



## 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 Çolakoglu – 21401946

**Member 4:** Ceren Akbaş – 22401028

Academic Year 2025–2026

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Project Context . . . . .	4
1.2	Dataset Used . . . . .	4
1.3	Project Objectives . . . . .	4
1.4	Azure Services Used . . . . .	5
<b>2</b>	<b>Architectural Design</b>	<b>5</b>
2.1	Medallion Architecture . . . . .	6
2.2	Azure Data Lake Storage Gen2 Structure . . . . .	7
2.3	Data Flow Diagram . . . . .	7
<b>3</b>	<b>Implementation</b>	<b>7</b>
3.1	Batch Processing Pipeline . . . . .	8
3.2	Trigger Configuration . . . . .	9
3.3	Databricks Notebooks . . . . .	10
3.4	Streaming Simulation . . . . .	10
3.5	ETL Transformations . . . . .	11
3.6	Delta Lake Table Management . . . . .	13
<b>4</b>	<b>Data Analysis and Machine Learning</b>	<b>13</b>
4.1	Accessing Gold Tables and SQL Execution (Databricks) . . . . .	14
4.2	Analytical SQL Queries (Results) . . . . .	15
4.3	Power BI Visualization . . . . .	20
4.4	Machine Learning Model . . . . .	24
4.5	Model Results . . . . .	25
4.6	Feature Importance Analysis . . . . .	25
4.7	MLflow Tracking . . . . .	26
<b>5</b>	<b>Challenges Encountered and Solutions</b>	<b>26</b>
5.1	Azure Resource Limitations . . . . .	27
5.2	File Collisions in Streaming Simulation . . . . .	27
5.3	Data Volume and Cost Constraints . . . . .	27
5.4	Azure Storage Key Management . . . . .	27
5.5	Access Management Complexity (IAM) . . . . .	27
5.6	Databricks Cluster Access Issues . . . . .	28
<b>6</b>	<b>Conclusion</b>	<b>28</b>
6.1	Project Summary . . . . .	29
6.2	Lessons Learned . . . . .	29
6.3	General Conclusion . . . . .	29
<b>Appendices</b>		<b>30</b>

# List of Figures

1	Azure Portal view of Bronze/Silver/Gold folder structure in ADLS Gen2 . . . . .	7
2	Data flow diagram . . . . .	8
3	Execution details of the <i>Copy Data</i> activity confirming successful ingestion of 4 source files (212 MB) to the Bronze container. . . . .	8
4	pl_master_esports pipeline diagram in Azure Data Factory . . . . .	9
5	“Succeeded” status and “Triggered by tr_daily_esports” in the ADF Monitor screen . . . . .	10
6	stream_simulator.py execution output . . . . .	11
7	Tier classification output in silver_to_gold notebook . . . . .	12
8	List of tables under esports_gold schema in Databricks Catalog . . . . .	13
9	Loading Gold tables from ADLS Gen2 and creating temporary views in Databricks . . . . .	15
10	SQL Result: Top 10 players by average KDA (with kills, deaths, assists) . . . . .	16
11	SQL Result: daily statistics overview (match_count and average duration in minutes) . . . . .	17
12	SQL Result: Top 10 heroes by average GPM and XPM . . . . .	18
13	SQL Result: Top 10 combat-oriented heroes (kills/assists) and number of games played . . . . .	19
14	SQL Result: Top 10 most popular heroes by times_played (with wins and win_rate) . . . . .	20
15	Power BI — Overview Page: temporal filter, daily trend of match count, and global KPIs . . . . .	21
16	Power BI — Top 10 Players Page: average KDA vs win rate comparison and detail table . . . . .	22
17	Power BI — Hero Meta Page: presence rate vs win rate relationship and Top 10 most present heroes . . . . .	23
18	Power BI — Duration and Correlations Page: average duration trend and duration vs kills correlation . . . . .	24
19	Model results and parameters in MLflow interface . . . . .	26

## List of Tables

1	Dataset characteristics . . . . .	4
2	Microsoft Azure services used . . . . .	5
3	Silver layer tables . . . . .	6
4	Gold layer tables . . . . .	6
5	Trigger configuration . . . . .	9
6	Databricks cluster configuration . . . . .	10
7	Meta Tier classification criteria . . . . .	12
8	Duration analysis results . . . . .	12
9	Features used . . . . .	24
10	Confusion Matrix . . . . .	25
11	Feature importance ranking . . . . .	25
12	Main results obtained . . . . .	29
13	Project technology stack . . . . .	35

# 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/

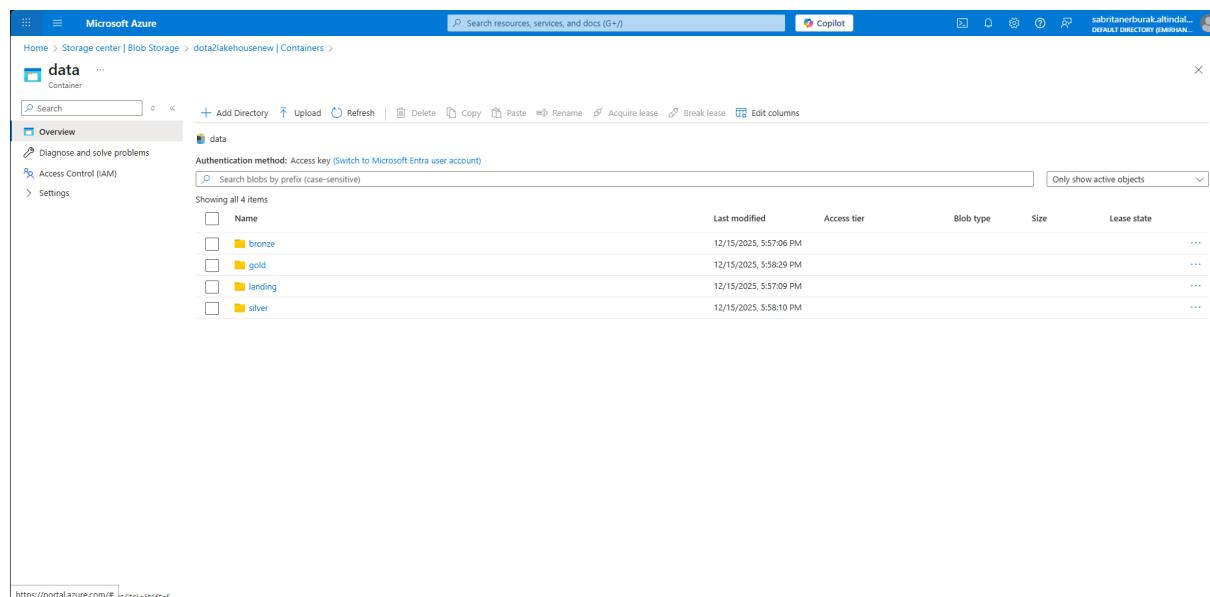


Figure 1: Azure Portal view of Bronze/Silver/Gold folder structure in ADLS Gen2

## 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.

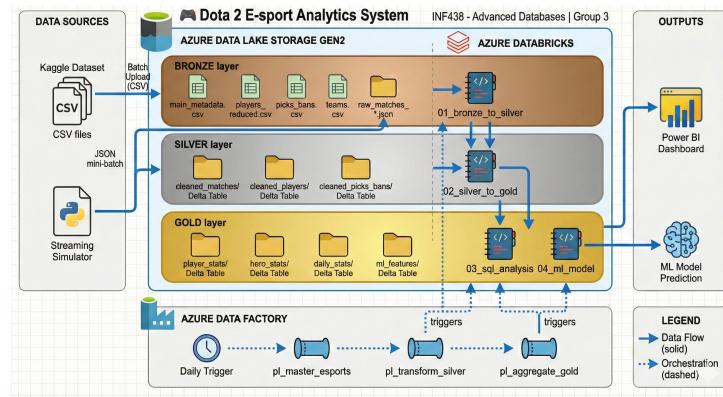


Figure 2: Data flow diagram

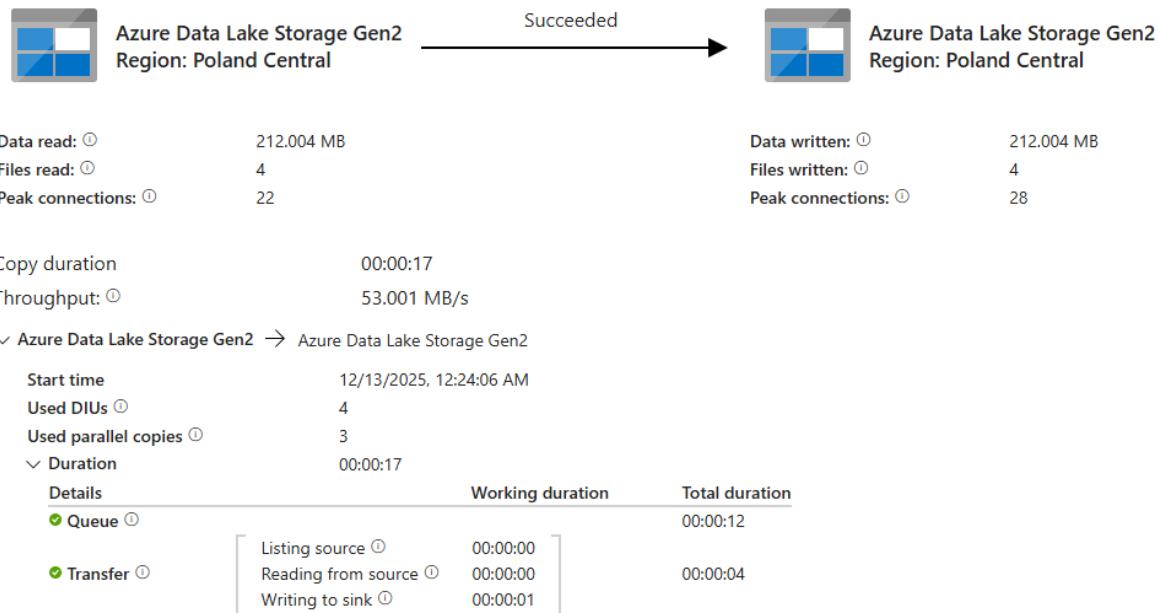
### 3. Implementation

#### 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.

Activity run id: 0d82b85a-cd5f-4180-8ed5-f1b8f717fae3

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)

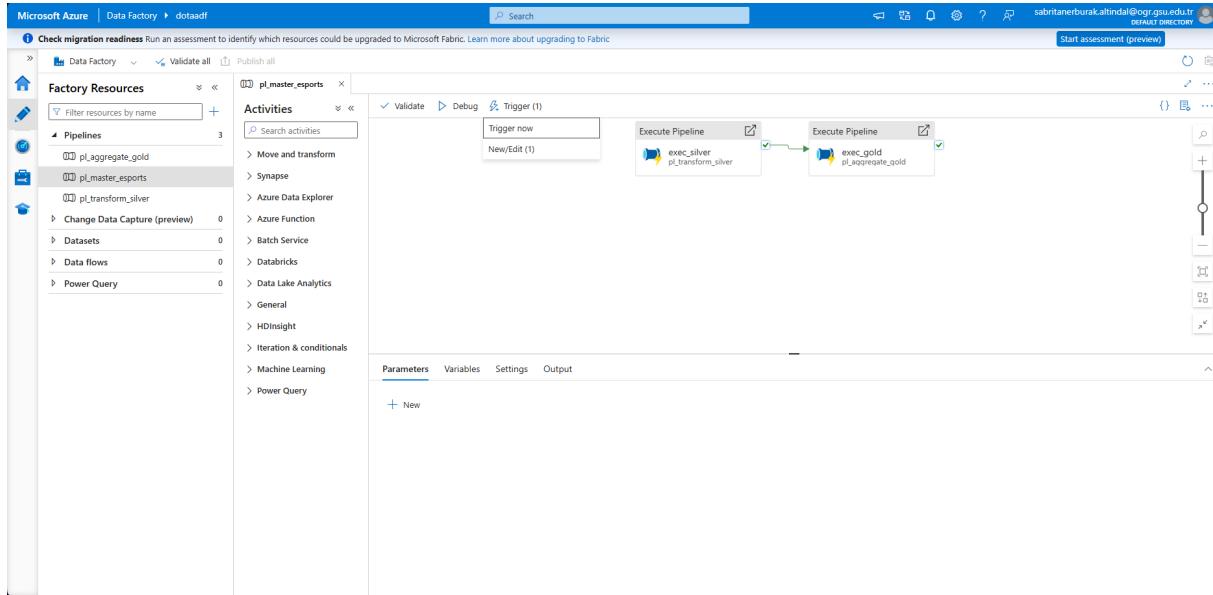


Figure 4: pl\_master\_esports pipeline diagram in Azure Data Factory

### 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

Pipeline name	Run start	Run end	Duration	Triggered by	Status	Run ID	
pl_aggregate_gold	12/21/2025, 2:04:22 AM	12/21/2025, 2:05:13 AM	51s	98ab0d28-5a69-42e...	Succeeded	Original	46f7882f-4a4f-4a30-959e-a75
pl_transform_silver	12/21/2025, 2:00:05 AM	12/21/2025, 2:04:20 AM	4m 15s	80f9d01b-b281-454...	Succeeded	Original	b18bef90-4582-4661-821b-19
pl_master_esports	12/21/2025, 2:00:01 AM	12/21/2025, 2:05:14 AM	5m 14s	tr_daily_esports	Succeeded	Original	c1794e0b-9e56-443c-9bd1-bc
pl_aggregate_gold	12/21/2025, 1:54:06 AM	12/21/2025, 1:55:01 AM	56s	dbcd979-b51c-4b1f...	Succeeded	Original	c539cb2c-3567-4b28-bd74-11
pl_transform_silver	12/21/2025, 1:43:23 AM	12/21/2025, 1:54:04 AM	10m 42s	e592d65c-6531-4c6...	Succeeded	Original	1912678d-6a93-4947-8267-f7
pl_master_esports	12/21/2025, 1:43:15 AM	12/21/2025, 1:55:03 AM	11m 48s	Manual trigger	Succeeded	Original	c95e9487-bd0b-48c4-bb1e-f9

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 ??](#).

### 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

```

data > bronze
Authentication method: Access key (Switch to Microsoft Entra user)
Search blobs by prefix (case-sensitive)

Showing all 7 items
Name
[.] main_metadata.csv
picks_bans.csv
players_reduced.csv
raw_matches_20251213_091757.json
raw_matches_20251213_091801.json
raw_matches_20251213_091804.json
teams.csv

25  def stream_data():
26      file_client.flush_data(len(json_data))
27      print("Done.")
28  except Exception as e:
29      print(f"Upload Error: {e}")
30
31  # 5. Wait (Simulate real-time gap)
32  time.sleep(SLEEP_TIME)
33
34  print("\n--- Simulation Complete ---")
35
36  if __name__ == "__main__":
37      stream_data()

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS python
PS C:\Users\emirhan\ders\04_01\ileri-veri-tabanlari\projet> python
--- Starting Streaming Simulation using main_metadata.csv ---
Loading dataset...
Loaded 29810 rows. Starting stream...
Uploading raw_matches_20251213_091757.json... Done.
Uploading raw_matches_20251213_091801.json... Done.
Uploading raw_matches_20251213_091804.json... Done.
Uploading raw_matches_20251213_091807.json... Done.

```

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 ??](#).

## 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

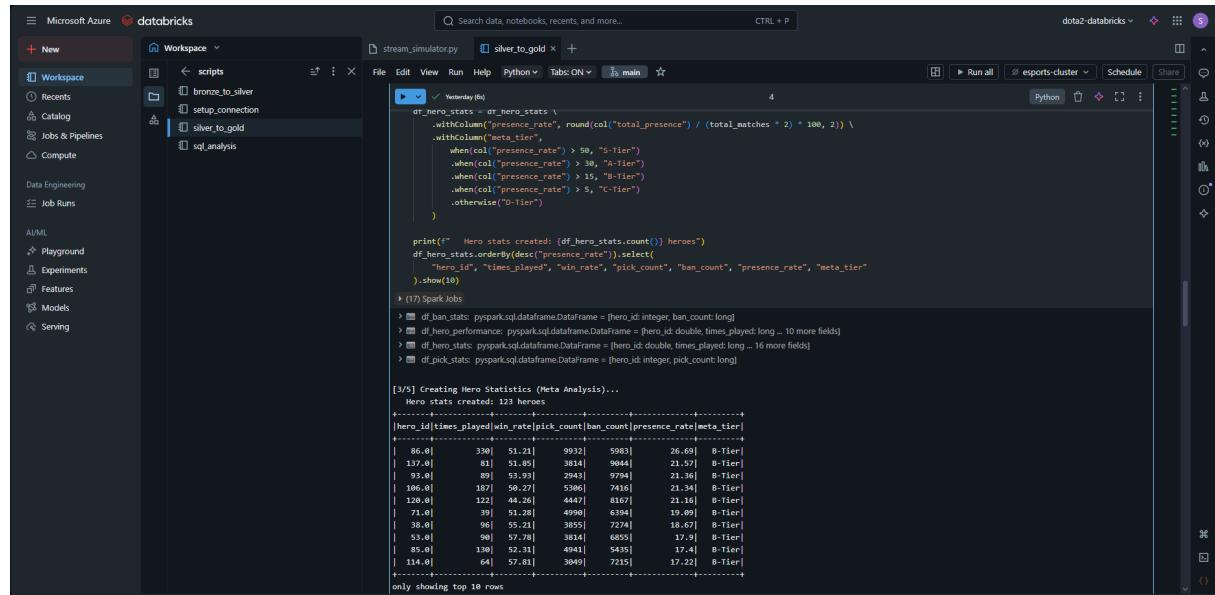
The cleaned data received business logic and advanced calculations.

**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 ??).

Table 7: Meta Tier classification criteria

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



```

# [1/1] Notebook (6)
df_hero_stats = df_hero_stats \
    .withColumn("presence_rate", round((total_presence) / (total_matches * 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")
    )

print(f"Hero stats created: {df_hero_stats.count()} heroes")
df_hero_stats.orderBy(desc("presence_rate")).select(
    "hero_id", "times_played", "win_rate", "pick_count", "ban_count", "presence_rate", "meta_tier"
).show(10)

# [2/5] Creating Hero Statistics (Meta Analysis)...
Hero stats created: 123 heroes
+-----+-----+-----+-----+-----+
|hero_id|times_played|win_rate|pick_count|ban_count|presence_rate|meta_tier|
+-----+-----+-----+-----+-----+
|  1.0|      51.41|   99.99|     50.0|     26.0|      50.0| B-Tier|
| 17.0|      51.55|   99.99|     30.0|     24.0|      50.0| B-Tier|
| 92.0|      51.93|   99.43|     29.0|     24.0|      51.57| B-Tier|
| 106.0|     107.0|   99.99|     50.27|     53.06|      51.36| B-Tier|
| 128.0|     122.0|   44.26|   4447|     8167|      21.16| B-Tier|
| 71.0|      39.1|   51.28|   4998|     6394|      19.89| B-Tier|
| 38.0|      96.0|   55.21|   3855|     7274|      18.67| B-Tier|
| 53.0|      98.0|   57.78|   3814|     6855|      17.9| B-Tier|
| 85.0|     138.0|   52.31|   4941|     9435|      17.4| B-Tier|
|114.0|      64.0|   57.81|   3049|     7215|      17.22| B-Tier|
+-----+-----+-----+-----+-----+
only showing top 10 rows

```

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 ??](#).

The screenshot shows the Databricks Catalog Explorer interface. On the left, there is a sidebar with navigation links for Workspace, Recents, Catalog, Jobs & Pipelines, Compute, Data Engineering, Job Runs, A/ML, Playground, Experiments, Features, Models, and Serving. The Catalog section is currently selected. The main area displays the Catalog Explorer with a search bar at the top. Below the search bar, there is a 'For you' section with a 'All' tab selected. Under the 'Catalog' heading, there is a tree view of schemas. The 'samples' schema is expanded, showing its sub-tables: default, nyctaxi, and tpch. The 'tpch' schema has further sub-tables: customer, lineitem, nation, orders, part, partsupp, region, and supplier. Below the 'samples' schema, the 'Legacy' section is expanded, showing the 'hive\_metastore' schema, which contains the 'esports\_gold' schema. This schema has sub-tables: default, sqldf, and sql. The 'sql' table has sub-tables: daily\_stats, hero\_stats, ml\_features, and player\_stats.

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)

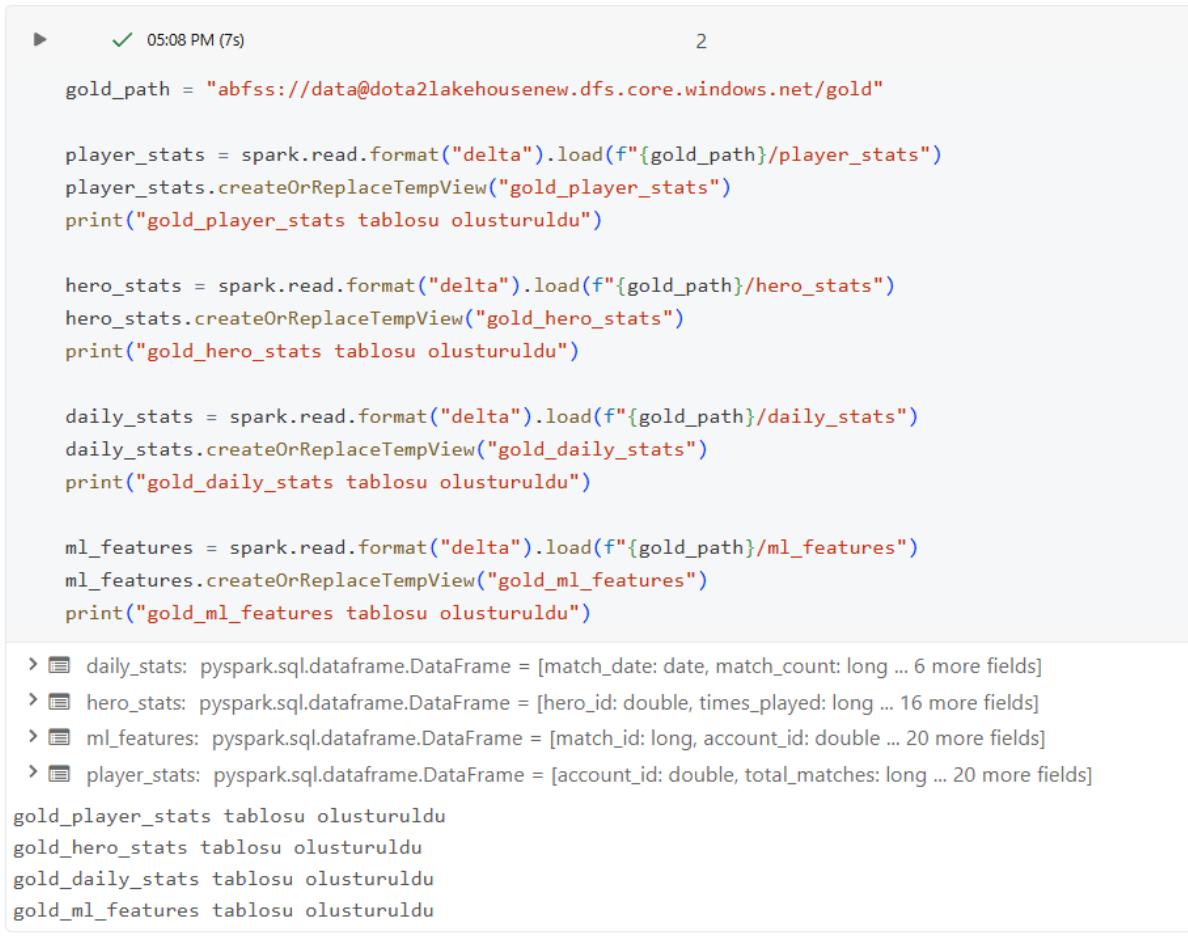
#### 4.1.1 Connection to Azure Data Lake Storage (ADLS Gen2)

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.



```

▶ ✓ 05:08 PM (7s) 2

gold_path = "abfss://data@data2lakehousenew.dfs.core.windows.net/gold"

player_stats = spark.read.format("delta").load(f"{gold_path}/player_stats")
player_stats.createOrReplaceTempView("gold_player_stats")
print("gold_player_stats tablosu olusturuldu")

hero_stats = spark.read.format("delta").load(f"{gold_path}/hero_stats")
hero_stats.createOrReplaceTempView("gold_hero_stats")
print("gold_hero_stats tablosu olusturuldu")

daily_stats = spark.read.format("delta").load(f"{gold_path}/daily_stats")
daily_stats.createOrReplaceTempView("gold_daily_stats")
print("gold_daily_stats tablosu olusturuldu")

ml_features = spark.read.format("delta").load(f"{gold_path}/ml_features")
ml_features.createOrReplaceTempView("gold_ml_features")
print("gold_ml_features tablosu olusturuldu")

> daily_stats: pyspark.sql.dataframe.DataFrame = [match_date: date, match_count: long ... 6 more fields]
> hero_stats: pyspark.sql.dataframe.DataFrame = [hero_id: double, times_played: long ... 16 more fields]
> ml_features: pyspark.sql.dataframe.DataFrame = [match_id: long, account_id: double ... 20 more fields]
> player_stats: pyspark.sql.dataframe.DataFrame = [account_id: double, total_matches: long ... 20 more fields]

gold_player_stats tablosu olusturuldu
gold_hero_stats tablosu olusturuldu
gold_daily_stats tablosu olusturuldu
gold_ml_features tablosu olusturuldu

```

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]

	1.2 account_id	1 <sup>2</sup> <sub>3</sub> total_kills	1 <sup>2</sup> <sub>3</sub> total_deaths	1 <sup>2</sup> <sub>3</sub> 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

➤ \_sqlldf: pyspark.sql.dataframe.DataFrame = [match\_date: date, match\_count: long ... 1 more field]

	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\_gpm: double, avg\_xpm: double]

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 fields]

Table +

	1.2 hero_id	1.2 avg_kills	1.2 avg_assists	1.2 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 %).

```

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	1.3 total_picks	1.3 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.

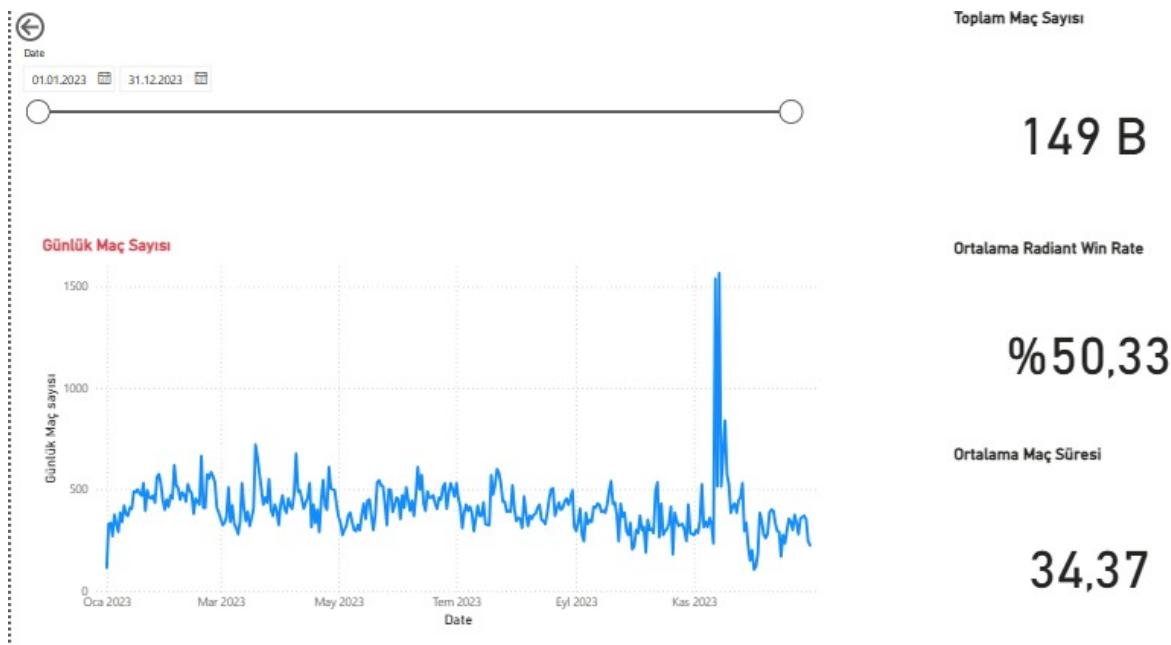


Figure 15: Power BI — Overview Page: temporal filter, daily trend of match count, and global KPIs

#### 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.



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.

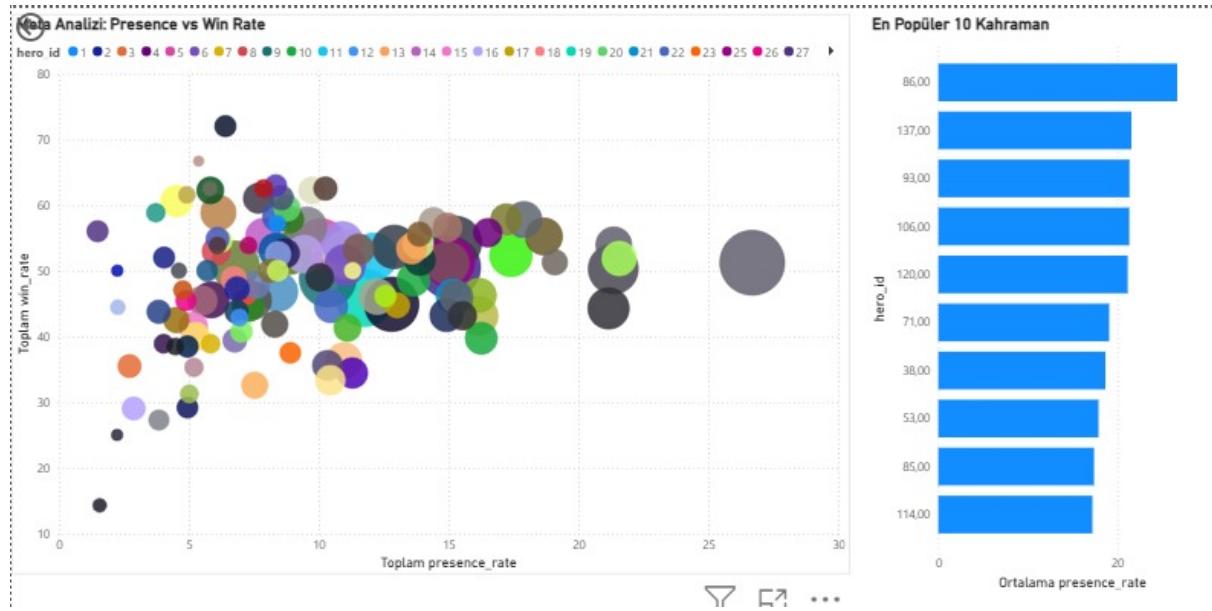


Figure 17: Power BI — Hero Meta Page: presence rate vs win rate relationship and Top 10 most present heroes

#### 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.

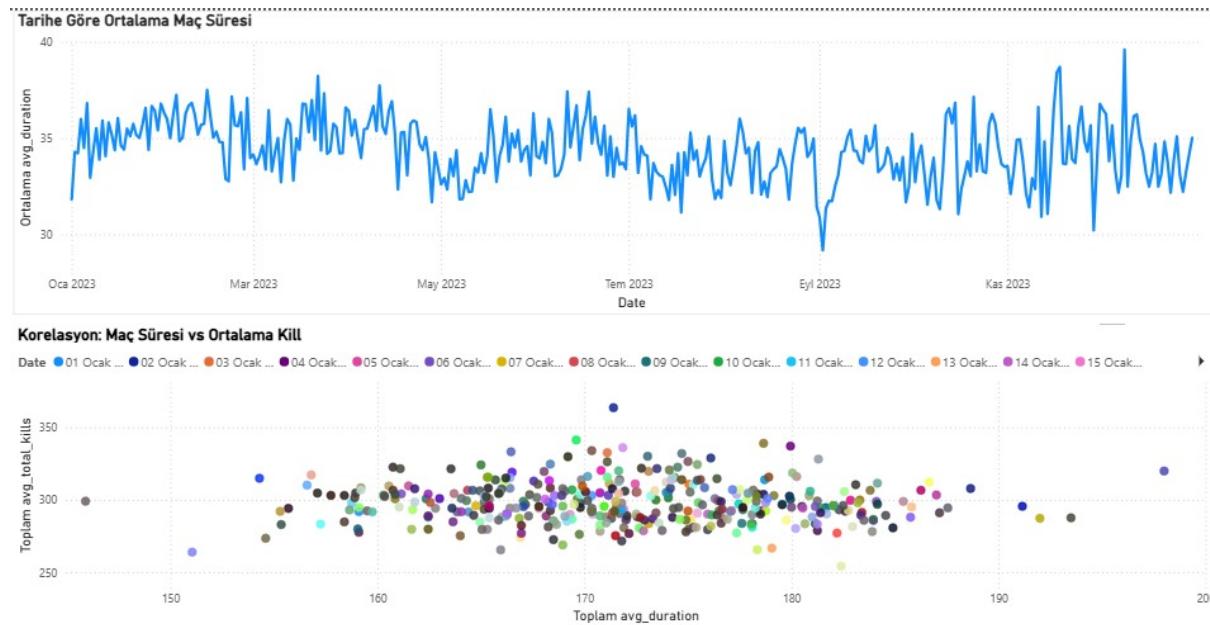


Figure 18: Power BI — Duration and Correlations Page: average duration trend and duration vs kills correlation

#### 4.4 Machine Learning Model

A binary classification model was developed to predict match outcome. The target variable is `win` (1 = Win, 0 = Loss).

Table 9: Features used

Feature	Description	Category
kills	Number of kills	Combat
deaths	Number of deaths	Combat
assists	Number of assists	Combat
gold_per_min	Gold per minute	Economy
xp_per_min	Experience per minute	Economy
hero_damage	Hero damage	Damage
tower_damage	Tower damage	Damage
last_hits	Number of last hits	Progression
level	Player level	Progression
duration_minutes	Match duration	Match Info

The **Random Forest Classifier** algorithm was chosen with parameters: `numTrees=50`, `maxDepth=10`, `Train/Test Split=80%/20%`.

The complete ML pipeline source code is available in [Appendix ??](#).

## 4.5 Model Results

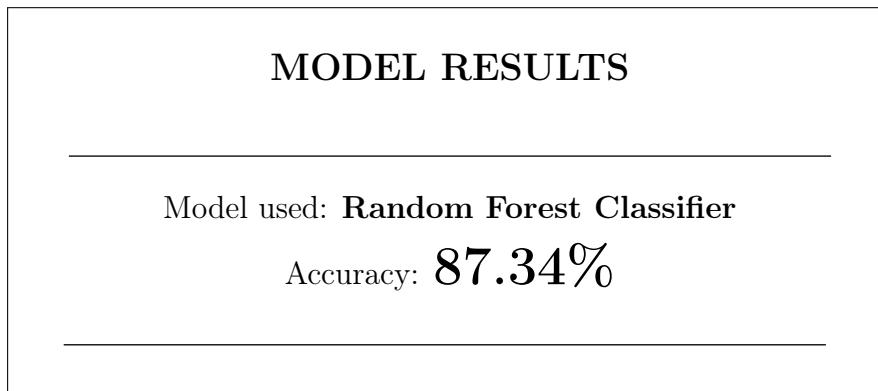


Table 10: Confusion Matrix

Actual / Predicted	Lose (0)	Win (1)
Lose (0)	727 (TN)	131 (FP)
Win (1)	81 (FN)	735 (TP)

**Calculated metrics:**

- True Positive Rate (Recall):  $\frac{735}{735+81} = 90.07\%$
- True Negative Rate:  $\frac{727}{727+131} = 84.73\%$
- Precision:  $\frac{735}{735+131} = 84.87\%$

## 4.6 Feature Importance Analysis

Table 11: Feature importance ranking

Rank	Feature	Importance
1	assists	0.311480
2	deaths	0.204537
3	tower_damage	0.147192
4	xp_per_min	0.073274
5	last_hits	0.058863
6	duration_minutes	0.049268
7	gold_per_min	0.048150
8	kills	0.040887
9	hero_damage	0.040081
10	level	0.026267

These results show that **teamwork** (`assists` with 0.311) is the most determining factor on match outcome, followed by **survival** (`deaths` with 0.205) and **objective damage** (`tower_damage` with 0.147).

Contrary to expectations, economic factors (`gold_per_min`: 0.048, `xp_per_min`: 0.073) have relatively low importance. This suggests that in professional Dota 2 matches, team coordination and objective-oriented play are more predictive of victory than individual resource accumulation.

## 4.7 MLflow Tracking

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

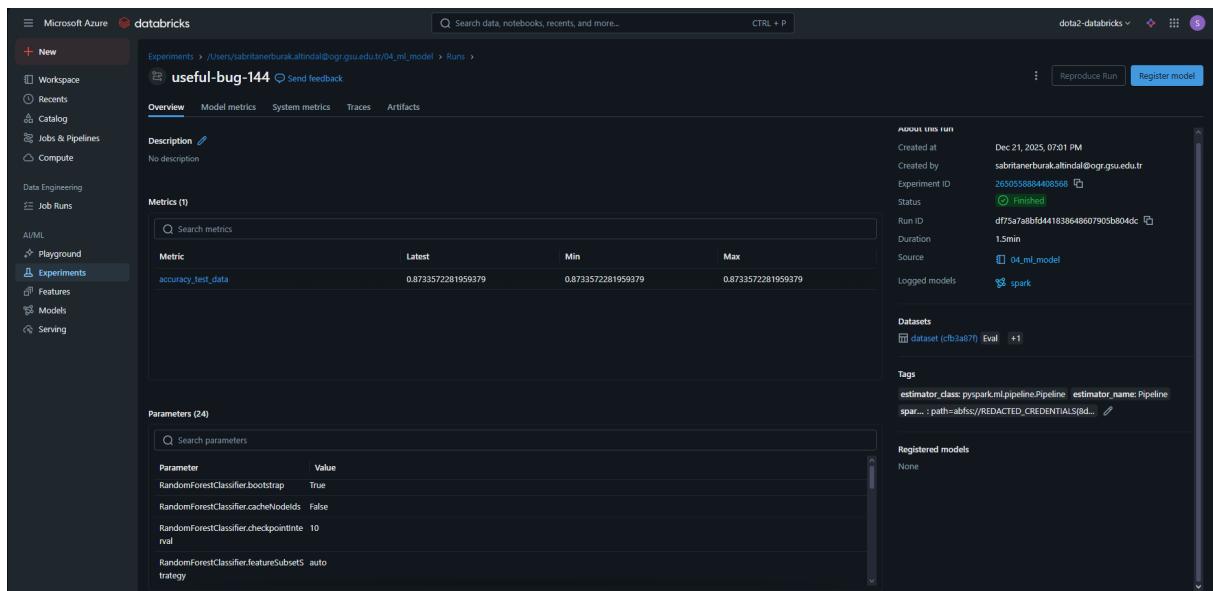


Figure 19: Model results and parameters in MLflow interface

## 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, I could not create or select a Databricks cluster, which prevented notebook execution and thus launching SQL analyses on Gold data.

**Solution:** To ensure project progress, a group member with the necessary permissions temporarily delegated access to their Databricks cluster (*esports-cluster*). This allowed me to execute SQL queries on Gold tables and continue 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 12: 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)
ML Model	Random Forest with 87.34% 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.

## APPENDICES

### Databricks Notebooks Source Code

#### setup\_connection.ipynb

```

1 # Notebook: setup_connection
2 import os
3
4 storage_account = "data2lakehousenew"
5 container = "data"
6 storage_account_key = os.environ.get("AZURE_STORAGE_KEY")
7 print(f"Key loaded: {storage_account_key is not None}")
8
9 spark.conf.set(
10     f"fs.azure.account.key.{storage_account}.dfs.core.windows.net",
11     storage_account_key)
12
13 BRONZE = f"abfss://{{container}}@{{storage_account}}.dfs.core.windows.net/bronze"
14 SILVER = f"abfss://{{container}}@{{storage_account}}.dfs.core.windows.net/silver"
15 GOLD = f"abfss://{{container}}@{{storage_account}}.dfs.core.windows.net/gold"
16
17 print("== Files in Bronze Layer ==")
18 for f in dbutils.fs.ls(BRONZE):
19     print(f"  {f.name} ({f.size/1024/1024:.2f} MB)")

```

Code 1: Azure ADLS Gen2 connection test

#### 01\_bronze\_to\_silver.ipynb (Extracts)

```

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

```

Code 2: Data loading and cleaning

#### 02\_silver\_to\_gold.ipynb (Extracts)

```

1 df_player_stats = df_players \
2     .filter(col("account_id").isNotNull()) \
3     .groupBy("account_id").agg(
4         count("match_id").alias("total_matches"),
5         sum("win").alias("total_wins"),
6         round(avg("kills"), 2).alias("avg_kills"),
7         round(avg("kda_calculated"), 2).alias("avg_kda"),
8         round(avg("gold_per_min"), 2).alias("avg_gpm")) \
9         .withColumn("win_rate",
10             round(col("total_wins")/col("total_matches")*100, 2))
11 total = df_matches.count()

```

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

Code 3: Gold tables creation

## Streaming Simulator Source Code

```

1 import time, pandas as pd, os
2 from datetime import datetime
3 from azure.storage.filedatalake import DataLakeServiceClient
4
5 STORAGE_ACCOUNT = "dota2lakehousenew"
6 STORAGE_KEY = os.getenv("AZURE_STORAGE_ACCOUNT_KEY")
7 CONTAINER = "data"
8 DIRECTORY = "bronze"
9 SOURCE_FILE = "main_metadata.csv"
10
11 def get_client():
12     url = f"https://{{STORAGE_ACCOUNT}}.dfs.core.windows.net"
13     return DataLakeServiceClient(account_url=url, credential=STORAGE_KEY)
14
15 def stream_data():
16     print("--- Starting Streaming Simulation ---")
17     df = pd.read_csv(SOURCE_FILE)
18     print(f"Loaded {len(df)} rows.")
19
20     client = get_client()
21     fs_client = client.get_file_system_client(CONTAINER)
22     dir_client = fs_client.get_directory_client(DIRECTORY)
23
24     BATCH_SIZE, SLEEP_TIME = 5, 3
25
26     for i in range(0, 50, BATCH_SIZE):
27         chunk = df.iloc[i:i+BATCH_SIZE]
28         json_data = chunk.to_json(orient='records')
29
30         ts = datetime.now().strftime("%Y%m%d_%H%M%S")
31         fname = f"raw_matches_{ts}.json"
32
33         fc = dir_client.get_file_client(fname)
34         fc.create_file()
35         fc.append_data(data=json_data, offset=0, length=len(json_data))
36         fc.flush_data(len(json_data))
37         print(f"Uploaded {fname}")
38
39         time.sleep(SLEEP_TIME)
40
41     print("--- Simulation Complete ---")
42
43 if __name__ == "__main__":
44     stream_data()

```

Code 4: stream\_simulator.py

## Table Creation SQL Commands

```
1 CREATE DATABASE IF NOT EXISTS esports_gold;
2 CREATE TABLE IF NOT EXISTS esports_gold.player_stats
3 USING DELTA LOCATION
4   'abfss://data@dota2lakehousenew.dfs.core.windows.net/gold/player_stats';
5 CREATE TABLE IF NOT EXISTS esports_gold.hero_stats
6 USING DELTA LOCATION
7   'abfss://data@dota2lakehousenew.dfs.core.windows.net/gold/hero_stats';
8 CREATE TABLE IF NOT EXISTS esports_gold.daily_stats
9 USING DELTA LOCATION
10  'abfss://data@dota2lakehousenew.dfs.core.windows.net/gold/daily_stats';
11 CREATE TABLE IF NOT EXISTS esports_gold.ml_features
12 USING DELTA LOCATION
13  'abfss://data@dota2lakehousenew.dfs.core.windows.net/gold/ml_features';
14 SHOW TABLES IN esports_gold;
```

Code 5: Creation of schema and Gold tables

## ML Model Source Code

```
1 import mlflow
2 from pyspark.ml.feature import VectorAssembler
3 from pyspark.ml.classification import RandomForestClassifier
4 from pyspark.ml.evaluation import MulticlassClassificationEvaluator
5 from pyspark.ml import Pipeline
6 path = f"abfss://{{container}}@{{storage_account}}.dfs.core.windows.net/gold/ml_features"
7 df = spark.read.format("delta").load(path)
8 cols = ["kills", "deaths", "assists", "gold_per_min", "xp_per_min",
9         "hero_damage", "tower_damage", "last_hits", "level",
10        "duration_minutes", "win"]
11 data = df.select(cols).dropna()
12 train, test = data.randomSplit([0.8, 0.2], seed=42)
13 features = [c for c in cols if c != "win"]
14 assembler = VectorAssembler(inputCols=features, outputCol="features")
15 rf = RandomForestClassifier(
16     labelCol="win", featuresCol="features", numTrees=50, maxDepth=10)
17 pipeline = Pipeline(stages=[assembler, rf])
18 mlflow.autolog()
19 model = pipeline.fit(train)
20 predictions = model.transform(test)
21 evaluator = MulticlassClassificationEvaluator(
22     labelCol="win", predictionCol="prediction", metricName="accuracy")
23 acc = evaluator.evaluate(predictions)
24 print(f"Accuracy: {acc*100:.2f}%")
25
26 predictions.groupBy("win", "prediction").count().show()
27 import pandas as pd
28 imp = model.stages[-1].featureImportances.toArray()
29 df_imp = pd.DataFrame({"Feature": features, "Importance": imp})
30 print(df_imp.sort_values("Importance", ascending=False))
```

Code 6: Complete Machine Learning pipeline

## Technologies Used

Table 13: 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	-