Orell Garten

FORGET THE CLOUD: BUILDING LEAN BATCH PIPELINES FROM TCP STREAMS WITH PYTHON AND DUCKDB

Orell Garten

A FEW QUESTIONS

WHO BUILDS DATA PIPELINES?

WHO BUILDS DATA PIPELINES IN THE CLOUD?

WHO PROCESSES LESS THEN 100 GB OF DATA PER DAY?

WHO AM I

- Data Engineering Consultant building pipelines for tech products
- Self-employed consultant for SMEs in tech
- 7+ years of experience with Python
- Contact:
  - LinkedIn
  - hello@orellgarten.com

INTRODUCTION

WHAT ARE WE GONNA DO TODAY?

1. Intro
2. Pipeline Design
    1. Overview
    2. Ingestion
    3. Data Validation
    4. Storage
3. Orchestration
4. Outlook

DISCLAIMER

I don't hate the cloud

😅

WHAT IS THIS TALK ABOUT

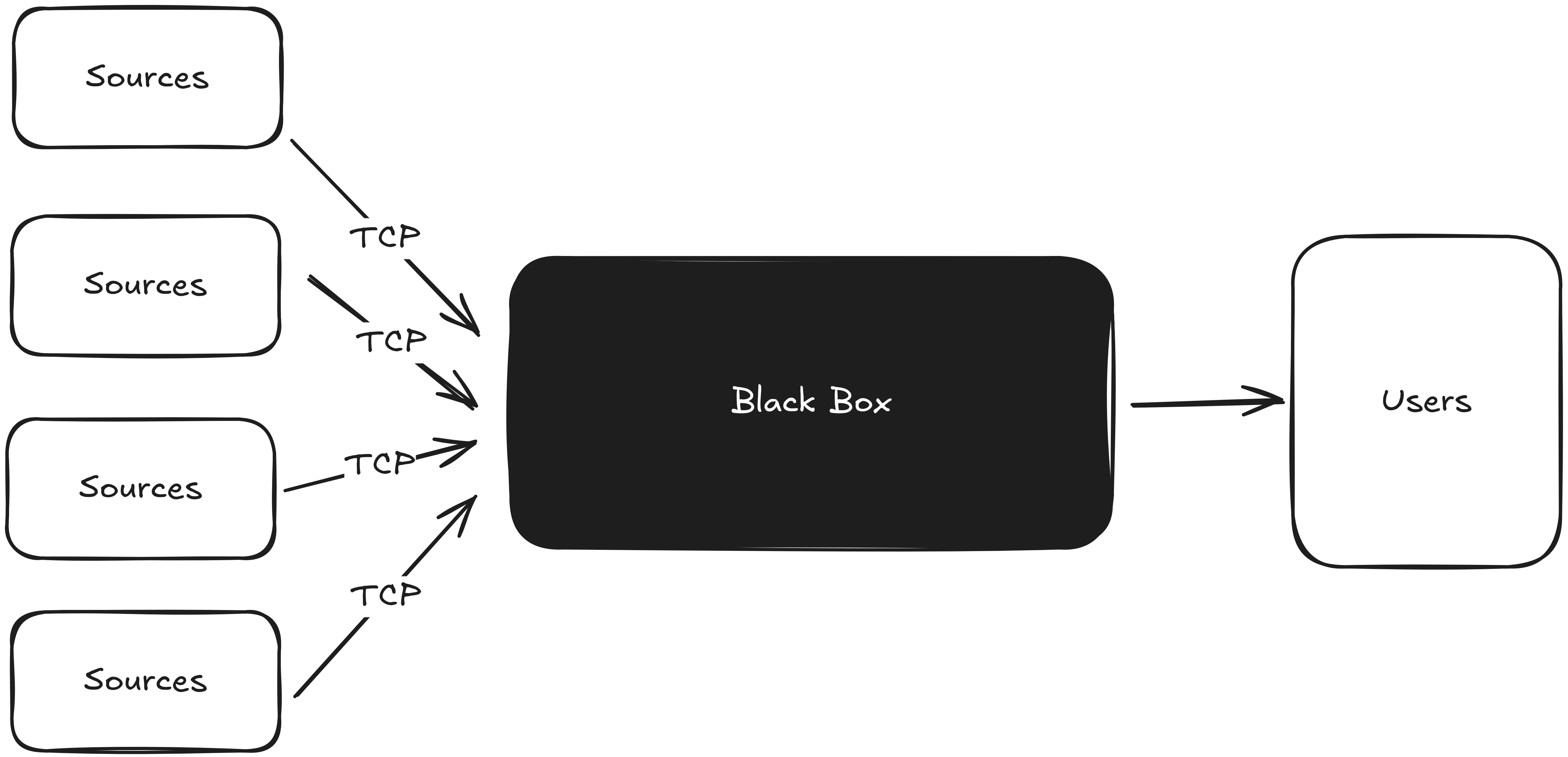
- Cloud-based solutions are often not the best solution
- A lot of data systems do not need cloud-scale.
- The cloud does not make you modern.

WHAT WILL I PRESENT?

- A pragmatic approach for a specific class of systems
  - data is provided via TCP streams
- We do not have control of the source
- Python + DuckDB for simple but effective data pipelines

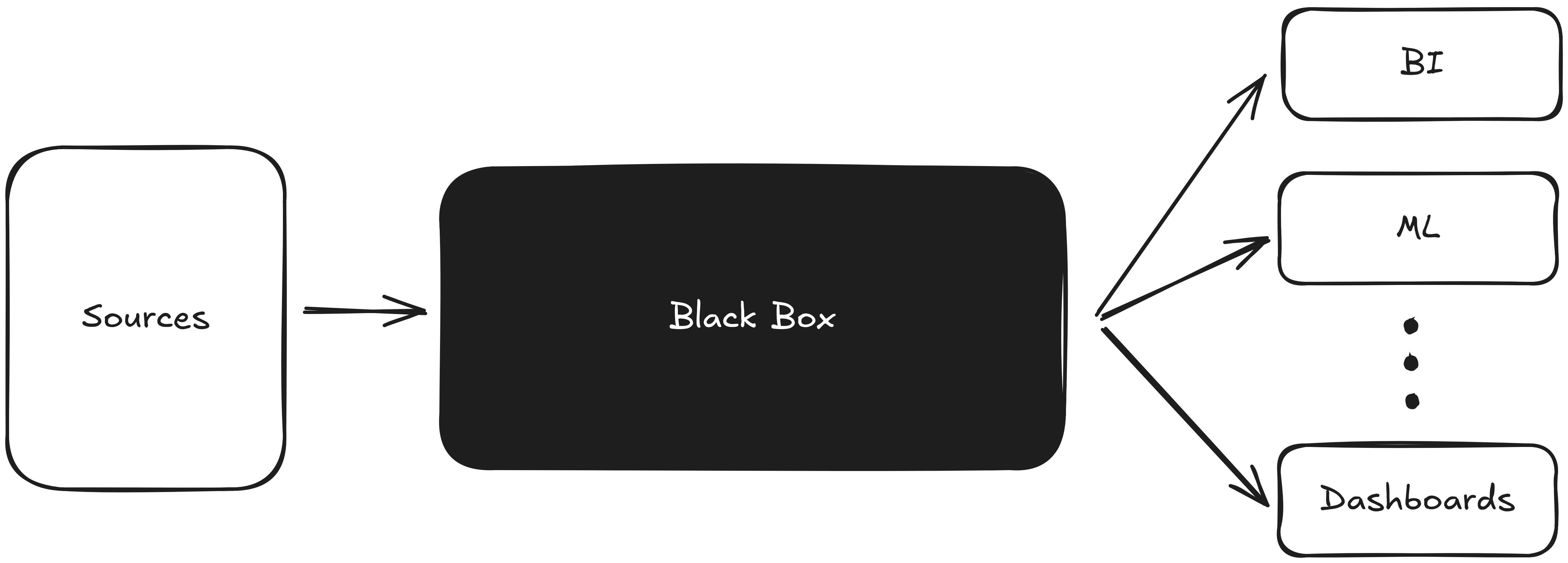
PIPELINE DESIGN

OVERVIEW

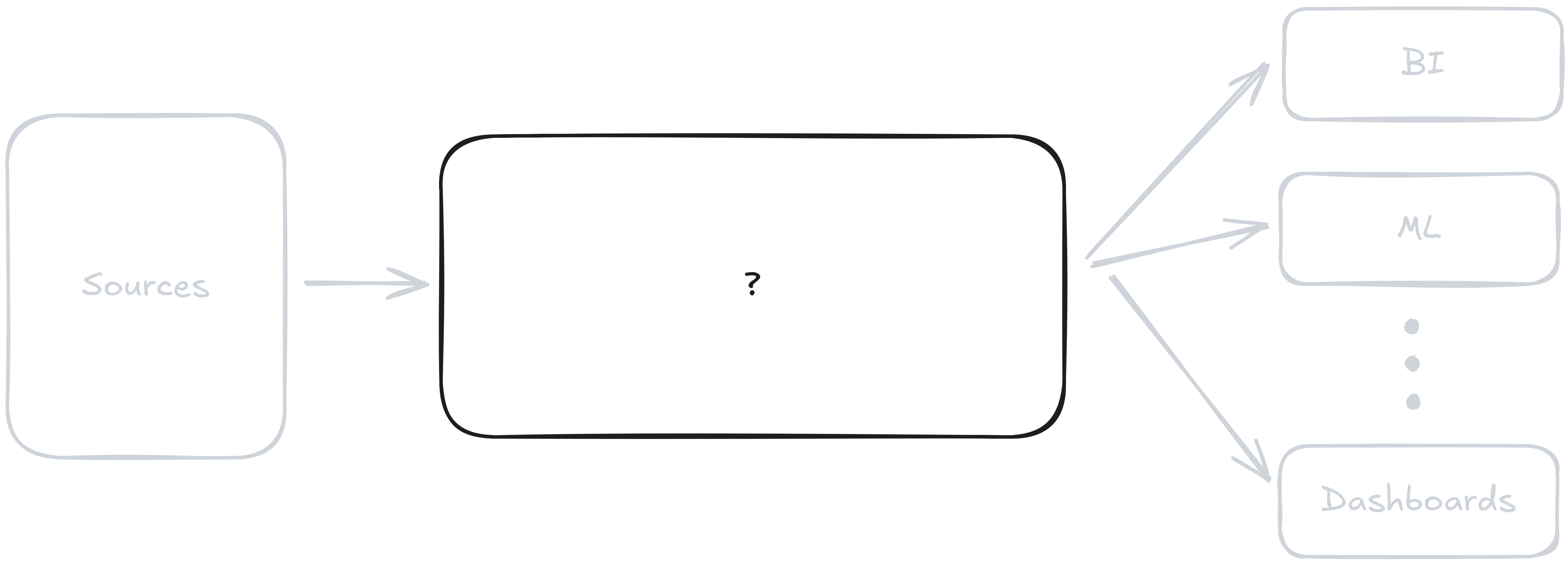


Sources
Sources
Sources
Sources
TCP
TCP
TCP
TCP
Black Box
Users

OVERVIEW - USERS

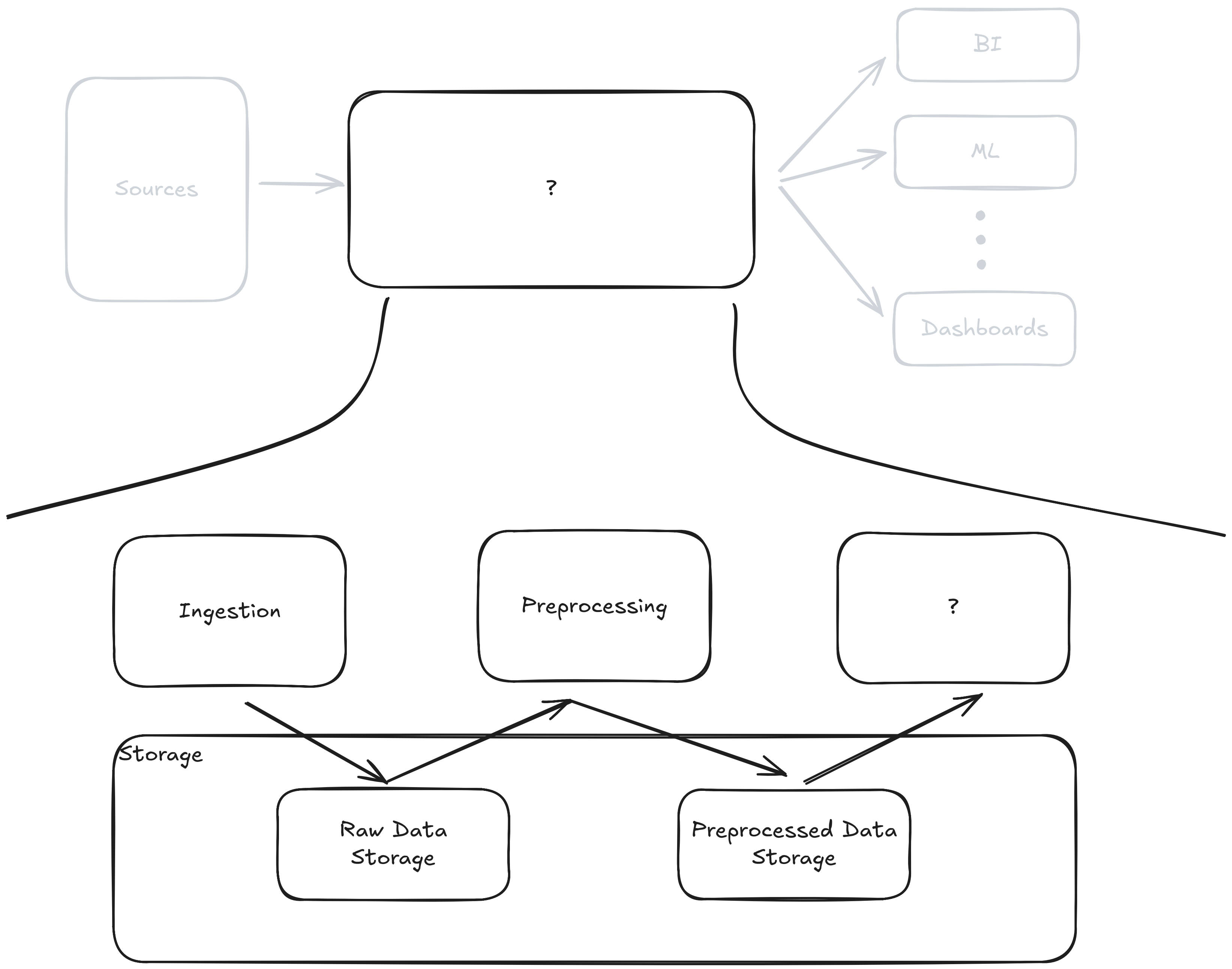


OVERVIEW - WHAT'S IN THE BLACK BOX?



🎉 OUR DATA PIPELINE 🎉

OVERVIEW - WHAT'S IN THE BLACK BOX?



PIPELINE COMPONENTS

DATA SOURCES

- TCP Streams

PIPELINE

- Ingestion
- Validation
- Storage & Data Management
- Processing

CONSUMERS

- Business Intelligence
- ML use cases
- Reporting

DATA SOURCES

- TCP Streams are still relevant:
  - Common in manufacturing, logistics, energy
  - Often on-premise and resource-constrained
  - Mission-critical downtime is costly
- Why they still matter:
  - Legacy systems built for deterministic, low-latency communication
  - Direct device-to-device data transfer without external dependency
  - Proven reliability in isolated environments

CHARACTERISTICS OF TCP STREAMS

- Continuous flow of structured or semi-structured messages
- Examples:
  - Factory sensor readings
  - Machine operation logs
  - Legacy telemetry from field equipment

REQUIREMENTS

- Convert raw streams into batch datasets for analytics
  - batch processing is easier to handle and is good enough in most cases
- Should not drop data
  - no persistence in TCP
- Challenges:
  - No inherent replay or persistence
  - Variable message formats
  - Requires custom ingestion before batch processing
  - We do not control the source(!)

INGESTION

INGESTION

We want to ingest data from TCP Streams.

WHAT DOES OUR DATA LOOK LIKE?

DATA MODELING

General challenge:

We receive bytes and need to do something with them.

We do not have control over data generation.

🎉 **WE CAN CONTROL THE INGESTION!** 🎉

DATA MODELING

- There are many different data formats:
  - JSON
  - byte strings
  - proprietary formats
  - ...
- We need to understand and model the data that we receive!

DATA MODELING WITH PYDANTIC

- Pydantic's `BaseModel` allows us to build intuitive data models:

```python
from enum import StrEnum
from pydantic import BaseModel


class MeasurementUnit(StrEnum):
  CELCIUS = "Celcius"
  FAHRENHEIT = "Fahrenheit"

class MeasurementKind(StrEnum):
  TEMPERATURE = "Temperature"

class Measurement(BaseModel):
  kind: MeasurementKind
  value: float | str
  unit: MeasurementUnit

class TelemetryData(BaseModel):
  timestamp: int
  sensor_id: int
  measurement: Measurement
```

DATA MODELING ⟶ DATA VALIDATION

- Data modeling is the first step towards data validation
- Data modeling is useful for two reasons:
  - To think about your systems
  - To perform data validation

But why do we need data validation?

- Without data validation no data quality
- Data validation helps spot systemic issues

DATA MODELING WITH PYDANTIC

- We can validate JSON data against this model by using `validate_json`:

```python
from pydantic import TypeAdapter
import json

validator = TypeAdapter(Telemetry)

[...]

data = json.dumps(
    {
        "timestamp": 1756803600,
        "sensor_id": 10001,
        "measurement": {
            "kind": "Temperature",
            "value": 5.3,
            "unit": "Celcius"
        }
    }
)
validator.validate_json(data, strict=True)
```

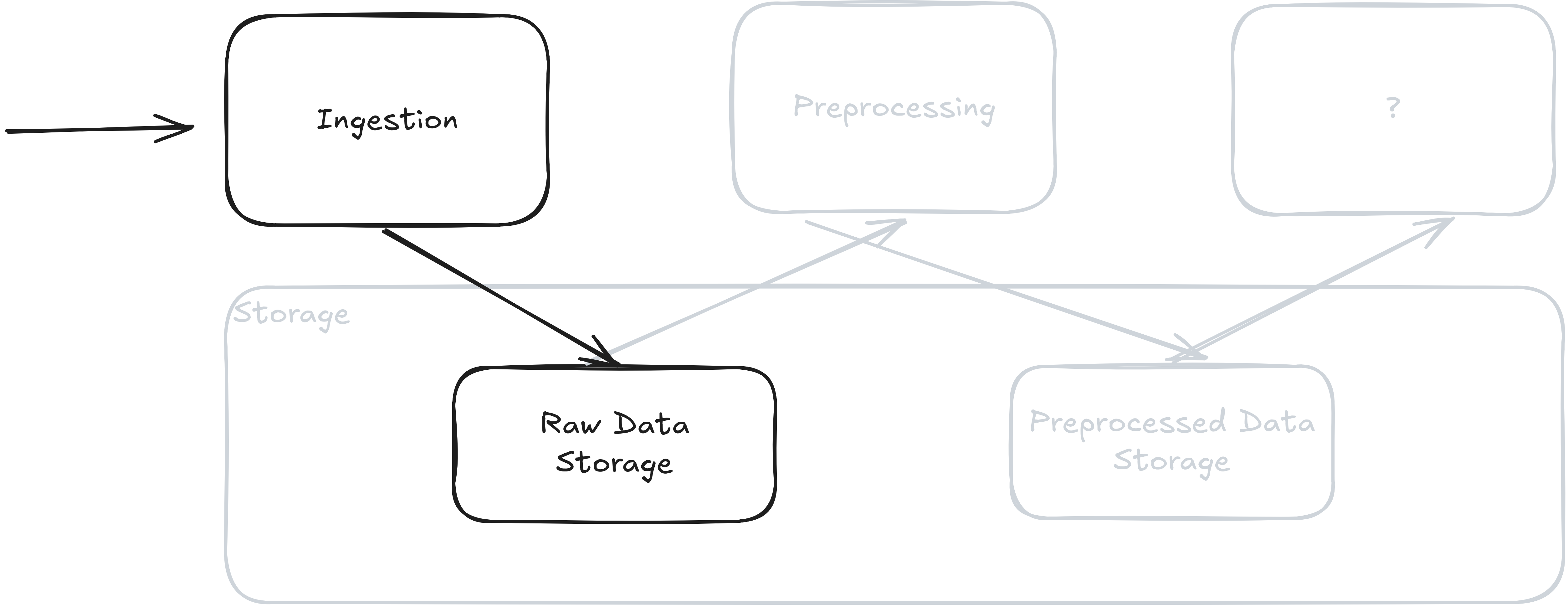
- Output:

```
TelemetryData(
    timestamp=1756803600,
    sensor_id=10001,
    measurement=Measurement(
        kind=<MeasurementKind.TEMPERATURE: 'Temperature'>,
        value=5.3,
        unit=<MeasurementUnit.CELCIUS: 'Celcius'>
    )
)
```

INGESTION

- Our ingestion should do two things:
  - persist incoming data
  - validate data against our models

INGESTION



Two general options:

1. Write before validate
2. Validate before write

INGESTION

We have TCP Streams. What do we need for the ingestion?

🎉 **A TCP SERVER** 🎉

THE ACTUAL INGESTION

Our ingestion is "just" running a TCP server

To sync or to async?

Let's do async here.

WHY ASYNC?

🏎️💨

WHY ASYNC?

I/O is great for async.

WHY ASYNC?

- Our ingestion has I/O:
  - network
  - storage

MORE TALKS ABOUT `async`

- Bojan Miletic: Mastering Asynchronous Python in FastAPI
  - PyCon DE & PyData Berlin 2024
- Miguel Grinberg: Asynchronous Python for the Complete Beginner
  - PyCon 2017
- Niels Denissen: A practical guide to speed up your application with Asyncio
  - PyData Amsterdam 2017

✅ ASYNC VS SYNC

INGESTION

```python
server = TCPServer(
    host=host,
    port=port,
    data_source_type=data_source_type,
)

await server.serve()
```

```python
class TCPServer:

    [...]

    async def serve(self) -> None:
        server = await asyncio.start_server(self.handle_connection, self.host, self.port)

        async with server:
            await server.serve_forever()
```

```python
class TCPServer:
    BUFFER_SIZE: int = 1000

    def __init__(self, host: str, port: int):
        self.buffers = defaultdict(list)
        self.host = host
        self.port = port
        self.data_source_type = data_source_type

        self.writer = AsyncBatchWriter(self.data_path, settings.buffer_size)

        match data_source_type:
            case DataSourceType.TelemetryData:
                self.validator = TypeAdapter(TelemetryData)
            case DataSourceType.Measurement:
                self.validator = TypeAdapter(Measurement)
            case ...
```

```python
async def handle_connection(self, reader, writer):
    while not reader.at_eof():
        data = await reader.readline()

        if data == b"":
            continue

        try:
            jdata = self.validator.validate_json(data)
            timestamp = jdata.timestamp

            await self.writer.add_to_batch(timestamp, jdata.model_dump_json() + "\n")

        except pydantic.ValidationError as e:
            self.logger.error(f"Skipping due to validation error: {e.error()}")
            continue

        [...]

    writer.close()
    await writer.wait_closed()
```

This is where you do all your data logic.

TAKEAWAYS

- Data modeling is important
- Data validation builds on top of data modeling
- For JSON data pydantic is a good choice
- Validation before writing is best (imho)
- Consider `async` to get the most out of your system

STORAGE

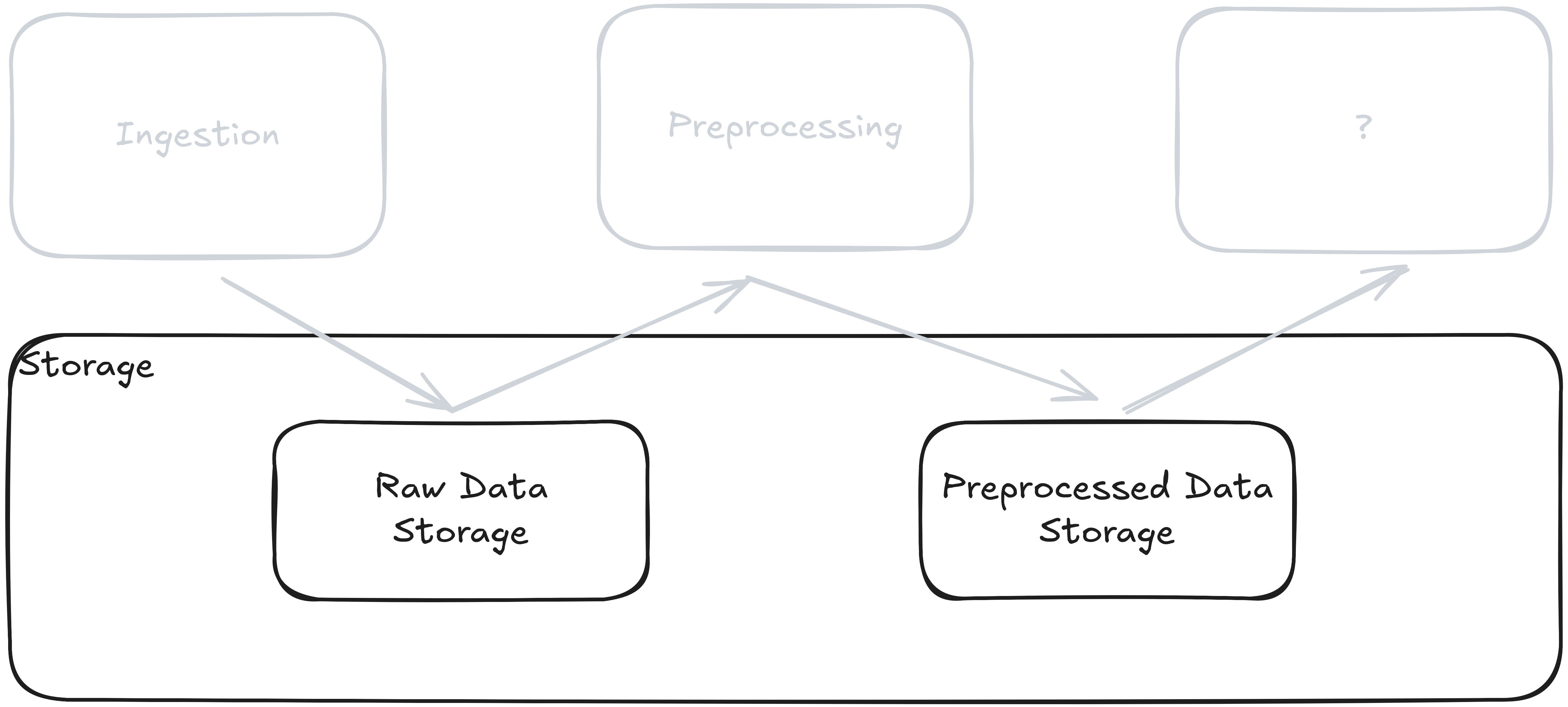
```
await self.writer.add_to_batch(timestamp, jdata.model_dump_json() + "\n")
```

STORAGE

```python
await self.writer.add_to_batch(timestamp, jdata.model_dump_json() + "\n")
```

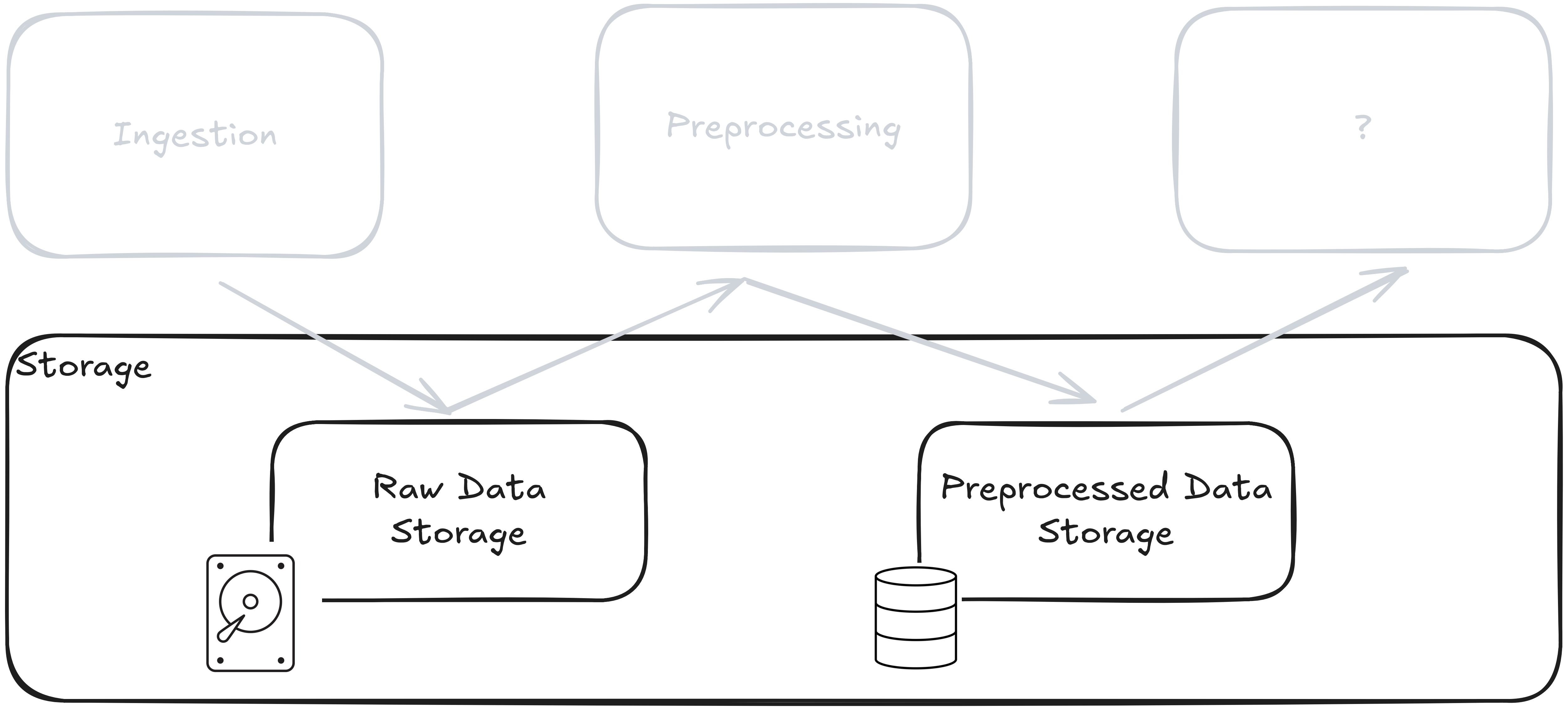
- `writer` keeps track of buffered data
- when enough data is ready, asynchronously write to disk

STORAGE



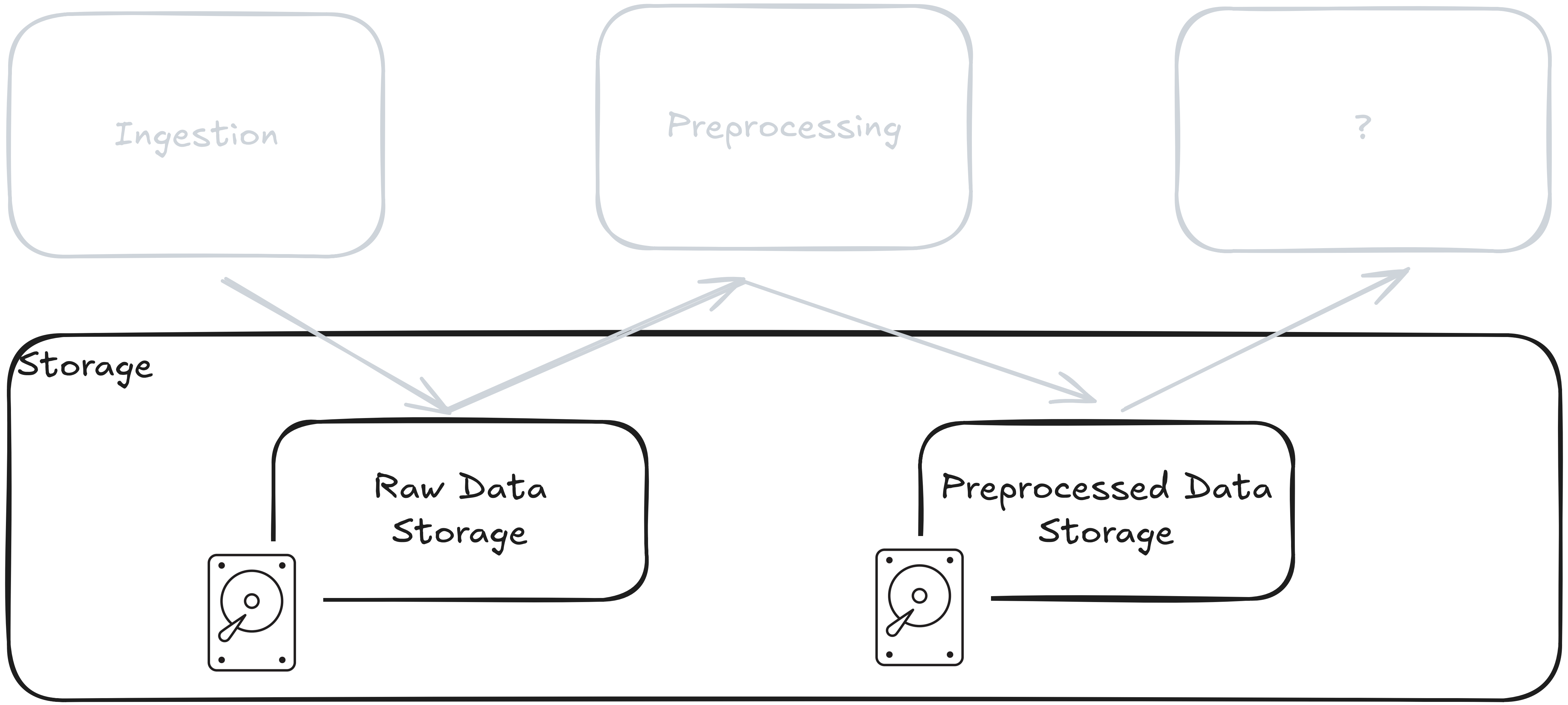
Things to consider for storage?

STORAGE



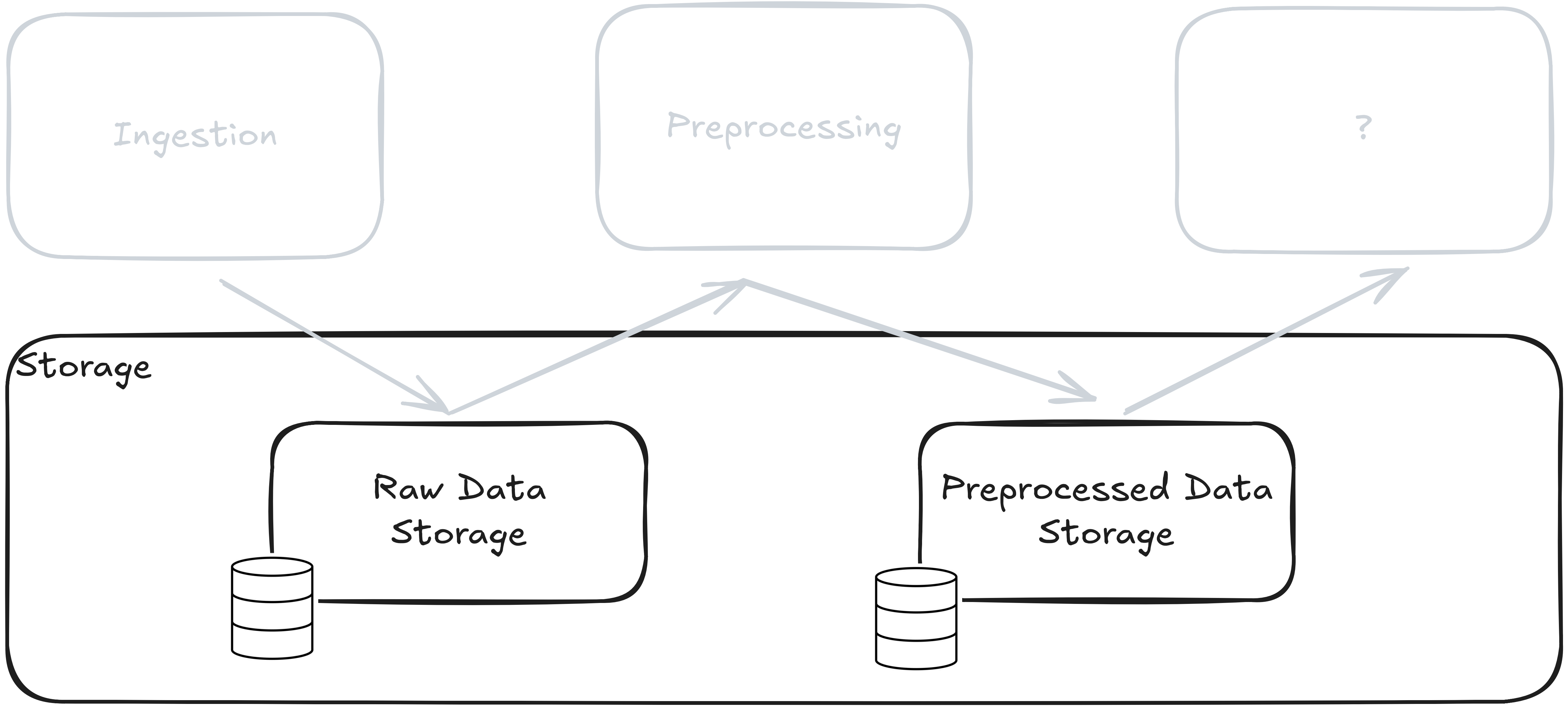
Things to consider for storage?

STORAGE



Things to consider for storage?

STORAGE



Things to consider for storage?

STORAGE OPTIONS

- local Filesystem
- S3, blob storage
- Database
- use `fsspec` for more flexiblity as you scale
    - Einat Orr, Barak Amar: *Distributed file-systems made easy with Python's fsspec* @ PyCon & PyData DE 2025

FILESYSTEM

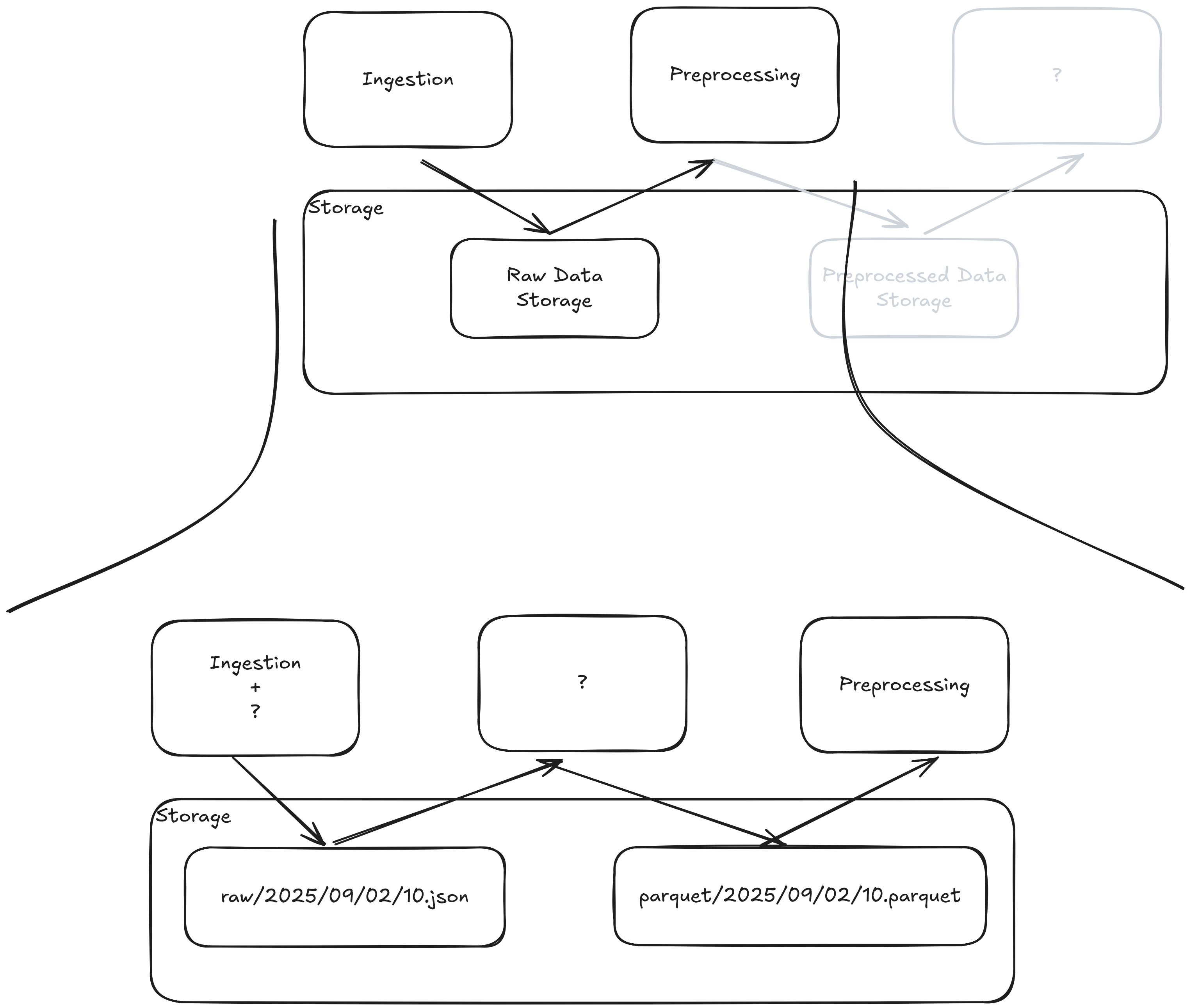
- Keep it simple.
- Filesystem first, than go from there

Advantage?

- low latency, because no networking
- no extra costs
- no data transfer

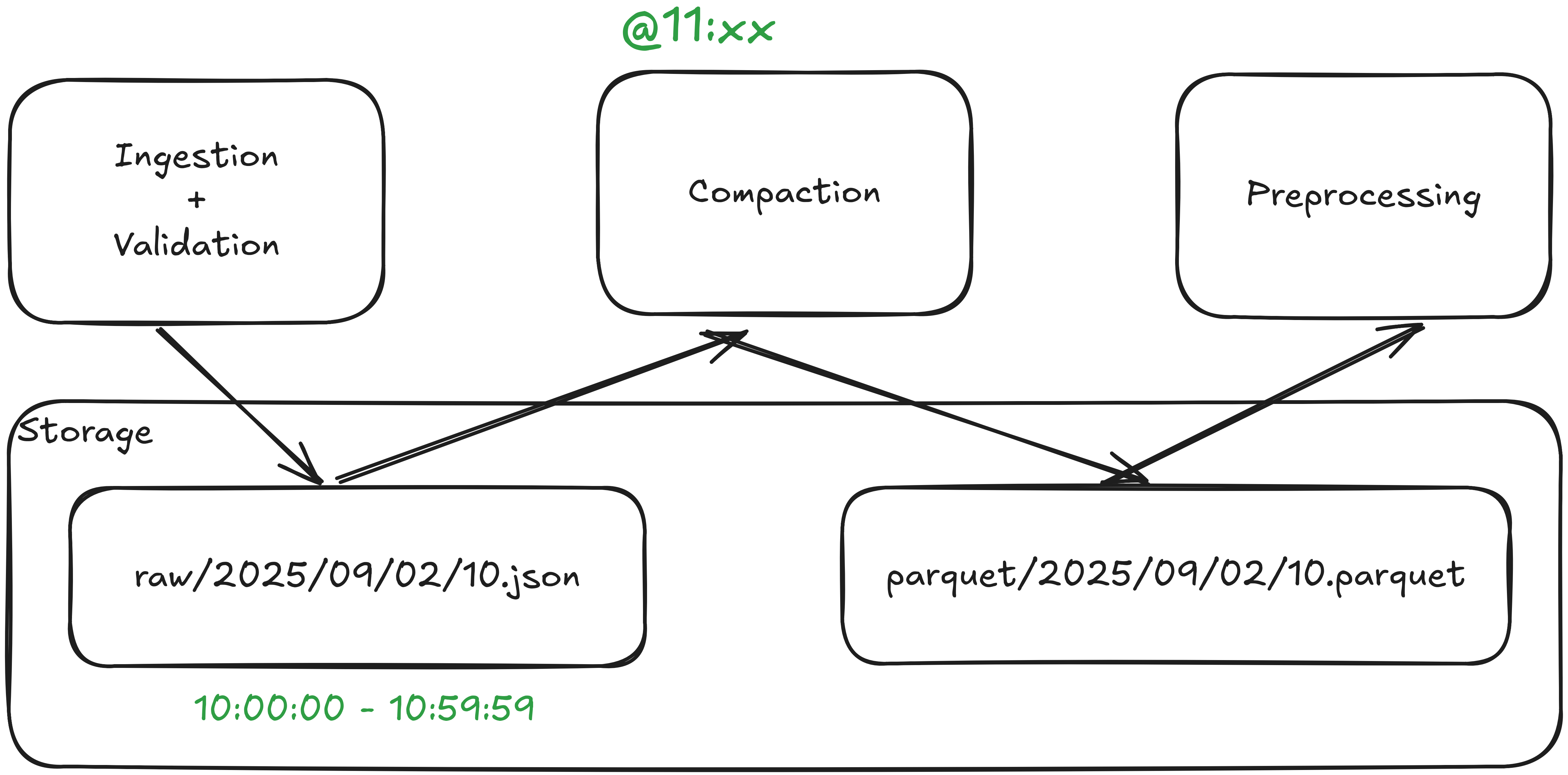
BUT very limited.

STORAGE PIPELINE

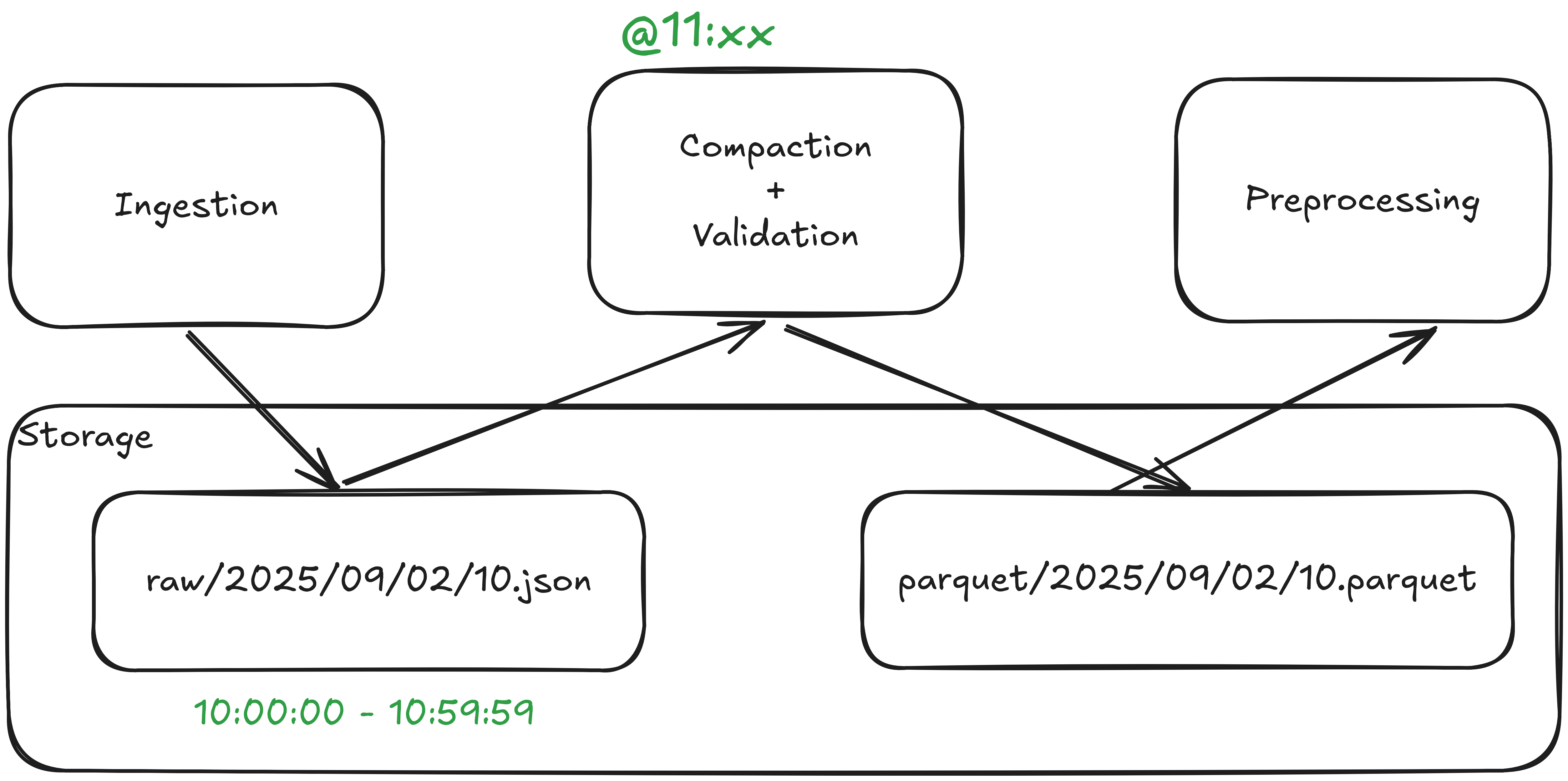


STORAGE & VALIDATION?

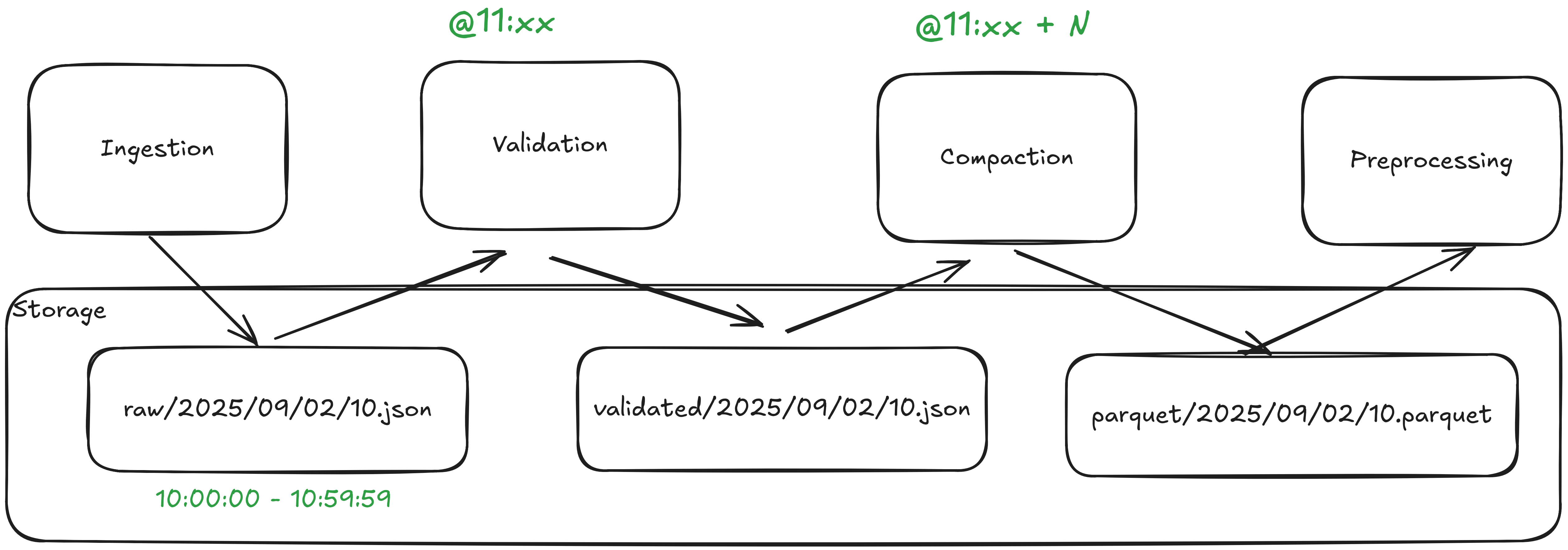
STORAGE & VALIDATION PIPELINE I



STORAGE P& VALIDATION PIPELINE II



STORAGE & VALIDATION PIPELINE II



TAKEAWAYS

Keep it local first.

Shift left if possible.

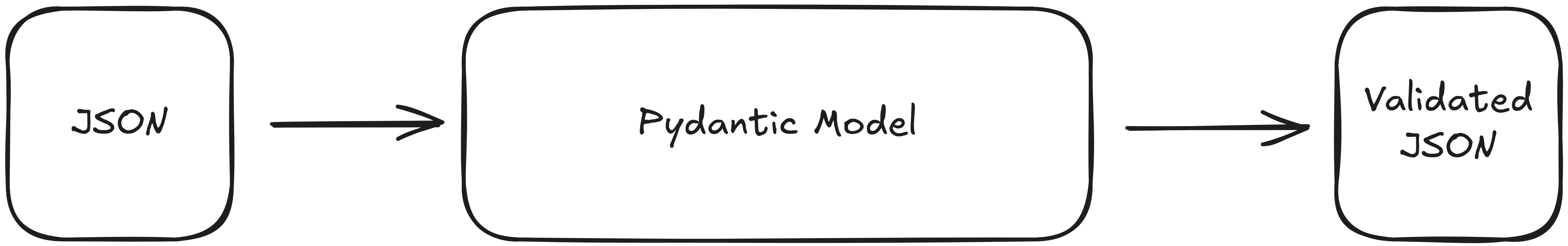
Fail early.

Be as strict as necessary.

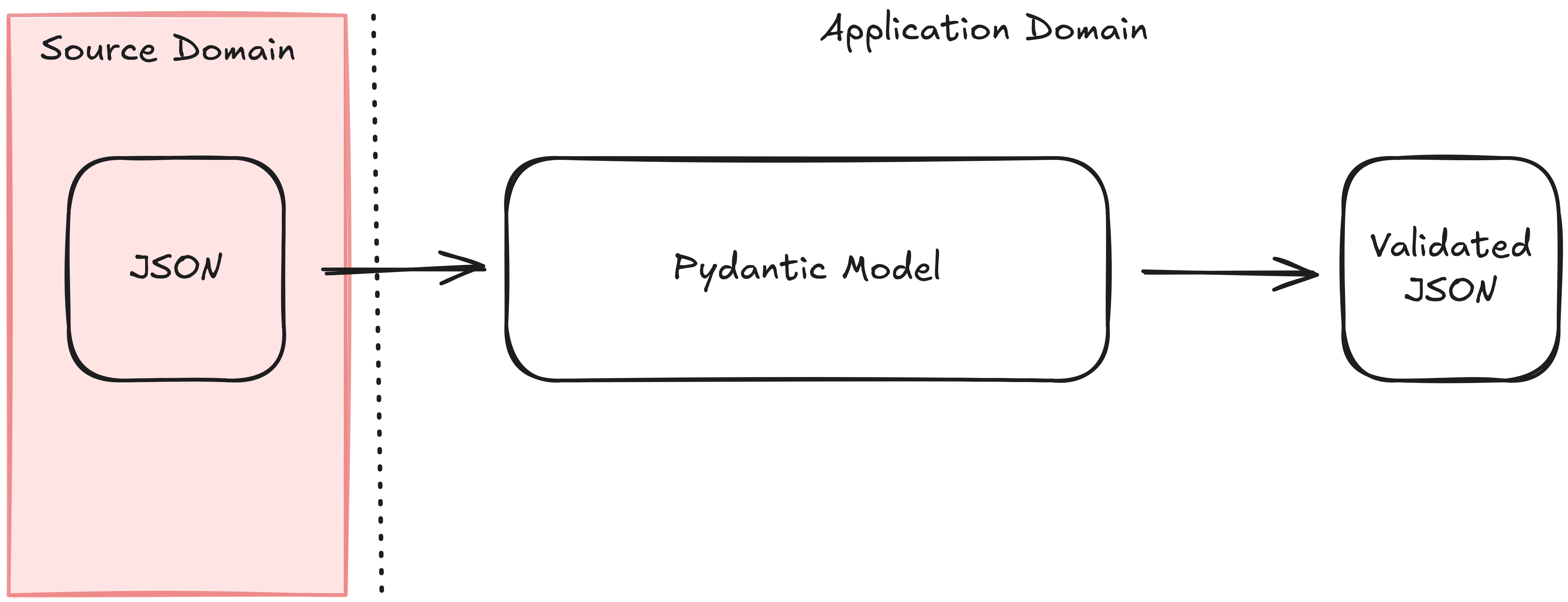
INGESTION REVISITED

```python
async def handle_connection(self, reader, writer):
    while not reader.at_eof():
        data = await reader.readline()

        if data == b"":
            continue

        try:
            jdata = self.validator.validate_json(data)
            timestamp = jdata.timestamp

            await self.writer.add_to_batch(timestamp, jdata.model_dump_json() + "\n")


        except pydantic.ValidationError as e:
            self.logger.error(f"Skipping due to validation error: {e.error()}")
            continue

        [...]

    writer.close()
    await writer.wait_closed()
```

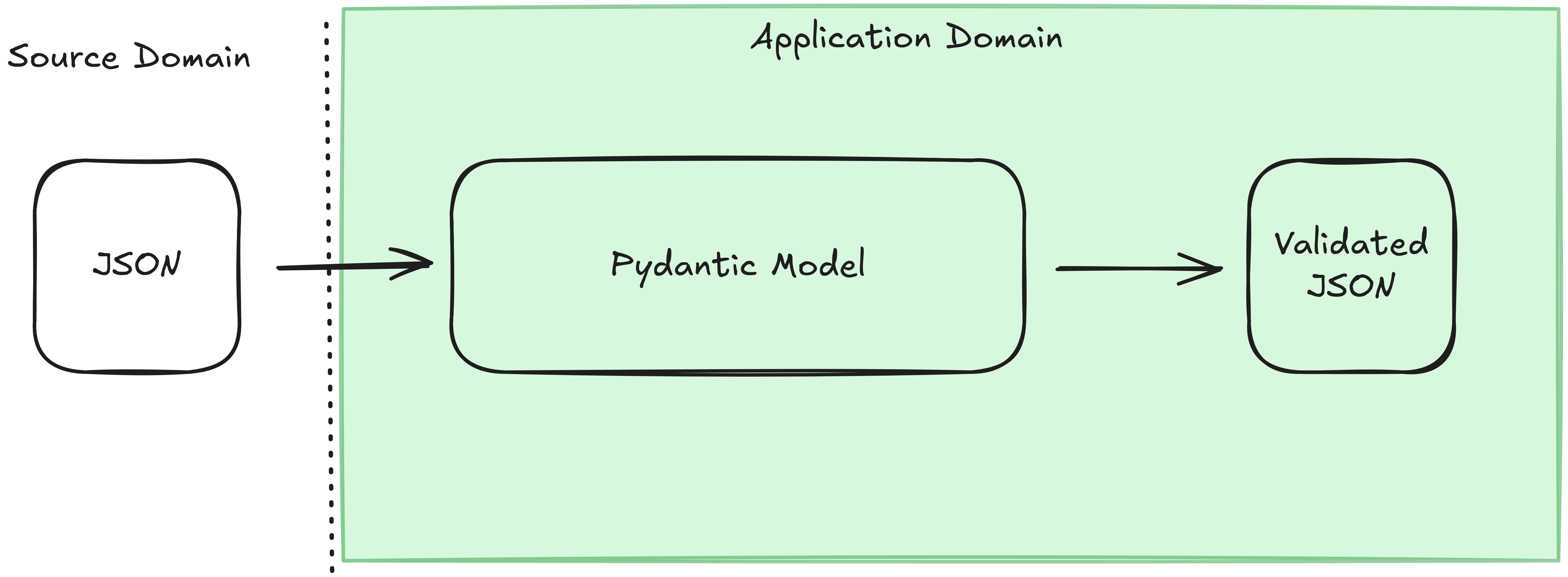
DATA LIFECYCLE



DATA LIFECYCLE



DATA LIFECYCLE



SO FAR SO GOOD?

Weeeeell ...

WE STILL HAVE JSON

🚫 WE DON'T WANT JSON

DuckDB to the rescue

WHO HAS USED DUCKDB BEFORE?

WHAT IS DUCKDB

- in-memory OLAP database
- "SQLite for analytics"
- two main use cases:
    - transformation engine
    - query engine

COMPACTION: JSON TO PARQUET

- Goals:
  - reduce storage requirements
  - better format for compute
  - increased interoperablity

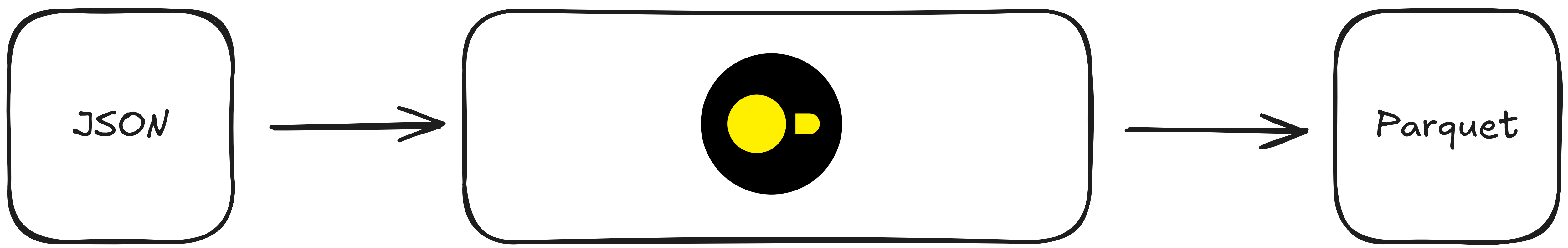
COMPACTION: JSON TO PARQUET

- DuckDB handles compaction for us
- 3GB of JSON to 300MB parquet:
  - 30s on a regular laptop
  - few seconds on server
- Note: Performance depends on the nature of your data

EXCEUTING SQL STATEMENTS WITH DUCKDB

```
duckdb.sql(
  """
  <SQL>
  """
)
```

- I will omit the duckDB call for better syntax highlighting for the rest of the talk


JSON
Parquet

READING DATA INTO A TEMPORARY TABLE

```
CREATE OR REPLACE TEMPORARY TABLE validated_data AS (
  SELECT
      *
  FROM
      read_json('raw/2025/09/02/10.json', format='nd');
)
```

WRITING DATA TO PARQUET

```
COPY (
    SELECT
        *
    FROM validated_data
)
TO 'parquet/2025/09/02/10.parquet' (FORMAT parquet, COMPRESSION zstd);
```

ALL AT ONCE

```sql
COPY (
    SELECT
        *
    FROM read_json('raw/2025/09/02/10.json', format='nd')
)
TO 'parquet/2025/09/02/10.parquet' (FORMAT parquet, COMPRESSION zstd);
```

PROBLEMS?

We still have our JSON files laying around.

😩

DELETING OLD FILES

- We need to delete JSON files that are not needed anymore
- We'll just use plain python:

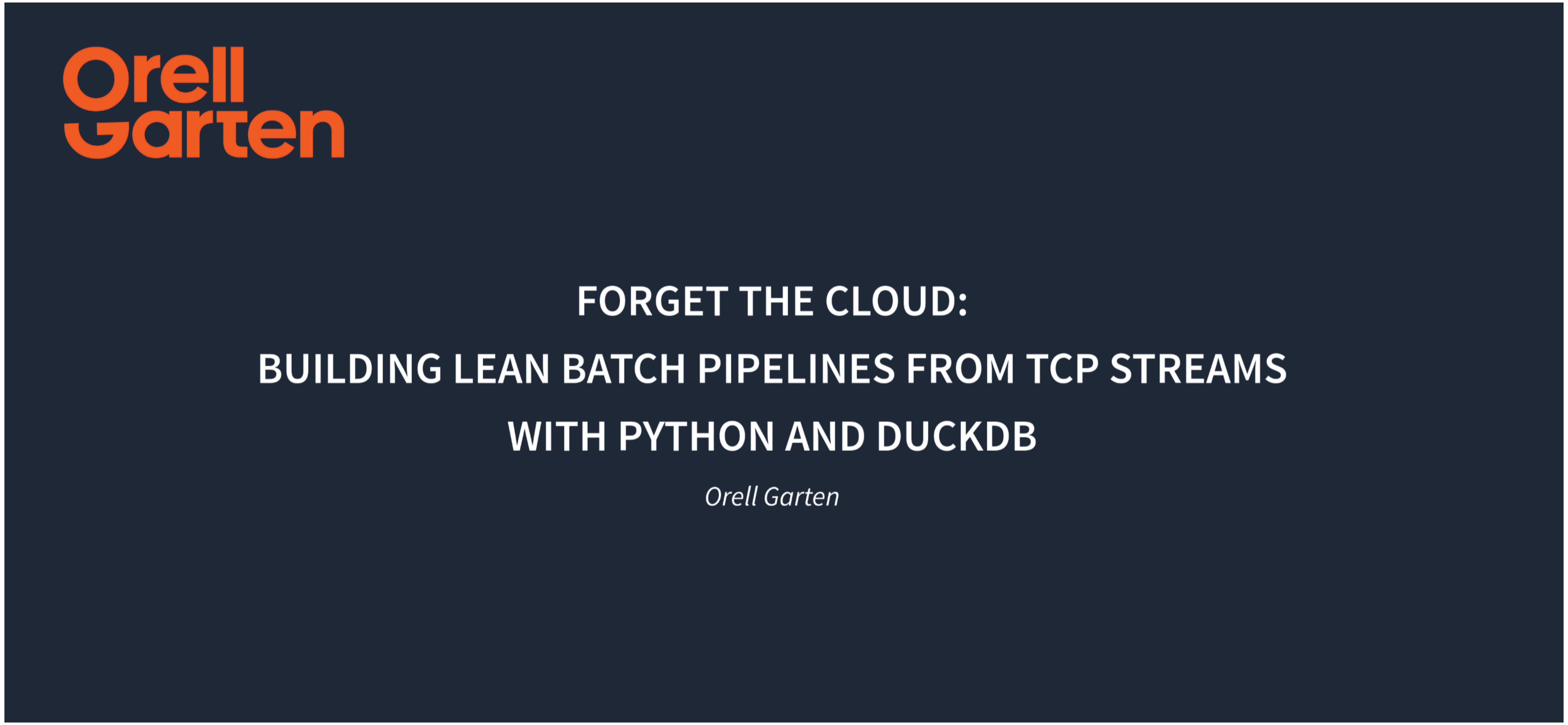
```
(source_path / f"{hour}.json").unlink()
```

HMMM...

Mom, can we have deltalake at home?

We have deltalake at home.

Deltalake at home:



WHY NOT JUST USE DELTALAKE?

- provided by `delta-rs`
- Python bindings only reached v1 end of May
- required stable data without too many problems
- unrealistic if you do not control data generation
- duckDB has partial support for deltalake, mainly read ops

ANALYTICS

DATA PREPARATION

- data likely needs to prepared for downstream analytics
- we use duckDB as query engine
- we can take our parquet files and run queries against them 🎉

DUCKDB FOR ANALYTICS

READ DATA FROM PARQUET FILE

```
CREATE OR REPLACE TABLE telemetry AS (
    SELECT
        *
    FROM
        read_parquet('telemetry/raw/../10.parquet')
)
```

- Quite similar to `read_json`, but much quicker
- Also possible:

```
read_parquet('telemetry/raw/../*.parquet')
```

PROCESS AND ENRICH DATA

```sql
CREATE OR REPLACE TABLE temperatures AS (
    SELECT
        date_trunc('hour', TIMESTAMP '{start_time}') as start_time,
        date_trunc('hour', TIMESTAMP '{end_time}') as end_time,
        s.id as sensor_id
        AVG(t.temperatur) as avg_temp
    FROM telemetry t
    JOIN sensors s ON t.sensor_id = s.id
    WHERE t.measurement_kind = 'temperature'
    GROUP BY s.id
)
```

- Enrichtment with time information
- Aggregation function `AVG` to reduce information
- Supports all common operations
- DuckDB is absolutely great if you know SQL!

WHAT ARE WE GONNA DO WITH THAT DATA?

🤔

TO THE WAREHOUSE

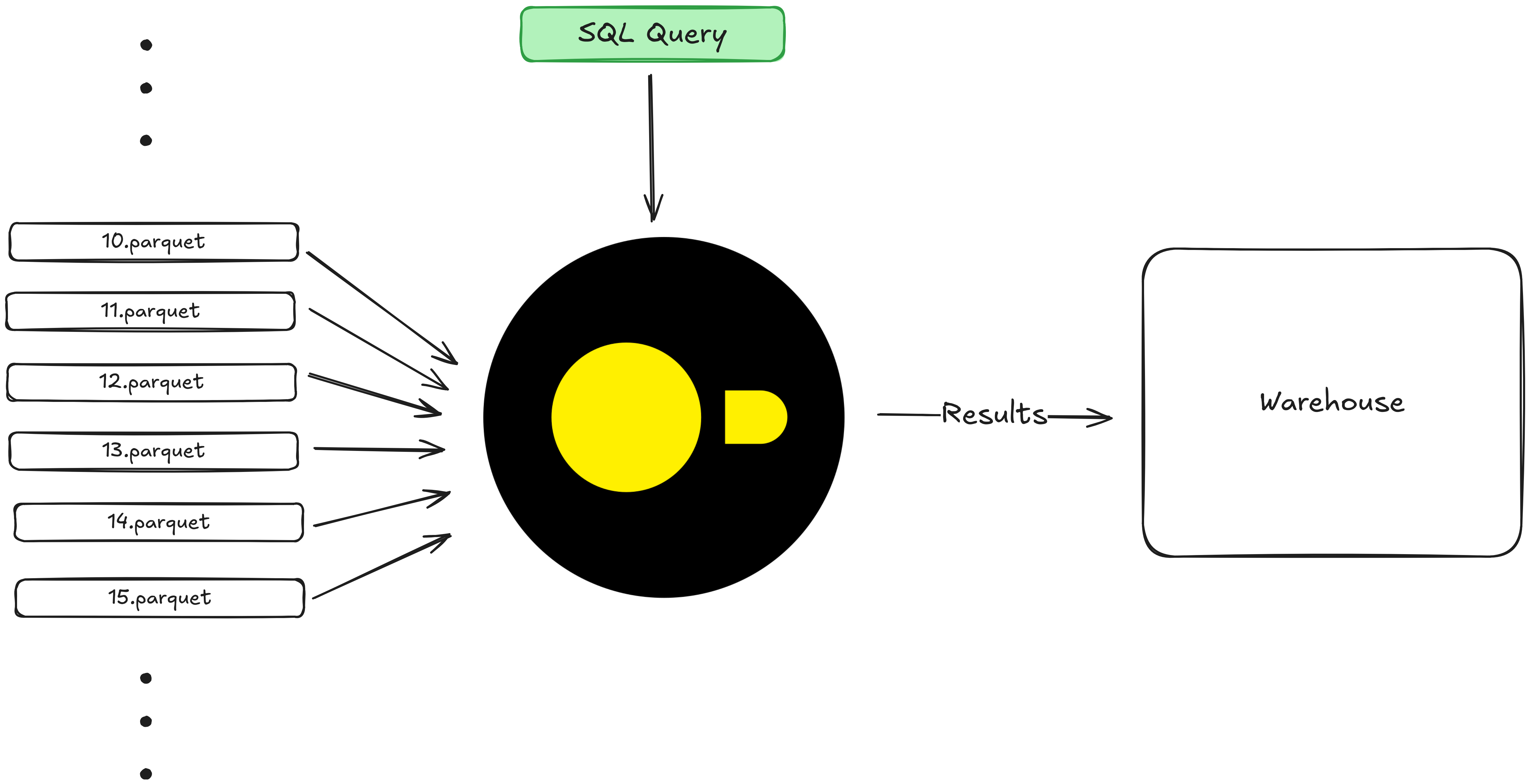
- Attach the target database
- Connection information stored in environment and read via Pydantic settings

```
ATTACH 'host={settings.db_host}
        user={settings.db_user}
        port={settings.db_port}
        password={settings.db_password}
        database={settings.db_database}'
AS target_db (TYPE mysql);
USE target_db;
"""
```

LOAD INTO WAREHOUSE

- take the processed data from `memory.temperatures`
- create an `id` based on existing entries
- `INSERT INTO` to load into database

```
con.execute(
    """
    INSERT INTO target_db.reports.fTemperatures BY NAME (
        SELECT (
            SELECT COALESCE(MAX(id), 0) FROM target_db.reports.fTemperatures
        ) + row_number() OVER () as id,
            start_time,
            end_time,
            sensor_id
            avg_temp
        FROM memory.temperatures
    )
    """
)
```


SQL Query
10.parquet
11.parquet
12.parquet
13.parquet
14.parquet
15.parquet
Results
Warehouse

TAKEAWAYS

- DuckDB is a great tool for running SQL queries against a set of parquet files
- DuckDB can handle a lot of data, e.g. through out-ofcore querying
- Attach to existing databases

ORCHESTRATION

Options?

Airflow

Dagster

Prefect

...

HOW ABOUT THIS?

```
$ crontab -l
10 * * * * compaction.sh
```

Syntax:

```
*    *    *    *    *   /home/user/bin/somecommand.sh
|    |    |    |    |             |
|    |    |    |    |    Command or Script to execute
|    |    |    |    |
|    |    |    |  Day of week(0-6 | Sun-Sat)
|    |    |    |
|    |    |  Month(1-12)
|    |    |
|    |  Day of Month(1-31)
|    |
|   Hour(0-23)
|
Min(0-59)
```

CRON JOBS FOR ORCHESTRATION

- very low complexity for straight-forward tasks
- supported by all unix systems
- many orchestration tools provide a similar mechanism

Problems?

- purely time-based
- does not model data dependencies

WHAT WE HAVEN'T TALKED ABOUT

HOUSEKEEPING ON-PREM

- Deleting old parquet files
- Deleting processed files
- Moving data to cold storage
- ...

Downside of this approach is that it requires more work keeping the system clean.

DEVOPS

- Containerization
- Container orchestration
- Deployment
- Observability
- ...

ACTUAL ANALYTICS

CONCLUSION

KEY TAKEAWAYS

- Key components of a data pipeline for TCP Stream data was presented
- Stream to batch happens in Python
- DuckDB for transformations AND queries
- Interoberability through availability of connectors to other databases

THANK YOU

Q&A

Connect with me on LinkedIn ☺️