ML LABS INTELLIGENCE

Building a Self-Healing AI Inference Pipeline

Model failures caused duplicate billing, phantom results, and stuck records across a production AI platform. ML LABS engineered reliability into the inference layer — zero duplicate charges, no stuck records, fully self-healing.

CASE STUDY

Author Omar Trejo

Date 2026-01-11

A production AI platform ran inference across multiple model providers for each incoming record. The pipeline worked under normal conditions. When a model call failed — transient network error, provider timeout, partial response — it did not fail cleanly. Records stuck in "Processing" indefinitely. Retry logic called the model again, the billing system charged twice, and two conflicting results existed for the same input. Operators spent hours each week manually reconciling stuck records, investigating duplicate charges, and determining which result was authoritative.

ML LABS engineered reliability into the inference layer so that every failure mode recovered automatically — no duplicate billing, no stuck records, no manual reconciliation. The pipeline went from manual-intervention-required to self-healing.

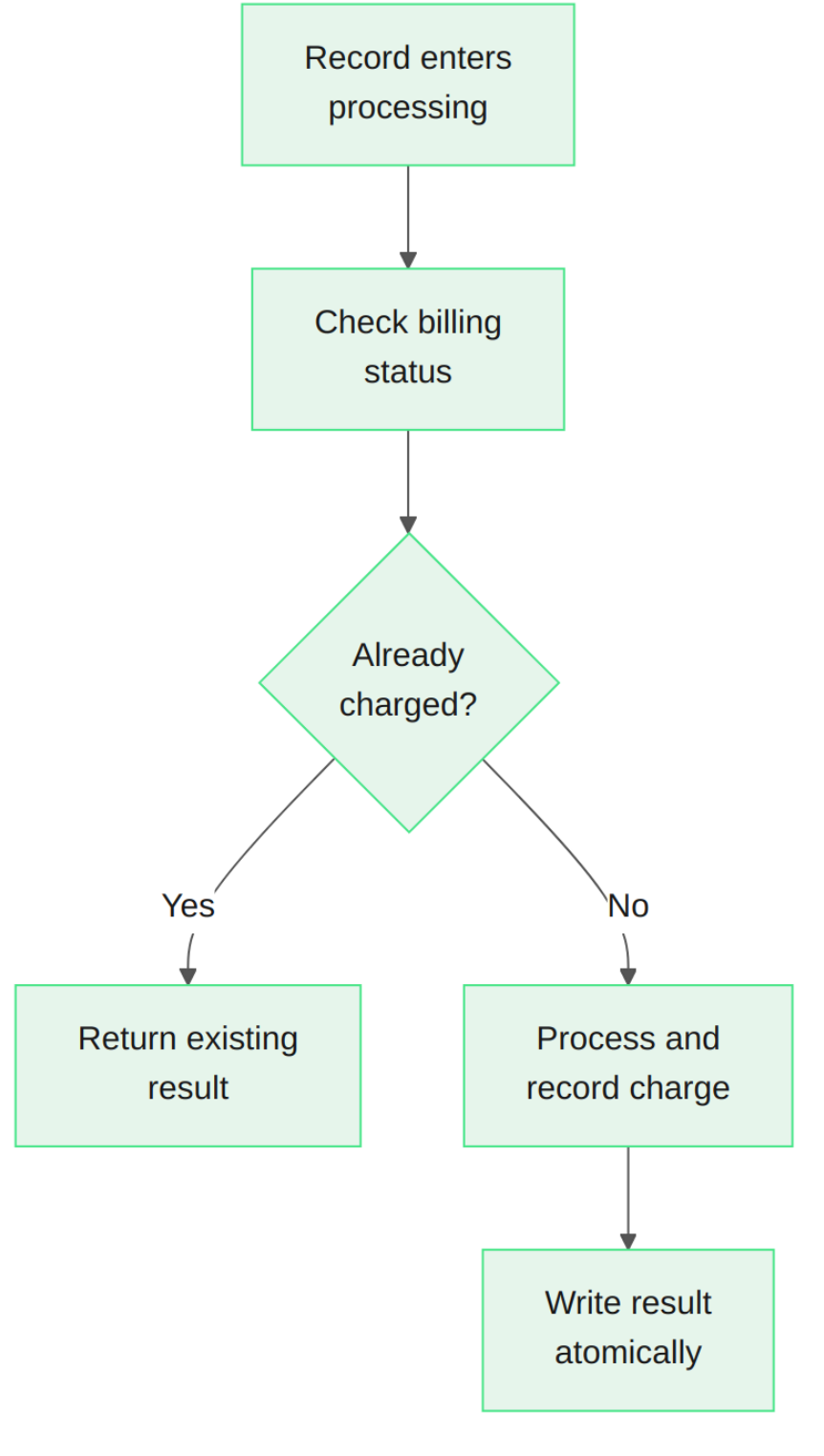
The Failure Modes

The cascading failures traced back to a single architectural gap: the system had no recovery semantics for model inference. When a model returned a valid result but the record's status failed to update, the record appeared stuck even though the work was done. Retries re-invoked the model, generated duplicate charges, and created conflicting results with no automated way to determine which was authoritative.

- Records stuck in intermediate states with no path to resolution
- Duplicate billing on retries because charges were coupled to the model call, not to the record
- Multiple conflicting results for the same input after retry sequences
- No way to test failure recovery paths without waiting for organic production failures

Eliminating Duplicate Billing

The core fix decoupled billing from processing. Previously, every model call generated a charge — whether the result was ultimately used or discarded. ML LABS redesigned billing to be tracked at the record level, so the system knows whether a record has already been charged before any model call executes. Retries that detect an existing charge skip the billing event entirely, regardless of whether the previous result persisted successfully.


Record enters processing
Check billing status
Already charged?
Yes
No
Return existing result
Process and record charge
Write result atomically

The retry behavior was also unified across all model providers. Previously, each provider implemented its own retry assumptions. ML LABS built a shared processing layer with consistent error handling and safeguards against concurrent retries racing each other into duplicate states.

> The most expensive assumption in model inference is that a failed status update means the work was not done. In practice, the model often completed successfully — and retrying it creates a second charge and a state contradiction that only manual investigation can untangle.

Testing Failure Recovery in CI

Retry paths that only exist in production are untestable. ML LABS built failure simulation directly into the platform so that every failure mode could be triggered on demand in any environment. The simulation infrastructure exercises the same code paths that production retries follow — not mocks, but controlled failures through the real pipeline.

This caught a class of bugs that unit tests could not reach. The ambiguous failure mode — model succeeds, status update fails, record appears stuck — required real infrastructure interactions to reproduce. With simulation harnesses, this scenario ran in the CI pipeline on every commit. Regressions that would have surfaced as production incidents surfaced as test failures instead.

Reliable Async Processing

The synchronous architecture was the root fragility. A slow model call blocked the processing thread. A mid-chain failure left records in ambiguous states with no recovery path. ML LABS replaced this with an async architecture that gave the pipeline three properties it previously lacked: durability (work survives process crashes), observability (processing health is directly measurable), and isolation (a slow model does not block new work from entering the pipeline).

Records that exhaust retries are routed to investigation rather than being silently lost. The "stuck in intermediate state" failures were eliminated — every record now reaches a terminal status regardless of whether processing succeeds, fails, or is unavailable.

Unified Ingestion

The platform accepted records through multiple paths: web uploads, network ingestion from clinic systems, and API integrations. Each path had its own error handling and its own assumptions about what happens when inference fails. Some retried aggressively. Others failed silently. Others returned ambiguous errors.

ML LABS unified all ingestion paths so that every record, regardless of how it entered the system, flows through identical processing logic with identical reliability guarantees. This eliminated the class of bugs where "it works from one input path but fails from another."

When This Level of Reliability Is Not Justified

This level of reliability engineering adds infrastructure complexity. For systems where model calls are stateless, cheap, and have no billing implications — internal experimentation pipelines, batch jobs where reprocessing the full set is acceptable, or development environments with no cost constraints — the overhead is not justified. The investment pays off specifically when model calls are expensive, results feed stateful workflows, and duplicate processing creates real-dollar consequences. If rerunning the entire pipeline from scratch is cheaper than building recovery infrastructure, the simpler approach wins.

The distinction matters because over-engineering reliability into a pipeline that does not need it creates maintenance burden without corresponding value. The honest assessment requires measuring the cost of failure — hours of manual reconciliation, duplicate charges, downstream data corruption — against the cost of the infrastructure that prevents it.

Results

The reliability engineering eliminated every class of cascading failure that had required manual intervention.

- Zero duplicate billing across all model providers
- No records stuck in intermediate states — every record reaches a terminal status
- Failure simulation catches retry regressions in CI before they reach production
- All ingestion paths serve identical processing logic and reliability guarantees
- Failed records are surfaced for investigation rather than silently lost

First Steps

1. **Instrument your current failure rate.** Measure how often model calls fail, how many records are stuck in intermediate states, and how many duplicate charges exist. These numbers define the return on reliability investment and set the improvement baseline.
2. **Decouple billing from processing on your highest-cost model.** Ensure the system tracks whether a record has already been charged before any model call executes. This single change eliminates the most expensive class of duplicate processing.
3. **Build one failure simulation test for the ambiguous case.** Trigger a model success with a failed status update and verify that the retry path recovers the record correctly, does not double-charge, and does not produce a conflicting result. This test exercises the exact failure mode that causes the most production damage.

Practical Solution Pattern

Decouple billing from processing so retries never generate duplicate charges. Replace synchronous inference with async processing that recovers automatically from failures. Unify all ingestion paths so reliability guarantees apply identically regardless of how records enter the system. Build failure simulation that exercises ambiguous failure modes in CI on every commit, catching regressions before they reach production.

This works because reliability failures in AI inference are not model failures — they are integration failures in the gap between the model call and the state update. The architecture closes the double-charge gap, makes every processing step independently recoverable, and ensures no work is silently lost. Speed to

production reliability is the differentiator: organizations that compress the cycle from "we know about the failure modes" to "the pipeline self-heals" stop bleeding manual intervention hours and duplicate charges while competitors still reconcile spreadsheets. The measure of success is not whether retries happen, but whether the system state is correct after they do — a working system generating real reliability data beats a monitoring dashboard that catalogs recurring failures. If one defined AI workflow has reliability problems causing operational damage, **AI Workflow Integration** is the direct build path.

References

1. Featonby, M. **Making Retries Safe with Idempotent APIs**. *Amazon Builders' Library*, 2020.
2. Sculley, D., et al. **Hidden Technical Debt in Machine Learning Systems**. *NeurIPS*, 2015.
3. Rosenthal, C., & Jones, N. **Chaos Engineering: System Resiliency in Practice**. *O'Reilly Media*, 2020.
4. Principles of Chaos Engineering. **Principles of Chaos**. *principlesofchaos.org*, 2019.
5. Amazon Web Services. **Using Dead-Letter Queues**. *AWS Documentation*, 2024.
6. Portkey AI. **Retries, Fallbacks, and Circuit Breakers in LLM Apps**. *Portkey*, 2024.
7. NIST. **Healthcare Security Rule Guidance (SP 800-66)**. *NIST*, 2024.