Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica
IE-0499 – Proyecto Eléctrico

II Ciclo 2025

Sistema de análisis y predicción de series de tiempo usando agentes de inteligencia artificial basados en LLM

Erick Vargas Monge C08215
Grupo 03

Profesores:
Marvin Coto Jiménez, PhD (Profesor del proyecto)
Taina Ramírez Cortés (Profesor guía)

24 de noviembre, 2025

Índice

Índice de figuras

1. Resumen

Este proyecto aborda el diseño e implementación de un sistema automatizado para el análisis y predicción de series de tiempo de consumo energético nacional, empleando agentes de inteligencia artificial basados en modelos de lenguaje (LLMs). El sistema se desarrolla utilizando exclusivamente Python y Ollama como motor local de inferencia, sin requerir GPU ni acceso a servicios en la nube. La solución se inspira en el enfoque propuesto por Mauro Di Pietro y permite procesar conjuntos de datos reales, generar reportes en lenguaje natural y construir predicciones de demanda eléctrica mediante modelos estadísticos y herramientas abiertas. El trabajo se realizó en base al enunciado y objetivos propuestos por el profesor Marvin Coto [1]. La comunicación se realizó mediante la plataforma de Teams en conjunto con reuniones presenciales cada cierto tiempo. Las clases teóricas del proyecto eléctrico con la Profesora Taina Ramírez fue muy útil para realizar correctamente las presentaciones, resúmenes y la bitácora.

2. Objetivo General

Implementar un sistema con agentes de inteligencia artificial capaces de analizar y predecir series de tiempo de consumo energético utilizando modelos locales y herramientas de código abierto.

2.1. Objetivos específicos

- Adaptar el sistema propuesto por Mauro Di Pietro a un conjunto de datos energéticos reales del país.
- Implementar un agente que pueda ejecutar instrucciones en lenguaje natural para analizar series de tiempo.
- Construir un módulo de predicción basado en regresores simples (SARIMA, Prophet o modelos basados en pandas).
- Evaluar la precisión de las predicciones generadas y su utilidad práctica.
- Documentar el sistema en un repositorio con instrucciones para replicarlo localmente sin conexión a la nube.

3. Estructura y Lógica del Sistema

El proyecto se fundamenta en la arquitectura de agentes propuesta por Mauro Di Pietro, la cual permite que un modelo de lenguaje opere no solo como generador de texto, sino como un sistema capaz de ejecutar código, analizar series de tiempo y manipular dataframes de manera autónoma [2]. En este enfoque, el modelo deja de ser un asistente pasivo y se transforma en un agente que razona, selecciona herramientas, planifica acciones y produce resultados verificables. Todo esto se ejecuta localmente mediante Ollama, sin necesidad de GPU ni servicios en la nube, lo que garantiza privacidad y reproducibilidad.

3.1. LLMs como agentes autónomos

Un modelo de lenguaje tradicional genera texto prediciendo tokens basados en patrones lingüísticos. Sin embargo, cuando se integra dentro de un agente, su comportamiento cambia profundamente. Un *Agente IA* combina tres componentes esenciales:

1. **Razonamiento en lenguaje natural**: el LLM interpreta la instrucción del usuario, detecta la intención y evalúa si necesita ejecutar código, graficar, filtrar datos o simplemente responder con texto.

2. **Selección de herramientas**: el agente decide si debe activar funciones externas (herramientas) que amplían sus capacidades. Este mecanismo se basa en el estándar de "tool calling" incorporado en Ollama.

3. **Ejecución iterativa**: el agente alterna entre razonamiento y acción hasta producir un resultado final coherente, verificable y completo.

Este comportamiento iterativo de razonamiento, decisión, ejecución, evaluación y respuesta; replica el diseño descrito por Di Pietro [2] y permite que el sistema resuelva tareas que requieren cálculos reales sobre datos, algo que un LLM por sí solo no puede hacer.

3.2. Interacción entre el LLM y las herramientas del agente

El modelo recibe un mensaje del usuario y genera una salida estructurada indicando si necesita activar una herramienta. Esta salida puede contener un bloque `tool_calls`, donde especifica:

- el nombre de la herramienta,
- los argumentos requeridos,
- y el propósito de la llamada.

El sistema intercepta este bloque y ejecuta la herramienta correspondiente. Luego, el resultado (texto, valores numéricos, tablas o gráficos) se devuelve al LLM, que lo incorpora en su razonamiento para decidir si debe ejecutar otra acción o finalizar la respuesta mediante la herramienta `final_answer`. De esta forma, el LLM actúa como un planificador de alto nivel, mientras que las herramientas ejecutan las operaciones de bajo nivel sobre datos reales.

3.3. Carga y estructura del dataset

El usuario puede cargar un archivo de datos, por ejemplo `datos_limpios.csv`, directamente en memoria. Se identifica de forma automática la columna temporal mediante heurísticas sobre el nombre (`fecha`, `date`, `hora`), se convierte a formato `datetime`, se define como índice y se fija la frecuencia a intervalos regulares de 15 minutos. En esta etapa el sistema garantiza que:

- el índice temporal sea uniforme,
- no existan valores desordenados,
- y la estructura esté lista para series de tiempo.

El dataset, una vez cargado, queda accesible como la variable global `dtf`, utilizada por todas las herramientas del agente.

3.4. Herramientas del agente

El agente pone a disposición cuatro herramientas principales. Cada herramienta amplía las capacidades del LLM y permite ejecutar tareas que no podrían resolverse solo mediante texto.

1. Herramienta `final_answer`

Esta herramienta finaliza el ciclo de razonamiento del agente. Su propósito es generar la respuesta en lenguaje natural una vez que todas las operaciones necesarias han sido ejecutadas. Su diseño es simple, pero fundamental para detener el flujo iterativo del agente.

2. Herramienta `code_exec`

Esta herramienta permite al agente ejecutar código Python real. A diferencia del razonamiento textual, este código altera o inspecciona directamente el dataframe `dtf`. La herramienta captura la salida estándar y retorna cualquier cálculo solicitado. Además, debido a la tendencia de los modelos pequeños a generar código incompleto o malformado, se implementaron sanitizadores y validadores que corrigen:

- desbalance de paréntesis,
- comillas faltantes,
- referencias erróneas como `df` en lugar de `dtf`,
- formatos inválidos de `print()`.

Estas correcciones fueron necesarias para garantizar que el agente no falle durante la ejecución de cálculos estadísticos o filtrado de datos. Esto fue útil especialmente al inicio donde se estaba trabajando con un modelo pequeño.

3. Herramienta `plot_data`

Esta herramienta genera gráficos directamente desde el dataframe. El agente puede:

- graficar series completas,
- mostrar curvas en rangos definidos por fecha,
- visualizar columnas específicas como MW o MW_P,

El LLM solicita esta herramienta cuando identifica palabras clave como "gráfico", "visualiza", "trend" o "plot", siguiendo las reglas del sistema.

4. Herramienta `predict_data`

Esta herramienta implementa predicciones mediante dos regresores especiales: Prophet y ARIMA. En el proyecto se utilizan únicamente para estimar valores futuros de potencia eléctrica. La herramienta:

- obtiene la columna a predecir,
- define un horizonte en días o mediante una fecha objetivo,
- ajusta el modelo estadístico correspondiente,
- genera las predicciones,
- produce un gráfico,
- y guarda los resultados en un archivo CSV.

El agente selecciona esta herramienta cuando detecta instrucciones relacionadas con "predecir", "forecast", "próximos días" o "estimar valores futuros".

3.5. Mecanismo de ejecución: función `use_tool`

La función `use_tool` interpreta la salida producida por el modelo. Puede recibir:

1. **tool_calls formales**: estructuras JSON generadas por Ollama.
2. **JSON plano dentro de `content`**: generado cuando el modelo no usa tool-calling correctamente.
3. **Texto sin herramientas**: típico cuando la instrucción no requiere cómputo.

La función ejecuta la herramienta correspondiente, captura resultados y los reenvía al agente para continuar su razonamiento.

3.6. Ciclo del agente: función `run_agent`

Esta función implementa el flujo completo del agente siguiendo el esquema iterativo. En cada ciclo:

1. el agente recibe la consulta del usuario;
2. el LLM decide si debe usar una herramienta;
3. la herramienta se ejecuta y produce un resultado;
4. el resultado se incorpora como nueva entrada al LLM;
5. el proceso continúa hasta que se selecciona la herramienta `final_answer`.

3.7. Prompt del agente

El prompt del sistema define el rol del agente como analista especializado en series de tiempo eléctricas. Contiene instrucciones estrictas que impiden:

- crear o reemplazar el dataset,
- inventar columnas o valores,
- usar funciones no autorizadas
- devolver resultados sin haber usado herramientas cuando corresponda.

Este prompt proporciona el marco conceptual que guía el comportamiento del agente y garantiza que sus acciones estén alineadas con los objetivos del sistema. Esto mejora las respuestas del sistema ya que da çontexto.ªl modelo para tener una idea del tema.

3.8. Resumen conceptual

El sistema implementado integra:

- un modelo de lenguaje local,
- un conjunto de herramientas especializadas,
- un flujo iterativo de razonamiento,

- un dataset real procesado correctamente,
- y capacidades de análisis, graficación y predicción.

Todo esto se basa en el enfoque modular descrito por Mauro Di Pietro [2], demostrando que es posible construir un agente autónomo funcional, capaz de operar completamente en local y realizar análisis avanzados de series de tiempo energéticas.

4. Archivos del proyecto

A continuación se presenta la estructura del proyecto, organizada por categorías según su función dentro del sistema. Esta clasificación permite una comprensión más clara del flujo de trabajo, desde la carga de datos hasta la ejecución del agente y la generación de predicciones.

Archivos principales del sistema

- `main.py`: Script principal del proyecto; coordina el análisis, las consultas del usuario y la interacción con el agente basado en LLM.
- `main2.py`: Variante del script principal enfocada directamente en los datos del ICE.
- `system_Mauro.py`: Adaptación del sistema original propuesto por Mauro Di Pietro para integrar herramientas, razonamiento iterativo y ejecución de código.

Scripts de procesamiento y validación de datos

- `obtencion_Datos.py`: Realiza la recopilación, limpieza y estructuración inicial de los datos de consumo eléctrico.
- `revisión_Datos.py`: Verifica la integridad del dataset, incluyendo fechas, calidad de datos y estructura general.
- `env_prueba.py`: Comprueba que el entorno virtual y las librerías necesarias estén instalados correctamente.

Datasets utilizados

- `archivos/datos_cence.csv`: Dataset original proporcionado por el CENCE con datos de consumo energético.
- `datos_limpios.csv`: Dataset procesado y depurado, utilizado por el agente para análisis y predicción.
- `predicciones/Pruebas/DatosReales`: Datos reales empleados para evaluar predicciones realizadas por el sistema.

Documentos de soporte

- `archivos/ProyectoElectrico_PrediccionEnTiempoReal.pdf`: Enunciado oficial del proyecto y sus lineamientos.
- `README.md`: Documentación general con instrucciones de instalación, uso y estructura del sistema.
- `requirements.txt`: Lista de dependencias necesarias para ejecutar el proyecto.

5. Repositorio y guía de instalación

Para la realización del proyecto se trabajó en Visual Studio Code bajo un entorno virtual.

5.1. Clonar el repositorio

Para comenzar, es necesario clonar el repositorio con sus archivos localmente. Para esto:

- Asegurarse de que Git está instalado. Es posible probar con:

  ```
  $ git --version
  ```

- Ubicarse en el directorio donde estará ubicado el proyecto, con:

  ```
  $ cd
  ```

- Clonar el proyecto:

  ```
  $ git clone https://github.com/ErickVM7/Proyecto-Electrico
  ```

- Moverse al directorio del proyecto:

  ```
  $ cd Proyecto-Electrico
  ```

- Si no fue hecho antes, configurar las credenciales de Git en el sistema local, de modo que quede vinculado con la cuenta de GitHub:

  ```
  $ git config --global user.name "Nombre Apellido"
  $ git config --global user.email "your-email@example.com"
  ```

5.2. Descargar e instalar Ollama

Seguir los pasos de descarga e instalación de Ollama desde su página según su sistema operativo:

https://ollama.com/download

5.3. Crear un ambiente virtual de Python

En una terminal, en el directorio raíz del repositorio, utilizar:

```
python3 -m venv env
```

donde `env` es el nombre del ambiente. Esto crea una carpeta con ese nombre.
Para activar el ambiente virtual, utilizar:

Linux/macOS

```
source env/bin/activate
```

donde `env/bin/activate` es el `PATH`. El *prompt* de la terminal cambiará a algo similar a:

```
base env ~/.../pipeline $
```

Windows (CMD)

```
.\env\Scripts\Activate.bat
```

Windows (PowerShell)

```
.\env\Scripts\Activate.ps1
```

Nota: si aparece un error sobre ejecución de scripts o permisos, ejecutar:

```
Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass
```

En este ambiente virtual no hay paquetes de Python instalados. Es posible verificar esto con:

```
pip list
```

que devolverá algo como:

```
Package    Version
---------- -------
pip        24.0
setuptools 65.5.0
```

5.4. Instalar los paquetes necesarios para ejecutar el proyecto

Con el ambiente virtual activado, instalar los paquetes indicados en el archivo `requirements.txt` con:

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

Para verificar la instalación, es posible usar nuevamente:

```
pip list
```

que ahora mostrará una buena cantidad de nuevos paquetes y sus dependencias. Además, se puede ejecutar el archivo `env_prueba.py` para verificar que se haya instalado correctamente el entorno virtual.

6. Registro de interacciones y resultados

A continuación se presenta el registro de consultas realizadas al agente, junto con las respuestas generadas en forma de gráficos y análisis. Cada figura corresponde a una interacción específica con el sistema. Pero primero se calculó la estadística general de los datos para comprobar que el modelo respondiera correctamente.

6.1. Estadistica de los datos reales

Estadísticas descriptivas:

	fechaHora	MW	MW_P
count	35136	35136.000000	35136.000000
mean	2025-02-24 23:52:30	1475.141095	1484.819669
min	2024-08-26 00:00:00	838.760000	907.500000
25%	2024-11-25 11:56:15	1229.037500	1241.900000
50%	2025-02-24 23:52:30	1519.450000	1528.000000
75%	2025-05-27 11:48:45	1700.882500	1709.200000
max	2025-08-26 23:45:00	1940.230000	1939.000000
std	NaN	255.486746	253.788202

Figura 1: Primer análisis general realizado con Pandas sobre `dtf`.

6.2. Cargar datos

Carga correcta del archivo

```
> cargar datos_limpios.csv
> load_csv -> Inputs: {'path': 'datos_limpios.csv'}
                     MW    MW_P
fechaHora
2024-08-26 00:00:00  1089.94  1078.10
2024-08-26 00:15:00  1075.71  1078.10
2024-08-26 00:30:00  1064.77  1078.10
2024-08-26 00:45:00  1036.90  1078.10
2024-08-26 01:00:00  1039.78  1041.89
Archivo datos_limpios.csv cargado correctamente con 35136 filas y columnas ['MW', 'MW_P']
> {}
>
```

Figura 2: Carga exitosa del archivo `datos_limpios.csv`.

Error al intentar cargar un archivo inexistente

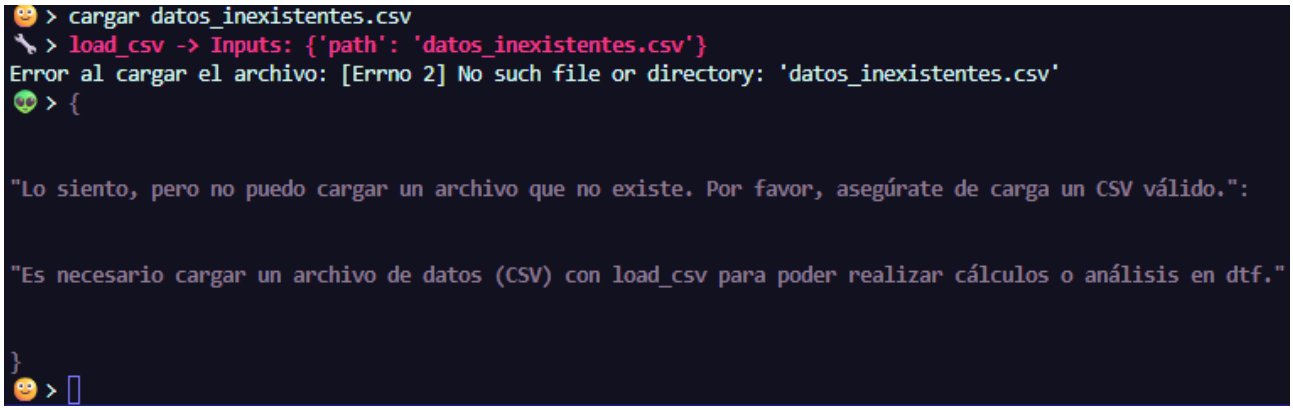

Figura 3: Intento fallido de cargar un archivo que no existe.

Intento de análisis sin datos cargados

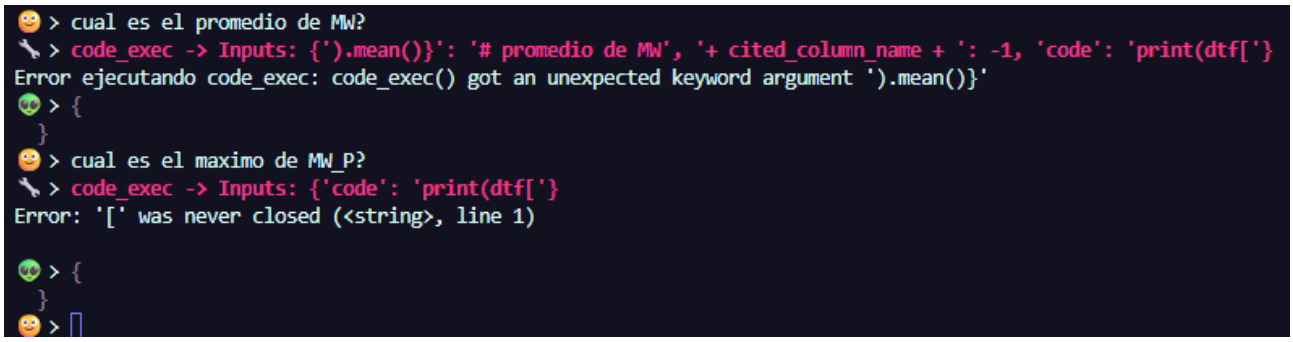

Figura 4: Mensaje del agente cuando se intenta consultar sin datos cargados.

6.3. Consultas sobre estadísticas y filtrado

Consumo máximo del año

```
> ¿Cuál fue el consumo más alto este año?
> code_exec -> Inputs: {'code': "print(dtf['MW_P'].max())"}
1939.0

> {
"texto": "El consumo más alto este año es de 1939.0 MW."
}
> ¿Cuál fue el consumo más alto MW  este año?
> code_exec -> Inputs: {'code': "print(dtf['MW_P'].max())"}
1939.0

> {
"texto": "El consumo más alto de MW este año es de 1939.0 MW."
}
> MW, no del MW_P
> code_exec -> Inputs: {'code': "print(dtf['MW'].max())"}
1940.23

> { "texto": "El consumo más alto de MW este año es de 1940.23 MW." }
>
```

Figura 5: Pregunta: "¿Cuál fue el consumo más alto de MW este año?"

Valor mínimo de la columna `MW_P`

```
> cual es el mínimo de MW_P?
> code_exec -> Inputs: {'code': "print(dtf['MW_P'].min())"}
907.5

> {
  "text": "El valor mínimo de MW_P es 907.5."
}
>
```

Figura 6: Pregunta: "¿Cuál fue el mínimo de MW_P?"

Suma total de la columna `MW`

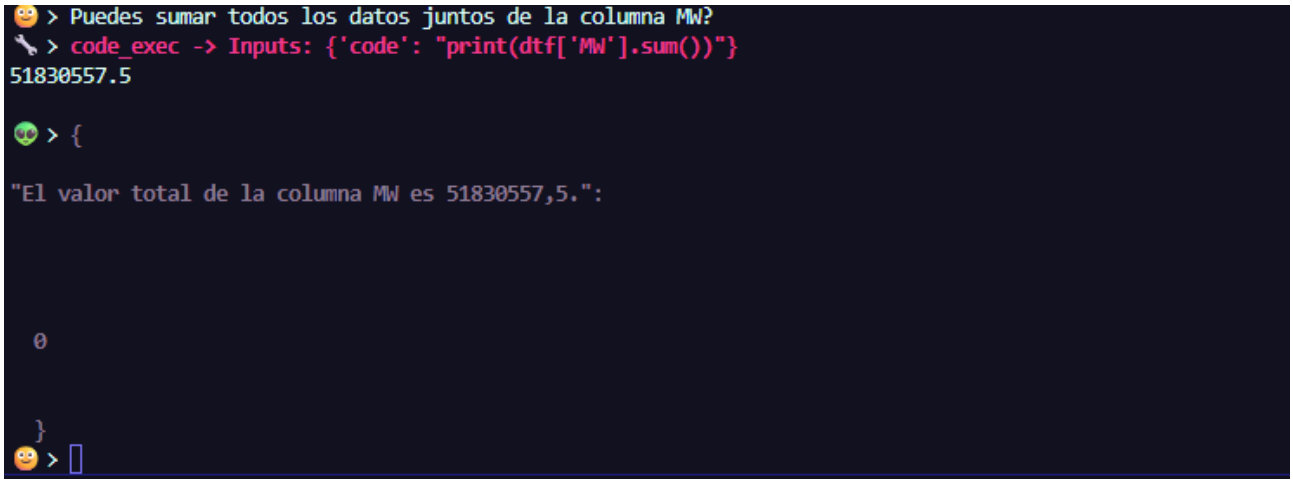

Figura 7: Pregunta: "¿Puedes sumar todos los datos de la columna MW?"

Cantidad total de registros del dataset

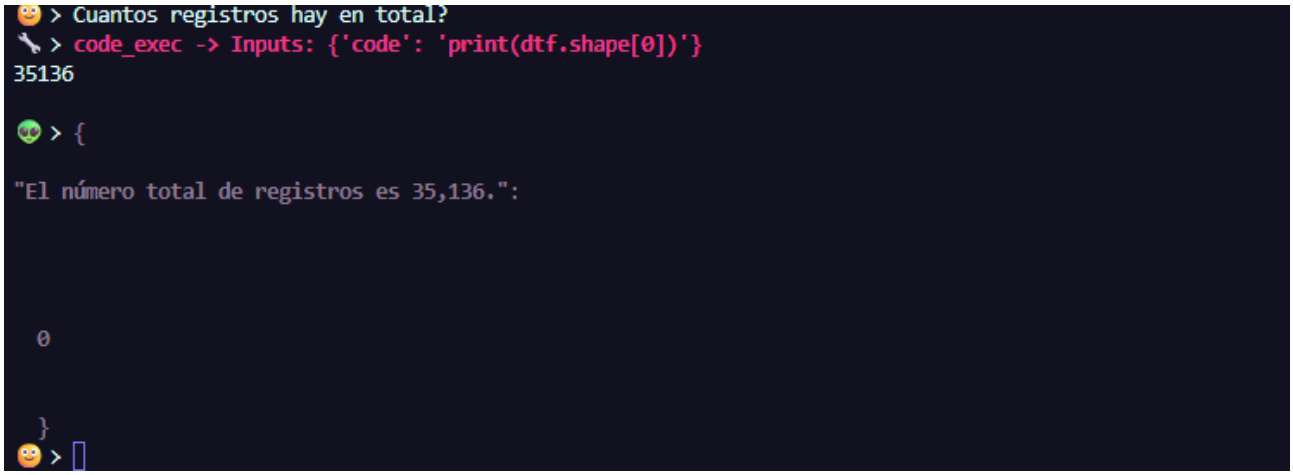

Figura 8: Pregunta: "¿Cuántos registros hay?"

Datos del día 2024-09-26

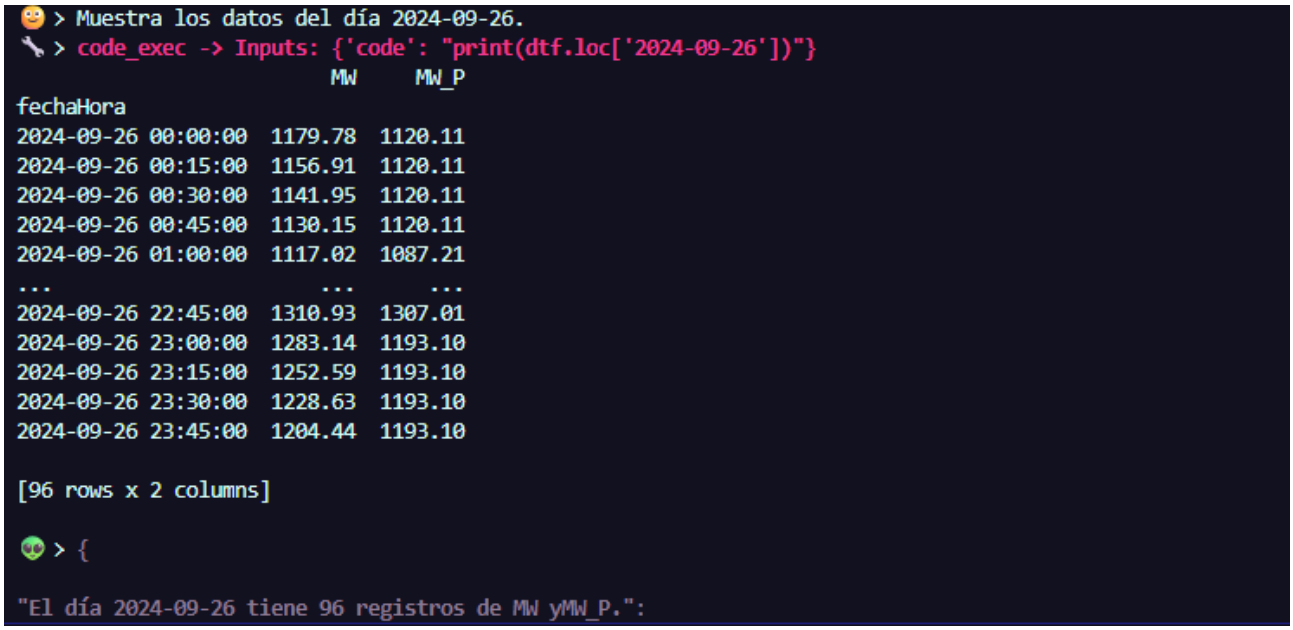

Figura 9: Pregunta: “Muestra los datos del día 2024-09-26”.

Promedio mensual en un rango específico

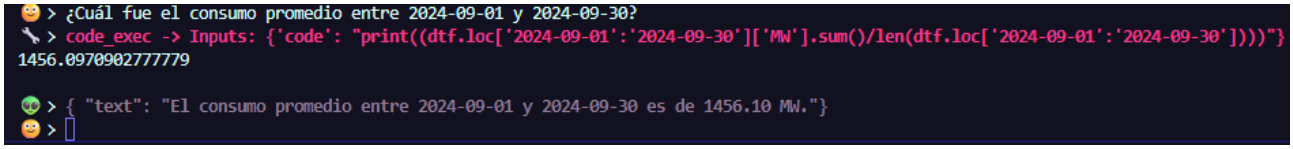

Figura 10: Pregunta: “¿Cuál fue el consumo promedio entre 2024-09-01 y 2024-09-30?”

Gráfico de un día específico

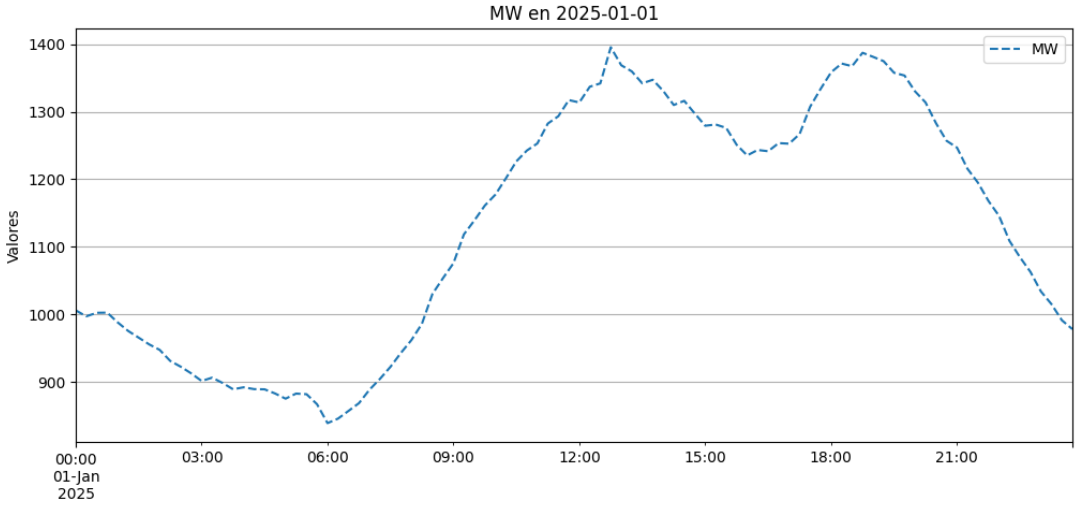

Figura 11: Pregunta: "Haz un gráfico del día 2025-01-01".

6.4. Predicciones generadas por el agente

Predicción Prophet a 1 día

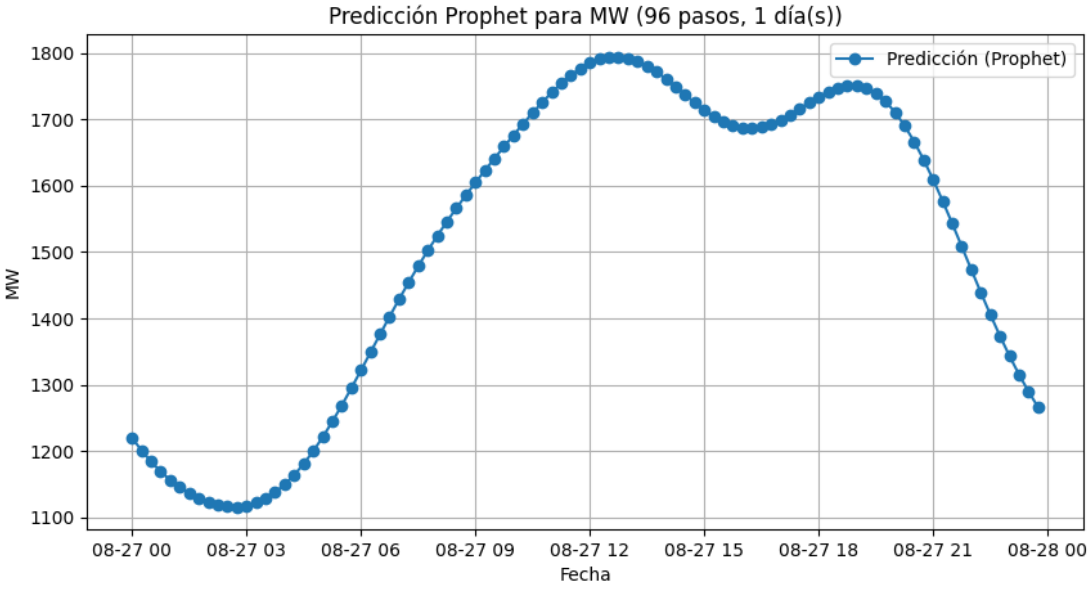

Figura 12: Predicción de MW utilizando Prophet para 1 día.

Predicción Prophet a 3 días

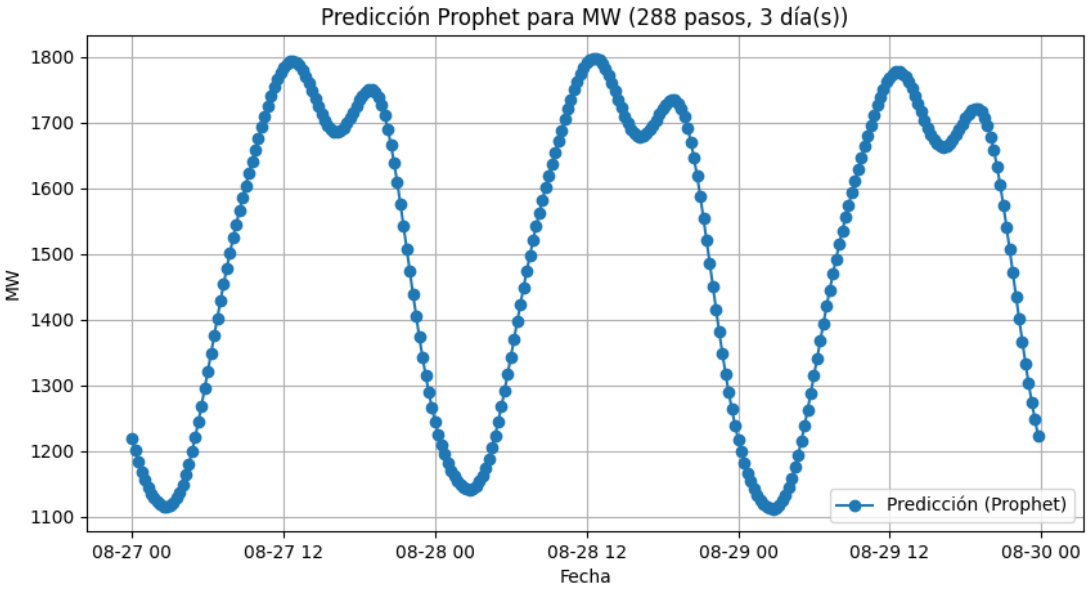

Figura 13: Predicción de MW utilizando Prophet para 3 días.

Predicción ARIMA a 2 días

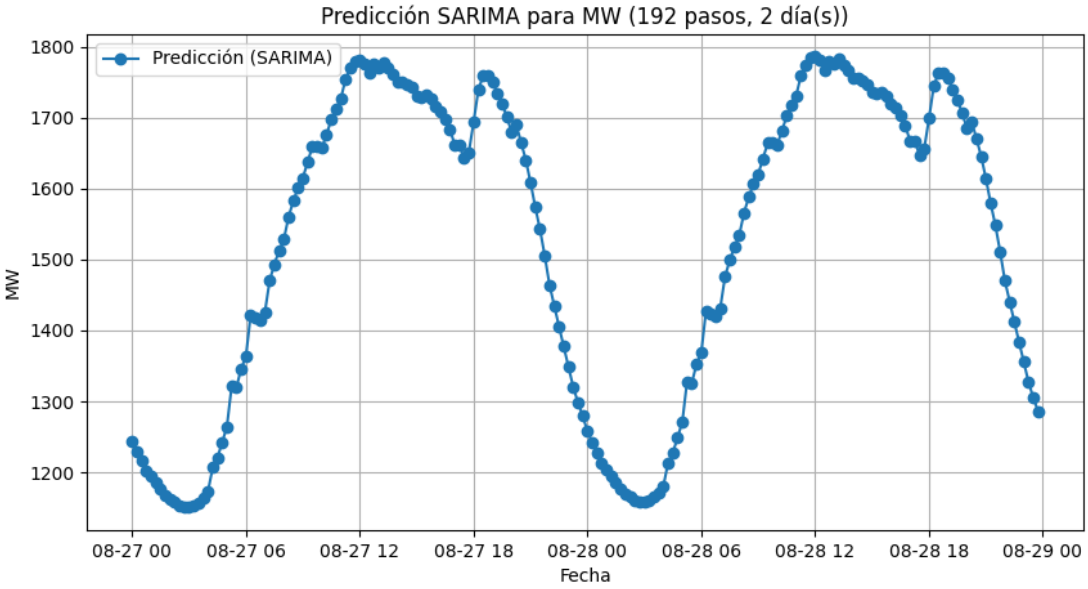

Figura 14: Predicción de MW utilizando ARIMA para 2 días.

Predicción ARIMA a 3 días

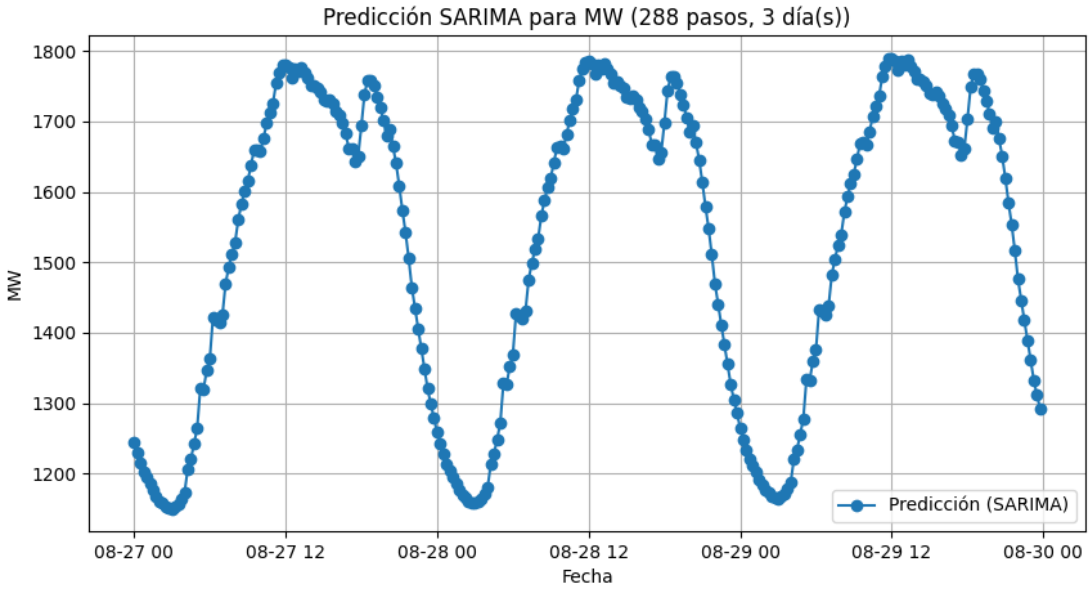

Figura 15: Predicción de MW utilizando ARIMA para 3 días.

6.5. Evaluación de las predicciones

```
=== Resultados de comparación ===
      Modelo          Archivo Real  MAE (MW)  RMSE (MW)  MAPE (%)  N muestras
0    ARIMA_2d  DatosReales2dias.csv    43.105     55.833      2.67         192
1    ARIMA_3d  DatosReales3dias.csv    41.598     51.799      2.66         288
2  Prophet_1d  DatosReales1dias.csv    62.406     73.160      3.85          96
3  Prophet_3d  DatosReales3dias.csv    47.171     60.579      2.98         288
```

Figura 16: Errores MAE, RMSE y MAPE de las predicciones generadas.

7. Problemas

Durante el desarrollo del proyecto surgieron distintos problemas técnicos que fueron apareciendo conforme se avanzaba en la implementación del agente y en el procesamiento de los datos. Uno de los primeros desafíos estuvo relacionado con las limitaciones de hardware: no fue posible instalar modelos grandes como `mistral` o `llama3` debido a la memoria disponible, por lo que se trabajó con una versión más ligera (`qwen2.5:7b`). Esto no solo afectó la capacidad del modelo, sino que también se reflejó en ciertos comportamientos erráticos al interactuar con las herramientas definidas.

En varias ocasiones el modelo enviaba estructuras JSON malformadas, por ejemplo utilizando la clave `"final_answer"` en lugar de `"text"`, lo que generaba errores en la ejecución de las funciones. Para resolverlo, se implementó un mapeo automático que corregía esa clave antes de ejecutar la herramienta. También se observó que, en algunos casos, el modelo no lograba identificar correctamente qué herramienta debía usar: respondía con un texto explicativo cuando

la consulta requería ejecutar código, o intentaba usar `code_exec` incluso en preguntas que no estaban relacionadas con el DataFrame. Esto obligó a reforzar el prompt inicial y ajustar la lógica interna para que las decisiones fueran más consistentes.

Otro problema recurrente fue la inconsistencia en los nombres de variables y columnas. El modelo generaba código utilizando `df` cuando internamente el proyecto trabaja con `dtf`, o referenciaba columnas con mayúsculas o nombres distintos a los reales. Para evitar errores de ejecución, se creó un sanitizador que corrige automáticamente estas discrepancias. Además, surgieron dificultades con rutas de archivos, ya que mover los scripts ocasionaba fallos al intentar cargar los datos; esto se solucionó utilizando rutas robustas con `os.path.join`. En la etapa de procesamiento de datos también aparecieron contratiempos. Si la descarga fallaba o la conexión a internet se interrumpía, el archivo `datos_limpios.csv` podía quedar incompleto y era necesario repetir el proceso. A esto se sumó que los datos reales empleaban la columna `fechaHora`, mientras que las predicciones utilizaban `fecha`, lo que hacía imposible comparar ambas series sin unificar previamente los formatos.

8. Conclusiones

El proyecto logró cumplir el objetivo general de implementar un sistema capaz de analizar y predecir series de tiempo de consumo energético utilizando agentes de inteligencia artificial ejecutados completamente en local. A partir de la arquitectura propuesta por Mauro Di Pietro, se desarrolló un agente funcional que combina el razonamiento de un modelo de lenguaje con herramientas para ejecutar código, graficar datos y generar predicciones. Esto permitió integrar en un solo flujo conversacional tareas que normalmente requieren programación manual, facilitando el análisis exploratorio y la generación de resultados.

En relación con los objetivos específicos, se consiguió adaptar el sistema original al dataset energético real del país. Se construyó un entorno reproducible en Python y Ollama, capaz de cargar, estructurar y manipular datos históricos de demanda eléctrica con frecuencia de 15 minutos. El agente pudo interpretar consultas en lenguaje natural y traducirlas en operaciones sobre el dataframe, cumpliendo así con la implementación del módulo interactivo solicitado.

Asimismo, se desarrolló un módulo de predicción basado en regresores especializados como Prophet y ARIMA. Estos modelos permitieron generar pronósticos de uno, dos y tres días, los cuales fueron evaluados mediante métricas como MAE, RMSE y MAPE. Los resultados mostraron un desempeño adecuado, con errores relativos bajos considerando la magnitud absoluta de la demanda eléctrica. Esto evidencia que el sistema es útil para tareas de predicción a corto plazo, cumpliendo el objetivo de validar la precisión del módulo predictivo. Además se observó que la precisión de las predicciones variaba según el modelo y el horizonte temporal. Prophet tendía a acumular más error en predicciones cortas, mientras que ARIMA mostró un desempeño más estable. Estos hallazgos permitieron refinar el sistema, documentar sus limitaciones y mejorar la robustez general del flujo de trabajo.

Todo el proyecto quedó documentado en un repositorio con instrucciones de instalación, uso y replicación. El sistema puede ejecutarse sin conexión a la nube, garantizando privacidad y portabilidad. En conjunto, el proyecto demuestra que es posible construir un agente conversacional orientado al análisis energético utilizando únicamente herramientas locales y modelos ligeros, cumpliendo de manera satisfactoria los objetivos planteados.

Referencias

[1] Marvin Coto. *Enunciado del Proyecto sistema de análisis y predicción de series de tiempo usando agentes de inteligencia artificial basados en LLM y notas de clase del curso Inteligencia Artificial Aplicada a la Ingeniería Eléctrica*. 2024.

[2] Mauro Di Pietro, *AI Agents Processing Time Series and Large Dataframes. Build from Scratch using only Python & Ollama*, Towards Data Science, 2025. `AIAgentsProcessingTimeSeriesandLargeDataframes_TowardsDataScience.pdf`