Energy Consumption Forecasting Project – Venu Yerramsetti

This report provides a detailed, step-by-step explanation of an end-to-end energy consumption forecasting project. It covers the underlying principles, code execution, technological choices, and deployment strategies, making it accessible even to those new to data science, machine learning, and cloud platforms.

1. Project Purpose and Vision

The primary goal of this project is to build an **automated and reproducible pipeline for forecasting future energy consumption**. This allows stakeholders to make informed decisions regarding energy management, resource allocation, and strategic planning.

Key objectives include:

- **Accurate Forecasting**: Utilizing robust time-series models (ARIMA) to predict future energy needs.
- **Clear Visualization**: Presenting historical data and forecasts interactively using Power BI dashboards.
- **Scalability and Reproducibility**: Deploying the forecasting solution on Azure Machine Learning (Azure ML) to ensure consistent results and handle larger datasets.
- **Version Control**: Managing code changes and project evolution using Git and GitHub.
- **Operational Efficiency**: Automating the forecasting process for regular updates and insights.

2. Project Structure and Key Files

The project is organized to promote clarity and ease of use. Here's a breakdown of the main files and their roles:

File / Folder	Description

`energy_forecast_arima.py`	**The core Python script** for performing ARIMA modeling, generating forecasts, and saving outputs locally.
`run_forecast_job.py`	A slightly adapted version of the main script, specifically configured for execution within an **Azure ML Compute Instance**.
`requirements.txt`	Lists all necessary Python libraries (e.g., `pandas`, `pmdarima`, `matplotlib`) for setting up the project environment.
`energy_forecast.csv`	**Primary output CSV**: Contains the combined historical energy consumption data, the forecasted values, and their confidence intervals.
`forecast_only.csv`	A smaller CSV containing only the forecasted energy consumption values for future periods.
`original_energy_data.csv`	The cleaned and preprocessed version of the initial raw energy consumption dataset, ready for modeling.
`energy_forecast_plot.png`	A visual output: A line plot showing both historical energy consumption and the future forecast with confidence bounds.
`original_energy_consumption.png`	A plot showcasing only the historical energy consumption data for initial analysis.
`energy_forecast.pbix`	The **Power BI dashboard file**. This file can be opened in Power BI Desktop to interactively explore the forecast.
`dashboard_preview.png`	A screenshot providing a quick visual preview of the Power BI dashboard.
`job_logs/`	A directory specifically designated for storing **outputs and logs generated during Azure ML job runs**.

3. Understanding the Code Functionality

The project's intelligence resides within its Python scripts. Let's delve into what each script does and why.

3.1 energy_forecast_arima.py – The Local Workhorse

This is the central script you would run on your local machine. Workflow Explained (Block by Block / Line by Line:

Key Steps:

1. **CSV File Input (Raw Data)**
   You started with a CSV file that likely contains time-series energy consumption data. It uses European energy data (ENTSO-E). The key columns are usually:

Date	Energy Consumption
2023-01-01	2456
2023-01-02	2562
...	...

   ☑ **Date** column: Parsed as datetime, used for indexing
   ☑ **Consumption**: Daily/hourly energy values for forecasting
2. **Loading Raw Data:**

```
df = pd.read_csv("original_energy_data.csv", parse_dates=["Date"], index_col="Date")
```

   a. **Purpose**: This line reads your historical energy consumption data from a CSV file.
   b. **Why**: Data is the foundation of any forecasting. `pd.read_csv` from the `pandas` library is a standard and efficient way to load tabular data. `parse_dates=["Date"]` converts the 'Date' column into proper datetime objects, which is crucial for time-series analysis. `index_col="Date"` sets the 'Date' column as the DataFrame's index, making it a time-series DataFrame where each row is uniquely identified by its date.
3. **Data Preprocessing (Ensuring Time Series Continuity):**

```
df = df.asfreq('D') # Set daily frequency
df = df.fillna(method='ffill') # Fill missing values
```

   a. **Purpose**: These lines ensure the time series data is continuous and without gaps, which is a common requirement for ARIMA models.
   b. **Why**:

i. `df.asfreq('D')`: Resamples the time series to a daily frequency ('D'). If there were missing days, it would insert them with `NaN` values. This is important because ARIMA models assume regular time intervals.
ii. `df.fillna(method='ffill')`: Fills any `NaN` (missing) values that might have been introduced by `asfreq` or existed in the original data. `ffill` (forward-fill) means it uses the last valid observation to fill the current missing value. This is a common and often reasonable strategy for time-series data.

4. **Fitting the ARIMA Model:**

```
# Example using pmdarima.auto_arima for automatic order selection
from pmdarima import auto_arima
model = auto_arima(df['Energy_Consumption'],
        seasonal=True, m=7,
        trace=True,
        suppress_warnings=True,
        stepwise=True)
# m=7 for weekly seasonality
# Alternatively, if you know the order (p,d,q)
from statsmodels.tsa.arima.model import ARIMA
model = ARIMA(df['Energy_Consumption'], order=(p, d, q))
model_fit = model.fit()
```

a. **Purpose**: This is where the core forecasting logic happens. An ARIMA (AutoRegressive Integrated Moving Average) model is trained on the historical energy consumption data.
b. **Why**:
    i. **ARIMA**: A powerful statistical model for time series forecasting. It accounts for:
        1. **AR (AutoRegressive)**: Relationship between the current observation and a number of lagged observations.
        2. **I (Integrated)**: The use of differencing to make the time series stationary (constant mean, variance, and autocorrelation over time), which is a key assumption for ARIMA.
        3. **MA (Moving Average)**: Relationship between the current observation and a residual error from a moving average model applied to lagged observations.

ii. **`pmdarima.auto_arima`**: This function is chosen for its convenience. Instead of manually trying different combinations of p, d, and q (the orders for AR, I, and MA components, respectively), `auto_arima` automatically searches for the best ARIMA model parameters based on information criteria like AIC or BIC. `seasonal=True` and `m=7` indicate that the model should also consider weekly seasonality (e.g., energy consumption patterns might repeat every 7 days). The trace=True parameter for auto_arima is excellent as it provides verbose output during model selection, which is helpful for debugging and understanding the process.

iii. **`model_fit = model.fit()`**: Once the model structure is determined, this line trains the model using the historical data.

5. **Forecasting Future Values:**

```
forecast_steps = 30 # For example, forecast for the next 30 days
forecast_result = model_fit.predict(n_periods=forecast_steps,
return_conf_int=True)
forecast_values = forecast_result[0]
conf_int = forecast_result[1]
```

a. **Purpose**: Predicts forecast for next 30 days.

b. **Output example:**

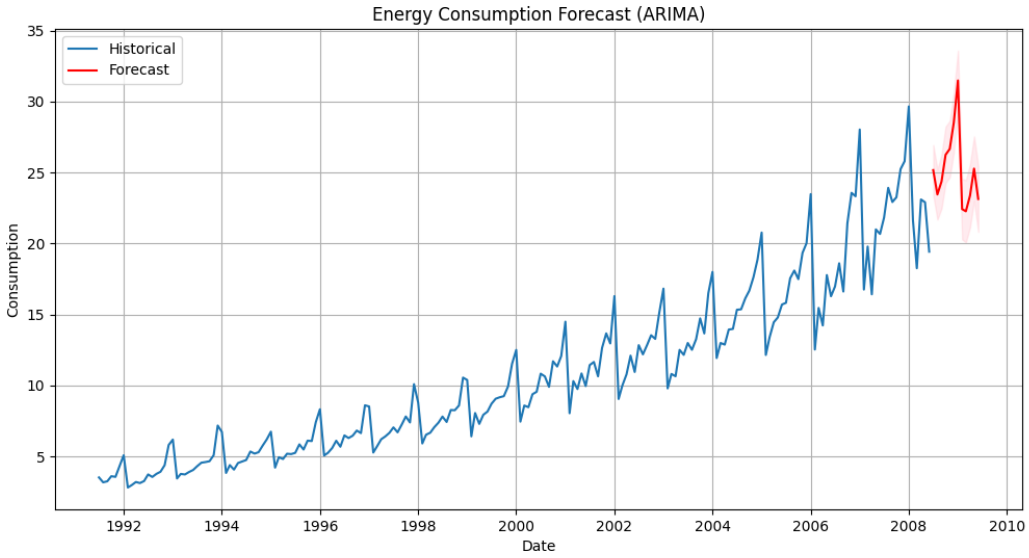
Date	Energy_Consumption Forecast
2024-06-01	2650
2024-06-02	2635
...	...

```
# Create a date range for the forecast
last_date = df.index[-1]
forecast_dates = pd.date_range(start=last_date + pd.Timedelta(days=1),
periods=forecast_steps, freq='D')
# Combine forecast with confidence intervals into a DataFrame
forecast_df = pd.DataFrame({'Date': forecast_dates,
                            'Forecast': forecast_values,
                            'Lower_CI': conf_int[:, 0],
                            'Upper_CI': conf_int[:, 1]})
forecast_df.set_index('Date', inplace=True)
```

c. **Purpose**: To generate predictions for future energy consumption based on the trained ARIMA model.
d. **Why**: This is the ultimate output of the forecasting model. `model_fit.predict()` uses the learned patterns to extrapolate values into the future. `n_periods` specifies how many steps into the future to forecast. `return_conf_int=True` is crucial for understanding the uncertainty around the forecast, providing lower and upper bounds for the predictions. The subsequent lines construct a DataFrame to hold these forecast values along with their associated dates and confidence intervals, making them easy to save and visualize.

6. **Plotting and Exporting Results:**

```
# Plot historical and forecast
plt.figure(figsize=(12, 6))
plt.plot(df['Energy_Consumption'], label='Historical Consumption')
plt.plot(forecast_df['Forecast'], label='Forecasted Consumption',
color='red')
plt.plot(forecast_df['Lower_CI'], color='orange', linestyle='--',
label='Lower Confidence Interval')
plt.plot(forecast_df['Upper_CI'], color='orange', linestyle='--',
label='Upper Confidence Interval')
plt.title('Energy Consumption Forecast')
plt.xlabel('Date')
plt.ylabel('Consumption')
plt.legend()
plt.grid(True)
plt.savefig('energy_forecast_plot.png')
```




```
# Save combined historical and forecast data to CSV
combined_df = pd.concat([df, forecast_df[['Forecast', 'Lower_CI',
'Upper_CI']]], axis=0)
combined_df.to_csv('energy_forecast.csv')

# Save only the forecast to CSV
forecast_df.to_csv('forecast_only.csv')
```

a. **Purpose**: To make the results tangible and consumable.
b. **Why**:
    i. **Plotting**: Visualizations are paramount for understanding trends and the model's performance. The `matplotlib` library is used to create a clear line plot, allowing easy comparison between historical data and the forecast. Including confidence intervals visually emphasizes the range within which the actual consumption is expected to fall.
    ii. **CSV Export**: Saving the data to CSV files (`energy_forecast.csv` and `forecast_only.csv`) ensures that the numerical results are readily available for further analysis, integration with other systems, or loading into visualization tools like Power BI. `energy_forecast.csv` is particularly useful as it merges historical and forecasted data, providing a complete picture.

3.2 `run_forecast_job.py` – Optimized for Azure ML

This script likely contains very similar logic to `energy_forecast_arima.py`, but with subtle adaptations for execution within an Azure Machine Learning environment.

Key Differences (and why):

- **Path Handling**: It might use Azure-specific paths for input data or ensure outputs are saved to the `job_logs/` directory, which is a standard location for job outputs in Azure ML. This makes it easy to retrieve artifacts from the Azure ML Studio UI.
- **Logging**: It might incorporate more verbose logging statements that are captured by Azure ML's logging system, allowing for better monitoring and debugging of jobs running in the cloud.
- **Input/Output Handling**: While the core logic remains the same, how data is passed into the script (e.g., as part of an Azure ML dataset) and how outputs are registered might be slightly different.

4. Local Execution Instructions

To get the project up and running on your local machine:

1. **Clone the Repository**:

```
git clone https://github.com/VenuYerramsetti/energy-forecast-arima.git
cd energy-forecast-arima
```

   a. **Why**: `git clone` downloads a copy of the entire project from GitHub to your local machine. `cd` navigates into the project directory.

2. **Install Dependencies**:

```
pip install -r requirements.txt
```

   a. **Why**: The `requirements.txt` file lists all the Python libraries (like `pandas`, `pmdarima`, `matplotlib`) needed for the scripts to run. `pip install -r` reads this file and installs all listed packages automatically. This ensures that everyone running the project has the exact same environment, preventing "it works on my machine" issues.

3. **Run the Forecast Script**:

```
python energy_forecast_arima.py
```

   a. **Why**: This command executes the main Python script, triggering the data loading, preprocessing, ARIMA modeling, forecasting, and output generation steps.
4. **View Outputs**: After successful execution, you will find:
   a. `energy_forecast.csv` and `forecast_only.csv` in the project's root directory.
   b. `energy_forecast_plot.png` and `original_energy_consumption.png` (plots) also in the root directory.
   c. Open `energy_forecast.pbix` in Power BI Desktop to see the interactive dashboard.

5. Azure Machine Learning Integration and Deployment

This section explains how to leverage Azure ML for scalable and reproducible execution of the forecasting pipeline.

5.1 Why Azure Machine Learning?

- **Scalability**: Run computationally intensive tasks on powerful cloud compute instances without taxing your local machine.
- **Reproducibility**: Define and register environments (Python versions, libraries) to ensure that your code always runs with the exact same dependencies, regardless of when or where it's executed.
- **Centralized Management**: Manage experiments, models, and deployments from a single portal (Azure ML Studio).
- **Automation**: Integrate with Azure Functions or Logic Apps for scheduled forecasting runs.
- **Collaboration**: Easily share workspaces and jobs with team members.

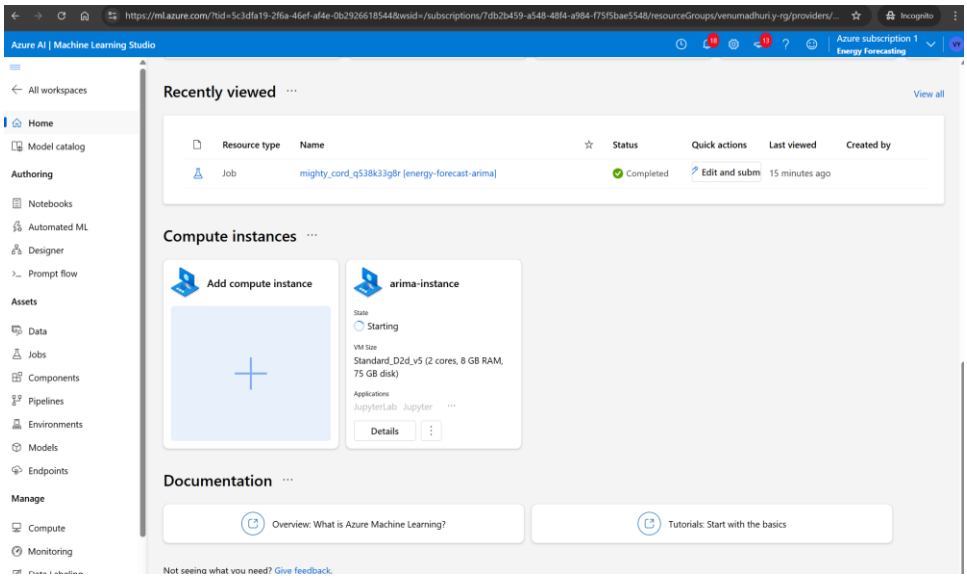
5.2 Azure ML Setup (Step-by-Step)

1. **Create Azure ML Workspace**:
   a. Go to Azure Portal -> Search for "Machine Learning" -> Create new Machine Learning Workspace.
   b. **Value/Action**: `energy-ml-ws.` This is your central hub for all ML activities in Azure.

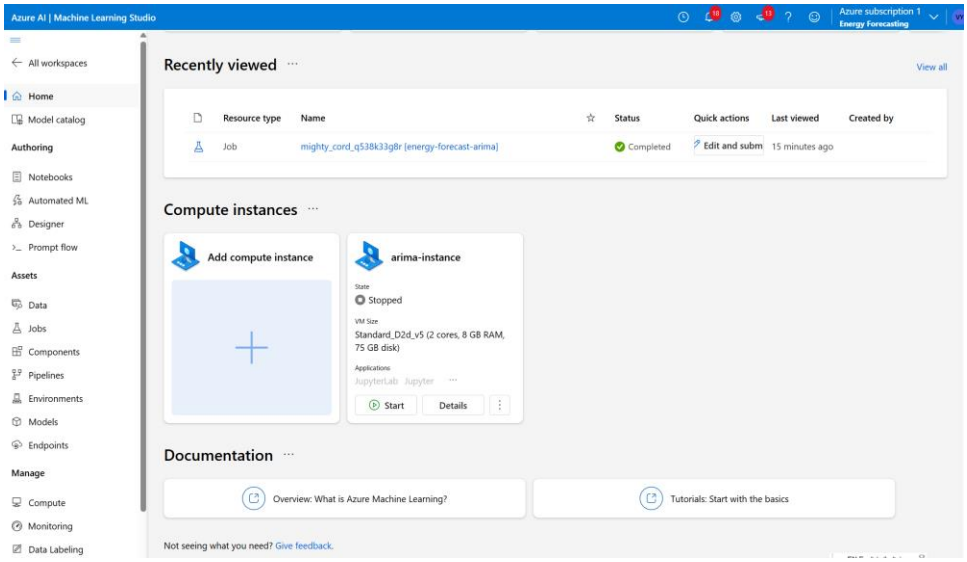
2. **Create Compute Instance**:
    a. In Azure ML Studio -> Compute -> Compute Instances -> New.
    b. **Value/Action**: Name it `arima-instance`. Select a CPU VM size like `Standard_DS11_v2`.
    c. **Why**: A compute instance is a cloud-based development environment (a virtual machine) pre-configured with data science tools. It's where you'll run your scripts directly or launch jobs. `Standard_DS11_v2` is a good general-purpose CPU VM suitable for this task.
3. **Set Up Environment**:
    a. You can create a custom environment within Azure ML Studio or via the terminal.
    b. **Value/Action**: Name it `arima-env`. Use Python 3.10 as the base image.
    c. **Why**: An environment specifies the Python version and all required libraries (`pmdarima`, `pandas`, `matplotlib`, etc.). Registering this environment ensures that your script runs with the exact same dependencies every time, guaranteeing reproducibility.

```
# From the terminal within your Azure ML Compute Instance (Jupyter or SSH)
conda create -n arima-env python=3.10
conda activate arima-env
pip install -r requirements.txt # Assuming requirements.txt is uploaded.
```

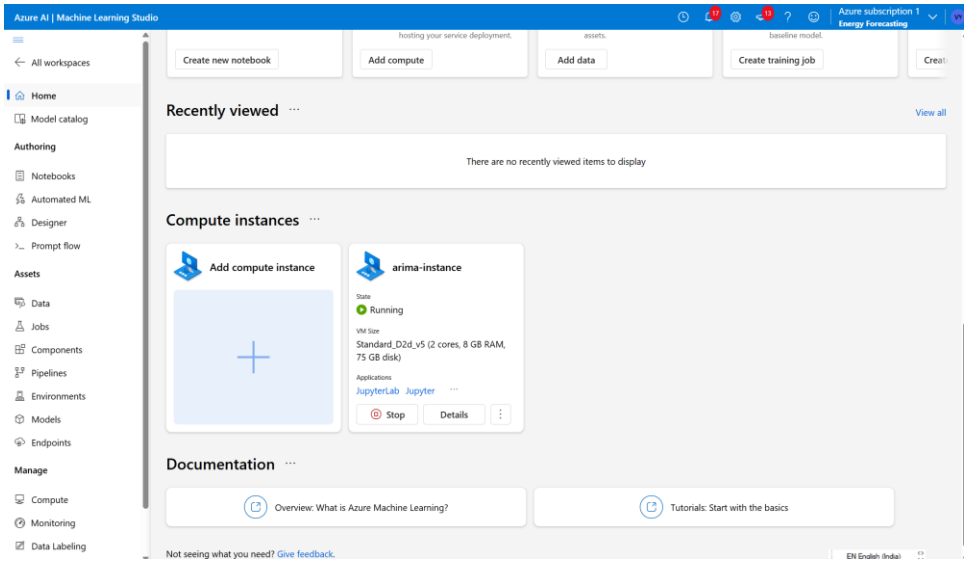
    d. If you're using Azure ML SDK for job submission, you would register this environment explicitly.



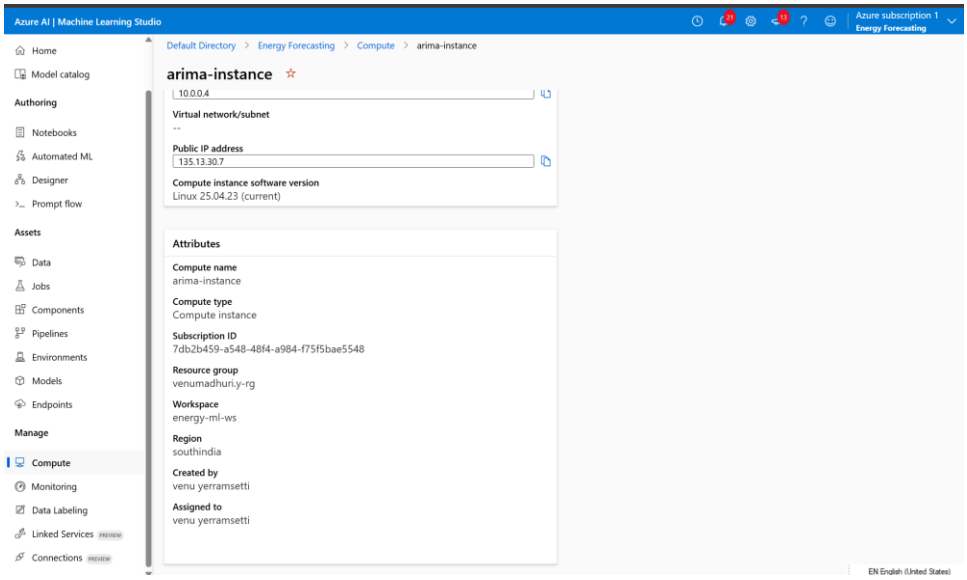
Compute Instance: arima-instance, Starting



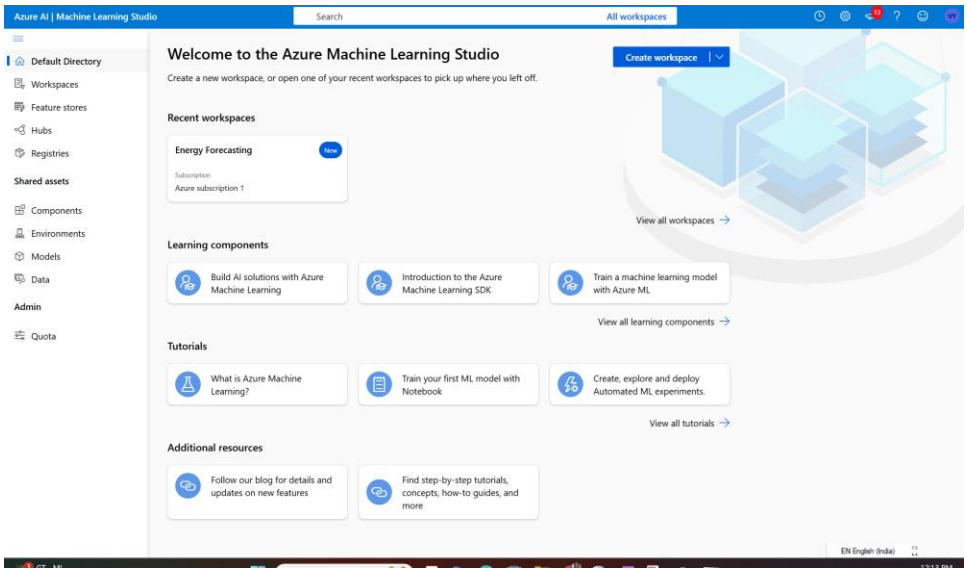
Compute Instance: arima-instance, Stopped



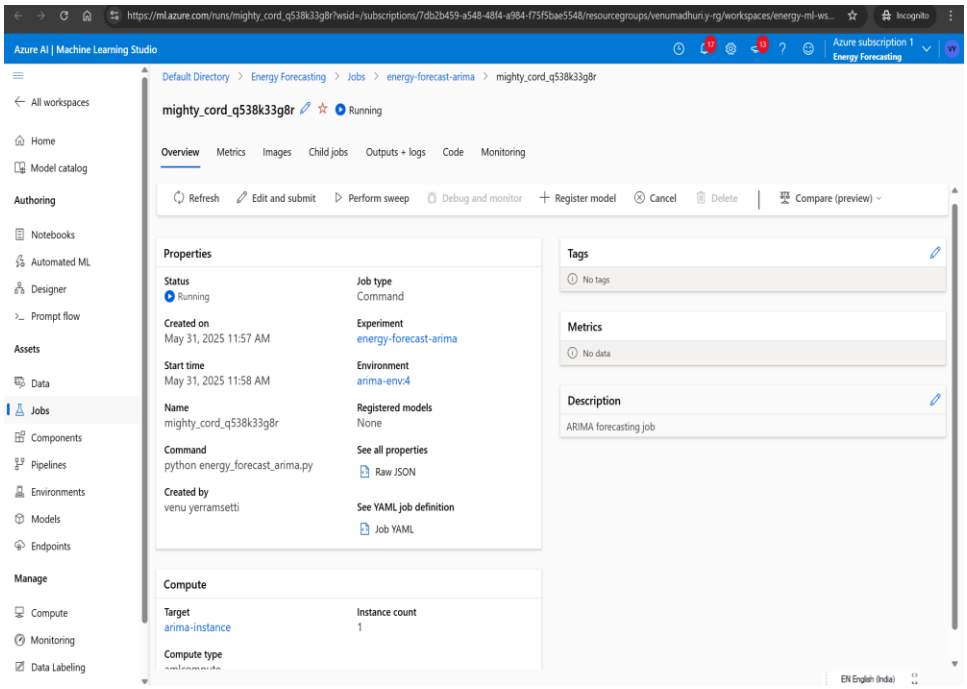
Compute Instance: arima-instance, Running



Compute Instance: arima-instance, attributes



Workspace: Energy Forecasting



5.3 Running the Forecast Job on Azure ML

You have two main ways to execute your script on Azure ML:

1. **Terminal Execution on Azure Compute (Recommended for initial testing/development)**:
    a. SSH into your `arima-instance` or open the Jupyter terminal in Azure ML Studio.
    b. Navigate to your project directory (assuming you've uploaded your code or cloned the repo) :`cd /mnt/batch/tasks/shared/LS_root/mounts/clusters/arima-instance/code/energy-forecast-arima`
    c. Run the Azure-specific script: `python run_forecast_job.py`
    d. **What happens**: This command executes the `run_forecast_job.py` script directly on your compute instance. It will generate forecasts, save plots and CSVs within the `job_logs/` directory, and log its progress to the terminal (and typically to 70_driver_log.txt in Azure ML job logs).
2. **Job Submission (Optional Advanced, for MLOps pipelines)**:
    a. For more automated and production-ready workflows, you would define and submit the script as an Azure ML job using the Azure ML SDK.

b. **Why**: This method is ideal for creating robust MLOps pipelines, where jobs can be triggered automatically, parameters can be easily managed, and full tracking of experiments is desired. The provided snippet shows the Python code using `azure.ai.ml` client to define and submit a `CommandJob`.

3. **Azure ML Integration (Environment Registration)**:
   You registered the model environment to Azure ML:

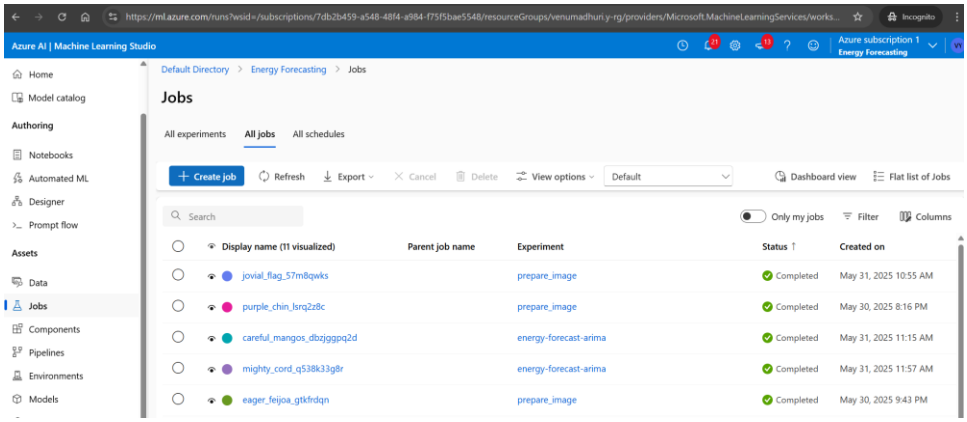
5.4 Register_environment.py (example):

```
from azure.ai.ml import MLClient
from azure.identity import DefaultAzureCredential
from azure.ai.ml.entities import Environment

ml_client = MLClient(DefaultAzureCredential(), subscription_id,
resource_group, workspace)

arima_env = Environment(name="arima-env",
image="mcr.microsoft.com/azureml/base:latest", conda_file="conda.yml")
ml_client.environments.create_or_update(arima_env)
```

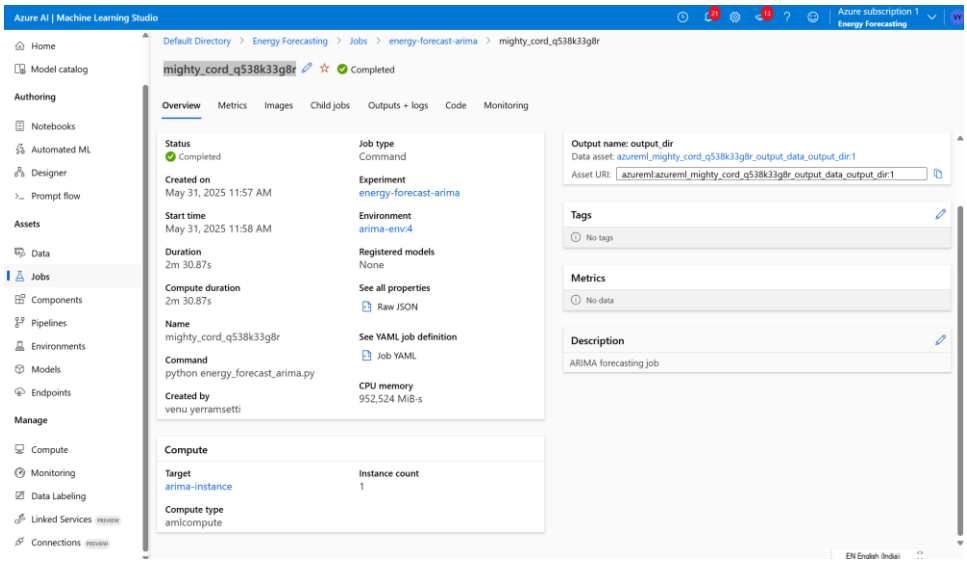
- Ensures reproducibility
- Makes model available as a service or pipeline

5.5 Getting Output Plots and Logs from Azure ML

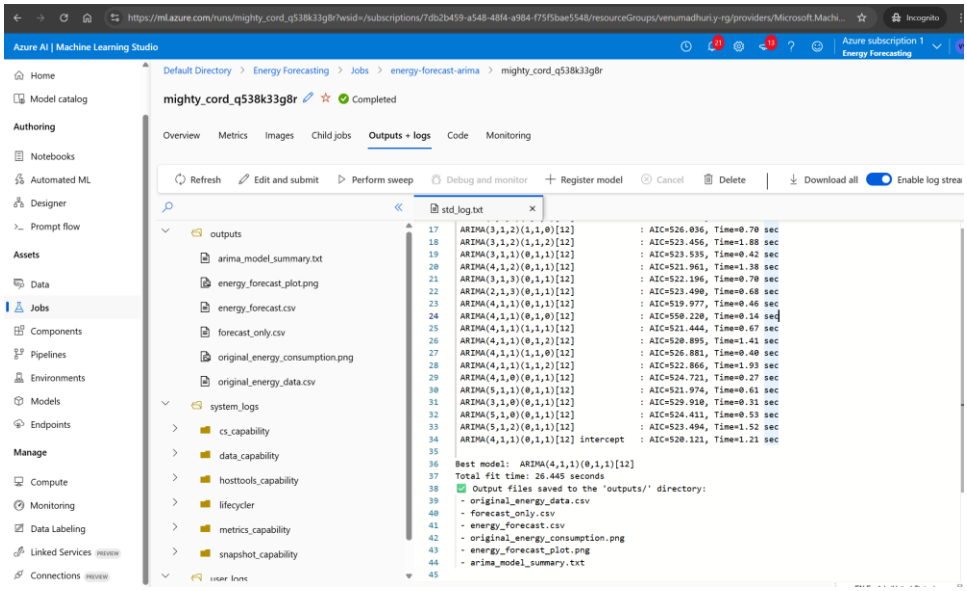
- **Azure ML Studio UI**: Navigate to "Jobs" in your Azure ML workspace. Click on your specific job run (e.g., mighty_cord_q538k33g8r). Go to the "Outputs + Logs" tab. Here, you will find `energy_forecast_plot.png` and other generated files under the `job_logs/` directory. You can view or download them directly.



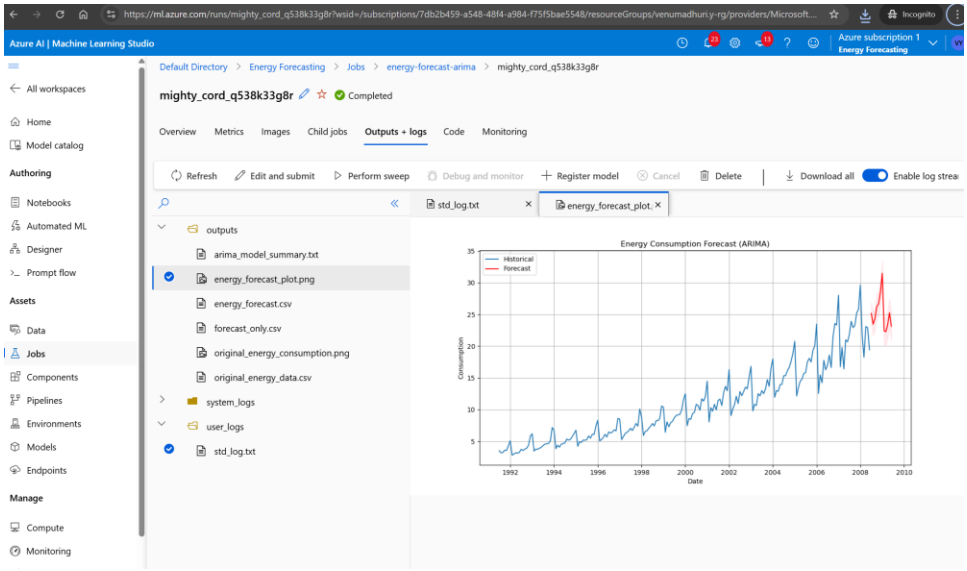
"Jobs" in your Azure ML workspace



Job: mighty_cord_q538k33g8r **in your Azure ML workspace**

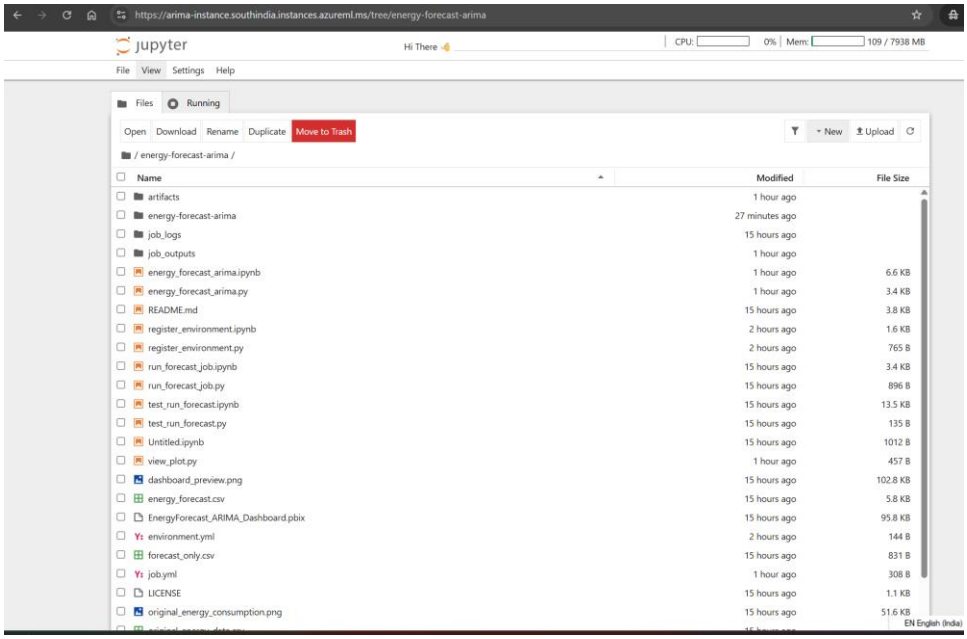


Job: mighty_cord_q538k33g8r, Outputs + Logs **in your Azure ML workspace**

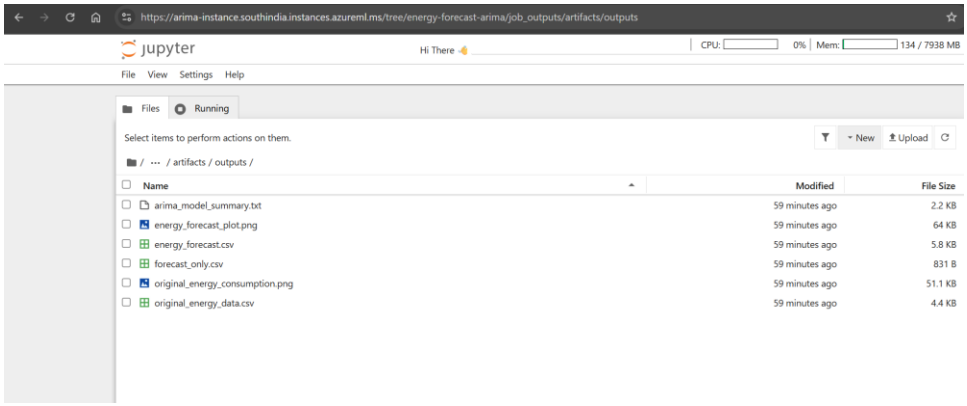


Job: mighty_cord_q538k33g8r, Outputs + Logs >> energy_forecast_plot.png **in your Azure ML workspace**

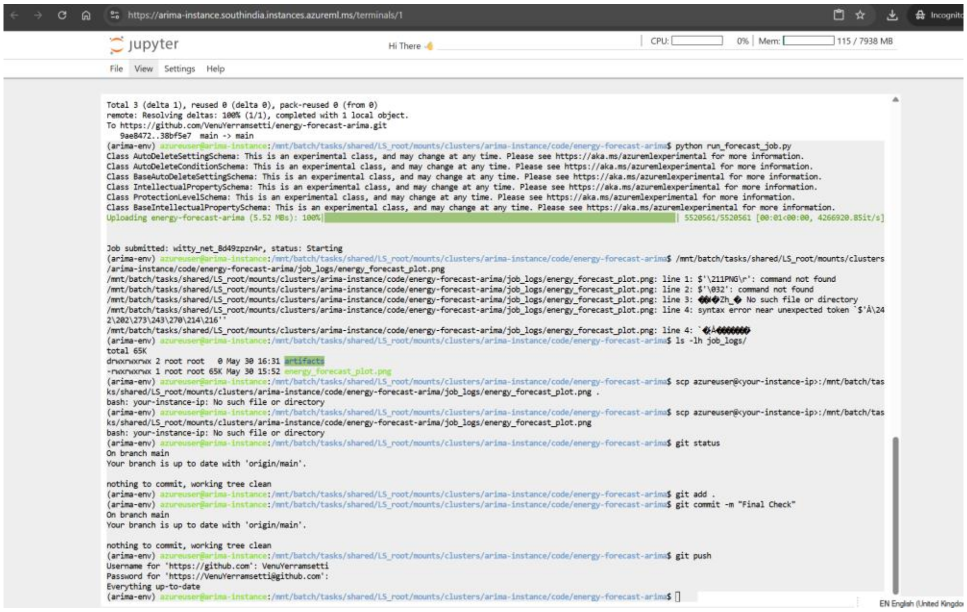
- **Compute Instance Terminal**: If you ran the script directly on the compute instance, you can access the `job_logs/` directory via Jupyter or SSH and download the files.



`job_logs/` **directory via Jupyter in your Azure ML workspace**



`job_logs >> Outputs` **via Jupyter in your Azure ML workspace**



`job_logs/` **directory via Jupyter Terminal in your Azure ML workspace**

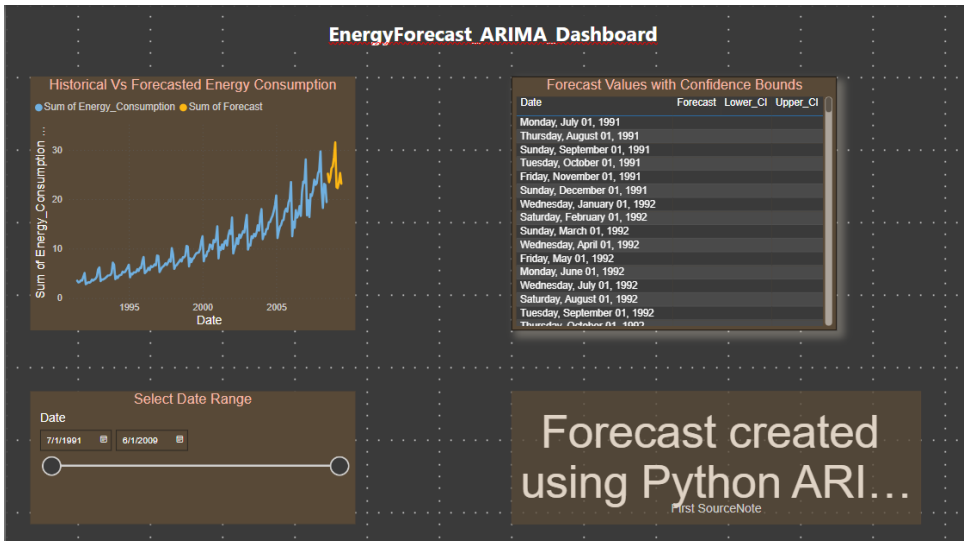
`View_plot.py` **execution via Jupyter Terminal in your Azure ML workspace**

6. Power BI Dashboard Integration

Power BI is used to create interactive and visually appealing dashboards from the forecast data.

6.1 Steps to Use Power BI:

1. **Open the Power BI file**: Double-click `energy_forecast.pbix` (assuming you have Power BI Desktop installed).
2. **Update Data Source (if needed)**: Power BI dashboards connect to data sources. If you've run the script locally and the `energy_forecast.csv` file is in the same directory, Power BI should automatically connect. If you've downloaded the CSV from Azure ML or moved it, you might need to update the data source path within Power BI Desktop (File -> Options and settings -> Data source settings -> Change Source).
3. **Explore the Dashboard**: The dashboard is pre-configured with:
    a. **Line Chart**: Displaying both historical energy consumption and the future forecast, often with confidence intervals. This provides an immediate visual summary.
    b. **Date Slicer**: Allows users to filter the data by specific date ranges, enabling focused analysis.
    c. **Table**: Presents the numerical forecast values, including lower and upper confidence intervals, for detailed inspection.



Power BI Dashboard

6.2 Why Power BI?

- **Interactive Visualization**: Allows non-technical stakeholders to explore data without writing code.

- **Business Insights**: Helps in understanding trends, patterns, and future predictions at a glance.
- **Data Connectivity**: Can connect to various data sources, including local CSVs, databases, and cloud storage (like Azure Blob Storage, as mentioned in future plans).
- **Reporting**: Easy to create and share professional reports.

7. Version Control with Git and GitHub

Git and GitHub are indispensable for managing code and collaborating on projects.

7.1 Git Workflow Explained:

1. **Initialize/Clone**: You start by either cloning an existing repository (`git clone`) or initializing a new one (`git init`).
2. **Make Changes**: You modify your code, add new files, etc.
3. **Stage Changes (`git add .`)**:
    a. **Why**: This command tells Git to start "tracking" the changes you've made. It prepares your modified files to be included in the next commit, means stage all changes in the current directory and subdirectories.
4. **Commit Changes (`git commit -m "Your clear message"`)**:
    a. **Why**: A commit is like taking a snapshot of your project at a specific point in time. The `-m` flag allows you to add a **clear and concise message** describing what changes you made in that commit. Good commit messages are crucial for understanding the project's history.
5. **Pull Latest Changes (`git pull origin main`)**:
    a. **Why**: Before pushing your changes, it's good practice to `pull` the latest changes from the remote repository (GitHub, in this case). This prevents merge conflicts if others have pushed changes while you were working. `origin` refers to the remote repository, and `main` is the branch name (often the primary branch).
6. **Push Changes (`git push origin main`)**:
    a. **Why**: This command uploads your committed changes from your local repository to the remote repository on GitHub, making them accessible to others and serving as a backup.

7.2 Why Git and GitHub?

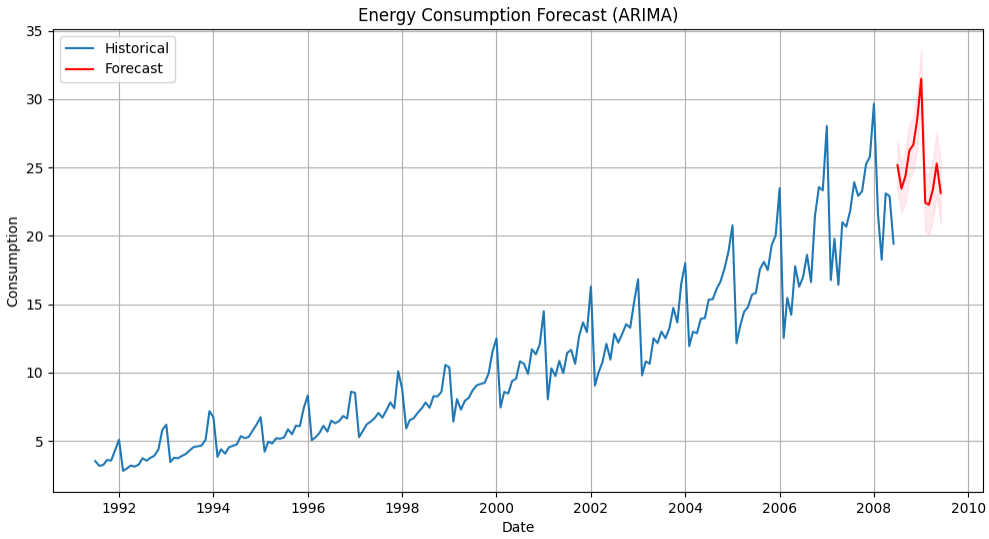
- **Version Tracking**: Keeps a complete history of every change, allowing you to revert to previous versions if needed.
- **Collaboration**: Enables multiple developers to work on the same project simultaneously without overwriting each other's work.
- **Backup**: Your code is stored securely in the cloud.
- **Project Management**: Issues, pull requests, and project boards help manage development tasks.
- **Open Source**: Facilitates sharing and contributing to open-source projects.

8. Testing and Validation

Ensuring the forecast is accurate and the pipeline is robust is crucial.

8.1 Validating Forecast Locally:

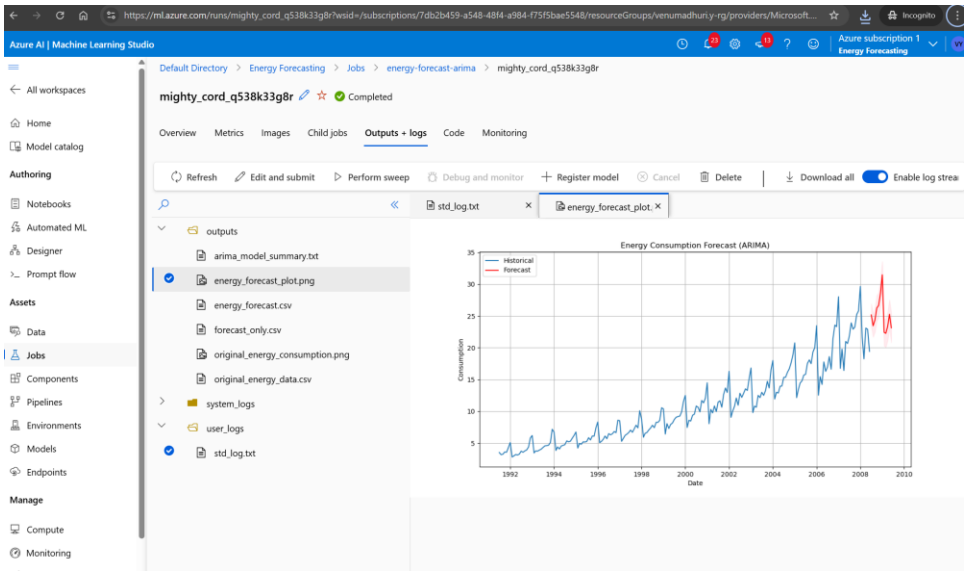
- **`energy_forecast_plot.png`**: Visually inspect this plot. The historical data (blue line) should transition smoothly into the forecast (red line). The confidence intervals (orange dashed lines) should expand as the forecast extends further into the future, reflecting increased uncertainty.



- **`forecast_only.csv`**: Review the numerical predictions. Do they seem reasonable given the historical trends?

8.2 Validating in Azure:

- **Azure ML Studio Job Logs**: After a job runs in Azure ML, always check the "Outputs + Logs" tab for your job run.
    - Confirm that `energy_forecast_plot.png` is present, indicating the script ran successfully and generated the plot.
    - Review `70_driver_log.txt` for any errors or warnings during execution.



Azure ML Outputs + Logs

- **Download and Verify**: Download `energy_forecast.csv` from Azure ML and open it in Power BI or a spreadsheet program to confirm the data integrity.

9. What Each Output Means

File Name	Description
`energy_forecast.csv`	This is the most comprehensive output. It contains the **entire time series**: your original historical energy consumption data, concatenated with the newly generated future forecast values. Crucially, it also includes the **lower and upper confidence intervals** for the forecast, providing a range of likely outcomes rather

	than just a single point estimate. This file is directly consumed by the Power BI dashboard.
`forecast_only.csv`	A streamlined version of the forecast, containing just the **future predicted energy consumption values** for the specified number of steps (e.g., 30 days), typically along with the dates and confidence intervals. This is useful if you only need the future predictions and don't require the historical context within the same file.
`energy_forecast_plot.png`	A visual summary. This PNG image displays a **line chart** where the blue line represents the historical energy consumption, and the red line represents the forecasted consumption. The shaded or dashed orange lines around the forecast indicate the confidence intervals, illustrating the model's uncertainty. This plot offers a quick way to assess the trend and quality of the forecast.
`original_energy_data.csv`	This is the **cleaned and formatted version of your input dataset**. It's the data that was actually fed into the ARIMA model after any preprocessing steps (like handling missing values or setting frequency). It serves as a record of the exact historical data used for training.
`energy_forecast.pbix`	The **Power BI Desktop file**. This is the interactive dashboard that brings all the data to life. It contains pre-built visualizations (like line charts, tables, and slicers) that allow users to explore the historical and forecasted energy consumption, filter by date, and see numerical details including confidence intervals. It's the primary tool for communicating insights to non-technical audiences.
`job_logs/`	This is a **directory used by Azure Machine Learning** to store all outputs and logs generated by a specific job run. When your `run_forecast_job.py` script executes on an Azure compute instance, any files it saves (like `energy_forecast_plot.png` or output CSVs) are typically directed here. It also contains system logs (like

	`70_driver_log.txt`) that provide details about the job's execution, which are vital for debugging and monitoring cloud runs.

🔍 Where to Find What in the GitHub Repo

- **Main logic**: `energy_forecast_arima.py`
- **Azure version**: `run_forecast_job.py`
- **Power BI**: `energy_forecast.pbix`
- **Forecast CSVs**: root folder after script runs
- **Visual Preview**: `dashboard_preview.png`
- **Setup info**: `README.md`
- **Requirements**: `requirements.txt`

🧠 Final Outcome and Future Enhancements

This project successfully delivers:

- A fully functional **time series forecasting solution** using ARIMA.
- **Forecasted 30 days** of energy consumption
- **Visualizations** (plots and Power BI dashboard) for intuitive understanding.
- **Reproducible environments** via `requirements.txt` and Azure ML.
- **Cloud deployment capabilities** for scalability and automation.
- **Version control** for robust development.

Future Considerations for Enhanced Automation and Connectivity:

- **Azure Blob Storage Integration**: The project hints at uploading `energy_forecast.csv` to Azure Blob Storage. This is a critical step for true automation. Power BI can directly connect to Azure Blob Storage, and with an Azure Power BI Gateway or service principal, the dashboard can **auto-refresh** with the latest forecasts whenever the CSV is updated in Blob Storage.
- **Azure Functions/Logic Apps**: To automate the entire pipeline, you could use:
    - **Azure Functions**: To trigger the `run_forecast_job.py` script on a schedule (e.g., daily or weekly).
    - **Azure Logic Apps**: To orchestrate complex workflows, such as:

        - Triggering the forecasting job.
        - Uploading the output CSV to Blob Storage.
        - Notifying stakeholders of new forecasts.
- **Azure ML Pipelines**: For more complex machine learning workflows involving multiple steps (data ingestion, preprocessing, modeling, evaluation, deployment), using Azure ML Pipelines would provide end-to-end orchestration and lineage tracking.