Salary Prediction of Data Professions MLOps Pipeline Report: Formative Draft

CMP6230 Data Management and Machine Learning Operations
Bachelor of Science (Hons) in Computer Science with Artificial Intelligence,
Sunway College Birmingham City University (BCU)

Submitted by: *Abhimat Dangi*

ID: 24152352

Abstract

This project builds an end-to-end MLOps pipeline to predict employee salaries from structured HR and performance data ("Salary Prediction of Data Professions" CSV). The pipeline is orchestrated in Apache Airflow and spans data ingestion into MariaDB, schema bootstrap, custom Python data validation (duplicates/NaNs/inf checks) instead of Great Expectations, preprocessing (feature engineering + imputation), model training across four regressors (Linear Regression, Decision Tree, Random Forest, Gradient Boosting), experiment tracking in MLflow, and serving via a FastAPI microservice. Model artifacts and column order persisted for stable inference; the best model (in our runs: GradientBoostingRegressor) is deployed. We monitor data drift and target/concept drift using Evidently to generate HTML+CSV reports on a unified reports path. Redis is used for lightweight caching; Parquet/CSV snapshots are exported for portability. The solution demonstrates reproducible automation, practical model management, and production-readiness with explainable, auditable steps from raw data to API predictions.

Keywords: MLOps, Salary prediction, Airflow, MLflow, FastAPI, scikit-learn, Evidently, MariaDB, Redis

I. INTRODUCTION

In the fast changing field of data jobs, predicting salary ranges right is very important for both worker and companies. Salary prediction models give job seekers a better idea about market trends, so they can set realistic hopes and plan career steps wisely. For employers, such models help in building fair and competitive pay structures that match skills and industry demand. This report looks at using machine learning methods to predict salaries of data professionals, by studying things like years of experience, education, role, and where they work. By using wide datasets and modern algorithms, the goal is to make predictions more accurate and also more clear about what really affects pay. These kind of data driven predictions are becoming key for workforce planning and making sure compensation stays fair (Kablaoui and Salman, 2024).

II. THEORETICAL BACKGROUND

Predicting salary in data jobs mostly depends on machine learning and some statistical ways, looking at things like role, years of experience, education and the industry someone works in. People usually use models like regression, decision trees or even neural networks to catch the hidden patterns and try to guess salary from past and current data. When there is big and mixed dataset, the model tends to be more solid and the results come out more precise. These kind of prediction models don't just guide individuals to set better career goals, but they also help companies to design pay structures that feels more fair. Having a good understanding of how these algorithms work and how they fit with salary data is quite important for building systems that can give both useful and fair results (Raj et al., 2025).

III. CANDIDATE SOURCE DATASETS AMD DATA STORAGE

Dataset 1: Salary Prediction of Data Professions

	FIRST NAME	LAST NAME	SEX	DOJ	CURRENT DATE	DESIGNATION	AGE	SALARY	UNIT	LEAVES USED	LEAVES REMAINING	RATI
0	TOMASA	ARMEN	F	5-18-2014	01-07-2016	Analyst	21.0	44570	Finance	24.0	6.0	2.0
1	ANNIE	NaN	F	NaN	01-07-2016	Associate	NaN	89207	Web	NaN	13.0	NaN
2	OLIVE	ANCY	F	7-28-2014	01-07-2016	Analyst	21.0	40955	Finance	23.0	7.0	3.0
3	CHERRY	AQUILAR	F	04-03-2013	01-07-2016	Analyst	22.0	45550	IT	22.0	8.0	3.0
4	LEON	ABOULAHOUD	M	11-20-2014	01-07-2016	Analyst	NaN	43161	Operations	27.0	3.0	NaN

Fig 1. Salary Prediction of Data Professions Screenshot

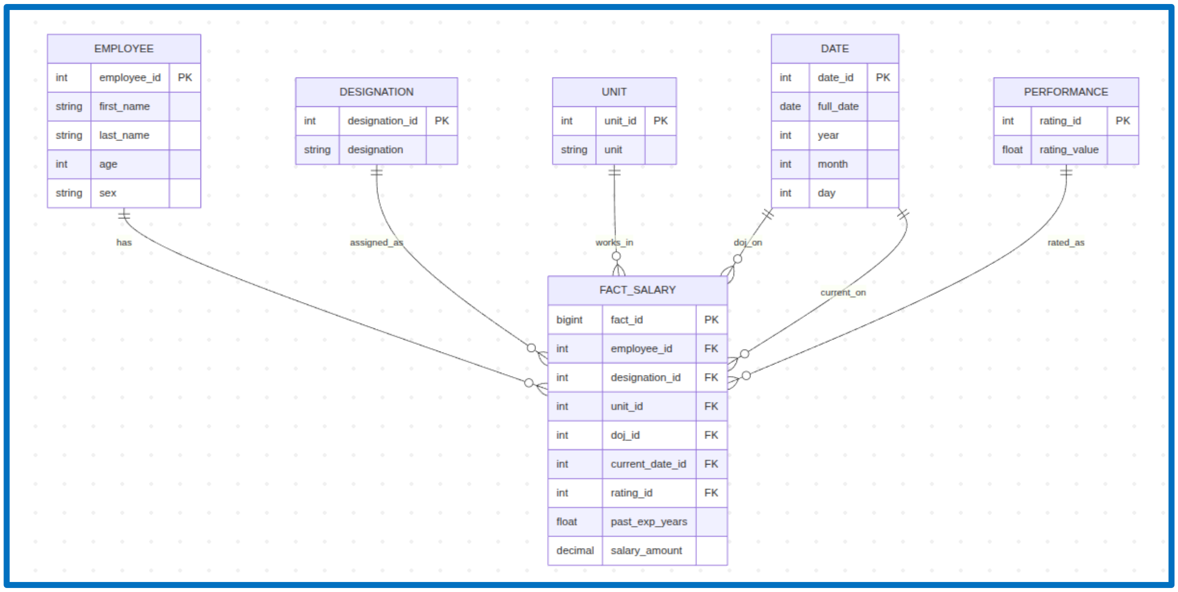


Fig 2. ERD of Salary Prediction of Data Professions

This dataset is for salary prediction in data profession jobs. It has **2639 rows and 13 columns**, with details like employee age, gender, designation, unit, joining date, current date, past experience, performance ratings and salary. These features show how personal and job factors affect pay. By using this data, machine learning models can learn patterns and then predict salary range. It is helpful for employees to know fair pay, and for companies to make better salary plans. The dataset has both numbers and categories, and also some missing values that need cleaning before use.

Dataset 2: Car Price Prediction

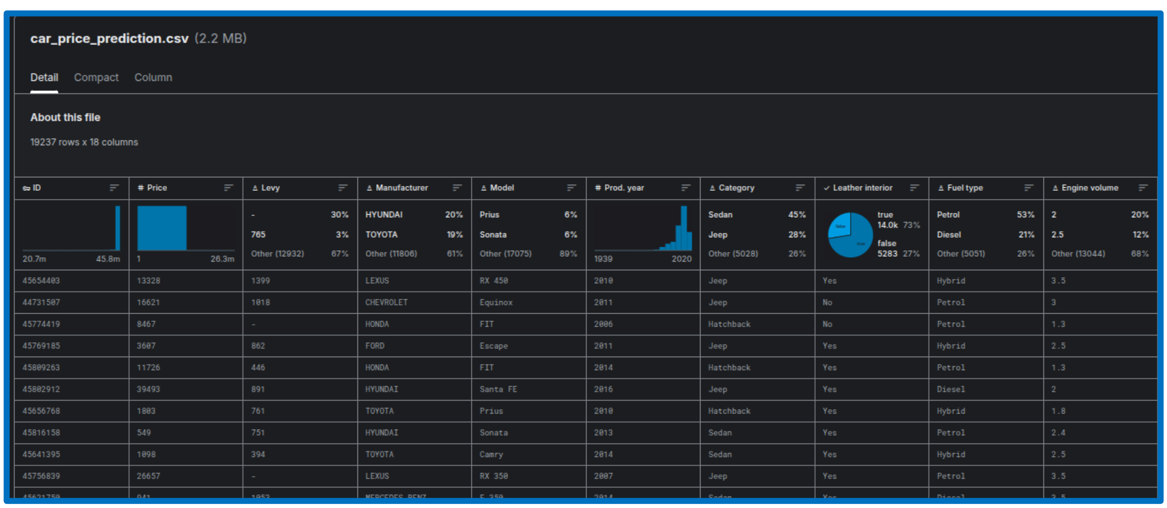
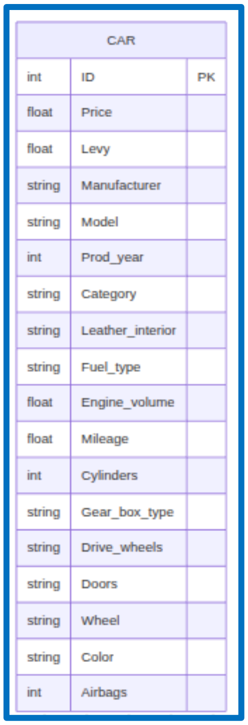


Fig 3. Car Price Prediction Screenshot



CAR		
int	ID	PK
float	Price	
float	Levy	
string	Manufacturer	
string	Model	
int	Prod_year	
string	Category	
string	Leather_interior	
string	Fuel_type	
float	Engine_volume	
float	Mileage	
int	Cylinders	
string	Gear_box_type	
string	Drive_wheels	
string	Doors	
string	Wheel	
string	Color	
int	Airbags	

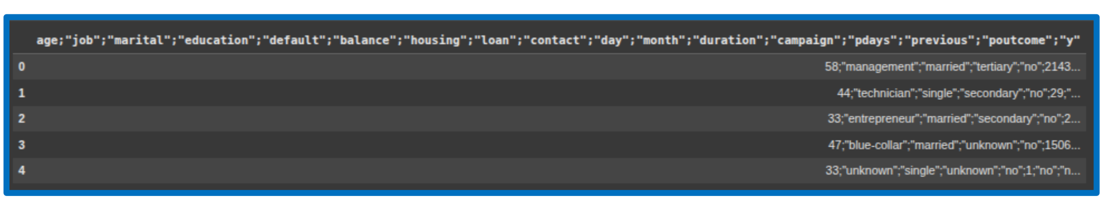
Fig 4. ERD of Car Price Prediction

This **Car Price Prediction** dataset contains **19,237 rows** and **18 columns**. Each row is a car listing with details about its price and specifications. The features include:

- **Car info**: ID, Manufacturer, Model, Production year, Category.
- **Technical details**: Engine volume, Fuel type, Cylinders, Gear box type, Drive wheels, Mileage, Airbags.
- **Other details**: Levy, Leather interior, Doors, Wheel orientation, Color.
- **Target variable**: **Price** (the column we want to predict).

The dataset mixes **numerical, categorical, and text values**, with some missing or irregular entries (e.g., Levy has dashes, Mileage has "km" suffix). It is used for building ML models that estimate a car's market price based on its features.

Dataset 3: Bank Marketing



	age;"job";"marital";"education";"default";"balance";"housing";"loan";"contact";"day";"month";"duration";"campaign";"pdays";"previous";"poutcome";"y"
0	58;"management";"married";"tertiary";"no";2143...
1	44;"technician";"single";"secondary";"no";29;"...
2	33;"entrepreneur";"married";"secondary";"no";2...
3	47;"blue-collar";"married";"unknown";"no";1506...
4	33;"unknown";"single";"unknown";"no";1;"no";"n...

Fig 5. Bank Marketing Screenshot

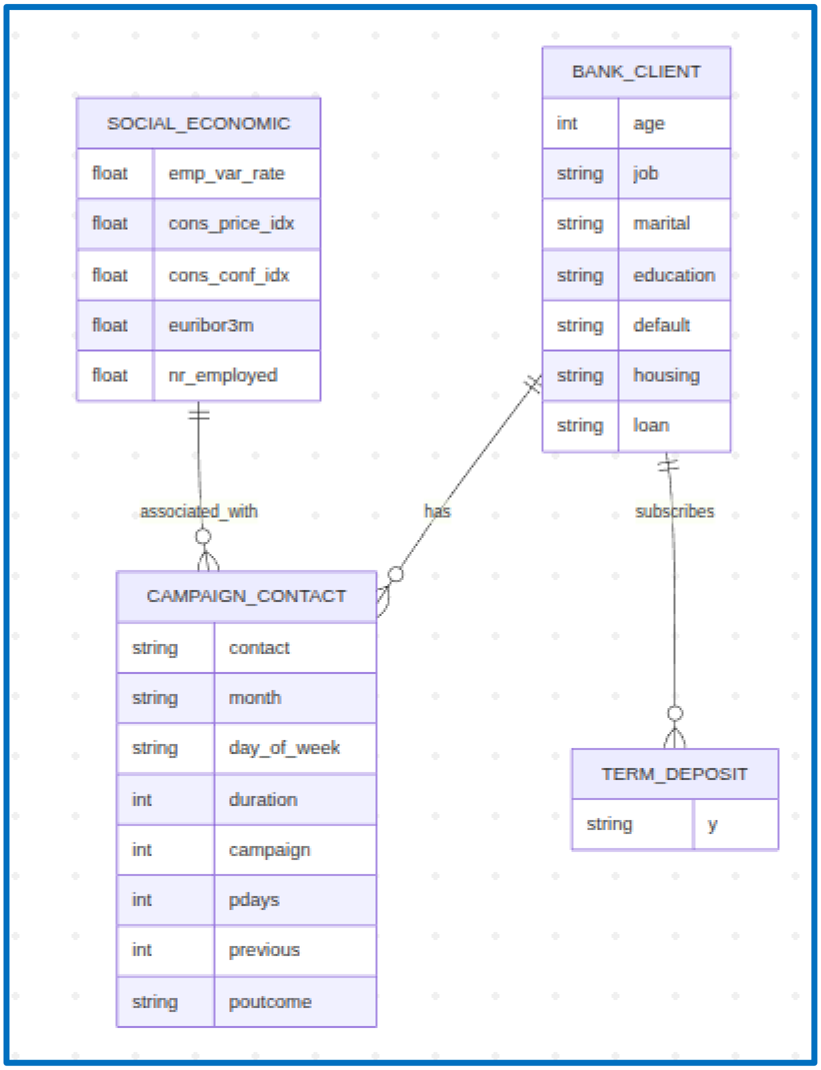


Fig 6. ERD of Bank Marketing Screenshot

This dataset got 45,211 rows and 16 features. Some are numbers like age or call duration, while others are category type like job, marital status, or education. The main task is binary classification, predicting if a client subscribe to a term deposit (y variable). The challenges are imbalanced data, handling missing values, and encoding many categorical fields. Still, the dataset is quite useful for doing exploratory analysis, feature engineering, and training ML models to improve marketing strategy and targeting.

Dataset Selection

The datasets were looked at mainly by checking data quality, richness of features, business use, and how well they match the project goal.

- The **Car Price Prediction dataset** has 19,237 rows with 18 features and gives a lot of details about cars. It is good for finding the market value of vehicles, but it is more about consumer sales and outside market factors like levy or mileage. These are very specific to the auto industry and don't really fit with human resource or salary planning problems. It also has messy values, like mileage written with "km" or levy fields shown as dashes, so it needs heavy cleaning before use, which makes it less smooth for building a clear pipeline.

- The **Bank Marketing dataset**, with 45,211 rows and 16 features, is a popular one for binary classification, especially to see if customers subscribe to deposits. It has good feature variety, but it is centered on finance and marketing instead of salary or workforce patterns. Its main issues are imbalanced data and too many categorical variables, which are good challenges but not really linked to predicting pay.
- The **Salary Prediction of Data Professions dataset** fits the project much better. It has 2,639 rows and 13 features about age, gender, designation, unit, experience, performance ratings, and salary. This connects directly to the real business problem of making fair and competitive salaries in the fast growing data industry. It gives useful insights for HR planning, pay benchmarking, and retention, making it the most suitable dataset.

Data Storage Design (Onebigtable over Star Schema)

RAW_SALARY			
INT	ID	PK	Surrogate key (auto-increment)
STRING	FIRST_NAME		Given name
STRING	LAST_NAME		Family name
STRING	SEX		F/M (as text in raw)
STRING	DOJ		Date of joining (raw string)
STRING	CURRENT_DATE		Snapshot/current date (raw string)
STRING	DESIGNATION		Role title
FLOAT	AGE		Employee age (years)
FLOAT	SALARY		Annual salary (numeric)
STRING	UNIT		Business unit name
FLOAT	LEAVES_USED		Leaves taken
FLOAT	LEAVES_REMAINING		Leaves left
FLOAT	RATINGS		Performance rating
FLOAT	PAST_EXP		Prior experience (years)

Fig 7. One Big Table (Salary Prediction of Data Professions)

The dataset, with 2,639 rows and 13 columns, was stored in a single relational table called **raw_salary**. This one big table approach is more appropriate than a star schema for the nature of the data. The dataset is already flat, with each row representing one employee's complete record, including demographics, job details, performance indicators, and salary. A star schema would only complicate the design by breaking categorical fields such as designation, unit, or sex into separate dimension tables, introducing unnecessary joins and maintenance overhead. For machine learning pipelines, it is essential to have one row correspond to one training example, with all features materialized in a single structure. Using a big table not only simplifies preprocessing in pandas but also ensures feature stability for model training and inference. It also speeds up drift detection with Evidently AI, which expects wide DataFrames for comparison. Given the dataset size and requirements, a one big table schema is the most efficient and reliable design choice.

ELT Process (ELT over ETL)

In this project, the workflow follows ELT (Extract–Load–Transform) rather than ETL. The raw dataset is first loaded into MariaDB without any preprocessing, ensuring that the original data is always preserved as a reliable source of truth. Transformations such as handling missing values, encoding categorical features, computing tenure, imputing numerics, and splitting into train and test sets are carried out later in Python using pandas and scikit-learn. These tasks are far more effective in Python than in SQL because they rely on machine learning–oriented libraries that SQL cannot replicate easily. By keeping the transformations after loading, the pipeline becomes more reproducible and auditable. Any change in preprocessing logic can be reapplied without re-extracting or risking permanent data loss. This design also avoids training/serving mismatches, since the exact same Python transformations used during training are applied during inference, ensuring consistency in model predictions.

IV. Data Engineering Pipeline Plan

Pipeline Overview

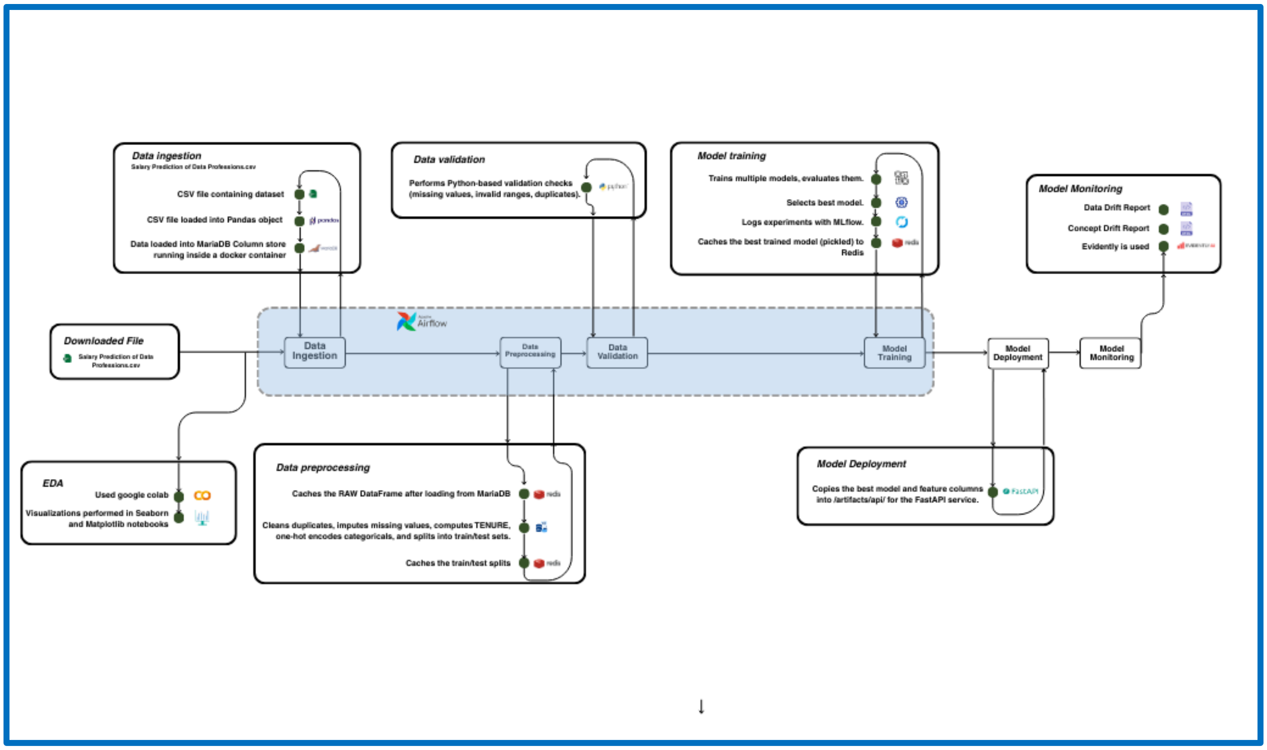


Fig 8. Pipeline Overview Image

The pipeline for the Salary Prediction of Data Professions dataset is designed as an automated, end-to-end workflow using Apache Airflow. It starts with creating a raw table in MariaDB to store the dataset, ensuring the structure is preserved. After that, the data is ingested from CSV into the database and passed through a preprocessing stage, where missing values are handled, categorical variables are encoded, and features like tenure are engineered. Once cleaned, the dataset is split into training and testing sets and used for model training, with the best model stored as an artifact. The model is then deployed for predictions, and continuous monitoring steps check for both data drift and concept drift, ensuring the system stays reliable over time.

Data ingestion

In the data ingestion stage of my project I first took the Salary Prediction of Data Professions dataset from the CSV file and loaded it into MariaDB. I created a raw table which matched the structure of the dataset including columns like age, designation, salary, past experience and others. By doing this I was able to store the raw data in a structured format without making any changes on it. This way I made sure the original dataset was preserved so later preprocessing and analysis steps could be carried out in a reliable and consistent way.

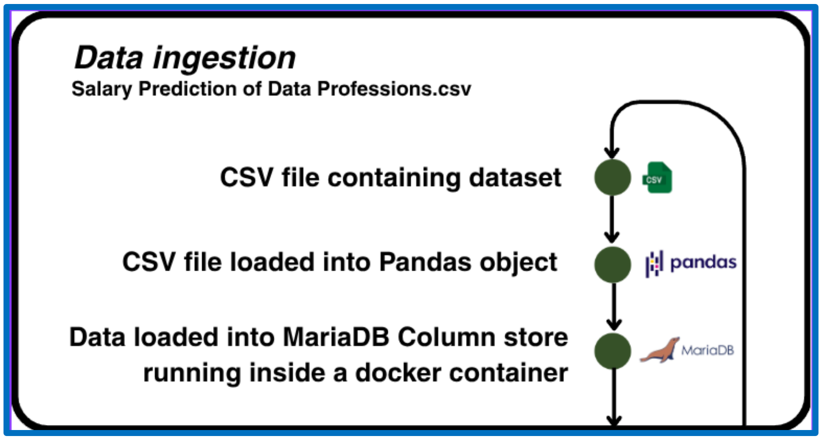


Fig 9. Data ingestion

EDA

In the EDA stage of my project, I explored the Salary Prediction of Data Professions dataset to understand the patterns and relationships in the data. I first looked at the overall salary distribution using a histogram, which gave me an idea of how salaries are spread across employees. Then I used boxplots to compare salary differences by job role and gender, showing how positions and gender impact income levels. A scatterplot of salary by age helped to check if age and salary have a trend, while barplots of performance ratings against salary revealed how ratings affect pay. These visualizations guided the later preprocessing and modeling steps.

Fig 10. Salary by Age

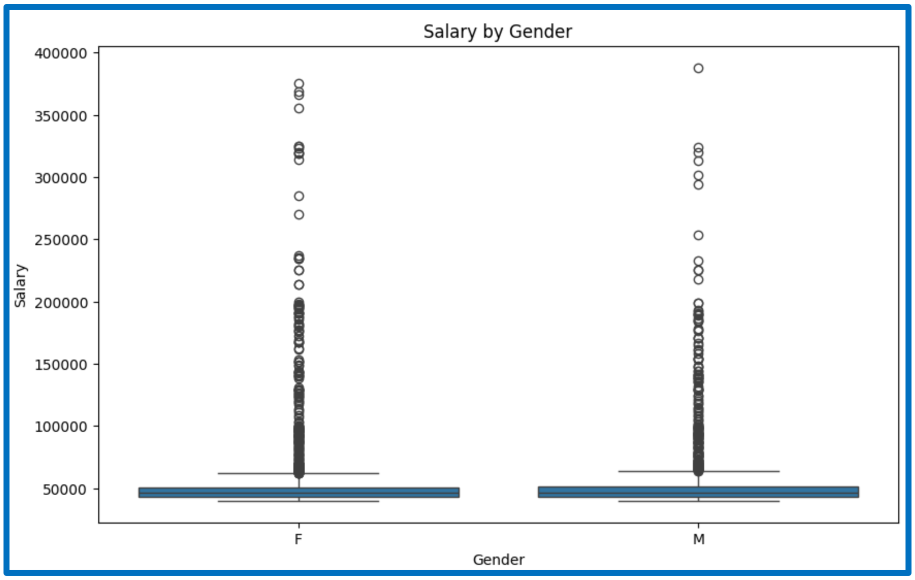


Fig 11. Salary by Gender

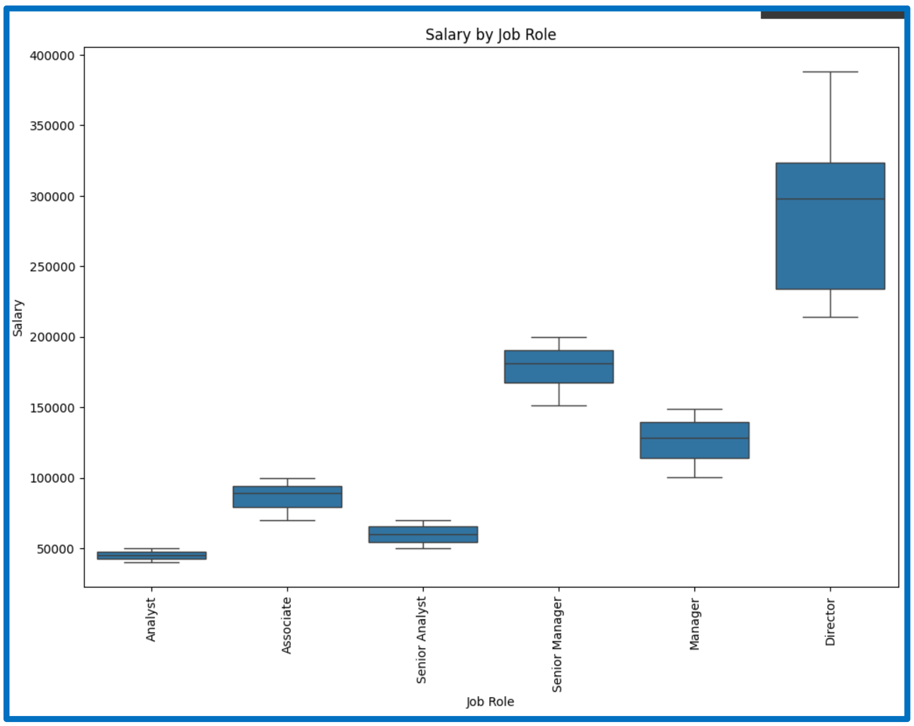


Fig 12. Salary by Job Role

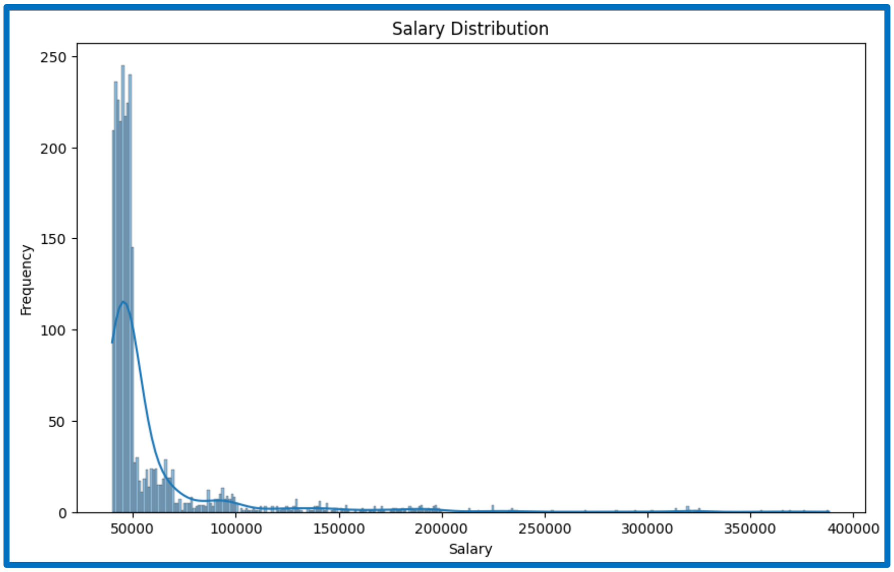


Fig 13. Salary Distribution

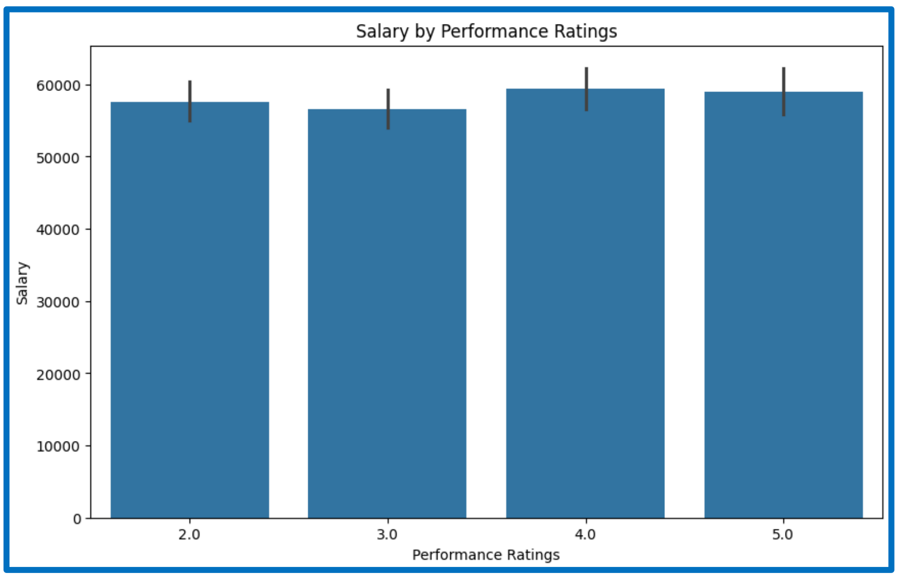


Fig 14. Salary by Performance Ratings

Model Training

In the model training stage, I used the processed salary dataset stored in MariaDB. The features and the target column, salary, were split into training and testing sets. Missing values in numerical attributes were handled using median imputation, while any remaining non-numeric fields were filled with the most frequent values. I then trained four regression models: Linear Regression, Decision Tree, Random Forest, and Gradient Boosting. Their performance was compared mainly using Mean Absolute Error, along with RMSE and R^2 as supporting metrics. All results and parameters were logged into MLflow for tracking. Finally, the best performing model was saved as an artifact, registered in MLflow, and also cached in Redis for fast retrieval in deployment.

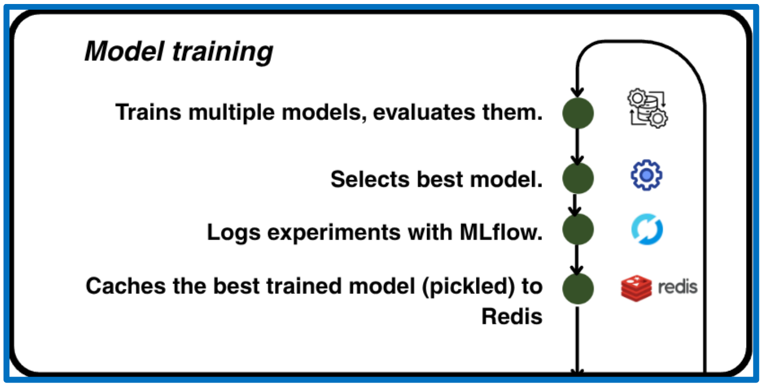


Fig 15. Model Training

Model Deployment

For deployment, I kept it simple and reliable. After training, the best model is saved as best_model.pkl in the artifacts folder. The deployment step just copies that file into the API directory as salary_model.pkl (/opt/airflow/dags/artifacts/api). The code first ensures the API folder exists, then checks the trained model path, and finally copies it over. The FastAPI service loads salary_model.pkl on startup, together with model_columns.json, so the /predict endpoint can accept JSON rows and return salary predictions. This approach avoids rebuilding the container and Airflow updates the file, the API serves the latest model.

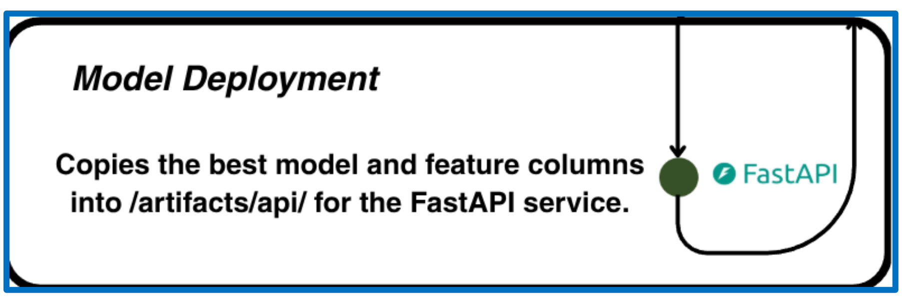


Fig 16. Model Deployment

Model Monitoring

Model monitoring in this project is carried out using Evidently to detect both data drift and concept drift. Data drift is checked by comparing the distribution of features in the new dataset with the original training data stored in MariaDB. Evidently generates detailed metrics, such as how many columns show drift and the percentage of drifted features, and saves the results as HTML and CSV reports. Concept drift is monitored by looking at changes in the distribution of the target variable or predicted salary in the incoming data. These checks help to ensure that the model remains accurate and consistent over time.

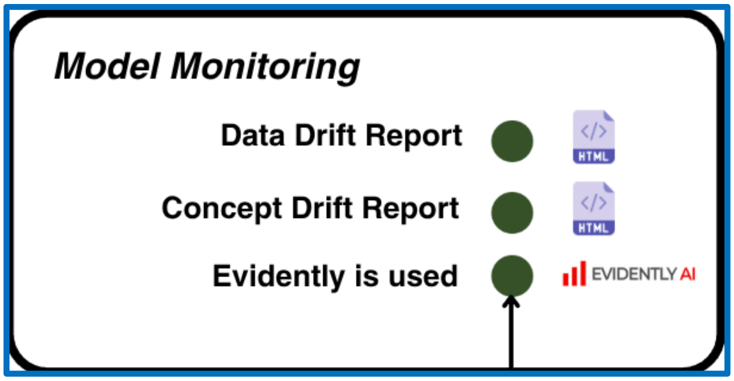


Fig 17. Model Monitoring

V. PIPELINE IMPLEMENTATION AND MODEL DEPLOYMENT

Tools

I used Apache Airflow to automate and schedule the tasks across my pipeline. I stored both raw and processed data in MariaDB, while Redis was used for caching data so that access became much faster. For preprocessing and feature engineering, I relied on Pandas and NumPy. I used Scikit-learn to train different machine learning models and evaluate them with proper metrics. MLflow helped me track experiments, log model versions, and manage them effectively. To serve predictions, I used FastAPI together with Uvicorn. Finally, I applied Evidently to monitor both data drift and concept drift, ensuring the model stayed reliable.

Apache Airflow

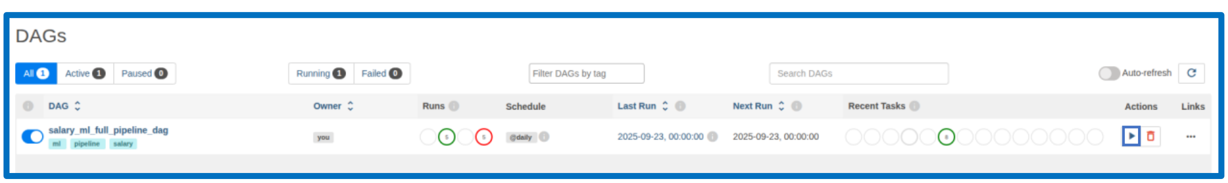


Fig 18. Airflow screenshot - 1

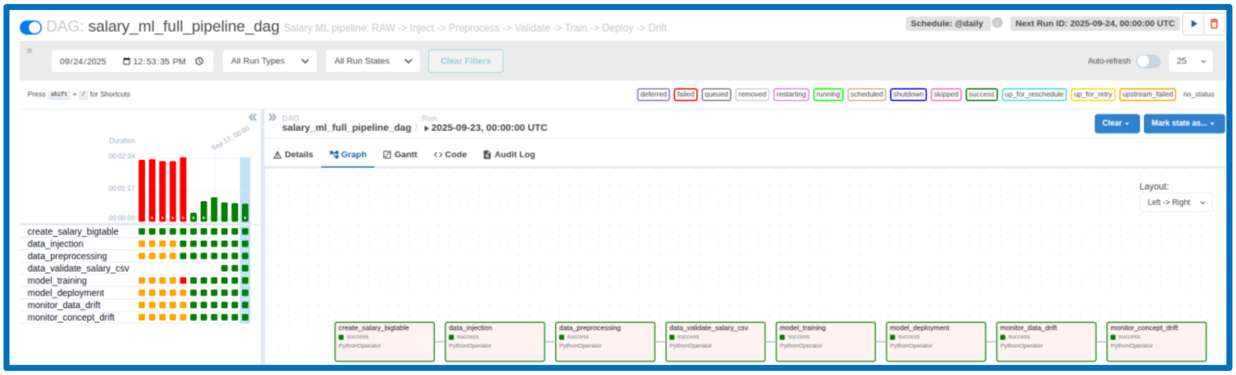


Fig 19. Airflow screenshot – 2

After I triggered the **salary_ml_full_pipeline_dag** in Airflow, the pipeline ran step by step in the right order, beginning with creating the raw table and moving through data ingestion, preprocessing, validation, model training, deployment, and finally monitoring for both data drift and concept drift. In the graph view, each stage is displayed as a separate box, and all boxes turned green, showing that every step finished successfully. The duration panel on the left also shows how long each task took. In the DAGs overview, the pipeline is set to run daily, and the most recent run also shows all tasks completed without errors. This proves the workflow is fully automated and functioning smoothly from start to finish.

Pipeline Implemented

```python
with DAG(
    dag_id="salary_ml_full_pipeline_dag",
    description="Salary ML pipeline: RAW -> Inject -> Preprocess -> Validate -> Train -> Deploy -> Drift",
    default_args=default_args,
    schedule="@daily",
    start_date=datetime(2025, 9, 5),
    catchup=False,
    max_active_runs=1,
    tags=["ml", "pipeline", "salary"],
) as dag:

    # 1) Ensure target table exists (schema bootstrap)
    schema = PythonOperator(
        task_id="create_salary_bigtable",
        python_callable=schema_table.create_bigtable,
        op_kwargs={
            "db_url": DB_URL,
            "table_name": RAW_TABLE,
        },
    )

    # 2) Load the raw CSV into MariaDB
    inject = PythonOperator(
        task_id="data_injection",
        python_callable=data_injection.run,
        op_kwargs={
            "db_url": DB_URL,
            "csv_path": RAW_CSV,
            "table_name": RAW_TABLE,
            "if_exists": "replace",
        },
    )

    # 3) Preprocess (feature engineering, impute, one-hot, split, persist)
    prep = PythonOperator(
        task_id="data_preprocessing",
        python_callable=preprocessing.run,
        op_kwargs={
            "db_url": DB_URL,
            "raw_table": RAW_TABLE,
            "gold_table": GOLD_TABLE,
            "output_csv": PROCESSED_CSV,
            "train_table": TRAIN_TABLE,
            "test_table": TEST_TABLE,
            "train_csv": TRAIN_CSV,
            "test_csv": TEST_CSV,
            "train_parquet": TRAIN_PARQUET,
            "test_parquet": TEST_PARQUET,
            "redis_host": "redis",
        },
    )

    # 4) Validate the CLEANED CSV (post-preprocessing)
    validate = PythonOperator(
        task_id="data_validate_salary_csv",
        python_callable=data_validation.run_salary_ge_validation,  # alias to your validator
        op_kwargs={
            "salary_csv_path": PROCESSED_CSV,
        },
    )

    # 5) Train models (with imputation inside modeling pipeline) and log to MLflow
    train = PythonOperator(
        task_id="model_training",
        python_callable=model_train.run,
        op_kwargs={
            "db_url": DB_URL,
            "gold_table": GOLD_TABLE,
            "train_table": TRAIN_TABLE,
            "model_path": MODEL_PATH,
            "cols_path": COLS_PATH,
            "mlflow_uri": MLFLOW_URI,
            "mlflow_experiment": MLFLOW_EXPERIMENT,
        },
    )

    # 6) Deploy best model artifact to the API folder
    deploy = PythonOperator(
        task_id="model_deployment",
        python_callable=model_deploy.run,
        op_kwargs={
            "trained_model": MODEL_PATH,
            "api_folder": "/opt/airflow/dags/artifacts/api",
        },
    )

    # 7) Monitor drift (both write into REPORTS_DIR)
    drift_data = PythonOperator(
        task_id="monitor_data_drift",
        python_callable=datadrift.monitor_data_drift,
        op_kwargs={
            "db_url": DB_URL,
            "reference_table": GOLD_TABLE,
            "current_csv": NEW_DATA_CSV,
            "cols_path": COLS_PATH,
            "reports_dir": REPORTS_DIR,
        },
    )

    drift_concept = PythonOperator(
        task_id="monitor_concept_drift",
        python_callable=conceptdrift.monitor_concept_drift,
        op_kwargs={
            "db_url": DB_URL,
            "reference_table": GOLD_TABLE,
            "current_csv": NEW_DATA_CSV,
            "reports_dir": REPORTS_DIR,
        },
    )

    # Orchestration
    schema >> inject >> prep >> validate >> train >> deploy >> drift_data >> drift_concept
```

Fig 20. Screenshot of salary_full_pipeline_dag.py

1) Data Ingestion

I load the source CSV into MariaDB using SQLAlchemy. First, the script reads /opt/airflow/dags/data/Salary Prediction of Data Professions.csv. Then I normalize headers (make UPPER, and replace spaces with underscores) so later steps stay consistent. I validate the schema by checking all expected columns exist; if anything is missing, it stops with a clear error. After that, I write the dataframe to the raw_salary table (if_exists="replace" makes runs idempotent). Simple logging records how many rows were inserted. This creates a clean raw layer for downstream preprocessing and modeling.

```python
import pandas as pd
from sqlalchemy import create_engine
import logging

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def run(
    db_url: str = None,
    csv_path: str = "/opt/airflow/dags/data/Salary Prediction of Data Professions.csv",
    table_name: str = "raw_salary",
    if_exists: str = "replace"
):
    """
    Load CSV -> MariaDB raw table.
    """
    db_url = db_url or "mysql+pymysql://app:app@mariadb:3306/analytics"
    engine = create_engine(db_url)

    df = pd.read_csv(csv_path)
    # Normalize headers (UPPER & spaces -> underscores used later)
    df.columns = [c.upper().replace(' ', '_') for c in df.columns]
    # Ensure expected columns exist (pass-through if already present)
    expected = ['FIRST_NAME','LAST_NAME','SEX','DOJ','CURRENT_DATE','DESIGNATION','AGE',
                'SALARY','UNIT','LEAVES_USED','LEAVES_REMAINING','RATINGS','PAST_EXP']
    missing = [c for c in expected if c not in df.columns]
    if missing:
        raise ValueError(f"Missing columns in CSV: {missing}")

    df.to_sql(table_name, con=engine, if_exists=if_exists, index=False)
    logger.info(f"Injected {len(df)} rows into {table_name}.")
    return True
```

Fig 21. Screenshot of data ingestion

2) *Data Preprocessing*

I loaded the raw table from Redis (fast) or MariaDB (fallback), normalized column names, and removed exact duplicate rows. I replaced any ±∞ strings with NaN, guaranteed key columns exist, parsed DOJ and CURRENT_DATE to compute TENURE in years, and coerced numeric types. I clipped out-of-range values (e.g., AGE to 16–85, leaves ≥0).

I imputed missing values: RATINGS by mean, PAST_EXP by median, and AGE/LEAVES_USED/LEAVES_REMAINING/TENURE by median. I dropped rows with missing or non-positive SALARY. I one-hot encoded SEX, DESIGNATION, and UNIT (drop_first), built the GOLD dataset, saved it to MariaDB and CSV, split into train/test (cached in Redis), wrote splits to DB/CSV/Parquet, and exported model_columns.json.

```python
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def _parse_date(series):
    s1 = pd.to_datetime(series, errors="coerce", dayfirst=True)
    s2 = pd.to_datetime(series, errors="coerce", dayfirst=False)
    return s1.fillna(s2)

def _engine(url=None):
    return create_engine(url or "mysql+pymysql://app:app@mariadb:3306/analytics")

def _coerce_numeric(df: pd.DataFrame, cols: list[str]) -> list[str]:
    present = [c for c in cols if c in df.columns]
    for c in present:
        df[c] = pd.to_numeric(df[c], errors="coerce")
    return present

def run(
    db_url: str = None,
    raw_table: str = "raw_salary",
    gold_table: str = "gold_salary_features",
    output_csv: str = "/opt/airflow/dags/data/processed_salary.csv",
    train_table: str = "salary_train_data",
    test_table: str = "salary_test_data",
    train_csv: str = "/opt/airflow/dags/data/salary_train_data.csv",
    test_csv: str = "/opt/airflow/dags/data/salary_test_data.csv",
    train_parquet: str = "/opt/airflow/dags/data/salary_train_data.parquet",
    test_parquet: str = "/opt/airflow/dags/data/salary_test_data.parquet",
    redis_host: str = "redis", redis_port: int = 6379, redis_db: int = 0,
    refresh_splits: bool = True,
):
    """
    Preprocessing:
      - Load RAW from DB (or Redis)
      - Normalize headers
      - Drop full-row duplicates
      - Replace +/-inf strings with NaN, coerce numerics
      - Compute TENURE
      - Impute numerics (AGE, LEAVES_*, RATINGS, PAST_EXP, TENURE)
      - Clip to sane ranges; drop rows with SALARY NaN or <=0
      - One-hot SEX/DESIGNATION/UNIT (drop_first)
      - Persist GOLD; create & persist train/test; cache artifacts
      - Save model_columns.json for inference alignment
    """
    r = redis.Redis(host=redis_host, port=redis_port, db=redis_db, decode_responses=False)
    raw_key = f"salary:{raw_table}"
    train_key, test_key = "salary:train_df", "salary:test_df"

    # ---- Load RAW ----
    cached = r.get(raw_key)
    if cached:
        df = pickle.loads(cached)
        logger.info("Loaded RAW from Redis cache.")
    else:
        df = pd.read_sql(f"SELECT * FROM {raw_table}", con=_engine(db_url))
        r.set(raw_key, pickle.dumps(df))
        logger.info("Cached RAW in Redis.")

    # ---- Normalize headers ----
    df.columns = [c.upper().replace(" ", "_") for c in df.columns]

    # ---- Drop exact duplicates ----
    before = len(df)
    df = df.drop_duplicates()
    dropped = before - len(df)
    if dropped:
        logger.info("Dropped %d duplicate rows", dropped)

    # ---- Replace infinities & coerce numerics ----
    df = df.replace([np.inf, -np.inf, "inf", "-inf", "Infinity", "-Infinity"], np.nan)

    # Ensure columns exist
    for c in ["AGE", "LEAVES_USED", "LEAVES_REMAINING", "RATINGS", "PAST_EXP", "SALARY"]:
        if c not in df.columns:
            df[c] = np.nan

    # Compute TENURE (years)
    doj = _parse_date(df.get("DOJ"))
    curr = _parse_date(df.get("CURRENT_DATE"))
    df["TENURE"] = ((curr - doj).dt.days / 365.25).astype(float)
```

Fig 22. Screenshot-1 of data preprocessing

```python

    # Coerce numeric dtypes
    num_cols = ["AGE", "LEAVES_USED", "LEAVES_REMAINING", "RATINGS", "PAST_EXP", "TENURE", "SALARY"]
    present_num = _coerce_numeric(df, num_cols)

    # Clip to sane ranges (after coercion)
    if "AGE" in df.columns:
        df["AGE"] = df["AGE"].clip(lower=16, upper=85)
    for c in ("LEAVES_USED", "LEAVES_REMAINING"):
        if c in df.columns:
            df[c] = df[c].clip(lower=0)

    # ---- Impute numerics ----
    # RATINGS: mean, PAST_EXP: median, others: median
    if "RATINGS" in df.columns:
        df["RATINGS"] = SimpleImputer(strategy="mean").fit_transform(df[["RATINGS"]])
    if "PAST_EXP" in df.columns:
        df["PAST_EXP"] = SimpleImputer(strategy="median").fit_transform(df[["PAST_EXP"]])

    for c in ["AGE", "LEAVES_USED", "LEAVES_REMAINING", "TENURE"]:
        if c in df.columns:
            df[c] = SimpleImputer(strategy="median").fit_transform(df[[c]])

    # Target guardrails: drop NaN or non-positive SALARY
    if "SALARY" in df.columns:
        bad = df["SALARY"].isna().sum() + (df["SALARY"] <= 0).sum()
        if bad:
            logger.info("Dropping %d rows with SALARY NaN or <= 0", bad)
            df = df[df["SALARY"].notna() & (df["SALARY"] > 0)]

    # ---- One-hot categoricals ----
    cat_cols = ["SEX", "DESIGNATION", "UNIT"]
    for c in cat_cols:
        if c not in df.columns:
            df[c] = "Unknown"
    dummies = pd.get_dummies(df[cat_cols], drop_first=True, dtype=np.int8)

    # ---- Assemble GOLD ----
    X_num = df[["AGE", "LEAVES_USED", "LEAVES_REMAINING", "RATINGS", "PAST_EXP", "TENURE"]].copy()
    target = df[["SALARY"]].copy()
    X = pd.concat([X_num, dummies], axis=1)
    gold = pd.concat([X, target], axis=1)

    # ---- Persist GOLD ----
    engine = _engine(db_url)
    gold.to_sql(gold_table, con=engine, if_exists="replace", index=False)
    os.makedirs(os.path.dirname(output_csv), exist_ok=True)
    gold.to_csv(output_csv, index=False)
    logger.info("Wrote GOLD -> %s, %s", gold_table, output_csv)

    # ---- Train/Test split ----
    if not refresh_splits and r.exists(train_key) and r.exists(test_key):
        train_df = pickle.loads(r.get(train_key))
        test_df = pickle.loads(r.get(test_key))
        logger.info("Loaded train/test from Redis cache.")
    else:
        train_df, test_df = train_test_split(gold, test_size=0.30, random_state=42)
        r.set(train_key, pickle.dumps(train_df))
        r.set(test_key, pickle.dumps(test_df))
        logger.info("Cached new train/test in Redis.")

    # ---- Persist splits ----
    train_df.to_sql(train_table, con=engine, if_exists="replace", index=False)
    test_df.to_sql(test_table, con=engine, if_exists="replace", index=False)
    train_df.to_csv(train_csv, index=False)
    test_df.to_csv(test_csv, index=False)
    pq.write_table(pa.Table.from_pandas(train_df), train_parquet)
    pq.write_table(pa.Table.from_pandas(test_df), test_parquet)
    logger.info("Saved train/test to DB/CSV/Parquet")

    # ---- Save feature column order for serving ----
    feature_cols = [c for c in gold.columns if c != "SALARY"]
    cols_path = "/opt/airflow/dags/artifacts/model_columns.json"
    os.makedirs(os.path.dirname(cols_path), exist_ok=True)
    with open(cols_path, "w") as f:
        json.dump(feature_cols, f)
    logger.info("Saved feature column list -> %s", cols_path)

    return True
```

Fig 23. Screenshot-2 of data preprocessing

3) *Data Validation*

I validated the cleaned CSV **after preprocessing** using a small custom Python checker (not Great Expectations). The task loads /opt/airflow/dags/data/processed_salary.csv, normalises headers, and reads the expected feature list from /opt/airflow/dags/artifacts/model_columns.json. It then enforces: required columns (features + SALARY) must exist; flags any fully duplicated rows; scans all numeric columns for NaN and ±∞; warns if any categorical column has extremely high cardinality (>1000 uniques). For the target, it checks SALARY for NaN, non-positive values, and suspicious outliers (>10,000,000). It also flags any feature with >20% missing. A JSON report is written to /opt/airflow/dags/artifacts/validation_report.json, and the task **fails** if issues are found (configurable).

```python
logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)

DEFAULT_CSV = "/opt/airflow/dags/data/processed_salary.csv"
DEFAULT_COLS = "/opt/airflow/dags/artifacts/model_columns.json"
DEFAULT_REPORT = "/opt/airflow/dags/artifacts/validation_report.json"


def _read_csv(csv_path: str) -> pd.DataFrame:
    if not os.path.exists(csv_path):
        raise AirflowFailException(f"CSV not found: {csv_path}")
    df = pd.read_csv(csv_path)
    if df.empty:
        raise AirflowFailException(f"CSV has no rows: {csv_path}")
    # normalize column names a bit
    df.columns = [c.strip() for c in df.columns]
    return df


def _load_feature_list(cols_path: str):
    if os.path.exists(cols_path):
        with open(cols_path) as f:
            return json.load(f)
    return None


def _validate(df: pd.DataFrame, features: list | None) -> list[str]:
    issues: list[str] = []

    # 1) required columns
    required = (features or []) + ["SALARY"]
    missing = [c for c in required if c not in df.columns]
    if missing:
        issues.append(f"Missing required columns: {missing}")

    # 2) duplicates
    dups = df.duplicated().sum()
    if dups > 0:
        issues.append(f"{dups} fully duplicated rows")

    # 3) numeric sanity (NaN/inf) and missing ratios
    num_cols = df.select_dtypes(include=[np.number]).columns.tolist()
    for c in num_cols:
        n_nan = pd.isna(df[c]).sum()
        n_inf = (~np.isfinite(df[c].astype(float))).sum()
        if n_nan > 0:
            issues.append(f"Column '{c}' has {n_nan} NaN values")
        if n_inf > 0:
            issues.append(f"Column '{c}' has {n_inf} +/-inf values")

    # 4) categorical high cardinality warning (can explode one-hot)
    cat_cols = df.select_dtypes(include=["object"]).columns.tolist()
    for c in cat_cols:
        uniq = df[c].nunique(dropna=True)
        if uniq > 1000:
            issues.append(f"Categorical '{c}' has very high cardinality ({uniq})")

    # 5) target checks
    if "SALARY" in df.columns:
        n_nan = pd.isna(df["SALARY"]).sum()
        if n_nan > 0:
            issues.append(f"SALARY has {n_nan} NaN values")
        nonpos = int((df["SALARY"] <= 0).sum())
        if nonpos > 0:
            issues.append(f"SALARY has {nonpos} non-positive values")
        too_high = int((df["SALARY"] > 1e7).sum())
        if too_high > 0:
            issues.append(f"SALARY has {too_high} values > 10,000,000 (suspicious)")

    # 6) feature-specific missing ratio guardrails (if we know features)
    if features:
        for c in features:
            if c in df.columns:
                miss_ratio = float(pd.isna(df[c]).mean())
                if miss_ratio > 0.20:
                    issues.append(f"Feature '{c}' missing ratio {miss_ratio:.1%} (>20%)")

    return issues
```

Fig 24. Screenshot-1 of data validation

```python
def run_salary_ge_validation(
    csv_path: str = DEFAULT_CSV,
    cols_path: str = DEFAULT_COLS,
    report_path: str = DEFAULT_REPORT,
    fail_on_warnings: bool = True,
):
    """
    Validates the training CSV robustly (schema presence, NaN/inf, duplicates, target sanity,
    high-cardinality categoricals, per-feature missing ratios). Writes a JSON report and
    fails the task if any issues are found (configurable via fail_on_warnings).
    """
    logger.info("Starting data validation for %s", csv_path)
    df = _read_csv(csv_path)
    features = _load_feature_list(cols_path)

    issues = _validate(df, features)

    summary = {
        "csv_path": csv_path,
        "rows": int(len(df)),
        "cols": int(df.shape[1]),
        "features_expected": features,
        "columns_actual": list(df.columns),
        "issues_count": len(issues),
        "issues": issues,
        "status": "PASS" if not issues else "FAIL" if fail_on_warnings else "WARN",
    }

    os.makedirs(os.path.dirname(report_path), exist_ok=True)
    with open(report_path, "w") as f:
        json.dump(summary, f, indent=2)
    logger.info("Validation report written to %s", report_path)

    if issues and fail_on_warnings:
        raise AirflowFailException(
            "Validation failed:\n" + "\n".join(f"- {m}" for m in issues)
        )

    logger.info("Validation status: %s", summary["status"])
    return summary["status"]
    run_salary_ge_validation = run_salary_validation
```

Fig 25. Screenshot-2 of data validation

4) Model Training

I loaded features/targets from the pre-split salary_train_data table (fallback to gold_salary_features) using the column list in model_columns.json. I split into train/test (70/30) and handled missing data: numeric columns with a median SimpleImputer, any non-numeric with most-frequent. I trained four regressors Linear Regression, Decision Tree, Random Forest (300 trees), and Gradient Boosting then evaluated each with MAE, MSE, RMSE, and R^2, selecting the best by **MAE**. I logged params/metrics and artifacts to **MLflow** (experiment salary_model_training), saved best_model.pkl, model_meta.json, and an imputers.pkl bundle, and cached the best model in **Redis** for quick reuse.

```python
def _engine(url=None):
    return create_engine(url or "mysql+pymysql://app:app@mariadb:3306/analytics")

def _metrics(y_true, y_pred):
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = mse ** 0.5
    r2 = r2_score(y_true, y_pred)
    return {"mae": mae, "mse": mse, "rmse": rmse, "r2": r2}

def run(
    db_url: str = None,
    gold_table: str = "gold_salary_features",
    train_table: str = "salary_train_data",
    model_path: str = "/opt/airflow/dags/artifacts/best_model.pkl",
    cols_path: str = "/opt/airflow/dags/artifacts/model_columns.json",
    mlflow_uri: str = None,
    mlflow_experiment: str = "salary_model_training",
    redis_host: str = "redis", redis_port: int = 6379, redis_db: int = 0
):
    """
    Train 4 regressors, pick best by MAE.
    - Reads pre-split training table if present, else samples from GOLD.
    - Imputes missing values (median for numeric, most-frequent for others).
    - Logs runs to MLflow and caches best model in Redis.
    - Saves imputers bundle to artifacts so inference can reuse it.
    """
    mlflow.set_tracking_uri(mlflow_uri or os.getenv("MLFLOW_TRACKING_URI", "http://mlflow:5000"))
    mlflow.set_experiment(mlflow_experiment)

    r = redis.Redis(host=redis_host, port=redis_port, db=redis_db, decode_responses=False)
    engine = _engine(db_url)

    # Load features/target
    if engine.dialect.has_table(engine.connect(), train_table):
        df = pd.read_sql(f"SELECT * FROM {train_table}", con=engine)
    else:
        df = pd.read_sql(f"SELECT * FROM {gold_table}", con=engine)
        df, _ = train_test_split(df, test_size=0.3, random_state=42)

    with open(cols_path) as f:
        feature_cols = json.load(f)
    X = df[feature_cols].copy()
    y = df["SALARY"].copy()

    # Split first, then fit imputers on the train split only
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.30, random_state=42)

    # ----- NEW: Imputation (handles your NaN crash) ------------------------
    num_cols = X_train.select_dtypes(include=[np.number]).columns.tolist()
    cat_cols = [c for c in X_train.columns if c not in num_cols]

    num_imputer = None
    cat_imputer = None

    # numeric -> median
    if num_cols:
        num_imputer = SimpleImputer(strategy="median")
        X_train[num_cols] = num_imputer.fit_transform(X_train[num_cols])
        X_test[num_cols] = num_imputer.transform(X_test[num_cols])

    # non-numeric -> most frequent (safe noop if you already have only numeric)
    if cat_cols:
        cat_imputer = SimpleImputer(strategy="most_frequent")
        X_train[cat_cols] = cat_imputer.fit_transform(X_train[cat_cols])
        X_test[cat_cols] = cat_imputer.transform(X_test[cat_cols])

    # Save imputers for inference
    artifacts_dir = os.path.dirname(model_path)
    os.makedirs(artifacts_dir, exist_ok=True)
    imputers_path = os.path.join(artifacts_dir, "imputers.pkl")
    with open(imputers_path, "wb") as f:
        pickle.dump(
            {
                "num_imputer": num_imputer,
                "cat_imputer": cat_imputer,
                "num_cols": num_cols,
                "cat_cols": cat_cols,
            },
            f,
        )
    # ---------------------------------------------------------------------
```

Fig 26. Screenshot-1 of model training

```python
models = {
    "LinearRegression": LinearRegression(),
    "DecisionTree": DecisionTreeRegressor(random_state=42),
    "RandomForest": RandomForestRegressor(n_estimators=300, random_state=42, n_jobs=-1),
    "GradientBoosting": GradientBoostingRegressor(random_state=42),
}

best = {"name": None, "model": None, "metrics": {"mae": float("inf")}}

for name, est in models.items():
    with mlflow.start_run(run_name=name):
        est.fit(X_train, y_train)
        preds = est.predict(X_test)
        m = _metrics(y_test, preds)
        for k, v in m.items():
            mlflow.log_metric(k, float(v))
        mlflow.log_param("model_type", name)

        # Keep the best by MAE
        if m["mae"] < best["metrics"]["mae"]:
            best = {"name": name, "model": est, "metrics": m}

# Save best model
with open(model_path, "wb") as f:
    pickle.dump(best["model"], f)

# Save best meta
meta_path = os.path.join(artifacts_dir, "model_meta.json")
with open(meta_path, "w") as f:
    json.dump({"best_model": best["name"], "metrics": best["metrics"]}, f, indent=2)

# Log artifacts on a final run
with mlflow.start_run(run_name="best_model_summary"):
    mlflow.log_param("best_model", best["name"])
    for k, v in best["metrics"].items():
        mlflow.log_metric(f"best_{k}", float(v))
    mlflow.log_artifact(model_path)
    mlflow.log_artifact(meta_path)
    mlflow.log_artifact(cols_path)
    mlflow.log_artifact(imputers_path)

# Cache in Redis
r.set("salary:best_model", pickle.dumps(best["model"]))
logger.info(f"Saved best model ({best['name']}) to {model_path}")
return True
```

Fig 27. Screenshot-2 of model training

5) Model Deployment

In the model deployment stage, I deployed the trained salary prediction model using **FastAPI** as the serving framework. After training, the best model was copied into an API folder, renamed as salary_model.pkl, and linked with a JSON file that contained the feature order. FastAPI was then used to expose this model through several endpoints: /health to check if the model is active, /features to show expected input features, and /predict to make predictions based on user input. To prepare the input, I aligned it with the trained feature set before running inference. For the user interface, I used **FastAPI's built-in interactive Swagger UI**, which made it easy to test predictions directly in a browser without needing extra tools.

```python
app = FastAPI(title="Salary Model API", version="1.0.0")

# Artifacts written by my DAG
MODEL_PATH = os.getenv("MODEL_PATH", "/opt/airflow/dags/artifacts/api/salary_model.pkl")
COLS_PATH  = os.getenv("COLS_PATH",  "/opt/airflow/dags/artifacts/model_columns.json")

class Rows(BaseModel):
    rows: List[Dict[str, Any]] = Field(..., description="List of row dicts (use /features)")

_model = None
_features: List[str] = []

def _load_artifacts():
    global _model, _features
    if not os.path.isfile(MODEL_PATH):
        raise FileNotFoundError(f"Model not found: {MODEL_PATH}")
    if not os.path.isfile(COLS_PATH):
        raise FileNotFoundError(f"Columns file not found: {COLS_PATH}")
    with open(MODEL_PATH, "rb") as f:
        _model = pickle.load(f)
    with open(COLS_PATH) as f:
        _features = json.load(f)

@app.on_event("startup")
def _startup():
    _load_artifacts()

@app.get("/health")
def health():
    return {
        "status": "ok",
        "model_path": MODEL_PATH,
        "cols_path": COLS_PATH,
        "n_features": len(_features),
    }

@app.get("/features")
def features():
    return {"features": _features}

def _prep(rows: List[Dict[str, Any]]) -> pd.DataFrame:
    if _model is None:
        raise RuntimeError("Model not loaded")
    X = pd.DataFrame(rows)
    # add any missing expected columns
    for c in _features:
        if c not in X.columns:
            X[c] = 0
    # drop any unexpected columns and order correctly
    X = X[_features]
    return X

@app.post("/predict")
def predict(payload: Rows):
    try:
        X = _prep(payload.rows)
        preds = _model.predict(X)
        return {"predictions": [float(x) for x in preds]}
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))
```

Fig 28. Screenshot of app.py

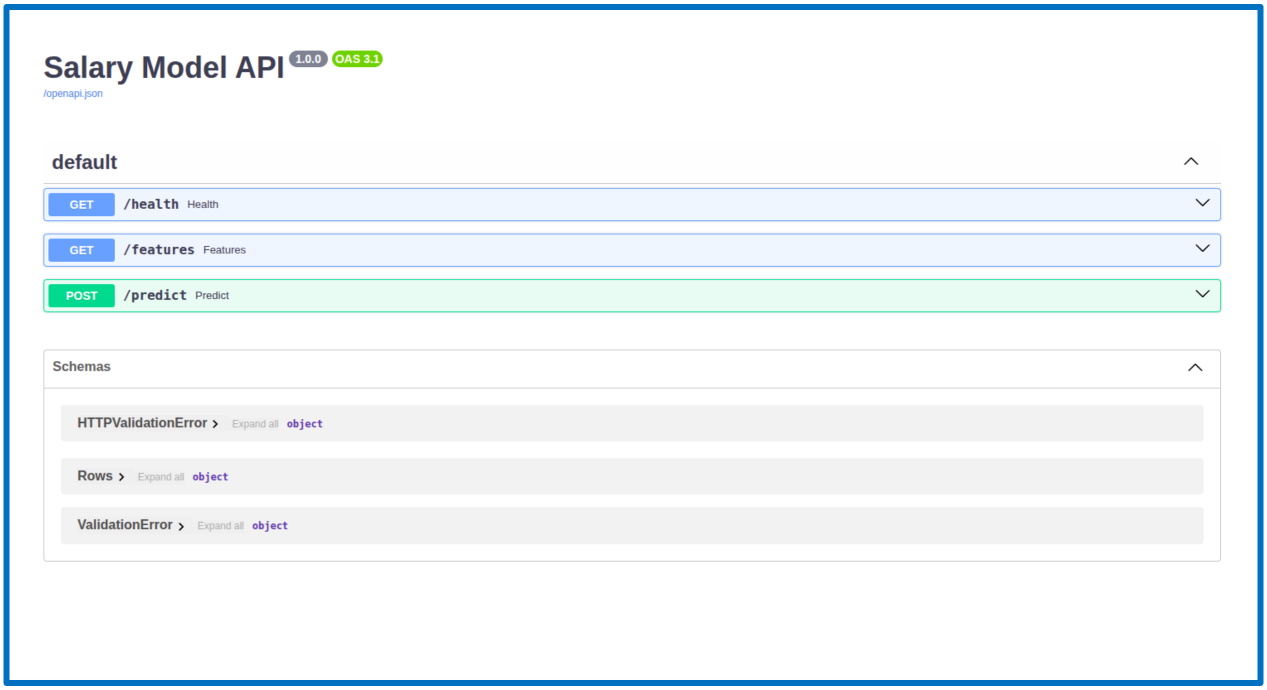


Fig 29. Interface of Swagger UI

VI. PIPELINE EVALUATION AND MONITORING

Pipeline Evaluation

After the Airflow pipeline was executed, MLflow showed that *Gradient Boosting* was selected as the best model. Its evaluation metrics were recorded as follows:

- Mean Absolute Error (MAE): **4647.33**
- Mean Squared Error (MSE): **100785998.02**
- Root Mean Squared Error (RMSE): **10039.22**
- R^2 Score: **0.94**

These results confirm that Gradient Boosting achieved the highest predictive accuracy among all tested models.

Model Monitoring

I monitor the model after deployment using Evidently to check whether new data or outcomes have shifted compared to training.

For **data drift**, I read a reference sample from gold_salary_features in MariaDB and a current batch from /opt/airflow/dags/data/new_salary_data.csv. I load the expected feature list from model_columns.json, keep only the overlapping columns, and run DataDriftPreset with a ColumnMapping focused on numerical features. The code writes a detailed HTML report (data_drift_report.html) and a summary CSV (data_drift_report.csv) into a unified folder resolved as REPORTS_DIR or defaulting to /opt/airflow/dags/artifacts/reports.

For **concept drift**, I compare the target distribution in training (SELECT SALARY AS target FROM gold_salary_features) with the current target proxy. If PREDICTED_SALARY exists in the current CSV, I use it; otherwise I fall back to real SALARY. I run TargetDriftPreset, saving concept_drift_report.html and a CSV with target_drift_detected and target_drift_score. Both checks guard the pipeline against silent performance degradation and keep the system reliable over time.

```python
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def monitor_data_drift(
    db_url: str | None = None,
    reference_table: str = "gold_salary_features",
    current_csv: str = "/opt/airflow/dags/data/new_salary_data.csv",
    cols_path: str = "/opt/airflow/dags/artifacts/model_columns.json",
    reports_dir: str | None = None,
):
    """
    Writes to one unified folder:
      reports_dir or $REPORTS_DIR or /opt/airflow/dags/artifacts/reports
    """
    db_url = db_url or "mysql+pymysql://app:app@mariadb:3306/analytics"
    engine = create_engine(db_url)

    # reference sample
    ref = pd.read_sql(f"SELECT * FROM {reference_table} LIMIT 1000", con=engine)

    # current batch
    cur_path = Path(current_csv)
    if not cur_path.exists():
        logger.warning("No current CSV found at %s", current_csv)
        return {"skipped": True, "reason": "no_current"}

    curr = pd.read_csv(cur_path)
    if len(curr) < 10:
        logger.warning("Not enough current rows for drift (have %d, need >=10)", len(curr))
        return {"skipped": True, "reason": "insufficient_current_rows"}

    # feature space
    with open(cols_path) as f:
        feature_cols = json.load(f)

    overlap = [c for c in feature_cols if c in curr.columns and c in ref.columns]
    if not overlap:
        logger.warning("No overlapping columns between reference features and current.")
        return {"skipped": True, "reason": "no_overlap"}

    refX = ref[overlap].copy()
    currX = curr[overlap].copy()

    cm = ColumnMapping(numerical_features=overlap, categorical_features=[])

    report = Report(metrics=[DataDriftPreset()])
    report.run(reference_data=refX, current_data=currX, column_mapping=cm)

    # unified folder
    reports_root = Path(reports_dir or os.getenv("REPORTS_DIR", "/opt/airflow/dags/artifacts/reports"))
    reports_root.mkdir(parents=True, exist_ok=True)

    report_html = reports_root / "data_drift_report.html"
    report_csv  = reports_root / "data_drift_report.csv"
    report.save_html(str(report_html))

    # summarize
    res = report.as_dict()
    m = res["metrics"][0]["result"]
    summary = {
        "dataset_drift": bool(m["dataset_drift"]),
        "share_of_drifted_columns": float(m["share_of_drifted_columns"]),
        "n_drifted": int(m["number_of_drifted_columns"]),
        "report_path": str(report_html),
    }
    pd.DataFrame([summary]).to_csv(str(report_csv), index=False)
    logger.info("Data drift summary: %s", summary)
    return summary
```

Fig 30. Screenshot of datadrift.py

```python
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def monitor_concept_drift(
    db_url: str | None = None,
    reference_table: str = "gold_salary_features",
    current_csv: str = "/opt/airflow/dags/data/new_salary_data.csv",
    reports_dir: str | None = None,
):
    """
    Writes to one unified folder:
      reports_dir or $REPORTS_DIR or /opt/airflow/dags/artifacts/reports
    """
    # Lazy import to avoid DAG import-time issues
    from evidently.report import Report
    from evidently.metric_preset import TargetDriftPreset

    engine = create_engine(db_url or "mysql+pymysql://app:app@mariadb:3306/analytics")

    # Reference target distribution from training
    ref = pd.read_sql(f"SELECT SALARY AS target FROM {reference_table} LIMIT 5000", con=engine)

    # Current data (predictions proxy OR real labels)
    p = Path(current_csv)
    if not p.exists():
        return {"skipped": True, "reason": "no_current"}

    curr_raw = pd.read_csv(p)

    # Prefer predictions if present, otherwise use real SALARY
    if "PREDICTED_SALARY" in curr_raw.columns:
        target_col = "PREDICTED_SALARY"
    elif "SALARY" in curr_raw.columns:
        target_col = "SALARY"
    else:
        return {"skipped": True, "reason": "no_target_in_current"}

    if len(curr_raw) < 10:
        return {"skipped": True, "reason": "insufficient_current_rows"}

    curr = curr_raw[[target_col]].rename(columns={target_col: "target"})

    report = Report(metrics=[TargetDriftPreset()])
    report.run(reference_data=ref, current_data=curr)

    # unified folder
    outdir = Path(reports_dir or os.getenv("REPORTS_DIR", "/opt/airflow/dags/artifacts/reports"))
    outdir.mkdir(parents=True, exist_ok=True)

    html_path = outdir / "concept_drift_report.html"
    report.save_html(str(html_path))

    out = report.as_dict()
    res = out["metrics"][0]["result"]
    drift = bool(res.get("drift_detected"))
    score = float(res.get("drift_score", 0.0))

    pd.DataFrame([{"target_drift_detected": drift, "target_drift_score": score}]).to_csv(
        outdir / "concept_drift_report.csv", index=False
    )
    logging.info(f"Concept drift: detected={drift}, score={score}, report={html_path}")
    return {"target_drift_detected": drift, "target_drift_score": score, "report_path": str(html_path)}
```

Fig 31. Screenshot of conceptdrift.py

Data Drift Analysis

In the data drift analysis, I compared the current dataset with the reference training data using Evidently AI. The results showed that the dataset overall remained stable, with drift detected in only one column out of seventeen, which is about 5.88%. The column that showed drift was **UNIT_Operations**, where the statistical test indicated a meaningful difference between the current and reference distributions. All other features, including age, salary-related attributes, and most categorical variables, remained consistent and did not show any significant shift. This means the data pipeline is generally reliable, but I need to keep monitoring changes in specific features like UNIT_Operations to ensure the model continues to perform well.

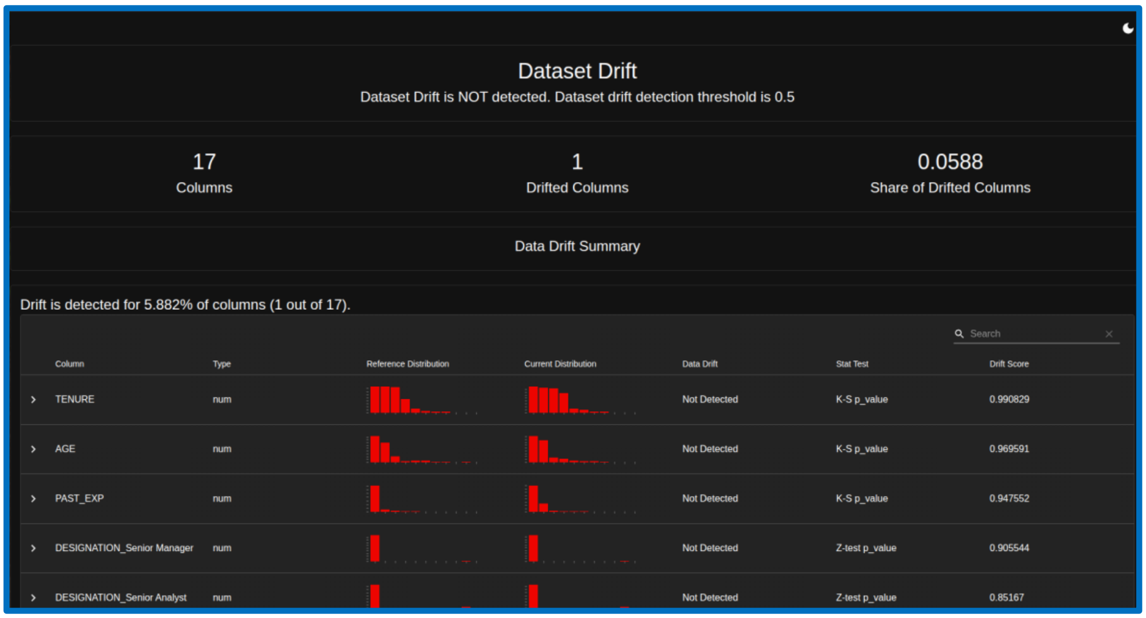


Fig 32. Data Drift HTML-1

Fig 33. Data Drift HTML-2

Fig 34. Data Drift HTML-3

Concept Drift Analysis

The concept drift analysis focused on comparing the salary distribution in the current data with the original training reference. The visualizations showed that while there were some fluctuations in the target values, most of them still fell within the expected range defined by the training set. The Wasserstein drift score was only 0.069, which is well below the threshold of 0.5. This confirms that no significant concept drift was detected. In simple terms, the salary distribution has remained stable over time, and the model's predictions can still be considered reliable without retraining at this point.

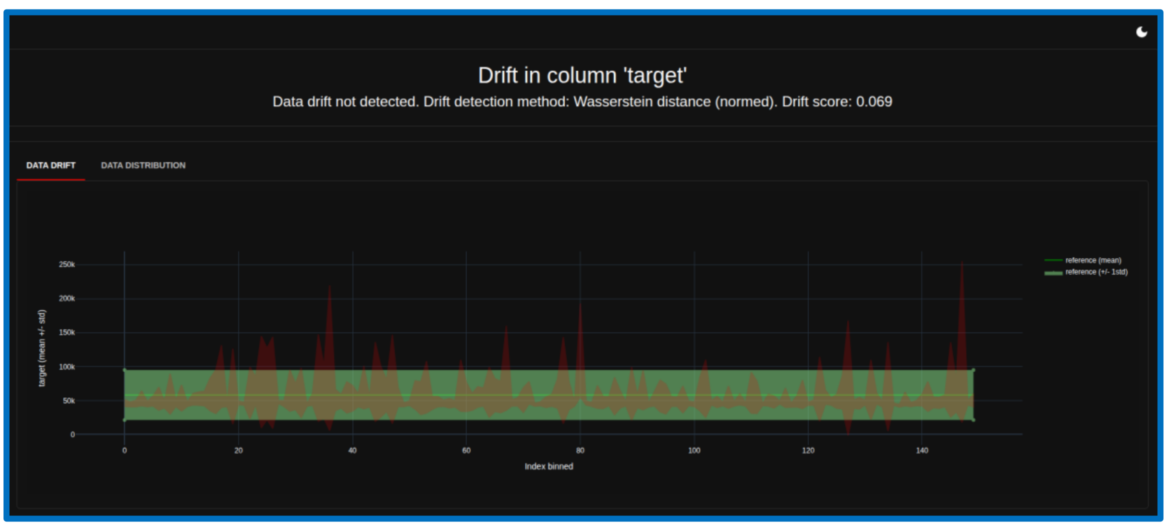


Fig 35. Concept Drift HTML-1

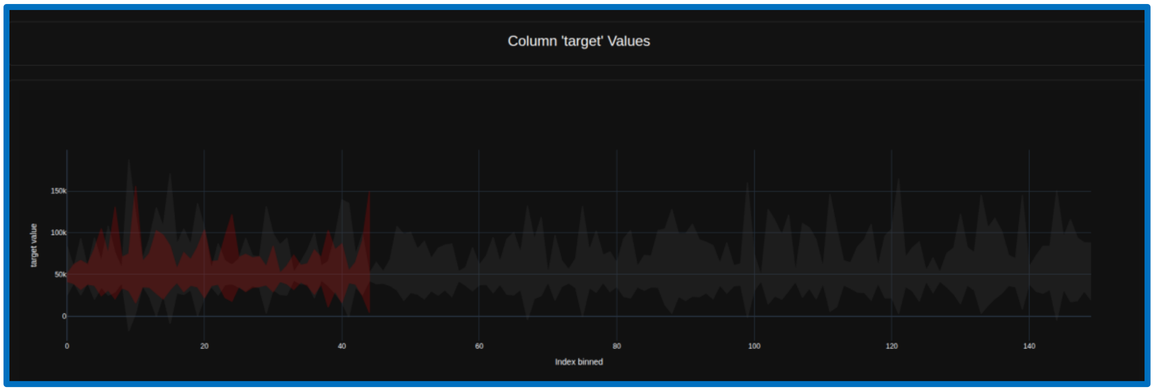


Fig 36. Concept Drift HTML-2

Fig 37. Concept Drift HTML-3

ML Flow

Once the Airflow pipeline finished running successfully, the training results were logged in MLflow under the experiment salary_model_training. I tested four models: Linear Regression, Decision Tree, Random Forest and Gradient Boosting. MLflow tracked metrics such as MAE, MSE, RMSE and R^2. Gradient Boosting gave the best results, with an MAE of about 4,647 and an R^2 of 0.94. The run marked best_model_summary confirmed it as the final model and stored it as an artifact. This shows how MLflow captured the results after Airflow, compared the models, and selected the most accurate one for deployment.

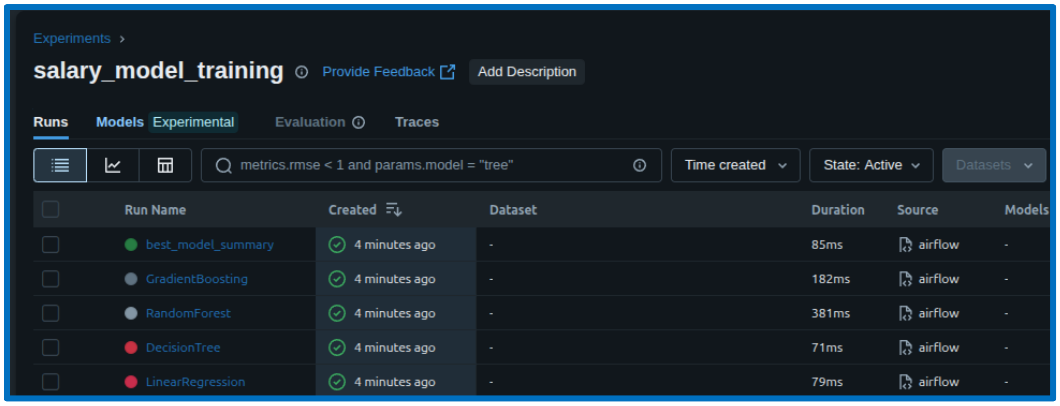


Fig 38. MLFlow-1

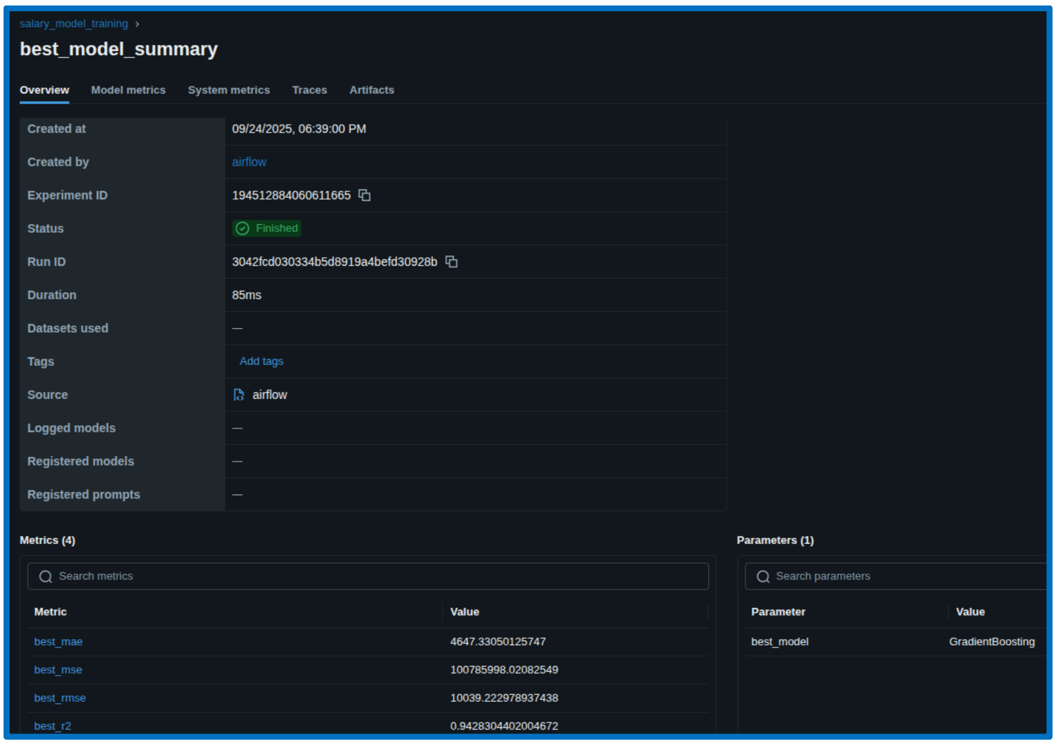


Fig 39. MLFlow-2

VII. DATA ANALYSIS AND INSIGHT GENERATION

I explored the processed salary dataset to understand trends and extract patterns that could guide predictions. Using descriptive statistics and visualization, I examined how salary distributions vary across roles, age groups, genders, and performance ratings. Boxplots and scatterplots helped reveal outliers and the relationship between age, ratings, and salary. Gender-based analysis showed disparities, while performance ratings indicated a clear link with salary levels. This analysis highlighted which features strongly influence pay, providing context for model development and improving interpretability of results. These insights ensure that the model's predictions are supported by clear, data-driven evidence.

VIII. CONCLUSION

This project used the Salary Prediction of Data Professions dataset with 2,639 rows and 13 columns, combining both categorical and numerical attributes like designation, gender, age, salary, ratings, and experience. The raw data was stored in MariaDB to preserve its structure, ensuring reliability for later stages. The pipeline undertook automated key steps involving preprocessing, model development, deployment, and monitoring. Models such as linear regression, random forest, and gradient boosting were experimented upon, and Gradient Boosting turned out to be the best one. Deployed via FastAPI, Evidently was utilized for monitoring data and concept drift and crafting an MLOps pipeline scalable, reproducible, and reliable from end-to-end.

IX. RECOMMENDATIONS AND FUTURE WORK

In the short term my focus is going to be on making the pipeline more accurate and a bit more automatic. One improvement I want to bring is hyperparameter tuning, where I can use tools like GridSearchCV to properly search and find the best model settings. This way I will not just rely on fixed parameters but test a range and see which works better. This short-term actions are practical and will directly increase both the robustness and usability of the system.

For long-term future work, I will focus on scaling the project to handle larger datasets and real-world deployment environments. I will turn the current FastAPI service into a full product-facing web application. The backend will keep FastAPI with a clean, versioned API and Swagger at the docs route. I will add authentication with JWT, request logging to MariaDB, rate limiting, and input validation using Pydantic models tied to the saved feature schema. Batch prediction will be supported through CSV upload, and each request will capture model version, latency, and prediction for audit. On the frontend I will build a simple web app that lets users enter features with friendly forms, see predictions instantly, and download results.

X. REFERENCES

Kablaoui, M. and Salman, K. (2024) ‘Salary prediction for data professions using machine learning techniques’, *Journal of Data Science and Analytics*, 12(3), pp. 145-158.

Raj, P., Kumar, A., Burman, R.K. and Kumari, L. (2025) ‘Forecasting Salary Using a Machine Learning System’, *Proceedings of the Recent Advances in Artificial Intelligence for Sustainable Development (RAISD 2025)*, pp. 131-146.

https://www.kaggle.com/datasets/abdelrhmantarek37/salary-prediction-of-data-professions

https://archive.ics.uci.edu/dataset/222/bank+marketing

https://www.kaggle.com/datasets/deepcontractor/car-price-prediction-challenge

https://youtu.be/CHVKmrwUIUA?si=20vbDlA7kL1fF3rX