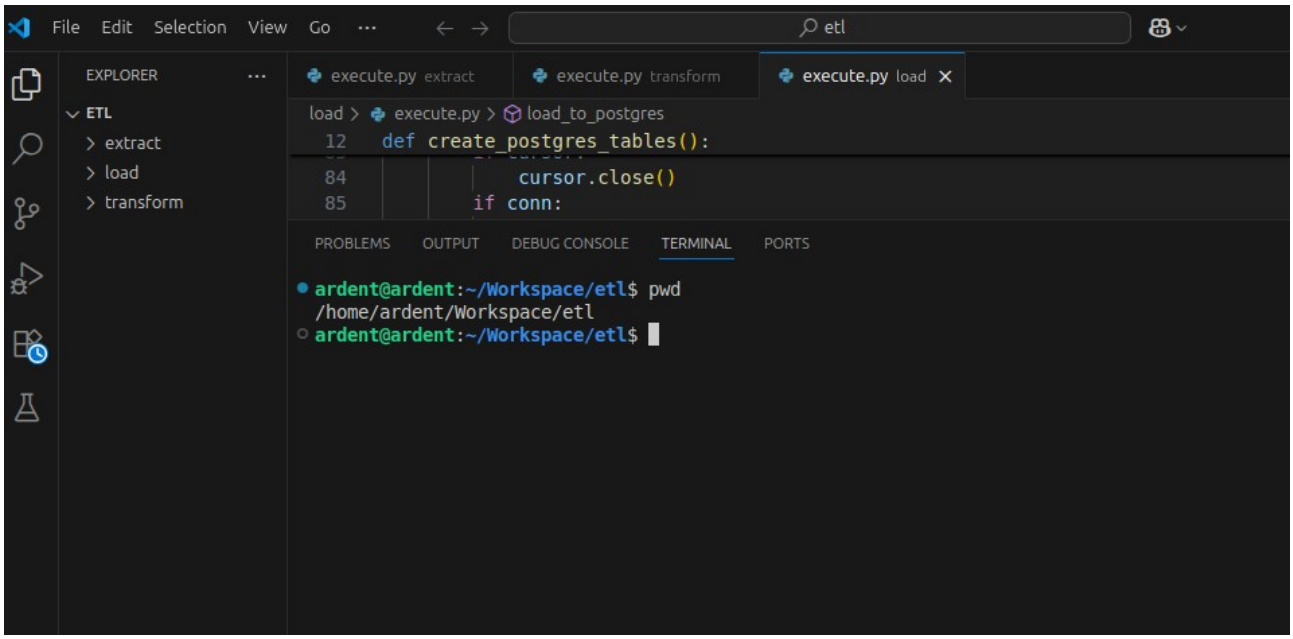
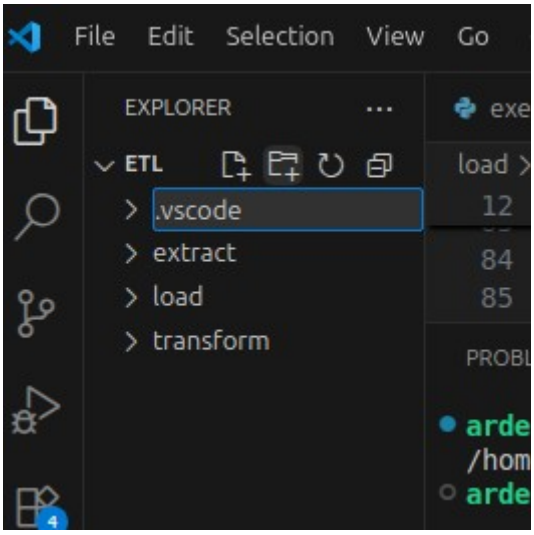
Let us clean our workspace/etl project a little bit. There are several issues with the current code, we will not dive too deep into those. Also, note that this is not the one and only way to do things. Everyone should figure out what works for them and stick with that.

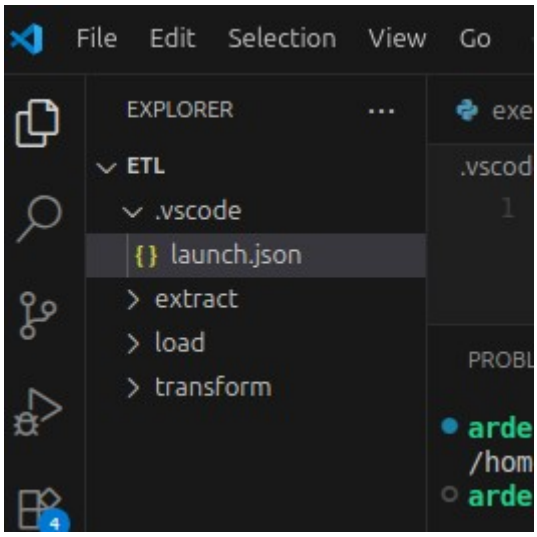
1. Open your ETL project in vscode. It should have three different folders extract,load,transform and an individual execute.py file inside all of it.



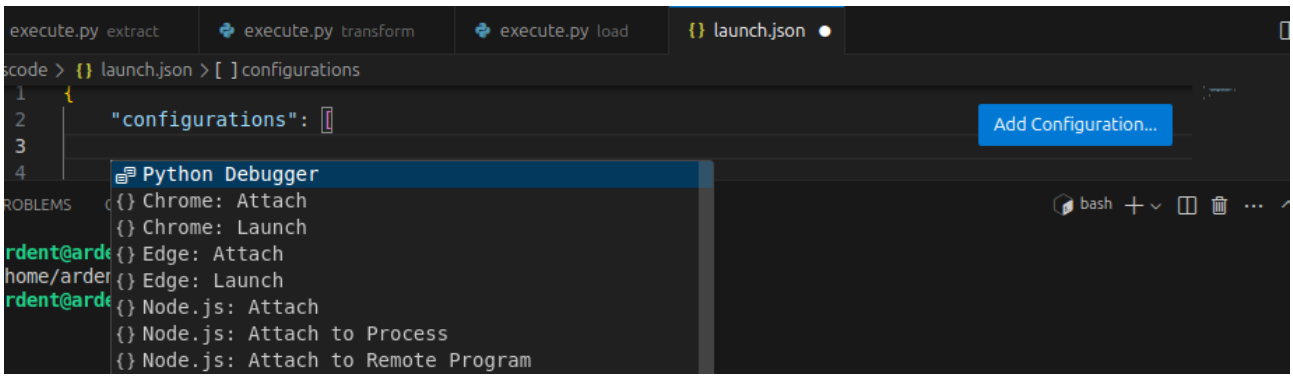
2. Create a new folder titled .vscode in your workspace directory. Make sure the name is exact ".vscode" as it is what the visual studio will look for

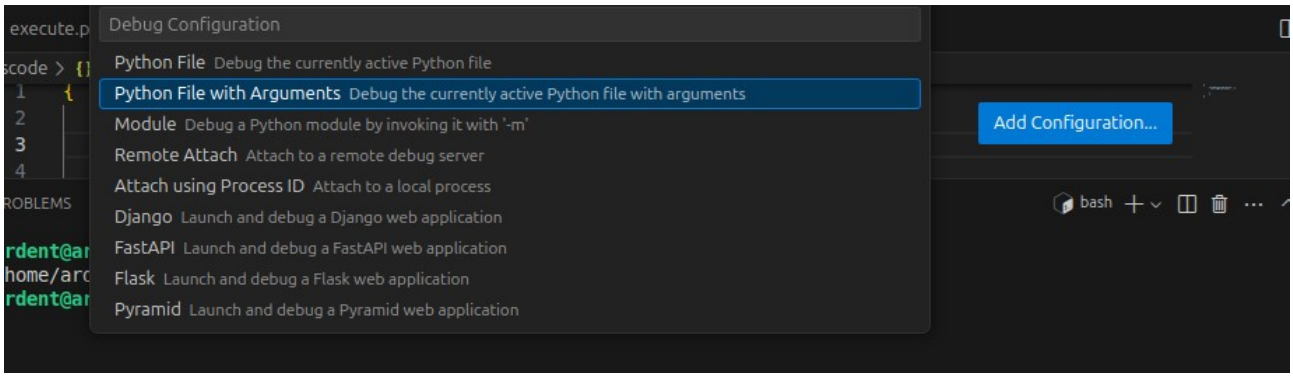


3. Inside the newly created .vscode directory create a file named launch.json



4. Click on Add Configuration button and then python debugger. Then select python file with arguments. If the add configuration or other options after that did not appear. You should finally type the config parameters shown in the next section





5. Modify the config parameter to point to your file and accommodate your needs

```
execute.py extract    execute.py transform    execute.py load    {} launch.json ×
scode > {} launch.json > JSON Language Features > [ ] configurations
1  {
2      "configurations": [
3          {
4              "name": "ETL: Extraction",
5              "type": "debugpy",
6              "request": "launch",
7              "program": "extract/execute.py",
8              "console": "integratedTerminal",
9              "python": "../pyspark/venv/bin/python",
10             "args": [
11                 "/home/ardent/Data/Extraction"
12             ]
13         }
14     ]
15 }
                                                    Add Configuration...
```

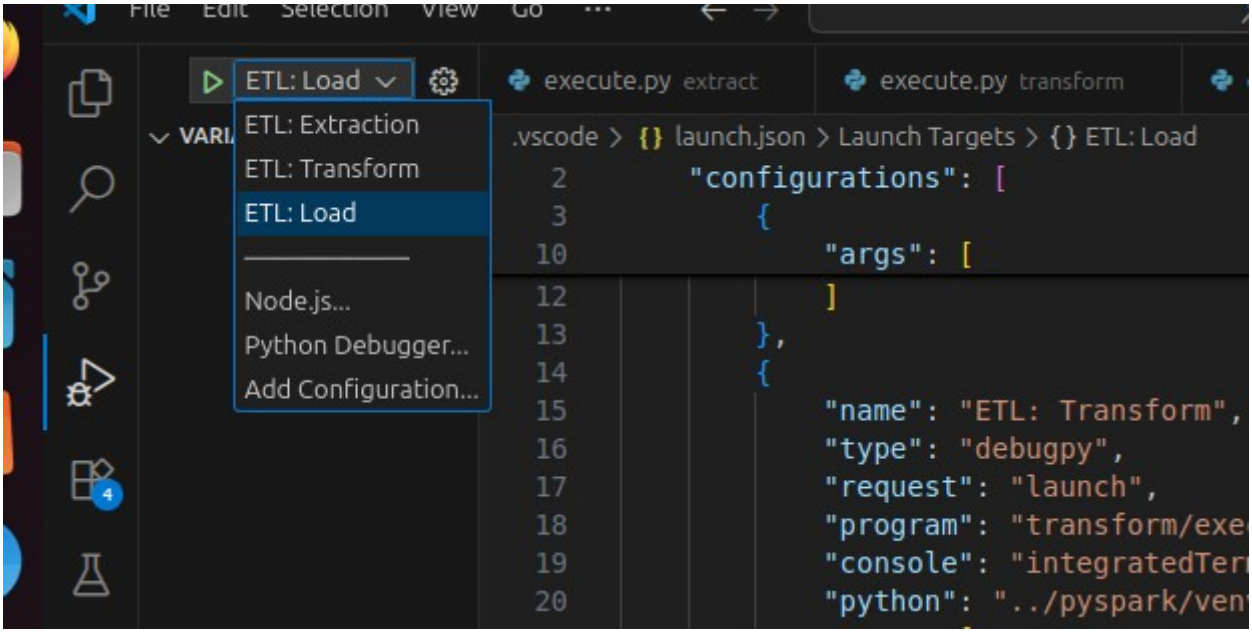
Here,
type parameter is set to debugpy to use the python debugger. Make sure you install it from extensions marketplace.
request parameter is set to launch to launch the program.
program parameter takes the path to the python file to run.
console is the terminal you want run the code on. We have selected integratedTerminal.
python parameter is the python executable to use to run the program. Note that I have used the relative path to provide the python from the pyspark virtual environment. Depending on use case using absolute path might be a better option
args parameter is the array of arguments to be passed to the program during execution
name is the name given to the specific configuration. You can add multiple configuration to the configurations list. **Try to do this for transform and load** before looking at below configuration. It is pretty straightforward

```
    {
        "args": [
        ]
    },
    {
        "name": "ETL: Transform",
        "type": "debugpy",
        "request": "launch",
        "program": "transform/execute.py",
        "console": "integratedTerminal",
        "python": "../pyspark/venv/bin/python",
        "args": [
            "/home/ardent/Data/Extraction",
            "/home/ardent/Data/Transform"
        ]
    },
    {
        "name": "ETL: Load",
        "type": "debugpy",
        "request": "launch",
        "program": "load/execute.py",
        "console": "integratedTerminal",
        "python": "../pyspark/venv/bin/python",
        "args": [
            "/home/ardent/Data/transform"
        ]
    }
```

Add Configurat

6. Switch to debugger tab by clicking on the debug icon at the left hand side

You should be able to see the name of the configurations you have set in launch.json file here. Now you can simply run the file by selecting the desire configuration and clicking the green button. This will run our code in debugger mode allow us to set breakpoint and access to watch window among many other features. Explore this on your own. This will be a valuable addition to your toolset



7. We need to modify our load module. Currently our load module contains sensitive information such as password in plain text. The code will eventually be pushed to github and it is very bad practice to expose password in plain text to public repositories.

There are several ways of solving this issue. The required parameters can be passed to the program during execution as an argument, or these can be set as environment variable, or several other password protector tools can be utilized. We will use the first approach as it is the easiest to implement in our current scenario.

7.1 Modify the entry point to take in username and password as input arguments:

```
if __name__ == "__main__":
    if len(sys.argv) != 4:
        print("Usage: python load/execute.py <input_dir> <pg_un> <pg_pw>")
        sys.exit(1)

    input_dir = sys.argv[1]
    pg_un = sys.argv[2]
    pg_pw = sys.argv[3]

    if not os.path.exists(input_dir):
```

Notice the len has been changed from 2 to 4 and two more variables are being stored

7.2 Modify the two function to take username and password as input

```
    input_dir = sys.argv[1]
    pg_un = sys.argv[2]
    pg_pw = sys.argv[3]

    if not os.path.exists(input_dir):
        print(f"Error: Input directory {input_dir} does not exist")
        sys.exit(1)

    spark = create_spark_session()
    create_postgres_tables(pg_un, pg_pw)
    load_to_postgres(spark, input_dir, pg_un, pg_pw)

    print("Load stage completed")
```

7.3 Modify the actual function definition to take it as input as well

```
        .getOrCreate()

def create_postgres_tables(pg_un, pg_pw):
    """Create PostgreSQL tables if they don't exist using psycopg2."""
    conn = None
    cursor = None

    try:
        conn = psycopg2.connect(
            dbname="postgres",
            user=pg_un,
            password=pg_pw,
            host="localhost",
            port="5432"
        )
        cursor = conn.cursor()

        create_table_queries = [
```

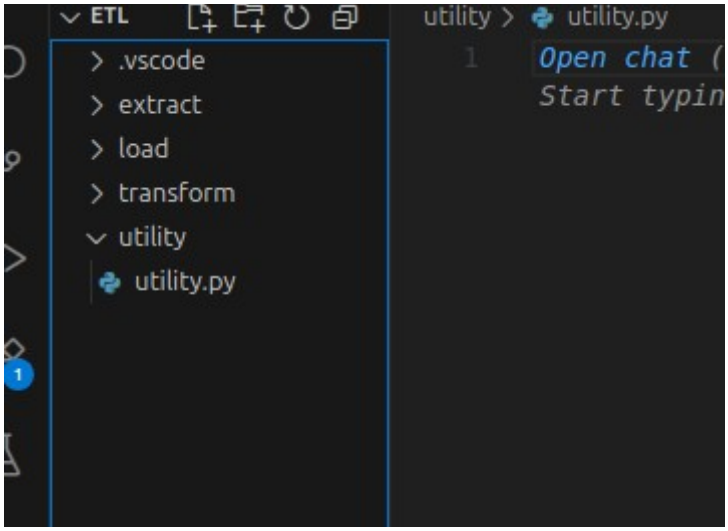
```python
            conn.close()

def load_to_postgres(spark, input_dir, pg_un, pg_pw):
    """Load Parquet files to PostgreSQL."""
    jdbc_url = "jdbc:postgresql://localhost:5432/postgres"
    connection_properties = {
        "user": pg_un,
        "password": pg_pw,
        "driver": "org.postgresql.Driver"
    }

    tables = [
        ("stage2/master_table", "master_table"),
        ("stage3/recommendations_exploded", "recommendations_exploded"),
        ("stage3/artist_track", "artist_track"),
        ("stage3/track_metadata", "track_metadata"),
        ("stage3/artist_metadata", "artist_metadata")
    ]
```

7.4 Finally, modify the launch.json file we created earlier to pass these two new input arguments

```python
            "python": "../pyspark/venv/bin/python",
            "args": [
                "/home/ardent/Data/Extraction", //input directory
                "/home/ardent/Data/Transform"  // output directory
            ]
        },
        {
            "name": "ETL: Load",
            "type": "debugpy",
            "request": "launch",
            "program": "load/execute.py",
            "console": "integratedTerminal",
            "python": "../pyspark/venv/bin/python",
            "args": [
                "/home/ardent/Data/transform", //input directory
                "postgres", //username
                "postgres" //password
            ]
        }
    ]
}
```

7.5 Pass the host, database, and port as an argument as well. Do this on your own

8. In the ETL directory. Create a new folder titled utility and create an new python file inside it titled utility.py



8.1 Let's setup logging functionality first. Create a function named setup_logger which will take the name of the log file and create instance of root logger. Then add two handlers. Handlers are responsible for handling the output of logs. We will setup one handler to output to stdout or terminal and the next handler to write to a log file. Logging is not limited to this. Learn about it in detail on your own

Do this in the newly create utility.py file

```
import logging

def setup_logging(log_file_name):

    logFormatter = logging.Formatter("%(asctime)s [%(threadName)-12.12s] [%(levelname)-5.5s]  %(message)s")
    rootLogger = logging.getLogger()
    rootLogger.setLevel(logging.INFO)


    fileHandler = logging.FileHandler(log_file_name)
    fileHandler.setFormatter(logFormatter)
    rootLogger.addHandler(fileHandler)

    consoleHandler = logging.StreamHandler()
    consoleHandler.setFormatter(logFormatter)
    rootLogger.addHandler(consoleHandler)
    return rootLogger
```

8.3 Import the logger in the execute.py file of load directory
Notice that the absolute path to the etl folder has been added to the system path before importing from the utility directory. This is required because the python has no idea where to search for the utility file from

```python
import os
import psycopg2
from pyspark.sql import SparkSession
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
from utility.utility import setup_logging

def create_spark_session(logger):
    """Initialize Spark session with PostgreSQL JDBC driver."""
    logger.debug("Initializing Spark Session with default parameters")
    return SparkSession.builder \
```

8.4 Let us modify the execute.py of load module to include the logger.

```python
load > execute.py > ...
def load_to_postgres(spark, input_dir, pg_un, pg_pw):
            print(f"Loaded {table_name} to PostgreSQL")
        except Exception as e:
            print(f"Error loading {table_name}: {e}")

if __name__ == "__main__":

    logger = setup_logging("load.log")

    if len(sys.argv) != 4:
        logger.error("Usage: python load/execute.py <input_dir> <pg_un> <pg_pw>")
        sys.exit(1)

    input_dir = sys.argv[1]
    pg_un = sys.argv[2]
    pg_pw = sys.argv[3]
```

8.5 Replace all instances of print using logger. For function make sure logger is passed as first argument and later used throughout it instead of print. There are 4 different methods of logger that we will use mostly. logger.info, logger.debug, logger.warning, logger.error Use these accordingly based on the type of message you are trying to log

```python
    pg_pw = sys.argv[5]

    if not os.path.exists(input_dir):
        logger.error(f"Error: Input directory {input_dir} does not exist")
        sys.exit(1)

    spark = create_spark_session(logger)
    create_postgres_tables(logger, pg_un, pg_pw)
    load_to_postgres(logger, spark, input_dir, pg_un, pg_pw)

    logger.info("Load stage completed")
```

Do the same for other functions and print statement as well.

```python
def create_spark_session(logger):
        .getOrCreate()

def create_postgres_tables(logger, pg_un, pg_pw):
    """Create PostgreSQL tables if they don't exist using psycopg2."""
    conn = None
    cursor = None

    try:
        conn = psycopg2.connect(
            dbname="postgres",
            user=pg_un,
            password=pg_pw,
            host="localhost",
            port="5432"
        )
        cursor = conn.cursor()

        logger.debug("Successfully connected to postgres database")

        create_table_queries = [
            """
```

```python
def create_postgres_tables(logger, pg_un, pg_pw):
                followers FLOAT,
                popularity INTEGER
            );
            """
        ]

        for query in create_table_queries:
            cursor.execute(query)
        conn.commit()
        logger.info("PostgreSQL tables created successfully")

    except Exception as e:
        logger.warning(f"Error creating tables: {e}")
    finally:
        logger.debug("Closing connection and cursor to postgres db")
        if cursor:
            cursor.close()
        if conn:
            conn.close()
```

```python
        conn.close()

def load_to_postgres(logger, spark, input_dir, pg_un, pg_pw):
    """Load Parquet files to PostgreSQL."""
    jdbc_url = "jdbc:postgresql://localhost:5432/postgres"
    connection_properties = {
```

```python
        ("stage3/artist_metadata", "artist_metadata")
    ]

    for parquet_path, table_name in tables:
        try:
            df = spark.read.parquet(os.path.join(input_dir, parquet_path))
            mode = "append" if 'master' in parquet_path else "overwrite"
            df.write \
                .mode(mode) \
                .jdbc(url=jdbc_url, table=table_name, properties=connection_properties)
            logger.info(f"Loaded {table_name} to PostgreSQL")
        except Exception as e:
            logger.warning(f"Error loading {table_name}: {e}")
```

9. Make similar changes for execute and transform module on your own

10. In the utility.py file add new function that will take seconds as an input and return formatted string with hours, minutes and seconds

utility > utility.py > format_time

```python
def setup_logging(log_file_name):
    consoleHandler.setFormatter(logFormatter)
    rootLogger.addHandler(consoleHandler)
    return rootLogger


def format_time(seconds):
    hours, remainder = divmod(seconds, 3600)
    minutes, seconds = divmod(remainder, 60)
    return f"{int(hours)} hours, {int(minutes)} minutes, {int(seconds)} seconds"
```

10.1 Import this newly created function in the execute.py of load module Also import the time module

load > execute.py > ...

```python
import time
import sys
import os
import psycopg2
from pyspark.sql import SparkSession
sys.path.append(os.path.abspath(os.path.join(os.path.dirname(__file__), '..')))
from utility.utility import setup_logging, format_time

def create_spark_session(logger):
    """Initialize Spark session with PostgreSQL JDBC driver."""
    logger.debug("Initializing Spark Session with default parameters")
    return SparkSession.builder \
```

10.2 In the entry point start recording time before execute the main logic and record the time after the code has been executed

```
        sys.exit(1)

    logger.info("Load stage started")
    start = time.time()

    spark = create_spark_session(logger)
    create_postgres_tables(logger, pg_un, pg_pw)
    load_to_postgres(logger, spark, input_dir, pg_un, pg_pw)

    end =time.time()
    logger.info("Load stage completed")
```

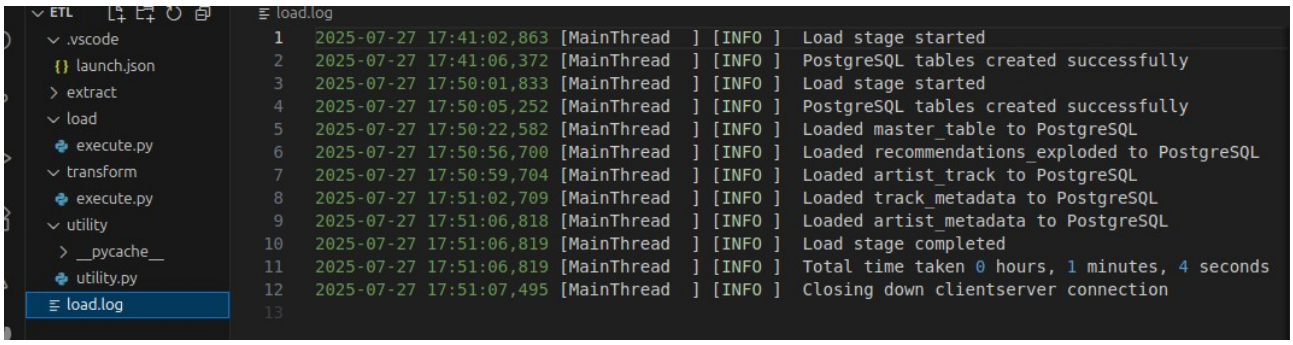
10.3 Finally pass the difference between these time to newly created function and log the output of that response

```
    logger.info("Load stage started")
    start = time.time()

    spark = create_spark_session(logger)
    create_postgres_tables(logger, pg_un, pg_pw)
    load_to_postgres(logger, spark, input_dir, pg_un, pg_pw)

    end =time.time()
    logger.info("Load stage completed")
    logger.info(f"Total time taken {format_time(end-start)}")
```

11. Do this for execute and transform module as well. Do this on your own. The process is exactly same as shown above

The below is the sample output after everything above has been implemented. Also a load.log file will be created in the etl directory. Similarly for execute and transform implement and verify it on your own

```
2025-07-27 17:50:05,252 [MainThread  ] [INFO ]  PostgreSQL tables created successfully
2025-07-27 17:50:22,582 [MainThread  ] [INFO ]  Loaded master_table to PostgreSQL
2025-07-27 17:50:56,700 [MainThread  ] [INFO ]  Loaded recommendations_exploded to PostgreSQL
2025-07-27 17:50:59,704 [MainThread  ] [INFO ]  Loaded artist_track to PostgreSQL
2025-07-27 17:51:02,709 [MainThread  ] [INFO ]  Loaded track_metadata to PostgreSQL
2025-07-27 17:51:06,818 [MainThread  ] [INFO ]  Loaded artist_metadata to PostgreSQL
2025-07-27 17:51:06,819 [MainThread  ] [INFO ]  Load stage completed
2025-07-27 17:51:06,819 [MainThread  ] [INFO ]  Total time taken 0 hours, 1 minutes, 4 seconds
2025-07-27 17:51:07,495 [MainThread  ] [INFO ]  Closing down clientserver connection
(venv) ardent@ardent:~/Workspace/etl$
```



Git related:

Gitignore Documentation: https://git-scm.com/docs/gitignore

Commit message convention:
https://gist.github.com/qoomon/5dfcdf8eec66a051ecd85625518cfd13

Example of .gitignore filel for python project:
https://github.com/github/gitignore/blob/main/Python.gitignore