PROTOVATE / AI FIELD NOTES

Your AI Project Worked in the Demo. So Why Didn't It Launch?

Most AI trials work. Most never reach a real user. These are the five gaps that stall them - and the checks that close each one.

by Prateek Sharma, AI Engineer

A successful AI demo can still stall before production when testing, cost, data, failure handling, and ownership remain unresolved.

Contents

A team spends six weeks building something with AI. It demos beautifully. Everyone in the room agrees it's impressive. Then it sits.

A **pilot** is a small, time-boxed trial: you build a limited version of the idea, try it with a handful of people, and decide whether to fund the real thing. It's the step between "we should try AI for this" and "our customers use this every day."

Six months later the same team is running a new pilot, on a newer model, with the same architecture and the same blind spots. The build was never the problem. Nobody had defined what "working" meant, what it would cost at real volume, who owned it after launch, or what it should do when the AI gets something wrong in front of a paying customer.

These notes come out of production AI work at Protovate — a facility-management voice assistant used by field technicians, an internal analytics bot, consumer apps shipping in three languages, and a run of model training and data-labeling projects. Client names are left out throughout. The numbers and the failure modes are not.

A demo proves the AI can do it *once*. Launching asks whether you can do it ten thousand times, cheaply, and know when it breaks.

Below are the five gaps that stall these projects, what each one looks like from the inside, and the specific check that closes it. If you have a trial running right now, the scorecard further down will tell you fairly quickly which of the five you're standing in.

The five gaps that stall AI projects

They show up in roughly this order, and each one is cheaper to close before the build than after it.

Gap explorer

Pick the one that sounds most like your project.

- ✓ No definition of good
- ✓ Cost found too late
- ✓ Data was never ready
- ✓ Built for the happy path
- ✓ Nobody owns the rest

Nobody agreed what "good" means

Symptom: someone changes a prompt, everyone tries it a few times, and the team argues about whether it got better.

This is the most common reason a project stalls, and it stays invisible until you try to improve something. Without a fixed set of test cases and expected answers, every change is a matter of opinion. The team can't prove progress to whoever is funding it, can't safely switch to a different model, and can't tell whether last week's fix quietly broke something else.

It gets worse with scale. A setup that handles the twelve examples someone remembers will fail on the two hundred it has never seen. On a consumer app shipping in three languages, we found a change that clearly improved English answers made one of the other two languages worse — something no amount of manual spot-checking would have caught, because nobody was spot-checking in that language.

THE CHECK THAT CLOSES IT

- ✓ Write 50 to 200 real inputs with expected answers **before** you build. Pull them from support tickets, logs or user interviews, not from imagination.
- ✓ Include the ugly ones: empty input, wrong language, off-topic questions, and the cases where the right answer is "I don't know."
- ✓ Score automatically where you can, by hand where you must, and record the number every time. A pass rate that moves is a project. A pass rate that doesn't exist is a demo.

The cost model arrived after the architecture

Symptom: the trial cost almost nothing to run, and the first realistic volume estimate makes the finance conversation awkward.

Trials run on tiny volumes with the largest available model, because that's the fastest way to make something impressive. Then someone multiplies by the real user count and the economics collapse. The feature isn't wrong — the design assumed a cost per request nobody checked against the price of the product.

Two multipliers catch teams out. The first is conversation history: if you resend the whole conversation on every turn, cost grows with the square of its length, not in a straight line. The second is non-English text, which can consume noticeably more of the model's billing units for the same content, so a feature that pencils out in English can be materially more expensive in another language.

THE CHECK THAT CLOSES IT

- ✓ Model the cost per user per month **before** committing to an architecture. Usage in, usage out, calls per session, sessions per user.
- ✓ Test whether a smaller, cheaper model clears your quality bar. It often does, and the gap between model tiers is frequently an order of magnitude in price.
- ✓ Log usage per request, tagged by feature, from day one. Retrofitting this is painful, and without it you can't tell which feature is expensive.
- ✓ Set a hard ceiling per user and decide in advance what happens at the ceiling: degrade, queue, or charge.

The data was never actually ready

Symptom: "we have years of data" turns into three weeks of cleaning, and the labels disagree with each other.

Almost every organisation has more data than usable data. It's spread across systems, inconsistently formatted, full of internal shorthand, and unlabeled for the task you now want. Search-based approaches don't rescue you either: searching a messy archive returns messy material, and the model dutifully summarises the mess.

Where labeling is involved, expect the first pass to be wrong. On a labeling project we ran, two capable annotators working from the same written guidelines disagreed on a meaningful share of borderline cases — not through carelessness, but because the guidelines hadn't anticipated those cases. That disagreement rate isn't a failure. It's the measurement telling you to sharpen the definition before scaling up.

THE CHECK THAT CLOSES IT

- ✓ Have two people label the same 100 items independently and measure how often they agree. Fix the guidelines before labeling ten thousand.
- ✓ Audit a random sample of your source material by hand. Duplicates, dead documents and superseded policy are the usual finds.
- ✓ Write down your in-house vocabulary explicitly — abbreviations, site codes, equipment names. This is what general-purpose AI gets wrong most visibly in front of expert users.

Everything was built for the path in the demo

Symptom: it works beautifully in the meeting room and falls apart in a plant room with bad signal.

Demo conditions are quiet, connected and cooperative. Real conditions are none of those. Building a voice assistant for field technicians taught us that the interesting engineering isn't the conversation — it's everything around it. Background machinery noise. A caller who interrupts mid-sentence. A dropped call halfway through. Someone answering a question the assistant hasn't asked yet. A technician using a site abbreviation the system has never seen.

The same holds for speed. A response that takes several seconds is fine in a demo, where everyone is watching politely, and unacceptable when a user is standing in a corridor. Performance problems also hide until volume arrives: one feature of ours ran fine for months and then slowed badly, and the cause turned out to be a database query that only became expensive once the table passed roughly seventeen thousand rows.

THE CHECK THAT CLOSES IT

- ✓ Write down the failure modes before you build: no network, slow network, service unavailable, AI returns nonsense, user goes off-script.
- ✓ Decide the fallback for each one. A useful, honest failure beats a confident wrong answer every time.
- ✓ Set a response-time budget and test against it with realistic inputs, not the short ones from the demo.

- ✓ Test with the messiest real inputs you can find, in every language you ship in.

Nobody owned the part after the AI works

Symptom: the trial is finished, and there's no answer to "who gets called when it breaks at 2am?"

Getting an AI to produce good output is a fraction of the work. The rest is running it, versioning it, monitoring it, tracking its spend, handling provider outages, and noticing when quality drifts. Teams that treat that as a deployment afterthought discover it is most of the schedule.

There's a governance version of the same gap. Where does the data go, how long is it kept, who can see the logs, and what happens if a regulator asks. A trial that reaches a security review without answers doesn't get rejected — it just never gets scheduled again.

THE CHECK THAT CLOSES IT

- ✓ Name an owner for the running system before the trial starts, not after it succeeds.
- ✓ Ship a health check, per-feature cost tracking, and an alert on errors and slow responses alongside version one.
- ✓ Version your prompts and models like code, so you can answer "what changed?" when quality moves.
- ✓ Answer the data-handling questions in writing early. Retention, personal data and hosting region are cheap to decide up front and expensive to retrofit.

A trial answers a different question than a product

These gaps are so consistent because a trial and a live product are built to answer different questions, and teams often don't notice they've changed questions.

Concern	What the trial asked	What launching asks
Quality	Did it look right in the demo?	What's the pass rate on a fixed test set, and is it moving?
Cost	Negligible at this volume.	What does one user cost per month, and where's the ceiling?
Data	Enough examples to show the idea.	Is the material clean, labeled and consistent enough to trust?
Failure	Didn't come up.	What does the user see when the AI is wrong or unavailable?
Ownership	Whoever built it.	Who monitors, pays for and updates it in twelve months?

None of this argues against running trials. It argues for trials scoped to answer the second column. In practice that's a small change in how you define finished: a trial is done when you have a test set with a recorded score, a cost per user, a written list of failure modes with decided fallbacks, and a named owner. Not when the demo lands.

Some trials should be stopped, and that's a good outcome. One that shows the cost per user is three times the revenue per user has done its job — it saved you the build. The failure isn't stopping; it's spending nine months not knowing.

Is your project ready to launch?

Tick everything that's true today, not everything you intend to do. Nothing is sent anywhere — the score is worked out in your browser.

Launch readiness check

Twelve questions across the five gaps.

Definition of good

- ✓ We have a written test set of real inputs with expected answers.
- ✓ We can state our current pass rate as a number.
- ✓ The test set includes failure cases and every language we ship in.

Cost

- ✓ We know the cost per user per month at projected volume.
- ✓ We log AI usage per request, tagged by feature.
- ✓ We tested whether a cheaper model clears our quality bar.

Data

- ✓ Someone has manually audited a random sample of the source data.
- ✓ Where we label data, we've measured how often labelers agree.

Failure paths

- ✓ We have a written list of failure modes with a decided fallback for each.
- ✓ We've tested under realistic conditions, not demo conditions.

Ownership

- ✓ A named person owns the system after launch.
- ✓ Monitoring, alerting and cost tracking ship with version one.

The smallest useful thing you can do this week

If you only close one gap, close the first. A test set takes a day or two to assemble and immediately makes every other decision cheaper: you can compare models, justify a smaller one, prove a change helped, and show whoever is funding it a number that moves.

It doesn't need a framework. A file of cases and a script that runs them is enough to start, and it already puts you ahead of most projects.

```
# The whole point: one number you can watch move over time.
import json, csv

def run_eval(cases, answer_fn, judge_fn):
    results, passed = [], 0
    for c in cases:
        got = answer_fn(c["input"])
        ok = judge_fn(got, c["expected"])
        passed += int(ok)
        results.append({"id": c["id"], "input": c["input"],
                        "got": got, "expected": c["expected"], "pass": ok})
    return passed / len(cases), results

# Start with simple exact/contains checks. Add a model-as-judge only for
# cases where correctness genuinely needs judgement — and spot-check it.
def contains_judge(got, expected):
    return expected.lower() in got.lower()

if __name__ == "__main__":
    cases = json.load(open("cases.json"))       # id / input / expected
    score, rows = run_eval(cases, my_feature, contains_judge)
    print(f"pass rate: {score:.1%} ({len(cases)} cases)")

    # Append to a CSV so you can see the trend across changes.
    with open("eval_log.csv", "a", newline="") as f:
        csv.writer(f).writerow(["2026-09-02", "prompt-v7", round(score, 3)])
```

Run it before and after every change. The moment you have two numbers to compare, the arguing stops and the engineering starts.