Dropzone Challenge

Data Engineering Pipeline

Presented by: Dhouha HMEM

Content

Project Overview

Problem Statement

What problem are we solving?

- Incoming JSON events arrive continuously (~10 req/sec)
- No guaranteed immediate persistence → risk of data loss
- Inconsistent timestamps & formats
- No automated Data Quality validation

Goal:

- Build a reliable, configurable pipeline
- Transform raw API data to analytics-ready results

Project Objectives

What the pipeline must achieve

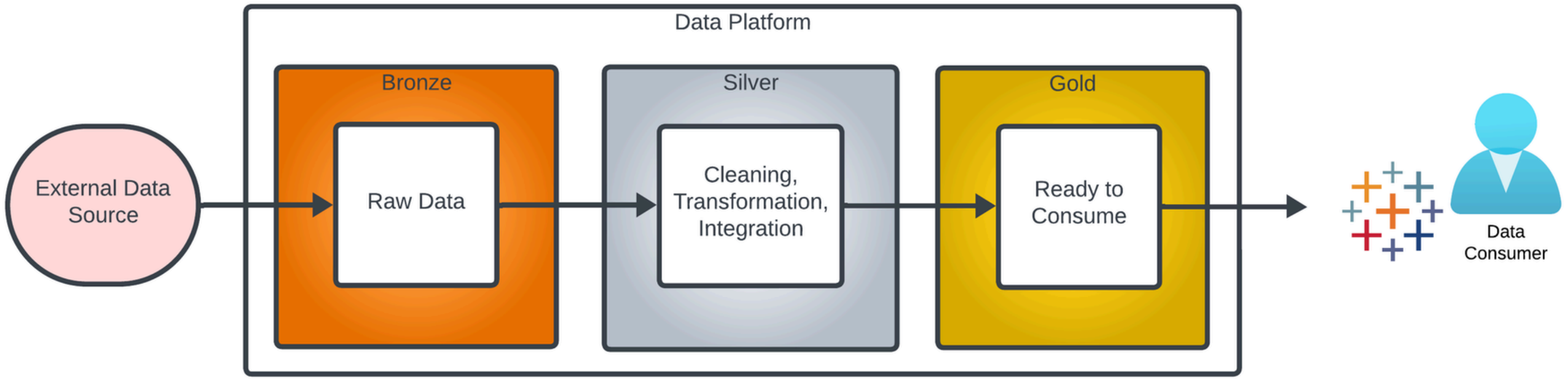
- Reliable ingestion layer
- Immediate raw storage
- Data validation & transformation
- Standardized timestamps
- Produce aggregated metrics
- Persistent structured storage

Project Architecture

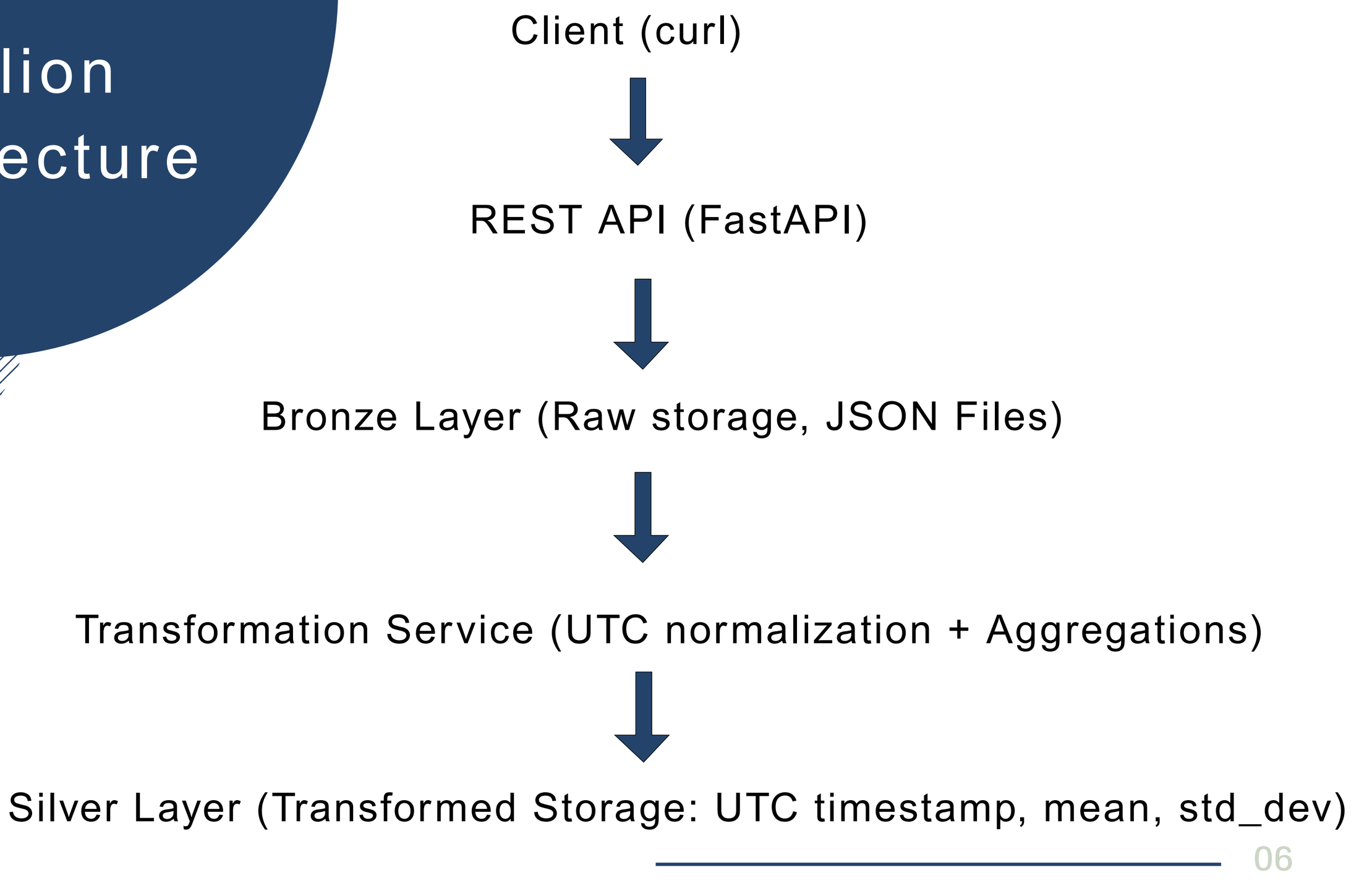
Medallion Architecture

The pipeline follows the Medallion Architecture:

- Bronze: raw ingested data (JSON)
- Silver: clean, typed, normalized, time-aligned data
- Gold: would expose business KPIs (not implemented yet)



Medallion Architecture



Project Execution

Technologies Used

- Visual Studio Code : project organization
- Python : orchestration & logic
- FastAPI : REST API framework
- PostgreSQL: database for storing transformed
- Requests : API communication
- pytest: unit and integration testing

Data source

What is the data source?

External clients sending measurement data via a REST API.

Could be: IoT devices, Applications, External systems, Scripts (cURL, batch jobs)

- Each request contains a single JSON payload with time_stamp and data field

```
{
  "time_stamp": "2019-05-01T06:00:00-04:00",
  "data": [0, 2]
}
```

Step 1: Data Ingestion

- Purpose: Safely accept incoming measurement data and acknowledge receipt immediately, without blocking processing.

Client / System
↓
POST /ingest
↓
FastAPI, REST API

Input Contract:

- Endpoint: POST /ingest
- Payload (JSON):
  -time_stamp → ISO-8601 string with timezone
  -data → array of numeric values
- Validation: Pydantic schemas

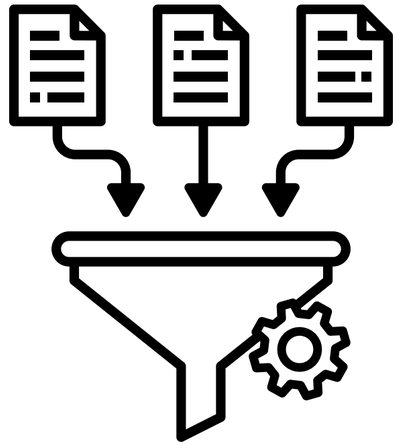
API Response:

- HTTP 202 Accepted to acknowledge receipt
- returns a reference to the stored raw payload
- Confirms ingestion without waiting for processing

Step 1: Data Ingestion

Why this design?

- Fast & lightweight → handles ~10 req/sec easily
- Stateless API → easy to scale horizontally
- Easy integration with many systems
- Decouples ingestion from processing
- Validation happens early (Pydantic)
- Clear contract via OpenAPI / Swagger



Step 2: Bronze Layer (Raw Data storage)

What happens in this step?

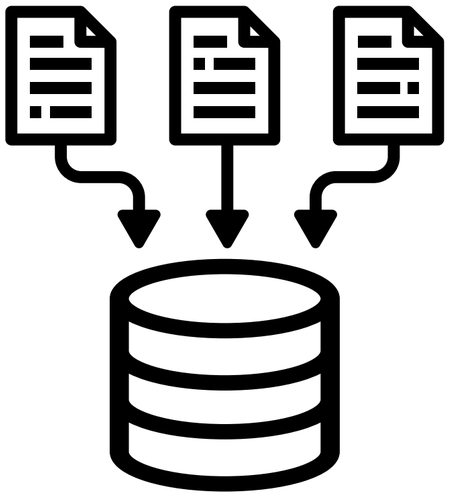
- Every incoming payload is stored immediately as a raw JSON file
- Stored before any data-quality validation or transformation
- Each payload gets a unique identifier (UUID): raw_id

How it's stored?

- File-based dropzone: DROPZONE_RAW_DIR/<raw_id>.json
- Original structure preserved exactly as received

Why this design?

- Durability: no data loss if downstream processing fails
- Replayability: raw data can be reprocessed anytime
- Debugging & audit: inspect original payloads when errors occur



Step 3: Processing Layer - Data transformation

What happens in /process?

- Reads a batch of raw JSON files from DROPZONE_RAW_DIR (limit = N)
- Processes each payload independently (one invalid file doesn't block others)
- Outputs analytics-ready metrics for storage

Transformations applied:

- UTC normalization: convert time_stamp → UTC (standardized timestamp)
- Aggregations: compute:
  - mean (average value): Measures the central value of the data by averaging all measurements.
  - std_dev (population standard deviation): Measures how much the values vary within the entire batch of data → Chosen because each payload is treated as a complete measurement set

Example:

- Input: "2019-05-01T06:00:00-04:00", [0,2]
- Output: timestamp_utc=2019-05-01 10:00:00 , mean=1.0, std_dev=1.0

Step 3: Processing Layer - Data transformation

Data Quality rules:

- time_stamp must be valid ISO8601 and include timezone
- data must be non-empty, numeric, and finite (no NaN/inf)

Track outcomes: add processing metrics

- checked files: raw files read from Bronze
- processed files: successfully transformed
- failed files: rejected due to data quality issues

```
{"processed":1,"failed":0,"checked":1}
```

Why this design?

- Reliable pipeline: bad data is counted as failed but raw stays for debugging
- Scalable: ingestion is decoupled; processing can run separately / in batches

Step 4: Silver Layer - Storing transformed data

Purpose:

- Persist clean, validated, analytics-ready data
- Serve as a reliable source for downstream analysis

What is stored?

- measured_at → UTC timestamp (primary key)
- mean → aggregated value
- std_dev → population standard deviation

How it's stored?

- PostgreSQL table: measurements
- UPSERT logic: Insert a new row if it doesn't exist; otherwise update the existing row when a conflict (same primary key) occurs.

Why UPSERT?

- Ensures idempotency (same timestamp can be reprocessed safely)
- Latest computation overwrites previous values
- Supports reprocessing and late-arriving data

Step 4: Silver Layer - Storing transformed data

[Transformation Service]

↓ UPSERT
(ON CONFLICT → UPDATE)

measurements

-measured_at (UTC) PK
- mean
- std_dev

1 row returned

	measured_at timestamp without time zone	mean double precision	std_dev double precision
1	2019-05-01 10:00:00	0.9985282517566032	1.01280885259315

Optional features

Optional Features

Testing (pytest):

What is implemented:

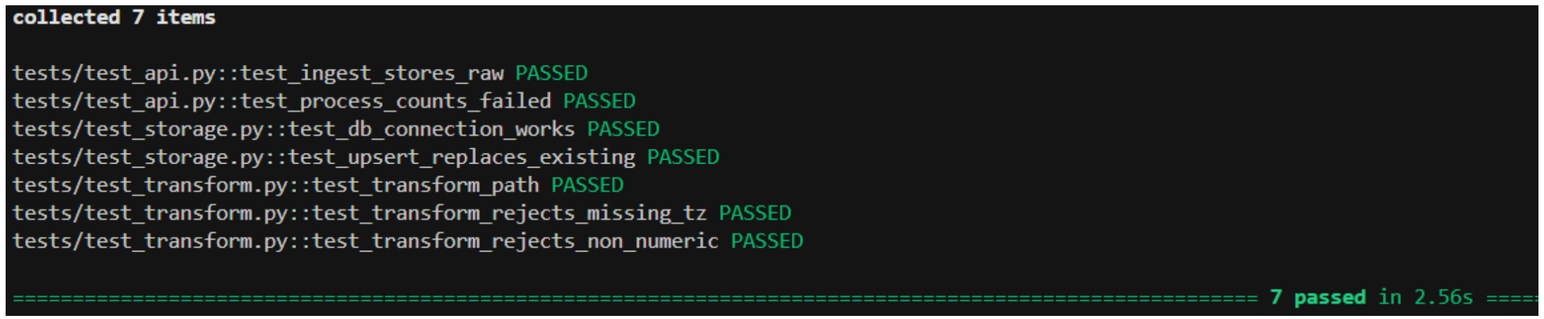
- Automated tests using pytest across core layers
- Clear separation between unit tests and integration tests

How it's structured:

- test_transform.py → pure unit tests
- test_api.py → API tests with temp raw storage + mocked DB
- test_storage.py → Postgres integration tests

Why it matters:

- Ensures correctness of transformations
- Catches regressions early
- Increases confidence in data quality and storage logic



Optional Features

Security (API Key):

What is implemented

- Simple header-based authentication using x-api-key

How it works

- API key checked via FastAPI dependency
- Key value read from environment variable
- Applied to sensitive endpoints (/process, /ingest)

Why it matters

- Prevents unauthorized access
- Lightweight and sufficient for internal / service-to-service APIs

Scalability:

What we implemented

- Decoupled ingestion and processing
  /ingest → accepts data fast, stores raw
  /process → batch processes raw files later

Why it matters

- Ingestion remains fast under load
- Processing can be scaled or retried independently
- Aligns with real-world data pipeline design

Optional Features

OpenAPI / Swagger Documentation:

What is implemented

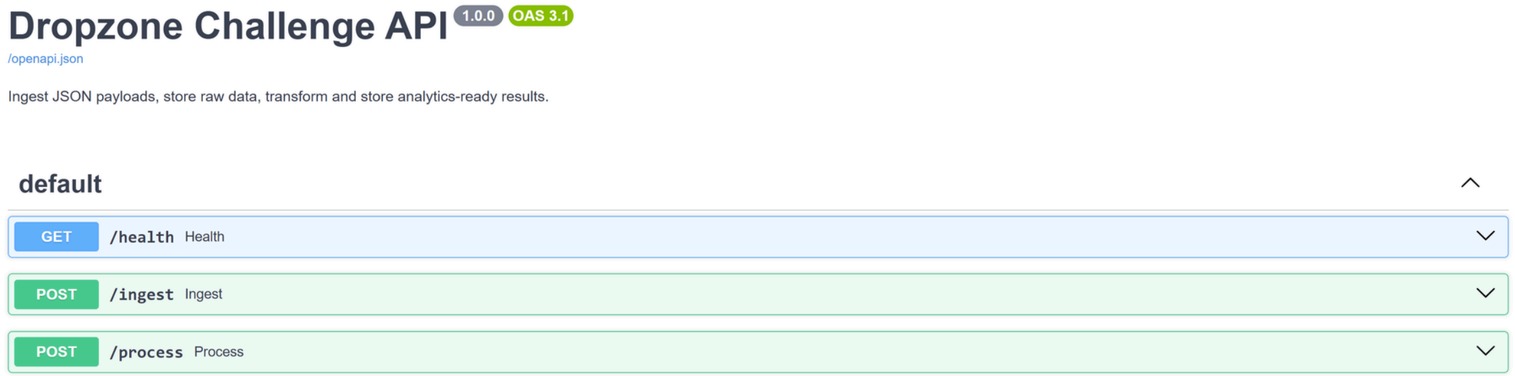
- Automatic API documentation via FastAPI

How it’s exposed

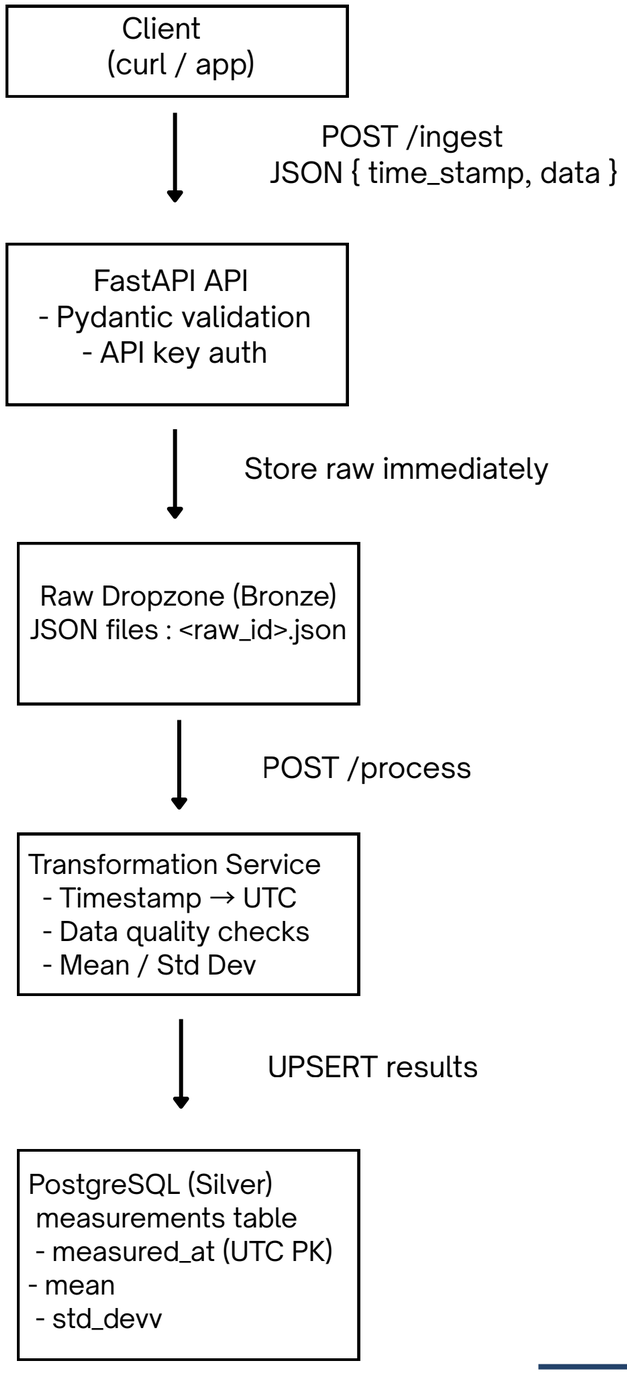
- Available at: /docs
- Includes endpoints, request schemas, and response models

Why it matters

- Clear contract for clients
- Easy testing and onboarding
- No extra documentation effort



Conclusion


Client
(curl / app)
POST /ingest
JSON { time_stamp, data }
FastAPI API
- Pydantic validation
- API key auth
Store raw immediately
Raw Dropzone (Bronze)
JSON files : <raw_id>.json
POST /process
Transformation Service
- Timestamp → UTC
- Data quality checks
- Mean / Std Dev
UPSERT results
PostgreSQL (Silver)
measurements table
- measured_at (UTC PK)
- mean
- std_devv

What this project demonstrates:

- Scalable architecture
- End-to-end data pipeline design
- Data quality enforcement
- Production-oriented API design

Limitations:

- Local execution only
- Batch processing only (no real-time / streaming processing)
- Local file-based bronze storage
- No schema evolution handling (payload structure is fixed)

Possible Improvements

1. **Cloud-based storage**
- Replace local file bronze storage with cloud object storage (S3 / GCS) to improve scalability, durability, and replayability of raw data.
2. **Asynchronous & scalable processing**
- Use a message queue (Kafka..) between ingestion and processing to improve throughput and isolates ingestion from heavy transformations.
3. **Schema validation & evolution**
- Introduce schema validation to support future payload changes without breaking the pipeline.
4. **Observability & monitoring**
- Add structured logging, metrics, and processing latency tracking to make the system operable and debuggable in production.



THANK YOU FOR YOUR ATTENTION