Building the Platform That Would Judge Us

A Dogfood 2026 Write-Up & Engineering Post-Mortem

Repository	https://github.com/thevipulbangar/DogFood
Team Members	Vipul Bangar (Backend) & Vaishnavi (Frontend)
Tech Stack	Next.js 16 + React 19, Express, PostgreSQL, Docker Compose

The brief was to build an open-source, self-hostable platform for running a hackathon: managing teams, submissions, judging, and public voting. It had to run smoothly with a single docker compose up command on a laptop with the network completely turned off. The irony was obvious from minute one: we were building the very machine that would decide whether our own project was any good.

What We Shipped

- **T1 (Core):** Auth with four roles, events with tracks and prizes, teams with invite codes, draft-then-submit project submissions, and a public gallery.
- **T2 (Judging):** Weighted rubrics, round-robin judge assignment, per-judge z-score normalization, and CSV export.
- **T3 (Public):** Community voting configurable per event, comments, randomized ordering, results hidden until an organizer publishes them, an append-only audit log, and a threat model.
- **T4 (Stretch):** Only the API-first bonus. A full OpenAPI 3.0 spec covers every endpoint and is served live at /api/openapi.yaml. The rest of T4 was intentionally cut (detailed below).

The rest of this post-mortem is about the architectural decisions, mathematical traps, and edge cases that didn't go according to plan.

1. The Schema I Would Redo

The database schema features twelve tables and functions correctly in production, but three core decisions would be discarded on day one of a rewrite.

Roles Live in One Global Column

```
role TEXT NOT NULL DEFAULT 'participant'
  CHECK (role IN ('participant', 'judge', 'organizer', 'admin'))
```

This schema assumes a person is exactly one role, everywhere, forever. That premise is false for real hackathons: this weekend's judge is next month's participant. Because roles are global, the judge pool is global as well. Assignment queries execute SELECT id FROM users WHERE role = 'judge' with no event scope in sight. As detailed in Section 3, this column directly enabled our worst security bug.

> **Schema Redesign:** *What I'd do instead: Implement an event_roles (event_id, user_id, role) mapping table. Every permission question becomes 'What is this person in this event?', and potential conflicts of interest become a clean join instead of an afterthought.*

Editing the Rubric Deletes Scores

'Update the rubric' was implemented as a simple delete-all-then-insert operation. Meanwhile, scores.criterion_id is configured with ON DELETE CASCADE. Consequently, if an organizer fixes a simple typo in a criterion name halfway through judging, the database quietly deletes every submitted score for that criterion.

We caught this bug in time to add a prominent warning banner on the rubric editor. However, a warning banner is an apology, not a architectural fix.

> **Schema Redesign:** *What I'd do instead: Make rubrics strictly versioned and immutable once any score references them. Enforce ON DELETE RESTRICT so the database explicitly rejects destructive modifications rather than destroying judges' hard work.*

Tracks Are a Text Array

```
tracks TEXT[] NOT NULL DEFAULT '{}'   -- stored on events
track  TEXT                          -- stored on submissions
```

Nothing structurally links the event tracks array to the submission track string. Rename a track on an event, and every existing submission in that track points at a string that no longer exists in the event record. It was fast to build during the initial rush, but completely wrong to keep. It should be a dedicated tracks table with proper foreign key constraints.

Honourable Mention: The Test Suite and the Live Database

Our integration tests execute real HTTP requests against the exact same Postgres instance used by the application. For a while, every npm test run left leftover 'test teams' in the public gallery, leading to hours spent hunting down 'fake data' in the frontend that backend integration tests had generated.

We added an npm run reset script as a stopgap. The proper architectural fix is isolated test databases spun up per run, which sits at the top of our technical debt roadmap.

2. The Normalization Maths That Fought Us (and Mostly Lost)

Judges are not calibrated instruments. One judge's 5/10 represents 'solid, well-executed work,' while another's 5/10 means 'please start over.' If you simply average raw scores, a project assigned to harsh judges is penalized regardless of its underlying quality.

The Idea: Z-Scores Per Judge

For each judge, we calculate their weighted totals across all submissions they reviewed in an event. We compute their mean (μ) and sample standard deviation (σ), then evaluate each project relative to that specific judge's scoring habits:

```
z = (weighted_total - μ_judge) / σ_judge
```

Consider a worked example: Judge A is harsh and gives scores of 4, 5, and 6 (μ = 5, σ = 1). Judge B is generous and gives scores of 7, 8, and 9 (μ = 8, σ = 1).

Project	Scored By	Raw Score	Normalized Z-Score
Project P	Judge A (Harsh)	6.0	**+1.0 (A's top score)**
Project Q	Judge B (Generous)	7.0	-1.0 (B's lowest score)

Raw averages rank Project Q above Project P (7.0 vs 6.0). Normalization correctly flips the ranking: Project P was the absolute best submission Judge A evaluated, while Project Q was the worst submission Judge B evaluated.

Round One: The Fallback That Erased Judges

Calculating a standard z-score requires σ > 0. A judge who has only completed one review has no defined standard deviation. Similarly, a judge who awards a 7 to every project produces σ = 0. Our initial implementation attempted to handle this edge case in a single line:

```
const normalized = stats.normalizable ? (raw - stats.mean) / stats.stddev : 0;
```

A z-score of 0 represents an 'exact average' score. Consequently, a late-joining judge's very first review—regardless of how glowing or critical—landed as perfectly average. The code comment promised a fallback to the raw score, but the implementation did something completely different. Furthermore, the API never exposed the normalizable flag, masking the behavior from organizers.

Round Two: Fixed, but with a Flaw We Shipped

We updated the fallback to return the raw weighted total and attached a fully_normalized flag to every output object so organizers could identify sparse data points:

```
const normalized = stats.normalizable ? (raw - stats.mean) / stats.stddev : raw;

// ...
fully_normalized: judgeScores.every((j) => j.normalizable),
```

Here is the honest post-mortem: this logic mixes completely different mathematical units. Standard z-scores fall roughly between -2.0 and +2.0, whereas raw weighted totals span from 0.0 to 10.0.

Suppose late Judge C reviews Project R at 7.2 as their single assignment. Project R's 'normalized' average becomes 7.2, easily outscoring Project P's top-tier z-score of +1.0. While the first bug made sparse-data projects invisible, the fix made them virtually unbeatable. The flag made the issue visible in the UI, but the mathematical ranking remained flawed.

What Actually Works: Mapping Z-Scores Back to Points

To equalize units, compute the overall pooled mean (μ_pool) and sample standard deviation (σ_pool) of every score cast across the entire event, then rescale each individual z-score back into point space:

```
rescaled_score = μ_pool + (z × σ_pool)
```

Evaluating across all scores with μ_pool ≈ 6.60 and σ_pool ≈ 1.73 yields clear, standardized point ratings:

Project	Old Shipped Value	Rescaled Point Score
Project P (Judge A's Top Pick)	+1.00 (z-score)	**8.33 / 10**
Project R (Judge C's Only Review)	7.20 (raw fallback)	7.20 / 10
Project Q (Judge B's Bottom Pick)	-1.00 (z-score)	4.87 / 10

This rescaled model restores intuitive ranking: Project P leading, followed by R, then Q. The ultimate refinement is Bayesian shrinkage, which pulls sparse judge scores toward the event pool mean based on sample size. The core lesson: implementing the primary formula took ten minutes; understanding and properly scaling the mathematical edge cases took hours.

3. The Role-Isolation Bug

The project prompt anticipated a major security flaw discovered around hour 60. Ours surfaced even later, unfolding in two distinct stages.

Stage One: A Role with No Door

Late in the build phase, we asked a fundamental question: How does a user actually become a judge? The answer was that they couldn't. The registration endpoint hardcoded role = 'participant', and no API route existed to promote users. The entire T2 judging suite was accessible only via seeded demo accounts. Furthermore, the /dashboard route directed judges to a dummy mock component displaying static assignment data, despite functional judging pages existing one click away.

The Fix: Strict admin control enforced at the database level rather than application memory:

```
CREATE UNIQUE INDEX one_admin_only ON users ((role = 'admin')) WHERE role = 'admin';
```

A partial unique index on a constant expression guarantees that exactly one row can possess role = 'admin'. That primary admin creates judge and organizer accounts. The database outright rejects any attempt to insert a second admin.

Stage Two: The Judge on the Team

This was the critical vulnerability. Consider these three application behaviors side by side:

- **1. Unrestricted Access:** POST /api/teams and POST /api/teams/join only required any authenticated session, performing zero role checks.
- **2. Naive Assignment:** The round-robin judge assignment algorithm queried WHERE role = 'judge', completely ignoring team_members relationships.
- **3. Ownership Check Flaw:** Scoring endpoints verified assignment ownership correctly (assignment.judge_id === req.user.id), which succeeded if the judge was assigned to their own team.

Consequently, a judge could join a participant team via invite code, submit a project, assign themselves as the reviewer, and score their own submission. While every individual component performed its designated check, an exploitable hole existed between them.

> **Vulnerability Analysis:** *The Irony: We had rigorously protected public community voting against this exact flaw—POST /api/voting explicitly blocks votes for one's own team. We secured the single community vote while leaving the primary prize-awarding judging pipeline exposed.*

The immediate software fix isolates assignments:

```
-- Prevent routing submissions to judges who belong to the team
```

```
AND NOT EXISTS (
  SELECT 1 FROM team_members tm
  JOIN submissions s ON s.team_id = tm.team_id
  WHERE s.id = $submission_id AND tm.user_id = $judge_id
)
```

Combined with requiring requireRole('participant') on team endpoints and adopting event_roles, the database structural layer renders 'judge in event X' and 'participant in event X' mutually exclusive.

4. The Feature We Cut and Don't Regret

We cut the Analytics Dashboard. In early mockups, it looked impressive: detailed metric cards, real-time charts, activity feeds, and a system status panel showing Database: SQLite · 18.4 MB, API p95: 11 ms, and Webhooks: 3 endpoints · 100%.

None of it reflected real system state. Our actual backend runs PostgreSQL, not SQLite. We had not integrated p95 latency tracking, and no webhook infrastructure existed. The panel was a relic from early frontend design that appeared so complete that no one questioned its accuracy.

During the final preparation push, we enforced a strict rule: Every reachable UI screen must display authentic backend data or explicitly state that it is not implemented. This rule was established by the human engineer after our AI coding assistant repeatedly flagged features as 'done' simply because an unintegrated API route existed. We systematically rewired landing metrics, countdown timers, and gallery views, while converting /analytics and /certificates into honest 'Not Implemented' state screens.

We deliberately cut remaining T4 stretch features (PDF certificates, cryptographic judge signatures, webhooks, bulk import) for the same reason: shipping one polished, fully reliable core pipeline is far better than presenting four incomplete or deceptive components.

5. The Moment the Spec Was Harder Than It Looked

The specification contained one particularly demanding phrase: 'configurable community voting.'

Our initial T3 implementation included basic voting: one vote per project enforced via UNIQUE(submission_id, voter_id), rate limiting, self-voting prevention, and audit logging. We initially marked it as complete.

Upon re-reading the requirement, we realized key capabilities were missing: organizers could not disable voting during active judging, no voting window controls existed, and rate limits were hardcoded at 10/min. We had built 'voting exists,' not 'voting is configurable.'

We added voting_open and vote_rate_limit columns to the event model. Organizers gain granular toggle and threshold controls in the event configuration UI, backend routes enforce

403 (Closed) or 429 (Rate Limited) status, and every state change is committed to the immutable audit log.

This realization reshaped how we parsed the entire project specification. The core complexity resided in the qualifiers:

- • Configurable community voting with per-event operational toggles.
- • Results strictly hidden until explicitly published by an organizer.
- • An append-only audit trail with zero delete/update paths, even for administrators.
- • Strict offline execution capable of running entirely without external internet dependencies.

What I'd Tell Myself at Hour Zero

- **1. Event-Scoped RBAC:** Model user roles per event in the initial migration. Global role flags create severe privilege escalations and scoping bugs.
- **2. Mathematical Sanity Checks:** Verify mathematical edge cases produce consistent units before combining them in unified scoring tables.
- **3. True Definition of Done:** Define 'Done' strictly as reachable UI flows powered by real backend data.
- **4. Isolated Test Environments:** Establish dedicated, isolated test databases from the very first integration test.

The platform was built to judge our hackathon submission. In the process of building it, we ended up judging our engineering decisions even harder.