1. Problem Formulation

1.1 Business Problem Definition

RecoMart is an e-commerce platform that aims to improve **customer engagement and conversion rate** by providing **personalized product recommendations** to its users.

Currently, users browse products without personalized guidance, which results in:

- Lower click-through rates
- Missed cross-selling opportunities
- Reduced average order value

Business Goal:
Design a data-driven recommendation system that suggests relevant products to users based on their past behavior and product characteristics, thereby improving:

- Conversion rate
- User engagement
- Cross-selling effectiveness

The recommendation system must be continuously updated using fresh user interaction and transaction data.

1.2 Key Data Sources and Attributes

RecoMart collects data from multiple sources. The pipeline integrates the following key datasets:

1. User Interaction Data (Clickstream Logs)

Captured from web and mobile platforms.

Attributes:

- `user_id` – Unique identifier for users
- `product_id` – Identifier of interacted product
- `event_type` – View, click, add-to-cart
- `timestamp` – Time of interaction
- `device` – Web or mobile

This data reflects **implicit user preferences**.

---

2. Transactional Purchase Data

Records confirmed purchases made by users.

Attributes:

- `transaction_id`
- `user_id`
- `product_id`
- `quantity`
- `price`
- `timestamp`

This data represents **explicit user intent and value-based interactions**.

---

3. Product Metadata (Catalog / API)

Fetched from internal or external product services.

Attributes:

- `product_id`
- `category`
- `brand`
- `price`
- `popularity_score`

This data supports **content-based recommendations** and feature enrichment.

1.3 Expected Outputs from the Pipeline

The end-to-end data pipeline is expected to generate the following outputs:

1. Cleaned and Validated Datasets

- Structured and validated datasets for exploratory data analysis (EDA)
- Removal of duplicates, handling missing values, and schema consistency

2. Engineered Feature Sets

Features suitable for recommendation algorithms, such as:

- User activity frequency
- Product popularity
- Average user-item interaction strength
- Aggregated behavioral statistics

These features support:

- Collaborative filtering

- Content-based recommendation models

3. Deployable Recommendation Model

- A trained recommendation model capable of generating personalized product suggestions
- A simple inference interface to retrieve top-N product recommendations for a user

---

1.4 Evaluation Metrics

The recommendation model will be evaluated using **ranking-based metrics**, which are standard for recommendation systems:

- **Precision@K**
 Measures the proportion of relevant items among the top-K recommended products.
- **Recall@K**
 Measures the proportion of relevant items successfully retrieved in the top-K recommendations.
- **Normalized Discounted Cumulative Gain (NDCG)**
 Evaluates ranking quality by assigning higher importance to correctly ranked items at higher positions.

These metrics align directly with the business objective of improving recommendation relevance and user engagement.

1.5 Expected Pipeline Outcomes

By implementing this pipeline, RecoMart will achieve:

- A scalable and automated data ingestion and processing system
- High-quality, versioned datasets for machine learning

- Reproducible feature generation for training and inference
- A recommendation model that learns continuously from fresh data

2. Data Collection and Ingestion

2.1 Data Sources Ingested

The pipeline ingests data from **multiple heterogeneous sources**, simulating a real-world e-commerce data ecosystem:

1. User Interaction Data (Clickstream)

- **Source Type:** CSV files
- **Origin:** Web and mobile platforms
- **Ingestion Mode:** Batch (daily)

This data captures **implicit user behavior**, such as views, clicks, and add-to-cart events.

---

2. Product Metadata

- **Source Type:** REST API

API Endpoint:

```
https://fakestoreapi.com/products
```

-
- **Ingestion Mode:** Automated API-based ingestion

This API provides product catalog information including price, category, ratings, and descriptions, which supports **content-based feature generation**.

2.2 Ingestion Design

Each ingestion script is designed following modern data engineering best practices.

Automated and Periodic Fetching

- Ingestion scripts are designed to run periodically (e.g., daily)
- Date-based folder partitioning enables incremental ingestion and replay

---

Error Handling and Retry Mechanism

- API ingestion includes retry logic to handle transient failures
- HTTP errors and network issues are captured and retried up to a configurable limit
- Failures do not corrupt previously ingested data

Logging and Audit Trails

- All ingestion activities are logged using a centralized logging mechanism
- Logs capture:
  - Start and end of ingestion
  - Success or failure status
  - Retry attempts and error messages

2.3 Ingestion Implementation

Clickstream Data Ingestion

- Reads interaction data from CSV files

Writes raw data into a timestamp-partitioned data lake structure:

```
data/raw/clickstream/YYYY/MM/DD/
```

-

Product Metadata Ingestion

- Fetches product data from the Fake Store REST API
- Stores the **raw API response without transformation** to preserve source fidelity

Writes data to:

```
data/raw/products/YYYY/MM/DD/products.json
```

-

Storing raw API responses ensures reproducibility and allows downstream transformations to be re-applied if needed.

2.4 Raw Data Storage Structure

```
data/raw/
├── clickstream/
│   └── YYYY/MM/DD/clickstream.csv
└── products/
    └── YYYY/MM/DD/products.json
```

This structure mirrors cloud-based data lake designs and supports scalable downstream processing.

2.5 Logs Showing Ingestion Success and Failure

All ingestion runs generate logs stored at:

```
logs/ingestion.log
```

Successful Ingestion Example

```
INFO - Starting product API ingestion
INFO - Product data ingested successfully from API at
data/raw/products/2025/01/01/products.json
```

Failure and Retry Example

```
ERROR - Attempt 1 failed: Connection timeout
ERROR - Attempt 2 failed: Connection timeout
ERROR - Product ingestion failed after retries
```

2.6 Summary

This ingestion layer:

- Integrates batch and API-based data sources
- Ensures fault tolerance through retries and logging
- Preserves raw data for lineage and reproducibility
- Provides a reliable foundation for downstream validation and feature engineering

3. Raw Data Storage

3.1 Storage Approach

The ingested data is stored in a **local filesystem-based data lake**, which simulates cloud object storage systems such as AWS S3.
Using the local filesystem allows the pipeline to be executed and validated without external dependencies while still following modern data lake design principles.

All storage paths are **centrally managed using configuration files**, ensuring that data locations are not hardcoded in the ingestion logic.

3.2 Data Lake Folder Structure

Raw data is organized using a structured, hierarchical folder layout based on:

- **Data source** (clickstream, products)
- **Ingestion date** (YYYY/MM/DD)

Raw Data Layout

```
data/raw/
├── clickstream/
│   └── YYYY/
│       └── MM/
│           └── DD/
│               └── clickstream.csv
│
└── products/
    └── YYYY/
        └── MM/
            └── DD/
                └── products.json
```

This layout mirrors industry-standard data lake structures used in large-scale data platforms.

3.3 Partitioning Strategy

A **date-based partitioning strategy** is used for raw data storage:

```
/<data_source>/<YYYY>/<MM>/<DD>/
```

Benefits:

- Enables incremental data processing
- Supports historical reprocessing and backfills
- Simplifies debugging and auditability
- Preserves ingestion-time data lineage

3.4 Storage Configuration

All storage locations are defined in a centralized configuration file:

📄 `config/paths.yaml`

```yaml
raw: data/raw
validated: data/validated
processed: data/processed
features: data/features
logs: logs
source_files: data/source_files
```

Ingestion scripts dynamically resolve storage paths using this configuration, allowing the pipeline to remain flexible and environment-independent.

3.5 Data Upload Mechanism

During execution, ingestion scripts:

- Read source data from configured upstream locations
- Create date-partitioned directories automatically
- Write raw data files to the data lake
- Log storage paths and execution status

No manual upload steps are required, as data is written programmatically during ingestion.

3.6 Traceability and Reproducibility

- Raw data files are stored as **immutable artifacts**
- Each ingestion run creates a new, timestamp-partitioned directory
- Ingestion logs capture file paths and execution timestamps

This design ensures full traceability and allows downstream pipeline stages to be re-run reliably using historical raw data.

Data Quality Report

This report summarizes data profiling and validation checks performed on raw datasets.

Clickstream Data Validation Summary

Dataset	File	Rows	Missing Values	Duplicates	Invalid Events	Invalid Device
Clickstream	data\raw\clickstream\2026\01\10\clickstream.csv	61	0	0	-62	-62
Clickstream	data\raw\clickstream\2026\01\20\clickstream.csv	61	0	0	-62	-62

Product Data Validation Summary

Dataset	File	Rows	Missing Values	Duplicates	Invalid Price	Invalid Rating
Products	data\raw\products\2026\01\10\products.json	20	0	0	0	0
Products	data\raw\products\2026\01\20\products.json	20	0	0	0	0

5. Data Preparation and Exploratory Data Analysis

5.1 Preparation Approach

The data preparation stage transforms validated raw data into a **clean, structured, and machine-learning-ready dataset** suitable for feature engineering and recommendation modeling.

This stage focuses on:

- Cleaning and filtering interaction data
- Enriching user interactions with product metadata
- Encoding categorical attributes
- Normalizing numerical variables
- Generating reproducible exploratory analysis artifacts

All transformations are performed programmatically to ensure **consistency and reproducibility**.

5.2 Data Cleaning and Enrichment

The following cleaning steps are applied to the clickstream interaction data:

- Removal of records with missing `user_id` or `product_id`
- Filtering to retain valid interaction types (`view`, `click`, `add_to_cart`)
- Conversion of timestamps into standardized datetime format

User interaction data is then **enriched** by joining with product metadata using `product_id`, allowing interaction records to include product attributes such as category, price, and ratings.

5.3 Feature Encoding and Normalization

To prepare the dataset for downstream modeling, the following transformations are applied:

Categorical Encoding

- Interaction types are mapped to numerical interaction strength values:
  - `view = 1`
  - `click = 2`
  - `add_to_cart = 3`
- Product categories are label-encoded into numerical identifiers

Numerical Normalization

- Product prices are normalized using min–max scaling
- Temporal information is extracted from timestamps (hour of day) to capture time-based user behavior patterns

These steps ensure that all features are represented in a format suitable for machine learning algorithms.

5.4 Exploratory Data Analysis (EDA)

Exploratory analysis is conducted to understand the structure and characteristics of the prepared data. The following analyses are performed:

- **Interaction distribution** across event types

- **Item popularity analysis** based on user interactions
- **User–item sparsity analysis** to quantify the sparsity of the interaction matrix

All EDA visualizations are generated automatically and saved as reproducible artifacts.

5.5 Prepared Data and EDA Artifacts

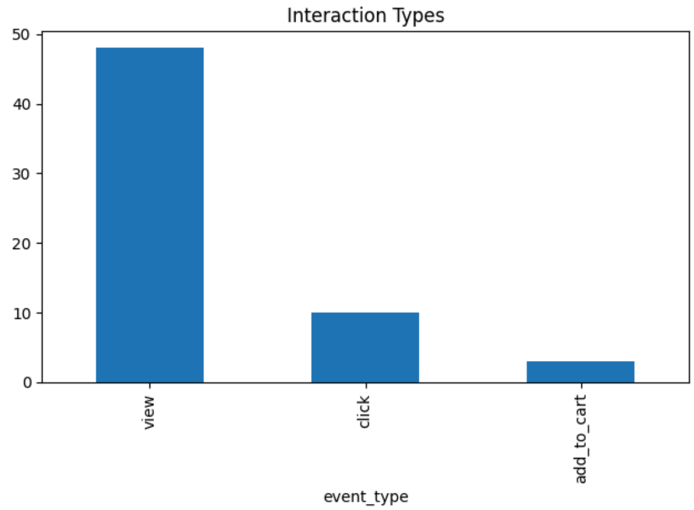
Processed Data Output

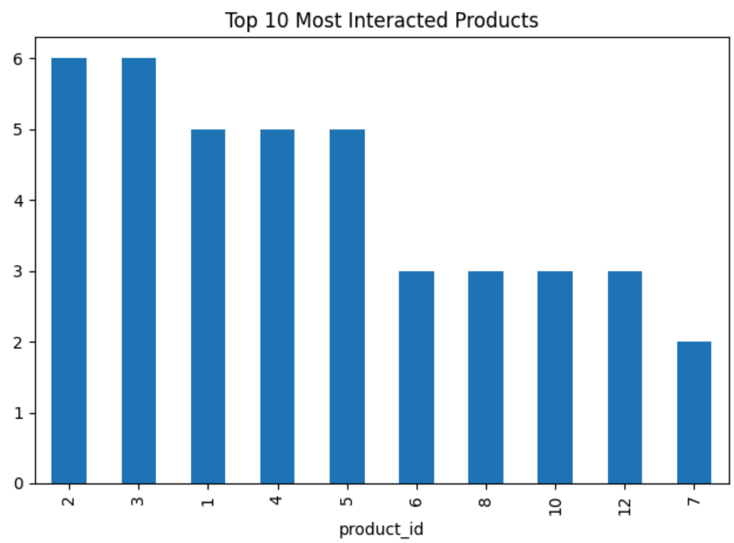
```
data/processed/prepared_interactions.csv
```

This dataset contains cleaned and enriched interaction records and is ready for feature engineering and model training.

EDA Artifacts

```
data/processed/eda/
├── interaction_distribution.png
└── top_products.png
```





These visualizations summarize key interaction patterns and are used for analysis and reporting.

5.6 Reproducibility and Pipeline Readiness

- All preparation steps are implemented as a standalone script
- Outputs are deterministic and reproducible
- Generated datasets and plots can be versioned and reused in downstream stages

This preparation layer provides a reliable foundation for **feature engineering and transformation** in the subsequent pipeline stage.

6. Feature Engineering and Transformation

6.1 SQL Schema

The following PostgreSQL schemas are used to store engineered features for the recommendation system. These tables act as a structured feature warehouse layer.

User Features Table

```
CREATE TABLE user_features (
    user_id VARCHAR PRIMARY KEY,
    total_interactions INT,
    avg_interaction_score FLOAT
);
```

Description:

- Stores aggregated user-level behavioral features
- Captures overall user activity and engagement strength

Item Features Table

```
CREATE TABLE item_features (
    product_id INT PRIMARY KEY,
    total_interactions INT,
    avg_interaction_score FLOAT,
    avg_rating FLOAT
);
```

Description:

- Stores item-level popularity and quality signals

- Supports popularity-based and content-aware recommendations

---

User–Item Interaction Features Table

```
CREATE TABLE user_item_features (
    user_id VARCHAR,
    product_id INT,
    interaction_count INT,
    total_interaction_score FLOAT,
    PRIMARY KEY (user_id, product_id)
);
```

Description:

- Captures historical interaction strength between users and items
- Forms the basis for collaborative filtering and matrix factorization models

---

6.2 Transformation Scripts

Feature transformation is implemented using a standalone Python script:

Script Location:

```
src/transformation/feature_engineering.py
```

Responsibilities of the Script:

- Load prepared interaction data from `data/processed/prepared_interactions.csv`
- Aggregate user-level, item-level, and user–item features
- Connect to PostgreSQL using configuration-driven credentials

- Persist engineered features into relational tables

The script is designed to be **idempotent**, allowing safe re-execution during experimentation or pipeline reruns.

6.3 Summary of Feature Logic

User-Level Features

- **Total Interactions:** Number of interactions performed by a user
- **Average Interaction Score:** Mean interaction strength across all user actions

Purpose:

- Represents user engagement intensity
- Useful for user profiling and cold-start handling

Item-Level Features

- **Total Interactions:** Number of interactions received by an item
- **Average Interaction Score:** Mean engagement strength per item
- **Average Rating:** Mean product rating from metadata

Purpose:

- Captures item popularity and perceived quality
- Supports popularity-based and hybrid recommendation models

User–Item Interaction Features

- **Interaction Count:** Number of times a user interacted with an item
- **Total Interaction Score:** Cumulative interaction strength

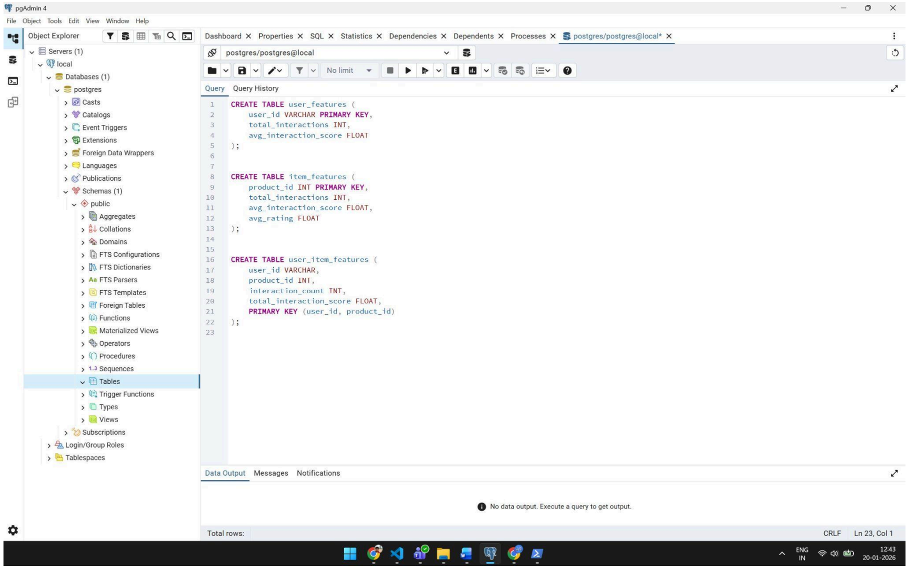
Purpose:

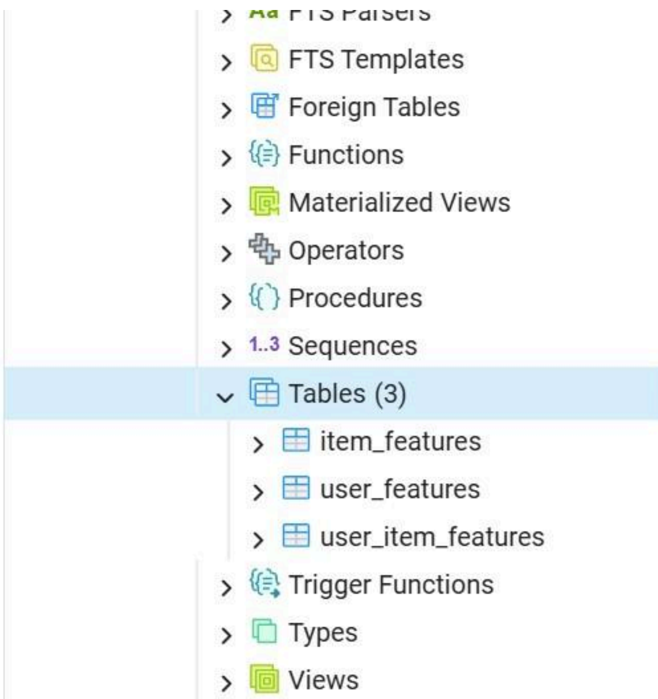
- Encodes historical preference strength
- Directly usable in collaborative filtering algorithms

6.4 Outcome

The engineered features are stored in a structured relational format and are ready for:

- Model training
- Inference-time feature retrieval
- Feature store abstraction in subsequent pipeline stages


pgAdmin 4
File Object Tools Edit View Window Help
Object Explorer
Dashboard Properties SQL Statistics Dependencies Dependents Processes postgres/postgres@local*
postgres/postgres@local
No limit
Query Query History
CREATE TABLE user_features (
user_id VARCHAR PRIMARY KEY,
total_interactions INT,
avg_interaction_score FLOAT
);
CREATE TABLE item_features (
product_id INT PRIMARY KEY,
total_interactions INT,
avg_interaction_score FLOAT,
avg_rating FLOAT
);
CREATE TABLE user_item_features (
user_id VARCHAR,
product_id INT,
interaction_count INT,
total_interaction_score FLOAT,
PRIMARY KEY (user_id, product_id)
);
Servers (1)
local
Databases (1)
postgres
Casts
Catalogs
Event Triggers
Extensions
Foreign Data Wrappers
Languages
Publications
Schemas (1)
public
Aggregates
Collations
Domains
FTS Configurations
FTS Dictionaries
FTS Parsers
FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
Sequences
Tables
Trigger Functions
Types
Views
Subscriptions
Login/Group Roles
Tablespaces
Data Output Messages Notifications
No data output. Execute a query to get output.
Total rows:
CRLF Ln 23, Col 1
ENG IN
12:43
20-01-2026


FTS Templates
Foreign Tables
Functions
Materialized Views
Operators
Procedures
1..3 Sequences
Tables (3)
item_features
user_features
user_item_features
Trigger Functions
Types
Views

public.user_features/postgres/postgres@local

No limit

Query | Query History

```sql
SELECT * FROM public.user_features
```

Data Output | Messages | Notifications

Showing rows: 1 to 20 | Page No: 1 of 1

	user_id text	total_interactions bigint	avg_interaction_score double precision
1	U1	5	1.6
2	U10	2	1
3	U11	3	1.3333333333333333
4	U12	2	1
5	U13	3	1.3333333333333333
6	U14	2	1
7	U15	3	1.6666666666666667
8	U16	2	1
9	U17	3	1.3333333333333333
10	U18	2	1
11	U19	2	1
12	U2	4	1.25
13	U20	4	1
14	U3	4	1.25
15	U4	4	1.25
16	U5	4	1.5
17	U6	3	1.3333333333333333
18	U7	3	1.3333333333333333
19	U8	3	1
20	U9	3	1.3333333333333333

✓ Successfully run. Total query runtime: 74 msec. 20 rows affected.

✓ local/postgres - Database connected

Total rows: 20 Query complete 00:00:00.074 CRLF Ln 1, Col 1

public.user_item_features/postgres/postgres@local

No limit

Query | Query History

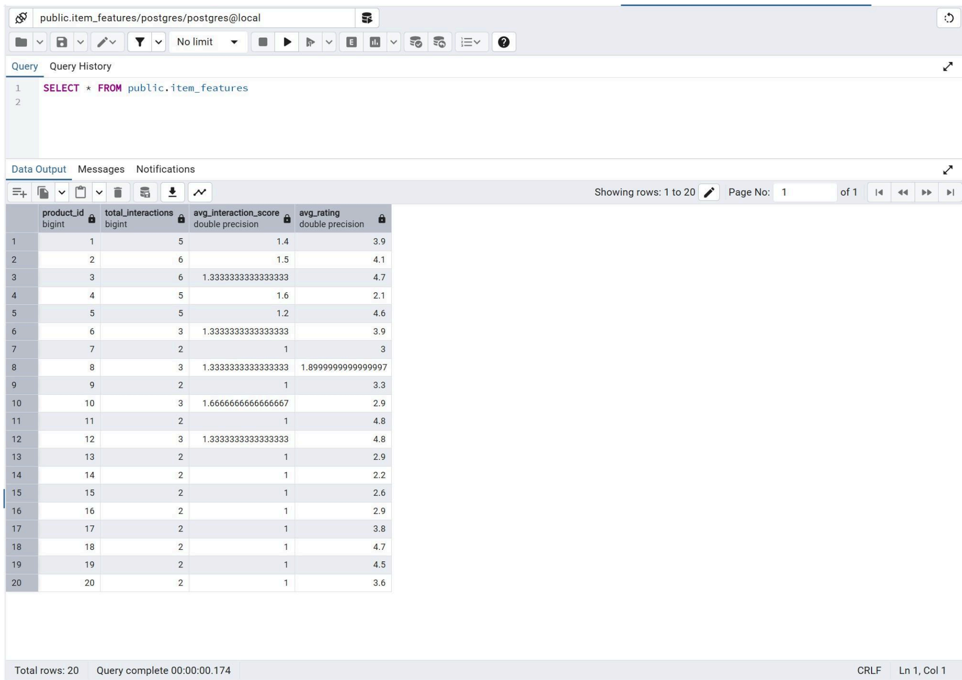
```sql
SELECT * FROM public.user_item_features
```

Data Output | Messages | Notifications

Showing rows: 1 to 48 | Page No: 1 of 1

	user_id text	product_id bigint	interaction_count bigint	total_interaction_score bigint
1	U1	1	2	3
2	U1	2	2	4
3	U1	3	1	1
4	U10	5	1	1
5	U10	13	1	1
6	U11	6	2	3
7	U11	14	1	1
8	U12	7	1	1
9	U12	15	1	1
10	U13	8	2	3
11	U13	16	1	1
12	U14	9	1	1
13	U14	17	1	1
14	U15	10	2	4
15	U15	18	1	1
16	U16	11	1	1
17	U16	19	1	1
18	U17	12	2	3
19	U17	20	1	1
20	U18	13	1	1
21	U18	14	1	1
22	U19	15	1	1
23	U19	16	1	1
24	U2	3	2	3

Total rows: 48 Query complete 00:00:00.104 CRLF Ln 1, Col 1



public.item_features/postgres/postgres@local

Query Query History

```sql
SELECT * FROM public.item_features
```

Data Output Messages Notifications

Showing rows: 1 to 20 Page No: 1 of 1

	product_id bigint	total_interactions bigint	avg_interaction_score double precision	avg_rating double precision
1	1	5	1.4	3.9
2	2	6	1.5	4.1
3	3	6	1.3333333333333333	4.7
4	4	5	1.6	2.1
5	5	5	1.2	4.6
6	6	3	1.3333333333333333	3.9
7	7	2	1	3
8	8	3	1.3333333333333333	1.8999999999999997
9	9	2	1	3.3
10	10	3	1.6666666666666667	2.9
11	11	2	1	4.8
12	12	3	1.3333333333333333	4.8
13	13	2	1	2.9
14	14	2	1	2.2
15	15	2	1	2.6
16	16	2	1	2.9
17	17	2	1	3.8
18	18	2	1	4.7
19	19	2	1	4.5
20	20	2	1	3.6

Total rows: 20 Query complete 00:00:00.174 CRLF Ln 1, Col 1

This transformation layer bridges raw interaction data and recommendation models in a scalable and reproducible manner.

7. Feature Store

7.1 Feature Store Approach

A **custom feature store** is implemented to manage, document, and retrieve machine learning features in a consistent and versioned manner.

The feature store is designed to:

- Centralize access to engineered features
- Ensure consistency between training and inference
- Support feature versioning and metadata management

PostgreSQL is used as the **offline feature storage layer**, while a lightweight Python-based registry manages feature definitions and versions.

7.2 Feature Store Architecture

The feature store is built on top of the feature engineering pipeline and consists of the following components:

```
Prepared Dataset
      ↓
Feature Engineering (Step 6)
      ↓
PostgreSQL Feature Tables
      ↓
Feature Registry (Metadata + Versions)
      ↓
Feature Retrieval APIs
```

This architecture mirrors the design of production-grade feature stores while remaining simple and easy to maintain.

7.3 Feature Storage Layer

Engineered features are stored in PostgreSQL tables created during Step 6:

- `user_features`
- `item_features`
- `user_item_features`

These tables act as the **offline feature store**, enabling efficient querying and reuse across different stages of the ML pipeline.

7.4 Feature Metadata Registry

A centralized **feature registry** is used to document feature metadata and manage feature versions.

📄 Registry Location

`src/feature_store/registry.py`

The registry captures:

- Feature group (user, item, user–item)
- Feature names
- Source tables
- Feature descriptions
- Feature version identifiers

This ensures feature discoverability and traceability.

7.5 Feature Versioning Strategy

Feature versions are managed using explicit version identifiers (e.g., `v1`).

Each version:

- Represents a stable set of feature definitions
- Can be independently retrieved for training or inference
- Enables backward compatibility when features evolve

Versioning guarantees reproducibility of model training and evaluation.

7.6 Feature Retrieval Mechanism

Feature retrieval is implemented using reusable Python functions that query PostgreSQL based on the requested feature version.

Retrieval functions support:

- User-level feature lookup
- Item-level feature lookup
- User–item interaction feature lookup

The same retrieval logic is used during **model training and inference**, ensuring feature consistency across the ML lifecycle.

7.7 Sample Feature Retrieval Demonstration

A standalone script demonstrates feature retrieval from the feature store.

📄 Script Location

`src/feature_store/demo_feature_retrieval.py`

The script:

- Retrieves user features for a given user ID
- Retrieves item features for a given product ID
- Retrieves user–item interaction features
- Uses versioned feature access (v1)

```
PROBLEMS 2   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   COMMENTS

(.venv) PS D:\BITS\DM4ML\DM4ML-Assignment-1> python -m src.feature_store.demo_feature_retrieval
=== Feature Store Demo (Version v1) ===

User Features for user_id = U1
  user_id  total_interactions  avg_interaction_score
0      U1                   5                    1.6

Item Features for product_id = 3
   product_id  total_interactions  avg_interaction_score  avg_rating
0           3                   6               1.333333         4.7

User-Item Features for user_id = U1, product_id = 3
  user_id  product_id  interaction_count  total_interaction_score
0      U1           3                  1                        1
(.venv) PS D:\BITS\DM4ML\DM4ML-Assignment-1>
```

This demonstrates how downstream ML components can consume features reliably.

7.8 Training and Inference Consistency

- Both training and inference pipelines access features through the same APIs
- Feature definitions and transformations remain synchronized
- Versioned retrieval prevents feature drift between environments

This ensures reliable and repeatable model behavior.

7.9 Summary

The feature store provides a structured, versioned, and documented mechanism for managing machine learning features. It bridges the gap between feature engineering and model consumption, enabling scalable and reproducible recommendation workflows.

8. Data Versioning and Lineage

8.1 Versioning Approach

Data versioning and lineage tracking are implemented using **DVC (Data Version Control)**. DVC enables version control of large datasets without storing the actual data files in the Git repository.

This approach ensures:

- Reproducibility of experiments
- Clear tracking of dataset evolution
- Separation of code and data versioning

Both raw and processed datasets are versioned to maintain end-to-end data lineage.

8.2 Datasets Under Version Control

The following data layers are tracked using DVC:

Raw Data

`data/raw/`

This includes:

- Clickstream interaction data
- Product metadata ingested from the REST API

Processed Data

`data/processed/`

This includes:

- Cleaned and prepared interaction datasets
- EDA artifacts such as plots and summaries

Tracking both layers ensures that transformations can always be traced back to the original data source.

8.3 DVC Initialization

DVC is initialized at the repository level using:

```
dvc init
```

This creates the required DVC configuration files:

```
.dvc/
.dvcignore
```

These files are committed to Git to enable data versioning across the project.

8.4 Dataset Versioning Workflow

Raw and processed datasets are added to DVC tracking using:

```
dvc add data/raw
dvc add data/processed
```

This generates lightweight metadata files:

```
data/raw.dvc
data/processed.dvc
```

Only these `.dvc` files are committed to Git, while the actual data remains stored locally.

8.5 Repository Structure After Versioning

After enabling DVC, the repository structure includes:

```
DM4ML-Assignment-1/
├── data/
│   ├── raw/                 # Tracked by DVC
│   ├── processed/           # Tracked by DVC
│   ├── raw.dvc
│   └── processed.dvc
├── .dvc/
├── .dvcignore
├── src/
├── config/
└── reports/
```

This structure clearly separates code, configuration, and versioned data artifacts.

8.6 Data Lineage Tracking

Data lineage is established through:

- Hierarchical data layers (raw → processed → features)
- Timestamp-based ingestion directories
- DVC metadata files that record dataset hashes and versions
- Git versioning of transformation scripts

This provides a clear mapping between:

data source → transformation logic → resulting datasets

8.7 Reproducibility Workflow

A specific version of the dataset can be restored using:

```
git checkout <commit-hash>
dvc checkout
```

This restores the exact versions of raw and processed datasets associated with that commit, ensuring reproducible data pipelines and experiments.

8.8 Summary

The use of DVC provides a reliable mechanism for data versioning and lineage tracking across the data management pipeline. It ensures that all datasets used for feature engineering and model training are traceable, reproducible, and aligned with their corresponding transformation logic.

9. Model Training and Evaluation

9.1 Training and Evaluation Scripts

Model training, evaluation, and inference are implemented as **separate, modular scripts** to reflect production-grade machine learning workflows.

Training Script

📄 **Location:**

`src/models/train_model.py`

Responsibilities:

- Reads user–item interaction features from the PostgreSQL feature store
- Trains a collaborative filtering model using **Matrix Factorization (SVD)**
- Stores the trained model as a serialized artifact
- Logs training parameters and artifacts using MLflow

Evaluation Script

📄 **Location:**

`src/models/evaluate_model.py`

Responsibilities:

- Loads the trained model artifact
- Evaluates the model using ranking-based metrics:

    - Precision@K
    - Recall@K
- Logs evaluation metrics to MLflow for experiment tracking

Inference Script

📄 **Location:**

`src/models/inference.py`

Responsibilities:

- Loads the trained recommendation model
- Generates top-K product recommendations for a given user
- Demonstrates inference-time feature consumption

```
(.venv) PS D:\BITS\DM4ML\DM4ML-Assignment-1> python -m src.models.train_model
2026/01/20 13:57:44 INFO mlflow.store.db.utils: Creating initial MLflow database tables...
2026/01/20 13:57:44 INFO mlflow.store.db.utils: Updating database tables
2026/01/20 13:57:44 INFO alembic.runtime.migration: Context impl SQLiteImpl.
2026/01/20 13:57:44 INFO alembic.runtime.migration: Will assume non-transactional DDL.
2026/01/20 13:57:44 INFO alembic.runtime.migration: Context impl SQLiteImpl.
2026/01/20 13:57:44 INFO alembic.runtime.migration: Will assume non-transactional DDL.
Model trained and saved successfully
(.venv) PS D:\BITS\DM4ML\DM4ML-Assignment-1> python -m src.models.evaluate_model
2026/01/20 13:57:54 INFO mlflow.store.db.utils: Creating initial MLflow database tables...
2026/01/20 13:57:54 INFO mlflow.store.db.utils: Updating database tables
2026/01/20 13:57:54 INFO alembic.runtime.migration: Context impl SQLiteImpl.
2026/01/20 13:57:54 INFO alembic.runtime.migration: Will assume non-transactional DDL.
2026/01/20 13:57:54 INFO alembic.runtime.migration: Context impl SQLiteImpl.
2026/01/20 13:57:54 INFO alembic.runtime.migration: Will assume non-transactional DDL.
Precision@5: 0.1900
Recall@5: 0.9500
(.venv) PS D:\BITS\DM4ML\DM4ML-Assignment-1> python -m src.models.inference
Recommendations for U1:
[ 2  1  3  7 11]
(.venv) PS D:\BITS\DM4ML\DM4ML-Assignment-1>
```

9.2 Model Performance Report

Model performance is evaluated using **standard recommendation system metrics** that measure ranking quality.

Evaluation Metrics

- **Precision@5:** Measures the relevance of the top-5 recommended items
- **Recall@5:** Measures the coverage of relevant items in the top-5 recommendations

Performance Summary

A model performance report is generated based on the logged evaluation metrics and includes:

- Model type and configuration
- Training data source (PostgreSQL feature store)
- Precision@K and Recall@K scores
- Observations on model behavior and performance

These results are reproducible and traceable through MLflow experiment logs.

9.3 Tracked Model Metadata

All model-related metadata is tracked using **MLflow**, ensuring reproducibility and auditability.

Tracked Information

- **Run IDs:** Unique identifiers for each experiment run
- **Parameters:** Model type, latent dimensions, data source
- **Metrics:** Precision@5, Recall@5

- **Artifacts:** Serialized trained model file

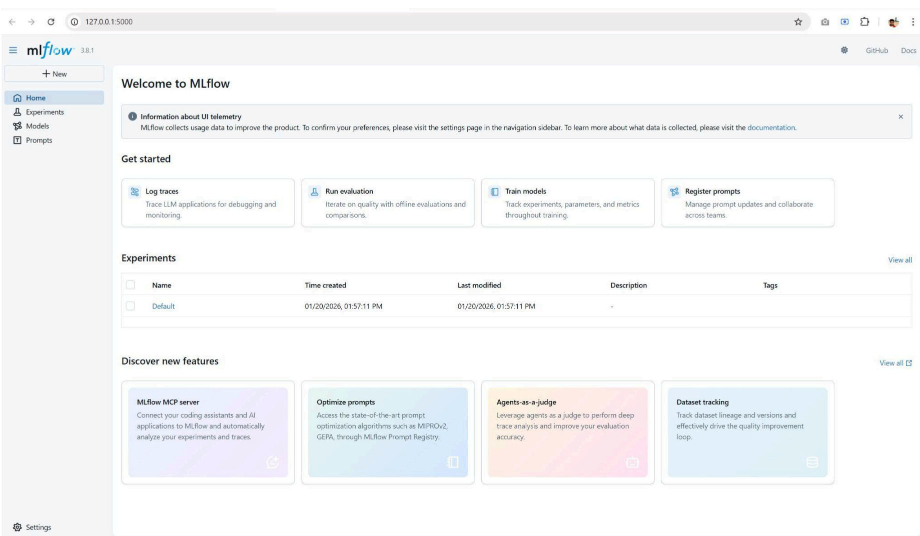
Model Artifact

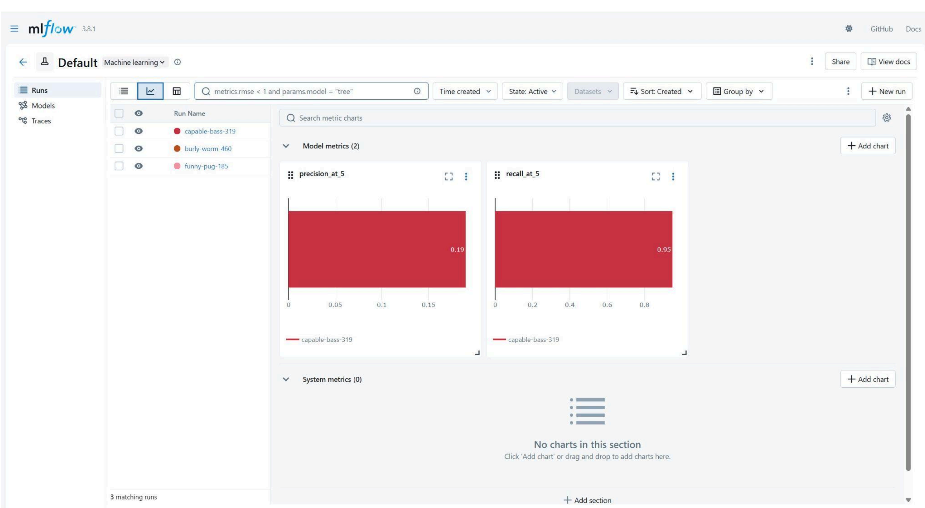
```
models/svd_model.pkl
```

Experiment Tracking

MLflow provides a centralized UI to compare runs and inspect model performance:

```
mlflow ui
```





9.4 Summary

The model training and evaluation stage implements a complete and reproducible workflow for recommendation systems. Training, evaluation, and inference are clearly separated, performance metrics are computed using standard measures, and all model metadata is tracked using MLflow. This ensures transparency, repeatability, and alignment with modern MLOps best practices.

10. Pipeline Orchestration

10.1 Orchestration Approach

The end-to-end data and machine learning pipeline is orchestrated using **Prefect**, a Python-native workflow orchestration framework that runs seamlessly on Windows and Linux environments.

Prefect is used to:

- Automate pipeline execution
- Define task dependencies
- Provide task-level logging and retries
- Monitor pipeline execution status

This orchestration layer ensures reliable and repeatable execution of the complete recommendation system pipeline.

10.2 Pipeline Workflow

The pipeline is executed as a single flow with the following sequence of stages:

```
Data Ingestion
  → Data Validation
    → Data Preparation and EDA
      → Feature Engineering
        → Feature Store Access
          → Model Training
            → Model Evaluation
```

Each stage corresponds to an independent task implemented in earlier pipeline steps.

10.3 Orchestration Implementation

The pipeline is implemented as a Prefect flow with multiple tasks, where each task invokes an existing processing script.

📄 Flow Definition

`src/orchestration/prefect_flow.py`

The flow coordinates execution of:

- Clickstream and product data ingestion
- Data validation checks
- Data preparation and exploratory analysis
- Feature engineering and storage in PostgreSQL
- Feature store access demonstration
- Model training and evaluation

10.4 Logging and Monitoring

Prefect provides built-in logging and monitoring capabilities:

- Each task logs execution start and completion status
- Failures are captured and retried automatically
- Downstream tasks are blocked on upstream task failures
- Execution status can be monitored via the Prefect UI or terminal logs

This enables effective observability and failure handling across the pipeline.

10.5 Execution and Verification

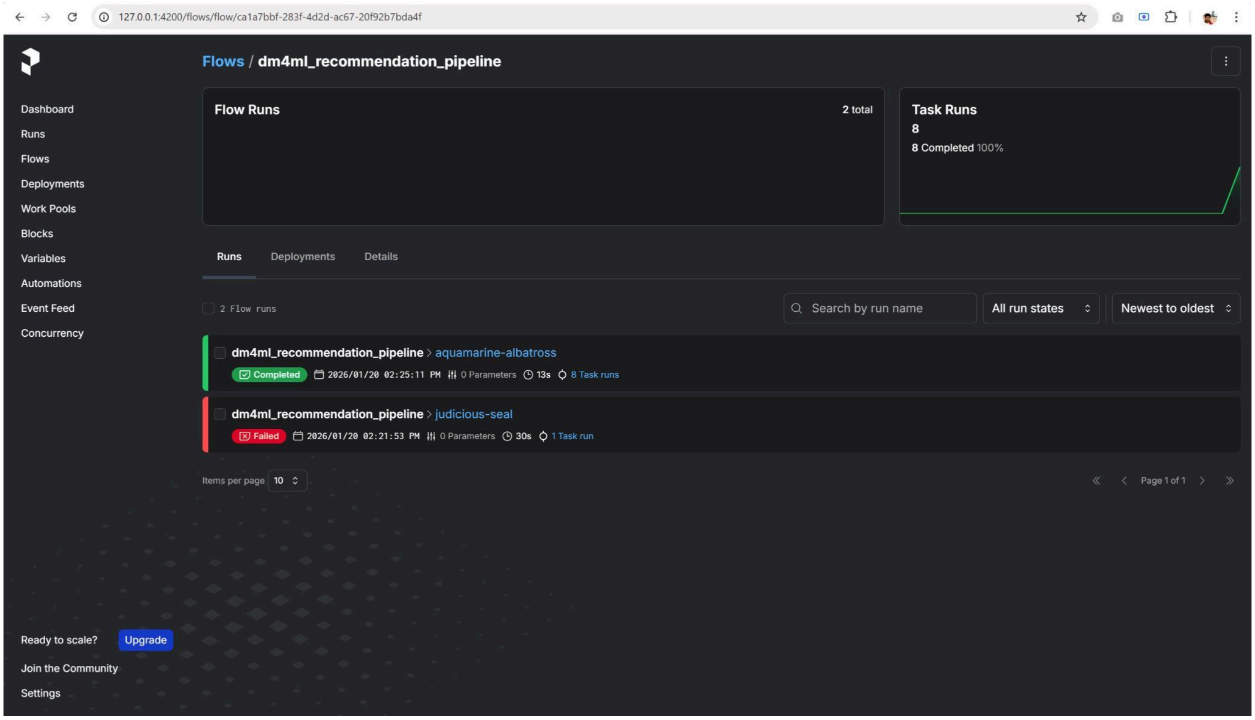
The pipeline is executed using:

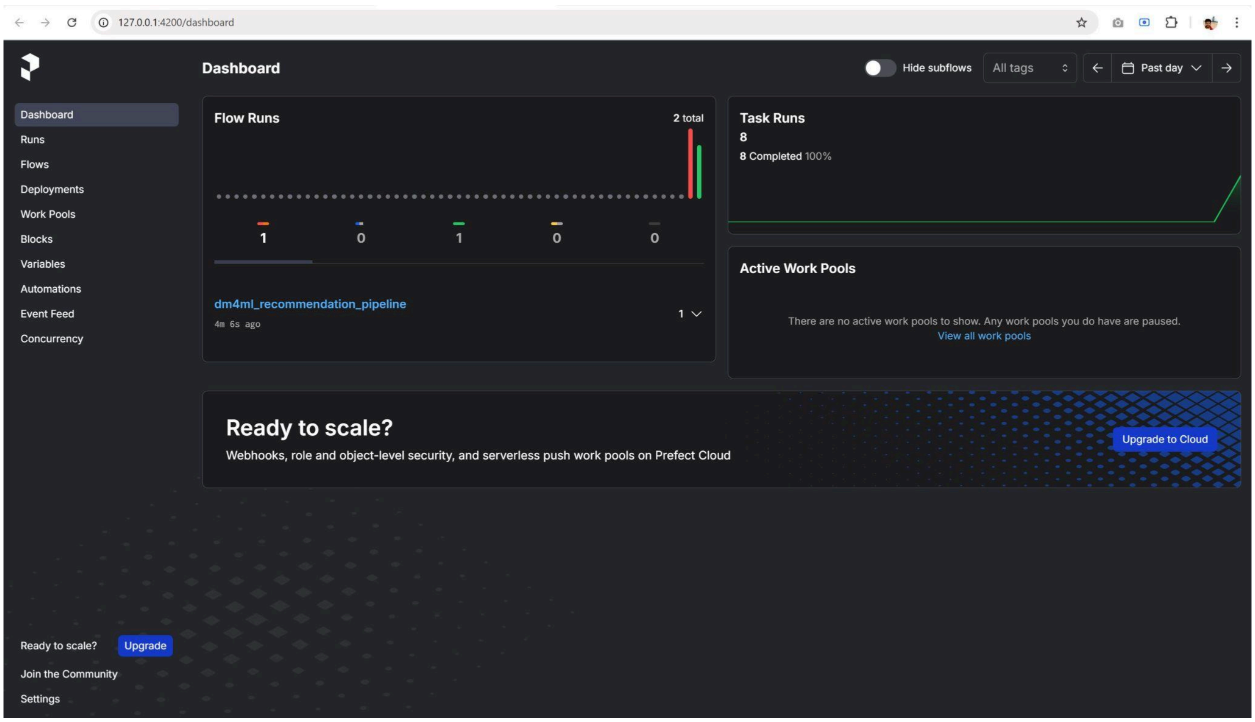
```
python -m src.orchestration.prefect_flow
```

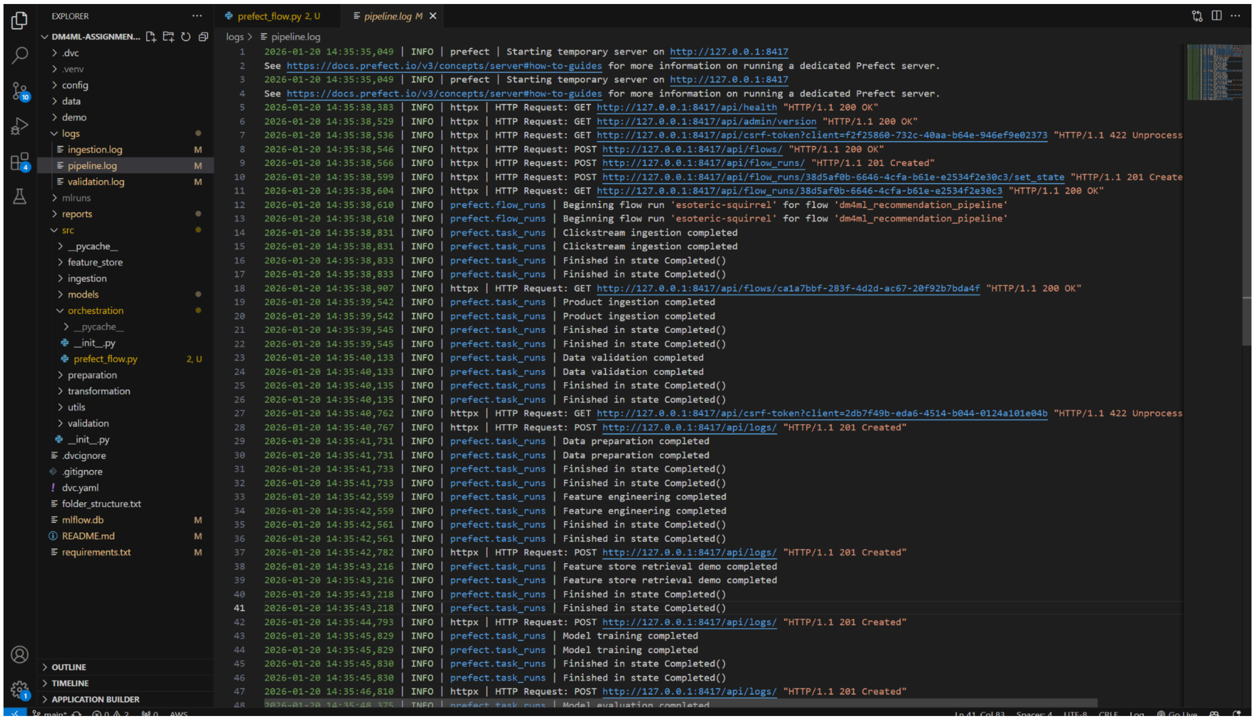
Successful execution is verified through:

- Console logs indicating completion of each task
- Prefect UI showing successful flow runs and task statuses

Screenshots or logs from successful executions are included as evidence of orchestration.


127.0.0.1:4200/flows/flow/ca1a7bbf-283f-4d2d-ac67-20f92b7bda4f
Flows / dm4ml_recommendation_pipeline
Dashboard
Runs
Flows
Deployments
Work Pools
Blocks
Variables
Automations
Event Feed
Concurrency
Flow Runs
2 total
Task Runs
8
8 Completed 100%
Runs
Deployments
Details
2 Flow runs
Search by run name
All run states
Newest to oldest
dm4ml_recommendation_pipeline > aquamarine-albatross
Completed
2026/01/20 02:25:11 PM
0 Parameters
13s
8 Task runs
dm4ml_recommendation_pipeline > judicious-seal
Failed
2026/01/20 02:21:53 PM
0 Parameters
30s
1 Task run
Items per page
10
Page 1 of 1
Ready to scale?
Upgrade
Join the Community
Settings


127.0.0.1:4200/dashboard
Dashboard
Hide subflows
All tags
Past day
Dashboard
Runs
Flows
Deployments
Work Pools
Blocks
Variables
Automations
Event Feed
Concurrency
Flow Runs
2 total
1
0
1
0
0
dm4ml_recommendation_pipeline
4m 6s ago
1
Task Runs
8
8 Completed 100%
Active Work Pools
There are no active work pools to show. Any work pools you do have are paused.
View all work pools
Ready to scale?
Webhooks, role and object-level security, and serverless push work pools on Prefect Cloud
Upgrade to Cloud
Ready to scale?
Upgrade
Join the Community
Settings



10.6 Summary

The Prefect-based orchestration layer automates the entire data and machine learning workflow, ensuring structured execution, observability, and fault tolerance. This completes the end-to-end pipeline implementation for the recommendation system.