Documentación tecnica del Proyecto

Autores: Nicolas Hurtado A. & Jacobo Restrepo M. **Curso:** Tópicos Especiales en Telemática

3. Estructura de Directorios

```
Proyecto3-bigdata/
├── capture/
│   ├── open_meteo_fetcher.py
│   └── db_fetcher.py
├── config/
│   └── buckets.json
├── scripts/
│   └── spark_jobs/
│       ├── weather_etl.py
│       └── weather_analysis.py
├── pipeline_orchestrator.py
├── requirements.txt
└── README.md
```

4. Flujo del Proyecto y Componentes

El pipeline es orquestado por `pipeline_orchestrator.py` e involucra varias etapas distintas:

4.1. Configuración

- **`config/buckets.json`** : Define las rutas S3 para las diferentes etapas de datos.

```
{
  "raw_api_bucket": "s3://weather-etl-raw-nicojaco/open_meteo/",
  "raw_db_bucket": "s3://weather-etl-raw-nicojaco/open_meteo/data_extract_from_db/",
  "trusted_processed_bucket": "s3://weather-etl-trusted-nicojaco/processed_weather/",
  "refined_predictions_bucket": "s3://weather-etl-refined-nicojaco/weather_predictions/",
  "logs_bucket": "s3://aws-logs-390845263746-us-east-1/elasticmapreduce/"
}
```

- **`requirements.txt`** : Lista los paquetes de Python necesarios.

```
boto3
requests
pandas
SQLAlchemy
PyMySQL
pyspark
```

4.2. Orquestación: `pipeline_orchestrator.py`

Este es el script principal que controla todo el pipeline.

Inicialización (`__init__`):

- Configura la región de AWS e inicializa los clientes `boto3` para EMR y S3.
- Carga las configuraciones de los buckets S3 desde `config/buckets.json` .
- Valida el formato y la presencia de las rutas de bucket S3 requeridas.

```
import boto3
import json
import time
from datetime import datetime
import logging
import subprocess
import os
import sys

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(module)s - %(message)s')
logger = logging.getLogger(__name__)
```

```
class ProjectOrchestrator:
    EMR_RELEASE_LABEL = 'emr-6.15.0'
    MASTER_INSTANCE_TYPE = 'm5.xlarge'
    CORE_INSTANCE_TYPE = 'm4.xlarge'
    CORE_INSTANCE_COUNT = 2
    KEEP_JOB_FLOW_ALIVE = True # O False si se desea terminar automáticamente
    TERMINATION_PROTECTED = False

    def __init__(self, config_path='config/buckets.json', aws_region=None):
        self.aws_region = aws_region if aws_region else os.getenv('AWS_DEFAULT_REGION', 'us-east-1')
        logger.info(f"Using AWS region: {self.aws_region}")
        self.emr_client = boto3.client('emr', region_name=self.aws_region)
        self.s3_client = boto3.client('s3', region_name=self.aws_region)
        # Inicializar el cliente de API Gateway si se va a usar la función expose_results_as_api
        # self.api_client = boto3.client('apigatewayv2', region_name=self.aws_region)
        self.api_client = None # Placeholder
        self.config_path = config_path
        self.buckets = self._load_config()
        self._validate_bucket_config()

    def _load_config(self):
        try:
            logger.info(f"Loading configuration from {self.config_path}")
            with open(self.config_path, 'r') as f:
                config = json.load(f)
            logger.info("Configuration loaded successfully.")
            return config
        except FileNotFoundError:
            logger.error(f"Configuration file not found: {self.config_path}")
            raise
        except json.JSONDecodeError:
            logger.error(f"Error decoding JSON from configuration file: {self.config_path}")
            raise
        except Exception as e:
            logger.error(f"An unexpected error occurred while loading config: {e}")
            raise

    def _validate_bucket_config(self):
        required_buckets = [
            'raw_api_bucket', 'raw_db_bucket',
            'trusted_processed_bucket', 'refined_predictions_bucket',
            'scripts_bucket', 'logs_bucket'
        ]
        for rb_key in required_buckets:
            if rb_key not in self.buckets:
                msg = f"Missing required S3 path key in config: '{rb_key}'"
                logger.error(msg)
                raise ValueError(msg)

            s3_path = self.buckets[rb_key]
            if not s3_path.startswith("s3://"):
                msg = f"Invalid S3 path format for '{rb_key}': '{s3_path}'. Must start with s3://"
                logger.error(msg)
                raise ValueError(msg)

            parsed_bucket, parsed_prefix = self._parse_s3_path(s3_path, rb_key)
            if not parsed_bucket:
                msg = f"Could not parse bucket name from '{rb_key}': '{s3_path}'"
                logger.error(msg)
                raise ValueError(msg)
            logger.info(f"Validated S3 path for '{rb_key}': bucket='{parsed_bucket}', prefix='{parsed_prefix}'")

    def _parse_s3_path(self, s3_path, config_key_name=""):
        if not s3_path.startswith("s3://"):
            logger.warning(f"S3 path for {config_key_name} '{s3_path}' does not start with s3://. Cannot parse.")
            return None, None
```

```
            return None, None
        parts = s3_path.replace("s3://", "").split("/", 1)
        bucket_name = parts[0]
        key_prefix = parts[1] if len(parts) > 1 else ""
        return bucket_name, key_prefix
```

Etapa 1: Ingesta de Datos (`run_ingestion_stage`):

- Ejecuta scripts locales de obtención de datos.
- **`capture/open_meteo_fetcher.py`** :
  - Obtiene datos meteorológicos históricos de la API Open-Meteo para parámetros predefinidos.
  - Sube los datos JSON recuperados al bucket S3 `raw_api_bucket` .
  - *Nota Importante: Este script debe analizar correctamente la ruta S3 para extraer el nombre del bucket y el prefijo de la clave, en lugar de usar la ruta completa como nombre del bucket.*

```
capture/open_meteo_fetcher.py (Lógica de carga a S3 - necesita corrección)
RAW_BUCKET_NAME, S3_KEY_PREFIX deben ser parseados desde config['raw_api_bucket']
filename_s3 es el nombre del archivo ej: "open_meteo_data_2022-01-01_to_2022-12-31.json"
s3_object_key = f"{S3_KEY_PREFIX}{filename_s3}"

try:
    s3_client.put_object(
        Bucket=RAW_BUCKET_NAME, # Solo el nombre del bucket
        Key=s3_object_key,    # Prefijo + nombre de archivo
        Body=response.text,
        ContentType='application/json'
    )
...
```

- **`capture/db_fetcher.py`** :
  - Se conecta a una base de datos MySQL.
  - Obtiene datos y los convierte a CSV.
  - Sube los datos CSV al bucket S3 `raw_db_bucket` .

```
capture/db_fetcher.py (Lógica de carga a S3)
RAW_DB_BUCKET_NAME y RAW_DB_S3_KEY_PREFIX se parsean de config['raw_db_bucket']
s3_object_key = f"{RAW_DB_S3_KEY_PREFIX}{s3_file_name}"

try:
    s3_client.put_object(
        Bucket=RAW_DB_BUCKET_NAME,
        Key=s3_object_key,
        Body=csv_string,
        ContentType='text/csv'
    )
...
```

- El orquestador utiliza `_run_local_script` para ejecutar estos scripts.

```
# pipeline_orchestrator.py
    def _run_local_script(self, script_path, script_args=None, cwd=None):
        if script_args is None:
            script_args = []
        command = [sys.executable, script_path] + script_args
        logger.info(f"Executing local script: {' '.join(command)} {'in CWD: '+cwd if cwd else ''}")
        try:
            process = subprocess.Popen(command, stdout=subprocess.PIPE, stderr=subprocess.PIPE, text=True, cwd=cwd)
            stdout, stderr = process.communicate()

            if process.returncode == 0:
                logger.info(f"Script {script_path} executed successfully.")
                logger.debug(f"Script {script_path} STDOUT:\n{stdout}")
                if stderr:
                    logger.warning(f"Script {script_path} STDERR:\n{stderr}")
                return True
            else:
                # ... (manejo de errores)
                return False
        # ... (manejo de excepciones)
        return False

    def run_ingestion_stage(self):
        logger.info("====== Starting Ingestion Stage ======")
        orchestrator_dir = os.path.dirname(os.path.abspath(__file__))
        api_fetcher_script = os.path.join(orchestrator_dir, 'capture', 'open_meteo_fetcher.py')
        logger.info("Running Open-Meteo API data ingestion...")
        api_success = self._run_local_script(api_fetcher_script, cwd=os.path.join(orchestrator_dir, 'capture'))
        if not api_success:
            logger.error("Open-Meteo API ingestion failed. Halting pipeline.")
            raise RuntimeError("Open-Meteo API ingestion failed.")
        logger.info("Open-Meteo API data ingestion completed.")

        db_fetcher_script = os.path.join(orchestrator_dir, 'capture', 'db_fetcher.py')
        logger.info("Running Database data ingestion...")
        db_success = self._run_local_script(db_fetcher_script, cwd=os.path.join(orchestrator_dir, 'capture'))
        if not db_success:
            logger.error("Database ingestion failed. Halting pipeline.")
            raise RuntimeError("Database ingestion failed.")
        logger.info("Database data ingestion completed.")

        logger.info("====== Ingestion Stage Completed Successfully ======")
        return True
```

Etapa 2: Subir Scripts de Spark (`_upload_spark_scripts_to_s3`):

- Sube los archivos Python de trabajos de Spark (`weather_etl.py`, `weather_analysis.py`) desde `scripts/spark_jobs/` al bucket S3 `scripts_bucket`.

```
# pipeline_orchestrator.py
    def _upload_spark_scripts_to_s3(self):
        logger.info("Uploading Spark scripts to S3...")
        scripts_s3_full_path = self.buckets['scripts_bucket']
        scripts_bucket_name, scripts_base_prefix = self._parse_s3_path(scripts_s3_full_path, 'scripts_bucket')
        scripts_base_prefix = scripts_base_prefix.rstrip('/') + '/' if scripts_base_prefix else ''

        orchestrator_dir = os.path.dirname(os.path.abspath(__file__))
        local_spark_jobs_dir = os.path.join(orchestrator_dir, 'scripts', 'spark_jobs')

        if not os.path.isdir(local_spark_jobs_dir):
            # ... (manejo de errores)
            pass

        uploaded_script_paths = {}
        for filename in os.listdir(local_spark_jobs_dir):
            if filename.endswith(".py"):
                # ... (sube el archivo a S3)
                pass
        if not uploaded_script_paths:
            logger.warning(f"No Spark scripts found or uploaded from {local_spark_jobs_dir}")
        logger.info("Spark scripts upload completed.")
        return uploaded_script_paths
```

Etapa 3: Gestión del Clúster EMR:

- **Crear Clúster EMR (`create_emr_cluster`)**:
    - Define la configuración del clúster EMR (nombre, URI de logs, etiqueta de lanzamiento, aplicaciones como Spark, Hadoop, Hive, Livy).
    - Especifica grupos de instancias (Master, Core) con tipos y recuentos.
    - Incluye par de claves EC2, configuraciones de mantenimiento y roles de servicio.
    - Establece configuraciones de Spark.
    - Lanza el clúster usando `emr_client.run_job_flow` .
    - *Nota: Las acciones de arranque (bootstrap actions) fueron eliminadas previamente.*

```
# pipeline_orchestrator.py
    def create_emr_cluster(self):
        logger.info(f"Creating EMR cluster without bootstrap script")
        cluster_name = "MyEMR" # Puede ser parametrizado o generado dinámicamente

        logs_s3_full_path = self.buckets['logs_bucket']
        logs_bucket_name, logs_base_prefix = self._parse_s3_path(logs_s3_full_path, 'logs_bucket')
        emr_log_uri = f"s3://{logs_bucket_name}/{logs_base_prefix.rstrip('/') + '/' if logs_base_prefix else ''}emr_logs/"
        logger.info(f"EMR Log URI will be: {emr_log_uri}")

        cluster_config = {
            'Name': cluster_name,
            'LogUri': emr_log_uri,
            'ReleaseLabel': self.EMR_RELEASE_LABEL,
            'Applications': [{'Name': 'Spark'}, {'Name': 'Hadoop'}, {'Name': 'Hive'}, {'Name': 'Livy'}],
            'Instances': {
                'InstanceGroups': [
                    {
                        'Name': 'MasterNode',
                        'Market': 'ON_DEMAND',
                        'InstanceRole': 'MASTER',
                        'InstanceType': self.MASTER_INSTANCE_TYPE,
                        'InstanceCount': 1,
                    },
                    {
                        'Name': 'CoreNodes',
                        'Market': 'ON_DEMAND',
                        'InstanceRole': 'CORE',
                        'InstanceType': self.CORE_INSTANCE_TYPE,
                        'InstanceCount': self.CORE_INSTANCE_COUNT,
                    }
                ],
                'Ec2KeyName': 'nicojaco-ec2-key-pair', # IMPORTANTE: Configurar con tu par de claves
                'KeepJobFlowAliveWhenNoSteps': self.KEEP_JOB_FLOW_ALIVE,
                'TerminationProtected': self.TERMINATION_PROTECTED,
            },
            'ServiceRole': 'EMR_DefaultRole', # IMPORTANTE: Asegurar que los roles tengan los permisos necesarios
            'JobFlowRole': 'EMR_EC2_DefaultRole',
            'VisibleToAllUsers': True,
            'Configurations': [
                {
                    'Classification': 'spark-defaults',
                    'Properties': {
                        'spark.sql.adaptive.enabled': 'true',
                        'spark.sql.adaptive.coalescePartitions.enabled': 'true',
                        'spark.serializer': 'org.apache.spark.serializer.KryoSerializer'
                    }
                }
            ]
        }

        try:
            response = self.emr_client.run_job_flow(**cluster_config)
            cluster_id = response['JobFlowId']
            logger.info(f"EMR cluster creation initiated. Name: {cluster_name}, ID: {cluster_id}")
            return cluster_id
        except Exception as e:
            logger.error(f"Failed to create EMR cluster: {e}")
            raise
```

- **Esperar a que el Clúster esté Listo (`_wait_for_cluster_ready`)**:
    - Usa un "waiter" de EMR (`cluster_running`) para pausar hasta que el clúster esté listo.
- **Terminar Clúster EMR (`terminate_emr_cluster`)**:
    - Permite la terminación programática si `KEEP_JOB_FLOW_ALIVE` es `False` .

Etapa 4: Ejecutar Trabajo ETL (`run_etl_stage`):

- Envía `scripts/spark_jobs/weather_etl.py` como un paso de EMR.
- **`scripts/spark_jobs/weather_etl.py`** :
  - Crea una SparkSession.
  - Lee datos JSON (Open-Meteo) y CSV (base de datos) desde S3 (zona cruda).
  - Transforma los datos JSON: expande arrays, selecciona y renombra columnas.
  - Realiza un cross join entre datos meteorológicos y de estaciones.
  - Escribe el DataFrame resultante como Parquet en S3 (zona confiable).

```
 # scripts/spark_jobs/weather_etl.py (Lógica ETL clave)
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, arrays_zip, col

spark = SparkSession.builder.appName("WeatherETL").getOrCreate()


raw_api_input_path = "s3a://weather-etl-raw-nicojaco/open_meteo/open_meteo_data_2022-01-01_to_2022-12-31.json"
raw_db_input_path = "s3a://weather-etl-raw-nicojaco/open_meteo/data_extract_from_db/weather_data_from_db.csv"
trusted_output_path = "s3a://weather-etl-trusted-nicojaco/processed_weather/" # Ejemplo

weather_raw = spark.read.option("multiline", "true").json(raw_api_input_path)
weather_df = weather_raw.select(
    explode(
        arrays_zip(
            col("daily.time"),
            col("daily.temperature_2m_max"),
            col("daily.precipitation_sum")
        )
    ).alias("daily_record")
).select(
    col("daily_record.time").alias("date"),
    col("daily_record.temperature_2m_max").alias("temperature_2m_max"),
    col("daily_record.precipitation_sum").alias("precipitation_sum")
)

stations_df = spark.read.csv(raw_db_input_path, header=True, inferSchema=True)

combined_df = weather_df.crossJoin(stations_df) # Asegurarse que el join es lo deseado

combined_df.write.mode("overwrite").parquet(trusted_output_path)
print("✅ ETL completado correctamente.")
```

- El orquestador usa `_submit_emr_step` y `_wait_for_step_completion` .

```
# pipeline_orchestrator.py
    def _submit_emr_step(self, cluster_id, step_name, script_s3_path, script_args):
        logger.info(f"Submitting EMR step: {step_name} (Script: {script_s3_path}, Args: {script_args})")
        full_args = ['spark-submit', '--deploy-mode', 'cluster', script_s3_path] + script_args
        step_config = {
            'Name': step_name,
            'ActionOnFailure': 'CONTINUE', # O 'TERMINATE_JOB_FLOW'
            'HadoopJarStep': {'Jar': 'command-runner.jar', 'Args': full_args}
        }
        # ... (envía el paso a EMR)
        pass

    def _wait_for_step_completion(self, cluster_id, step_id, step_name):
        # ... (usa un "waiter" de EMR para step_complete)
        pass

    def run_etl_stage(self, cluster_id, spark_script_s3_paths):
        logger.info("====== Starting ETL Stage ======")
        script_filename = 'weather_etl.py'
        script_s3_path = spark_script_s3_paths.get(script_filename)
        # ... (manejo de errores si no se encuentra el script)

        args = [ # Estos argumentos se pasan al script de Spark
            self.buckets['raw_api_bucket'],
            self.buckets['raw_db_bucket'],
            self.buckets['trusted_processed_bucket']
        ]
        etl_step_id = self._submit_emr_step(cluster_id, "WeatherETLSparkJob", script_s3_path, args)
        status = self._wait_for_step_completion(cluster_id, etl_step_id, "WeatherETLSparkJob")
        # ... (registro de estado)
        return status == 'COMPLETED'
```

Etapa 5: Ejecutar Trabajo de Análisis (`run_analysis_stage`):

- Envía `scripts/spark_jobs/weather_analysis.py` como otro paso de EMR.
- **`scripts/spark_jobs/weather_analysis.py`** :
    - Crea una SparkSession.
    - Lee datos Parquet desde S3 (zona confiable).
    - Realiza análisis descriptivo.
    - Prepara datos para Machine Learning: renombra columnas, usa `VectorAssembler` .
    - Entrena un modelo de Regresión Lineal.
    - Realiza predicciones.
    - Guarda las predicciones como JSON en S3 (zona refinada).

```
 # scripts/spark_jobs/weather_analysis.py (Lógica ML clave)
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.regression import LinearRegression
import sys

# ... (funciones auxiliares como create_spark_session, load_and_prepare_initial_data, etc.)

def main(input_s3_path, predictions_s3_output_path):
    spark = create_spark_session()
    prepared_df = load_and_prepare_initial_data(spark, input_s3_path)
    # ... (análisis descriptivo)
    ml_df = prepare_data_for_ml(prepared_df, ...)
    vectorized_df = engineer_features(ml_df, input_cols=["rain"], output_col="features") # Ejemplo de característica
    model = train_linear_regression_model(vectorized_df, features_col="features", label_col="label")
    predictions_result_df = make_predictions(model, vectorized_df)
    save_predictions(predictions_result_df, predictions_s3_output_path)
    spark.stop()

 if __name__ == "__main__":
     input_path = sys.argv[1]
     predictions_output_path = sys.argv[2]
     main(input_path, predictions_output_path)
```

- Esta etapa también usa `_submit_emr_step` y `_wait_for_step_completion` .

Etapa 6: Exponer Resultados (Opcional/Placeholder)

- **`make_json_public()`** : Intenta hacer público un archivo JSON específico. *Nota: La clave del objeto S3 está hardcodeada y debería ser dinámica.*
- **`expose_results_as_api()`** : Placeholder para crear un endpoint de API Gateway. *Nota: `self.api_client` debe inicializarse.*

```
# pipeline_orchestrator.py
    def make_json_public(self):
        try:
            self.s3_client.put_object_acl(
                ACL='public-read',
                Bucket='weather-etl-refined-nicojaco', # Ejemplo
                Key='weather_predictions/part-00000-...json' # IMPORTANTE: Clave hardcodeada
            )
            logger.info(f"Made weather_predictions/part-00000-...json public in weather-etl-refined-nicojaco")
        except Exception as e:
            logger.error(f"Could not make JSON public: {e}")


    def expose_results_as_api(self):
        if not self.api_client:
            logger.warning("API Client not initialized. Cannot create API Gateway endpoint.")
            return None
        try:
            response = self.api_client.create_api( # Ejemplo usando API Gateway V2
                Name='weather-predictions-api',
                ProtocolType='HTTP',
                Target='https://weather-etl-refined-nicojaco.s3.amazonaws.com/weather_predictions/part-00000-...json' # IMPORTANTE
            )
            api_endpoint = response['ApiEndpoint']
            logger.info(f"API Gateway endpoint created: {api_endpoint}")
            return api_endpoint
        except Exception as e:
            logger.error(f"Could not create API Gateway endpoint: {e}")
            return None
```

Bloque Principal de Ejecución (`run_full_pipeline` y `if __name__ == "__main__":`):

- `run_full_pipeline` llama a todas las etapas secuencialmente.
- Maneja errores y la terminación del clúster.
- El bloque `if __name__ == "__main__":` ejecuta el pipeline y, si tiene éxito, intenta las operaciones post-pipeline.

```
# pipeline_orchestrator.py
    def run_full_pipeline(self, cluster_name_prefix="EMR-Pipeline"):
        logger.info(f"🚀🚀🚀 Starting Full Weather Data Pipeline (EMR cluster: {cluster_name_prefix}) 🚀🚀🚀")
        start_time = datetime.now()
        cluster_id = None

        try:
            self.run_ingestion_stage()

            spark_s3_paths = self._upload_spark_scripts_to_s3()
            if not spark_s3_paths.get('weather_etl.py') or not spark_s3_paths.get('weather_analysis.py'):
                raise RuntimeError("Essential Spark scripts (ETL or Analysis) failed to upload or were not found.")

            cluster_id = self.create_emr_cluster()
            self._wait_for_cluster_ready(cluster_id)

            if not self.run_etl_stage(cluster_id, spark_s3_paths):
                raise RuntimeError("ETL stage failed.")

            if not self.run_analysis_stage(cluster_id, spark_s3_paths):
                raise RuntimeError("Analysis stage failed.")

            end_time = datetime.now()
            logger.info(f"🎉🎉🎉 Full Weather Data Pipeline Completed Successfully on EMR cluster {cluster_id}! 🎉🎉🎉")
            logger.info(f"Total execution time: {end_time - start_time}")
            return {
                "status": "SUCCESS", "cluster_id": cluster_id,
                "total_duration_seconds": (end_time - start_time).total_seconds()
            }
        except Exception as e:
            # ... (manejo de errores)
            return {
                "status": "FAILED", "cluster_id": cluster_id, "error_message": str(e),
                # ...
            }
        finally:
            if cluster_id and not self.KEEP_JOB_FLOW_ALIVE:
                logger.info(f"KEEP_JOB_FLOW_ALIVE is False. Initiating termination for cluster {cluster_id}.")
                time.sleep(60) # Dar tiempo para que los logs finales se suban
                self.terminate_emr_cluster(cluster_id)
            elif cluster_id and self.KEEP_JOB_FLOW_ALIVE:
                logger.info(f"KEEP_JOB_FLOW_ALIVE is True. Cluster {cluster_id} will remain running.")

if __name__ == "__main__":
    orchestrator = ProjectOrchestrator()
    pipeline_result = orchestrator.run_full_pipeline() # Podrías pasar un prefijo de nombre de clúster aquí

    if pipeline_result and pipeline_result.get("status") == "SUCCESS":
        logger.info("Pipeline successful. Attempting to make results public and expose API.")
        try:
            orchestrator.make_json_public()
            if orchestrator.api_client: # Solo intentar si el cliente API está inicializado
                api_endpoint = orchestrator.expose_results_as_api()
                if api_endpoint:
                    print(f"API endpoint: {api_endpoint}")
                else:
                    logger.warning("Failed to create or retrieve API endpoint.")
            else:
                logger.warning("API client not initialized. Skipping expose_results_as_api.")
        except Exception as e:
            logger.error(f"Error during post-pipeline operations (make_json_public/expose_results_as_api): {e}")
    else:
        logger.error("Pipeline failed. Skipping post-pipeline operations.")
```

Cómo Ejecutar

1. Asegúrese de que todos los prerrequisitos y configuraciones (especialmente `config/buckets.json` y el nombre del par de claves EC2 en `pipeline_orchestrator.py`) estén correctamente establecidos.
2. Asegúrese de que AWS CLI esté configurado con los permisos necesarios.
3. Ejecute el orquestador desde la raíz del proyecto:

   ```
   python pipeline_orchestrator.py
   ```

   Opcionalmente, proporcione una ruta de configuración personalizada y una región de AWS:

   ```
   python pipeline_orchestrator.py
   ```

7. NOTA

- **Permisos IAM:** Los permisos IAM insuficientes son una fuente común de errores. Asegúrese de que los roles puedan acceder a S3, gestionar clústeres EMR y que el usuario/rol que ejecuta el orquestador tenga los permisos necesarios.