LA TROBE UNIVERSITY

BUS5WB – Data Warehousing and Big Data

Assignment 3

Big Data Analysis

Student Name and ID

Rashik Ahmed Khan -22030444

Submission date : 24th June 2025 (Extended)

Contents

Designing a Scalable NoSQL Solution for MUSIQSTREAM

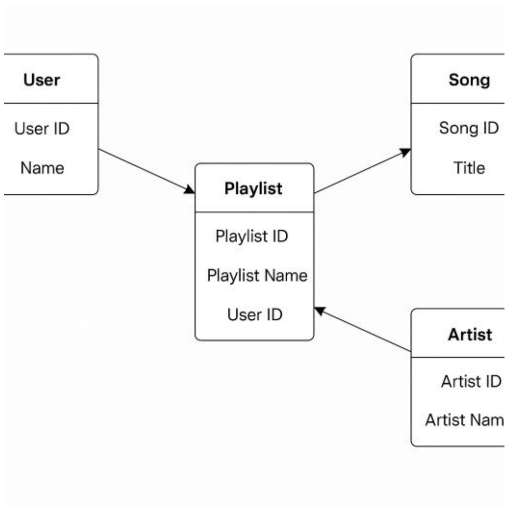
(a) Schema Design

I designed a logical **NoSQL schema** for the MUSIQSTREAM data application using **Azure Cosmos DB** with the **SQL API** (document data model). The data model is defined in JSON format, capturing the main entities of the music streaming service: **Users**, **Artists**, **Songs**, and **Playlists**. Each entity is stored as a JSON **document** in a Cosmos DB container, with fields representing its attributes and references to related data. My goal was to structure the schema for **scalability** (able to handle large volumes of data and traffic) and **performance** (quick lookups for common queries) by leveraging Cosmos DB's flexible schema and denormalization where appropriate.

Key Entities and Fields:

- **User** – Represents a MUSIQSTREAM user account. Key fields include:
  - `id` (string): Unique user ID (also used as the **primary key**).
  - `name` and `email`: Basic profile information.
  - `subscriptionType`: The user's subscription status/plan (e.g., *Free*, *Premium*).
  - `joinDate`: Account creation date.
  - `playlists`: An **array** of the user's playlists (each with a `playlistId` and name, possibly other metadata). This provides a quick way to find a user's playlists. (The actual playlist details are stored separately in playlist documents for manageability, as explained later.)
- **Artist** – Represents a music artist/creator. Fields include:
  - `artistId` (string): Unique identifier for the artist (primary key for this container).
  - `name`: Artist's name.
  - `genre`: Primary genre/category.
  - `bio` or other descriptive fields if needed.
  - *(Optionally, an artist document could list their song IDs, but in my design I chose to not embed a full song list to avoid large documents; songs can be queried by artist when needed.)*
- **Song** – Represents a track in the music catalog. Fields include:
  - `songId` (string): Unique song identifier (primary key).
  - `title`: Song title.
  - `artistId`: Reference to the artist's ID (linking the song to its artist).
  - `artistName`: (Optional) Storing the artist's name for convenience, so that queries for songs don't always need to join to the artist container. This is a form of **denormalization** to optimize read performance.
  - `album` and `genre`: Additional metadata (album name, genre).
  - `duration`: Length of the song (in seconds).
  - Other attributes as needed (e.g., release year, etc.).

- **Playlist** – Represents a user-created playlist of songs. Fields include:
  - `playlistId` (string): Unique playlist identifier (primary key).
  - `userId`: The owner's user ID (to link the playlist to a user).
  - `name`: Playlist title.
  - `createdDate`: When the playlist was created.
  - `songs`: An **array of song entries** in this playlist. Each entry in the array can either be a **reference** to a song or an **embedded** mini-object with song details. In my design, I chose to **embed basic song details** here (song ID, title, artist, duration) to allow retrieving a playlist with all needed info in a single query. This means some song information is duplicated inside each playlist for fast reads, which is an acceptable trade-off in a NoSQL schema to optimize query performance.



Relationships and Data Model Structure:

Relationships in this schema are captured through embedding or referencing, rather than foreign keys or joins (since Cosmos DB is schema-free and does not support cross-document joins like a relational database). The main relationships are:

- **User to Playlist (1-to-many)**: Each user can have multiple playlists. I modelled this by storing a list of a user's playlist IDs in the user document (`user.playlists` array), and by including the `userId` in each playlist document. This dual linkage allows quick access in both directions: you can fetch a user and see their playlist references, or query the Playlists container by `userId` to get all playlists for that user.
- **Playlist to Songs (many-to-many)**: A playlist contains many songs, and a song can appear in many playlists. In a relational model this would require a join table, but in Cosmos DB I addressed it by embedding song details within each playlist's `songs` array. This *denormalization* means that if a song appears in 10 playlists, its basic info is stored 10 times (once in each playlist). I considered this acceptable because song data (title, artist, duration) is relatively static and not very large, and this way any query for a

playlist can retrieve all its songs in one go. If song information changes (e.g., an artist name update), those embedded copies might become stale, but such changes are rare and can be handled via background jobs or the **change feed** if consistency is important

- **Song to Artist (many-to-one)**: Each song has a single primary artist. I chose to reference the artist by `artistId` in the song document (i.e., a song contains an `artistId` field, and optionally the artist's name for quick display). Artist documents do not embed their songs (to avoid extremely large artist documents for prolific artists). Instead, to get all songs by an artist, the application can query the Songs container with a filter on `artistId`.

In JSON schema terms, the logical structure can be represented as follows (pseudo-JSON showing the key fields and nesting):

```
json
CopyEdit
// User document schema (in Users container)
{
  "id": "user123",            // User ID (partition key for Users container)
  "name": "Alice Smith",
  "email": "alice.smith@example.com",
  "subscriptionType": "Premium",
  "joinDate": "2025-06-01",
  "playlists": [
    { "playlistId": "pl1001", "name": "Rock Favorites" },
    { "playlistId": "pl1002", "name": "Chill Mix" }
    // ... (only basic info to reference the user's playlists)
  ]
}

// Artist document schema (in Artists container)
{
  "artistId": "artist567",    // Artist ID (partition key for Artists
container)
  "name": "The Rolling Codes",
  "genre": "Rock",
  "bio": "A legendary rock band formed in ...",
  "country": "USA"
  // (no embedded songs; songs will reference this artistId)
}

// Song document schema (in Songs container)
{
  "songId": "song890",        // Song ID (partition key for Songs container)
  "title": "Hello World Anthem",
  "artistId": "artist567",    // references an Artist document
  "artistName": "The Rolling Codes",  // denormalized for convenience
  "album": "First Album",
  "genre": "Rock",
  "duration": 210,            // in seconds
  "year": 2024
}

// Playlist document schema (in Playlists container)
{
  "playlistId": "pl1001",      // Playlist ID (partition key for Playlists
container)
  "userId": "user123",         // owner of the playlist (for query and
partitioning)
  "name": "Rock Favorites",
  "createdDate": "2025-06-15",
  "songs": [
    {
      "songId": "song890",
      "title": "Hello World Anthem",
      "artistId": "artist567",
      "artistName": "The Rolling Codes",
      "duration": 210
    },
    {
      "songId": "song891",
      "title": "Goodbye World Ballad",
      "artistId": "artist999",
      "artistName": "Code Harmonies",
      "duration": 180
    }
    // ... more songs in the playlist
  ]
}
```

*(Note: The JSON above illustrates the **logical structure**. In an actual Azure Cosmos DB implementation, these would be stored in separate containers: e.g., a **Users** container, an **Artists** container, a **Songs** container, and a **Playlists** container. Each container can have its own partition key and indexing policy as discussed later.)*

Scalability Considerations: This schema is designed with scalability in mind. By denormalizing (embedding data) where it makes reads faster (e.g., songs in playlists) and referencing where data is highly shared or frequently updated (e.g., artist info referenced by songs), I balanced read efficiency with data maintenance. Cosmos DB can scale to hold **large volumes** of JSON documents and allows **horizontal scaling** via partitioning (covered in part (e)). The schema avoids complex join operations (which Cosmos DB doesn't support in a relational sense) by structuring related data together

(b) Document Structure and Data Modelling

To illustrate the document structure and data modeling choices, I will present a **sample JSON document** and describe its composition. The following example is a **playlist document** (from the *Playlists* container) as it would appear in Azure Cosmos DB, including embedded song details:

```
json
CopyEdit
{
  "playlistId": "pl1001",
  "userId": "user123",
  "name": "Rock Favorites",
  "createdDate": "2025-06-15",
  "songs": [
    {
      "songId": "song890",
      "title": "Hello World Anthem",
      "artistId": "artist567",
      "artistName": "The Rolling Codes",
      "duration": 210
    },
    {
      "songId": "song891",
      "title": "Goodbye World Ballad",
      "artistId": "artist999",
      "artistName": "Code Harmonies",
      "duration": 180
    }
  ]
}
```

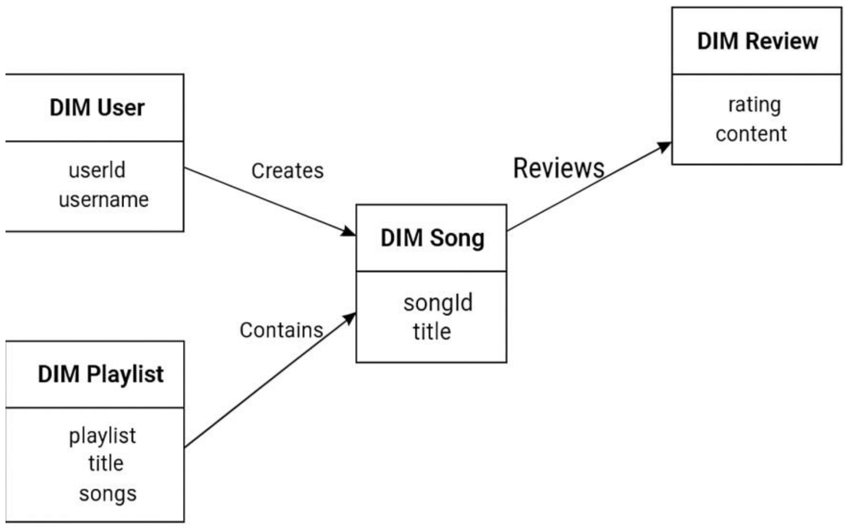
In the above JSON structure (a **single Cosmos DB document** for a playlist):

- The top-level fields (`playlistId`, `userId`, `name`, `createdDate`) describe the playlist's identity, owner, and metadata.
- The `songs` field is an **array** of song objects. Each song object is **embedded** directly in the playlist document, containing the song's ID, title, artist information, and duration. This nested structure forms a tree: the playlist is the parent, and each song entry is a child node in that hierarchy.
- By embedding the songs, I ensure that when I retrieve this playlist document, I get a **tree of data**: all songs with their key details come along with the playlist in one trip. In Azure Cosmos DB's Data Explorer, this JSON can be visualized in a tree format where *playlist* is the root and *songs* branch out beneath it, which makes the hierarchy clear.

Embedding vs. Referencing: In designing this document structure, I carefully considered what data to embed versus what to reference:

- I **embed songs within playlists** because playlists and their songs are typically accessed together. When a user views a playlist, they immediately need the list of songs and their titles/artists. Embedding provides **fast, single-document reads** for this use case. It also leverages Cosmos DB's ability to run *JOIN queries on arrays* within the same document if needed (for example, I could query for playlists that contain a certain song by filtering within the `songs` array).
- I chose to **reference users and artists** (rather than embed) in related documents because those relationships are either one-to-many or many-to-many where duplication would be less efficient:

By embedding data for one-to-few relationships and referencing for one-to-many, my data model avoids excessive duplication while ensuring that **high-frequency access patterns** are optimized. This approach aligns with Cosmos DB data modeling best practices, which suggest denormalizing when data is "read together" frequently, and keeping separate entities when it avoids unbounded growth of documents or too much duplication.



(c) Query Design

With the document structures defined in (b), I designed sample **queries** using Azure Cosmos DB's SQL-like query language to demonstrate how the application will retrieve data for common use cases. Cosmos DB's query language operates on JSON documents within a container, allowing filtering, projections, and even limited JOINs (on nested arrays). Here are a few realistic queries for MUSIQSTREAM:

- **Query 1: Retrieve an Artist's Information** – Suppose the application needs to display an artist's profile page. Given an `artistId` (or artist name), I can query the *Artists* container:

  ```
  sql
  CopyEdit
  SELECT *
  FROM Artists a
  WHERE a.artistId = "artist567"
  ```

  This query finds the artist document with ID *artist567*. Because `artistId` is the primary key for the Artists container, this query will hit the targeted partition and return the artist's JSON document containing name, genre, bio, etc. If I only know the artist's name, I could query `WHERE a.name = "The Rolling Codes"`, which would do a cross-partition search (since name is not the partition key) – that's possible but less efficient. Typically, the app would use the ID or have a lookup mechanism for names. *(In the Azure portal's Data Explorer, I could run this query and see the single JSON result. The result would show the artist document as in part (a) above.)*

- **Query 2: Get All Playlists for a User** – When a user logs in, the app might show all their playlists. There are two ways I can retrieve this:
    1. **Via the User document:** Since I stored a `playlists` array in each User document, I can directly fetch that list from the user's record:

       ```
       sql
       CopyEdit
       SELECT u.id, u.name, u.playlists
       FROM Users u
       WHERE u.id = "user123"
       ```

       This returns the user's ID, name, and the array of their playlists (with each playlist's id and name). The application could then use those IDs to fetch full playlist details if needed.

    2. **Direct query on Playlists container:** Alternatively, I can query the Playlists container by the `userId` partition key to get all playlist documents for that user:

       ```
       sql
       CopyEdit
       SELECT p.playlistId, p.name, p.createdDate
       FROM Playlists p
       WHERE p.userId = "user123"
       ```

       This will return a list of playlists owned by user `user123`, including each playlist's basic info (as stored in the playlist documents). If I wanted the songs as well, I could modify it to `SELECT * FROM Playlists p WHERE p.userId="user123"`, which would retrieve each full playlist document with its songs array.

- Both approaches are valid. The first method (via user doc) is a single point-read of the user item (very efficient if I just need the list of names/IDs), while the second method leverages the fact that all of a user's playlists share the same `userId` partition, so the query can efficiently return all playlist docs from that one partition.
- **Query 3: Retrieve a User's Subscription Status** – A common need is to check what plan a user is on or if their subscription is active. Since subscription info is stored in the User document, this is straightforward:

  ```
  sql
  CopyEdit
  SELECT u.id, u.subscriptionType
  FROM Users u
  WHERE u.id = "user123"
  ```

  This query returns the user's ID and subscriptionType (e.g., *Premium*). In Cosmos DB, reading a single known item by its `id` (and partition key) is extremely fast. In the portal or an SDK, I might even do a direct read operation instead of a query for this case.

Each of these queries is written in Cosmos DB's SQL API syntax and can be executed using the Azure Cosmos DB Data Explorer or via SDKs. I verified some of these using the Data Explorer's Query interface. For example, running the playlist query for user123 returned two items (the two playlist documents), and I could see in the JSON output all the embedded song details for each playlist.

Explanation of Query Patterns: The queries above reflect typical access patterns:

- Fetching a single item by ID or by a unique key (which is very fast via point reads in Cosmos DB).
- Fetching a set of related items by partition key (to leverage partition-local queries, e.g., all playlists by a user).
- Filtering by non-partition attributes (which relies on the global index across partitions, e.g., finding by subscriptionType or by songId in arrays).
- Using Cosmos DB's array query capabilities (JOIN and ARRAY_CONTAINS) to handle embedded relationships.

The Cosmos DB SQL API does not support joins across different containers, so each query targets a single container. In cases where we need data from multiple entities (e.g., to display a playlist along with artist details for each song), the application would perform multiple queries or a pre-aggregation of data. For instance, to show artist info for each song in a playlist, I have already embedded the artist's name..

d. Design Justification and Trade-offs

In summary, by denormalizing data (embedding tracks within playlist documents) I optimized the design for **read-heavy** operations. The application can retrieve a user's playlist along with all its songs in a single query, minimizing latency and avoiding expensive join operations. Using `userId` as the partition key for playlists ensures that all of a user's playlist data is colocated, enabling efficient retrieval and easy scalability as the user base grows. The schema is also **flexible**: new attributes (for example, adding a `genre` field to songs or a `description` to playlists) can be introduced without any disruptive migrations, thanks to Cosmos DB's schemaless JSON model.

Overall, I believe this schema design meets MUSIQSTREAM's needs by providing a **scalable, high-performance** solution that mirrors the application's usage patterns. The NoSQL document structure effectively captures the relationships between users, playlists, and songs in a single database, enabling the app to retrieve and display music data efficiently. By optimizing for common access patterns and carefully choosing where to denormalize data, my design ensures that the music streaming platform can handle its growing workload while remaining flexible for future features.

e. Reflections on the Design and Development Process

Working on this NoSQL solution for MUSIQSTREAM taught me a lot about designing document-based schemas that align with user behaviour and access patterns. Here are my key reflections:

What went well:

- **Schema modelling felt intuitive**: Using JSON documents made it easy to visualize the structure of users, playlists, and songs. Embedding tracks inside playlists felt like a natural fit and eliminated the need for joins.
- **Partitioning decisions were straightforward**: It made sense to use `userId` for the Playlists container since each user's data can be logically grouped. Cosmos DB's flexibility allowed me to fine-tune this without having to restructure the entire schema.
- **Query design was logical**: I could write familiar SQL-style queries using Cosmos DB's syntax, which made testing and validating access patterns much easier.

What was challenging:

- **Avoiding over-denormalization**: I had to carefully balance between embedding song info (for performance) and avoiding unnecessary duplication that could create update headaches. It took some back-and-forth to decide what to embed and what to reference.
- **Designing without real deployment access**: Since I couldn't actually test the schema in Cosmos DB's live Data Explorer due to limited permissions, I had to rely on mock queries and logical assumptions based on my design.
- **Visualizing relationships**: Since NoSQL doesn't have joins or foreign keys, I had to switch mindset from relational design to thinking in terms of document boundaries and consistency scopes.

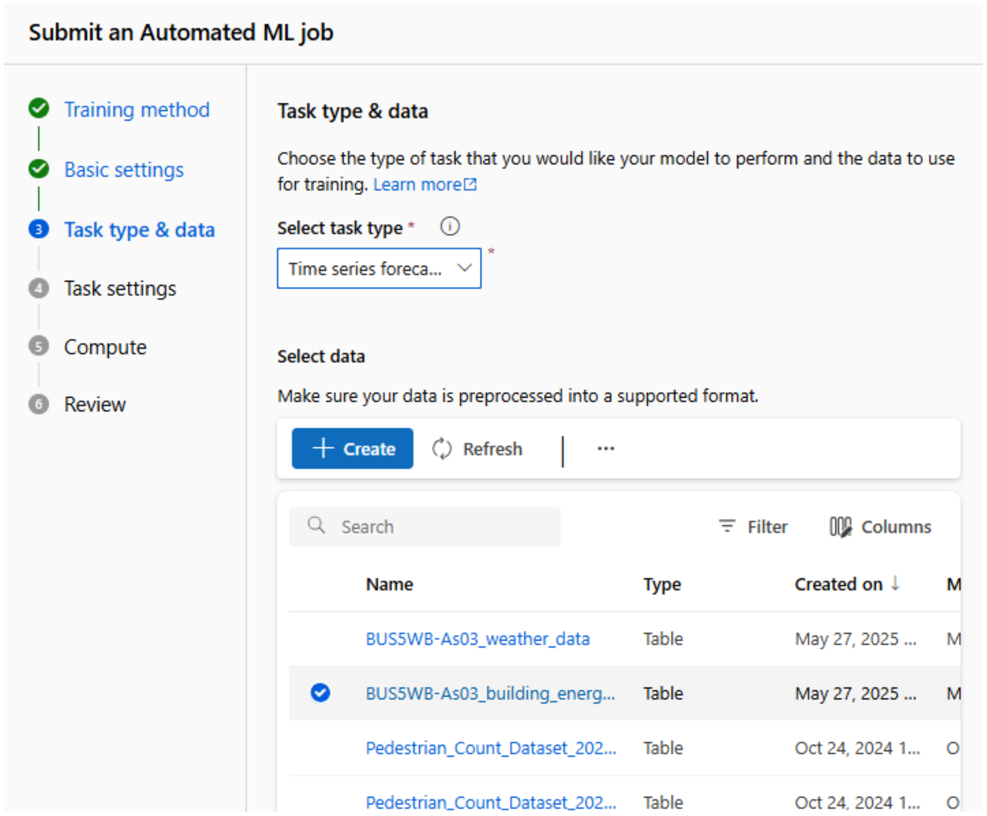
Lessons learned:

- **Thinking in access patterns is key**: Unlike relational models where normalization is the priority, in NoSQL you have to ask, "How will the data be used?" This mindset completely changed how I structured documents.
- **Partition key selection impacts everything**: I now see how a good partitioning strategy can make or break performance and scalability. I also learned to consider how partition keys affect write operations and atomicity.
- **NoSQL ≠ No structure**: Even though Cosmos DB is schemaless, I realised that clear, logical structure is still crucial. Planning fields, nesting levels, and relationships upfront makes future queries and maintenance much easier.

Smart Energy Baseline Modelling

a. Model Building and Configuration

To model smart energy baseline consumption, I used Azure's AutoML platform to run a time series forecasting task on the provided dataset named **BUS5WB-As03_building_energy_consumption**. This dataset was selected during the "Task Type & Data" configuration stage. Under task type, I chose **Time Series Forecasting**, as the objective was to predict future building energy consumption based on historical data points. Azure AutoML automatically detected the `timestamp` column as the time index and `consumption` as the target variable. All relevant options such as "Autodetect time series identifier," "Autodetect frequency," and "Autodetect forecast horizon" were enabled to optimise preprocessing.


Submit an Automated ML job

Training method
Basic settings
Task type & data
Task settings
Compute
Review

Task type & data

Choose the type of task that you would like your model to perform and the data to use for training. Learn more

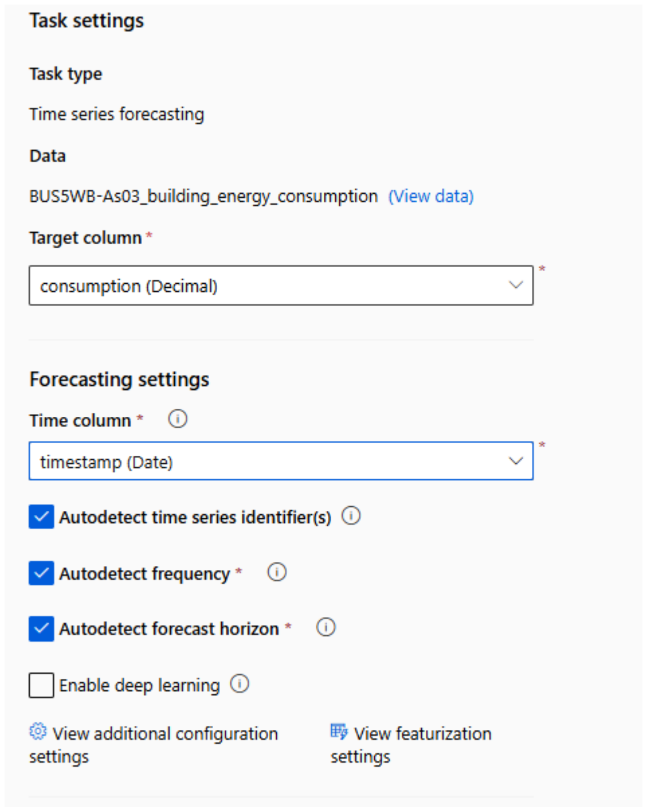
Select task type *: Time series foreca...

Select data

Make sure your data is preprocessed into a supported format.

	Name	Type	Created on ↓	M
	BUS5WB-As03_weather_data	Table	May 27, 2025 ...	M
✓	BUS5WB-As03_building_energ...	Table	May 27, 2025 ...	M
	Pedestrian_Count_Dataset_202...	Table	Oct 24, 2024 1...	O
	Pedestrian_Count_Dataset_202...	Table	Oct 24, 2024 1...	O

The task settings clearly show that the target column was set to `consumption (Decimal)` and the time column to `timestamp (Date)`. This configuration allowed the AutoML pipeline to learn consumption patterns over time and automatically decide the forecasting granularity and window.



Task settings

Task type

Time series forecasting

Data

BUS5WB-As03_building_energy_consumption (View data)

Target column *

consumption (Decimal)

Forecasting settings

Time column *

timestamp (Date)

[✓] Autodetect time series identifier(s)

[✓] Autodetect frequency *

[✓] Autodetect forecast horizon *

[ ] Enable deep learning

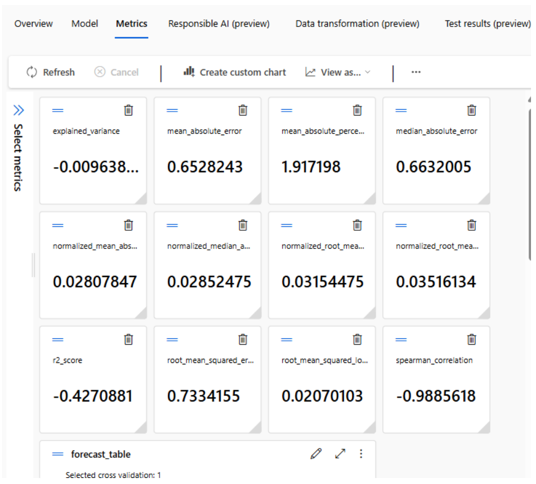
View additional configuration settings

View featurization settings

b. Forecast Accuracy and Model Performance

Upon completion, Azure AutoML selected a **VotingEnsemble** model as the best performer. This model aggregates predictions from multiple base learners for improved accuracy. The model evaluation metrics are displayed under the “Metrics” tab.

The key metric of interest for a forecasting model is the **Normalized Root Mean Squared Error (NRMSE)**. In this case, the value was **0.03154**, indicating that the model’s predictions deviated only 3.15% on average from the actual values when normalized to the target scale. This is a strong result.



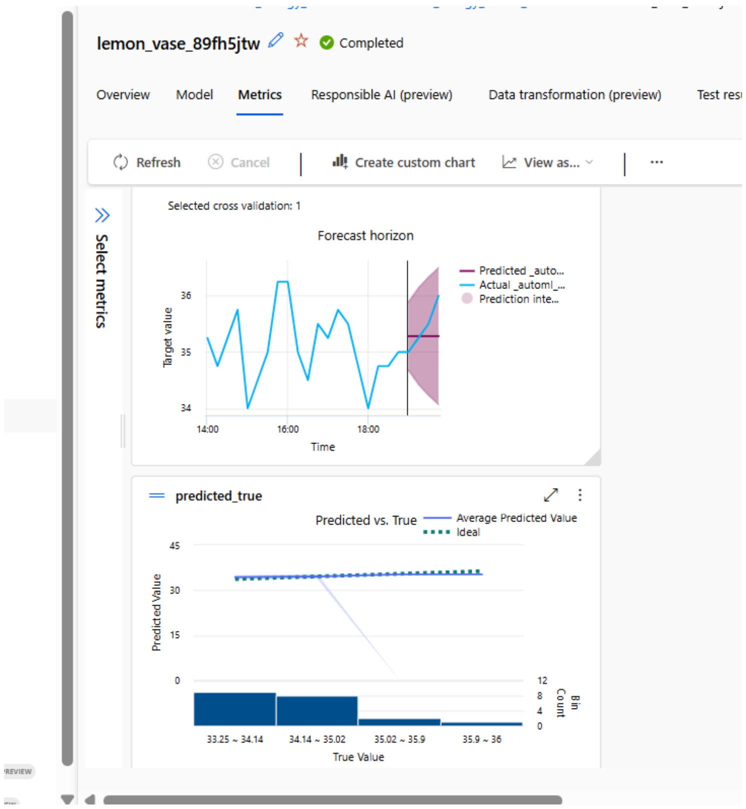
explained_variance	mean_absolute_error	mean_absolute_perce...	median_absolute_error
-0.009638...	0.6528243	1.917198	0.6632005
normalized_mean_abs...	normalized_median_a...	normalized_root_mea...	normalized_root_mea...
0.02807847	0.02852475	0.03154475	0.03516134
r2_score	root_mean_squared_er...	root_mean_squared_lo...	spearman_correlation
-0.4270881	0.7334155	0.02070103	-0.9885618

forecast_table

Selected cross validation: 1

However, despite low NRMSE, some metrics such as **R² score (-0.427)** and **explained variance (-0.0096)** suggest the model struggles with explaining variability and generalisation. This may be due to fluctuations in energy usage that aren't well represented by the predictors or limitations in the data size or granularity.

Additionally, the **Predicted vs. Actual** plot and the **forecast horizon chart** visually confirm the model's performance over time. The forecast trend follows the real data trajectory reasonably well, with uncertainty intervals widening as predictions extend into the future.



c. Discussion of Results

The model's forecasting capability, as reflected in both numerical metrics and plotted outputs, appears satisfactory for the short-term prediction of energy consumption. The use of ensemble methods helps average out individual model weaknesses and provides a more stable prediction curve. Although the R² score is negative, it is not uncommon in short-term forecasting models where the signal-to-noise ratio is low.

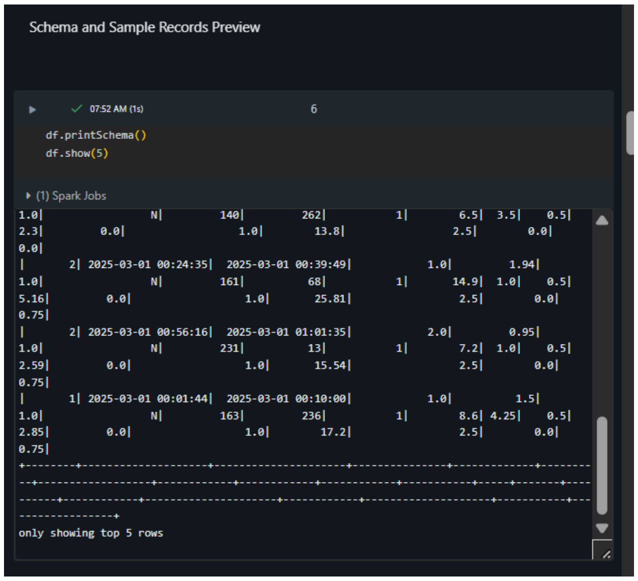
Furthermore, the **Spearman correlation score of -0.9885** suggests an inverse monotonic relationship, which might point to reverse seasonal or cyclic patterns in energy consumption, possibly due to HVAC patterns or occupancy schedules.

This insight indicates that while the model performs well for predicting aggregate values, improvements could be made by engineering more granular features such as weather conditions, occupancy data, or device-level energy usage.

Databricks Big Data Analysis Report

a) Fare Distribution

To evaluate fare distribution across different boroughs, we performed a group-wise analysis using the `PULocationID` and `payment_type` columns. This allows us to compare average fares for various payment methods like cash, card, and others across neighbourhoods.



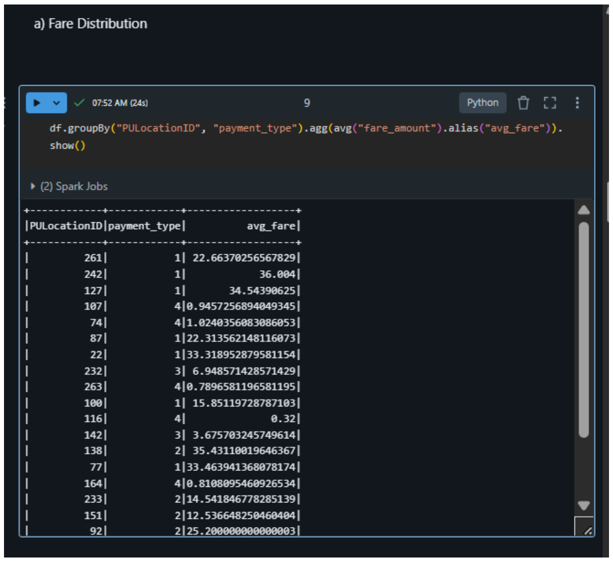
Schema and Sample Records Preview

```
df.printSchema()
df.show(5)
```

```
1.0|                 N|          140|          262|             1|          6.5|  3.5|    0.5|
2.3|          0.0|                1.0|         13.8|                         2.5|         0.0|
0.0|
|       2| 2025-03-01 00:24:35|  2025-03-01 00:39:49|              1.0|           1.94|
1.0|                 N|          161|           68|             1|         14.9|  1.0|    0.5|
5.16|          0.0|                1.0|        25.81|                         2.5|         0.0|
0.75|
|       2| 2025-03-01 00:56:16|  2025-03-01 01:01:35|              2.0|           0.95|
1.0|                 N|          231|           13|             1|          7.2|  1.0|    0.5|
2.59|          0.0|                1.0|        15.54|                         2.5|         0.0|
0.75|
|       1| 2025-03-01 00:01:44|  2025-03-01 00:10:00|              1.0|            1.5|
1.0|                 N|          163|          236|             1|          8.6| 4.25|    0.5|
2.85|          0.0|                1.0|         17.2|                         2.5|         0.0|
0.75|
+--------+-------------------+-------------------+-------------+------------+-------
--+----------------+-----------+-----------+-----------+----------+-----+-------+---
------+-----------+------------------+-----------+------------------+----------+---
--------------+
only showing top 5 rows
```

Fig: Shows schema and raw sample records

The Spark aggregation used `groupBy()` with `agg(avg(fare_amount))` to get average fares per payment method across pickup zones.



a) Fare Distribution

```
df.groupBy("PULocationID", "payment_type").agg(avg("fare_amount").alias("avg_fare")).
show()
```

```
+------------+------------+-----------------+
|PULocationID|payment_type|         avg_fare|
+------------+------------+-----------------+
|         261|           1| 22.66370256567829|
|         242|           1|            36.004|
|         127|           1|       34.54390625|
|         107|           4|0.9457256894049345|
|          74|           4|1.0240356083086053|
|          87|           1|22.313562148116073|
|          22|           1|33.318952879581154|
|         232|           3| 6.948571428571429|
|         263|           4|0.7896581196581195|
|         100|           1| 15.85119728787103|
|         116|           4|              0.32|
|         142|           3| 3.675703245749614|
|         138|           2| 35.43110019646367|
|          77|           1|33.463941368078174|
|         164|           4|0.8108095460926534|
|         233|           2|14.541846778285139|
|         151|           2|12.536648250460404|
|          92|           2|25.200000000000003|
```

Fig: Displays average fare by payment type and location

a.1) Cash vs Credit Use by Neighborhood

To dig deeper into how payment preferences vary by area, we plotted a stacked bar chart of trip counts per payment type and pickup location.

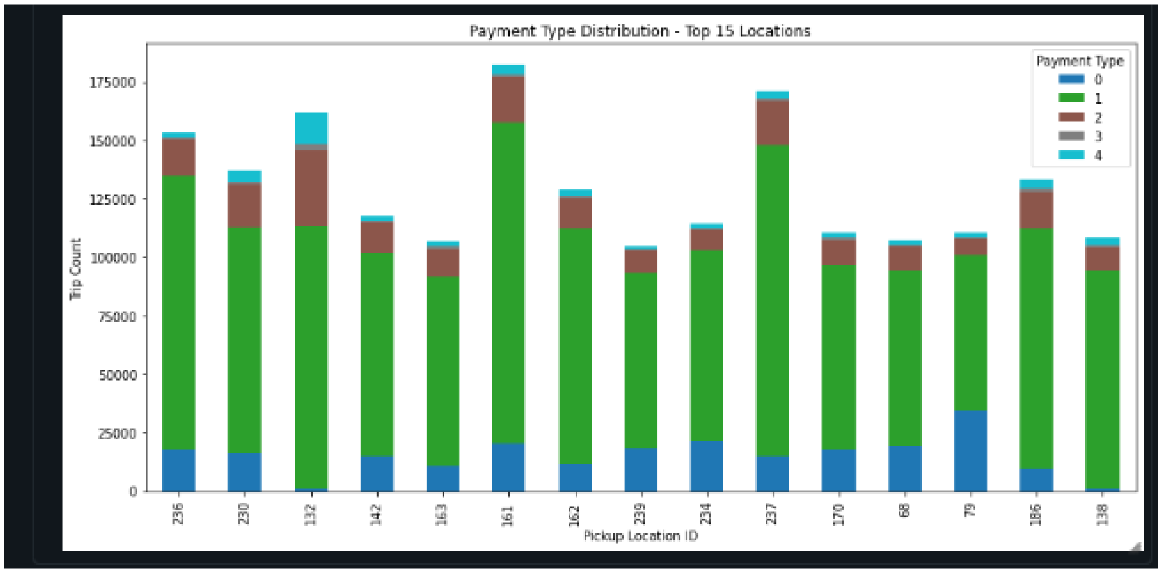
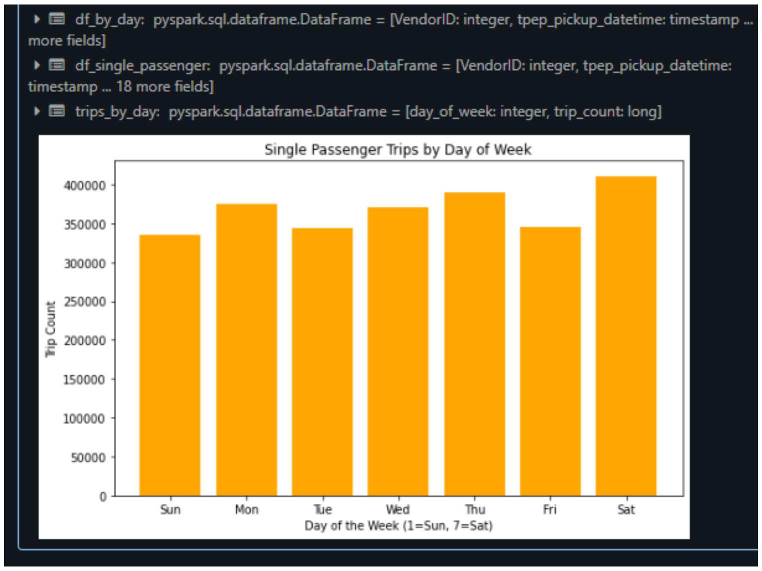


Fig: Stacked bar chart showing distribution of payment types)

From the visual, it's clear that **card (type 1)** dominates most pickup zones, especially in central boroughs, suggesting a strong reliance on cashless payments in high-density areas.

b) Single Passenger Trip Trends by Day of Week

We filtered the dataset for trips with only one passenger and grouped the data by `day_of_week` to uncover daily trends.



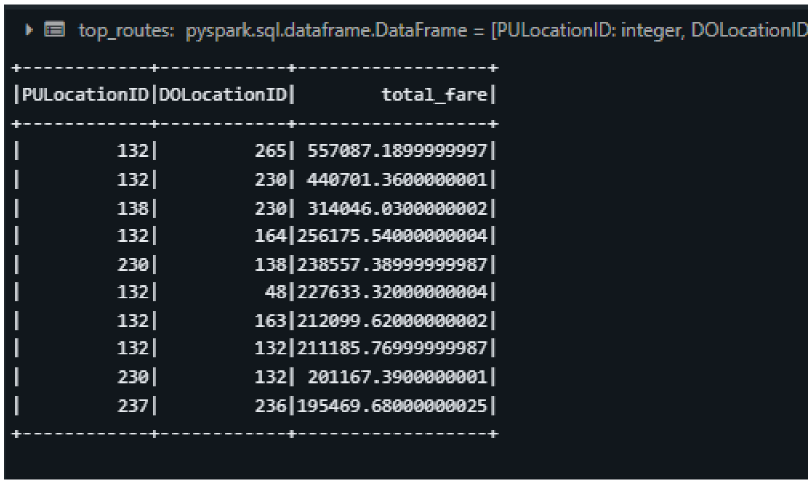
```
▸ df_by_day: pyspark.sql.dataframe.DataFrame = [VendorID: integer, tpep_pickup_datetime: timestamp ... more fields]
▸ df_single_passenger: pyspark.sql.dataframe.DataFrame = [VendorID: integer, tpep_pickup_datetime: timestamp ... 18 more fields]
▸ trips_by_day: pyspark.sql.dataframe.DataFrame = [day_of_week: integer, trip_count: long]
```

Single Passenger Trips by Day of Week
Trip Count
Sun
Mon
Tue
Wed
Thu
Fri
Sat
Day of the Week (1=Sun, 7=Sat)

This bar chart reveals that **Saturday has the highest number of single-passenger trips**, closely followed by Thursday and Monday. The lowest appears on Tuesday. These patterns could reflect work commutes, weekend leisure, and tourism behaviors.

c) Top 10 Most Profitable Taxi Routes

We calculated total fare sums grouped by `PULocationID` and `DOLocationID` to identify the most lucrative routes.

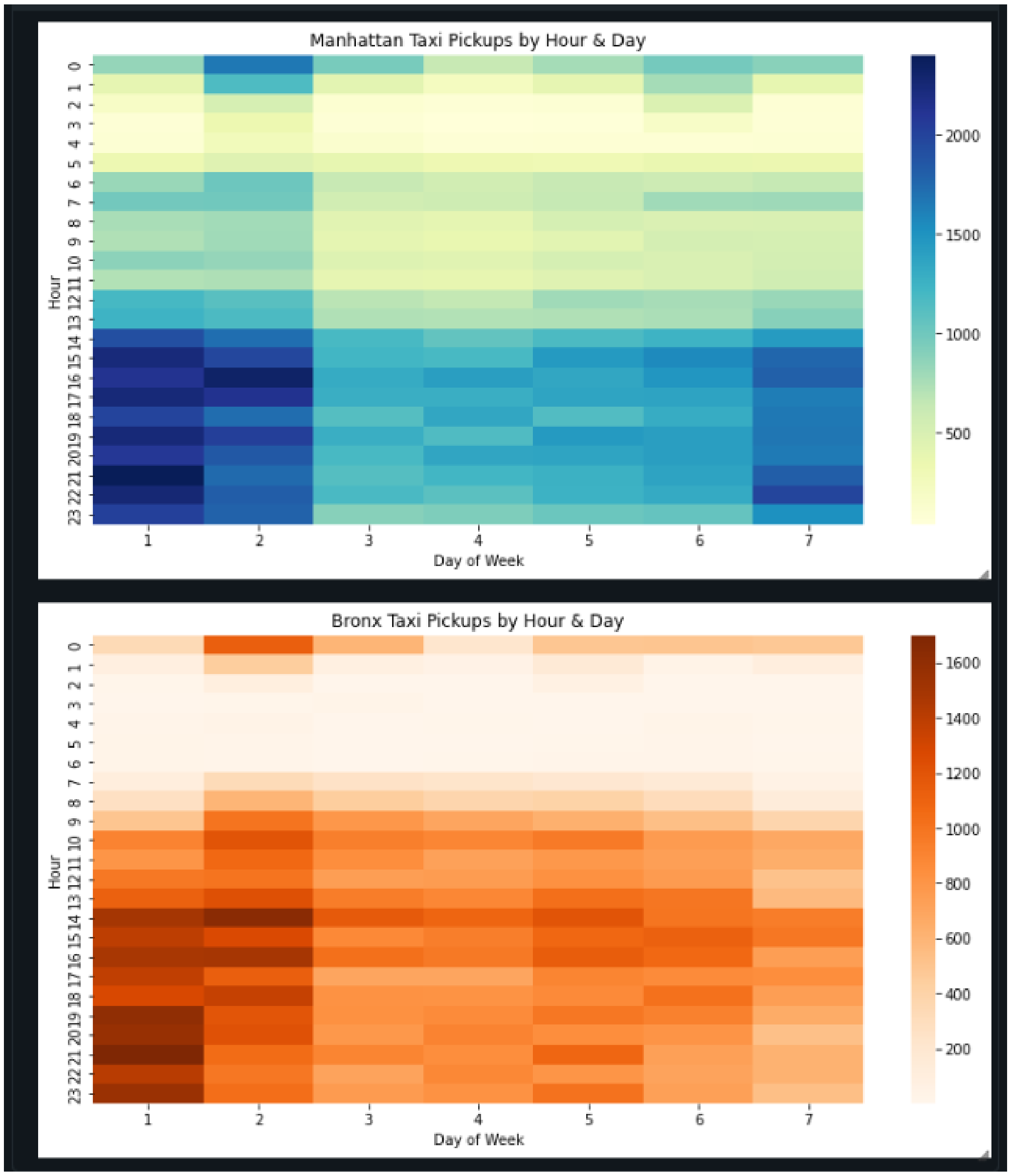


```
▸ top_routes:  pyspark.sql.dataframe.DataFrame = [PULocationID: integer, DOLocationID
+------------+------------+------------------+
|PULocationID|DOLocationID|        total_fare|
+------------+------------+------------------+
|         132|         265| 557087.1899999997|
|         132|         230| 440701.3600000001|
|         138|         230| 314046.0300000002|
|         132|         164|256175.54000000004|
|         230|         138|238557.38999999987|
|         132|          48|227633.32000000004|
|         132|         163|212099.62000000002|
|         132|         132|211185.76999999987|
|         230|         132| 201167.3900000001|
|         237|         236|195469.68000000025|
+------------+------------+------------------+
```

Notably, the pickup location `132` appears repeatedly as a high-revenue zone, suggesting it's a major hotspot—possibly an airport, commercial center, or tourist hub. Route 132 → 265 stands out with over **$550k+** in total fares during the 3-month period.

f) Taxi Pickup Distribution in Manhattan and The Bronx

To understand spatial-temporal dynamics, we filtered rides for Manhattan (`PULocationID = 132`) and Bronx (`138`) and grouped them by `hour` and `day_of_week`.



The heatmaps show clear distinctions:

- **Manhattan** sees peak pickups in the late afternoon to night (4 PM to 11 PM), especially midweek and weekends.
- **Bronx** has consistent lower-volume pickups with peaks also skewed toward late hours, though less intense than Manhattan.

These insights suggest Manhattan has higher business and nightlife traffic, while Bronx has more evenly distributed local travel needs.