Weather ETL Pipeline (Joe) – Design and Analysis Report

Professor: Mayy Habayeb

Introduction

The goal of this project is to design and prototype a data model and ETL pipeline for weather prediction, integrating historical and forecast data sources to generate a 24-hour temperature forecast.

Data Model

1) Data Model Specification

UnifiedWeatherData				
Field Name	Data Type	Key/Constraint	Unit	Description
date	DATETIME	**PK** / NOT NULL	-	The specific date of the weather observation or forecast.
temp_max	FLOAT		◦C	Maximum daily temperature.
temp_min	FLOAT		◦C	Minimum daily temperature
precipitation	FLOAT	≥0	mm	Total daily precipitation amount
source	STRING	ENUM	-	Original data provider: ‘ECCC’ or ‘Open-Meteo’.
data_type	STRING	ENUM	-	Data categorization: ‘historical’ or ‘forecast’.

2) Explanation of chosen Columns

1) **Date**: Needed to compare historical data with forecast data. Without dates, we can’t create a proper timeline or compare past and future weather.
2) **temp_max, temp_min, precipitation**: These three are the required parameters from the API and represent the most important weather indicators for prediction. They are also common across both data sources, which makes integration possible.
3) **Source**: Tells us which system (ECCC or Open-Meteo) provided the record. This helps with traceability and quality checks.
4) **data_type**: Clarifies whether the record is “historical” or “forecast.” This is important because models treat past and future data differently.

3) Explanation of Data Types Chosen

1) **Date → datetime:** Allows easy sorting and time calculations (e.g., finding gaps or calculating averages by week).
2) **temp_max, temp_min, precipitation → float:** Weather values often have decimals, so float keeps the precision (e.g., 15.4°C instead of just 15°C).
3) **Source, data_type → string(ENUM):** These are labels, not numbers. Strings (ENUM) make the table easy to read. Using ENUM ensures consistency and makes queries easier to write and understand.

4) Constraints or Rules

1) **Date cannot be null:** Every record must have a date, otherwise it's meaningless for time-series analysis.
2) **temp_max, temp_min, precipitation can be null**: Sometimes data is missing due to equipment failure or gaps. Allowing nulls keeps the dataset realistic and prevents fake values from being added.
3) **precipitation_mm → ≥ 0:**Negative precipitation doesn't make sense, so this constraint enforces valid values.
4) **source → ENUM('ECCC', 'Open-Meteo'):** Restricting values prevents typos and ensures clean, standardized labels across the dataset.
5) **data_type → ENUM('historical', 'forecast'):** Ensures clear categorization and prevents invalid or inconsistent entries.

5) How This Model Solves the Integration Problem

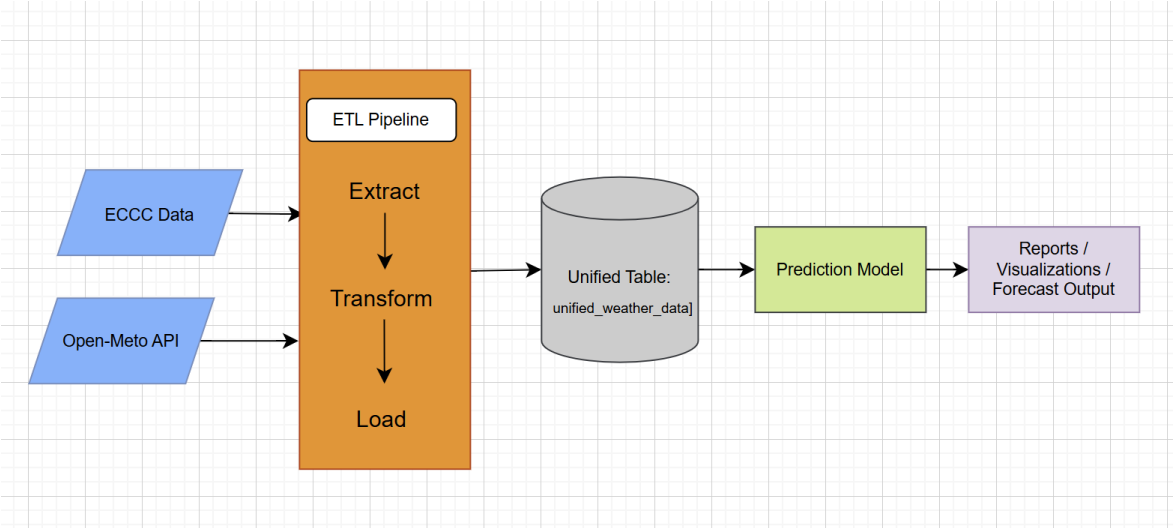
The ECCC and Open-Meteo datasets differ in naming conventions, formats, and time coverage. This unified schema solves the integration problem by:

- **Standardizing column names** (e.g., temp_max_c and temp_min_c instead of source-specific names).
- **consistent data types** for temperature, precipitation, and dates, ensuring compatibility during ETL.
- **Adding source and data_type columns** for traceability and context, without losing information about data origin or type.
- **Applying clear constraints** that guarantee data quality while allowing realistic handling of missing values.

As a result, all records—past and forecast—are stored in a single clean table, forming a continuous, standardized timeline. This makes it much easier to query, analyze trends, and train prediction models without worrying about mismatched structures between sources.

Architecture Diagram

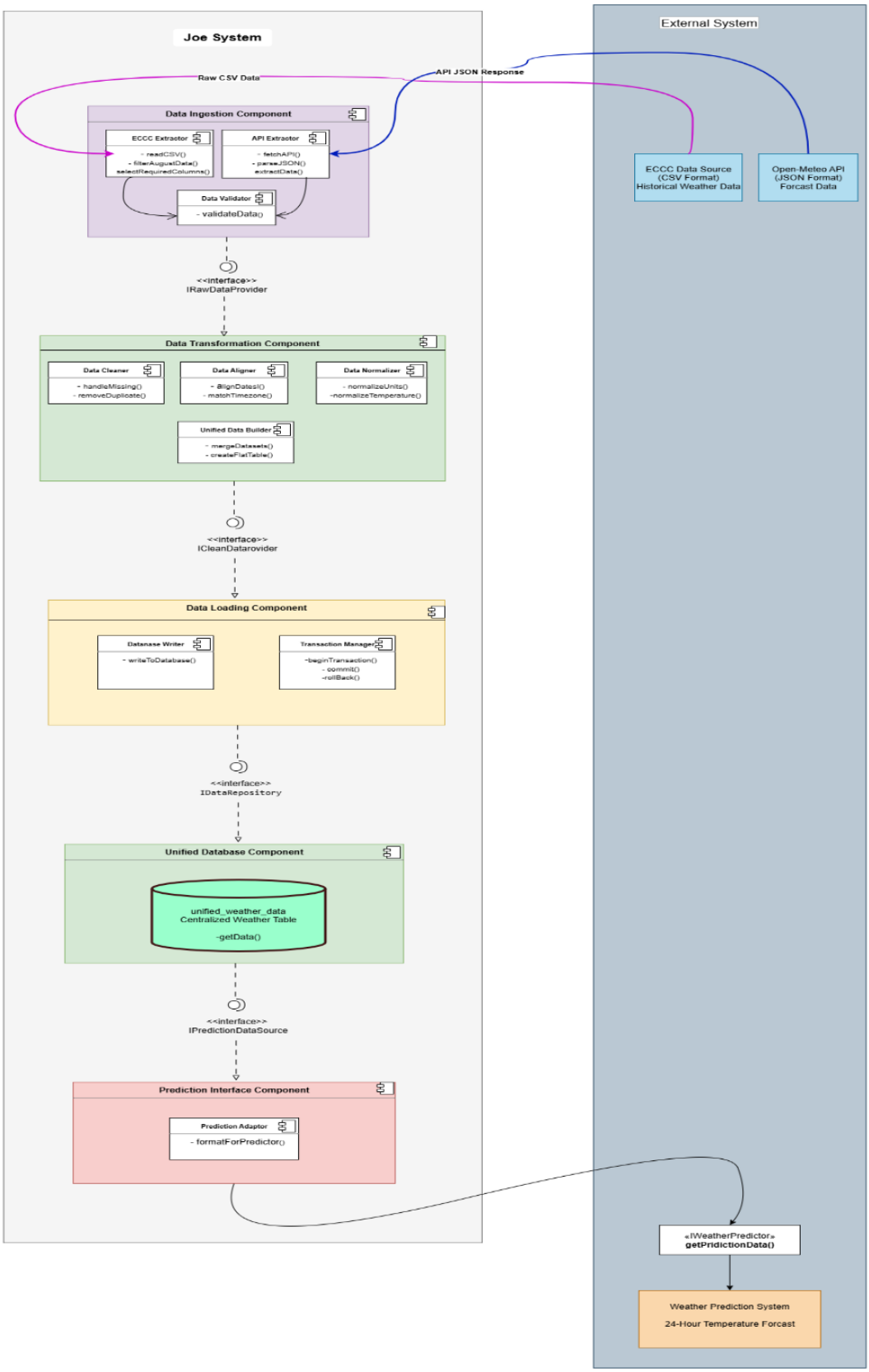
Figure 1: Architecture of Joe weather data system.



The architecture shows how data flows from two sources: **ECCC** for historical data and **Open-Meteo API** for forecast data. Both sources feed into the **ETL pipeline**, where the data is **extracted**, **transformed**, and **loaded** into a single **unified table**. This table standardizes the structure and format of both datasets. The **prediction model** then uses the unified data to generate forecasts, which are finally delivered through **reports, visualizations, or forecast outputs**. This architecture makes it easy to integrate different sources and provide consistent data for modeling.

Component Diagram & Component

The component diagram below illustrates the overall structure of the system and how different components interact through clearly defined interfaces. Each component handles a specific stage of the ETL pipeline, ensuring clean separation of responsibilities and smooth data flow from ingestion to prediction. The following sections provide a detailed explanation of the system, including the list of interfaces and their types, the role and main functions of each component, key design decisions with their trade-offs, and the design patterns applied throughout the architecture.


Joe System
External System
Raw CSV Data
API JSON Response
Data Ingestion Component
ECCC Extractor
- readCSV()
- filterAugustData()
selectRequiredColumns()
API Extractor
- fetchAPI()
- parseJSON()
extractData()
Data Validator
- validateData()
ECCC Data Source
(CSV Format)
Historical Weather Data
Open-Meteo API
(JSON Format)
Forcast Data
<<interface>>
IRawDataProvider
Data Transformation Component
Data Cleaner
- handleMissing()
- removeDuplicate()
Data Aligner
- AlignDates()
- matchTimezone()
Data Normalizer
- normalizeUnits()
-normalizeTemperature()
Unified Data Builder
- mergeDatasets()
- createFlatTable()
<<interface>>
ICleanDatarovider
Data Loading Component
Datanase Writer
- writeToDatabase()
Transaction Manager
-beginTransaction()
- commit()
-rollBack()
<<interface>>
IDataRepository
Unified Database Component
unified_weather_data
Centralized Weather Table
-getData()
<<interface>>
IPredictionDataSource
Prediction Interface Component
Prediction Adaptor
- formatForPredictor()
«IWeatherPredictor»
getPridictionData()
Weather Prediction System
24-Hour Temperature Forcast

Interfaces (names, type, purpose, signatures)

Interface	Type	Provided by	Required by	Purpose	Key operations (signature)
IRawDataProvider	Provided	Data Ingestion Component	Data Transformation Component	Expose validated raw datasets from both sources.	get_raw_eccc() -> DataFrame get_raw_api() -> DataFrame
ICleanDataProvider	Provided	Data Transformation Component	Data Loading Component	Provide cleaned, aligned, normalized, and unified flat table.	get_clean_df() -> DataFrame
IDataRepository	Provided	Data Loading Component	Unified Database Component	Provides a transactional and safe interface for saving data to the database.	write(df: DataFrame) -> None commit() -> None rollback() -> None
IPredictionDataSource	Provided	Unified Database Component	Prediction Interface Component	Read unified data for feature prediction.	get_unified(start: datetime, end: datetime) -> DataFrame
IWeatherPredictor	Provided (external service)	External Prediction System	Prediction Interface Component	Produce 24-hour temperature forecast from prepared features.	predict_24h(features: DataFrame) -> list[float]

Component roles & main functions

1. Data Ingestion Component

Component Role:

Entry point for all external data sources. Responsible for reading raw weather data from CSV files and APIs, filtering it to August data only, and performing initial validation.

Sub-components and Their Functions:

1.1 ECCC Extractor

Main Functions:

readCSV() --- Opens and reads CSV files from Environment and Climate Change Canada

Steps:

1. Locate CSV file path
2. Open file stream
3. Read file line by line
4. Parse each line into fields
5. Return raw data records

filterAugustData() -- Filters data to include only August months

Steps:

1. Receive all parsed CSV records
2. Check date field for each record
3. Extract month from date
4. Keep only records where month = August
5. Discard all other records
6. Return filtered August data

selectRequiredColumns() -- Selects only the necessary columns (date, temperature, location)

Steps:

1. Receive filtered data
2. Identify required column names
3. Extract only those columns from each record
4. Discard unnecessary columns
5. Return streamlined data with required fields only

1.2 API Extractor

Main Functions:

fetchAPI() -- Makes HTTP requests to Open-Meteo weather API

Steps:

1. Construct API URL with location parameters
2. Set date range to August only
3. Send GET request
4. Receive HTTP response
5. Check response status code
6. Return raw JSON response

parseJSON() -- Parses JSON response from API

Steps:

1. Receive JSON string from API
2. Parse JSON into objects
3. Navigate JSON structure to find temperature data
4. Extract temperature arrays
5. Extract timestamp arrays
6. Return structured data objects

extractData() -- Extracts specific temperature and date fields from parsed JSON.

Steps:

1. Receive parsed JSON objects
2. Locate temperature field
3. Locate datetime field
4. Extract values for August dates only
5. Create data records with date-temperature pairs
6. Return extracted data records

1.3 Data Validator

Main Functions:

validateData() -- Validates data from both ECCC Extractor and API Extractor.

Steps:

1. Receive data from ECCC Extractor
2. Receive data from API Extractor
3. Check for null/empty values
4. Verify temperature values are numeric
5. Check temperature range (-50°C to +50°C)
6. Verify dates are in correct format
7. Verify all dates are in August
8. Flag any invalid records
9. Log validation errors
10. Pass validated data to next component (Data Transformation)

2. Data Transformation Component

Component Role:

Cleans, aligns, and standardizes data from different sources (CSV and API). **Solves the date matching problem** by aligning timestamps and timezones. Prepares unified, clean data ready for database storage.

Sub-components and Their Functions:

2.1 Data Cleaner

Main Functions:

handleMissing() --- Handles missing temperature values in the dataset.

Steps:

1. Receive validated data from Data Ingestion Component
2. Scan each record for missing temperature values
3. Identify gaps in data (missing timestamps)
4. For small gaps (< 3 hours): apply linear interpolation
    - Take temperature before gap
    - Take temperature after gap
    - Calculate intermediate values
5. For large gaps (> 3 hours): mark as "missing" flag
6. Log all missing data locations
7. Return data with handled missing values

removeDuplicate() --- Removes duplicate records based on date and location.

Steps:

1. Receive data from handleMissing()
2. Sort records by date and time
3. Compare consecutive records
4. Check if date, time, and location are identical
5. If duplicates found:
    - Keep first occurrence
    - Delete subsequent duplicates
6. Log number of duplicates removed
7. Return deduplicated data

2.2 Data Aligner

Main Functions:

alignDatesI --- Aligns date formats from different sources to a single standard format and solves the date matching issue.

Steps:

1. Receive cleaned data (from CSV and API sources)
2. Identify source of each record (CSV vs API)
3. For CSV data (ECCC format):
    - Parse date string (e.g., "2024-08-15 14:30:00")

    - Extract year, month, day, hour, minute
4. For API data (Open-Meteo format):
    - Parse ISO 8601 or Unix timestamp
    - Convert to datetime object
5. **Standardize all dates to ISO 8601 format: "YYYY-MM-DDTHH:MM:SSZ"**
6. **Round minutes to nearest hour** (e.g., 14:37 → 15:00, 14:23 → 14:00)
7. Create aligned timestamp for each record
8. Return data with standardized dates

match Time zone() -- Converts all timestamps to the same timezone (UTC)

Steps:

1. Receive data with aligned date formats
2. Check timezone for each record:
    - CSV data might be in local time (EST/EDT)
    - API data might already be in UTC
3. Identify timezone offset for each record
4. Convert all timestamps to UTC:
    - If local time (EST): add 5 hours
    - If local time (EDT): add 4 hours
    - If already UTC: no change
5. Store timezone conversion metadata
6. **Create matching key: "YYYY-MM-DD-HH" for each record**
    - Example: "2024-08-15-14" for 2:00 PM on Aug 15
7. Return data with unified timezone (all UTC)

2.3 Data Normalizer

Main Functions:

normalizeUnits() --- Ensures all temperature measurements use the same unit system.

Steps:

1. Receive timezone-matched data
2. Check temperature unit for each record
3. Identify if temperature is in:
    - Celsius (from ECCC)
    - Kelvin (possibly from API)
    - Fahrenheit (if any source uses it)
4. Convert all temperatures to Celsius:
    - If Kelvin: subtract 273.15

    - If Fahrenheit: apply (F - 32) × 5/9
5. Round to 1 decimal place
6. Add unit metadata: "°C"
7. Return data with unified temperature units

normalizeTemperature() --- Standardizes temperature values to a consistent range/scale

Steps:

1. Receive unit-normalized data
2. Calculate statistics across all data:
    - Mean temperature
    - Standard deviation
3. **Optional: Apply z-score normalization** (for machine learning):
    - normalized_temp = (temp - mean) / std_dev
    - This scales values to center around 0
4. **Or: Keep original Celsius values** (for human readability)
5. Ensure values are within realistic bounds (-50°C to +50°C)
6. Flag any anomalies outside bounds
7. Return normalized temperature data

2.**4. Unified Data Builder**

Main Functions:

mergeDatasets() --- Combines CSV and API data based on matching timestamps

Steps:

1. Receive normalized data from all three sub-components above
2. Separate data by source:
3. Use matching key "YYYY-MM-DD-HH" from matchTimezone()
4. For each unique timestamp:
5. Handle conflicts (different temps for same time):
6. Create unified records with all fields
7. Return merged dataset

createFlatTable() --- Creates a single flat table structure ready for database loading

Steps:

1. Receive merged dataset
2. Define table schema:
3. For each merged record:
    - Map fields to table columns

    - Add metadata fields
    - Calculate quality score based on:
4. Validate table structure
5. Return flat table as list of records
6. Pass to Data Loading Component

Key Solution to Date Matching: The **Data Aligner** sub-component specifically solves the date matching issue through:

1. alignDatesI() - standardizes different date formats to ISO 8601
2. matchTimezone() - converts all times to UTC and creates matching keys

This ensures that temperature data from August 15, 2024 at 2:00 PM from both CSV and API sources will be recognized as the same timestamp and can be merged properly.

3. Data Loading Component

Component Role:

Manages the safe and reliable insertion of cleaned, unified data into the database. Ensures data integrity through transaction management and handles any errors during the database write process.

Sub-components and Their Functions:

3.1 Database Writer

Main Functions:

writeToDatabase() --- Writes the unified flat table data into the database

Steps:

1. Receive flat table from Data Transformation Component
2. Establish connection to the database
3. Prepare batch insert statement
4. Check for existing records
5. Perform write operations
6. Execute in batches (e.g., 100 records at a time) for efficiency
7. Coordinate with Transaction Manager for each batch
8. Log write statistics
9. Close database connection
10. Return operation status (success/failure)

3.2 Transaction Manager

Main Functions:

beginTransaction() --- Starts a database transaction before write operations

Steps:

1. Receive signal from Database Writer to start transaction
2. Send BEGIN TRANSACTION command to database
3. Set transaction isolation level
4. Create transaction context/session
5. Set transaction timeout (e.g., 30 seconds)
6. Mark transaction as "in progress"
7. Return transaction ID to Database Writer

commit() --- Commits successful transactions to make changes permanent

Steps:

1. Receive commit signal from Database Writer
2. Verify all write operations completed successfully
3. Check no integrity constraints were violated
4. If all checks pass:
    - Send COMMIT command to database
    - Make all changes permanent
    - Release all locks
5. Mark transaction as "completed"
6. Log successful commit with timestamp
7. Return success status to Database Writer

rollBack() --- Reverts all changes if an error occurs during the transaction

Steps:

1. Receive error signal from Database Writer
2. Identify what triggered the rollback
3. Send ROLLBACK command to database
4. Undo all changes made during transaction:
5. Release all locks held by transaction
6. Mark transaction as "failed"
7. Log rollback reason and details
8. Clean up transaction resources
9. Return error status to Database Writer
10. Database Writer can retry or report error

Why Transaction Management is Important:

- **Atomicity:** All records in a batch are written together or none at all (no partial writes)

- **Consistency:** Database constraints are maintained (no invalid data)
- **Isolation:** Other processes don't see partial/uncommitted data
- **Durability:** Once committed, data survives system crashes

This ensures the weather data remains reliable and complete.

4. Unified Database Component

Component Role:

Central data repository that stores all cleaned, unified, and validated weather data. Provides a single source of truth for historical weather information used by the prediction system.

Database Table: unified_weather_data

Table Structure (Schema):

A centralized table containing weather records with the following key fields:

- **date:** Primary key, specific date of weather observation or forecast (DATETIME, NOT NULL)
- **temp_max:** Maximum daily temperature in °C (FLOAT)
- **temp_min:** Minimum daily temperature in °C (FLOAT)
- **precipitation:** Total daily precipitation amount in mm (FLOAT, ≥0)
- **source:** Original data provider - 'ECCC' or 'Open-Meteo' (STRING, ENUM)
- **data_type:** Data categorization - 'historical' or 'forecast' (STRING, ENUM)

Main Function:

getData() --- Retrieves historical weather data for prediction generation.

Steps:

1. Receive query request with date range
2. Execute SQL query with filters on date and data_type
3. Fetch matching weather records (temp_max, temp_min, precipitation)
4. Format results as structured DataFrame
5. Return data to Prediction Interface Component

5. Prediction Interface Component

Component Role:

Serves as the interface between the unified database and external prediction systems. Prepares historical weather data in the format required by the 24-hour temperature forecast model.

Sub-component and Its Function:

5.1 Prediction Adaptor

Main Function:

formatForPredictor() --- Transforms database records into the feature format required by the external prediction system

Steps:

1. **Receive forecast request** from External Prediction System
2. **Query historical data** from Unified Database using getData() for last 30 days
3. **Engineer prediction features:**
    - Calculate daily temperature ranges and rolling averages
    - Extract precipitation trends and seasonal patterns
4. **Format features** into DataFrame structure required by prediction model
5. **Validate feature data** to ensure completeness and correct data types
6. **Send features** to External Prediction System via IWeatherPredictor interface
7. **Receive 24-hour forecast** (24 hourly temperature predictions)
8. **Format output** with timestamps and metadata
9. **Return forecast** to requesting system

Design Decisions and Trade-offs

1. Layered Architecture (Pipeline Design)

Decision: Separated system into sequential layers: Ingestion → Transformation → Loading → Storage → Prediction.

Trade-offs:

Pros: Clear separation of concerns, easy to test and debug, team can work on components independently

Cons: More complex, slower sequential processing, entire pipeline stops if one component fails

Justification: Makes data flow easy to understand and helps identify problems quickly - good for learning and debugging.

2. Unified Database vs. Separate Tables

Decision: Store all weather data in one unified_weather_data table.

Trade-offs:

Pros: Simple queries, single source of truth, faster predictions, easier maintenance

Cons: Less flexible for very different data structures, harder to track source-specific metadata

Justification: Both sources have similar data (temperature, precipitation), so unification simplifies architecture while the source field preserves origin.

3. Date/Time Normalization to UTC

Decision: Convert all timestamps to UTC and round to nearest hour.

Trade-offs:

Pros: No timezone confusion, easy matching across sources, industry standard

Cons: Loses sub-hourly precision, daylight saving time complexities

Justification: Hourly granularity sufficient for daily forecasts; UTC prevents timezone errors.

4. Synchronous Processing

Decision: Components communicate directly rather than using message queues.

Trade-offs:

Pros: Simpler to implement and debug, lower infrastructure needs, easier for students

Cons: Less scalable, no automatic retry, components tightly coupled

Justification: Daily/weekly updates don't need real-time streaming; simplicity appropriate for college project.

5. Transaction Management

Decision: Use explicit transactions (begin, commit, rollback) for database writes.

Trade-offs:

Pros: Guarantees data integrity, prevents partial/corrupted data, professional practice

Cons: Added complexity, slight performance overhead

Justification: Ensures database consistency - critical for accurate predictions.

Design Patterns Used

1. Repository Pattern

Where: Unified Database Component

Why: Abstracts database access behind getData() interface, hiding SQL queries from other components.

Benefits: Can change database technology (PostgreSQL → MongoDB) without affecting other components; centralized data access logic.

2. Adapter Pattern

Where: Prediction Interface Component (Prediction Adaptor)

Why: Bridges gap between database format (raw records) and prediction model format (engineered features).

Benefits: Database and prediction model evolve independently; can swap ML models without changing database structure.

3. Strategy Pattern

Where: Data Transformation Component (Data Cleaner)

Why: Different missing data strategies based on gap size - interpolation for small gaps, other approaches for large gaps.

Benefits: Easy to test different approaches and add new strategies without modifying existing code.

4. Factory Pattern

Where: Data Ingestion Component

Why: Creates appropriate extractor based on source type (CSV → ECCCExtractor, API → APIExtractor).

Benefits: Easy to add new data sources; centralized extractor creation logic.

Summary: The design balances simplicity (for student project) with professional practices (patterns, clean architecture). Trade-offs favor ease of understanding over maximum performance, making the system maintainable and extensible.

Attachment:

Python output:

```
============================================================
  STEP 1: DATA INGESTION
============================================================
📂 Reading ECCC CSV file...
✓ Successfully read 31 records from CSV
📋 Selecting and mapping required columns...
✓ Selected columns: ['date', 'temp_max', 'temp_min', 'precipitation', 'source', 'data_type']
🗓️ Filtering for August data...
✓ Filtered to 31 August records for year 2025
🌐 Fetching data from Open-Meteo API...

============================================================
  VALIDATION
============================================================

🔍 VALIDATING DATA...

============================================================
  TRANSFORMATION
============================================================

💾 Saving unified table to CSV: c:\Users\tesne\Documents\Semester 3\COMP 248 - AI System Design\Assignment 1\wea
ther_etl\data\unified_data.csv
✓ Unified table saved to c:\Users\tesne\Documents\Semester 3\COMP 248 - AI System Design\Assignment 1\weather_et
l\data\unified_data.csv

🔲 Done.
PS C:\Users\tesne\Documents\Semester 3\COMP 248 - AI System Design\Assignment 1\weather_etl>
```