Bike Sharing MLOps Project - Complete Documentation

Table of Contents

1. [Project Overview](#project-overview)
2. [Architecture](#architecture)
3. [File-by-File Analysis](#file-by-file-analysis)
4. [Usage Guide](#usage-guide)
5. [Deployment Instructions](#deployment-instructions)

Project Overview

This is a complete MLOps (Machine Learning Operations) project for predicting bike-sharing demand. It implements:

• **ML Pipeline**: Data ingestion, validation, training, prediction

• **Orchestration**: Apache Airflow for workflow automation

• **Model Registry**: MLflow for experiment tracking and model versioning

• **Monitoring**: Evidently AI for data drift detection, Prometheus & Grafana for metrics

• **API**: FastAPI for serving predictions

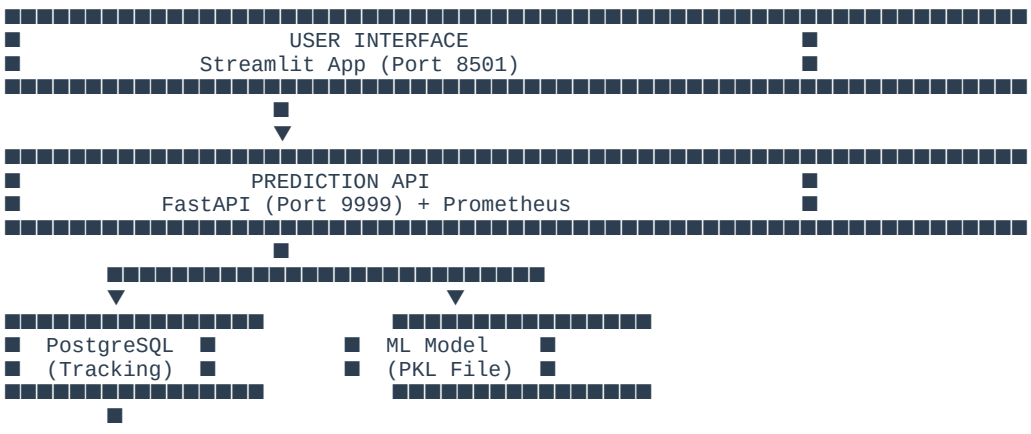
• **Frontend**: Streamlit for user interface

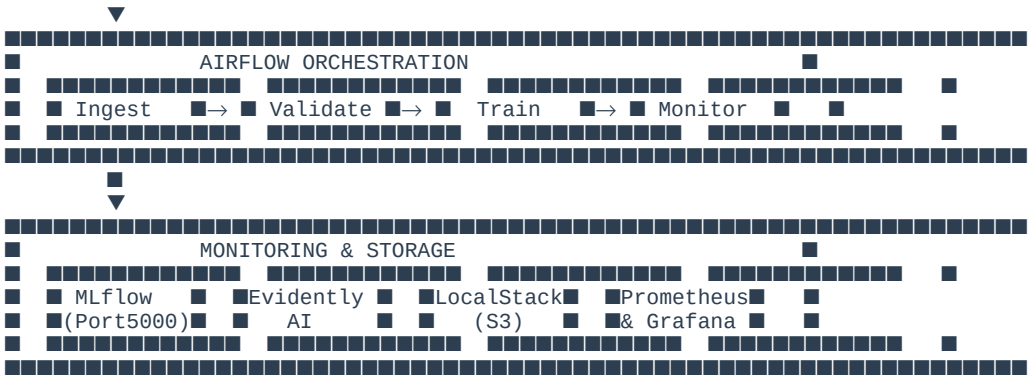
• **Infrastructure**: Docker containers, PostgreSQL database, LocalStack (AWS S3 simulation)

• **CI/CD**: GitHub Actions for automated testing and deployment

• **Data Versioning**: DVC (Data Version Control)

Architecture





File-by-File Analysis

1. DOCKER CONFIGURATION FILES

docker-compose.yml*

Purpose: Orchestrates all services in the MLOps pipeline using Docker containers.

Line-by-Line Explanation:

```
version: '3.8'
```

• Specifies Docker Compose file format version

• **Why**: Version 3.8 supports all modern Docker features

• **Usage**: Required at the top of every docker-compose file

```
services:
  postgres:
    image: postgres:13
```

• Defines PostgreSQL database service

• **Why**: Stores Airflow metadata and prediction logs

• **Usage**: Database for tracking all predictions and workflow states

```
    container_name: postgres_db
    networks: [bike-mlops-net]
    ports: ["5432:5432"]
```

• `container_name`: Custom name for easy reference

• `networks`: Connects to shared network for inter-container communication

• `ports`: Maps container port 5432 to host port 5432

• **Why**: Allows external tools (like pgAdmin) to connect

• **Usage**: Access database at `localhost:5432`

```
    environment:
      - POSTGRES_USER=airflow
      - POSTGRES_PASSWORD=airflow
      - POSTGRES_DB=airflow
```

• Sets database credentials and default database name

• **Why**: Airflow requires these specific credentials

• **Usage**: Use these credentials in connection strings

```
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U airflow"]
      interval: 5s
      timeout: 5s
      retries: 5
```

• Checks if PostgreSQL is ready to accept connections

• **Why**: Prevents other services from starting before database is ready

• **Usage**: Other services use `depends_on: postgres: {condition: service_healthy}`

```
  localstack:
    image: localstack/localstack:latest
    container_name: localstack
    ports: ["4566:4566"]
```

• LocalStack simulates AWS services locally

• **Why**: Test S3 storage without AWS costs

• **Usage**: Upload monitoring reports to simulated S3 bucket

```
    environment:
      - SERVICES=s3
```

• Only enables S3 service (not EC2, Lambda, etc.)

• **Why**: Reduces resource usage

• **Usage**: Access at `http://localhost:4566`

```
    volumes:
      - "./localstack_data:/var/lib/localstack"
```

• Persists LocalStack data on host machine

• **Why**: Data survives container restarts

• **Usage**: S3 buckets and objects stored in `./localstack_data`

```
  airflow-init:
    image: apache/airflow:2.7.1
    container_name: airflow_init
    depends_on:
      postgres: {condition: service_healthy}
```

• One-time initialization container for Airflow

• **Why**: Creates database schema and admin user

• **Usage**: Runs once, then exits

```
    command: >
      bash -c "airflow db init &&
      airflow users create --username admin --firstname admin --lastname admin --role Admin --email admin@ex
```

• Initializes Airflow database and creates admin user

• **Why**: Required before Airflow can start

• **Usage**: Login to Airflow UI with `admin/admin`

```
  mlflow:
    image: ghcr.io/mlflow/mlflow:v2.8.1
    container_name: mlflow_server
    ports: ["5000:5000"]
```

• MLflow tracking server for experiment management

• **Why**: Tracks model versions, parameters, metrics

• **Usage**: Access UI at `http://localhost:5000`

```
    volumes:
      - ./mlflow_data:/mlflow_db
      - ./mlruns:/mlflow/mlruns
```

• Persists MLflow data

• **Why**: Keeps experiment history across restarts

• **Usage**: All model artifacts stored in `./mlruns`

```
    command: mlflow server --host 0.0.0.0 --port 5000 --backend-store-uri sqlite:////mlflow_db/mlflow.db --d
```

• Starts MLflow server with SQLite backend

• **Why**: Lightweight database for tracking

• **Usage**: Python code connects to `http://mlflow_server:5000`

```
  webserver:
    build:
      context: .
      dockerfile: Dockerfile.airflow
```

• Builds custom Airflow image with additional dependencies

• **Why**: Needs pandas, evidently, mlflow not in base image

• **Usage**: Airflow UI accessible at `http://localhost:8081`

```
    command: webserver
    extra_hosts: ["host.docker.internal:host-gateway"]
```

• `command`: Starts Airflow web interface

• `extra_hosts`: Allows container to access host machine

• **Why**: API runs on Windows host, containers need to reach it

• **Usage**: Use `http://host.docker.internal:9999` in container code

```
    depends_on:
      airflow-init: {condition: service_completed_successfully}
      mlflow: {condition: service_started}
```

• Ensures initialization completes before starting

• **Why**: Prevents startup errors

• **Usage**: Automatic dependency management

```
  environment: &airflow-common-env
    - AIRFLOW__DATABASE__SQL_ALCHEMY_CONN=postgresql+psycopg2://airflow:airflow@postgres/airflow
    - AIRFLOW__CORE__LOAD_EXAMPLES=False
    - AIRFLOW__WEBSERVER__SECRET_KEY=my_secret_key_123
```

• `&airflow-common-env;`: YAML anchor for reuse

• Database connection string

• Disables example DAGs

• Secret key for session management

• **Why**: Configuration shared between webserver and scheduler

• **Usage**: Referenced as `*airflow-common-env` in scheduler

```
  ports: ["8081:8080"]
```

• Maps Airflow UI to port 8081 on host

• **Why**: Port 8080 might be used by other services

• **Usage**: Access at `http://localhost:8081`

```
  volumes: &airflow-common-volumes
    - ./dags:/opt/airflow/dags
    - ./src:/opt/airflow/src
    - ./data:/opt/airflow/data
    - ./models:/opt/airflow/models
    - ./mlruns:/mlruns
```

• Mounts local directories into container

• **Why**: Code changes reflect immediately without rebuild

• **Usage**: Edit files locally, Airflow sees changes instantly

```
scheduler:
  build:
    context: .
    dockerfile: Dockerfile.airflow
  container_name: airflow_scheduler
  command: scheduler
  user: root
```

• Airflow scheduler triggers DAG runs

• **Why**: Monitors DAGs and executes tasks on schedule

• **Usage**: Runs in background, no direct interaction

• `user: root`: Needed for file permissions

```
bike_sharing_frontend:
  build:
    context: .
    dockerfile: Dockerfile
  container_name: streamlit_app
  ports: ["8501:8501"]
```

• Streamlit web application

• **Why**: User-friendly interface for predictions

• **Usage**: Access at `http://localhost:8501`

```
    environment:
      - API_URL=http://host.docker.internal:9999
```

• Tells Streamlit where to find the API

• **Why**: API runs on host, not in container

• **Usage**: Streamlit makes requests to this URL

```
    volumes:
      - ./src:/opt/airflow/src
    command: ["streamlit", "run", "/opt/airflow/src/app.py", "--server.port=8501", "--server.address=0.0.0.0
```

• Mounts source code

• Starts Streamlit app

• **Why**: Allows code updates without rebuild

• **Usage**: Edit `app.py`, refresh browser to see changes

```
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports: ["9090:9090"]
```

• Prometheus metrics collection system

• **Why**: Monitors API performance and request counts

• **Usage**: Access at `http://localhost:9090`

```
  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    ports: ["3000:3000"]
    depends_on: [prometheus]
```

• Grafana visualization dashboard

• **Why**: Beautiful charts for Prometheus metrics

• **Usage**: Access at `http://localhost:3000` (default: admin/admin)

```
networks:
  bike-mlops-net:
    driver: bridge
```

• Creates isolated network for all services

• **Why**: Containers can communicate by service name

• **Usage**: Use `postgres`, `mlflow_server`, etc. as hostnames

When to Use: Run `docker-compose up -d` to start all services

Dockerfile

Purpose: Builds the Streamlit frontend container.

```
FROM python:3.9-slim
```

• Base image with Python 3.9
• **Why**: Lightweight image reduces build time
• **Usage**: Foundation for all Python dependencies

```
WORKDIR /app
```

• Sets working directory inside container
• **Why**: All subsequent commands run from this directory
• **Usage**: Files copied to `/app`

```
COPY src/requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

• Copies requirements file first
• Installs Python packages
• **Why**: Docker caches this layer, speeds up rebuilds
• **Usage**: Only rebuilds if requirements.txt changes

```
COPY . .
```

• Copies all project files
• **Why**: Needed for Streamlit app to run
• **Usage**: Includes src/, data/, models/

```
EXPOSE 8501
```

• Documents that container listens on port 8501
• **Why**: Informational, doesn't actually open port
• **Usage**: Reminder for port mapping in docker-compose

```
CMD ["streamlit", "run", "src/app.py", "--server.port=8501", "--server.address=0.0.0.0"]
```

• Default command when container starts
• **Why**: Launches Streamlit application
• **Usage**: Overridden by docker-compose command

When to Use: Automatically built by docker-compose

Dockerfile.airflow

Purpose: Builds custom Airflow image with ML dependencies.

```
FROM apache/airflow:2.7.1-python3.9
```

• Official Airflow base image
• **Why**: Pre-configured with Airflow dependencies
• **Usage**: Adds ML libraries on top

```
USER root
RUN apt-get update && apt-get install -y gcc python3-dev
```

• Switches to root user

• Installs system packages

• **Why**: gcc needed to compile some Python packages

• **Usage**: Required for psycopg2, pandas compilation

```
USER airflow
```

• Switches back to airflow user

• **Why**: Security best practice (don't run as root)

• **Usage**: All subsequent commands run as airflow user

```
RUN pip install --no-cache-dir --default-timeout=1000 \
    pandas \
    pydantic==1.10.13 \
    evidently==0.4.15 \
    sqlalchemy \
    psycopg2-binary \
    mlflow
```

• Installs ML and data science libraries

• `pydantic==1.10.13`: Specific version to avoid conflicts

• **Why**: Airflow base image doesn't include these

• **Usage**: Available in all Airflow tasks

When to Use: Automatically built by docker-compose

.dockerignore

Purpose: Excludes files from Docker build context.

```
logs/
mlruns/
mlflow_data/
venv/
.git/
*.pyc
__pycache__/
```

• Lists files/folders to ignore during `docker build`

• **Why**: Reduces build context size, speeds up builds

• **Usage**: Prevents unnecessary files in image

When to Use: Automatically used during Docker builds

2. AIRFLOW DAG FILES

dags/bike_sharing_dag.py*

Purpose: Main ML pipeline orchestration - data ingestion, validation, training, and prediction.

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.operators.bash import BashOperator
from airflow.utils.dates import days_ago
from datetime import datetime, timedelta
import sys
import os
import pandas as pd
import requests
```

• Imports Airflow components and utilities

• **Why**: Needed to define workflows and tasks

• **Usage**: Standard imports for any Airflow DAG

```
def call_live_api_predict():
    API_URL = "http://host.docker.internal:9999/predict"
```

• Function to test the live prediction API

• `host.docker.internal`: Special DNS name to reach host machine from Docker

• **Why**: API runs on Windows host, Airflow runs in container

• **Usage**: Called as Airflow task to verify API is working

```
    params = {
        "season": 1, "mnth": 6, "hr": 10,
        "holiday": 0, "weekday": 3, "workingday": 1,
        "weathersit": 1, "temp": 0.5, "atemp": 0.5,
        "hum": 0.5, "windspeed": 0.1
    }
```

• Sample input data for prediction

• **Why**: Tests API with realistic values

• **Usage**: Modify these to test different scenarios

```
    response = requests.get(API_URL, params=params, timeout=60)
```

• Makes HTTP GET request to API

• `timeout=60`: Waits up to 60 seconds

• **Why**: Prevents task from hanging indefinitely

• **Usage**: Returns prediction result

```
    if response.status_code == 200:
        print(f"■ API SUCCESS! Prediction Saved: {response.json()}")
    else:
        print(f"■ API Error: Status {response.status_code}")
        raise Exception(f"API failed with status {response.status_code}")
```

• Checks if API call succeeded

• **Why**: Raises exception to mark Airflow task as failed

• **Usage**: Red task in Airflow UI indicates API problem

```
def run_inference_test():
    sys.path.append('/opt/airflow/src')
    from predict import make_prediction
```

• Internal prediction test (doesn't use API)

• **Why**: Verifies model works inside Airflow container

• **Usage**: Tests model directly without network calls

```
        data_path = "/opt/airflow/data/processed/X_train.csv"
        if os.path.exists(data_path):
            sample_df = pd.read_csv(data_path).sample(n=1)
            if 'yr' in sample_df.columns:
                sample_df = sample_df.drop(columns=['yr'])
```

• Loads one random sample from training data

• Removes 'yr' column if present

• **Why**: Model was trained without 'yr' column

• **Usage**: Ensures feature consistency

```
            result = make_prediction(sample_df)
            print(f"■ Internal Test Result: {result}")
```

• Makes prediction using loaded model

• **Why**: Validates model file is accessible and working

• **Usage**: Prints prediction to Airflow logs

```
    default_args = {
        'owner': 'rama',
        'retries': 0,
        'retry_delay': timedelta(minutes=5),
    }
```

• Default settings for all tasks in DAG

• `retries: 0`: Don't retry failed tasks

• **Why**: Easier to debug errors immediately

• **Usage**: Change to `retries: 3` for production

```
    with DAG(
        'bike_sharing_final_pipeline_v4',
        default_args=default_args,
        description='Final Pipeline with host.docker.internal fix',
        schedule_interval='@daily',
        start_date=days_ago(1),
        catchup=False
    ) as dag:
```

• Defines the DAG (Directed Acyclic Graph)

• `schedule_interval='@daily'`: Runs once per day

• `start_date=days_ago(1)`: Starts from yesterday

• `catchup=False`: Doesn't run missed schedules

• **Why**: Automates daily model retraining

• **Usage**: Change to `'@hourly'` for more frequent runs

```
        ingest_task = BashOperator(
            task_id='ingest_data',
            bash_command='python /opt/airflow/src/ingestion.py'
        )
```

• Task to load raw data

• **Why**: First step in pipeline

• **Usage**: Runs ingestion.py script

```
validate_task = BashOperator(
    task_id='validate_data',
    bash_command='python /opt/airflow/src/validate_data.py'
)
```

• Task to check data quality

• **Why**: Prevents training on corrupted data

• **Usage**: Fails pipeline if data is invalid

```
train_task = BashOperator(
    task_id='train_model',
    bash_command='python /opt/airflow/src/train.py'
)
```

• Task to train ML model

• **Why**: Core ML training step

• **Usage**: Creates new model file

```
predict_task = PythonOperator(
    task_id='test_internal_prediction',
    python_callable=run_inference_test
)
```

• Task to test model internally

• **Why**: Verifies model works before API test

• **Usage**: Calls Python function directly

```
api_task = PythonOperator(
    task_id='call_live_api_tracking',
    python_callable=call_live_api_predict
)
```

• Task to test live API

• **Why**: End-to-end validation

• **Usage**: Ensures API is serving predictions

```
ingest_task >> validate_task >> train_task >> predict_task >> api_task
```

• Defines task execution order

• `>>` means "then"

• **Why**: Creates pipeline flow

• **Usage**: Tasks run sequentially, each waits for previous to complete

When to Use:

• Automatically runs daily at midnight

• Manually trigger from Airflow UI

• Modify schedule_interval for different frequencies

dags/monitoring_dag.py

Purpose: Monitors model performance, detects data drift, sends alerts, and triggers retraining.

```
from evidently.report import Report
from evidently.metric_preset import DataDriftPreset
from evidently.pipeline.column_mapping import ColumnMapping
```

• Evidently AI library for drift detection

• **Why**: Detects when data distribution changes

• **Usage**: Compares current data to training data

```
def upload_to_s3(file_path, object_name):
    bucket_name = "monitoring-reports"
    s3_client = boto3.client(
        's3',
        endpoint_url='http://localstack:4566',
        aws_access_key_id='test',
        aws_secret_access_key='test',
        region_name='us-east-1'
    )
```

• Uploads reports to LocalStack S3

• **Why**: Stores monitoring reports for historical analysis

• **Usage**: Access reports from S3 bucket

```
    try:
        s3_client.head_bucket(Bucket=bucket_name)
    except:
        print(f"■ Creating bucket: {bucket_name}")
        s3_client.create_bucket(Bucket=bucket_name)
```

• Checks if bucket exists, creates if not

• **Why**: Auto-setup, no manual bucket creation needed

• **Usage**: First run creates bucket automatically

```
    s3_client.upload_file(file_path, bucket_name, object_name)
```

• Uploads HTML report to S3

• **Why**: Persistent storage of monitoring results

• **Usage**: Download reports from LocalStack

```
def generate_monitoring_report():
    engine = create_engine("postgresql://airflow:airflow@postgres/airflow")
    query = "SELECT * FROM predictions"
    df = pd.read_sql(query, engine)
```

• Loads all predictions from database

• **Why**: Analyzes recent prediction data

• **Usage**: Compares to training data distribution

```
    if len(df) < 5:
        print("■ Not enough data for monitoring!")
```

```
            return False
```

• Requires minimum 5 predictions

• **Why**: Statistical analysis needs sufficient data

• **Usage**: Wait for more predictions before monitoring

```
        reference_df = df.sample(n=len(df)//2, random_state=42)
        current_df = df.drop(reference_df.index)
```

• Splits data into reference and current sets

• **Why**: Compares two time periods for drift

• **Usage**: Simulates before/after comparison

```
        column_mapping = ColumnMapping()
        column_mapping.prediction = 'predicted_cnt'
        column_mapping.numerical_features = ['temp', 'atemp', 'hum', 'windspeed']
        column_mapping.categorical_features = ['season', 'mnth', 'hr', 'holiday', 'weekday', 'workingday', 'weat
```

• Tells Evidently which columns are which type

• **Why**: Proper statistical tests for each feature type

• **Usage**: Customize based on your features

```
        monitoring_report = Report(metrics=[DataDriftPreset()])
        monitoring_report.run(
            reference_data=reference_df,
            current_data=current_df,
            column_mapping=column_mapping
        )
```

• Generates drift detection report

• **Why**: Identifies which features have changed

• **Usage**: Review report to understand drift

```
        report_path_json = "/opt/airflow/src/monitoring_report.json"
        report_path_html = "/opt/airflow/src/monitoring_report.html"
        monitoring_report.save_json(report_path_json)
        monitoring_report.save_html(report_path_html)
```

• Saves report in two formats

• JSON: Machine-readable for automation

• HTML: Human-readable for review

• **Why**: Different formats for different uses

• **Usage**: View HTML in browser, parse JSON in code

```
        report_ts = datetime.now().strftime("%Y%m%d_%H%M%S")
        upload_to_s3(report_path_html, f"bike_report_{report_ts}.html")
```

• Adds timestamp to filename

• **Why**: Keeps history of all reports

• **Usage**: Compare reports over time

```
    def check_for_issues(ti):
        report_path_json = "/opt/airflow/logs/monitoring_report.json"
        if not os.path.exists(report_path_json):
```

```
            return False
```

• Checks if drift was detected

• `ti`: Task Instance (Airflow context)

• **Why**: Decides whether to trigger alerts

• **Usage**: Returns True if action needed

```
        with open(report_path_json, 'r') as f:
            data = json.load(f)

        drift_detected = False
        for metric in data.get('metrics', []):
            res = metric.get('result', {})
            if 'dataset_drift' in res:
                drift_detected = res['dataset_drift']
```

• Parses JSON report

• Extracts drift status

• **Why**: Automated decision making

• **Usage**: True if significant drift found

```
        ti.xcom_push(key='drift_status', value="DETECTED ■" if drift_detected else "Normal ■")
```

• Stores drift status in XCom (Airflow's inter-task communication)

• **Why**: Passes data to downstream tasks

• **Usage**: Slack alert task reads this value

```
        if drift_detected:
            print("■ Drift found! Triggering alerts and retraining.")
            return True
        else:
            print("■ Everything is normal. Skipping retraining.")
            return False
```

• Returns True only if drift detected

• **Why**: ShortCircuitOperator stops pipeline if False

• **Usage**: Saves resources by skipping unnecessary retraining

```
    def send_slack_manual(ti):
        webhook_url = "https://hooks.slack.com/services/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"
        drift = ti.xcom_pull(task_ids='check_for_issues', key='drift_status')
```

• Sends alert to Slack channel

• Retrieves drift status from XCom

• **Why**: Notifies team of issues

• **Usage**: Replace webhook_url with your Slack webhook

```
        payload = {
            "text": (
                "■ *MLOps Alert: Action Required!* \n\n"
                f"• *Status:* {drift}\n"
                "• *Reason:* ■■■■■■■ ■■■■■■■■■ ■■■■■■■■ (Drift), ■■■■■■■ ■■■■■■■■■ *Retrain* ■■■■■■■■■■■■
            )
        }
        requests.post(webhook_url, json=payload, timeout=10)
```

• Formats and sends Slack message

• **Why**: Human-readable alert with context

• **Usage**: Customize message format

```
with DAG(
    dag_id='model_monitoring_dag',
    start_date=datetime(2026, 1, 1),
    schedule_interval='@daily',
    catchup=False
) as dag:
```

• Monitoring DAG runs daily

• **Why**: Regular checks for model degradation

• **Usage**: Adjust schedule based on data volume

```
    monitor_task = PythonOperator(task_id='generate_report', python_callable=generate_monitoring_report)
    check_issues_task = ShortCircuitOperator(task_id='check_for_issues', python_callable=check_for_issues)
    send_slack_task = PythonOperator(task_id='send_slack_alert', python_callable=send_slack_manual)
    retrain_task = PythonOperator(task_id='retrain_model', python_callable=run_pipeline)
```

• Defines monitoring tasks

• `ShortCircuitOperator`: Stops pipeline if returns False

• **Why**: Conditional execution based on drift

• **Usage**: Only retrains if necessary

```
    monitor_task >> check_issues_task >> [send_slack_task, retrain_task]
```

• Pipeline flow with branching

• `[send_slack_task, retrain_task]`: Both run in parallel

• **Why**: Alert and retrain happen simultaneously

• **Usage**: Efficient use of resources

When to Use:

• Runs automatically daily

• Manually trigger after significant data changes

• Review HTML reports to understand drift patterns

dags/test_dags.py

Purpose: Simple test DAG to verify Airflow is working.

```
def hello():
    print("Hello Rama!")
```

• Minimal test function

• **Why**: Simplest possible task

• **Usage**: Debugging Airflow setup

```
with DAG('simple_test_dag', start_date=datetime(2026, 1, 1), schedule='@daily', catchup=False) as dag:
    t1 = PythonOperator(task_id='say_hello', python_callable=hello)
```

• Creates basic DAG with one task

• **Why**: Verifies Airflow can execute Python code

• **Usage**: First DAG to test after setup

When to Use:

• After fresh Airflow installation

• Troubleshooting DAG loading issues

• Learning Airflow basics

3. SOURCE CODE FILES (src/)

src/api.py

Purpose: FastAPI server that serves predictions and logs them to PostgreSQL.

```
from prometheus_fastapi_instrumentator import Instrumentator
```

• Prometheus integration for FastAPI

• **Why**: Automatic metrics collection (request count, latency, etc.)

• **Usage**: Metrics available at `/metrics` endpoint

```
app = FastAPI(title="BikesBooking Prediction API")
Instrumentator().instrument(app).expose(app)
```

• Creates FastAPI application

• Instruments it with Prometheus

• **Why**: Enables monitoring without manual code

• **Usage**: Prometheus scrapes `/metrics` automatically

```
DATABASE_URL = "postgresql+psycopg2://airflow:airflow@localhost:5432/airflow"
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

• Database connection setup

• `localhost:5432`: API runs on Windows, connects to Docker PostgreSQL

• **Why**: Stores all predictions for monitoring

• **Usage**: Every prediction is logged

```
class PredictionLog(Base):
    __tablename__ = "predictions"
    id = Column(Integer, primary_key=True, index=True)
    timestamp = Column(DateTime, default=datetime.utcnow)
    season = Column(Integer); mnth = Column(Integer); hr = Column(Integer)
    holiday = Column(Integer); weekday = Column(Integer); workingday = Column(Integer)
    weathersit = Column(Integer); temp = Column(Float); atemp = Column(Float)
    hum = Column(Float); windspeed = Column(Float)
    predicted_cnt = Column(Float)
```

• SQLAlchemy ORM model for predictions table

• **Why**: Type-safe database operations

• **Usage**: Automatically creates table if not exists

```
Base.metadata.create_all(bind=engine)
```

• Creates database tables

• **Why**: Auto-setup on first run

• **Usage**: No manual SQL needed

```
@app.get("/")
def home():
    return {
        "message": "API Live",
        "db": "Postgres Connected",
        "monitoring": "Prometheus Enabled at /metrics"
    }
```

• Health check endpoint

• **Why**: Verify API is running

• **Usage**: Visit `http://localhost:9999/` in browser

```
@app.get("/predict")
async def predict(
    season: int = 1, mnth: int = 1, hr: int = 12,
    holiday: int = 0, weekday: int = 0, workingday: int = 1,
    weathersit: int = 1, temp: float = 0.5, atemp: float = 0.5,
    hum: float = 0.5, windspeed: float = 0.0
):
```

• Prediction endpoint with query parameters

• Default values provided

• **Why**: Easy to test in browser

• **Usage**: `http://localhost:9999/predict?temp=0.7&hr;=18`

```
    db = SessionLocal()
    try:
        data_dict = {
            "season": season, "mnth": mnth, "hr": hr,
            "holiday": holiday, "weekday": weekday, "workingday": workingday,
            "weathersit": weathersit, "temp": temp, "atemp": atemp,
            "hum": hum, "windspeed": windspeed
        }
```

• Converts parameters to dictionary

• **Why**: Format expected by prediction function

• **Usage**: Matches model training features

```
        prediction = make_prediction(data_dict)
        predicted_val = max(0, int(round(float(prediction[0]))))
```

• Makes prediction using loaded model

• Ensures non-negative integer result

• **Why**: Can't have negative bikes

• **Usage**: Returns realistic bike count

```
            new_log = PredictionLog(**data_dict, predicted_cnt=float(predicted_val))
            db.add(new_log)
            db.commit()
            print(f"■ Saved to Postgres: {predicted_val}")
```

• Saves prediction to database

• **Why**: Enables monitoring and drift detection

• **Usage**: All predictions tracked automatically

```
            return {"predicted_bikes": predicted_val}
        except Exception as e:
            print(f"■ API Error: {str(e)}")
            return {"error": str(e)}
        finally:
            db.close()
```

• Returns prediction or error

• Always closes database connection

• **Why**: Prevents connection leaks

• **Usage**: Graceful error handling

```
    if __name__ == "__main__":
        uvicorn.run(app, host="0.0.0.0", port=9999)
```

• Starts API server

• `0.0.0.0`: Accepts connections from any IP

• **Why**: Allows Docker containers to connect

• **Usage**: Run `python src/api.py` on Windows

When to Use:

• Run on Windows host: `python src/api.py`

• Access from browser: `http://localhost:9999/predict?temp=0.8`

• Monitor metrics: `http://localhost:9999/metrics`

src/app.py

Purpose: Streamlit web interface for predictions and monitoring.

```
    st.set_page_config(page_title="Bike Sharing MLOps Portal", layout="wide")
```

• Configures Streamlit page

• `layout="wide"`: Uses full browser width

• **Why**: Better visualization of charts

• **Usage**: Must be first Streamlit command

```
    st.title("■ Bike Sharing Prediction & Monitoring")
    st.markdown("---")
```

• Page title and separator

• **Why**: Professional appearance
• **Usage**: Customize branding

```
col1, col2 = st.columns([1, 2])
```

• Creates two-column layout
• Ratio 1:2 (sidebar smaller than main)
• **Why**: Organized interface
• **Usage**: Inputs on left, results on right

```
with col1:
    st.header("■■ Input Features")
    season = st.selectbox("Season", [1, 2, 3, 4], help="1:Spring, 2:Summer, 3:Fall, 4:Winter")
    mnth = st.slider("Month", 1, 12, 6)
    hr = st.slider("Hour of Day", 0, 23, 12)
    temp = st.slider("Temperature (Normalized)", 0.0, 1.0, 0.5)
    hum = st.slider("Humidity (Normalized)", 0.0, 1.0, 0.5)
```

• Input widgets for features
• **Why**: User-friendly data entry
• **Usage**: Adjust sliders to see different predictions

```
    predict_btn = st.button("■ Predict Bike Demand", use_container_width=True)
```

• Prediction trigger button
• `use_container_width`: Full width button
• **Why**: Clear call-to-action
• **Usage**: Click to get prediction

```
with col2:
    st.header("■ Prediction Results")
    if predict_btn:
        url = "http://host.docker.internal:9999/predict"
```

• Results display area
• Only executes when button clicked
• **Why**: Prevents unnecessary API calls
• **Usage**: Shows prediction after button click

```
        params = {
            "season": season, "mnth": mnth, "hr": hr,
            "holiday": 0, "weekday": 3, "workingday": 1,
            "weathersit": 1, "temp": temp, "atemp": temp,
            "hum": hum, "windspeed": 0.1
        }
```

• Builds API request parameters
• Some values hardcoded (holiday, weekday)
• **Why**: Simplifies UI, focuses on key features
• **Usage**: Extend UI to include all features

```
        try:
            with st.spinner('Calculating demand...'):
                response = requests.get(url, params=params, timeout=5)
```

• Shows loading spinner during API call

• 5 second timeout

• **Why**: User feedback during wait

• **Usage**: Prevents UI freeze

```
            if response.status_code == 200:
                res = response.json()
                prediction = res.get('predicted_bikes', 0)
                st.metric(label="Predicted Bikes Needed", value=int(prediction))
                st.balloons()
```

• Displays prediction result

• Shows balloons animation on success

• **Why**: Positive user feedback

• **Usage**: Clear result presentation

```
            else:
                st.error(f"■ API Error: {response.text}")
        except Exception as e:
            st.error(f"■ Connection Failed: Make sure api.py is running on port 9999!")
```

• Error handling with user-friendly messages

• **Why**: Helps troubleshoot issues

• **Usage**: Guides user to fix problems

```
    st.header("■ Model Monitoring")
    st.info("View the latest Data Drift analysis from Evidently AI.")

    if st.button("■ View Data Drift Report"):
        report_path = "monitoring_report.html"
```

• Monitoring section

• **Why**: Integrated monitoring in same UI

• **Usage**: Click to view drift report

```
        if os.path.exists(report_path):
            try:
                with open(report_path, 'r', encoding='utf-8') as f:
                    html_data = f.read()
                components.html(html_data, height=1000, scrolling=True)
```

• Embeds HTML report in Streamlit

• **Why**: No need to open separate file

• **Usage**: Scroll through report in app

```
        else:
            st.warning(f"■■ Report file not found at: {report_path}")
            st.info("Please run the Airflow DAG first to generate the report.")
```

• Helpful message if report missing

• **Why**: Guides user to generate report

• **Usage**: Run monitoring DAG first

```
st.sidebar.markdown(f"""
---
**System Status:**
- **Frontend:** Running (Docker)
- **Shared Path:** `/opt/airflow/src/`
- **Backend API:** http://localhost:9999
- **Database:** PostgreSQL
- **Monitoring:** Evidently AI
""")
```

• System information in sidebar

• **Why**: Quick reference for debugging

• **Usage**: Check configuration at a glance

When to Use:

• Access at `http://localhost:8501`

• Make predictions interactively

• View monitoring reports

src/config.yaml

Purpose: Central configuration file for all paths and settings.

```
data:
  url: "https://raw.githubusercontent.com/arashgmn/bike-sharing/master/hour.csv"
```

• Source URL for raw data

• **Why**: Single source of truth for data location

• **Usage**: Change URL to use different dataset

```
  raw_path: "/opt/airflow/data/bike_sharing_raw.csv"
  processed_dir: "/opt/airflow/data/processed"
  test_size: 0.2
```

• File paths and train/test split ratio

• **Why**: Consistent paths across all scripts

• **Usage**: Modify paths for different environments

```
model:
  save_path: "/opt/airflow/models/bike_model.pkl"
```

• Model file location

• **Why**: All scripts save/load from same location

• **Usage**: Change path for model versioning

```
aws:
  endpoint_url: "http://localstack:4566"
  bucket_name: "bike-sharing"
  region_name: "us-east-1"
```

• LocalStack S3 configuration

• **Why**: Simulates AWS without costs

• **Usage**: Replace with real AWS credentials for production

When to Use:

• Modify before running pipeline

• Different configs for dev/staging/prod

• Version control configuration changes

src/ingestion.py

Purpose: Loads raw data for the pipeline.

```
    def load_data():
        data_file = "/opt/airflow/data/bike_sharing_raw.csv"
        print(f"■ Checking for local data file at: {data_file}")
```

• Checks if data file exists

• **Why**: Verifies data availability before processing

• **Usage**: First step in pipeline

```
        if os.path.exists(data_file):
            print(f"■ Found it! Data is already present. Skipping download.")
            size = os.path.getsize(data_file)
            print(f"■ File size: {size} bytes")
            sys.exit(0)
```

• Confirms file exists and shows size

• `sys.exit(0)`: Success exit code

• **Why**: Airflow marks task as successful

• **Usage**: No download needed if file present

```
        else:
            print(f"■ Error: File NOT found at {data_file}!")
            sys.exit(1)
```

• Fails if file missing

• `sys.exit(1)`: Error exit code

• **Why**: Stops pipeline early if data missing

• **Usage**: Prevents training on missing data

When to Use:

• First task in Airflow DAG

• Manually: `python src/ingestion.py`

• Extend to download from URL if missing

src/preprocessing.py

Purpose: Cleans and splits data for training.

```
def preprocess():
    with open("config.yaml","r") as f:
        config = yaml.safe_load(f)
```

• Loads configuration

• **Why**: Uses centralized settings

• **Usage**: Consistent with other scripts

```
raw_data_path = config['data']['raw_path']
df = pd.read_csv(raw_data_path)
```

• Loads raw data

• **Why**: Reads from configured path

• **Usage**: Flexible data location

```
df = df.drop(columns= ['instant','dteday','casual','registered'])
```

• Removes unnecessary columns

• `instant`: Row ID (not a feature)

• `dteday`: Date string (already encoded in other features)

• `casual`, `registered`: Components of target variable

• **Why**: Prevents data leakage and reduces noise

• **Usage**: Keep only predictive features

```
X = df.drop(columns=['cnt'])
y = df['cnt']
```

• Separates features (X) from target (y)

• `cnt`: Total bike rentals (what we predict)

• **Why**: Standard ML practice

• **Usage**: X for input, y for output

```
test_size = config['data']['test_size']
X_train, X_test , y_train , y_test = train_test_split (X , y , test_size=test_size, random_state=42)
```

• Splits data 80/20 (train/test)

• `random_state=42`: Reproducible splits

• **Why**: Evaluate model on unseen data

• **Usage**: Test set measures generalization

```
processed_dir = config['data']['processed_dir']
os.makedirs(processed_dir, exist_ok=True)
```

• Creates output directory

• `exist_ok=True`: No error if already exists

• **Why**: Ensures directory exists before saving

• **Usage**: Automatic directory creation

```
X_train.to_csv(f"{processed_dir}/X_train.csv" , index=False)
X_test.to_csv(f"{processed_dir}/X_test.csv" , index=False)
```

```
y_train.to_csv(f"{processed_dir}/y_train.csv" , index=False)
y_test.to_csv(f"{processed_dir}/y_test.csv" , index=False)
```

• Saves processed data

• `index=False`: Don't save row numbers

• **Why**: Separate files for each dataset

• **Usage**: Training script loads these files

When to Use:

• After data ingestion

• Before model training

• Manually: `python src/preprocessing.py`

src/train.py

Purpose: Trains Random Forest model and registers it in MLflow.

```
def train_model():
    if os.path.exists("/opt/airflow"):
        TRACKING_URI = "http://mlflow_server:5000"
        BASE_PATH = "/opt/airflow"
    else:
        TRACKING_URI = "http://localhost:5000"
        BASE_PATH = "."
```

• Detects environment (Docker vs local)

• **Why**: Works in both Airflow and local development

• **Usage**: Automatic environment detection

```
mlflow.set_tracking_uri(TRACKING_URI)
mlflow.set_experiment("bike-sharing-experiment")
```

• Connects to MLflow server

• Creates/uses experiment

• **Why**: Organizes all training runs

• **Usage**: View experiments at `http://localhost:5000`

```
X_train_path = os.path.join(BASE_PATH, "data/processed/X_train.csv")
y_train_path = os.path.join(BASE_PATH, "data/processed/y_train.csv")
save_path = os.path.join(BASE_PATH, "models/bike_model.pkl")
```

• Defines file paths

• **Why**: Environment-agnostic paths

• **Usage**: Works in Docker and Windows

```
try:
    X_train_raw = pd.read_csv(X_train_path)
    y_train = pd.read_csv(y_train_path)
    features = ['season', 'mnth', 'hr', 'holiday', 'weekday', 'workingday', 'weathersit', 'temp', 'atemp
    X_train = X_train_raw[[col for col in features if col in X_train_raw.columns]]
```

• Loads training data

• Selects specific features

• **Why**: Ensures consistent feature set

• **Usage**: Prevents feature mismatch errors

```
with mlflow.start_run(run_name="Airflow_Training_Run") as run:
    model = RandomForestRegressor(n_estimators=random.randint(100, 200), random_state=42)
    model.fit(X_train, y_train.values.ravel())
```

• Starts MLflow run

• Trains Random Forest model

• `n_estimators`: Random between 100-200

• **Why**: Tracks each training run

• **Usage**: Compare different runs in MLflow UI

```
rmse = np.sqrt(mean_squared_error(y_train, model.predict(X_train)))
mlflow.log_metric("rmse", rmse)
mlflow.log_param("n_estimators", model.n_estimators)
```

• Calculates training error

• Logs metrics and parameters to MLflow

• **Why**: Track model performance over time

• **Usage**: Compare RMSE across runs

```
joblib.dump(model, "bike_model.pkl")
mlflow.log_artifact("bike_model.pkl", artifact_path="model")
```

• Saves model as artifact

• **Why**: Stores model with experiment

• **Usage**: Download model from MLflow UI

```
model_name = "Bike_Sharing_Model"
run_id = run.info.run_id
model_uri = f"runs:/{run_id}/model"

print(f"■ Registering model from {model_uri}...")
```

• Prepares model registration

• **Why**: Model registry for versioning

• **Usage**: Track production models

```
client = MlflowClient()
try:
    client.create_registered_model(model_name)
except:
    pass
```

• Creates model in registry

• Ignores error if already exists

• **Why**: First-time setup

• **Usage**: One model name, multiple versions

```
            res = client.create_model_version(name=model_name, source=model_uri, run_id=run_id)
            current_version = res.version

            client.set_registered_model_alias(model_name, "champion", str(current_version))
```

• Creates new model version

• Sets "champion" alias to latest version

• **Why**: Easy reference to production model

• **Usage**: Load model by alias instead of version number

```
            os.makedirs(os.path.dirname(save_path), exist_ok=True)
            joblib.dump(model, save_path)
```

• Saves model locally

• **Why**: API loads from local file

• **Usage**: Fast predictions without MLflow

```
            print(f"■ SUCCESS! RMSE: {rmse:.4f} | Version: {current_version}")
```

• Success message with metrics

• **Why**: Confirms training completed

• **Usage**: Check Airflow logs for RMSE

```
    except Exception as e:
        print(f"■ ERROR: {str(e)}")
        sys.exit(1)
```

• Error handling

• **Why**: Fails Airflow task on error

• **Usage**: Debug from error message

When to Use:

• Airflow training task

• Manually: `python src/train.py`

• After data preprocessing

src/predict.py

Purpose: Loads model and makes predictions.

```
def make_prediction(input_data):
    BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
    model_path = os.path.join(BASE_DIR, "models", "bike_model.pkl")
```

• Finds model file relative to script location

• **Why**: Works from any directory

• **Usage**: Flexible path resolution

```
    if not os.path.exists(model_path):
        print(f"■ Model not found at: {model_path}")
```

```
            return [0.0]
```

• Checks if model exists
• Returns 0 if missing
• **Why**: Graceful degradation
• **Usage**: Prevents crashes

```
        model = joblib.load(model_path)
```

• Loads trained model
• **Why**: Fast loading from pickle file
• **Usage**: Model ready for predictions

```
        if hasattr(model, 'monotonic_cst'):
            delattr(model, 'monotonic_cst')
```

• Removes scikit-learn version-specific attribute
• **Why**: Compatibility across versions
• **Usage**: Prevents version mismatch errors

```
        try:
            input_df = pd.DataFrame([input_data]) if isinstance(input_data, dict) else pd.DataFrame(input_data)
```

• Converts input to DataFrame
• Handles both dict and DataFrame inputs
• **Why**: Flexible input format
• **Usage**: Works with API and direct calls

```
            cols = ['season', 'mnth', 'hr', 'holiday', 'weekday', 'workingday', 'weathersit', 'temp', 'atemp',
            final_df = input_df.reindex(columns=cols, fill_value=0)
            final_df = final_df.apply(pd.to_numeric)
```

• Ensures correct column order
• Fills missing columns with 0
• Converts to numeric types
• **Why**: Matches training data format
• **Usage**: Prevents feature order errors

```
            prediction = model.predict(final_df)
            return [float(prediction[0])]
```

• Makes prediction
• Returns as list of floats
• **Why**: Consistent return format
• **Usage**: Easy to parse in API

```
        except Exception as e:
            print(f"■ Prediction Logic Error: {e}")
            return [0.0]
```

• Error handling

• Returns 0 on error

• **Why**: API doesn't crash

• **Usage**: Check logs for errors

When to Use:

• Called by API for each prediction

• Called by Airflow for testing

• Import in other scripts: `from predict import make_prediction`

src/evaluate.py

Purpose: Evaluates model performance on test set.

```
def evaluate_model():
    with open("config.yaml" , "r") as f:
        config = yaml.safe_load(f)
```

• Loads configuration

• **Why**: Uses centralized paths

• **Usage**: Consistent with other scripts

```
processed_dir = config['data']['processed_dir']
X_test = pd.read_csv(f"{processed_dir}/X_test.csv")
y_test = pd.read_csv(f"{processed_dir}/y_test.csv")
```

• Loads test data

• **Why**: Evaluates on unseen data

• **Usage**: Measures generalization

```
model_path = config['model']['save_path']
model = joblib.load(model_path)
```

• Loads trained model

• **Why**: Evaluates latest model

• **Usage**: Must run after training

```
predictions = model.predict(X_test)
```

• Makes predictions on test set

• **Why**: Compare to actual values

• **Usage**: Calculates metrics

```
mse = mean_squared_error(y_test, predictions)
r2 = r2_score(y_test , predictions)
```

• Calculates evaluation metrics

• MSE: Mean Squared Error (lower is better)

• R^2: Coefficient of determination (higher is better, max 1.0)

• **Why**: Quantifies model performance

• **Usage**: Compare models

```
print("\n---- Model Report------------" )
print(f"Mean Squared Error: {mse:.2f}")
print(f"R2 Score: {r2:.2f}")
```

• Prints metrics

• **Why**: Easy to read results

• **Usage**: Check console output

When to Use:

• After training

• Manually: `python src/evaluate.py`

• Add to Airflow DAG for automated evaluation

src/validate_data.py

Purpose: Checks data quality before training.

```
def validate_data():
    path = "/opt/airflow/data/bike_sharing_raw.csv"

    if not os.path.exists(path):
        print(f"■ Error: File {path} not found!")
        sys.exit(1)
```

• Checks if data file exists

• **Why**: Prevents training on missing data

• **Usage**: Early failure detection

```
df = pd.read_csv(path)
print(f"■ Data loaded. Columns: {list(df.columns)}")
```

• Loads data and shows columns

• **Why**: Verifies file is readable

• **Usage**: Debug column name issues

```
errors = []

if 'cnt' not in df.columns:
    errors.append("Column 'cnt' missing")
elif df['cnt'].isnull().any():
    errors.append("Null values found in 'cnt'")
```

• Checks for target column

• Checks for missing values

• **Why**: Target is required for training

• **Usage**: Ensures data quality

```
if 'temp' not in df.columns:
```

```
            errors.append("Column 'temp' missing")
```

• Checks for key feature

• **Why**: Model needs temperature data

• **Usage**: Add more checks for other features

```
        if not errors:
            print("■ Data Validation SUCCESSFUL (via Pandas)!")
            sys.exit(0)
        else:
            print(f"■ Data Validation FAILED: {errors}")
            sys.exit(1)
```

• Reports validation result

• **Why**: Stops pipeline if data is bad

• **Usage**: Airflow marks task as failed

When to Use:

• After data ingestion

• Before preprocessing

• Extend with more validation rules

src/run_pipeline.py

Purpose: Standalone training script for retraining.

```
    def run_pipeline():
        print("\n■ Real Data Training Started...")
        csv_path = r'C:\MLOPS\data\bike_sharing_raw.csv'
```

• Trains model from scratch

• **Why**: Used by monitoring DAG for retraining

• **Usage**: Called when drift detected

```
        if not os.path.exists(csv_path):
            print(f"■ File Not Found at: {csv_path}")
            return
```

• Checks for data file

• **Why**: Graceful failure

• **Usage**: Returns without error

```
        df = pd.read_csv(csv_path)

        expected_cols = [
            'season', 'yr', 'mnth', 'hr', 'holiday', 'weekday',
            'workingday', 'weathersit', 'temp', 'atemp', 'hum', 'windspeed'
        ]

        X = df[expected_cols]
        y = df['cnt']
```

• Loads data and selects features

• **Why**: Self-contained training

• **Usage**: No dependency on preprocessing script

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

• Splits data

• **Why**: Evaluates new model

• **Usage**: Standard train/test split

```
model = RandomForestRegressor(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

• Trains model

• **Why**: Creates new model version

• **Usage**: Replaces old model

```
accuracy = model.score(X_test, y_test)
print(f"■ New Model Accuracy: {accuracy:.4f}")
```

• Calculates R^2 score

• **Why**: Verifies new model quality

• **Usage**: Compare to old model

```
joblib.dump(model, 'bike_model.pkl')
print("■ Model Saved as 'bike_model.pkl'!")
```

• Saves new model

• **Why**: Replaces production model

• **Usage**: API uses new model immediately

When to Use:

• Called by monitoring DAG

• Manually: `python src/run_pipeline.py`

• When drift detected

src/drift_detection.py

Purpose: Detects data drift using Evidently AI.

```
def check_model_drift(reference_path, current_data_df):
    reference_df = pd.read_csv(reference_path)
```

• Loads reference data (training data)

• **Why**: Baseline for comparison

• **Usage**: Pass path to training data

```
drift_report = Report(metrics=[DataDriftPreset()])
```

• Creates Evidently report

• **Why**: Pre-configured drift metrics

• **Usage**: Comprehensive drift analysis

```
        drift_report.run(reference_data=reference_df, current_data=current_data_df)
```

• Compares datasets

• **Why**: Statistical tests for each feature

• **Usage**: Detects distribution changes

```
        report_dict = drift_report.as_dict()
        drift_detected = report_dict['metrics'][0]['result']['dataset_drift']
```

• Extracts drift status

• **Why**: Boolean decision for automation

• **Usage**: True if significant drift

```
        return drift_detected, report_dict
```

• Returns drift status and full report

• **Why**: Both automated and manual analysis

• **Usage**: Use boolean for alerts, dict for details

When to Use:

• Import in monitoring scripts

• Customize drift thresholds

• Add to monitoring DAG

src/monitor.py

Purpose: Monitors model performance and triggers retraining.

```
    def monitor_and_retrain():
        print("\n■ Step 1: Loading model and data for monitoring...")

        model_path = 'bike_model.pkl'
        csv_path = r'C:\MLOPS\data\bike_sharing_raw.csv'
```

• Loads model and data

• **Why**: Checks current model performance

• **Usage**: Run periodically

```
        if not os.path.exists(model_path):
            print("■ Model not found! Starting training...")
            run_pipeline()
            return
```

• Trains model if missing

• **Why**: Bootstrap initial model

• **Usage**: First-time setup

```
model = joblib.load(model_path)
df = pd.read_csv(csv_path)
```

• Loads model and data

• **Why**: Evaluates current performance

• **Usage**: Uses latest data

```
expected_cols = [
    'season', 'yr', 'mnth', 'hr', 'holiday', 'weekday',
    'workingday', 'weathersit', 'temp', 'atemp', 'hum', 'windspeed'
]

X = df[expected_cols]
y_true = df['cnt']
```

• Prepares data

• **Why**: Matches training format

• **Usage**: Consistent feature set

```
print("■ Calculating current accuracy...")
y_pred = model.predict(X)
current_score = r2_score(y_true, y_pred)
print(f"■ Current Accuracy: {current_score:.4f}")
```

• Calculates current R^2 score

• **Why**: Measures model degradation

• **Usage**: Compare to initial performance

```
THRESHOLD = 0.92

if current_score < THRESHOLD:
    print(f"■■ Accuracy is below threshold {THRESHOLD}. Triggering Retraining...")
    run_pipeline()
    print("■ Success: Model has been retrained and updated!")
else:
    print("■ Accuracy is good. No action needed.")
```

• Compares to threshold

• Retrains if below threshold

• **Why**: Automated model maintenance

• **Usage**: Adjust threshold based on requirements

When to Use:

• Run manually: `python src/monitor.py`

• Schedule with cron or Task Scheduler

• Integrate with monitoring DAG

src/requirements.txt

Purpose: Lists all Python dependencies.

```
fastapi
```

```
uvicorn
```

• FastAPI framework and ASGI server
• **Why**: API serving
• **Usage**: `pip install fastapi uvicorn`

```
mlflow
```

• Experiment tracking and model registry
• **Why**: ML lifecycle management
• **Usage**: Track experiments

```
pandas==1.5.3
scikit-learn==1.3.2
```

• Data manipulation and ML library
• Specific versions for compatibility
• **Why**: Core ML functionality
• **Usage**: Data processing and modeling

```
joblib
pyyaml
```

• Model serialization and config parsing
• **Why**: Save/load models and configs
• **Usage**: Persistence

```
great_expectations==0.16.15
evidently==0.4.15
```

• Data validation and drift detection
• **Why**: Data quality and monitoring
• **Usage**: Automated checks

```
sqlalchemy
psycopg2-binary
```

• Database ORM and PostgreSQL driver
• **Why**: Database operations
• **Usage**: Store predictions

```
pydantic==1.10.13
```

• Data validation
• Specific version for compatibility
• **Why**: FastAPI dependency
• **Usage**: Request validation

```
prometheus-fastapi-instrumentator
```

• Prometheus metrics for FastAPI

• **Why**: API monitoring

• **Usage**: Automatic metrics

```
streamlit
requests
```

• Web UI framework and HTTP client

• **Why**: Frontend and API calls

• **Usage**: User interface

When to Use:

• Install all: `pip install -r src/requirements.txt`

• Update versions as needed

• Freeze versions: `pip freeze > requirements.txt`

4. CONFIGURATION FILES

prometheus.yml

Purpose: Configures Prometheus to scrape metrics from the API.

```
global:
  scrape_interval: 5s
```

• How often to collect metrics

• **Why**: 5 seconds for near real-time monitoring

• **Usage**: Increase for less frequent updates

```
scrape_configs:
  - job_name: 'prediction_api'
```

• Defines a monitoring job

• **Why**: Groups related metrics

• **Usage**: Add more jobs for other services

```
    static_configs:
      - targets: ['host.docker.internal:9999']
```

• Where to collect metrics from

• **Why**: API runs on Windows host

• **Usage**: Prometheus scrapes `/metrics` endpoint

When to Use:

• Automatically loaded by Prometheus container

• Modify to add more services
• View metrics at `http://localhost:9090`

.gitignore*

Purpose: Tells Git which files to ignore.

```
venv/
__pycache__/
```

• Python virtual environment and cache
• **Why**: Environment-specific, not needed in repo
• **Usage**: Automatic

```
mlruns/
```

• MLflow experiment data
• **Why**: Large files, regenerated locally
• **Usage**: Each developer has own experiments

```
data/raw/*.csv
data/processed/*.csv
```

• Data files
• **Why**: Large files, use DVC instead
• **Usage**: Track data with DVC

```
models/*.pkl
```

• Model files
• **Why**: Large binary files
• **Usage**: Store in MLflow or DVC

```
.DS_Store
.env
```

• Mac system files and environment variables
• **Why**: System-specific and secrets
• **Usage**: Never commit secrets

When to Use:
• Automatically used by Git
• Add patterns for new file types
• Keep secrets out of version control

.dvcignore*

Purpose: Tells DVC which files to ignore.

```
# Add patterns of files dvc should ignore, which could improve
# the performance. Learn more at
# https://dvc.org/doc/user-guide/dvcignore
```

• Currently empty (uses defaults)

• **Why**: DVC ignores .git, .dvc, etc. by default

• **Usage**: Add patterns if needed

When to Use:

• Exclude temporary files from DVC tracking

• Improve DVC performance

• Similar to .gitignore but for DVC

.dvc/config

Purpose: DVC configuration for data versioning.

```
[core]
    remote = local_storage
```

• Default remote storage

• **Why**: Where to push/pull data

• **Usage**: DVC commands use this remote

```
['remote "local_storage"']
    url = C:\dvc_storage
```

• Local directory for data storage

• **Why**: Simulates remote storage locally

• **Usage**: Change to S3/GCS for production

When to Use:

• Automatically used by DVC commands

• Configure remote storage

• `dvc push` uploads to this location

5. CI/CD FILES

.github/workflows/main.yml

Purpose: GitHub Actions workflow for automated testing and deployment.

```
name: Bike-Sharing MLOps CI/CD
```

• Workflow name

• **Why**: Identifies workflow in GitHub UI

• **Usage**: Shows in Actions tab

```
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
```

• Triggers workflow on push or PR to main

• **Why**: Automated testing on code changes

• **Usage**: Runs automatically

```
jobs:
  test-and-lint:
    runs-on: ubuntu-latest
```

• First job: code quality checks

• **Why**: Catch errors before deployment

• **Usage**: Runs on GitHub servers

```
    steps:
      - name: Checkout Code
        uses: actions/checkout@v3
```

• Downloads repository code

• **Why**: Needed to run tests

• **Usage**: Standard first step

```
      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'
```

• Installs Python 3.9

• **Why**: Matches development environment

• **Usage**: Consistent Python version

```
      - name: Install Linting Tools
        run: |
          pip install flake8
```

• Installs code linter

• **Why**: Checks code style

• **Usage**: Enforces standards

```
      - name: Lint with flake8
        run: |
          flake8 . --count --select=E9,F63,F7,F82 --show-source --statistics
          flake8 . --count --exit-zero --max-complexity=10 --max-line-length=127 --statistics
```

• Runs linting checks

• First line: Syntax errors (fails build)

• Second line: Style warnings (doesn't fail)

• **Why**: Maintains code quality

• **Usage**: Fix errors before merging

```
  build-check:
    needs: test-and-lint
    runs-on: ubuntu-latest
```

• Second job: build verification

• `needs`: Runs only if linting passes

• **Why**: Ensures Docker image builds

• **Usage**: Catches build errors early

```
    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Build Docker Image (Simulated CD)
        run: |
          echo "■ Starting Docker Build to verify Cloud Readiness..."
          docker build -t bike-sharing-mlops:${{ github.sha }} .
          echo "■ Build Successful! This image is ready for AWS/Azure deployment."
```

• Builds Docker image

• Tags with commit SHA

• **Why**: Verifies deployment readiness

• **Usage**: Catches Docker errors

```
      - name: Success Message
        run: echo "Final CI/CD Check Passed. The pipeline is robust and production-ready!"
```

• Success confirmation

• **Why**: Clear feedback

• **Usage**: Shows in GitHub Actions log

When to Use:

• Automatically runs on every push

• View results in GitHub Actions tab

• Extend with deployment steps

Usage Guide

Initial Setup

1. **Prerequisites**

```bash

# Install Docker Desktop
```

Install Python 3.9
Install Git
```

2. **Clone Repository**
```bash
git clone <repository-url>
cd bike-sharing-mlops
```

3. **Prepare Data**
```bash
# Download data to data/bike_sharing_raw.csv
# Or use DVC: dvc pull
```

4. **Start Services**
```bash
docker-compose up -d
```

5. **Access Services**
- Airflow UI: http://localhost:8081 (admin/admin)
- MLflow UI: http://localhost:5000
- Streamlit App: http://localhost:8501
- Prometheus: http://localhost:9090
- Grafana: http://localhost:3000

6. **Start API on Windows**
```bash
python src/api.py
```

Running the Pipeline

Option 1: Airflow (Automated)
1. Open Airflow UI: http://localhost:8081
2. Enable DAG: `bike_sharing_final_pipeline_v4`
3. Click "Trigger DAG" button
4. Monitor task progress in Graph view

Option 2: Manual Execution
```

```
# Step 1: Ingest data
python src/ingestion.py

# Step 2: Preprocess
python src/preprocessing.py

# Step 3: Train model
python src/train.py

# Step 4: Evaluate
python src/evaluate.py

# Step 5: Make predictions
python src/api.py  # In separate terminal
```

Making Predictions

Via Streamlit UI

1. Open http://localhost:8501
2. Adjust sliders for features
3. Click "Predict Bike Demand"
4. View result

Via API (Browser)

```
http://localhost:9999/predict?temp=0.8&hr=18&season=2
```

Via API (Python)

```
import requests

params = {
    "season": 2,
    "mnth": 6,
    "hr": 18,
    "temp": 0.8,
    "hum": 0.6,
    "windspeed": 0.2
}

response = requests.get("http://localhost:9999/predict", params=params)
print(response.json())
```

Via API (cURL)

```
curl "http://localhost:9999/predict?temp=0.8&hr=18"
```

Monitoring

View Drift Reports

1. Run monitoring DAG in Airflow
2. Open Streamlit app
3. Click "View Data Drift Report"
4. Review Evidently AI report

View Prometheus Metrics

1. Open http://localhost:9090
2. Query: `http_requests_total`
3. View request counts and latencies

View Grafana Dashboards

1. Open http://localhost:3000
2. Login: admin/admin
3. Add Prometheus data source
4. Create dashboards

Troubleshooting

API Connection Failed

```
# Check if API is running
curl http://localhost:9999/

# Check Docker network
docker network inspect bike-mlops-net

# Restart API
python src/api.py
```

Airflow Task Failed

```
# View logs
docker logs airflow_scheduler

# Check task logs in Airflow UI
# Click on failed task > Log
```

Model Not Found

```
# Train model manually
python src/train.py

# Check model file
ls models/bike_model.pkl
```

Database Connection Error

```
# Check PostgreSQL
docker ps | grep postgres

# Restart database
docker-compose restart postgres
```

Deployment Instructions

Local Development

• Use docker-compose for all services

• Run API on Windows host

• Access via localhost

Production Deployment

AWS Deployment

1. **Setup ECS/EKS**

```bash
# Push images to ECR
docker tag bike-sharing-mlops:latest <ecr-url>
docker push <ecr-url>
```

2. **Configure RDS**

- Replace PostgreSQL with RDS
- Update connection strings

3. **Setup S3**

- Replace LocalStack with real S3
- Update bucket configurations

4. **Configure ALB**

- Load balancer for API
- SSL certificates

Azure Deployment

1. **Azure Container Instances**

```bash
az container create --resource-group mygroup --name bike-mlops --image <acr-url>
```

2. **Azure Database for PostgreSQL**

- Managed database service
- Update connection strings

3. **Azure Blob Storage**

- Replace LocalStack
- Update storage configs

Kubernetes Deployment

1. **Create Kubernetes manifests**

```yaml
# deployment.yaml
```

service.yaml

ingress.yaml

```

2. **Deploy**

```bash

kubectl apply -f k8s/

```

3. **Configure Helm charts**

- Package application

- Manage releases

Environment Variables

```
# Production .env file
DATABASE_URL=postgresql://user:pass@prod-db:5432/mlops
MLFLOW_TRACKING_URI=http://mlflow.prod:5000
S3_BUCKET=prod-bike-sharing
SLACK_WEBHOOK=https://hooks.slack.com/...
```

Security Considerations

• Use secrets management (AWS Secrets Manager, Azure Key Vault)

• Enable SSL/TLS for all services

• Implement authentication (OAuth, JWT)

• Network isolation (VPC, Security Groups)

• Regular security updates

• Audit logging

Scaling Considerations

• Horizontal scaling for API (multiple replicas)

• Database read replicas

• Caching layer (Redis)

• CDN for static assets

• Auto-scaling based on metrics

Key Concepts Explained

MLOps Pipeline

1. **Data Ingestion**: Load raw data

2. **Data Validation**: Check quality
```

3. **Preprocessing**: Clean and split
4. **Training**: Build model
5. **Evaluation**: Measure performance
6. **Deployment**: Serve predictions
7. **Monitoring**: Track drift and performance
8. **Retraining**: Update model when needed

Data Drift

• **What**: Changes in data distribution over time
• **Why Important**: Model trained on old data may not work on new data
• **Detection**: Evidently AI compares current vs reference data
• **Action**: Retrain model when drift detected

Model Registry

• **What**: Centralized storage for model versions
• **Why Important**: Track which model is in production
• **MLflow**: Provides versioning and aliasing
• **Usage**: Load model by alias ("champion")

Experiment Tracking

• **What**: Recording model parameters and metrics
• **Why Important**: Compare different models
• **MLflow**: Tracks runs, parameters, metrics, artifacts
• **Usage**: View all experiments in MLflow UI

Containerization

• **What**: Packaging application with dependencies
• **Why Important**: Consistent environment across machines
• **Docker**: Creates isolated containers
• **Usage**: `docker-compose up` starts all services

Orchestration

• **What**: Automating workflow execution
• **Why Important**: Scheduled and reliable pipelines
• **Airflow**: DAGs define task dependencies
• **Usage**: Schedule daily retraining

API Serving

• **What**: Exposing model as web service
• **Why Important**: Applications can get predictions

• **FastAPI**: Modern, fast Python framework
• **Usage**: HTTP GET/POST requests

Monitoring

• **What**: Tracking system health and performance
• **Why Important**: Detect issues early
• **Prometheus**: Metrics collection
• **Grafana**: Visualization
• **Usage**: Dashboards and alerts

Best Practices

Code Quality

• Use linting (flake8, pylint)
• Write docstrings
• Type hints for functions
• Unit tests for critical functions
• Code reviews

Data Management

• Version data with DVC
• Validate data quality
• Document data schema
• Handle missing values
• Monitor data drift

Model Development

• Track experiments in MLflow
• Use cross-validation
• Feature engineering
• Hyperparameter tuning
• Model versioning

Deployment

• Containerize applications
• Use environment variables
• Implement health checks
• Graceful error handling

• Logging and monitoring

Security

• Never commit secrets
• Use environment variables
• Implement authentication
• Regular updates
• Audit logs

Documentation

• README for setup
• Code comments
• API documentation
• Architecture diagrams
• Runbooks for operations

Conclusion

This project demonstrates a complete MLOps pipeline with:
• Automated training and deployment
• Continuous monitoring and retraining
• Scalable architecture
• Production-ready practices

Key Takeaways:
1. Automation reduces manual errors
2. Monitoring prevents model degradation
3. Containerization ensures consistency
4. Version control tracks changes
5. CI/CD enables rapid iteration

Next Steps:
• Add more features to model
• Implement A/B testing
• Add more monitoring metrics
• Scale to production workload
• Implement model explainability

Appendix

Useful Commands

```
# Docker
docker-compose up -d          # Start all services
docker-compose down           # Stop all services
docker-compose logs -f        # View logs
docker ps                     # List containers
docker exec -it <container> bash  # Enter container

# Airflow
airflow dags list             # List DAGs
airflow tasks test <dag> <task> <date>  # Test task
airflow dags trigger <dag>    # Trigger DAG

# MLflow
mlflow ui                     # Start UI
mlflow models serve -m <model>  # Serve model

# DVC
dvc init                      # Initialize DVC
dvc add data/                 # Track data
dvc push                      # Upload data
dvc pull                      # Download data

# Git
git add .                     # Stage changes
git commit -m "message"       # Commit
git push                      # Push to remote
```

Port Reference

- 5432: PostgreSQL
- 4566: LocalStack (S3)
- 5000: MLflow
- 8081: Airflow
- 8501: Streamlit
- 9999: FastAPI
- 9090: Prometheus
- 3000: Grafana

File Structure

```
bike-sharing-mlops/
■■■ .dvc/                 # DVC configuration
■■■ .github/workflows/    # CI/CD pipelines
■■■ dags/                 # Airflow DAGs
■■■ data/                 # Data files (not in Git)
■   ■■■ raw/
■   ■■■ processed/
■■■ models/               # Model files (not in Git)
■■■ src/                  # Source code
■   ■■■ api.py
■   ■■■ app.py
■   ■■■ train.py
■   ■■■ predict.py
■   ■■■ ...
■■■ docker-compose.yml    # Service orchestration
■■■ Dockerfile            # Streamlit image
■■■ Dockerfile.airflow    # Airflow image
■■■ prometheus.yml        # Monitoring config
■■■ README.md             # Project overview
```

Document Version: 1.0

Last Updated: February 15, 2026

Author: MLOps Team

Contact: [Your Contact Information]