**NYC-ML\_PROJECT**

**Why are we doing all this?**

We’re not just “making a model.” We’re building a **production service** that people can trust. In a real company, models touch money, users, SLAs, and on-call. Each step you saw maps to a real risk that burns teams when skipped. Think of this as **risk → control**.

**1) Data contract → (prevents silent data drift & blame games)**

**Risk if skipped:** A column’s meaning changes (“payment\_type=0” used to mean X, now Y). Your model tanks. Everyone argues whose fault it is.

**Control:** The contract freezes **schema, meanings, timezones, null rules, valid ranges, refresh SLA**. When the upstream changes, you have a documented expectation and can say “contract violated → fix upstream or version bump.”

**What this buys you:** Predictable inputs; clear ownership; fewer surprises.

**How you say it in interviews:**

“We start with a data contract so schema/semantics and SLA are explicit. Any change is versioned and re-materializes features.”

**2) Validator → (prevents bad data entering training/serving)**

**Risk if skipped:** Negative distances, invalid codes, non-joinable zone IDs slip in. You train on garbage; serving crashes or outputs nonsense.

**Control:** A gatekeeper that **coerces types**, **enforces ranges/codes**, **drops non-joinable rows**, **adds is\_anomaly**, and **checks freshness** against the SLA.

**What this buys you:** Stability. You keep bad rows out, and you have a **quality report** to debug incidents.

**Interview line:**

“All raw files pass through a validator. We enforce the contract, compute duration\_minutes, tag anomalies, and check monthly freshness.”

**3) Labeling (clear target) → (prevents leakage and confusion)**

**Risk if skipped:** You accidentally use future info to predict the present (data leakage). Offline metrics look amazing; production fails.

**Control:** Define a **clean, reproducible label** (e.g., HIGH\_TOTAL) and draw a wall between **label-only fields** and **serve-time features**.

**What this buys you:** Honest offline metrics; model that behaves similarly online.

**Interview line:**

“We separate label-only columns from serving features to eliminate leakage.”

**4) Serving-safe features → (prevents train/serve skew)**

**Risk if skipped:** Training uses features you don’t have at prediction time (e.g., dropoff info). Or one-hot columns appear/disappear between days.

**Control:** Build features **from values available at prediction time** and keep **fixed vocabularies** for categoricals so columns are stable.

**What this buys you:** Same features in training and production → consistent behavior.

**Interview line:**

“We generate features with the exact same code for train and serve, with fixed categorical vocab so the column set is stable.”

**5) MLflow tracking → (prevents “can’t reproduce last week’s model”)**

**Risk if skipped:** You can’t tell which params/data produced a model in production. A bug appears; rollback is guesswork.

**Control:** Log **params, metrics, artifacts, env** for every run; register versions.

**What this buys you:** Reproducibility and safe rollback.

**Interview line:**

“Every run is logged in MLflow; models are versioned so we can compare and roll back.”

**6) API + Docker + K8s manifest → (makes the model a service, not a notebook)**

**Risk if skipped:** The model is stuck in a notebook; nobody can call it. Or it behaves differently on each machine.

**Control:** A tiny **FastAPI** service, **Dockerfile** for identical environments, and a **K8s deployment** spec for real hosting.

**What this buys you:** Anyone can hit /predict; infra can deploy and scale it.

**Interview line:**

“We containerize a FastAPI inference service and provide a K8s manifest so ops can deploy with probes and rollbacks.”

**7) Metrics & monitoring (/metrics) → (prevents slow outages & silent failures)**

**Risk if skipped:** The API slows down, returns errors, or scores collapse—and you find out from angry users.

**Control:** Expose **request counts, latency histograms, error rates, score distribution** in Prometheus format; graph in Grafana; set alerts.

**What this buys you:** You see issues first and react quickly.

**Interview line:**

“We export Prometheus metrics (latency p95, 5xx) and watch score distributions for drift; alerts page on-call.”

**8) CI (tests + lint) → (prevents broken merges)**

**Risk if skipped:** Someone pushes code that breaks validation/features; production deploys junk.

**Control:** **pytest + lint** run on every PR. Red → no merge.

**What this buys you:** Fewer regressions; discipline that interviewers recognize.

**Interview line:**

“CI runs tests and lint per PR; we don’t merge if invariants fail.”

**9) Runbooks & postmortems → (reduces downtime, captures learning)**

**Risk if skipped:** When things break, people panic; fixes are ad hoc; the same incident repeats.

**Control:** A **runbook** with triage steps and a **blameless postmortem** template with owners and follow-ups.

**What this buys you:** Faster recovery, better organization memory.

**Interview line:**

“We keep a runbook for ‘all-zero predictions’ and write postmortems to harden tests/monitors after incidents.”

**10) Governance / PII → (keeps you compliant)**

**Risk if skipped:** Privacy violations, unclear retention, sensitive data in logs.

**Control:** Explicit **PII policy, access roles, retention windows**, and “no PII in logs/metrics.”

**What this buys you:** Safety with legal/compliance and trust with partners.

**Interview line:**

“We declare access roles, retention, and ‘no PII in logs’; feature snapshots have TTLs.”

**The simplest Big Picture (stick this to your monitor)**

**Idea → Data contract → Validator → Features → Model (tracked) → API (containerized) → Monitoring → CI → Runbook → Governance**

That is exactly what turns a notebook into a **hireable, production-ready project**. Interviewers hear these words and think: “This person understands how real ML gets shipped.”

**STEP 1: DATA CONTRACT (prevents silent data drift & blame games)**

“We start with a data contract so schema/semantics and SLA are explicit. Any change is versioned and re-materializes features.”

**STEP 2: VALIDATOR**

Mental model (repeat this sequence)

“**Load → Coerce → Validate → Derive → Freshness report**”

* **Load** one month’s CSV/Parquet.
* **Coerce** dtypes (timestamps→UTC, numerics→float/int).
* **Validate** per column rules (required columns, ranges/codes, joinability).
* **Derive** duration\_minutes, is\_anomaly.
* **Freshness**: check latest mirrored month against the SLA and output a **quality report** (counts, anomaly rate, rule hits).

If you can say that in order, you can code it under pressure.

**Files to add:** 1.src/contract\_spec.py

2. src/validator.py

3.tests/test\_validator.py

**How to run it**

# from your repo root

git checkout -b day3-validator

# add the two files:

# src/contract\_spec.py

# src/validator.py

# and the test:

# tests/test\_validator.py

git add src/contract\_spec.py src/validator.py tests/test\_validator.py

git commit -m "feat(validator): enforce data contract with anomaly flags + freshness report"

pytest -q

**CLI (optional):**

python -m src.validator data/nyc\_taxi\_2025-03.csv --month 2025-03

This prints a JSON **quality report** (rows kept, anomaly rate, rule hit counts, freshness).

**What you can confidently say to interviewers**

* “Our **validator** reflects the **data contract**: we coerce types to canonical forms, enforce ranges/codes, compute duration\_minutes, and raise anomaly flags instead of silently mutating training data.”
* “We generate a **freshness signal** tied to the SLA (latest month by the 5th, 12:00 CET) so downstream jobs can halt on stale inputs.”
* “Joinability to reference maps (Taxi Zones) is explicit; non-joinable rows are dropped by design.”

**The big picture (one sentence)**

A **data contract** is a safety manual for a table: what each column means, the units, what values are allowed, how often data arrives, and what to do when something looks wrong—so models don’t learn garbage and services don’t break.

**1) Refresh & SLA — “When do we expect new data?”**

* **Cadence:** “We mirror a new file every month when TLC publishes.”
* **SLA:** “By the **5th, 12:00 CET**, we *must* have last month’s data.”
* **Why it exists:** So downstream jobs (features/training) know whether to run or stop. If it’s missing on the 5th at noon, we alert instead of silently training on stale data.
* **What the validator will check:** “Do I see any dropoff timestamps inside the expected month window?” If not → freshness = false.

**2) Global semantics — “How do we read numbers/times consistently?”**

* **Timezone = UTC:** Everyone in the company sees the *same* time for a trip; dashboards can display CET later.
* **Units:** Money is **USD**, distance is **miles**. (No ‘is it km or miles?’ bugs.)
* **Primary key (best-effort):** We try to uniquely identify a row with vendor + pickup time + pickup zone (+ a hidden ingest id). It’s not perfect, but good enough to deduplicate obvious duplicates.

**3) Columns grouped by *type of meaning***

Think in groups, not a long list. Each group has the same “why”:

**A) Identifiers & keys**

* **VendorID (1,2,6,7)**  
  *Why:* Who recorded the trip. Fixed set.  
  *Rule:* If a new unknown code appears, we **flag** it. Unknown vendor can shift behavior.

**B) Time columns**

* **tpep\_pickup\_datetime, tpep\_dropoff\_datetime (UTC)**  
  *Why:* Define the trip window.  
  *Rules:* pickup ≤ dropoff; trip duration 1–720 minutes.  
  *Reason:* Durations like 0 minutes or 2 days are errors → **flag**.

**C) Location columns**

* **PULocationID, DOLocationID (zone IDs)**  
  *Why:* They power most features (demand patterns by zone).  
  *Rule:* Must exist in the official Taxi Zone list.  
  *Action:* If a zone ID doesn’t exist → **drop the row** (cannot map it reliably).

**D) Counts & continuous measures**

* **passenger\_count (0–8)**  
  *Why:* Useful feature; devices sometimes write 0 or nonsense.  
  *Rule:* <0 → set to NULL and **flag** (not fatal).
* **trip\_distance (miles)**  
  *Why:* Key feature and sanity check.  
  *Rule:* Must be ≥0 and <200. Negative means device error (set null + **flag**). Enormous values are capped + **flagged**.

**E) Categoricals / codes**

* **RatecodeID** (1 Standard, 2 JFK, 3 Newark, 4 Nassau/Westchester, 5 Negotiated, 6 Group, 99 Unknown)  
  *Why:* Pricing/route type.  
  *Rule:* If missing, use 99 (Unknown) so features stay stable.
* **store\_and\_fwd\_flag (Y/N)**  
  *Why:* Whether the device buffered the trip (connectivity).  
  *Rule:* Missing → default N; weird values → set N + **flag**.
* **payment\_type (0 Flex Fare, 1 Credit, 2 Cash, 3 No charge, 4 Dispute, 5 Unknown, 6 Voided)**  
  *Why:* Tells us about transaction patterns.  
  *Rule:* Outside this set → **flag** (and treat as unknown).

**F) Money columns (all USD)**

* **fare\_amount, extra, mta\_tax, tip\_amount, tolls\_amount, improvement\_surcharge, congestion\_surcharge, total\_amount**  
  *Why:* Targets and QA.  
  *Rule:* Money cannot be negative; negative → set null + **flag** (except total\_amount which we **flag** if negative because it breaks totals).  
  *Note:* tip\_amount may be zero (cash tips aren’t captured).

**G) Derived by us**

* **duration\_minutes = dropoff − pickup** (1–720 valid)
* **is\_anomaly = 1** if *any* rule tripped (range, code, etc.)
* **ingest\_date** for partitioning

**4) What the validator does in human terms**

1. **Load** a month’s file.
2. **Coerce** types: timestamps → UTC, numbers → numbers.
3. **Check** each group’s rules (above).
4. **Drop** only what is fundamentally unusable (e.g., fake zones, missing required keys).
5. **Flag** anything suspicious but still usable (e.g., unknown vendor, negative fare set to null).
6. **Derive** duration\_minutes, set is\_anomaly.
7. **Freshness**: confirm the file actually has rows in the month we expect.
8. **Report** anomaly counts and keep going.

If you can say that pipeline out loud, you “get” the contract.

**5) A tiny 1-row example (see the rules “light up”)**

Imagine we ingest one row:

* VendorID=9 (unknown)
* pickup=2025-03-01T08:00Z, dropoff=2025-03-01T08:10Z → duration=10 min ✅
* PULocationID=142 (valid), DOLocationID=999999 (not in map ❌)
* passenger\_count=-3 (nonsense ❌)
* trip\_distance=-1.0 (negative ❌)
* payment\_type=1 (Credit ✅)
* fare\_amount=12.50 (≥0 ✅)

**What happens:**

* Unknown vendor → is\_anomaly=1 (flag), keep row.
* Non-joinable DOLocationID → **drop the row** (can’t map the zone).
* passenger\_count <0 → set to NULL and **flag**.
* trip\_distance <0 → set to NULL and **flag**.
* Others OK.

**Bottom line:** one fatal error can drop the row (bad zone), while recoverable weirdness is just **flagged** so it won’t secretly poison training.

**6) Why modelers care (not just data engineers)**

* **Better data → better models.** Flags let you *exclude* or *weight down* dubious records.
* **Stable features.** Enums (codes) and defaults ensure the feature matrix doesn’t change shape day to day.
* **Reproducibility.** The contract + validator make your experiments repeatable and explainable.

**7) Your mental cheat-cards (say these)**

* **Times & units:** UTC + USD + miles.
* **SLA:** latest month by the **5th, 12:00 CET**.
* **Drop vs Flag:** *Drop* only non-joinable or missing required keys. *Flag* out-of-range or unknown codes.
* **Derived:** duration\_minutes 1–720; is\_anomaly if any rule hit.
* **Categoricals:** fixed codebooks → stable columns later.

**8) A 60-second self-check (answer without looking)**

1. If trip\_distance = -2, do we drop or flag? → **Flag** (set null), keep row.
2. If DOLocationID isn’t in the zone map? → **Drop row** (can’t join).
3. If payment\_type = 9? → Treat as unknown and **flag**.
4. What’s our freshness test tied to? → “Has rows in the target **month window** by the **5th, 12:00 CET**.”