As requested, please find the comments inline.

# SECTION TO FILL

## What are the advantages of the technologies you used for the project?

I used the following technologies:

1. **Python** : Enabled me with rapid development, therefore ideal for data parsing and scripting. We can extend data analytics aspect in future with libraries like pandas.
2. **Django** : Django framework has built in ORMs, allowing for flexibility to interact with just using Python code. Ideal for storing structured data and querying/reporting. Django provides efficient bulk inserts and can be scaled for processing large batches of NEM12files asynchronously. In futures, it will be easy to implement authentication, access control, audit logging etc.
3. **Multiprocessing** : For parallel execution, leveraging multiple CPU cores to improve performance.
4. **PostgreSQL** : Used as database because easier to enforce business rules, example implementing composite unique constraints (nem,timestamp). It is also better suited for high ingestion rates and large datasets as compared to MySQL/SQLite due to high-concurrency reads and writes without locking the tables. Support for parallel writes , bulk inserts and asynchronous commits is also better.

## How is the code designed and structured?

Code follows the basic Django Code structuring where **project is a container** for multiple **apps**, each with a single responsibility.

**Project** :

* meter\_data\_pipeline is the repo name. The project name is src. “src” contains settings, URLs and global config. For the scope of this project we do not use the URLs yet.
* settings.py hold the configs (DB, middleware, installed apps, etc.)

**App**

* The app focussed on NEM12 data ingestion. The app name is “data\_ingestion\_app”
* The entry point is management/commands/parse\_data.py – This is common structure when you want to invoke a process via custom command. In this case, the whole process can be invoked by running the command:   
  python manage.py parse\_data --file <input NEM12 file>
* The core logic sits in “services.py”. Code adheres to the **SOLID principles**, especially Single Responsibility and Open/Closed principles.
  + - DataParser: Parses raw data blocks and extracts structured records.
    - DDLScriptCreatorForMeterReadings: Converts parsed data into SQL insert scripts.
    - process\_data, read\_file\_in\_chunks, and run: Handle multiprocessing, streaming, and orchestration.
* Unit and integration tests reside in tests.py

meter\_data\_pipeline/src  
├── data\_ingestion\_app - App for NEM12 data ingestion

│ ├── admin.py

│ ├── apps.py

│ ├── constants.py - Configs specific to the app

│ ├── management

│ │ └── commands

│ │ └── parse\_data.py. - Custom command for initiating data parsing

│ ├── migrations - Versioned changed to database

│ │ ├── \_\_init\_\_.py

│ │ ├── 0001\_initial.py

│ │ ├── 0002\_alter\_meterreadings\_id.py

│ │ ├── 0003\_alter\_meterreadings\_timestamp.py

│ │ ├── 0004\_alter\_meterreadings\_timestamp.py

│ │ └── 0005\_alter\_meterreadings\_nmi.py

│ ├── models.py - Database tables

│ ├── services.py - Logic for parsing and writing output file

│ ├── tests.py - Contains unit tests and integration tests

│ └── views.py - Logic for API (not used for now)

├── manage.py

└── src - Project settings module

├── asgi.py

├── settings.py

├── urls.py

└── wsgi.py

## How does the design help to make the codebase readable and maintainable for other engineers?

1. Tried to follow SOLID principles for code writing, specially Single Responsibility and Open/Closed Principle
   1. Each method and class has a clear and limited scope of responsibility. Separating I/O, parsing and SQL generation reduces interdependencies
   2. Naming convention and modular structure are intuitive enough for quick understanding of code flow.
2. Django provides predictable project layout familiar to most backend engineers.

## Discuss any design patterns, coding conventions, or documentation practices you implemented to enhance readability and maintainability.

1. Applied pipeline pattern, where data flows in stages from parsing input file to SQL output
2. Coding convention followed is PEP8
3. Each method and class include docstrings to explain its behavior. Inline comments have also been included.
4. Tried to follow SOLID principles for class and function design. This improves flexibility and testability
5. Included a Readme.md file for instructions on installation and running of the application

## What would you do better next time?

1. Data Cleaning and validations
2. Refactor multiprocessing logic into a reusable class or module to reduce redundancy.
3. Use pythons logging module for run time observability

## Reflect on areas where you see room for improvement and describe how you would approach them differently in future projects.

1. Improve input validation and exception handling during parsing to reduce silent failures.
2. Support configurable file formats and output destinations via settings.
3. Test coverage can be better

## What other ways could you have done this project?

1. Instead of SQL script, would have used Django ORM for real-time data insertion
2. If data entry has to happen via SQL script, for some business reason, COPY statements are much faster than INSERT.
3. Exposed Django REST API toa allow real-time file ingestion via UI
4. Expose Database via Django Admin
5. I discovered that PostgreSQL has extensions like **TimescaleDB,** which turn it into a high-performance **time-series database** with:
   1. Automatic chunking
   2. Compression
   3. Continuous aggregates

It would be great to explore that in this context.

1. Use Celery and Redis queue.
   1. Its more suitable for production.
   2. This allows concurrent processing and batching across workers. Also it is safer in failure scenarios. If app crashes mid write, in-memory queue data is lost. With Celery, tasks remain queued in Redis and can retry automatically when services are up.
   3. Operational Monitoring is also better with Celery.
   4. Gives flexibility to schedule cron jobs, handle retries and trigger alerts.

## Explore alternative approaches or technologies that you considered during the development of the project.

1. Considered using **threading**, but opted for **multiprocessing** to bypass Python's GIL limitations.
2. Explored **pandas** for structured data handling, but raw parsing was faster for this specific task.
3. Evaluated using Django ORM and signals but chose plain SQL generation for portability. Also, it was specific requirement of the project.