Data Requirements, Data Provenance, and Acquisition Strategies

DAEN 459 – Fall 2026

Alexander Abuabara



Example: Bike-Share Pipeline

Capstone goal: build a pipeline that delivers clean, documented, analysis-ready data for forecasting hourly bike demand per station.

Candidate sources

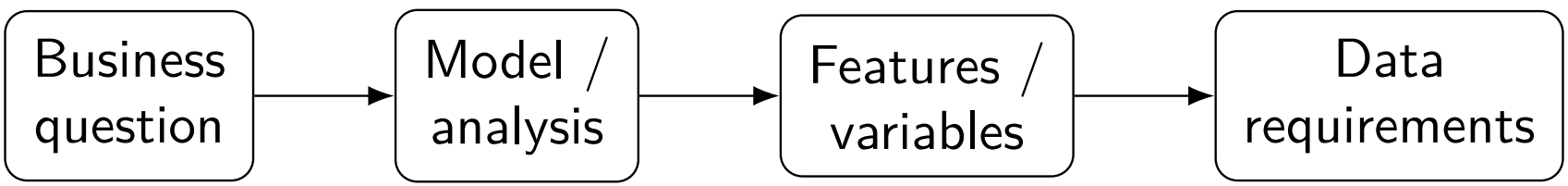
- Trip records (monthly CSV dumps from the operator)
- Live station status (GBFS JSON API)
- Weather (public API, hourly)
- City events calendar (web page, semi-structured)
- Census tract demographics (annual release)

Downstream consumers: a forecasting model, a dashboard, and a monthly report to the city.

Why Requirements Come First

- Requirements are derived from decisions and questions, not from whatever data happens to be available.
- Poorly specified requirements lead to rework, scope creep, and models trained on the wrong grain of data.
- Capture requirements as testable statements that can be checked automatically later.

Chain of reasoning



Dimensions of a Data Requirement

Dimension	**Question to answer**
Content	Which entities and attributes are needed?
Granularity	Per trip, per station-hour, per day?
Coverage	Which time span, stations, geography?
Volume & velocity	Rows per day? Batch or streaming?
Quality	Acceptable missingness, error rate, duplicates?
Timeliness	How stale can the data be?
Access & legal	Licenses, privacy, retention limits?

Example: Testable Requirements

Requirement	**Acceptance check**
R1: Station-hour demand for 36 months	No gaps longer than 2 hours per station
R2: Weather joined at station-hour	$\geq$ 98% of rows have temperature and precipitation
R3: Station IDs consistent across sources	100% of trip station IDs exist in the station dimension
R4: Forecast inputs refreshed daily by 06:00	Pipeline SLA monitor; alert if late
R5: No rider-level PII stored	Schema scan finds no user ID or birth year

Each requirement maps to an automated data-quality test in the pipeline.

Example: Gaps Found Early

- The forecasting model needs **station-hour** counts.
- Trip files record **trip start/end timestamps**, so counts must be derived by aggregation.
- Gap found: rebalancing trips (staff moving bikes) are not labeled, so they inflate demand.
- **Action:** add a requirement to identify or filter rebalancing events, and ask the operator whether a flag exists.

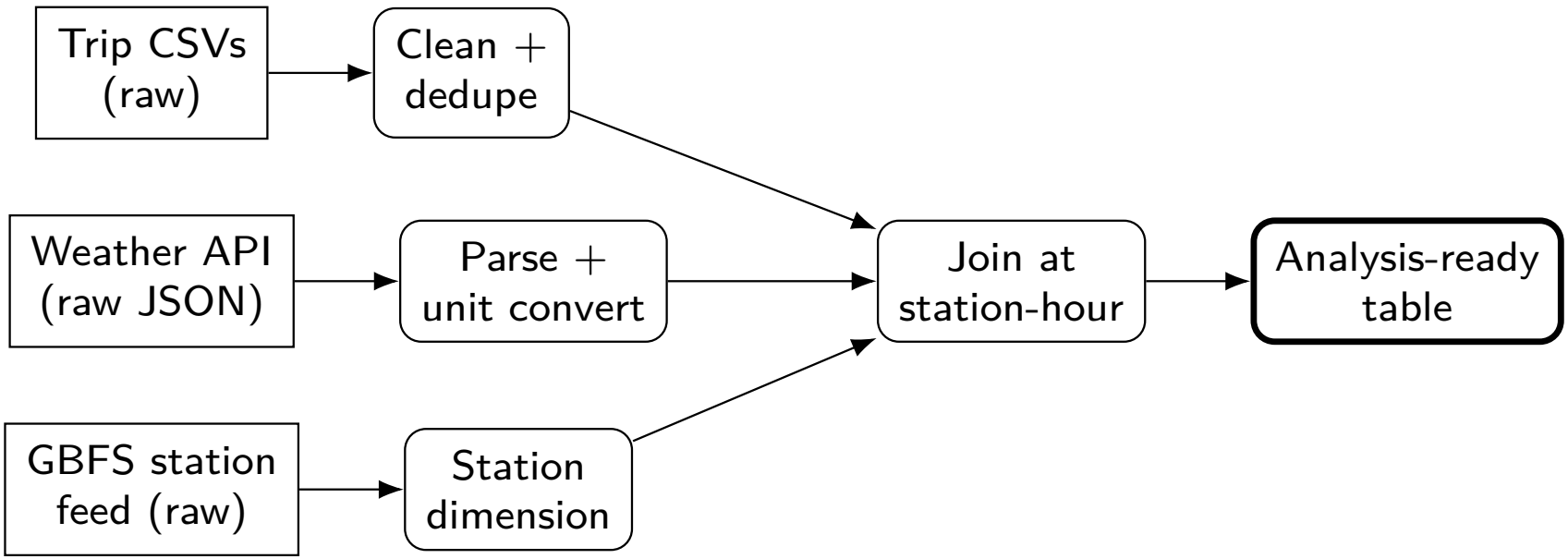
What Is Data Provenance?

Provenance is the recorded history of a dataset: where it came from, who or what transformed it, and when.

- **Origin:** source system, URL, owner, license
- **Process:** code version, parameters, environment
- **Lineage:** the chain of datasets and transformations to the final output
- **Time:** extraction time and the period the data describes

Why it matters: reproducibility, debugging, auditing, trust, and compliance.

Lineage Diagram for the Example



Every arrow should be traceable to a versioned script and a logged run.

Provenance Metadata Record

Store a small metadata file next to every raw extract:

```
source_name:    weather_hourly_api
source_url:     https://api.example.org/v1/hourly
license:        CC-BY-4.0
retrieved_at:   2026-10-04T05:12:09-05:00
period_covered: 2026-10-03T00:00 / 2026-10-03T23:00
request_params: {lat: 30.62, lon: -96.33, units: metric}
raw_file:       raw/weather/2026-10-03.json
sha256:         9f2c...e41a
code_version:   git commit 3b7d1a2
```

A checksum plus a commit hash lets anyone verify that a result came from a specific input and a specific version of the code.

Provenance Practices for a Capstone

- Keep raw data **immutable**; never edit in place.
- Separate layers: `raw` → `clean` → `curated`.
- Version code in Git; record the commit in each run's metadata.
- Pin environments (`requirements.txt`, `renv.lock`, or a container image).
- Keep a data dictionary and a changelog of schema changes.
- Use workflow tools (e.g., Make, Snakemake, Airflow, dbt) so lineage is declared, not remembered.

Example: Provenance Saves the Day

Symptom: Forecast error jumps in March.

Investigation using provenance records

(1) Compare run metadata: the weather source changed its temperature unit from Celsius to Fahrenheit on March 1.

(2) Raw files confirm the change; the parsing step assumed Celsius.

(3) Fix the parser, add a unit-validation test, and reprocess from immutable raw files.

Without raw snapshots and recorded parameters, the root cause would be hard to prove and the data impossible to rebuild.

Acquisition Options

Strategy	**Example**	**Main trade-off**
Bulk download	Monthly trip CSVs	Simple, but latency and format drift
API pull	Hourly weather	Structured, but rate limits and auth
Streaming feed	GBFS station status	Fresh, but needs durable storage
Web scraping	City events page	Fragile; check terms of use
Purchase / partner	Operator data agreement	Rich, but cost and legal terms
Generate / simulate	Synthetic trips for testing	Safe, but may not match reality

Choosing a Strategy: Decision Factors

- **Fit to requirements:** does it satisfy granularity, coverage, and timeliness?
- **Reliability:** uptime, versioning, documented schema, stable identifiers.
- **Cost and effort:** licensing fees, engineering time, storage.
- **Legal and ethical:** license, terms of service, privacy, consent.
- **Risk:** single point of failure, vendor lock-in, source discontinuation.

Score each candidate source against these factors before committing.

Example: Source Evaluation Matrix

Source	Fit	Reliability	Cost	Legal
Trip CSVs	High	High	Free	Open
Weather API A	High	High	Free tier	Attribution
Weather API B	Medium	Medium	Paid	Restrictive
Events scraping	Medium	Low	Free	Unclear
Census tables	Medium	High	Free	Public domain

Decision: use Trip CSVs, Weather API A, and Census; treat events scraping as optional and defer until the baseline pipeline works.

Example: Resilient API Acquisition

Retries with backoff, raw payload storage, and logged metadata:

```
import requests, time, json, hashlib, datetime as dt

def fetch(url, params, retries=4):
    for k in range(retries):
        r = requests.get(url, params=params, timeout=30)
        if r.status_code == 200:
            return r.content
        time.sleep(2 ** k)          # exponential backoff
    raise RuntimeError("fetch failed")

payload = fetch(URL, PARAMS)
path = f"raw/weather/{dt.date.today()}.json"
open(path, "wb").write(payload)
meta = {"retrieved_at": dt.datetime.now().isoformat(),
        "sha256": hashlib.sha256(payload).hexdigest(),
        "params": PARAMS}
json.dump(meta, open(path + ".meta.json", "w"))
```

Acquisition Risks and Mitigations

Risk	Mitigation
Schema drift	Schema validation at ingestion; alert on new or missing columns
API rate limits or outage	Backoff, caching, backfill jobs
Missing history	Plan a historical backfill early; document gaps
Licensing surprises	Record license in metadata; review before publishing
Privacy exposure	Minimize fields, aggregate early, avoid storing PII
Sampling bias	Compare coverage against known population; document limits

From Plan to Deliverables

Suggested design-document sections

(1) Problem statement and decisions supported
(2) Data requirements table with acceptance checks
(3) Source inventory and evaluation matrix
(4) Acquisition plan (method, frequency, backfill)
(5) Provenance plan (metadata, versioning, lineage)
(6) Risk register and mitigation
(7) Timeline and milestones

Milestone Checklist

- ☐ Requirements reviewed with stakeholders
- ☐ All sources evaluated and licenses recorded
- ☐ Raw ingestion working for every chosen source
- ☐ Metadata and checksums captured automatically
- ☐ Data-quality tests mapped to each requirement
- ☐ Lineage diagram and data dictionary published

Key Takeaways

- Start from decisions; express requirements as **testable** statements.
- Treat provenance as a first-class output: immutable raw data, metadata, versioned code.
- Choose acquisition strategies by weighing fit, reliability, cost, legality, and risk.
- Build the thin end-to-end pipeline first, then expand sources.

Questions?!?