Task 1: Crew Carbon Data Engineer Prompt by Abdel Alfahham

> Objective: Build a small and reproducible data pipeline as a part of a repository to ingest lab and waste water operations data, QA/QC data and calculate MRV.

Summary of Deliverables

> I opted to create the following repo structure for Task 1.

Dockerfile

(1) I decided to use a Dockerfile saved in `/Dockerfile`. This file allows me to consolidate all the commands needed to assemble a docker image. Using a docker image guarantees the portability of the pipeline and ensures that there are no external dependencies for the pipeline to run (whole app is self-contained). It also allows me to easily deploy the pipeline on cloud services such as AWS ECR (Amazon Elastic Container Registry).

docker-compose file

(2) A `docker-compose.yml` file was created to define all the services that are required for the pipeline to run and for data users to interact with the data.

I choose to define the following individual services:

- `python app` (primary back-end)
- `postgreSQL database` (to store the normalized data)
- `streamlit dashboard` (for multi stakeholder data plotting, visualization, review and export)
- `jupyter notebook` (for Data Science workflows and EDA)

docker-compose network:

- `streamlit dashboard` is exposed at `http://0.0.0.0:8501/`
- `jupyter notebook` is exposed at `http://localhost:8888/`
- `postgreSQL database` has the following credentials `postgresql://crewcarbon:localdev@postgres:5432/carbon_mrv` (local only)

docker-compose commands:

- **Step 1**: Build Container by running `docker-compose build`
- **Step 2**: Start Services by running `docker-compose up -d`
- **Step 3**: Run the command to create base schema defined tables and run all (data and MRV) pipelines by running `make run-all-pipelines`. For more information see Makefile section. The make command provided above will run the following `docker-compose` commands.

```
    docker-compose up -d
    # to reset the database tables and recreate the base tables
    docker-compose exec app python src/ingest/create_tables.py
    # to run raw data transformation pipelines
    docker-compose exec app python src/ingest/run_data_pipeline.py
    # to create the CO2 removal dataset
    docker-compose exec app python src/ingest/run_mrv_pipeline.py
```

- **Step 4**: Start Dashboard by running `docker-compose up dashboard`

Makefile

A Makefile is used to unify all the docker-compose commands that are necessary for a user/engineer to spin up, spin down, rebuild and rerun the services/pipelines sequentially in the appropriate order.

`make` commands:

```
  make up             - Start containers
  make down           - Stop containers
  make shell          - Open bash shell in app container
  make python         - Open Python REPL in app container
  make db-shell       - Open PostgreSQL shell
  make test           - Run tests
  make clean          - Remove all data and containers
  make run-all-pipelines -Create or recreate tables and ingest data then
calc MRV
```

Information about the Submitted Repository Structure

`requirements.txt`

All necessary python packages are saved here - this fille is used to create the python environment in the docker container.

`/data`

Data files provided for this prompt are saved here. Those files are also loaded from this directory.

`src/dashboards`

Logic to create the CO2 Removal Dashboard is saved here. This is a rudimentary `.py` file that will spin up a CO2 removal dashboard. Please be adviser that the dashboard requires that the data be populated. Make sure to run `make run-all-pipelines` beforehand.

`src/ingest`

The directory contains most of the ingest and data transformation functions/runners/dependencies.

`src/ingest` Pipelines

- `src/ingest/ca_pipeline.py` : Calcium Data Transformation Pipeline (uses shared utils).
- `src/ingest/ops_plant_a_pipeline.py` : Pipeline to transform PLANT A Ops data.
- `src/ingest/ops_plant_b_pipeline.py` : Pipeline to transform PLANT B Ops data.
- `src/ingest/ph_pipeline.py` : pH Data Transformation Pipeline (uses shared utils).

`src/ingest` Runners

- `src/ingest/create_tables.py` : Script will delete and recreate the schema allowing for rapid ingest iteration.
- `src/ingest/run_data_pipeline.py`: Script that runs the data transformation functions and writes to sql tables.
- `src/ingest/run_mrv_pipeline.py`: Script that runs the MRC calculation functions and writes to sql tables.

`src/ingest` Utilities

- `src/ingest/utils.py` : Shared functions that are used for data transformation.

`src/models`

This is the directory where tables, schemas and database related variables are saved.

- `src/models/database.py`: Helper function for database interactions
- `src/models/schemas.py`: All Data schemas defined in this file. Schema for post transformed data and schema for CO2 calculation attributes.

`src/mrv`

This is the directory where MRV related helpers are saved.

- `utils.py`: Functions for CO2 removal calculations - functions are defined as `calculate_co2_removal_from_sources` and `bulk_calculate_co2_removal`.

`/tests`

This is the directly where unit tests would be added. Right now a simple test MRV calcs was created to run using the following command

```
docker-compose exec app pytest
tests/test_mrv_calcs.py::test_calculate_co2_removal_valid_data -v
```

or

```
make test
```

Data Transformation Logic (Ca and pH)

Calcium Pipeline (run_ca_pipeline)

- Loads calcium lab data from CSV
- Transforms it using a generic utility: selects relevant columns, renames them to match database schema, converts data types
- Compresses extra metadata fields into a single JSON column for storage
- Returns cleaned, standardized dataframe ready for database insertion

pH Pipeline (run_ph_pipeline)

- Loads two separate pH sensor data files (minute-by-minute readings) and concatenates them
- Applies the same transformation process: column selection, renaming, type conversion
- Compresses extra metadata (sensor ID, source file, etc.) into a single JSON column
- Returns cleaned, standardized dataframe ready for database insertion

Common Pattern

Both pipelines follow an identical workflow:

```
Load → Transform (standardize Data) → Compress metadata → Return
Normalized Data
```

> The key difference is data source: calcium is lab-based (daily), pH is sensor-based (minute-level), but they both end up as structured records in the same `CrewCarbonLabReading` table.

Plant Operations Data Transformation Logic

Plant A Operations (run_ops_plant_a)

- Loads 3 Excel files (April, May, June 2025)
- Cleans column names (removes newlines, applies standard naming)
- Uses predefined mapping to rename operator data columns to consistent schema
- Extracts key columns: date, effluent flows (actual/max/min), bypass metrics
- Concatenates all months, standardizes dates, filters invalid entries
- Returns cleaned dataframe with operational metrics for PLANT_A

Plant B Operations (run_ops_plant_b)

- Loads 6 overlapping Excel files covering March-July 2025
- Cleans multi-level headers by flattening and joining column names
- Removes empty rows and summary statistics (TOTAL/MAX/MIN/AVG rows)
- Extracts key columns: effluent flow, influent flow, date
- Standardizes date format and filters invalid dates

- Returns cleaned dataframe with operational metrics for PLANT_B

Common Pattern

Both pipelines follow:

```
Load → Clean headers → Filter/standardize → Extract key columns → Return
Normalized Data
```

Key differences:

- Plant B: More files, multi-level headers, focuses on effluent/influent flow only
- Plant A: Fewer files, includes bypass data and min/max flow metrics

However, Both produce standardized records that conform to the `WasteWaterPlantOperation` schema.

CO2 Removal Calculation Logic

Step 1:

Get the delta between upstream and downstream Ca2+

```
    ca_upstream = ca_upstream_reading.value
    ca_downstream = ca_downstream_reading.value
    ca_delta (mg/L) = ca_downstream – ca_upstream (mg/L)
```

Step 2:

Get the flow rate in litres per day < from > m3 per day < from > MGD. The flow rate that will be used is the Effluent flow rate since it provides the most conservative values and it is the value that is available for both Plant A and Plant B.

However, It is important to acknowledge that future improvement to the algorithm should take into account evaporation and precipitation from/to the system. This would require recording those values for **All** Plants going forward.

```
    flow_mgd = ops.actual_eff_flow_mgd
    flow_m3_day = flow_mgd * 3785.41
    flow_l_day = flow_m3_day * 1000
```

Step 3:

Get the mass of CaCo3 in mg

```
caco3_mg = ca_delta * flow_l_day * ca_to_caco3
```

Step 4:

Get the mass of CO2 in mg

```
# Molecular weights (g/mol)
MW_Ca = 40.078
MW_CaCO3 = 100.0869
MW_CO2 = 44.0095

# Molecular weight ratios
ca_to_caco3 = MW_CaCO3 / MW_Ca
co2_to_caco3 = MW_CO2 / MW_CaCO3

# Mass calculations
caco3_mg = ca_delta * flow_l_day * ca_to_caco3
co2_mg = caco3_mg * co2_to_caco3
```

Step 5:

Convert the mass of CO2 from mg to MT/day

```
co2_mt_day = co2_mg / 1_000_000_000
```

QAQC and Validation

MRV Validation Logic Summary

This module `src/qaqc/mrv_utils.py` contains validation and data quality logic that is applied before/during calculating CO2 removal in wastewater treatment plants.

Validation Checks (in order)

1. Operational Data Check

- Ensures operational records exist with valid flow rates (> 0)
- Fails if: No data found or invalid/missing flow
- Result: Calculation skipped

2. Calcium Readings Check

- Verifies both upstream and downstream calcium measurements exist
- Fails if: Either measurement is missing
- Result: Calculation skipped

3. Calcium Delta Check

- Calculates difference between downstream and upstream calcium (the "delta")
- Flags if: Delta is zero or negative (indicates no CO2 removal)
- Result: Calculation proceeds but marked "INVALID"

Master Validation Function

- validate_all_inputs() runs all three checks sequentially and returns:
- should_calculate: True (proceed) or False (skip)
- quality_flag: "VALID", "INVALID", "NO_OPS_DATA", "INVALID_FLOW", or "MISSING_CA_READINGS" - message: Optional explanation

Outcome

- Critical failures (checks 1-2): Stop calculation entirely
- Non-critical issues (check 3): Allow calculation but flag for review

> Please note: This ensures only records with sufficient data are calculated while preserving problematic records for auditing.

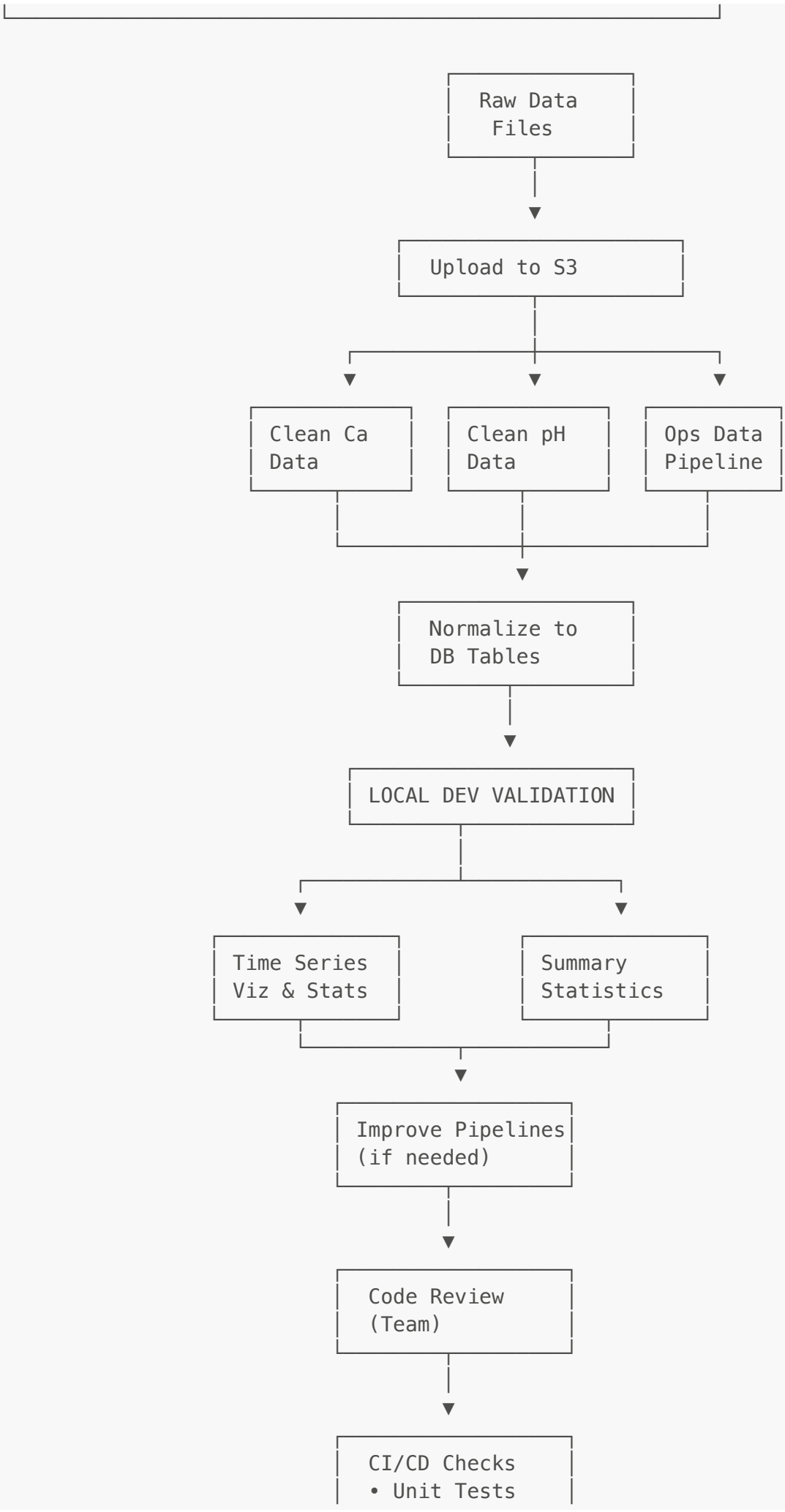
Task 2: Cloud Deployment Strategy

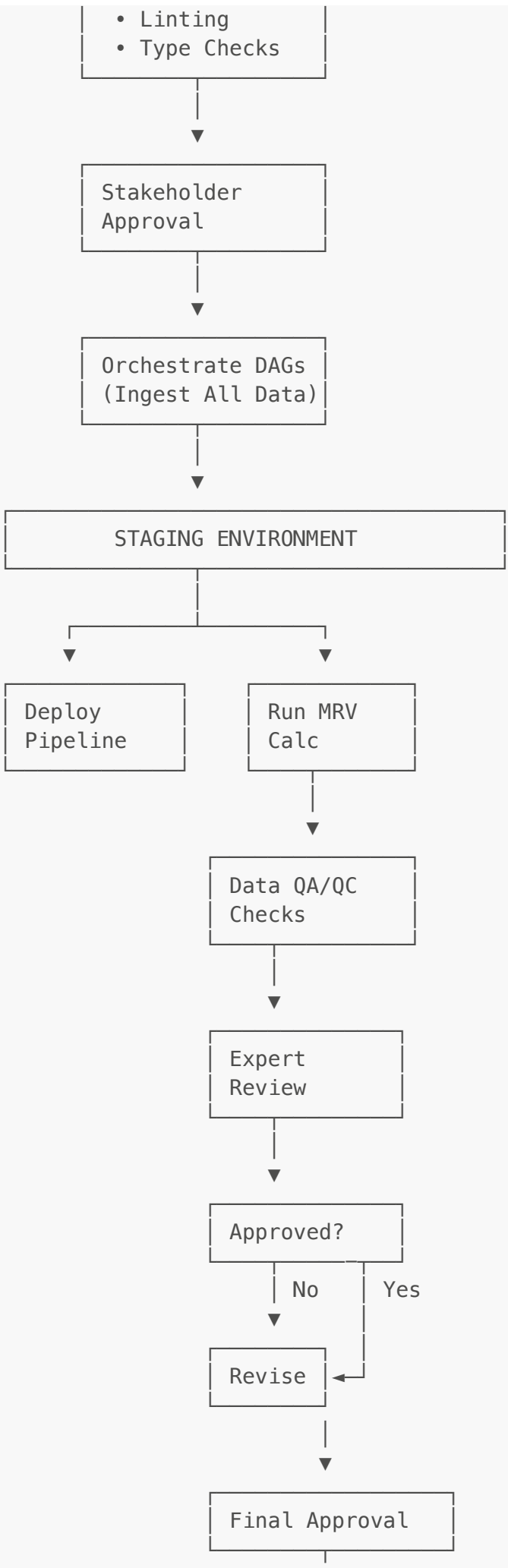
> Objective: Build a small and reproducible data pipeline as a part of a repository to ingest lab and waste water operations data, QAQC data and calculate MRV.

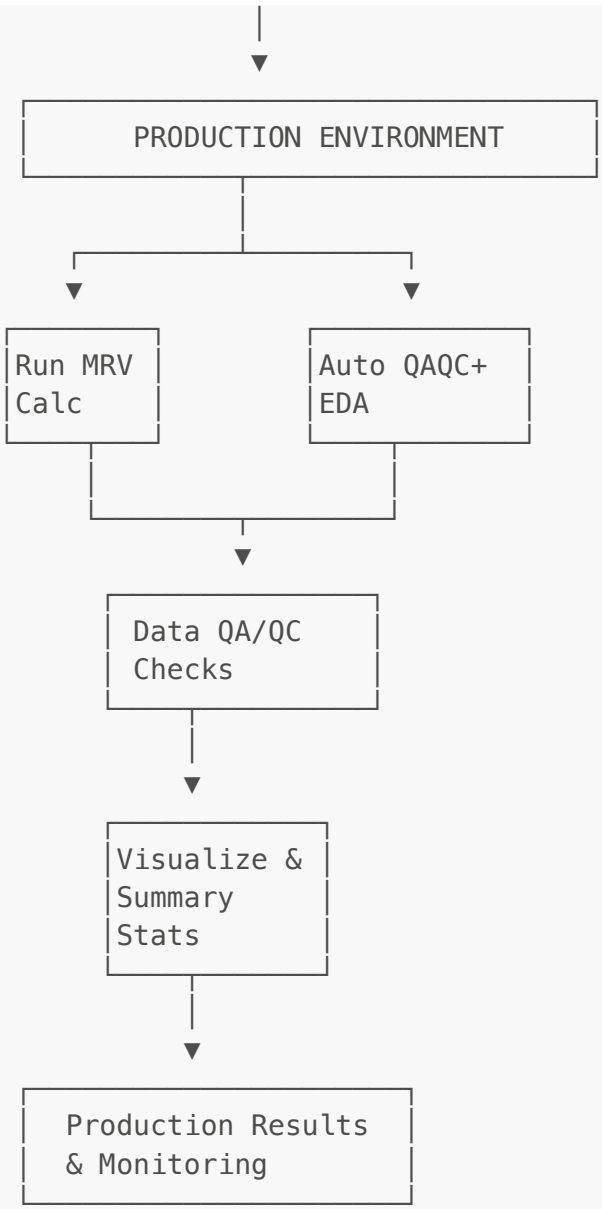
Data Platform:

- Raw data files would be stored in AWS S3 [Simple Storage Service] (or equivalent). Each waste water treatment plant would have its own assigned prefix/folder in order to ensure separation of raw data and to make it much easier to upload new data files.
- Transformed Data should be stored in intermediate PostgreSQL tables. The data that is most relevant for MRV calculation and plant operation metadata should be normalized as much as possible. However, It should be acceptable to create one-off tables that do not have a strict schema for analytical or experimental workflows.
- MRV Data and calculations should be the most normalized and most QAQC'd out of all datasets due to their critical role in business operations and institutional guarantees.

End to End Data Workflow


Raw Data Files
Upload to S3
Clean Ca Data
Clean pH Data
Ops Data Pipeline
Normalize to DB Tables
LOCAL DEV VALIDATION
Time Series Viz & Stats
Summary Statistics
Improve Pipelines (if needed)
Code Review (Team)
CI/CD Checks
• Unit Tests


• Linting
• Type Checks
Stakeholder Approval
Orchestrate DAGs (Ingest All Data)
STAGING ENVIRONMENT
Deploy Pipeline
Run MRV Calc
Data QA/QC Checks
Expert Review
Approved?
No
Yes
Revise
Final Approval



QAQC and Validation

Proposed Data QA/QC and Validation Hierarchy

Level 1: Raw Data Ingestion Checklist

- [ ] File format valid
- [ ] Encoding correct
- [ ] Basic parsing successful
- [ ] Duplicates detected/handled

Level 2: Data Completeness Checklist

- [ ] Non-null required columns
- [ ] Missing values identified
- [ ] Records exist for all dates
- [ ] Upstream/downstream readings present

Level 3: Schema & Type Validation Checklist

- [ ] Column data types correct
- [ ] Required fields present
- [ ] Date format validated
- [ ] Numeric ranges within bounds
- [ ] Foreign key integrity maintained

Level 4: Data Quality Flags

- [ ] Ca delta validation passed
- [ ] Flow rate checks passed
- [ ] Quality flags assigned correctly
- [ ] Invalid data flagged appropriately

Level 5: Business Logic & Scientific Validation

- [ ] MRV Calculations correct
- [ ] CO_2 removal logic accurate
- [ ] Domain rules applied
- [ ] Expert review completed

Data Access for Broader Organization

Data Engineer

Data engineers have full read/write access to development and staging environments. They work locally using Docker containers, Jupyter notebooks, and direct PostgreSQL queries to develop, test, and debug pipelines. They interact with the codebase through GitHub, push changes through CI/CD, and validate data quality using custom scripts. Their primary responsibility is ensuring data flows correctly through the system from raw ingestion to processed tables.

Data Scientist/Modeler

Data scientists access data through read-only database queries and the Streamlit dashboard for exploratory analysis. They use Jupyter notebooks to perform statistical analysis, create custom visualizations, and investigate trends. They can download CSV exports for external analysis tools and share findings with the team. They focus on understanding data patterns and generating insights without modifying production systems.

Domain Expert

Domain experts review validated results through the Streamlit dashboard in staging and production environments. They filter data by plant, date range, and quality flags to assess MRV calculations, then approve or request revisions before production deployment. They can download detailed CSV files for manual validation and have no database or code access. Their role is to ensure calculations are scientifically sound and meet validation criteria.

DevOps / Infrastructure

DevOps engineers manage the complete infrastructure across development, staging, and production using AWS Console, CI/CD pipelines, and monitoring tools. They deploy code changes, manage database backups and scaling, configure environment variables, and troubleshoot infrastructure issues. They have full access to all systems and monitoring dashboards. Their focus is maintaining system health, performance, and reliability.

Project Manager / Stakeholder

Stakeholders view high-level metrics and KPIs through the production Streamlit dashboard. They can filter data by plant and track CO_2 removal trends over time without technical access. Their role is monitoring overall project progress, making business decisions, and communicating results to leadership. They interact with the system purely through dashboards and reports.

Cost, Security, Performance and Scalability

Cost Optimization

Infrastructure Costs:

The system is designed with cost efficiency in mind using containerized Docker architecture that can run on minimal resources during (local) development. For production, AWS RDS PostgreSQL provides predictable pricing with reserved instances. S3 storage for raw data files can be extremely cheap with intelligent tiering automatically moving infrequently accessed data to cheaper storage classes. The Streamlit dashboard and application containers can run on ECS, allowing auto-scaling that charges only for actual compute time used, avoiding idle resource costs.

Data Pipeline Costs:

Orchestration runs on scheduled intervals (daily/weekly) rather than continuous processing, reducing compute costs. The validation pyramid approach catches data quality issues early in the pipeline, preventing expensive reprocessing. By storing only essential columns and compressing metadata into JSON fields, database storage requirements are minimized. CSV exports and visualizations are cached using Streamlit's @st.cache_data decorator, reducing redundant database queries and compute overhead.

Cost Monitoring:

AWS Cost Explorer tracks spending by service and environment (dev/staging/prod). CloudWatch metrics monitor database query performance to identify expensive operations.

Security

Access Control:

Role-based access control (RBAC) ensures team members only access resources necessary for their role. Database credentials are stored in AWS Secrets Manager and injected at runtime, never hardcoded.

Development, staging, and production environments are completely isolated with separate databases and AWS accounts.

Data Protection:

Data is encrypted at rest using AES-256 in RDS and S3. All data transfers use TLS 1.3 encryption in transit. PostgreSQL enforces SSL connections for all remote access. Database backups are encrypted and stored in separate AWS regions for disaster recovery.

Network Security:

Production databases run in private VPC subnets with no public internet access. Security groups restrict traffic to only necessary ports and IP ranges. Application containers communicate through internal service discovery. A more aspirational implementation would require VPN or bastion hosts for engineers to access staging/production systems.

Compliance & Auditing:

All database modifications are logged with timestamps and user identifiers. The quality_flag JSONB field stores a complete audit trail of validation decisions. Git commit history tracks all code changes with mandatory code reviews. MRV calculations include source_file and version for full data lineage.

Performance

Query Optimization:

PostgreSQL indexes should be created on frequently queried columns (plant_id, date, quality_flag) to speed up filtering and joins. The dashboard uses cached queries with @st.cache_data to avoid repeated database hits. Composite indexes on (plant_id, date) support efficient time-series queries. Database connection pooling prevents connection overhead for repeated queries.

Dashboard Performance: Time-series visualizations use Plotly which renders efficiently in the browser with client-side interactions. Data is aggregated at appropriate granularities (daily averages for pH, daily totals for CO_2) before visualization. The dashboard lazy-loads data only when users change filters. Large datasets are paginated and limited to reasonable date ranges (default 3 months) to prevent browser memory issues.

Monitoring: CloudWatch tracks database query execution times and identifies slow queries for optimization. Application performance monitoring (APM) with Datadog captures end-to-end request latency. Database connection pool metrics identify bottlenecks. The team reviews performance dashboards weekly to identify degradation trends.

Scalability

Horizontal Scaling:

The containerized architecture allows multiple application instances to run behind a load balancer as traffic increases. Database read replicas can be added to offload analytical queries from the primary write

instance. S3 scales automatically to handle unlimited file storage growth. The stateless dashboard design allows adding container instances without session coordination.

Vertical Scaling:

RDS instance types can be upgraded with minimal downtime during maintenance windows. CPU and memory can scale independently based on workload characteristics. Storage auto-scaling increases database disk space automatically when utilization reaches thresholds.

Data Growth:

The current schema supports millions of records with proper indexing. TimescaleDB extensions can be added if time-series performance degrades at scale. Table partitioning by date range can be implemented to maintain query performance as historical data grows. Archival strategies move old data (>2 years) to cheaper cold storage while maintaining query access.

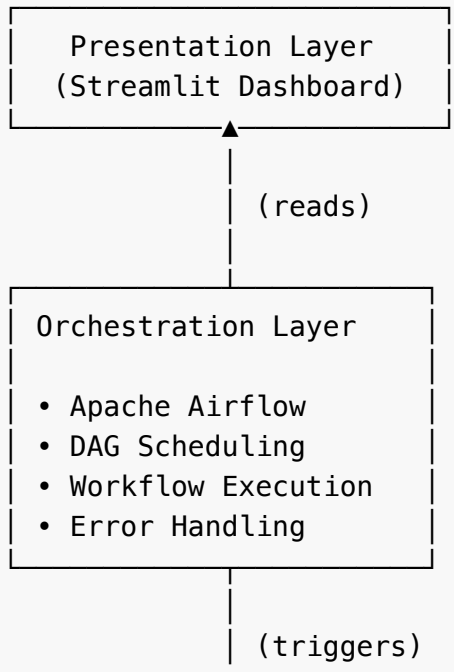
Geographic Expansion:

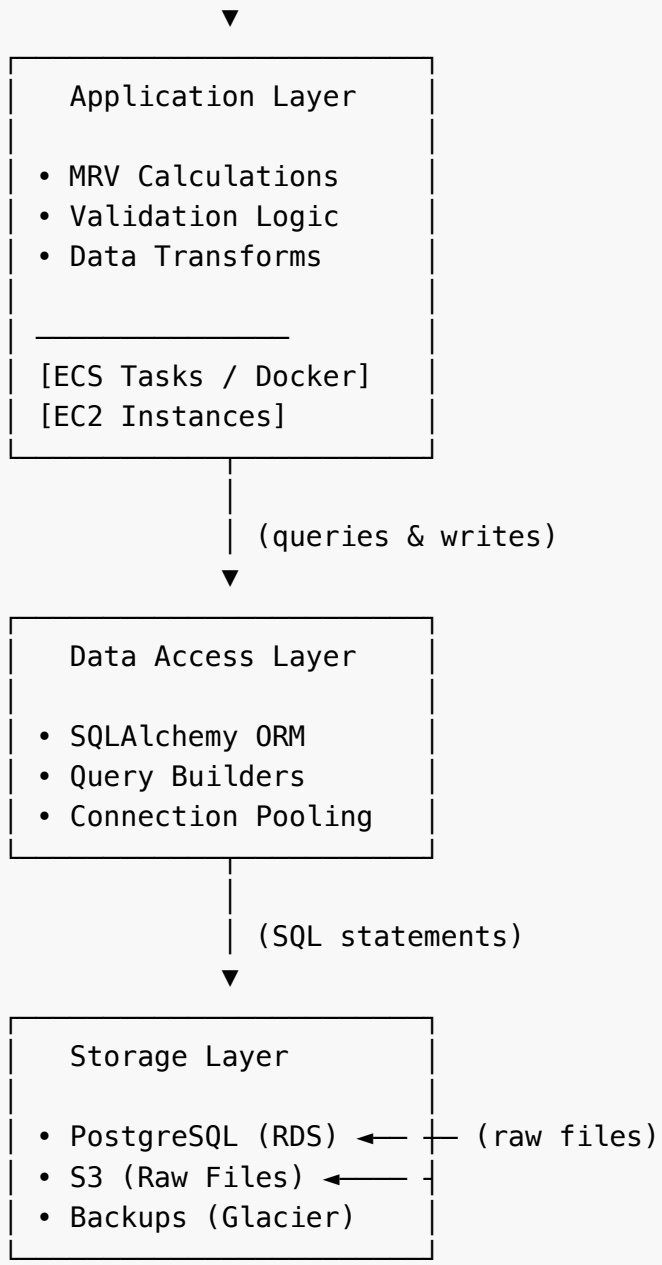
Multi-region deployment supports plants in different geographic locations. Database replication keeps staging/production in sync across regions. S3 cross-region replication ensures data availability during regional outages. CloudFront CDN caches dashboard assets globally for low-latency access worldwide.

Generalisability of Pipeline Components:

The modular pipeline architecture allows adding new data sources (additional plants, sensors, parameters) without redesigning the core system. The generic transform_crew_data() function standardizes ingestion for any new data format. JSON metadata fields provide schema flexibility for storing plant-specific attributes. The validation framework scales to accommodate new business rules and QA/QC checks as requirements evolve.

Data Architecture Diagram




Application Layer
• MRV Calculations
• Validation Logic
• Data Transforms
[ECS Tasks / Docker]
[EC2 Instances]
(queries & writes)
Data Access Layer
• SQLAlchemy ORM
• Query Builders
• Connection Pooling
(SQL statements)
Storage Layer
• PostgreSQL (RDS)
• S3 (Raw Files)
• Backups (Glacier)
(raw files)