DO YOU ACTUALLY NEED A FEATURE STORE?

Shared feature infrastructure

One batch model may work well with a versioned table. Several training and serving paths can need stronger shared contracts.

Today: decide from the system's needs, then understand what the tooling does.

What are your system's needs?

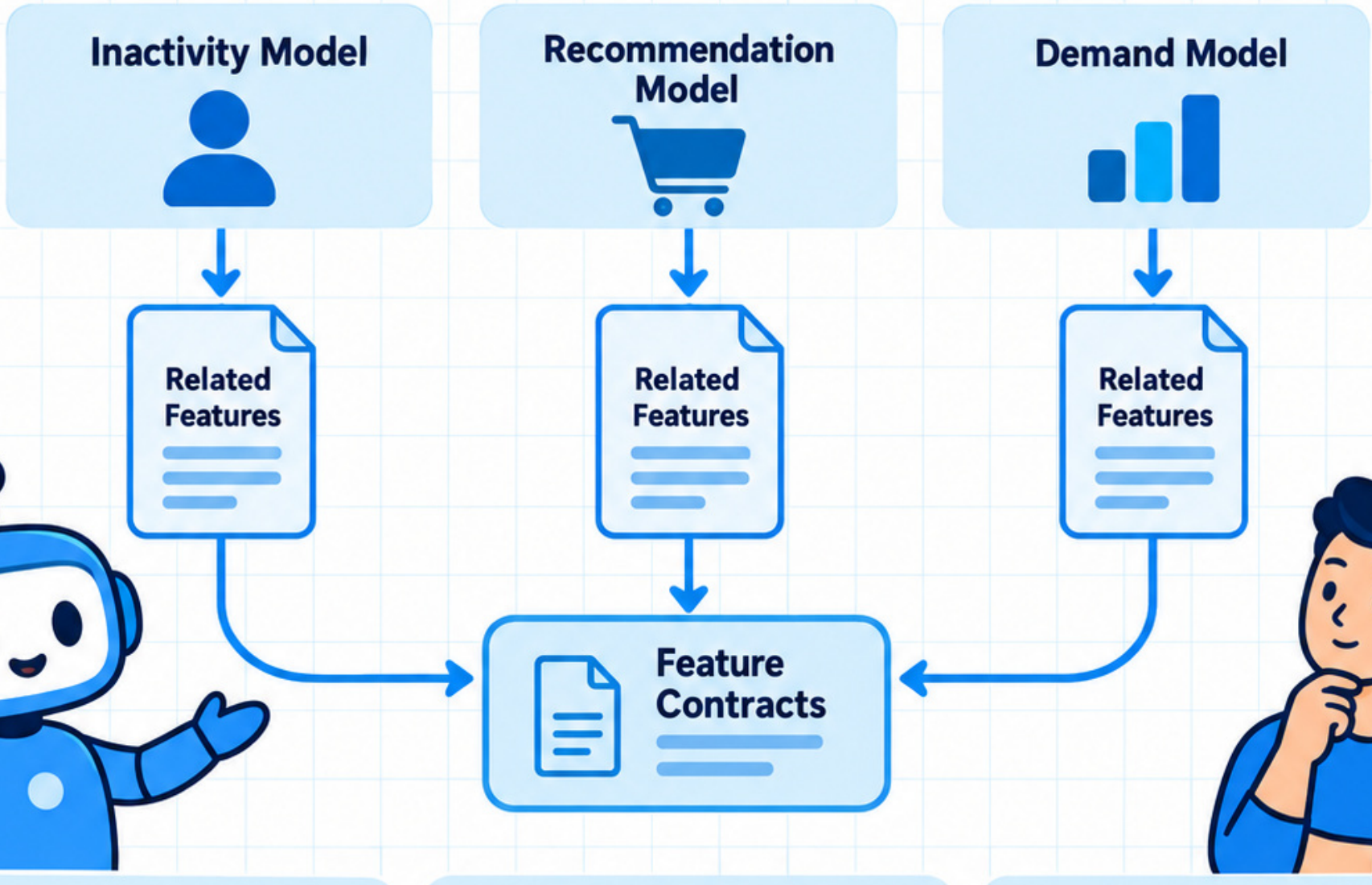
One batch model may work well with a versioned table.

Several training and serving paths can need stronger shared contracts.

Versioned table

Shared feature service

START WITH THE COORDINATION PROBLEM



Our fictional store now has an inactivity model, a recommendation model and a demand model.

Teams repeatedly build and retrieve related features. Definitions, timestamps and missing-value rules can diverge.

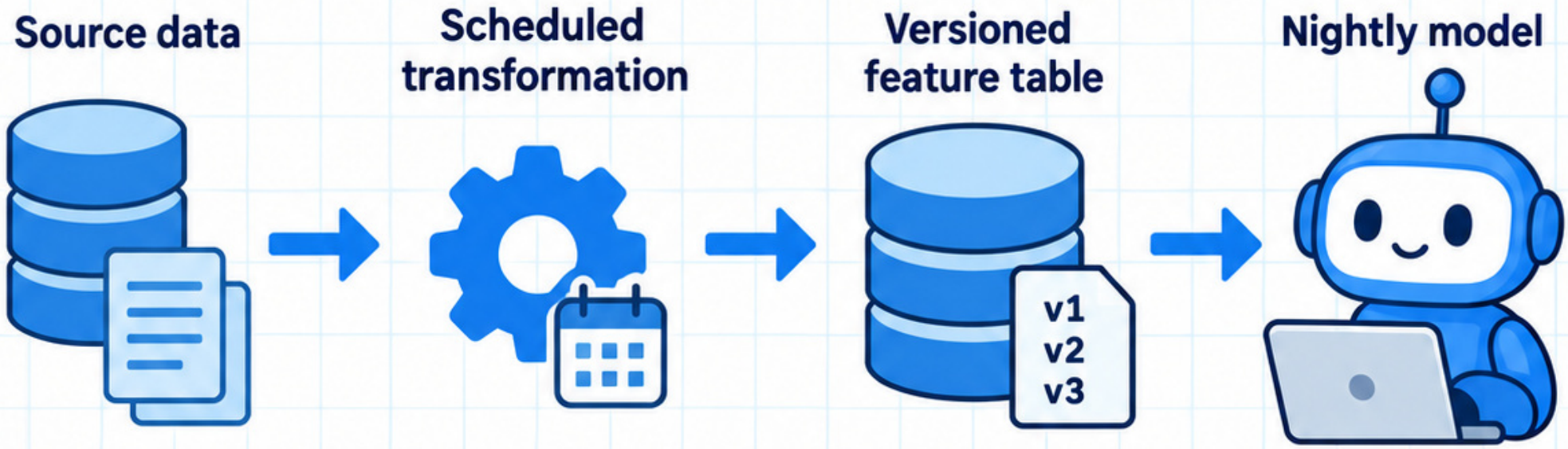
Shared infrastructure is useful only if the shared contract is real.

A TABLE CAN BE THE RIGHT FIRST DESIGN

One nightly model with a retained feature snapshot and clear ownership may not need a feature store.

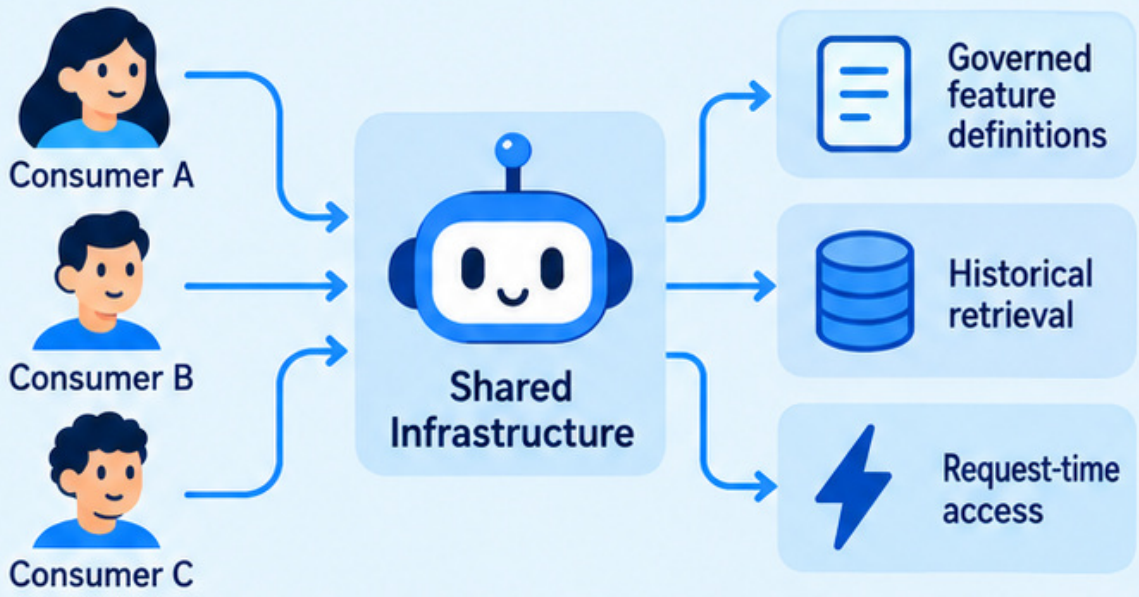
A scheduled transformation, versioned table and reproducible joins can satisfy the contract.

Keep the smallest design that meets history, freshness and serving requirements.



LOOK FOR REQUIREMENTS THAT REPEAT

Consider shared infrastructure when several consumers need governed feature definitions, historical retrieval and request-time access.

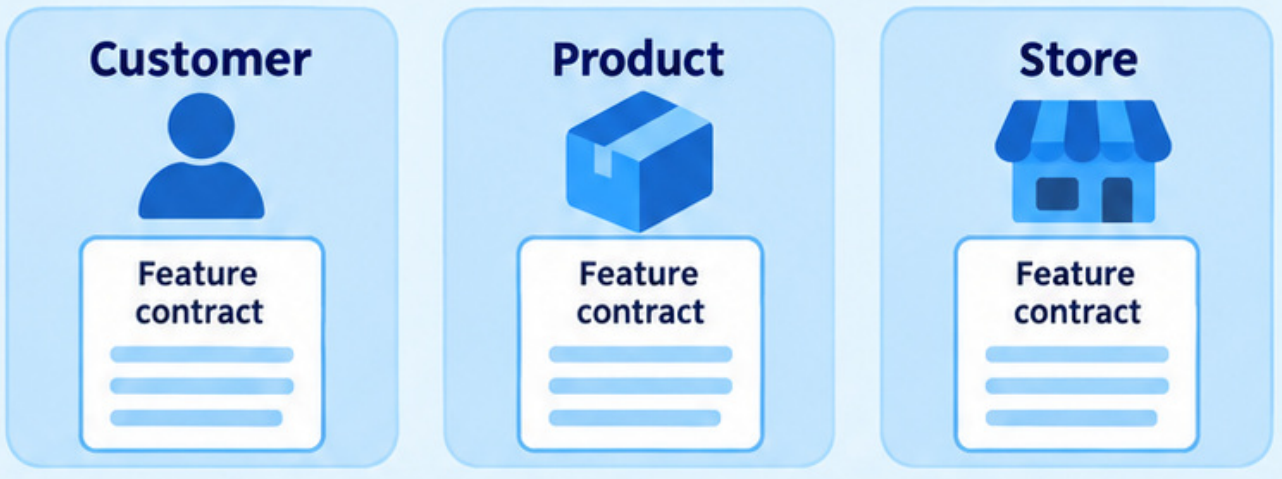


Also consider repeated training-serving mismatches and duplicated operational work.

More feature columns alone do not justify another service.

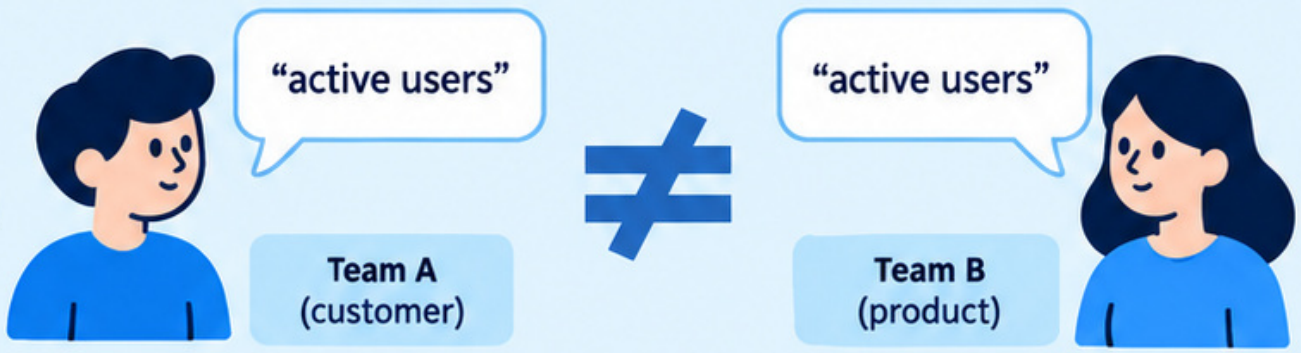
EVERY LOOKUP NEEDS THE CORRECT ENTITY KEY

An entity identifies what the feature describes: customer, product or store.



A feature definition adds meaning, units, time window, missing rules and owner.

Similar names do not prove two teams mean the same feature.



REUSE THE MEANING, NOT JUST A COLUMN NAME

Toy example:

Team A

purchase_count

means completed purchases in the past 30 days.

completed purchases in the past 30 days

Team B

purchase_count

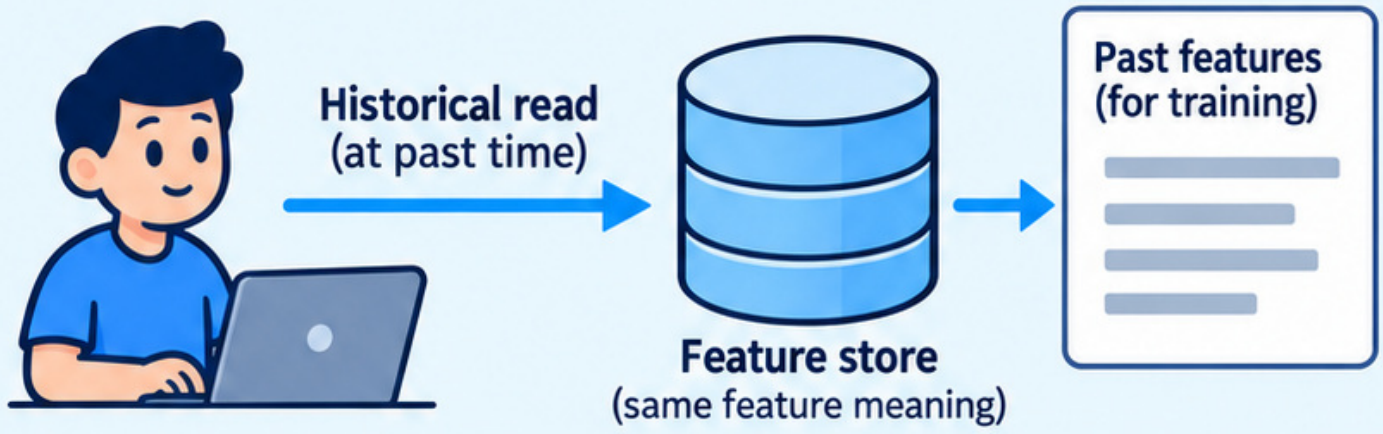
means all created orders in the past 7 days.

created orders in the past 7 days

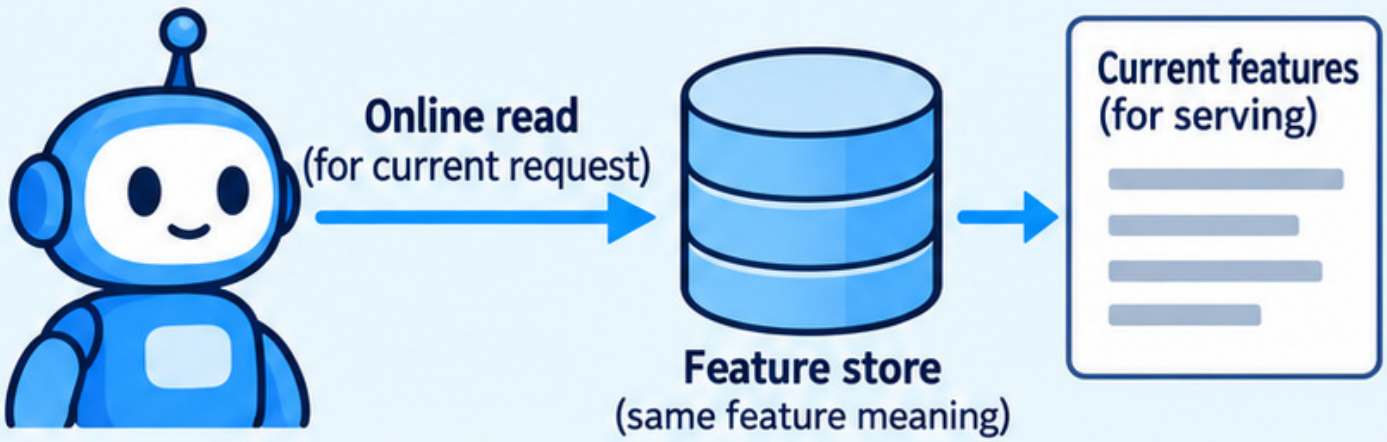
These are different features. Give them distinct definitions and versions before sharing.

HISTORICAL AND ONLINE READS ANSWER DIFFERENT QUESTIONS

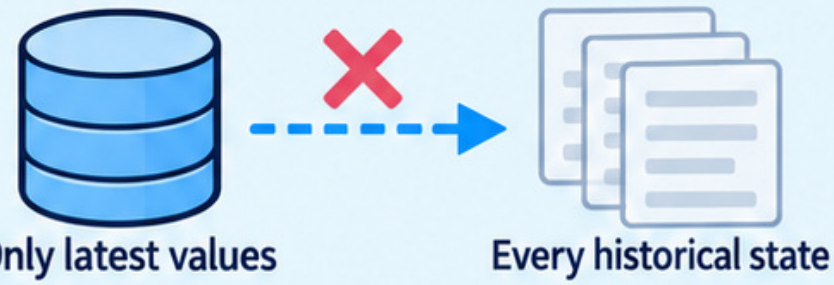
Training asks:
what features were available for this entity at that past prediction time?



Online serving asks:
what approved feature values can this request use now?



A store of only the latest values cannot by itself reconstruct every historical state.



FEATURES STILL NEED TO BE PRODUCED

Source events go through owned transformations and quality checks before feature values are published.

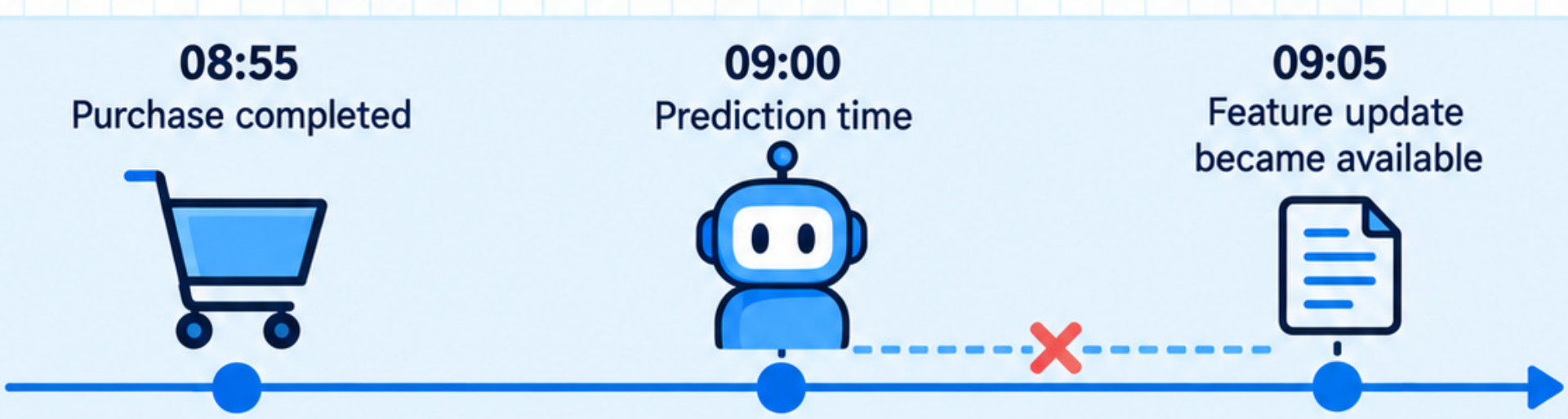
Materialization moves prepared values into a serving store.

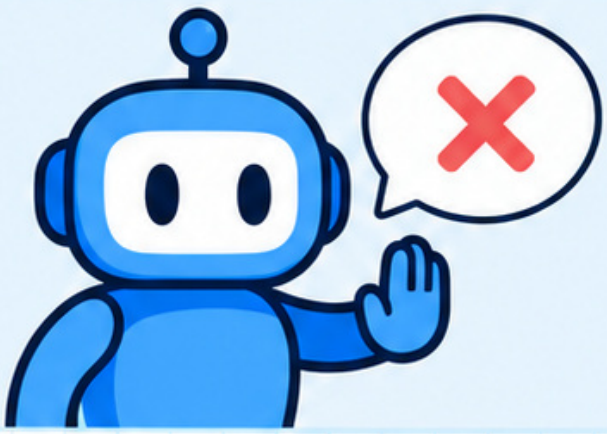
Registering a feature definition does not automatically build every business transformation or prove its correctness.

EVENT TIME ALONE CAN LEAK FUTURE KNOWLEDGE

Toy event:

Purchase completed at 08:55.
Feature update became available at 09:05.
Prediction time: 09:00.

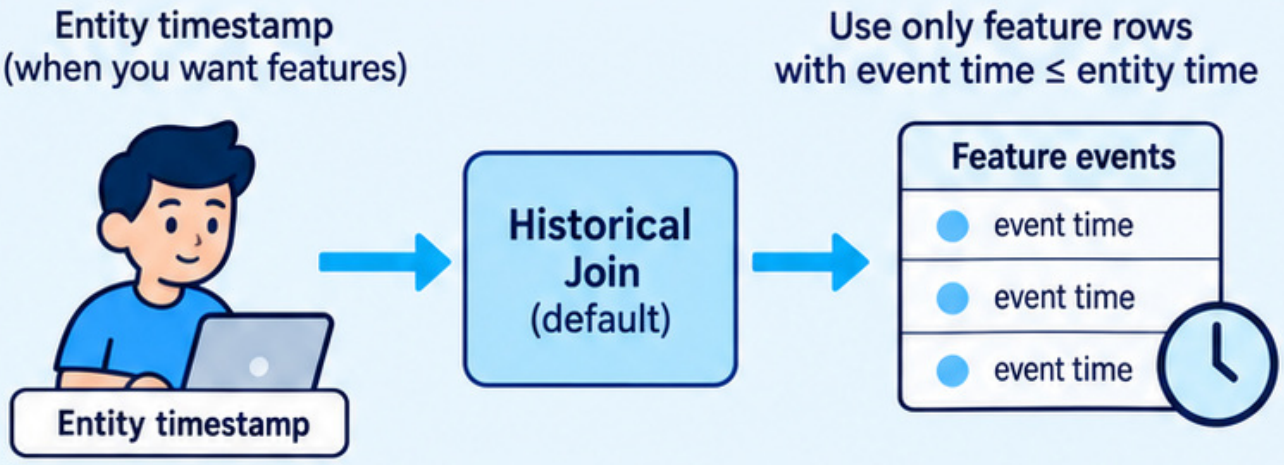




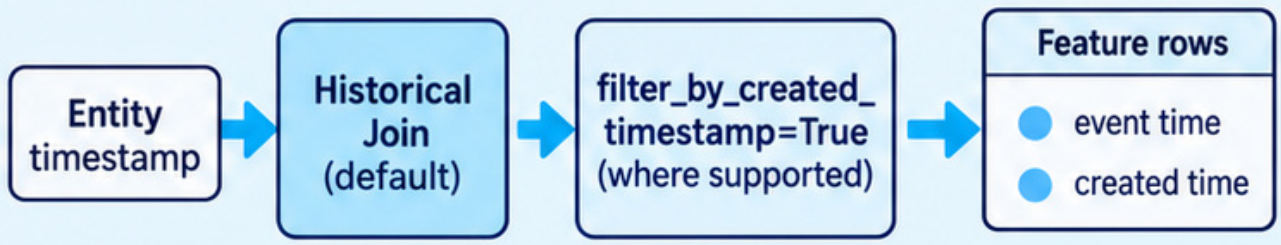
The update was not **available** for that prediction, even though the purchase happened earlier. Historical reconstruction must respect availability.

CHECK WHAT THE HISTORICAL JOIN ACTUALLY ENFORCES

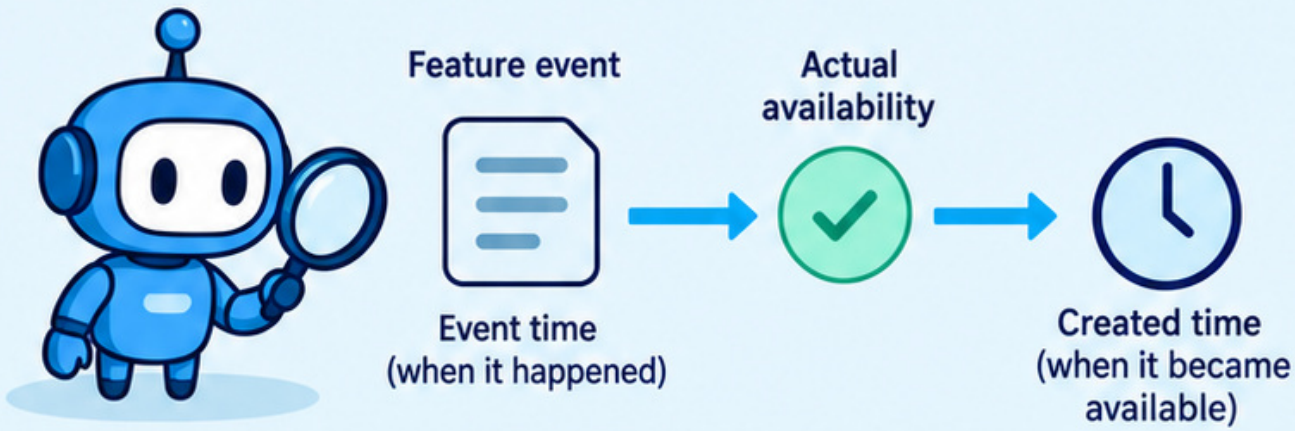
Feast can retrieve historical features using entity timestamps. Its default join constrains feature event time.



Where supported, filter_by_created_timestamp=True also limits created time.



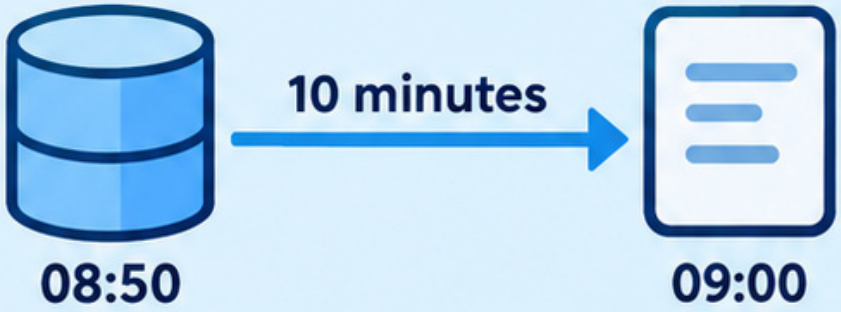
That timestamp must represent actual availability. Verify backend support and test late corrections.



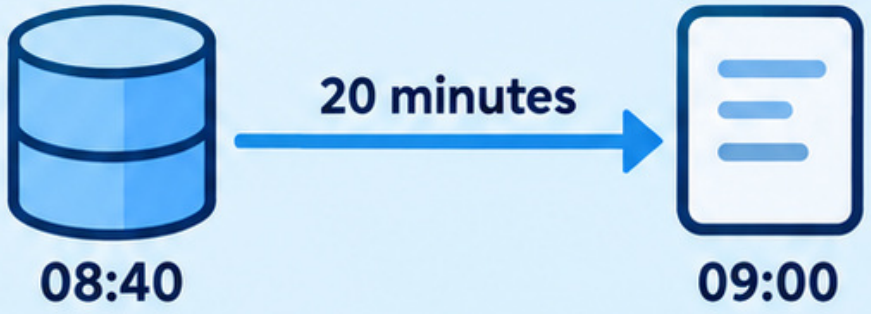
FRESHNESS IS A SERVING POLICY

Toy request time: 09:00. Maximum allowed age: 15 minutes.

Feature at 08:50: age 10 minutes, within limit.



Feature at 08:40: age 20 minutes, too old.



Historical lookback limits and storage TTL settings are not a complete application freshness contract.

A CORRECTION NEEDS A REVISION TRAIL

A late correction can change a feature for a past event time.

Original value	Later correction
event time 2024-01-10	event time 2024-01-10
feature value **A**	feature value **B**
available at 2024-01-11	available at 2024-01-20

Retain enough version and availability history to explain what past predictions actually used.

Feature history

Original value	
event time	2024-01-10
feature value	**A**
available at	2024-01-11

Later correction	
event time	2024-01-10
feature value	**B**
available at	2024-01-20

A corrected training dataset can be useful, but must not silently masquerade as the original request-time state.

≠

Original request-time state

TEST THE BOUNDARIES THAT CAUSE DISAGREEMENT

Use fixed fixtures for duplicate events, late updates, window edges, missing values and entity mismatches.

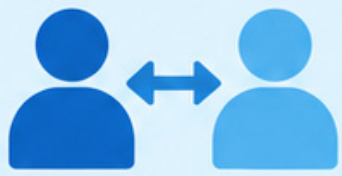
Duplicate events

Late updates

Window edges

Missing values

Entity mismatches



Historical reconstruction

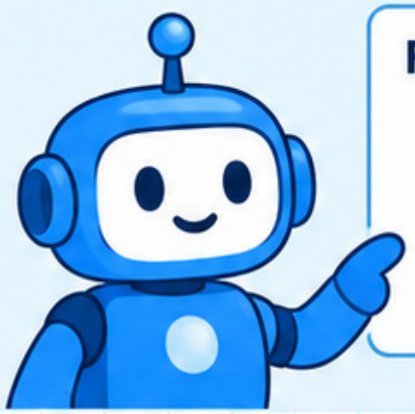
Compare historical reconstruction with the feature values captured during serving for the same contract.

Feature values during serving

Investigate differences before sharing the feature across more models.

SHARING REQUIRES OWNERSHIP AND ACCESS RULES

Name an owner for source quality, definition changes, refresh failures and consumer migration.



Feature Owner

- Source quality
- Definition changes
- Refresh failures
- Consumer migration

Restrict feature access by the product's permissions and data policy.

Consumer

Access Policy

- Access allowed
- Access restricted

A feature registry helps describe assets; it does not automatically settle every access or retention decision.

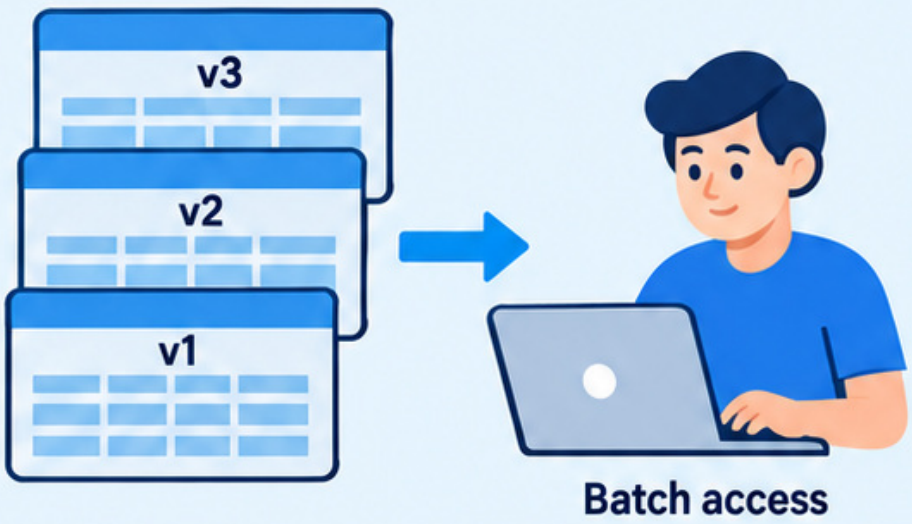
Feature Registry

- Name
- Description
- Owner
- Data policy
- Retention

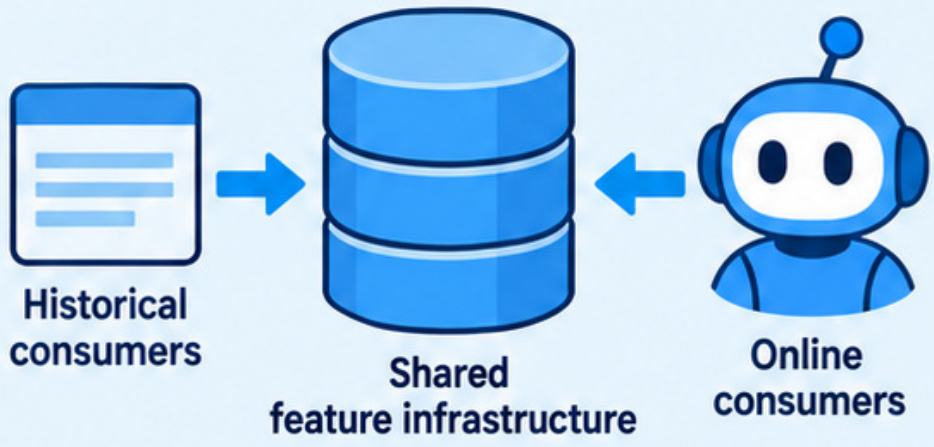
A feature registry helps describe assets; it does not automatically settle every access or retention decision.

COMPARE OPERATING COST WITH ACTUAL BENEFIT

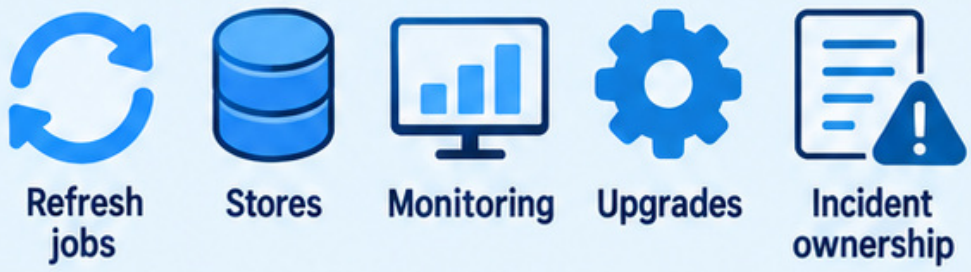
Versioned tables: fewer moving parts when batch access and ownership are simple.



Shared feature infrastructure: useful when historical and online consumers need coordinated definitions and access.



Include refresh jobs, stores, monitoring, upgrades and incident ownership in the decision.



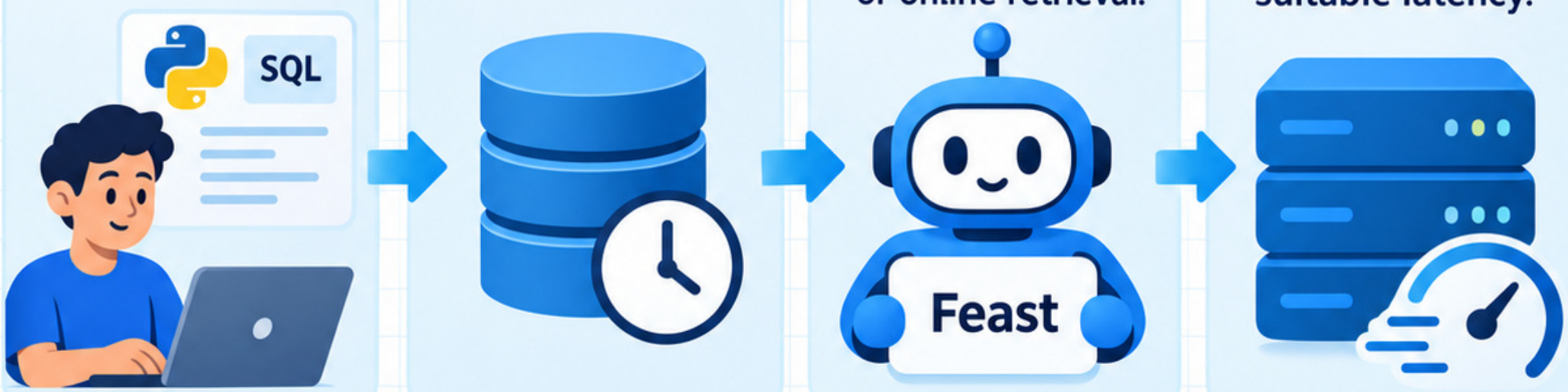
PLACE FEAST IN A COMPLETE DESIGN

SQL or Python jobs: prepare and validate features.

Historical storage: retain feature history.

Feast: register feature views and coordinate historical or online retrieval.

An online store: serve prepared values with suitable latency.

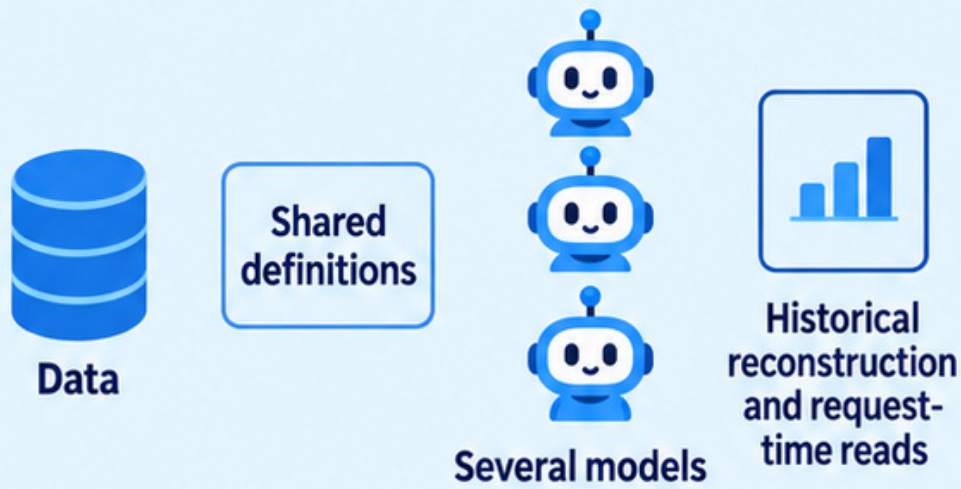


Your application owns semantic and freshness requirements.

YOUR TURN: CHOOSE THE SMALLER SUFFICIENT SYSTEM

Scenario A:
one nightly model; retained snapshots; no live feature lookup.

Scenario B:
several models; repeated shared definitions; historical reconstruction and request-time reads.



Scenario	Which design would you start with for each?	Two requirements you would verify before introducing Feast.	Owner
A			
B			

A FEATURE STORE IS A DESIGN CHOICE

Decide from reuse, history, request-time access and operational ownership.

1. Requirements

- ☑ reuse
- ☑ history
- ☑ request-time access
- ☑ operational ownership

2. Smallest sufficient architecture

Versioned tables

Choose from requirements

Shared feature infrastructure

3. Verify with concrete fixtures

- ☑ semantics
- ☑ availability
- ☑ freshness

Tool adoption does not replace those contracts.

DAY 16: Monitor drift and delayed outcomes.

Know when the data or the model's usefulness changes.