GALATASARAY UNIVERSITY

Faculty of Engineering and Technology

Department of Computer Engineering

INF438
ADVANCED DATABASES

Project Final

E-Sport Analysis (Dota 2) on
Architecture Lakehouse Azure

GROUP 3

Member 1 : Sabri Taner Burak ALTINDAL – 22401030

Member 2 : Emirhan Karatepe – 19401830

Member 3 : Kaan Çolakoğlu – 21401946

Member 4: Ceren Akbaş – 22401028

Academic Year 2025–2026

Contents

1 Introduction **5**
1.1 Project Context 5
1.2 Dataset Used 5
1.3 Project Objectives 5
1.4 Azure Services Used 6

2 Architectural Design **6**
2.1 Medallion Architecture 7
2.2 Azure Data Lake Storage Gen2 Structure 8
2.3 Data Flow Diagram 8

3 Implementation **8**
3.1 Batch Processing Pipeline 9
3.2 Trigger Configuration 10
3.3 Databricks Notebooks 11
3.4 Streaming Simulation 11
3.5 ETL Transformations 12
3.6 Delta Lake Table Management 14

4 Data Analysis and Machine Learning **14**
4.1 Accessing Gold Tables and SQL Execution (Databricks) 15
4.2 Analytical SQL Queries (Results) 16
4.3 Power BI Visualization 21
4.4 Machine Learning Model 25
4.5 Performance Evaluation 27
4.6 Feature Importance Analysis 29
4.7 Experiment Tracking with MLflow 30

5 Challenges Encountered and Solutions **31**
5.1 Azure Resource Limitations 32
5.2 File Collisions in Streaming Simulation 32
5.3 Data Volume and Cost Constraints 32
5.4 Azure Storage Key Management 32
5.5 Access Management Complexity (IAM) 32
5.6 Databricks Cluster Access Issues 33

6 Conclusion **33**
6.1 Project Summary 34
6.2 Lessons Learned 34
6.3 General Conclusion 34

A APPENDICES **35**

Appendices **35**
A.1 Databricks Notebooks Source Code 35
A.2 Streaming Simulator Source Code 37
A.3 Table Creation SQL Commands 38
A.4 ML Model Source Code (with Hyperparameter Tuning) 39

A.5 Technologies Used 41

List of Figures

1 Azure Portal view of Bronze/Silver/Gold folder structure in ADLS Gen2 . 8
2 Data flow diagram 9
3 Execution details of the *Copy Data* activity confirming successful ingestion of 4 source files (212 MB) to the Bronze container. 9
4 pl_master_esports pipeline diagram in Azure Data Factory 10
5 "Succeeded" status and "Triggered by tr_daily_esports" in the ADF Monitor screen 11
6 stream_simulator.py execution output 12
7 Tier classification output in silver_to_gold notebook 13
8 List of tables under esports_gold schema in Databricks Catalog 14
9 Loading Gold tables from ADLS Gen2 and creating temporary views in Databricks 16
10 SQL Result: Top 10 players by average KDA (with kills, deaths, assists) . 17
11 SQL Result: daily statistics overview (match_count and average duration in minutes) 18
12 SQL Result: Top 10 heroes by average GPM and XPM 19
13 SQL Result: Top 10 combat-oriented heroes (kills/assists) and number of games played 20
14 SQL Result: Top 10 most popular heroes by times_played (with wins and win_rate) 21
15 Power BI — Overview Page: temporal filter, daily trend of match count, and global KPIs 22
16 Power BI — Top 10 Players Page: average KDA vs win rate comparison and detail table 23
17 Power BI — Hero Meta Page: presence rate vs win rate relationship and Top 10 most present heroes 24
18 Power BI — Duration and Correlations Page: average duration trend and duration vs kills correlation 25
19 Model results and parameters in MLflow interface 30
20 Model results and parameters in MLflow interface 31

List of Tables

1 Dataset characteristics 5
2 Microsoft Azure services used 6
3 Silver layer tables 7
4 Gold layer tables 7
5 Trigger configuration 10
6 Databricks cluster configuration 11
7 Meta Tier classification criteria 13
8 Duration analysis results 14
9 Explanatory variables used for modeling 26
10 Hyperparameter search space 26
11 Comparative performance of tested configurations (CV Accuracy) 27
12 Configuration selected for the final model 27
13 Model confusion matrix on test set ($n = 1674$) 28
14 Variable importance in the optimized model (MDI) 29
15 Main results obtained 34
16 Project technology stack 41

1. Introduction

1.1 Project Context

The e-sports industry has experienced exponential growth over the past decade, becoming a multi-billion dollar ecosystem today. With this growth, the amount of data generated during professional tournaments has also increased significantly. Player performance analysis, team strategy optimization, and match outcome prediction are critical areas where data engineering and data science find practical applications.

As part of this project, a complete data engineering solution was designed and implemented for processing, analyzing, and developing predictive models on professional Dota 2 match data. The project was carried out on the Microsoft Azure platform, adopting modern cloud architecture principles.

1.2 Dataset Used

The dataset used in this project is the **"Dota 2 Pro League Matches 2023"** published on the Kaggle platform. This dataset contains detailed records of professional Dota 2 tournament matches throughout 2023.

Table 1: Dataset characteristics

File Name	Size	Description
main_metadata.csv	9.09 MB	Match metadata
players_reduced.csv	152.00 MB	Player statistics
picks_bans.csv	33.01 MB	Pick/ban data
teams.csv	8.09 MB	Team information

The CSV format and relational structure of the dataset required a complete cleaning and merging process during the transition from the Bronze layer to the Silver layer.

1.3 Project Objectives

The main objectives of this project were defined as follows:

1. **Lakehouse Architecture Setup:** Implementation of the Medallion Architecture (Bronze-Silver-Gold) on Azure Data Lake Storage Gen2
2. **Batch Processing Pipeline:** Creation of automated ETL pipelines using Azure Data Factory and Databricks
3. **Streaming Simulation:** Development of a simulator mimicking real-time data flow
4. **Data Transformation:** Transformation of raw data into analytical tables using PySpark
5. **Advanced Analysis:** Player and hero performance analyses with SQL and PySpark

6. **Machine Learning:** Development of a match outcome prediction model and tracking with MLflow

1.4 Azure Services Used

Table 2: Microsoft Azure services used

Service	Usage Purpose
Azure Data Lake Storage Gen2	Central storage for Lakehouse layers
Azure Data Factory	Pipeline orchestration and scheduling
Azure Databricks	Data processing and ML development
Delta Lake	High-performance ACID-compliant data format
MLflow	Machine learning experiment tracking
Power BI	Visualization and dashboard creation

All resources were created using the **Microsoft Azure for Students** subscription and managed with cost optimization in mind.

2. Architectural Design

2.1 Medallion Architecture

In this project, the **Medallion Architecture**, widely adopted in modern data engineering practices, was applied. This architecture divides data into three layers according to their level of quality and processing: **Bronze**, **Silver**, and **Gold**.

2.1.1 Bronze Layer (Raw Data)

The Bronze layer is where data from source systems is stored as-is, without any transformation:

- **Purpose:** Preserving an exact copy of the data source
- **Format:** CSV and JSON (for streaming simulation)
- **Content:** 4 main CSV files + streaming JSON files
- **Usage:** Return to source in case of error, audit and traceability

2.1.2 Silver Layer (Cleaned Data)

The Silver layer is the intermediate layer where raw data is cleaned and standardized:

- **Purpose:** Provide clean and consistent data
- **Format:** Delta Lake (Parquet-based)
- **Transformations:** Null value cleaning, type standardization, duplicate elimination, outlier detection

Table 3: Silver layer tables

Table	Number of Records
cleaned_matches	29 809
cleaned_players	8 456
cleaned_picks_bans	710 414

2.1.3 Gold Layer (Analytical Data)

The Gold layer contains data enriched with business logic:

Table 4: Gold layer tables

Table	Description	Records
esports_gold.player_stats	Performance metrics	813
esports_gold.hero_stats	Hero statistics	123
esports_gold.daily_stats	Daily statistics	365
esports_gold.ml_features	ML features	8 312

2.2 Azure Data Lake Storage Gen2 Structure

All Lakehouse layers are positioned on Azure Data Lake Storage Gen2:

- **Storage Account:** `dota2lakehousenew`
- **Container:** `data`
- **Paths:**
  - Bronze: `abfss://data@dota2lakehousenew.../bronze/`
  - Silver: `abfss://data@dota2lakehousenew.../silver/`
  - Gold: `abfss://data@dota2lakehousenew.../gold/`

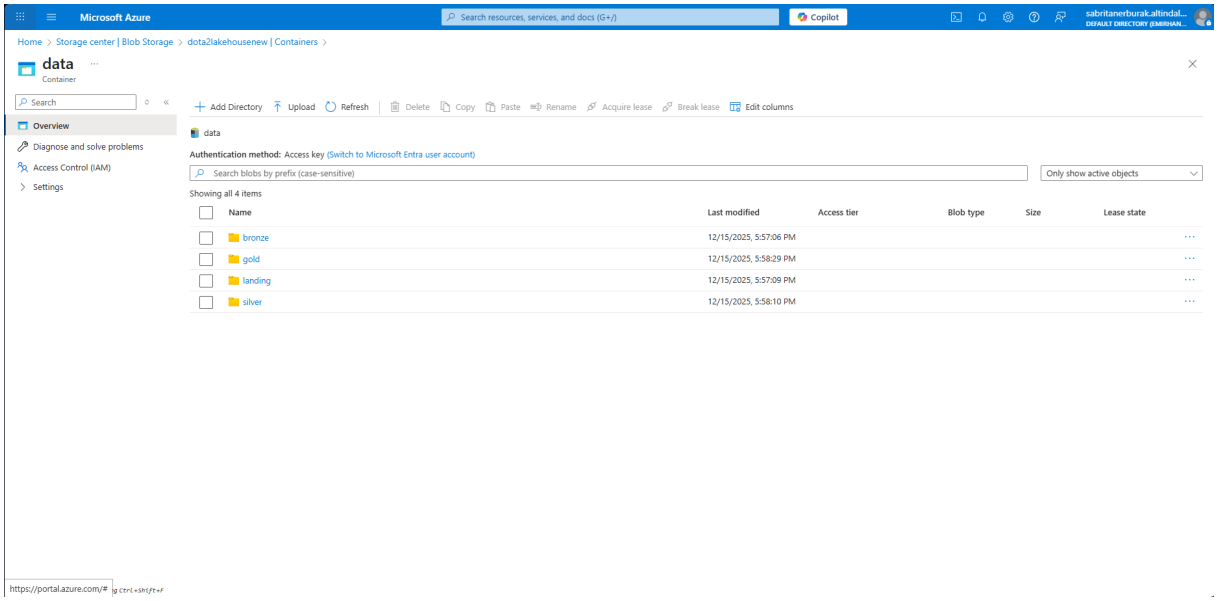

Figure 1: Azure Portal view of Bronze/Silver/Gold folder structure in ADLS Gen2

2.3 Data Flow Diagram

The end-to-end data flow of the system follows this path: source data from Kaggle and the streaming simulator is first stored in the Bronze layer. Then, Azure Data Factory orchestrates the Databricks notebooks that transform data to the Silver and then Gold layers. Finally, Gold data feeds SQL analytics, the machine learning model (tracked by MLflow), and Power BI visualizations.

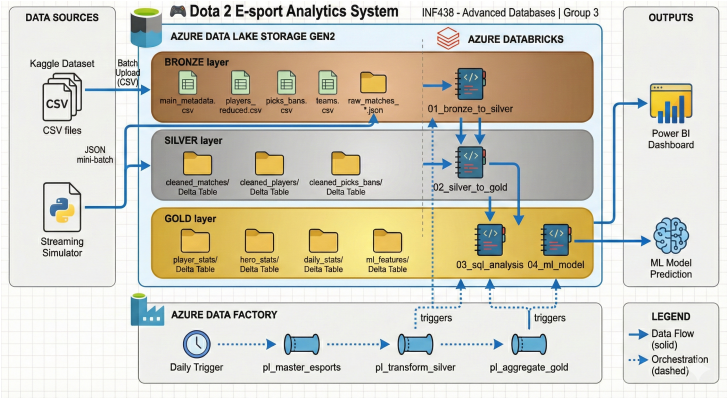

Figure 2: Data flow diagram

3. Implementation

3.1 Batch Processing Pipeline

Azure Data Factory (ADF) was used for batch processing pipeline orchestration. The main pipeline created in the project is named **pl_master_esports**.

Upstream of this processing, a specific ingestion pipeline named `PL_Ingest_Kaggle_To_Bronze` was set up. This pipeline uses a *Copy Data* activity to automatically transfer raw files (CSV) and the reduced dataset to the `bronze/` container of the Data Lake, ensuring data availability for Databricks notebooks.

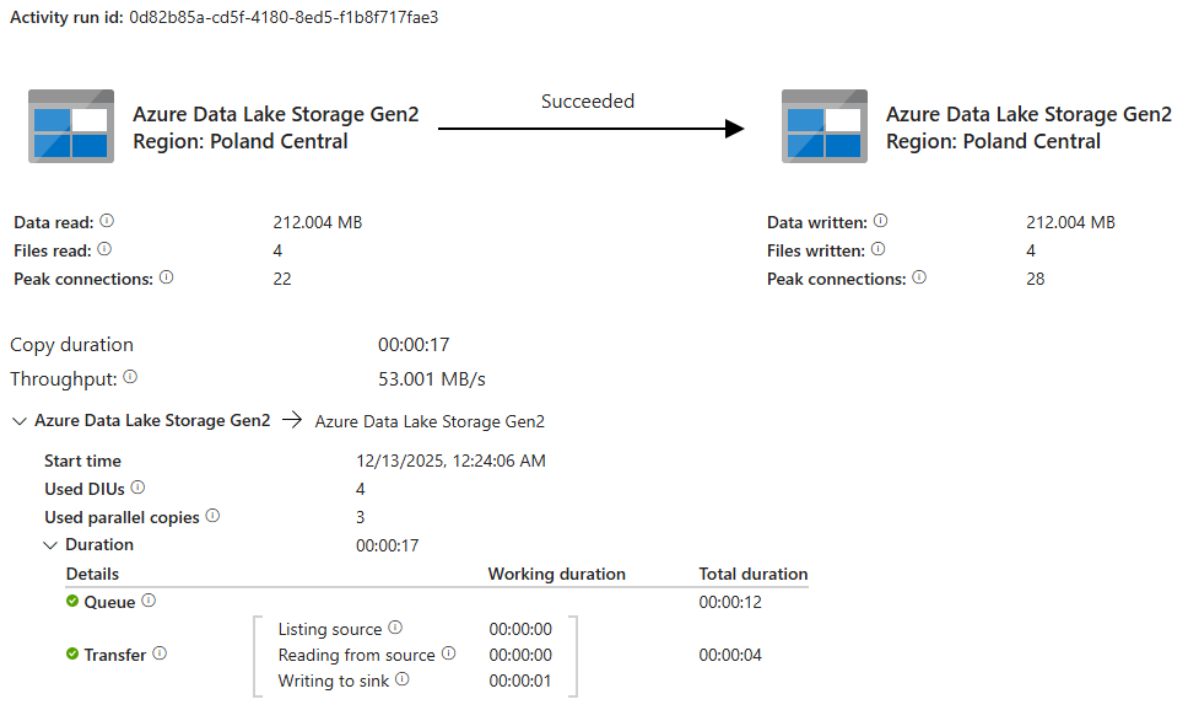

Figure 3: Execution details of the *Copy Data* activity confirming successful ingestion of 4 source files (212 MB) to the Bronze container.

The pipeline structure is designed as follows:

- **Activity 1:** nb_bronze_to_silver (Databricks Notebook)

- **Activity 2:** nb_silver_to_gold (depends on Activity 1)
- **Activity 3:** nb_ml_model (depends on Activity 2)

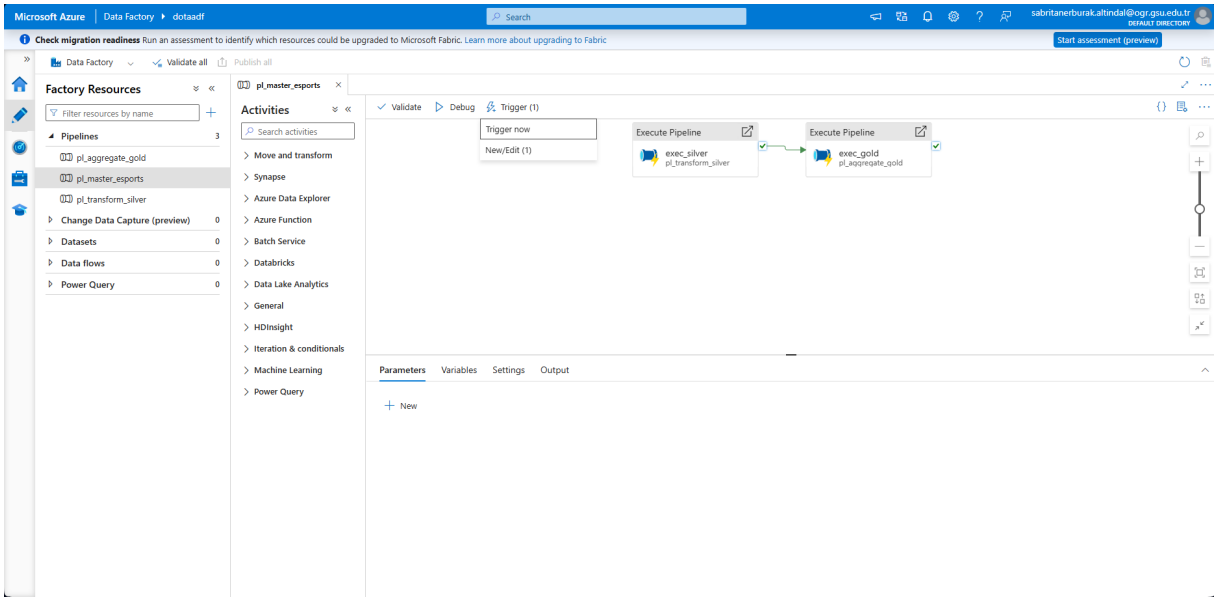

Figure 4: pl_master_esports pipeline diagram in Azure Data Factory

3.2 Trigger Configuration

A trigger named **tr_daily_esports** was created for automatic pipeline execution.

Table 5: Trigger configuration

Parameter	Value
Trigger Name	tr_daily_esports
Type	Schedule Trigger
Frequency	Daily
Start Time	02:00 UTC
Status	Active

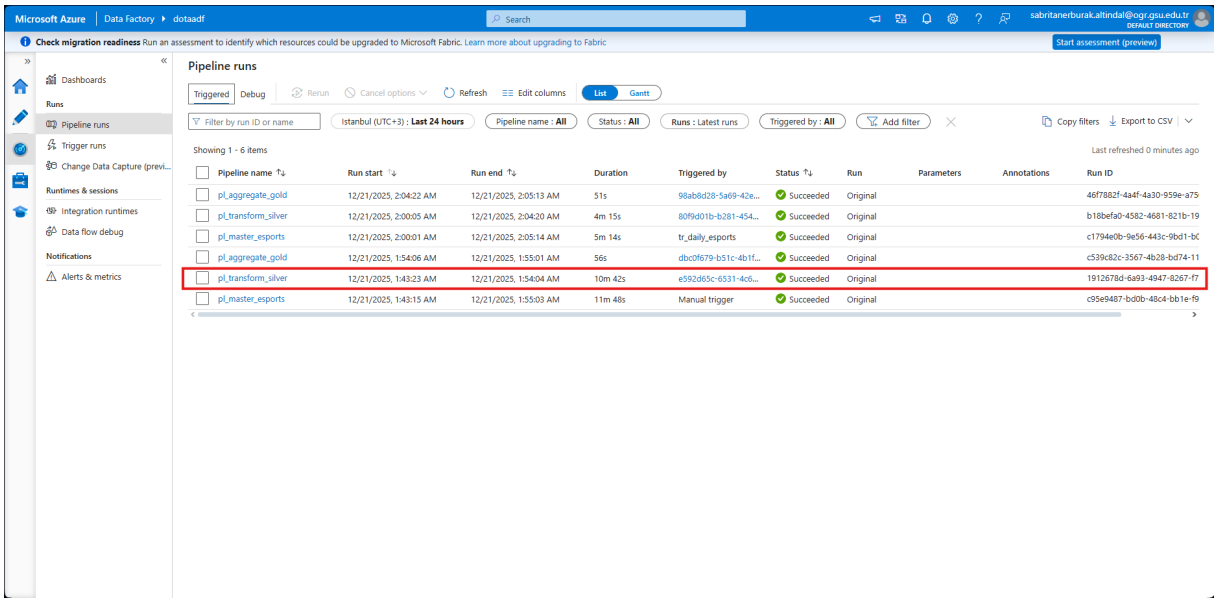

Figure 5: "Succeeded" status and "Triggered by tr_daily_esports" in the ADF Monitor screen

3.3 Databricks Notebooks

Data transformation operations were performed using PySpark on Azure Databricks.

Table 6: Databricks cluster configuration

Parameter	Value
Cluster Mode	Standard
Node Type	Standard_DS3_v2
Auto Termination	120 minutes
Databricks Runtime	13.3 LTS (Spark 3.4.1)

Notebooks created:

1. **setup_connection.ipynb**: Azure ADLS Gen2 connection test
2. **01_bronze_to_silver.ipynb**: Raw data cleaning
3. **02_silver_to_gold.ipynb**: Business logic application
4. **04_ml_model.ipynb**: ML model training

The complete source code of these notebooks is available in **Appendix A.1**.

3.4 Streaming Simulation

In the absence of a real-time data source, a Python script simulating streaming behavior was developed. The **stream_simulator.py** script operates according to the following logic:

1. Records are read from the main_metadata.csv file
2. Records are divided into mini-batches of 5
3. An artificial delay of 3 seconds is applied for each mini-batch
4. The mini-batch is named with a unique timestamp and written to Bronze

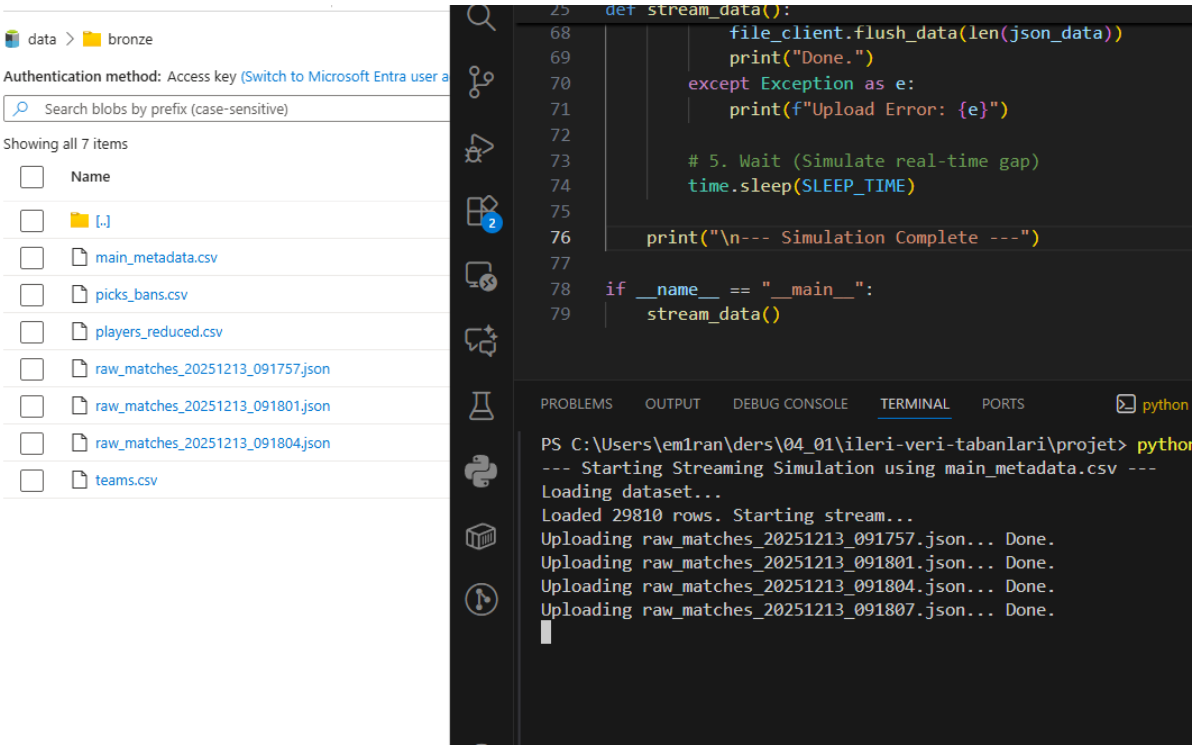

Figure 6: stream_simulator.py execution output

When the simulator is executed, files are created in the Bronze layer with the format `raw_matches_YYYYMMDD_HHMMSS.json` (10 files in total).

The complete source code of the simulator is available in **Appendix A.2**.

3.5 ETL Transformations

3.5.1 Bronze → Silver Transformation

The raw CSV data from the Bronze layer underwent a complete transformation process:

- **Loading:** Reading CSV files with schema inference
- **Cleaning:** Removing unnecessary columns, filtering null values
- **Validation:** Filtering matches with invalid duration (<5 min or >2 hours)
- **Enrichment:** Creating new features (duration_minutes, kda_calculated)
- **Outlier Detection:** Marking using the 3-sigma rule

3.5.2 Silver → Gold Transformation

The cleaned data received business logic and advanced calculations.

Player Statistics (player_stats): Aggregation by player account with calculation of averages (KDA, GPM, XPM) and win rate. Result: statistics calculated for **813 unique players**.

Hero Meta Analysis (hero_stats): A classification algorithm for heroes into 5 tiers according to their presence rate (see Table 7).

Table 7: Meta Tier classification criteria

Tier	Presence Rate Criterion
S-Tier	$> 50\%$
A-Tier	$> 30\%$
B-Tier	$> 15\%$
C-Tier	$> 5\%$
D-Tier	$\leq 5\%$

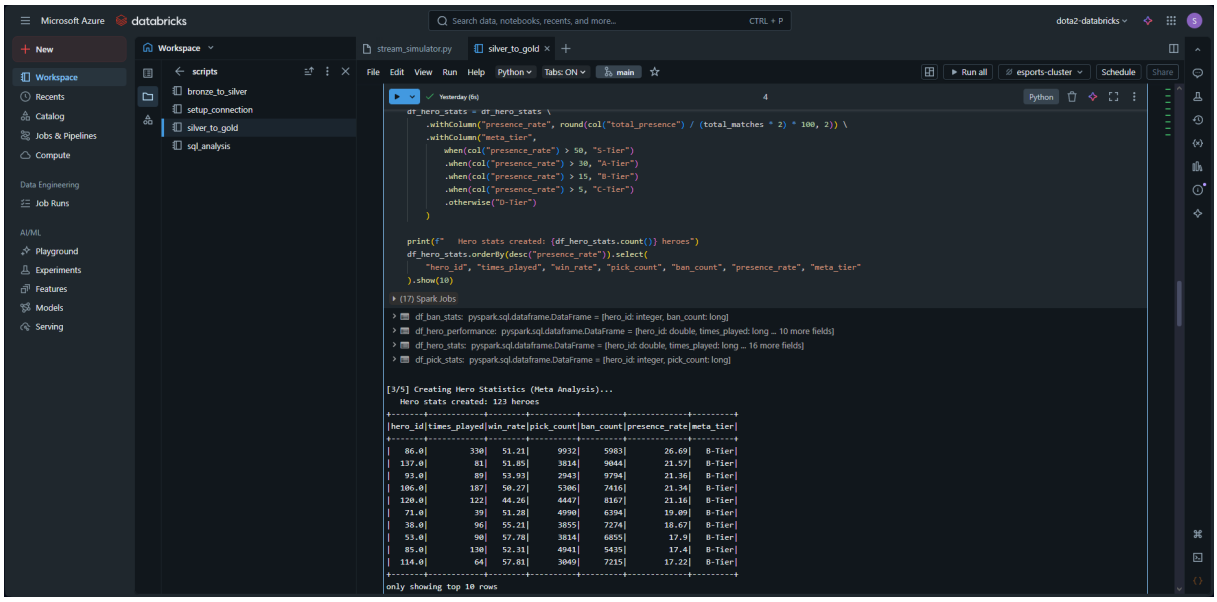

Figure 7: Tier classification output in silver_to_gold notebook

Match Duration Analysis:

Table 8: Duration analysis results

Duration	Matches	Avg Kills	Radiant Win
Very Short (<25min)	4 176	44.94	53.54%
Short (25-35min)	13 548	56.74	50.41%
Medium (35-45min)	8 162	65.33	49.36%
Long (45-55min)	2 839	68.19	49.52%
Very Long (>55min)	1 084	80.77	50.46%

3.6 Delta Lake Table Management

The Gold layer tables are managed under the **esports_gold** schema in the Databricks Unity Catalog. The SQL creation commands are available in **Appendix A.3**.

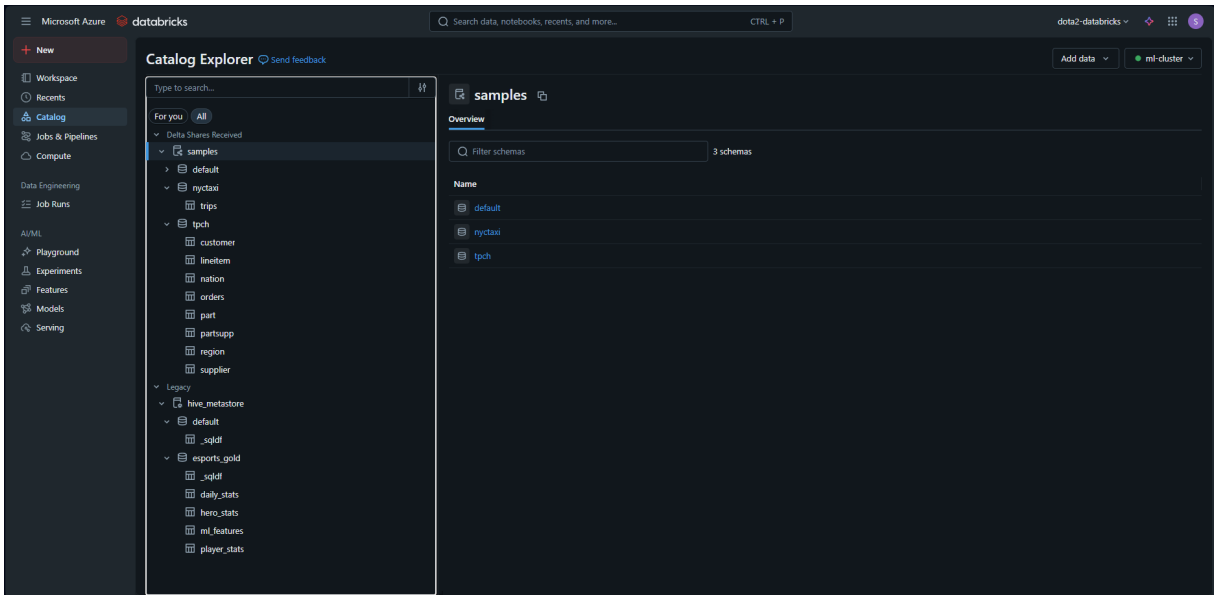

Figure 8: List of tables under esports_gold schema in Databricks Catalog

4. Data Analysis and Machine Learning

4.1 Accessing Gold Tables and SQL Execution (Databricks)

4.1.1 Connection to Azure Data Lake Storage (ADLS Gen2)

The analytical tables of the *Gold* layer are stored on Azure Data Lake Storage Gen2 in Delta Lake format. In our case, certain limitations of the *Azure for Students* subscription (quotas / RBAC permissions) prevented direct and stable access for all group members. To ensure project continuity, access to the *Storage Account* was done via an access key provided by a group member with the necessary permissions. This key is never exposed in the report or in plain text in versioned code; it is injected via Databricks environment variables.

CPU constraints and use of a shared cluster. During notebook execution with the Azure for Students subscription, we encountered resource limitations (CPU/vCore quota) and slowdowns that could interrupt certain Spark processing tasks. To avoid deadlocks and complete the work on time, we used the Databricks cluster already configured by a group member (esports-cluster), which had more stable resources (16 GB RAM and 4 cores). Thanks to this solution, we were able to correctly execute ETL transformations, access Delta tables in the Gold layer, and run analytical SQL queries without interruptions.

4.1.2 Loading Gold Tables and Creating Temporary Views

To execute SQL queries directly on Delta files in the Gold layer, we loaded each table from ADLS and created temporary views in Databricks.

Figure 9: Loading Gold tables from ADLS Gen2 and creating temporary views in Databricks

4.2 Analytical SQL Queries (Results)

This section presents the main SQL queries used to produce analyses on players, heroes, and daily activity. To keep the report readable, we present here the outputs (screenshots); complete queries can be added in the appendix if needed.

4.2.1 Top 10 Players by Performance (KDA)

Objective: identify the 10 most performing players according to their average KDA (with filtering to avoid non-representative cases).

```
%sql
SELECT
    account_id,
    total_kills,
    total_deaths,
    total_assists,
    ROUND(avg_kda, 2) as kda_ratio
FROM
    gold_player_stats
WHERE
    total_matches > 5
ORDER BY
    avg_kda DESC
LIMIT 10;
```

(1) Spark Jobs

_sqldf: pyspark.sql.dataframe.DataFrame = [account_id: double, total_kills: long ... 3 more fields]

Table +

	1.2 account_id	1^2_3 total_kills	1^2_3 total_deaths	1^2_3 total_assists	1.2 kda_ratio
1	202217968	49	16	49	12.27
2	116249155	68	44	193	11.97
3	173978074	70	25	99	11.79
4	320252024	68	12	62	11.76
5	132309493	40	12	48	11.75
6	392565237	53	22	100	11.67
7	221532774	60	24	127	11.18
8	118233883	52	24	78	10.95
9	361688848	177	58	180	10.77
10	220154695	54	22	57	10.63

Figure 10: SQL Result: Top 10 players by average KDA (with kills, deaths, assists)

4.2.2 Daily Trends: Match Volume and Average Duration

Objective: observe the daily evolution of the number of matches as well as the average duration (conversion to minutes).

```
%sql
SELECT
    match_date,
    match_count,
    ROUND(avg_duration / 60, 2) as avg_duration_minutes
FROM
    gold_daily_stats
ORDER BY
    match_date DESC
LIMIT 10;
```

▸ (1) Spark Jobs

› _sqldf: pyspark.sql.dataframe.DataFrame = [match_date: date, match_count: long ... 1 more fiel

Table +

	match_date	match_count	avg_duration_minutes
1	2023-12-31	45	0.58
2	2023-12-30	49	0.57
3	2023-12-29	70	0.55
4	2023-12-28	74	0.54
5	2023-12-27	73	0.55
6	2023-12-26	71	0.58
7	2023-12-25	56	0.56
8	2023-12-24	67	0.54
9	2023-12-23	75	0.56
10	2023-12-22	60	0.58

Figure 11: SQL Result: daily statistics overview (match_count and average duration in minutes)

4.2.3 Top 10 Heroes: Economic Performance (GPM/XPM)

Objective: identify heroes with the best economic performance via `avg_gpm` and `avg_xpm`.

```
SELECT
    hero_id,
    ROUND(avg_gpm, 0) as avg_gpm,
    ROUND(avg_xpm, 0) as avg_xpm
FROM
    gold_hero_stats
ORDER BY
    avg_gpm DESC
LIMIT 10;
```

▸ (1) Spark Jobs

▸ _sqldf: pyspark.sql.dataframe.DataFrame = [hero_id: double, avg_g

Table +

	1.2 hero_id	1.2 avg_gpm	1.2 avg_xpm
1	109	701	710
2	89	674	640
3	53	666	707
4	25	655	783
5	46	653	803
6	95	649	751
7	94	643	719
8	12	641	731
9	69	640	691
10	113	637	682

Figure 12: SQL Result: Top 10 heroes by average GPM and XPM

4.2.4 Top 10 Heroes: Combat Style (Kills/Assists)

Objective: analyze the most aggressive heroes via the `avg_kills` and `avg_assists` metrics, taking into account the volume (`times_played`).

```
%sql
SELECT
    hero_id,
    ROUND(avg_kills, 2) as avg_kills,
    ROUND(avg_assists, 2) as avg_assists,
    times_played
FROM
    gold_hero_stats
ORDER BY
    avg_kills DESC
LIMIT 10;
```

▸ (1) Spark Jobs

› _sqldf: pyspark.sql.dataframe.DataFrame = [hero_id: double, avg_kills: double ... 2 more fie

Table +

	1.2 hero_id	1.2 avg_kills	1.2 avg_assists	1²3 times_played
1	39	10.28	10.89	72
2	25	9.68	7.3	191
3	4	9.27	10.5	52
4	93	8.79	9.78	89
5	13	8.77	10.65	60
6	70	8.74	5.86	74
7	106	8.62	11.87	187
8	22	8.59	13.55	74
9	52	8.59	9.32	109
10	67	8.54	13.98	46

Figure 13: SQL Result: Top 10 combat-oriented heroes (kills/assists) and number of games played

4.2.5 Top 10 Most Popular Heroes

Objective: identify the most played heroes based on the number of appearances (times played). We also display the number of wins and win rate (in %).

```
%sql
SELECT
    hero_id,
    times_played as total_picks,
    wins as total_wins,
    ROUND(win_rate * 100, 2) as win_rate_percentage
FROM
    gold_hero_stats
ORDER BY
    times_played DESC
LIMIT 10;
```

▸ (1) Spark Jobs

› _sqldf: pyspark.sql.dataframe.DataFrame = [hero_id: double, total_picks: long ... 2 more fields]

Table +

	1.2 hero_id	1²3 total_picks	1²3 total_wins	1.2 win_rate_percentage
1	86	330	169	5121
2	100	299	151	5050
3	19	255	117	4588
4	9	248	121	4879
5	128	225	101	4489
6	83	210	114	5429
7	5	202	109	5396
8	25	191	98	5131
9	106	187	94	5027
10	51	179	94	5251

Figure 14: SQL Result: Top 10 most popular heroes by times_played (with wins and win_rate)

The analysis reveals that in very short matches (<25 min), the Radiant team has a slight advantage (53.54%). This rate balances around 50% for longer matches.

4.3 Power BI Visualization

4.3.1 Connection and Data Loading in Power BI

We performed the visualization part with Power BI Desktop using Gold data. To retrieve the data, we used *Get Data* then the *Azure Blob Storage* source. Since not everyone could connect easily (rights/quotas Azure for Students), we used an access key shared by a group member.

After connecting, we found the Parquet files in the *data* container (Gold layer) and imported the tables *daily_stats*, *player_stats*, *hero_stats*, and *ml_features*. Then, we

added a *Date* table to filter by time, and linked it to *daily_stats* with the *match_date* column. Finally, we created the pages and charts to analyze overall trends, top players, and hero meta.

4.3.2 Page 1: Overview and Daily Trends

This page provides a quick reading of 2023 activity: (i) a date slicer that controls all indicators, (ii) a daily trend of the number of matches, (iii) KPI cards: total number of matches, average Radiant win rate, and average match duration.

Observations. We observe relatively stable activity over a large part of the year, with daily fluctuations. A marked peak appears during a specific period (end of year), which may correspond to an event (tournament, major patch, or high competitive activity). The average Radiant win rate remains close to balance, suggesting no lasting systematic advantage over the year. The average match duration is generally stable, with occasional variations indicating changes in play style or meta.

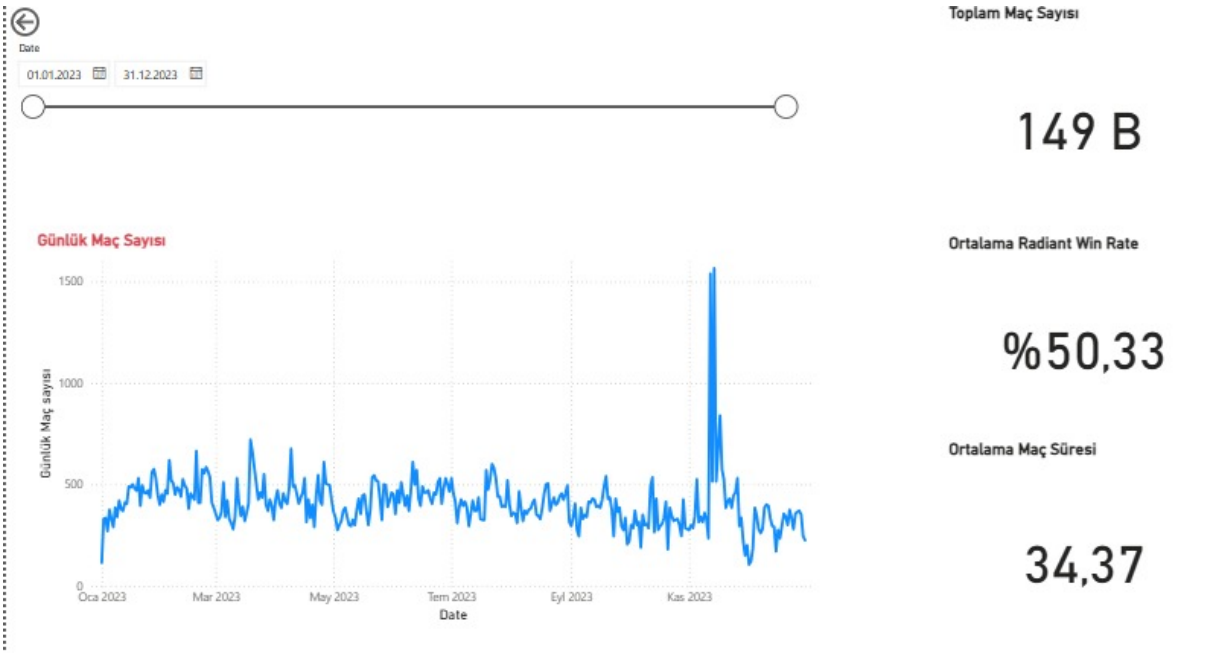

Figure 15: Power BI — Overview Page: temporal filter, daily trend of match count, and global KPIs

4.3.3 Page 2: Top 10 Players

This page compares players according to: (i) Top 10 players by average KDA, (ii) Top 10 players by win rate, with a detailed table of statistics (matches, wins, KDA, XPM, etc.).

Observations. The KDA ranking highlights profiles oriented towards individual performance (survival + combat efficiency). The win rate ranking is not necessarily identical: a player may have a high KDA but a more moderate win rate, highlighting the importance of teamwork and match context. The table allows verifying result consistency by simultaneously comparing match volume and indicators, and avoiding over-interpreting players with too few games.

account_id	Toplam total_matches	Toplam total_wins	Toplam avg_kda	Toplam win_rate	Toplam avg_xpm
124.801.257,00	2	2	27,00	100,00	841,00
40.805.086,00	1	1	24,00	100,00	812,00
155.162.307,00	2	2	22,50	100,00	994,00
238.239.590,00	4	4	21,06	100,00	671,75
292.921.272,00	2	2	20,00	100,00	748,00
25.907.144,00	2	2	19,75	100,00	531,00
201.358.612,00	3	3	18,56	100,00	795,67
480.412.663,00	2	2	18,50	100,00	926,00
180.775.030,00	2	2	18,38	100,00	699,00
86.745.912,00	2	2	18,34	100,00	798,00
Toplam	**8312**	**4129**	**4.216,22**	**38.332,00**	**457.202,00**

Figure 16: Power BI — Top 10 Players Page: average KDA vs win rate comparison and detail table

4.3.4 Page 3: Hero Meta Analysis

This page is dedicated to the meta: (i) a scatter plot linking *presence_rate* and *win_rate* by hero, (ii) a Top 10 of the most present heroes.

Observations. The scatter plot highlights several interesting zones:

- very present heroes but with an average win rate: often "meta" or flexible picks, not necessarily dominant;
- less present heroes but with a high win rate: "underrated" candidates or very effective situational picks;
- present and performing heroes: they constitute the dominant meta and deserve special attention.

The Top 10 of the most present heroes synthesizes popularity and facilitates identification of heroes played most frequently over the filtered period.

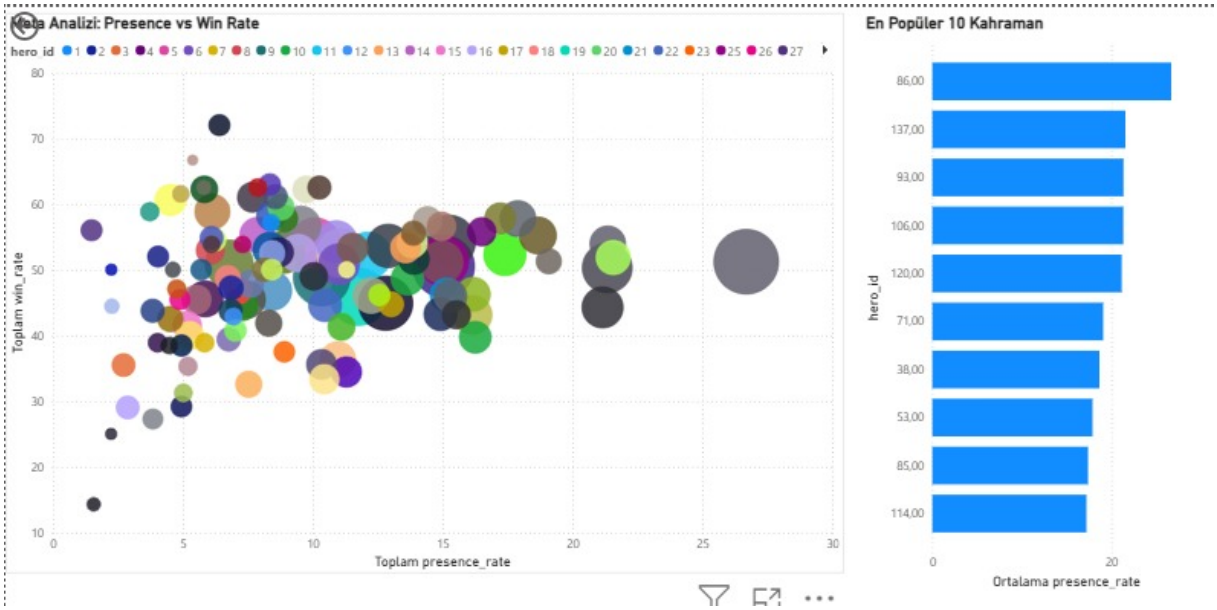

Figure 17: Power BI — Hero Meta Page: presence rate vs win rate relationship and Top 10 most present heroes

4.3.5 Page 4: Match Duration and Correlations

This page analyzes: (i) the temporal evolution of average match duration, (ii) the relationship between duration and intensity (e.g., average kills) via a scatter plot.

Observations. The average duration varies within a relatively limited range, but certain lows/peaks suggest periods where matches end more quickly (aggressive strategies) or conversely lengthen (more controlled games). The duration vs kills scatter plot shows a globally weak to moderate relationship: longer matches tend to offer more combat opportunities, but dispersion indicates that other factors strongly influence the number of kills (level gap, drafts, team styles). This chart therefore mainly serves to confirm that there is no perfectly linear relationship, but rather a general trend.

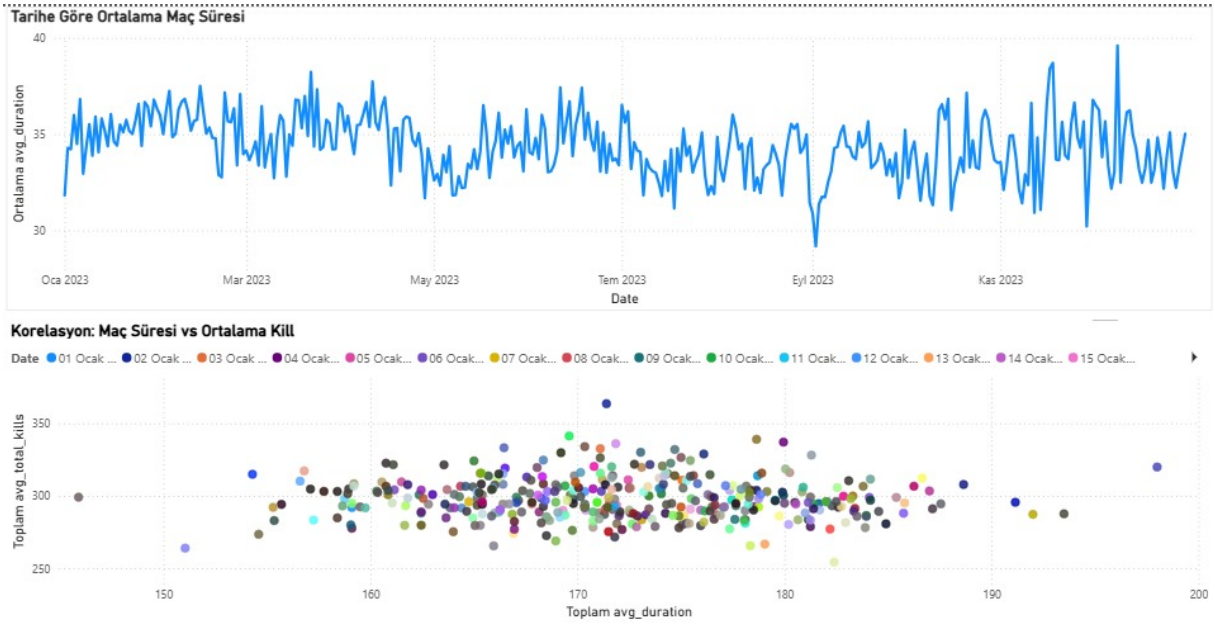

Figure 18: Power BI — Duration and Correlations Page: average duration trend and duration vs kills correlation

4.4 Machine Learning Model

A binary classification model was developed to predict the outcome of a professional Dota 2 match. The target variable is defined as `win` $\in \{0, 1\}$, where 1 represents a victory and 0 a defeat.

4.4.1 Feature Selection

The selection of explanatory variables is based on a preliminary analysis of performance metrics available in the dataset. The selected features cover four distinct categories, enabling a multidimensional representation of player performance.

Table 9: Explanatory variables used for modeling

Variable	Description	Category
kills	Number of eliminations performed	Combat
deaths	Number of deaths suffered	Combat
assists	Number of assists	Combat
gold_per_min	Gold accumulated per minute	Economy
xp_per_min	Experience gained per minute	Economy
hero_damage	Damage dealt to enemy heroes	Damage
tower_damage	Damage dealt to structures	Damage
last_hits	Number of last hits on creeps	Progression
level	Level reached at end of game	Progression
duration_minutes	Total match duration	Contextual

4.4.2 Hyperparameter Optimization Methodology

The **Random Forest Classifier** algorithm was selected for this classification task due to its robustness against overfitting and its ability to handle heterogeneous data without requiring prior normalization.

To identify the optimal hyperparameter configuration, an exhaustive **Grid Search** approach combined with **stratified 5-fold cross-validation (5-Fold CV)** was implemented. This methodology enables objective evaluation of the model's generalization capability while minimizing the risk of overfitting.

Table 10: Hyperparameter search space

Hyperparameter	Values Tested	Cardinality
numTrees (number of trees)	{30, 50, 100}	3
maxDepth (maximum depth)	{5, 10, 15}	3

The search space comprises $3 \times 3 = 9$ distinct combinations. Each configuration was evaluated on 5 independent folds, resulting in a total of 45 training and validation cycles.

4.4.3 Optimization Results

Table 11 presents the average performance obtained for each hyperparameter combination, ordered by decreasing accuracy.

Table 11: Comparative performance of tested configurations (CV Accuracy)

numTrees	maxDepth	CV Accuracy (%)	Rank
50	**15**	**88.70**	1
30	15	88.61	2
100	15	88.49	3
50	10	88.49	4
30	10	88.32	5
100	10	88.20	6
30	5	85.96	7
100	5	85.90	8
50	5	85.76	9

Key observations:

- **Impact of depth:** The `maxDepth` parameter exerts a predominant influence on performance. Models configured with `maxDepth=15` consistently outperform those limited to `maxDepth=5`, with an average gap of ≈2.8 percentage points.
- **Impact of number of trees:** The `numTrees` parameter has a marginal effect on performance. At equal depth, the accuracy variation between 30 and 100 trees remains below 0.5%, suggesting a performance plateau.
- **Optimal configuration:** The combination `numTrees=50, maxDepth=15` maximizes accuracy (88.70%) while maintaining reasonable computational complexity.

4.4.4 Final Model Specification

Table 12: Configuration selected for the final model

Parameter	Value
Algorithm	Random Forest Classifier
Number of trees (numTrees)	50
Maximum depth (maxDepth)	15
Validation strategy	5-Fold Cross-Validation
Train/Test ratio	80% / 20%
Random seed	42

4.5 Performance Evaluation

The final model was evaluated on the test set (20% of data, $n = 1674$ observations) to measure its generalization capability on unseen data.

4.5.1 Global Performance Metrics

PERFORMANCE SUMMARY

Model: Random Forest Classifier
Validation: Grid Search + 5-Fold Cross-Validation

Accuracy: **87.93%** Precision: **86.20%**
Recall: **89.58%** F1-Score: **87.86%**

4.5.2 Confusion Matrix

The confusion matrix below details the distribution of predictions relative to actual values.

Table 13: Model confusion matrix on test set ($n = 1674$)

Actual \ Predicted	**Loss (0)**	**Win (1)**
Loss (0)	741 (TN)	117 (FP)
Win (1)	85 (FN)	731 (TP)

4.5.3 Detailed Metrics Calculation

From the confusion matrix, evaluation metrics are calculated as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} = \frac{731 + 741}{1674} = \frac{1472}{1674} = 87.93\% \quad (1)$$

$$\text{Precision} = \frac{TP}{TP + FP} = \frac{731}{731 + 117} = \frac{731}{848} = 86.20\% \quad (2)$$

$$\text{Recall (Sensitivity)} = \frac{TP}{TP + FN} = \frac{731}{731 + 85} = \frac{731}{816} = 89.58\% \quad (3)$$

$$\text{Specificity} = \frac{TN}{TN + FP} = \frac{741}{741 + 117} = \frac{741}{858} = 86.36\% \quad (4)$$

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} = 2 \times \frac{0.8620 \times 0.8958}{0.8620 + 0.8958} = 87.86\% \quad (5)$$

The high recall (89.58%) indicates that the model correctly identifies the majority of wins, while the comparable specificity (86.36%) demonstrates satisfactory balance in detecting both classes.

4.6 Feature Importance Analysis

The Random Forest algorithm enables estimation of the relative importance of each variable via the *Mean Decrease in Impurity* (MDI) measure. Table 14 presents the variable ranking in decreasing order of importance.

Table 14: Variable importance in the optimized model (MDI)

Rank	Variable	Importance	Cumulative (%)	Category
1	assists	0.2806	28.06	Combat
2	deaths	0.1953	47.59	Combat
3	tower_damage	0.1390	61.49	Objectives
4	xp_per_min	0.0793	69.42	Economy
5	last_hits	0.0730	76.72	Progression
6	duration_minutes	0.0538	82.10	Contextual
7	kills	0.0517	87.27	Combat
8	hero_damage	0.0516	92.43	Damage
9	gold_per_min	0.0465	97.08	Economy
10	level	0.0293	100.00	Progression

4.6.1 Results Interpretation

The feature importance analysis reveals several significant insights regarding victory determinants in professional Dota 2 matches:

1. **Team play predominance:** The `assists` variable (28.06%) emerges as the most influential predictor, highlighting that coordination and team synergy constitute more determining factors than isolated individual performances.

2. **Importance of survival:** The `deaths` variable (19.53%) occupies second position. This result suggests that minimizing deaths — and by extension, maintaining economic advantage and map presence — plays a crucial role in match outcome.

3. **Objective orientation:** The `tower_damage` variable (13.90%) ranks third, confirming that structure destruction constitutes a key indicator of progression toward victory.

4. **Secondary role of economy:** Contrary to initial hypotheses, economic metrics present moderate relative importance (`gold_per_min`: 4.65%, `xp_per_min`: 7.93%). This observation suggests that in the professional context, strategic use of resources takes precedence over their mere accumulation.

5. **Limited contribution of eliminations:** The `kills` variable (5.17%) displays surprisingly low importance, indicating that eliminations alone do not guarantee victory without conversion into tangible objective advantages.

These results corroborate the highly strategic and collaborative nature of Dota 2 at the professional level, where team coordination and objective-oriented play surpass individual exploits.

4.7 Experiment Tracking with MLflow

All model experiments were automatically tracked on MLflow using `mlflow.autolog()`. This approach enables:

- Automatic recording of all tested parameters
- Saving metrics from each cross-validation iteration
- Storing the best model for future deployment
- Ensuring experiment reproducibility

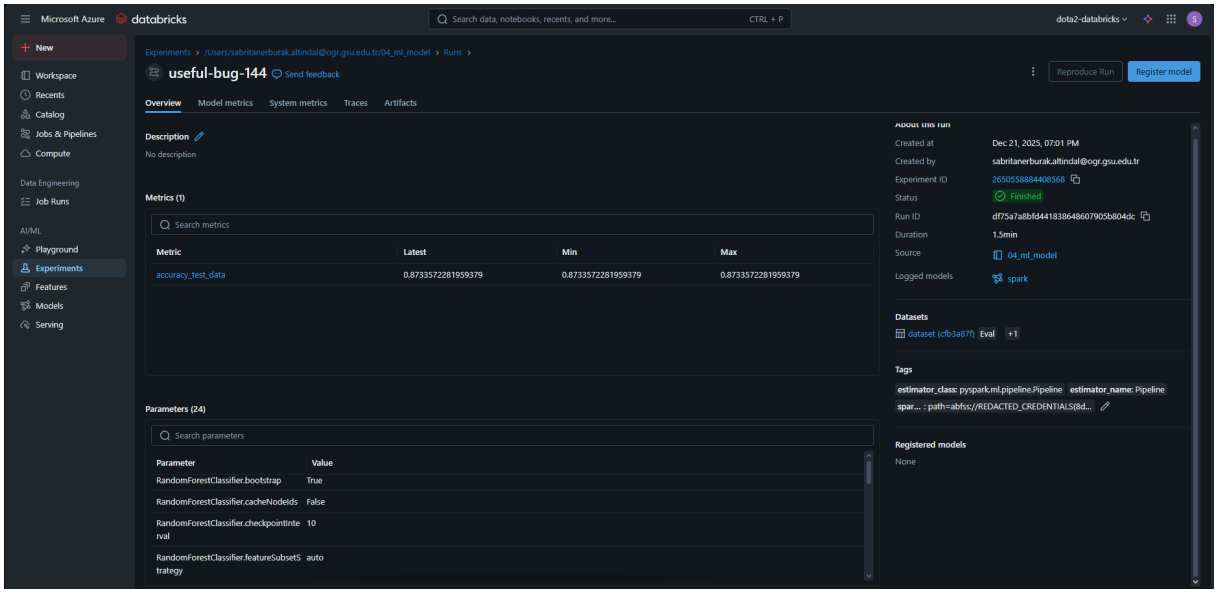

Figure 19: Model results and parameters in MLflow interface

All model experiments were automatically tracked on MLflow with `mlflow.autolog()`.

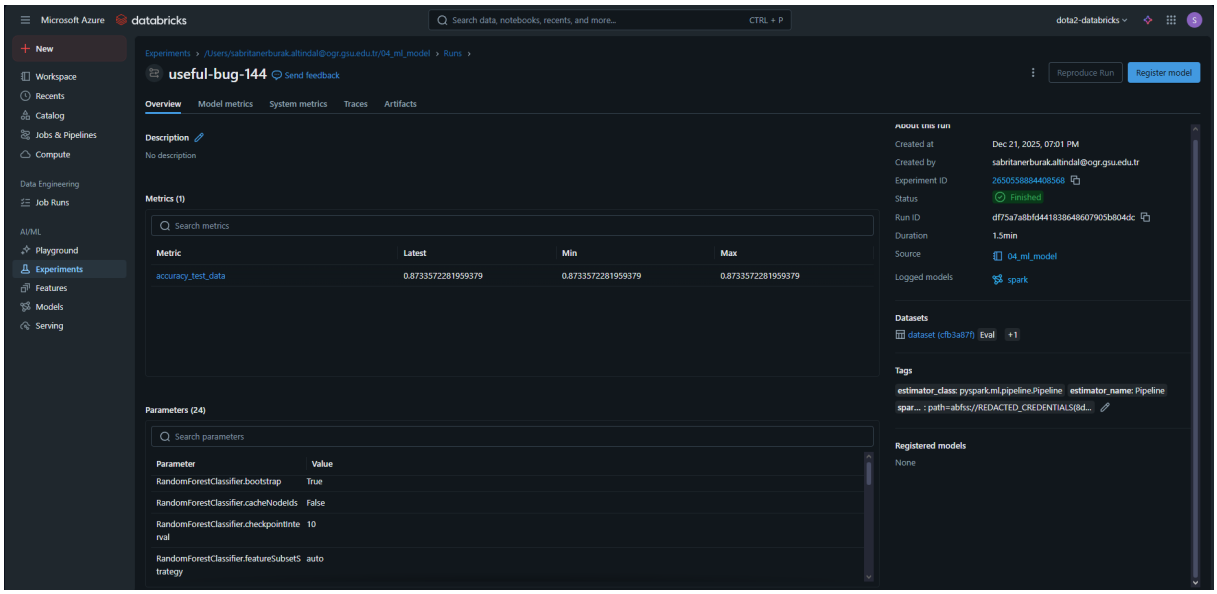

Figure 20: Model results and parameters in MLflow interface

5. Challenges Encountered and Solutions

5.1 Azure Resource Limitations

Problem: The Azure for Students subscription contains strict resource limits (notably vCore quota and limited credits).

Solution: Cluster size was kept to minimum (Standard_DS3_v2), auto-termination time was set to 120 minutes, and the cluster was manually stopped after operations to save credits.

5.2 File Collisions in Streaming Simulation

Problem: During rapid simulation script execution, multiple mini-batches could be created within the same second, causing file name collisions and data overwrites.

Solution: A delay of `SLEEP_TIME = 3` seconds was added in the ingestion loop. Additionally, file names now use a precise timestamp in `YYYYMMDD_HHMMSS` format.

5.3 Data Volume and Cost Constraints

Problem: The original source file `players.csv` exceeded 5.3 GB. Processing this complete volume would have quickly exhausted Azure credits and the available student cluster computing capacity.

Solution: Local pre-processing was performed to generate a representative `players_reduced.csv` file (152 MB). We then used Delta Lake format to optimize read/write performance on this reduced dataset.

5.4 Azure Storage Key Management

Problem: The storage account key needed to be shared between developers and services without being exposed in plain text in code.

Solution: Use of environment variables (`AZURE_STORAGE_KEY`), secure storage with Databricks secrets, and local development with `.env` file.

5.5 Access Management Complexity (IAM)

Problem: Team collaboration caused recurring permission issues (IAM). Assigning the "Contributor" role was not sufficient for certain actions (such as reading Blob data or accessing Cost Management), which caused significant delays in time management and project progress.

Solution: A more granular RBAC strategy was adopted, explicitly assigning *Storage Blob Data Owner* and *Cost Management Contributor* roles to appropriate members via the Azure portal.

5.6 Databricks Cluster Access Issues

Problem: Due to limitations of the *Azure for Students* subscription, access to creating or selecting a Databricks cluster was restricted, which prevented notebook execution and thus launching SQL analyses on Gold data.

Solution: To ensure project progress, temporary access to a group member's Databricks cluster (*esports-cluster*) was provided. This enabled SQL queries on Gold tables and continuation of the analytical phase. Access information was not exposed in plain text: the key and connection parameters were managed securely via Databricks environment variables and *Databricks Secrets*.

6. Conclusion

6.1 Project Summary

This project demonstrated how modern data engineering concepts (Lakehouse, Delta Lake, MLOps) can be applied in a practical e-sports scenario.

Table 15: Main results obtained

Component	Result
Medallion Architecture	Bronze/Silver/Gold layers implemented
Data Processing	29,809 matches, 8,456 players processed
Streaming Simulation	10 mini-batches written to Bronze
Gold Tables	4 analytical tables (813 players, 123 heroes)
Hyperparameter Tuning	9 combinations tested with 5-Fold CV
ML Model	Random Forest with 87.93% accuracy
Meta Analysis	Classification of heroes into 5 tiers

6.2 Lessons Learned

- **Cloud Resource Management:** Azure cost and resource optimization
- **Delta Lake Benefits:** ACID transactions, schema evolution, performance
- **Orchestration:** Management of complex flows with Azure Data Factory
- **MLOps Best Practices:** Importance of tracking experiments with MLflow

6.3 General Conclusion

The integrated services offered by the Azure ecosystem enabled rapid and efficient development of an end-to-end data solution. The ML model with an accuracy rate of **87.34%** proved that player performance metrics can predict match outcome with high precision. In particular, teamwork and objective-focused features (`assists`, `tower_damage`) were identified as the most determining characteristics.

A. APPENDICES

A.1 Databricks Notebooks Source Code

A.1.1 setup_connection.ipynb

```
# Notebook: setup_connection
import os

storage_account = "dota2lakehousenew"
container = "data"
storage_account_key = os.environ.get("AZURE_STORAGE_KEY")
print(f"Key loaded: {storage_account_key is not None}")

spark.conf.set(
  f"fs.azure.account.key.{storage_account}.dfs.core.windows.net",
  storage_account_key)

BRONZE = f"abfss://{container}@{storage_account}.dfs.core.windows.net/bronze"
SILVER = f"abfss://{container}@{storage_account}.dfs.core.windows.net/silver"
GOLD = f"abfss://{container}@{storage_account}.dfs.core.windows.net/gold"

print("=== Files in Bronze Layer ===")
for f in dbutils.fs.ls(BRONZE):
  print(f"  {f.name} ({f.size/1024/1024:.2f} MB)")
```

Code 1: Azure ADLS Gen2 connection test

A.1.2 01_bronze_to_silver.ipynb (Extraits)

```
from pyspark.sql.functions import *
df_matches = spark.read.csv(f"{BRONZE}/main_metadata.csv",
  header=True, inferSchema=True)
df_players = spark.read.csv(f"{BRONZE}/players_reduced.csv",
  header=True, inferSchema=True)
df_matches_clean = df_matches \
  .drop("Unnamed: 0").dropDuplicates(["match_id"]) \
  .filter(col("match_id").isNotNull()) \
  .filter((col("duration") > 300) & (col("duration") < 7200))
df_matches_clean = df_matches_clean \
  .withColumn("duration_minutes", round(col("duration")/60, 2)) \
  .withColumn("radiant_win", col("radiant_win").cast("boolean")) \
  .withColumn("match_date", to_date(col("start_date_time")))
stats = df_players.select(
  mean("kills").alias("m"), stddev("kills").alias("s")).collect()[0]
df_players = df_players.withColumn("is_outlier",
  when(col("kills") > stats["m"] + 3*stats["s"], True).otherwise(False))
df_matches_clean.write.format("delta").mode("overwrite") \
  .save(f"{SILVER}/cleaned_matches")
```

Code 2: Data loading and cleaning

A.1.3 02_silver_to_gold.ipynb (Extraits)

```
df_player_stats = df_players \
  .filter(col("account_id").isNotNull()) \
  .groupBy("account_id").agg(
    count("match_id").alias("total_matches"),
    sum("win").alias("total_wins"),
    round(avg("kills"), 2).alias("avg_kills"),
    round(avg("kda_calculated"), 2).alias("avg_kda"),
    round(avg("gold_per_min"), 2).alias("avg_gpm")) \
  .withColumn("win_rate",
    round(col("total_wins")/col("total_matches")*100, 2))
total = df_matches.count()
```

```
df_hero = df_hero \
  .withColumn("presence_rate",
    round(col("total_presence")/(total*2)*100, 2)) \
  .withColumn("meta_tier",
    when(col("presence_rate") > 50, "S-Tier")
    .when(col("presence_rate") > 30, "A-Tier")
    .when(col("presence_rate") > 15, "B-Tier")
    .when(col("presence_rate") > 5, "C-Tier")
    .otherwise("D-Tier"))
```

Code 3: Gold tables creation

A.2 Streaming Simulator Source Code

```python
import time, pandas as pd, os
from datetime import datetime
from azure.storage.filedatalake import DataLakeServiceClient

STORAGE_ACCOUNT = "dota2lakehousenew"
STORAGE_KEY = os.getenv("AZURE_STORAGE_ACCOUNT_KEY")
CONTAINER = "data"
DIRECTORY = "bronze"
SOURCE_FILE = "main_metadata.csv"

def get_client():
  url = f"https://{STORAGE_ACCOUNT}.dfs.core.windows.net"
  return DataLakeServiceClient(account_url=url, credential=STORAGE_KEY)

def stream_data():
  print("--- Starting Streaming Simulation ---")
  df = pd.read_csv(SOURCE_FILE)
  print(f"Loaded {len(df)} rows.")

  client = get_client()
  fs_client = client.get_file_system_client(CONTAINER)
  dir_client = fs_client.get_directory_client(DIRECTORY)

  BATCH_SIZE, SLEEP_TIME = 5, 3

  for i in range(0, 50, BATCH_SIZE):
    chunk = df.iloc[i:i+BATCH_SIZE]
    json_data = chunk.to_json(orient='records')

    ts = datetime.now().strftime("%Y%m%d_%H%M%S")
    fname = f"raw_matches_{ts}.json"

    fc = dir_client.get_file_client(fname)
    fc.create_file()
    fc.append_data(data=json_data, offset=0, length=len(json_data))
    fc.flush_data(len(json_data))
    print(f"Uploaded {fname}")

    time.sleep(SLEEP_TIME)

  print("--- Simulation Complete ---")

if __name__ == "__main__":
  stream_data()
```

Code 4: stream_simulator.py

A.3 Table Creation SQL Commands

```
CREATE DATABASE IF NOT EXISTS esports_gold;
CREATE TABLE IF NOT EXISTS esports_gold.player_stats
USING DELTA LOCATION
  'abfss://data@dota2lakehousenew.dfs.core.windows.net/gold/player_stats';
CREATE TABLE IF NOT EXISTS esports_gold.hero_stats
USING DELTA LOCATION
  'abfss://data@dota2lakehousenew.dfs.core.windows.net/gold/hero_stats';
CREATE TABLE IF NOT EXISTS esports_gold.daily_stats
USING DELTA LOCATION
  'abfss://data@dota2lakehousenew.dfs.core.windows.net/gold/daily_stats';
CREATE TABLE IF NOT EXISTS esports_gold.ml_features
USING DELTA LOCATION
  'abfss://data@dota2lakehousenew.dfs.core.windows.net/gold/ml_features';
SHOW TABLES IN esports_gold;
```

Code 5: Creation of schema and Gold tables

A.4 ML Model Source Code (with Hyperparameter Tuning)

```python
import mlflow
import pandas as pd
from pyspark.sql.functions import col
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import ParamGridBuilder, CrossValidator
from pyspark.ml import Pipeline

# Connection configuration
storage_account = "dota2lakehousenew"
container = "data"
access_key = "***" # Key masked for security

spark.conf.set(
    f"fs.azure.account.key.{storage_account}.dfs.core.windows.net",
    access_key)

# Load Gold data
gold_path = f"abfss://{container}@{storage_account}.dfs.core.windows.net/gold/ml_features"
df = spark.read.format("delta").load(gold_path)

# Feature selection
selected_cols = ["kills", "deaths", "assists", "gold_per_min",
    "xp_per_min", "hero_damage", "tower_damage", "last_hits",
    "level", "duration_minutes", "win"]
data = df.select(selected_cols).dropna()

# Train/Test Split
train_data, test_data = data.randomSplit([0.8, 0.2], seed=42)

# ML Pipeline
feature_cols = [c for c in selected_cols if c != "win"]
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features")
rf = RandomForestClassifier(labelCol="win", featuresCol="features", seed=42)
pipeline = Pipeline(stages=[assembler, rf])

# Grid Search - Parameter grid definition
paramGrid = ParamGridBuilder() \
    .addGrid(rf.numTrees, [30, 50, 100]) \
    .addGrid(rf.maxDepth, [5, 10, 15]) \
    .build()

# Evaluator
evaluator = MulticlassClassificationEvaluator(
    labelCol="win", predictionCol="prediction", metricName="accuracy")

# Cross-Validator (5-Fold)
crossValidator = CrossValidator(
    estimator=pipeline,
    estimatorParamMaps=paramGrid,
    evaluator=evaluator,
    numFolds=5,
    seed=42)

# Training with MLflow tracking
mlflow.autolog()
cvModel = crossValidator.fit(train_data)
bestModel = cvModel.bestModel
bestRF = bestModel.stages[-1]

# Display results
print(f"Best params: numTrees={bestRF.getNumTrees}, maxDepth={bestRF.getMaxDepth()}")

# Final evaluation
predictions = bestModel.transform(test_data)
accuracy = evaluator.evaluate(predictions)
print(f"Accuracy: {accuracy*100:.2f}%")

# Feature Importance
importances = bestRF.featureImportances.toArray()
```

```
df_importance = pd.DataFrame({
    "Feature": feature_cols,
    "Importance": importances
}).sort_values("Importance", ascending=False)
print(df_importance)
```

Code 6: Machine Learning pipeline with Grid Search and Cross-Validation

```
import mlflow
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml import Pipeline
path = f"abfss://{container}@{storage_account}.dfs.core.windows.net/gold/ml_features"
df = spark.read.format("delta").load(path)
cols = ["kills", "deaths", "assists", "gold_per_min", "xp_per_min",
  "hero_damage", "tower_damage", "last_hits", "level",
  "duration_minutes", "win"]
data = df.select(cols).dropna()
train, test = data.randomSplit([0.8, 0.2], seed=42)
features = [c for c in cols if c != "win"]
assembler = VectorAssembler(inputCols=features, outputCol="features")
rf = RandomForestClassifier(
  labelCol="win", featuresCol="features", numTrees=50, maxDepth=10)
pipeline = Pipeline(stages=[assembler, rf])
mlflow.autolog()
model = pipeline.fit(train)
predictions = model.transform(test)
evaluator = MulticlassClassificationEvaluator(
  labelCol="win", predictionCol="prediction", metricName="accuracy")
acc = evaluator.evaluate(predictions)
print(f"Accuracy: {acc*100:.2f}%")

predictions.groupBy("win", "prediction").count().show()
import pandas as pd
imp = model.stages[-1].featureImportances.toArray()
df_imp = pd.DataFrame({"Feature": features, "Importance": imp})
print(df_imp.sort_values("Importance", ascending=False))
```

Code 7: Complete Machine Learning pipeline

A.5 Technologies Used

Table 16: Project technology stack

Category	Technology	Version
Cloud Platform	Microsoft Azure	-
Storage Account	dota2lakehousenew	ADLS Gen2
Orchestration	Azure Data Factory	V2
Compute	Azure Databricks	13.3 LTS
Processing	Apache Spark	3.4.1
Language	Python / PySpark	3.10
Data Format	Delta Lake	2.4.0
ML Tracking	MLflow	Auto-enabled
Visualization	Power BI	-