IoT Interface for Sustainable Energy Projects: Final Report

Grupo de Investigación en Ingeniería de Diseño de Producto

Juan Camilo Ospino Martínez

1. Context

This project was developed as part of my special project for the Research group, under the 2030 Sustainable Energy initiative. Its purpose is to design a lightweight, resilient, and modular data acquisition pipeline using affordable hardware (Raspberry Pi 4) and open technologies (Python, SQL, Thingspeak, Power BI).

As pilot solar and wind projects are deployed, it's essential to store, aggregate, and visualize data not only for technical diagnostics but also for strategic insight. The ultimate goal is to support both engineering and administrative decision-making in our national-level Sustainable Energy for 2030 efforts.

The system simulates real sensor behavior and prepares the architecture to scale once the actual sensors and hardware are field-deployed.

2. System Architecture Overview

The current system was designed with simplicity, transparency, and modularity in mind. It aims to offer an end-to-end pipeline from raw sensor simulation to final data visualization while staying adaptable to the constraints of early-stage renewable energy deployments. This architecture is composed of lightweight components that can scale horizontally and be replaced or upgraded incrementally.

2.1 Core Pipeline Description

- Data is simulated every 30 seconds and stored locally as CSV.
- Every 30 minutes, these are aggregated into compressed Parquet files.
- Parquet data is uploaded to a MySQL database and sent to ThingSpeak.
- Visualizations are created using Power BI.
- All jobs are coordinated by a Python cron manager.

2.2 Data Flow Diagram

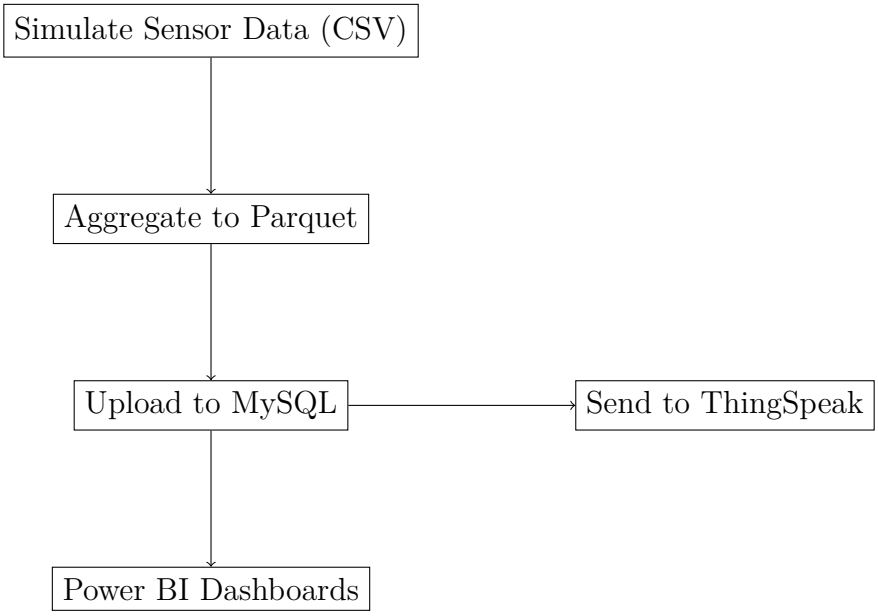

Figure 1: High-level Data Flow Architecture

2.3 Folder Structure

```
energy_monitoring/
|- scripts/
|  |- generate_sample_data.py
|  |- aggregate_parquet.py
|  |- upload_to_sql.py
|  |- upload_thingspeak.py
|  |- cron_manager.py
|- config/
|- data/
|- logs/
|- db/
|- tests/
|- requirements.txt
```

Each script is designed to be independent and testable. The architecture follows a modular design to allow for future extensions, such as new ingestion endpoints, dashboard layers, or data validation services.

2.4 Pipeline Orchestration and Design Rationale

Given the small scale and non-critical nature of the pipeline at this prototyping phase, orchestration was implemented with native `cron` jobs and Python. The decision to avoid containerization or workflow orchestrators (e.g., Docker, Airflow) was intentional to keep the system simple and easy to debug. With data collected only every 30 seconds (a rate agreed upon by the engineering team for relevant variables), there is no immediate need for more complex or high-availability setups.

Future work may include transitioning to containerized deployments for ease of scaling, especially when real sensors come online and distributed nodes are deployed.

2.2.1 Expected CSV Format

The data ingestion scripts expect input CSV files in a row-based format, where each row corresponds to a single sensor measurement taken at a specific time. The expected structure is as follows:

```
timestamp,project_id,sensor_id,sensor_type,value,unit
2025-06-05T08:41:47Z,Solar_Facade,Temp_1,temperature,29.75,C
2025-06-05T08:41:47Z,Solar_Facade,Volt_1,voltage,22.33,V
2025-06-05T08:41:47Z,Solar_Facade,Amp_1,current,1.63,A
2025-06-05T08:41:47Z,Solar_Facade,Wind_1,wind_speed,4.39,m/s
2025-06-05T08:41:47Z,Solar_Facade,Irr_1,irradiance,401.09,W/m²
2025-06-05T08:41:47Z,Solar_Facade,PressHigh_1,pressure,198898.83,Pa
2025-06-05T08:41:47Z,Solar_Facade,PressLow_1,pressure,205202.27,Pa
```

Field Descriptions:

- `timestamp` – ISO 8601 timestamp in UTC
- `project_id` – Identifier for the data source project (e.g., `SOLAR_1`, `HAWT_1`).
- `sensor_id` – Short code for the sensor (e.g., `TEMP_1`, `VOLT_1`).
- `sensor_type` – Human-readable type (e.g., temperature, wind_speed).
- `value` – Numeric measurement from the sensor.
- `unit` – Unit of measurement (e.g., `C`, `V`, `A`, `W/m²`).

CSV files not conforming to this format may raise parsing errors during aggregation or upload. This schema ensures that ingestion, aggregation, and SQL upload scripts can work consistently without requiring hardcoded assumptions. Validation of this format is currently minimal, but future plans include detecting malformed packets, dropouts, or packet corruption using both file- and row-level checks.

2.6 Additional Design Commentary

- **Database Engine:** MySQL was chosen primarily for familiarity and ease of use. While not specifically tuned for time-series data, its structured schema allowed straightforward OLTP-style queries and joins.
- **ThingSpeak:** As an IoT platform already available via the university's license, ThingSpeak enabled low-friction deployment of test dashboards, verifying connectivity, authentication, and cloud feasibility.
- **Power BI:** Chosen for its user-friendliness and broad support among non-programmer stakeholders. However, it is understood that for future scalability, a custom dashboard (e.g., Streamlit, Dash) hosted in the cloud would be more suitable.

3. Codebase Documentation, Logging, and Modularity

All major functions within the codebase have been instrumented with descriptive docstrings following the NumPy documentation format, and embedded with structured logging calls using Python's built-in `logging` module. Each log entry records timestamps, severity levels, and function context to enable clear traceback during runtime.

3.1 Logging Strategy

Log files are stored in the `logs/` directory and rotated daily. Events include:

- Sensor data generation
- File creation and deletion (CSV/Parquet)
- SQL insertions and connection errors
- API calls to ThingSpeak
- Cron job execution success/failure

An example log entry:

```
2025-06-04 18:30:02,113 - INFO - aggregate_parquet -
Successfully aggregated 60 CSV records into hourly_data.parquet
```

3.2 Scalability via Modular Design

The pipeline's folder and function structure promote separation of concerns. Each Python module encapsulates a single responsibility and exposes clearly defined entry points. This separation simplifies debugging and allows easy upgrades, such as switching from cron to a workflow orchestrator or from ThingSpeak to another cloud service.

Furthermore, centralized configuration files store keys, paths, and sampling intervals, allowing changes without modifying the core code logic.

3.3 Version Control and Development Cycle

The entire codebase has been version-controlled using Git and hosted on GitHub. Development followed a feature-branch strategy:

- `main`: Production-ready stable code
- `scripts/upload_thingspeak`: Cloud communication
- `scripts/aggregate_parquet`: Data aggregation and storage optimization
- `scripts/upload_to_sql`: Structured storage implementation

Commits were atomic and descriptive, facilitating rollbacks and debugging during rapid prototyping.

4. Database Schema and Query Design

4.1 Schema Overview

The database follows an OLTP model, normalized to Second Normal Form (2NF), with partial normalization into 3NF. The schema is designed for reliability, extensibility, and consistency across projects and sensor configurations.

Figure 2: Simplified Entity-Relationship Diagram (ERD) of the OLTP schema

4.2 Tables and Normalization Rationale

- **projects**: Defines metadata and identifiers for each energy monitoring prototype (e.g., façade, tile, turbine).
- **sensors**: Captures type, unit, and calibration data for each sensor, linked to a project.
- **readings**: Time-stamped measurement data, with foreign keys to sensor and project IDs.

This relational structure:

- Eliminates redundancy by ensuring each sensor/project is defined only once.
- Supports general queries (e.g., average voltage by project, temperature trends) efficiently.
- Allows scalability when adding new sensor types or experimental setups.

4.3 Example Query

```
SELECT p.project_name, s.sensor_type,
       AVG(r.value) AS avg_value
FROM readings r
JOIN sensors s ON r.sensor_id = s.id
JOIN projects p ON s.project_id = p.id
WHERE s.sensor_type = 'temperature'
  AND r.timestamp BETWEEN NOW() - INTERVAL 1 DAY AND NOW()
GROUP BY p.project_name, s.sensor_type;
```

This type of query benefits from the normalization, ensuring clarity and low coupling between metadata and measurements.

5. Sensor Simulation and Data Validation Strategy

5.1 Current Status

Since physical sensor deployment is pending, synthetic data is generated using Gaussian-distributed values centered around domain-accurate means. This facilitates testing of the full pipeline under realistic load and schema expectations.

5.2 Future Validation Mechanisms

The next development stage will implement robust validation strategies to account for:

- **Packet loss**: Checkpoint-based confirmation between cron jobs and file counters.
- **Corrupt data**: Apply schema validation via `pydantic` or `pandera`.
- **Outliers**: Deploy rolling Z-score methods or quantile clipping to flag anomalies.
- **Missing fields**: Trigger logging alerts and quarantine affected files.

5.3 Integration with Logging

All validation failures or suspicious metrics will be logged with warning or error severity levels, enabling both real-time alerts and retrospective forensic analysis.

6. Current Limitations

- **Storage:** Raspberry Pi SD cards (64 GB) can support several months of Parquet data. For example, with 15 sensors per row every 30 seconds and 3 projects, it is estimated it will use between **400 and 600 MB/Month** employing Parquet, which will take approximately 100 months in theory though in reality might just work for a year. Exact retention depends on the volume and frequency of simulation.
- **ThingSpeak:** Free accounts support only 8 channels and 1 update per 15 seconds.
- **Power BI:** The free tier restricts scheduled refresh and sharing across users. Pro license required for broader collaboration.

- **No Cloud DevOps:** No current system engineer is assigned to implement a production-grade cloud deployment.

7. Suggested Action Plan for Deployment

- Each Raspberry Pi stores local data as Parquet.
- Data is uploaded hourly to a centralized cloud-hosted SQL server.
- Select data may be sent to ThingSpeak for visual inspection.
- MQTT or REST APIs can be introduced to support real-time streaming if network reliability improves.

7.1 Future Scalability Considerations

- Choose cloud provider (AWS, Azure, GCP) based on institutional constraints.
- Implement security, access control, and backup protocols.
- Consider replacing Power BI with a custom dashboard in Dash or Streamlit.

7.2 Estimated Cloud Cost (Monthly)

Service	Provider	Est. Monthly Cost (USD)
Cloud SQL (basic tier)	AWS RDS / Azure SQL	\$15–30
Storage (20–50 GB)	S3 / Azure Blob	\$0.46–1.15
Power BI Pro (per user)	Microsoft	\$10
Optional ML/Monitoring	Varies	\$5–10

Table 1: Estimated monthly cost if scaled to the cloud

8. Repository and Codebase Access

The complete project will be available on GitHub:

`https://github.com/JCOM127/IoT_GRID`

9. Conclusion

This project demonstrates the feasibility and technical foundations for a modular, scalable, and cost-effective monitoring infrastructure tailored for small-scale renewable energy systems. Through the use of Raspberry Pi devices and a Python-based pipeline, we successfully simulated the acquisition, preprocessing, and structured storage of environmental and operational data from solar and wind energy prototypes.

The system leverages a hybrid file strategy—temporary CSV files for rapid local capture and Parquet files for efficient long-term aggregation. This approach balances write speed

with optimal compression, reducing storage requirements significantly (up to 75–90% compared to plain text files). Data is normalized and uploaded hourly to a university-provided IoT platform (ThingSpeak), illustrating potential for future cloud-based dashboards or real-time analytics. While ThingSpeak was chosen for convenience in the prototyping phase, the architecture remains cloud-agnostic and can transition to more robust platforms as needed.

Database design follows OLTP principles with normalization up to Second and partial Third Normal Form (2NF/3NF), supporting frequent write operations and facilitating typical aggregation queries, such as average temperature per project. This schema structure ensures data integrity while allowing extensibility for additional sensor types, projects, or derived metrics.

From an operational standpoint, the use of Python scripts executed via cron jobs provided a lightweight, maintainable orchestration layer. Though minimalistic, this approach is appropriate for the 30-second sampling rate, especially in the absence of real-time constraints. Logging and version control were incorporated to ensure traceability, with clear docstrings and modular code enabling future scaling, debugging, or migration to containerized solutions like Docker.

Although the system currently operates with simulated sensor inputs, the framework anticipates the challenges of real deployment, including corrupted packets, missing data, and intermittent connectivity. Future work will incorporate validation layers and redundancy mechanisms to improve robustness and reliability. Moreover, the potential addition of MQTT protocols and real-time visualization platforms will further enhance the system's responsiveness and usability in field conditions.

In essence, this project lays a solid foundation for a full-fledged telemetry and monitoring solution adaptable to diverse renewable energy applications in both academic and operational contexts. The current implementation balances practicality with foresight, ensuring that the system can grow in complexity and scale without sacrificing maintainability or performance.