AT A TIME · PR #6 ↗

This wasn't one decisive deletion so much as an ongoing habit: as contract coverage solidified at the layer that actually owns a piece of behavior, the redundant copy at a higher layer got removed.

- The admin suite's long "submit → approve → filter" test — Section 1's example of a test doing five things at once — is gone from the frontend layer entirely. That full round trip is now proven server-side, by a provider test that already covers it.
- The sorting test never got the Step 4 request-validation treatment other files did, for the same reason: "does sort-by-votes actually work" is now a backend-provider question, answered by `test_get_talks_sort_by_votes`.
- Most of the plain Python E2E port from Step 1 — the stepping stone this whole migration started with — has been removed test by test, as each one's coverage found a more precise home at the contract layer.

LOOK CLOSER

None of this required a single "we're done, delete it" moment. Each removal was a local decision — this specific check is now redundant, because this specific lower-layer test already covers it — repeated until there wasn't much left worth keeping at the old layer. That's a healthier pattern to aim for than a big-bang cutover: the old suite shrinks in step with the new one's coverage actually growing to replace it, never ahead of it.

WHERE THIS IS GOING

The next section puts both approaches side by side on one concrete example: a real change to talkvote that introduces a defect, and how — or whether — each testing approach actually catches it.

SECTION 3

Examples in Practice

Everything so far has been about tests passing. This section is about real, deliberately merged changes that break something — and what each testing approach actually does when that happens.

EXAMPLE 1

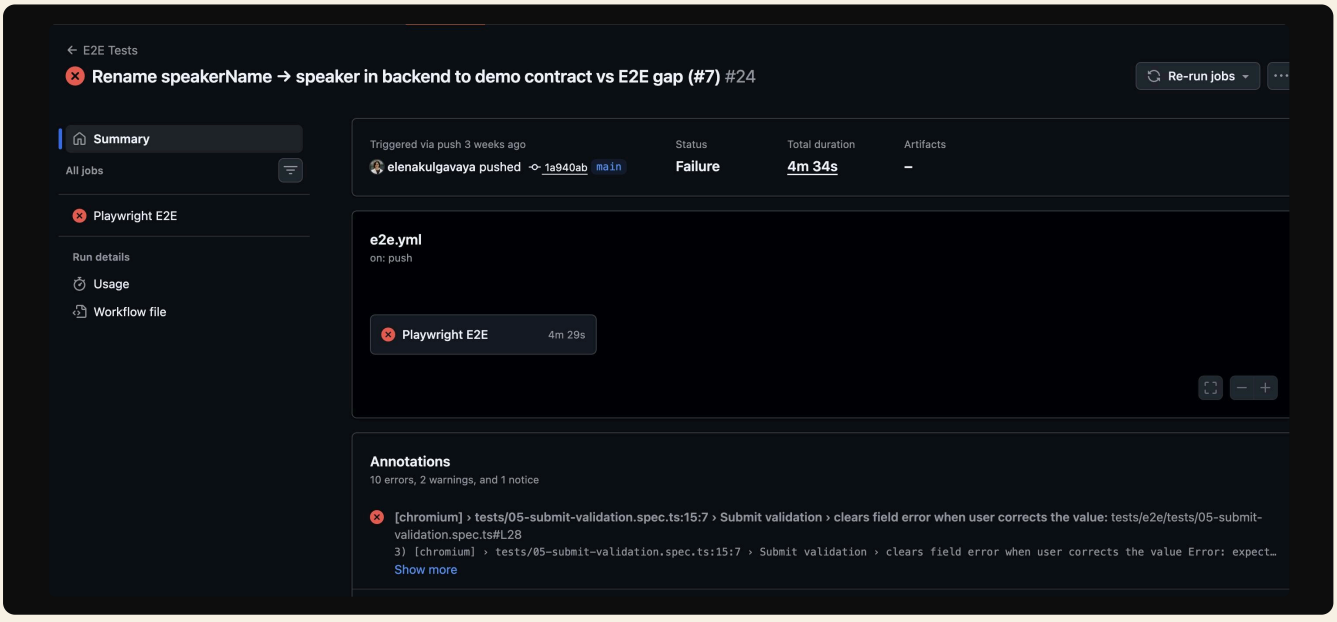
Renaming a Field

> THE CHANGE
>
> talkvote's backend renames one field — `speakerName` becomes `speaker` — via PR #7 ↗. The frontend and the shared contract are left untouched, on purpose. One small mismatch, seen twice: first the way the old E2E suite alone would have caught it, then the way contract tests do.

WITHOUT CONTRACT TESTS

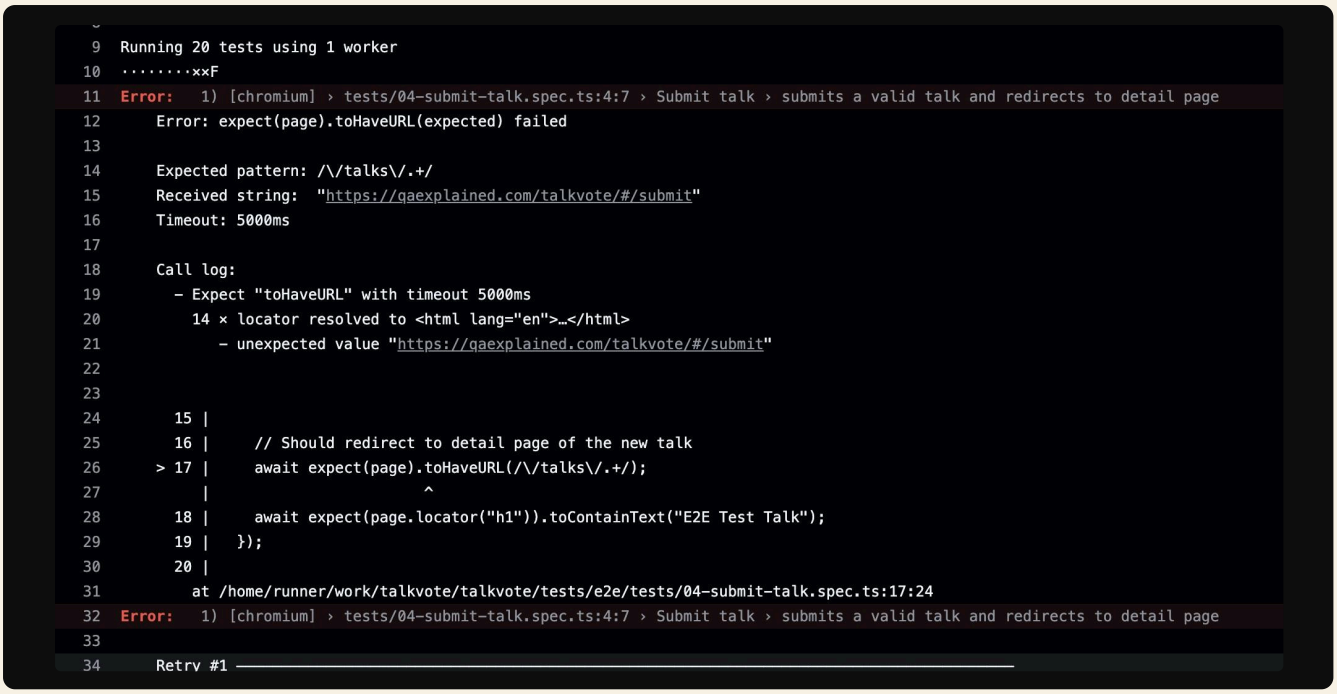
What E2E alone shows

First, without contract tests — just the original E2E suite on its own. Per Section 2, `e2e.yml` only runs on push to *main*, once the change is actually deployed. It runs there, and it fails:



The E2E workflow run triggered by the merge — the only point at which this pipeline could see the change at all.

But not cleanly. Four tests fail, across three unrelated-looking files, none of them mentioning a field name anywhere:



```
Running 20 tests using 1 worker
........xxF
Error:   1) [chromium] › tests/04-submit-talk.spec.ts:4:7 › Submit talk › submits a valid talk and redirects to detail page
    Error: expect(page).toHaveURL(expected) failed

    Expected pattern: /\/talks\/.+/
    Received string:  "https://qaexplained.com/talkvote/#/submit"
    Timeout: 5000ms

    Call log:
      - Expect "toHaveURL" with timeout 5000ms
        14 × locator resolved to <html lang="en">…</html>
           - unexpected value "https://qaexplained.com/talkvote/#/submit"


       15 |
       16 |     // Should redirect to detail page of the new talk
     > 17 |     await expect(page).toHaveURL(/\/talks\/.+/);
          |                        ^
       18 |     await expect(page.locator("h1")).toContainText("E2E Test Talk");
       19 |   });
       20 |
         at /home/runner/work/talkvote/talkvote/tests/e2e/tests/04-submit-talk.spec.ts:17:24
Error:   1) [chromium] › tests/04-submit-talk.spec.ts:4:7 › Submit talk › submits a valid talk and redirects to detail page

    Retry #1 ──────────────────────────────────────────────────────────────────
```

One of four failures — a generic URL-timeout error. Nothing here points at speakerName vs speaker.

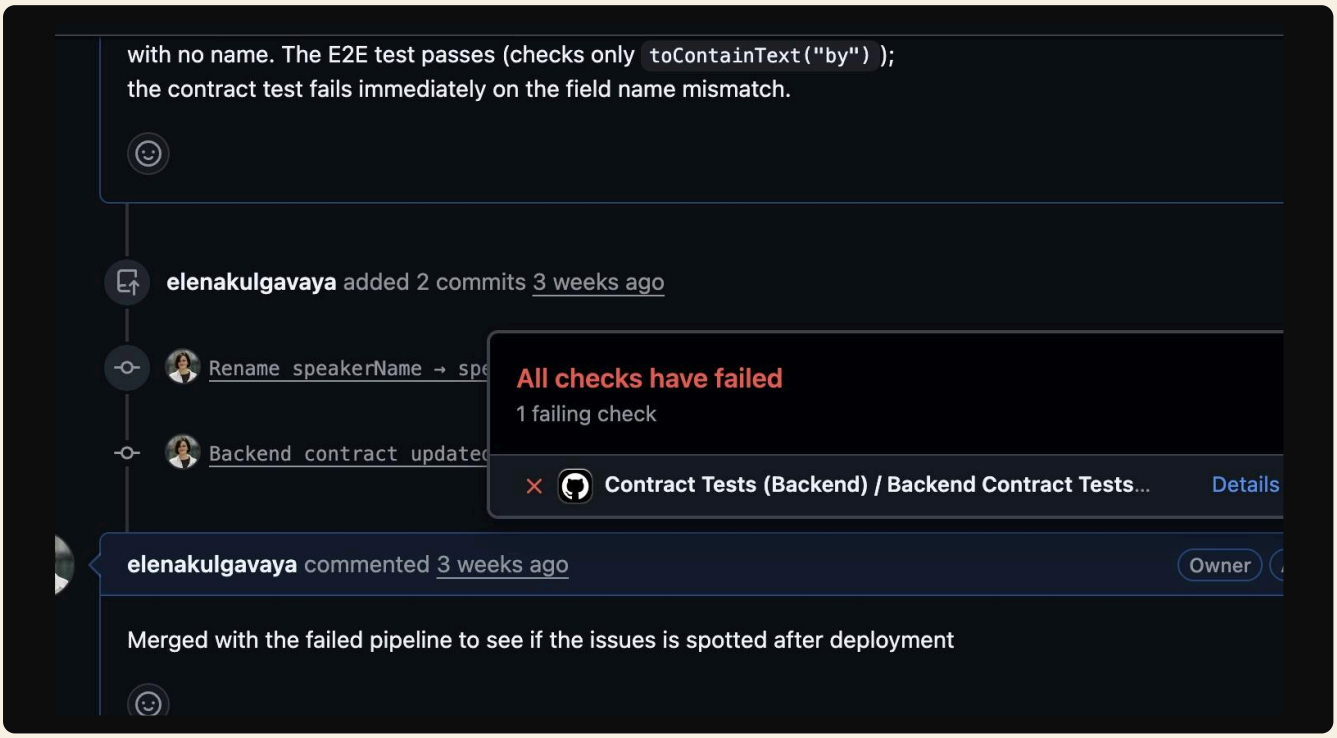
SAME ROOT CAUSE, FOUR DIFFERENT SYMPTOMS

Submitting a talk silently fails because the real backend now rejects the old field name — so every E2E test that submits a talk as a setup step fails too, including the admin test that isn't testing submission at all. Four failures, 4m 29s later, after the code is already live — and not one error message says why.

WITH CONTRACT TESTS

The backend catches it first

Run the same change through the pipeline setup from Section 2 instead. The rename touches only `src/backend/**` — the first link in that trigger chain — so `contract-backend.yml` runs immediately, before anything is deployed, before anyone merges:



GitHub PR check state right after the rename — only the backend contract check ran, and it failed.

The failure is a real HTTP round trip against the live backend, not a mock disagreeing with itself — the contract test still sends the old field name, and the real server now rejects it outright:

```
79  ================================= FAILURES =================================
80  ____________________________ test_submit_talk _____________________________
81  test_submit_talk.py:5: in test_submit_talk
82      SubmitTalk().request().verify()
83  ../calls.py:33: in verify
84      super().verify_response(
85  /opt/hostedtoolcache/Python/3.12.13/x64/lib/python3.12/site-packages/surety/api/caller.py:51: in verify_response
86      assert self.response.status_code == error_code, (
87             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
88  E   AssertionError: Unexpected code: 400
89  E   Response text: {"error":"Validation failed","details":{"speaker":["Required"]}}
90  =========================== short test summary info ============================
91  FAILED test_submit_talk.py::test_submit_talk - AssertionError: Unexpected code: 400
```

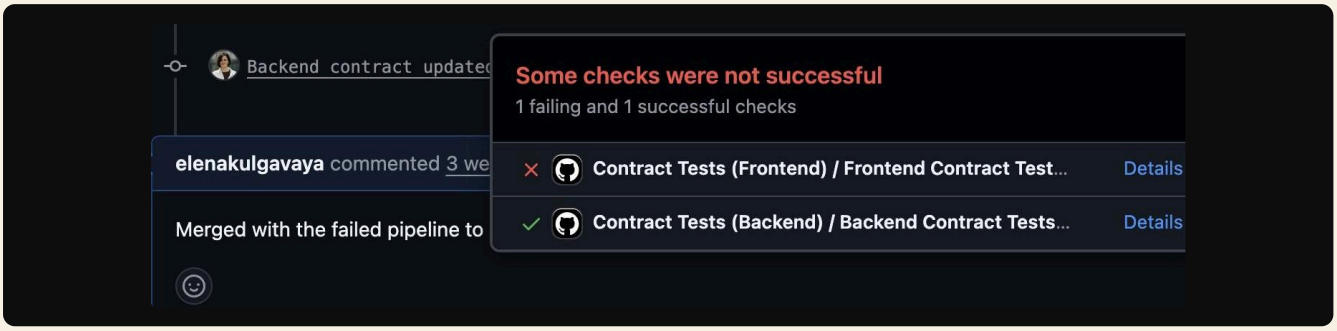
test_submit_talk.py — the backend genuinely returns 400, because the contract test still sends speakerName.

> **LOOK CLOSER**
>
> The frontend contract pipeline doesn't even run yet — exactly as Section 2's trigger chain describes. There's nothing for it to check until the shared contract itself changes.

WITH CONTRACT TESTS

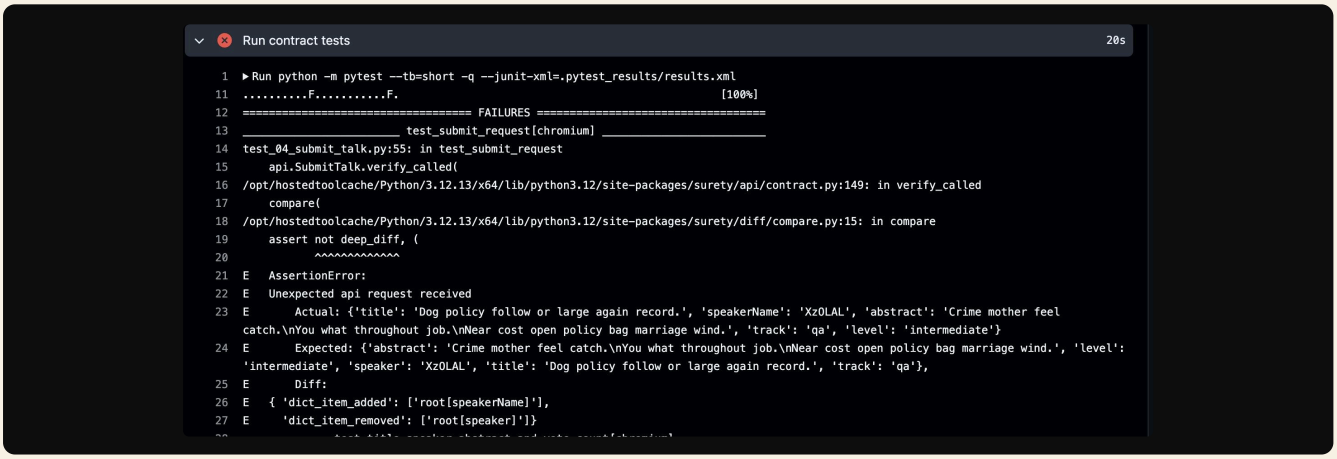
The contract updates — and the frontend catches what's left

Fixing the backend test means updating `api.py` to match reality. That update is itself a trigger — for both pipelines, per Section 2's chain. This time, the frontend contract check runs too:



After the contract update: backend contract passes, frontend contract now fails.

The backend side is green — the contract now matches reality. But the frontend was never touched, and its contract test catches that immediately, with a diff that names the exact field:



test_submit_request, Step 4's request-validation test — verify_called() catches the real outgoing request still using speakerName.

THE WHOLE POINT, IN ONE DIFF

One field added, one field removed, named explicitly: `speakerName` vs `speaker`. No reproduction steps, no guessing — the assertion failure points directly at the contract mismatch. This test only exists because of Step 4: without request-shape validation, nothing before deployment would have caught this at all.

THE COMPARISON, SIDE BY SIDE

Without contracts: fails four different ways, several minutes after merge and deploy, with a stack trace that never mentions the actual cause. With contracts: fails in under a second, on the exact commit that caused it, naming the exact field, before anything ships. Both are catching the same underlying bug. Only one of them tells you what it is.

EXAMPLE 2

Renaming an Enum Value

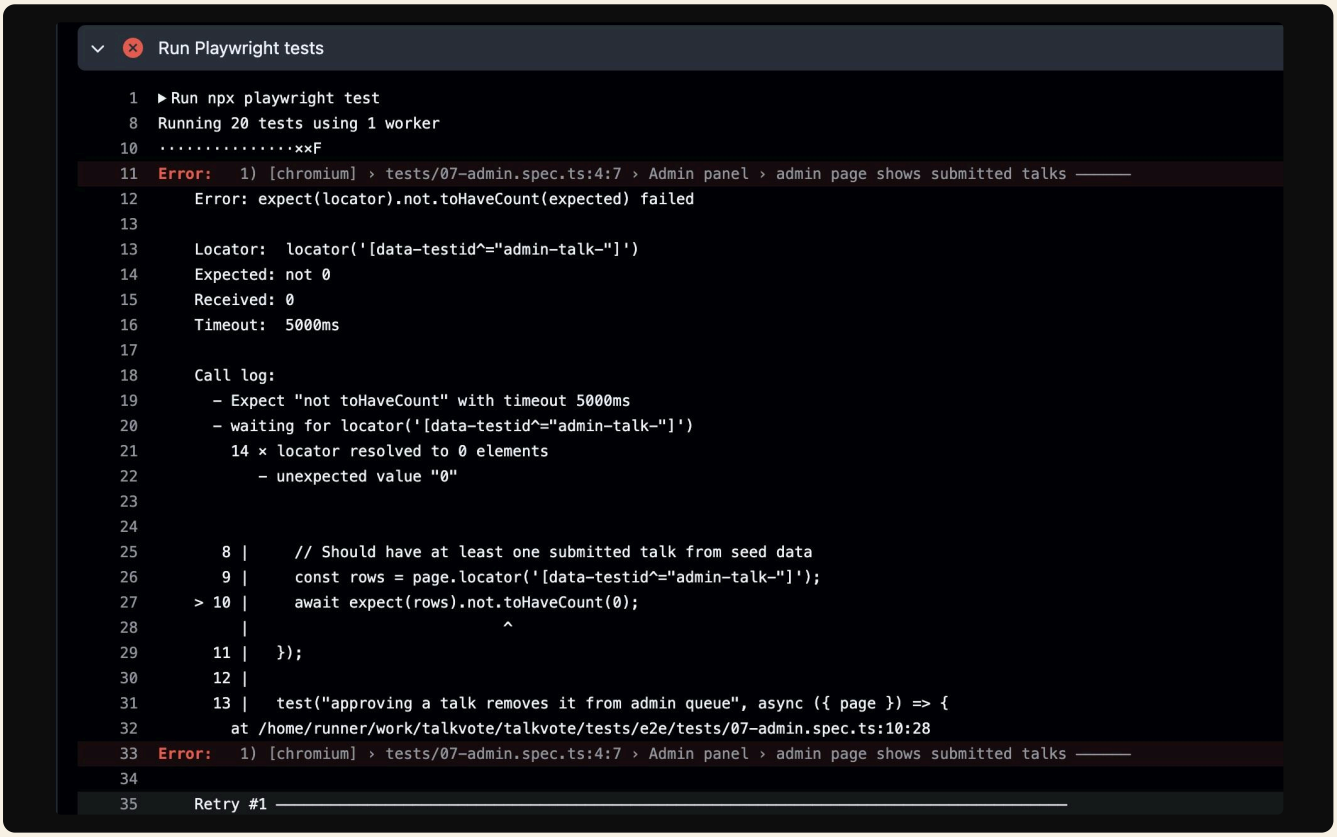
THE CHANGE

A different kind of change, same underlying question — PR #9 ↗ renames a talk's initial status from `submitted` to `pending`, backend-only, on purpose. Same setup as Example 1: watch it once without contract tests, then again with them.

WITHOUT CONTRACT TESTS

What E2E alone shows

Merged and deployed, watched through E2E alone. `e2e.yml` runs after the merge and fails — but quietly. The admin queue is simply empty:



Expected: not 0, Received: 0 — the admin page shows no submitted talks at all. Nothing here mentions a status rename.

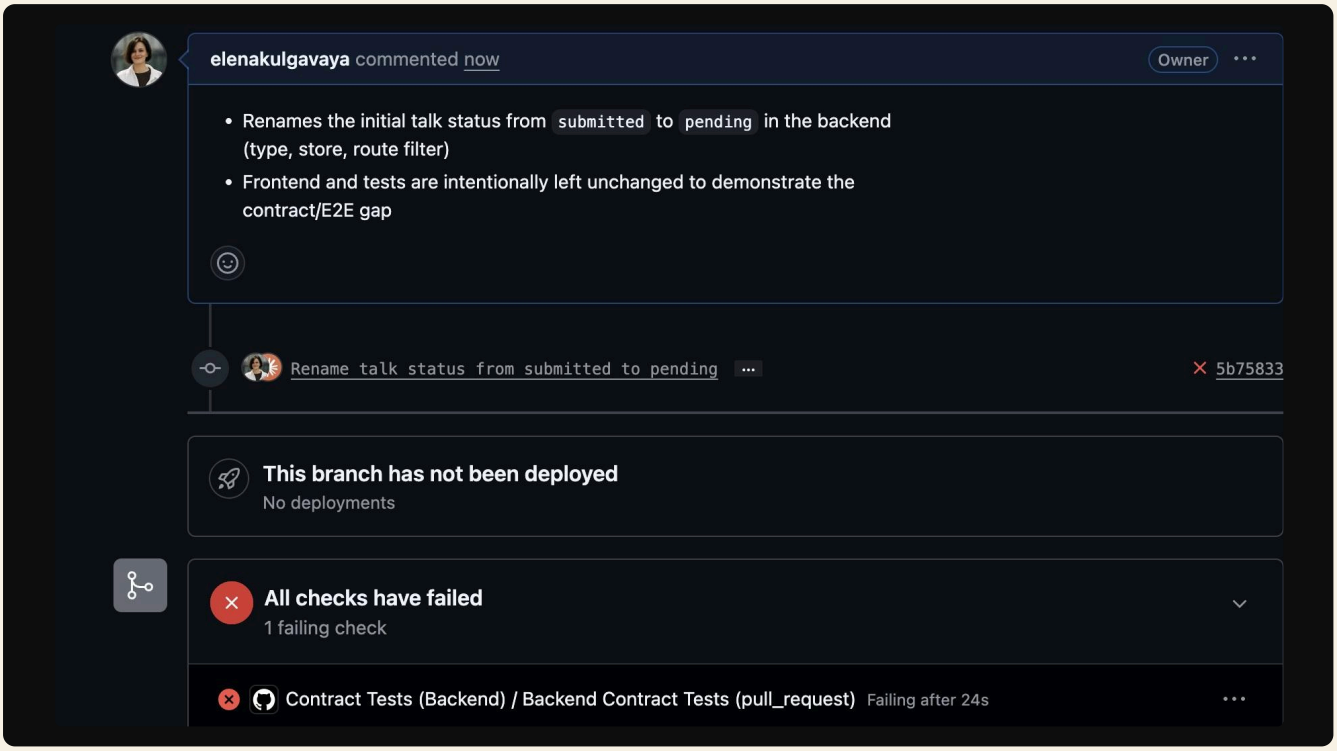
SAME PATTERN AS EXAMPLE 1

A generic assertion failure, no field or value named anywhere in the error. Whatever's filtering the admin queue by status no longer matches anything — but the test output gives no hint that a status value is the reason.

WITH CONTRACT TESTS

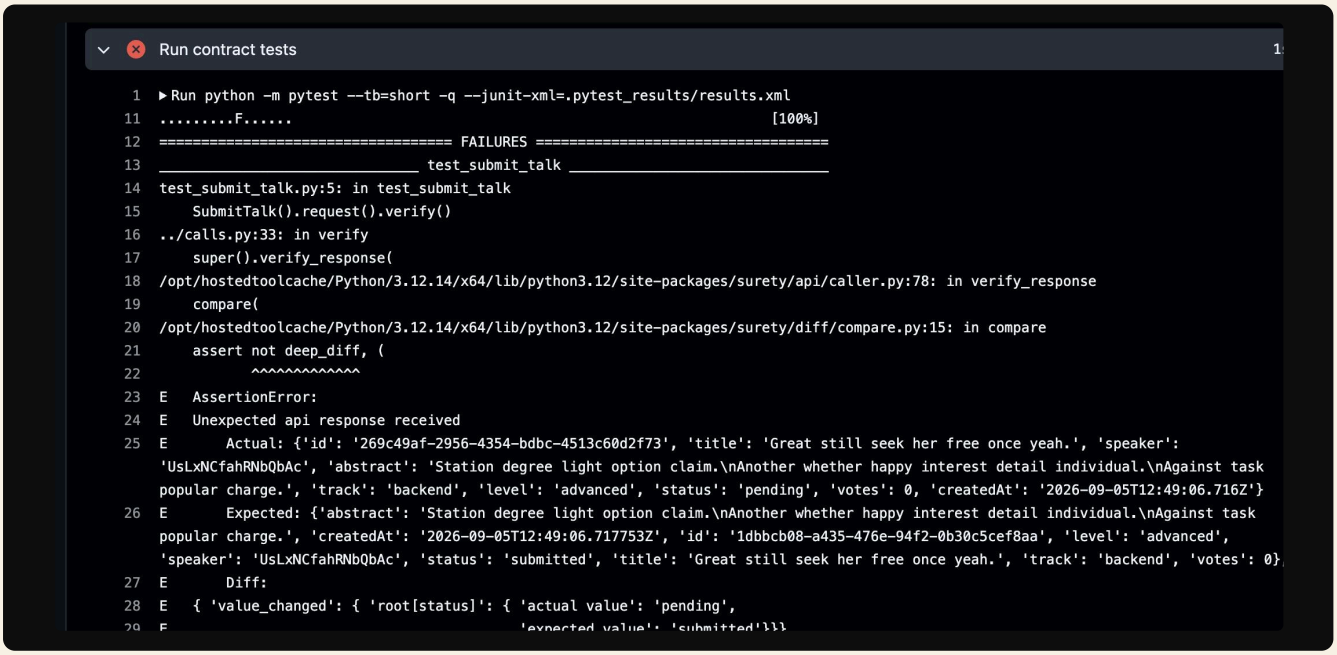
The backend catches it first

Same chain as Example 1: the rename touches the backend only, so `contract-backend.yml` is the only check that runs, and it fails immediately, before merge:



PR #9's own description — and the backend contract check fails immediately, before merge.

The backend contract test still expects the old value, and the real backend now returns the new one — caught in a single precise assertion:

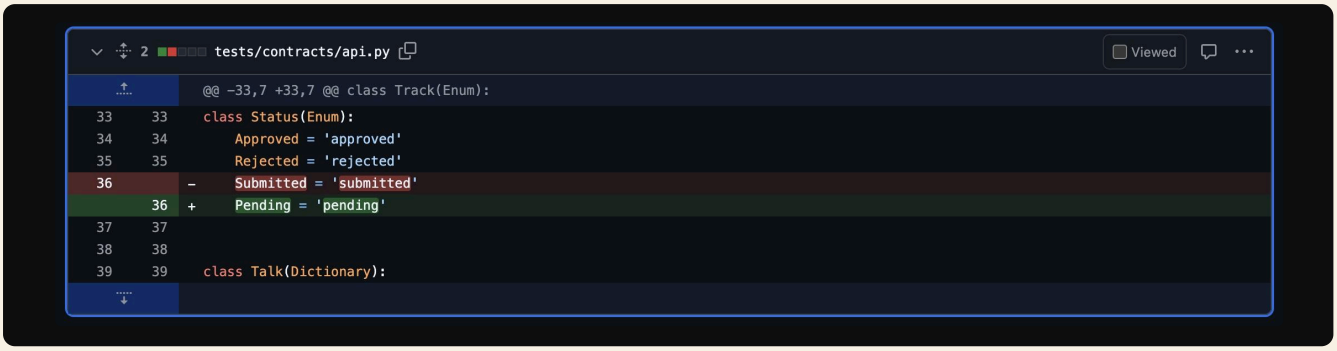


The backend contract test fails with an exact value_changed diff: actual 'pending' vs expected 'submitted'.

WITH CONTRACT TESTS

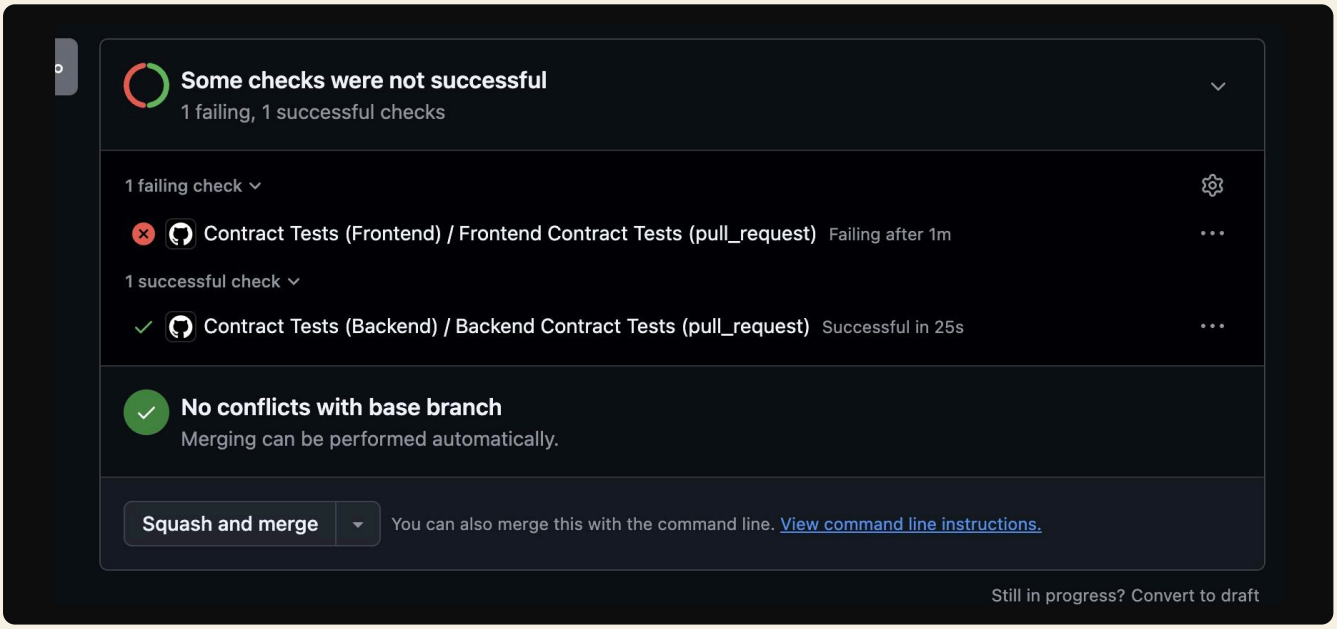
The contract updates — and this time, the frontend breaks louder

Updating the shared contract this time means renaming an enum member, not just a field:



api.py: Status.Submitted is renamed to Status.Pending — the symbol itself is gone.

Backend contract passes again. But the frontend contract check fails — and not with a quiet value mismatch this time:

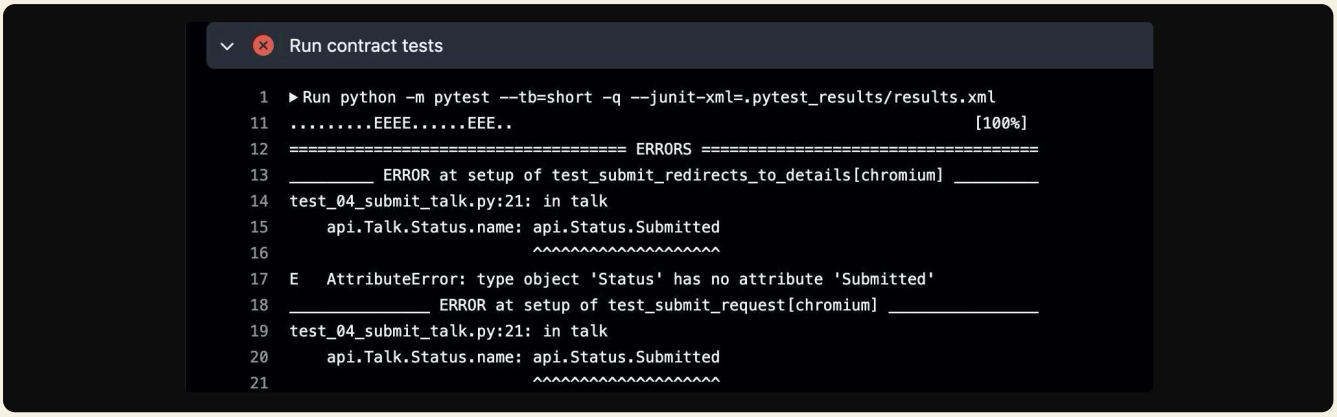


Same shape as before: backend green, frontend red.

> **A DIFFERENT, LOUDER KIND OF FAILURE**
>
> Renaming a field produces a value mismatch. Renaming an *enum member* removes a symbol entirely — and every frontend test file that still references `api.Status.Submitted` directly in its own fixture code breaks instantly, at setup, before a single assertion runs.

WITH CONTRACT TESTS

One renamed symbol, four setup errors



AttributeError: type object 'Status' has no attribute 'Submitted' — across test_04_submit_talk.py and test_07_admin.py alike.

Four test setups fail with the exact same message, in two different files, the instant the shared contract changes. There's no ambiguity about what broke or why — the traceback names the missing attribute directly.

> **THE COMPARISON, SIDE BY SIDE**
>
> Without contracts: the admin queue goes quietly empty, with an error that never says why. With contracts: four setup errors, in two files, all naming the exact missing symbol — before anything ships.

Closing the loop

Unlike Example 1, this one didn't stop at "merged with a known gap." PR #10 ↗, "Update frontend contract tests to use pending status," fixes both halves of what the failing contract tests pointed at, in two commits:

- **Update frontend contract tests to use pending status** — the test fixtures that referenced the old enum member get updated to the new one.
- **Update frontend to use pending status** — the real frontend application code gets updated too, not just the tests around it.

Both commits merged together, one check, passing.

> **THE POINT OF BOTH EXAMPLES, TOGETHER**
>
> Example 1 showed a break the contract catches immediately and E2E eventually notices, vaguely, after deploy. This one shows the same shape of break — plus what happens next: the contract test's error message wasn't just a red X, it was specific enough to drive the actual fix, in the same pull request that closed the gap.

AFTERWORD

Where Else This Applies

Everything in this guide sits at one boundary — the HTTP line between talkvote's frontend and its backend. That's not an accident of scope; it's the easiest boundary to see, which is exactly what makes it the easiest one to teach.

But the idea underneath it doesn't stop there. Anywhere something is produced and something else consumes it without seeing the producer's code — a queue message, a Kafka event, a database schema, a call to another team's service — the same question applies: does anyone actually verify both sides agree, or does everyone just trust it and find out in production? talkvote never got there in this guide. The five steps here are the pattern for reaching it — not the whole territory.

> **THE POINT, ONE MORE TIME**
>
> A contract test should prove something about the relationship between two systems, not just what one of them happens to render. Once that's the standard, the specific tools — Playwright, Surety, MockServer — stop mattering nearly as much as which question each test is actually answering.



FURTHER READING

Surety, the contract-testing framework used throughout this guide, is documented at surety.readthedocs.io ↗. For more on contract testing and QA practice generally, see qaexplained.com ↗.