Microsoft Fabric – Real-Time Intelligence Project Overview

In this project, we will explore the **key components of Real-Time Intelligence in Microsoft Fabric**, along with a hands-on implementation using a sample bike dataset. The workflow involves the following:

Understanding Core Concepts:

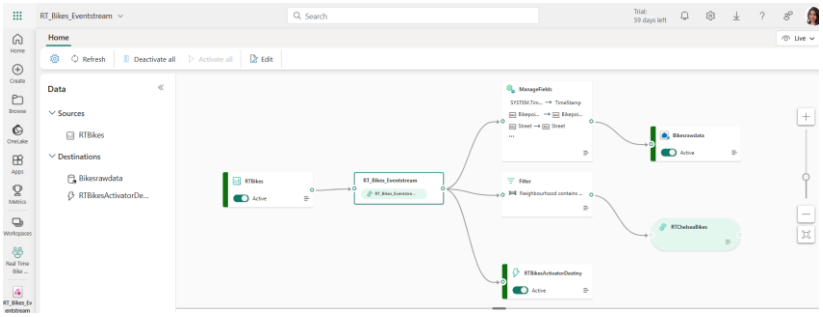
- Introduction to **Kusto Query Language (KQL)**
- Comparison between **Kusto** and **SQL** to highlight their differences in use cases and syntax

Project Workflow:
We will build a real-time analytical solution using a **bike-sharing real-time dataset**, implementing the following steps:

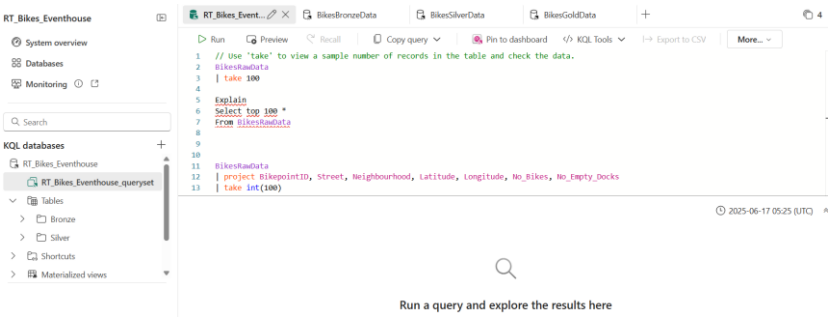
- **Ingestion**: Stream real-time data into **Eventstream**
- **Storage**: Save ingested data into a **KQL Database** as the **Bronze Layer**
- **Transformation**:
  - ✓ Use **Update Policies** and **Functions** to process and promote data into the **Silver Layer**
- **Aggregation**:
  - ✓ Create a **Materialized View** over the **Gold Layer**, stored in the **Eventhouse table**
- **Visualization**:
  - ✓ Build a **Real-Time Dashboard** using aggregated data from the Gold Layer
  - ✓ Additionally, visualize data directly from the Bronze Layer for raw insights
- **Alerting and Automation**:
  - ✓ Set up **alerts** on top of the Eventstream and the Real-Time Dashboard
- **Derived Streams**:
  - ✓ Create a **Derived Stream** from the Eventstream for custom real-time transformations

This end-to-end implementation will demonstrate how Microsoft Fabric's Real-Time Intelligence stack enables seamless streaming, processing, and real-time visualization of live data.

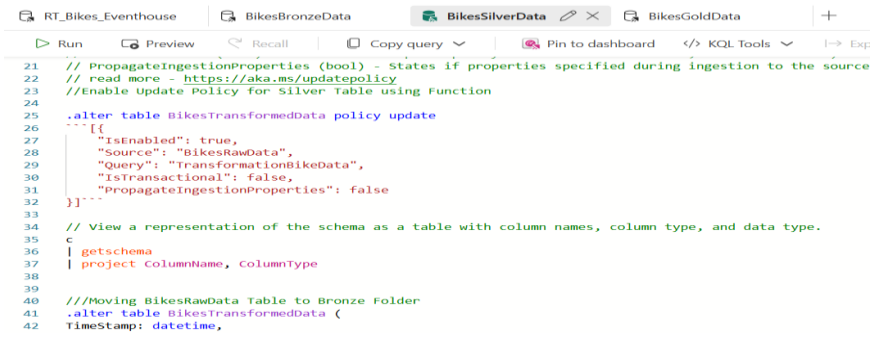
EVENTSTREAM



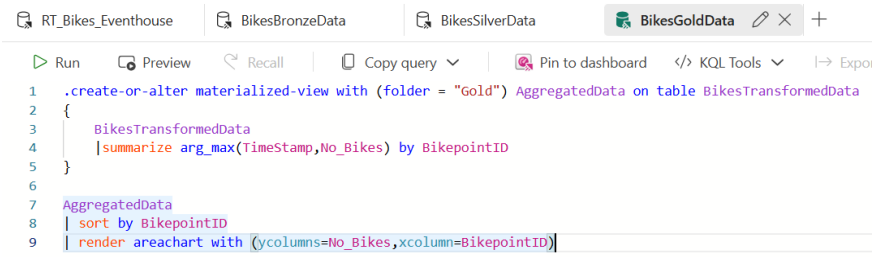
KQLDATABASE



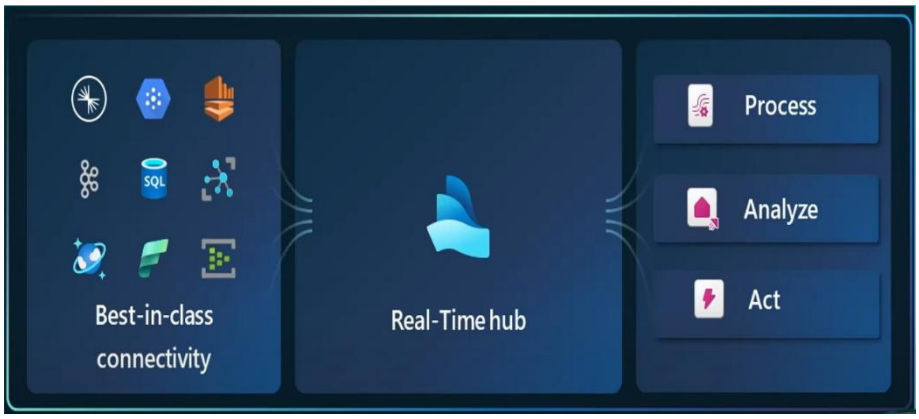
UPDATEPOLICY



MATERIALIZEDVIEW



Real-Time Intelligence Analytics Process



Key Components:

Data Source - Connection

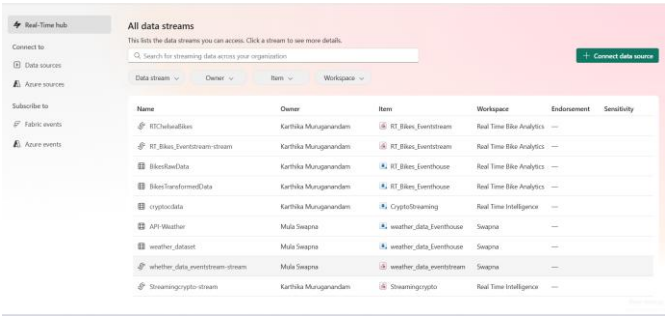
Process - Eventstream

Analyze - Eventhouse, KQL Database, KQL Queryset

Visualize - Real Time Dashboard, Power BI

Act – Activator

- **What is Fabric Real Time Hub?**
  - ✓ Single- Tenant wide(logical) place for all the data in motion



- **Real-Time Data Stream Features:**
  - ✓ Unbounded – which will keep coming perpetually and never ending
  - ✓ Temporal(time-based) - it will also have data records which will be temporal time based indicating when a particular event was recorded or occurred
  - ✓ Aggregations use (temporal windows) – we will be doing aggregations over temporal windows for example, if we are calculating a metric which is showing is rainfall per hour or showing orders which are placed in the last hour or clicks in the last one hour or 15 minutes temporal window basically and
  - ✓ Support real - it will be supporting real time solutions Support real

Microsoft Fabric Real-Time Kusto DB

- **Why not OneLake?**

  - ✓ Onelake is optimized for analytical workloads like data lakes (Delta/Parquet), warehouse and Power BI models.
  - ✓ KQL DB is optimized for streaming workload and high-ingestion telemetry (e.g. IoT, web logs, app traces)
  - ✓ These storage systems are purpose-built and separated by design in Fabric for performance and cost efficiency

- **What is Kusto?**

  - ✓ Kusto is a high-performance columnar database engine built by Microsoft.
  - ✓ It is used for real-time analytics, especially for scenarios like Applications logs, IoT sensor data, Website clickstream, and Security event monitoring.

Kusto vs SQL(T-SQL)

Feature	**KQL (Kusto Query Language)**	**SQL (T-SQL)**
Purpose	High-performance, large-scale data exploration and analytics, particularly for logs, telemetry, and time-series data.	Querying and managing relational data in SQL Server.
Use Cases	Log analytics, telemetry, monitoring, time-series data, real-time analytics.	OLTP (transactional), OLAP (analytical), business intelligence, reporting.
Data Model	Schema-less data, often used for semi-structured or unstructured data like logs.	Structured data with predefined schema (tables, rows, columns).
Query Language	KQL – designed for large-scale data exploration, real-time queries, and logs/telemetry analysis.	T-SQL – procedural SQL with extensions for SQL Server, used for relational queries.
Data Type Support	Supports primitive types like strings, ints, floats, and dynamic types for semi-structured data.	Supports relational data types like INT, VARCHAR, DATE, etc.
Performance & Scalability	Optimized for fast querying on massive datasets (log data, telemetry, etc.) across distributed systems.	Good for transactional (OLTP) workloads but not optimized for massive real-time analytics.
Aggregations	summarize for aggregation, time-series aggregation (e.g., bin ()).	GROUP BY, SUM (), AVG (), COUNT (), and other aggregate functions.
Joins	Limited join support; focuses more on time-series operations. Uses join keyword, but performance may degrade with large datasets.	Fully supports various types of joins (INNER, LEFT, RIGHT, FULL) for relational data operations.

Time-Series Analysis	Strong built-in functions for handling and analysing time-series data (e.g., bin (), range ()).	Time-series analysis is possible but requires more manual effort, not as optimized as KQL.
Syntax	Pipe (`	`) operator to chain commands, easy-to-read syntax for filtering, aggregation, and transformation.
Real-Time Analytics	Built for real-time analytics and telemetry data.	Primarily for transactional systems, not designed for real-time analytics.
Query Optimization	Automatically optimized for large-scale data (logs, telemetry) with parallel execution and distributed computing.	Relies on indexes, query optimization hints, and execution plans.
Storage	Built for handling large datasets with distributed storage across clusters.	Typically runs on a centralized relational database with indexed tables.
Complexity	Query language is simpler for time-series data but can be harder to learn for those familiar with SQL.	Well-known standard SQL language with added procedural capabilities.
Data Sources	Typically used with Azure Data Explorer, logs, telemetry, and time-series databases.	Works with relational databases like SQL Server, Azure SQL Database.

Component Storage and Purpose:

Component	Stores in OneLake?	Purpose
🌀 **Eventstream**	❌ (Not by default)	Ingests real-time data
🏠 **Lakehouse**	✅ (via sink)	Persisted analytics-ready storage
🧠 **KQL Database**	❌ (Not in OneLake)	Streaming queries over real-time data
🚨 **Data Activator**	❌	Triggers real-time alerts/actions

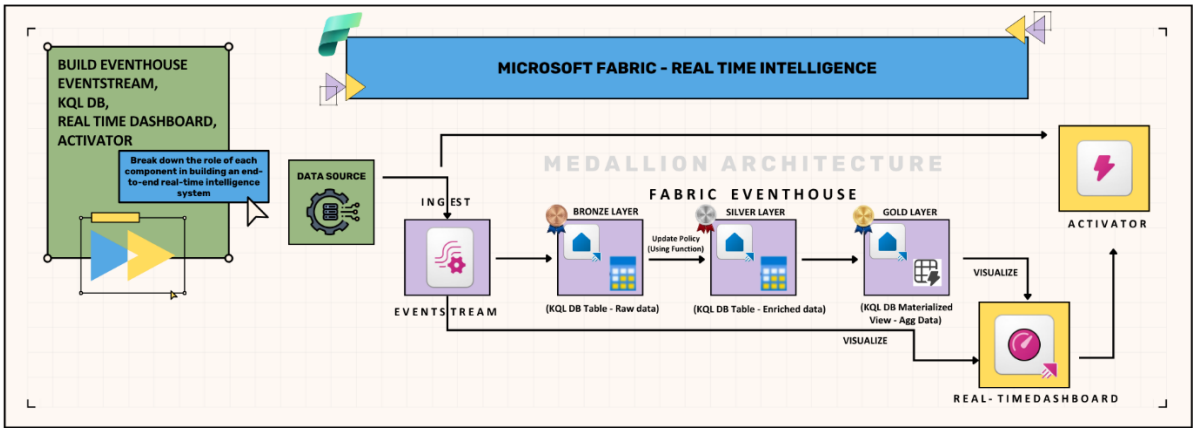
Storage Details and Use Cases:

Component	Stored in OneLake?	Primary Use	Notes
🧠 **KQL Database**	❌	Real-time analytics, telemetry data	Stores in high-performance internal storage

Component	Stored in OneLake?	Primary Use	Notes
Lakehouse	✅	Structured, persisted data	Used for BI and historical analysis

Let's dive into the complete real-time workflow—from live data to real-time intelligence

Architecture:



We are leveraging the real-time Bikes sample dataset available in Microsoft Fabric to build a real-time dashboard and deliver personalized alerts for bike availability.

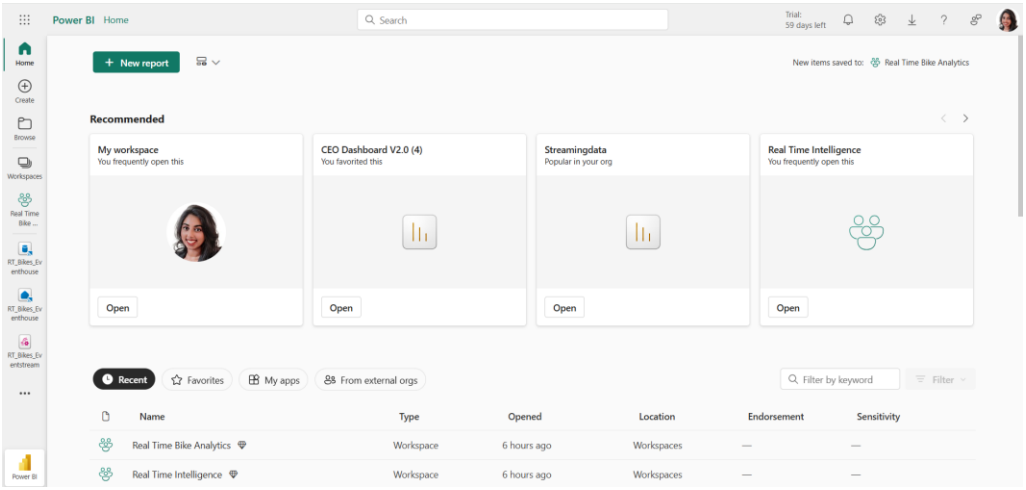
The dataset will be ingested using Fabric Eventstream, and the data will be streamed into Eventhouse and the KQL database. Using a defined function with an update policy, the data will be stored in the Silver layer (KQL database), where it will be enriched and prepared for advanced querying.

From there, a materialized view will be created in the Gold layer to perform necessary aggregations. This curated data will be visualized in a real-time dashboard.
In addition, data can also be visualized directly from the Bronze layer for raw insights.

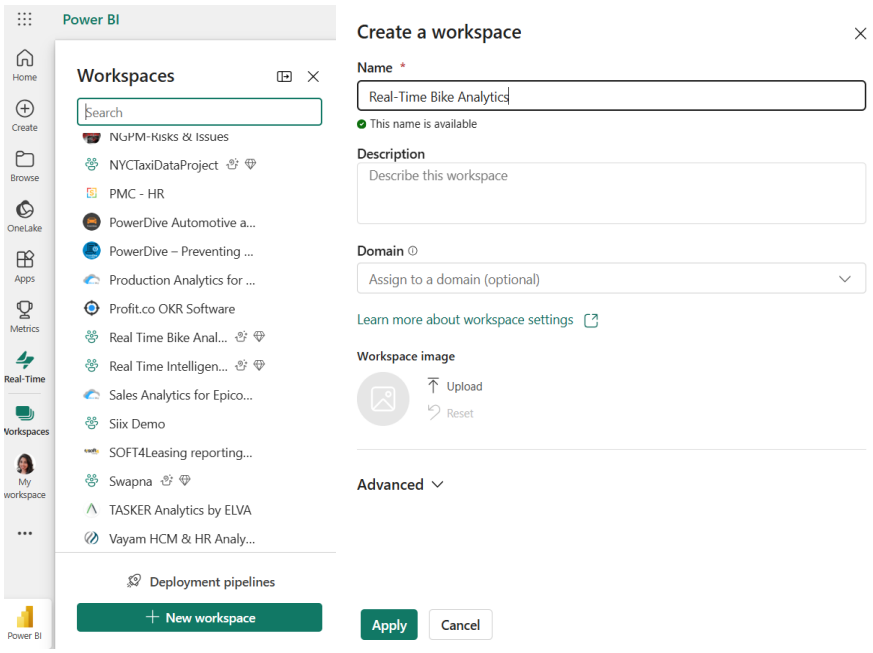
We will implement real-time alerts using Data Activator and configure an alert directly on the Eventstream to trigger immediate actions.

With this setup in place, Let's dive into Microsoft Fabric and bring the solution to life.

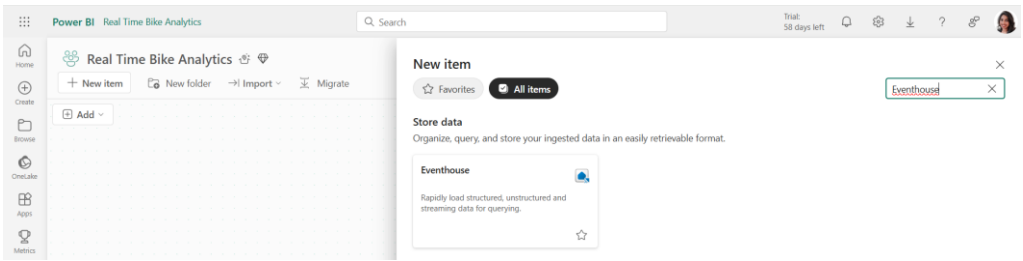
MICROSOFT FABRIC HOME PAGE



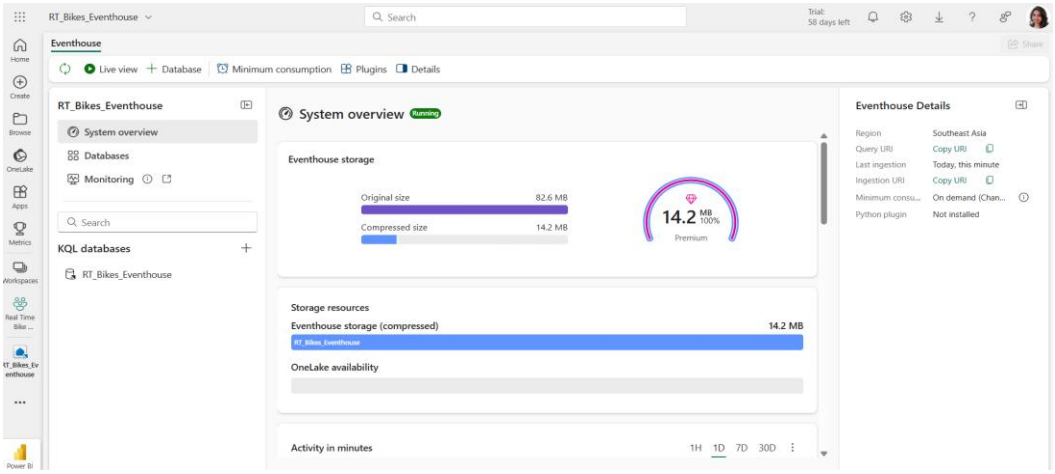
The first step is to set up **a new workspace** for our real-time project workflow.



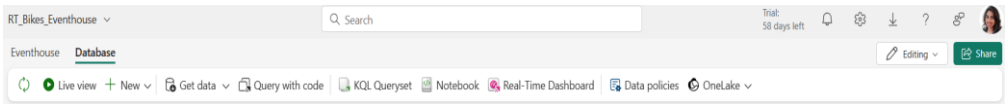
1. Create an **Eventhouse by clicking "New Item"** in the top-left corner of the workspace.



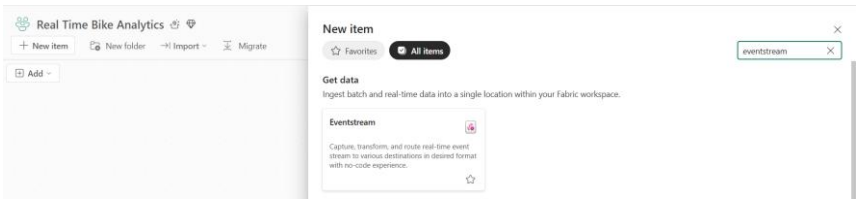
2. Once the **Eventhouse is created**, a corresponding **KQL database** with the same name is automatically generated. This page displays **the KQL database workspace**. You can create additional KQL databases within the Eventhouse if needed. For this example, I've created a database named RT_Bikes_Eventhouse, which provides an overview of the Eventhouse structure.



3. When you click on the KQL Database named RT_Bikes_Eventhouse, the Query Editor page will open. This page provides several options such **as New, Get Data, Notebook, and Real-Time Dashboard.** You can use these tools as needed based on your project workflow.

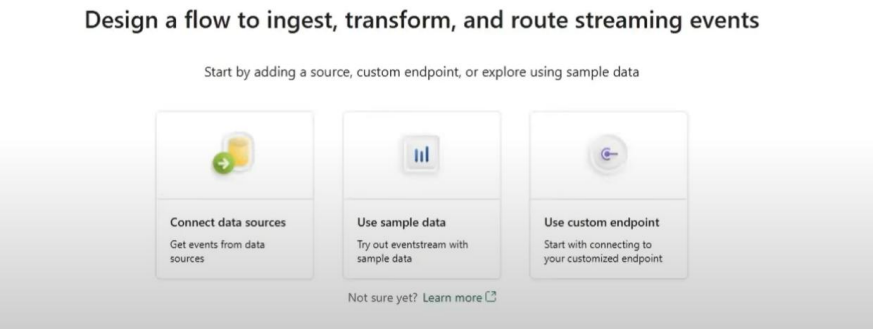


4. Again, we get back to Real Time Bike Analytics workspace which we have created for this project and need to create a new item which as **Eventstream.**



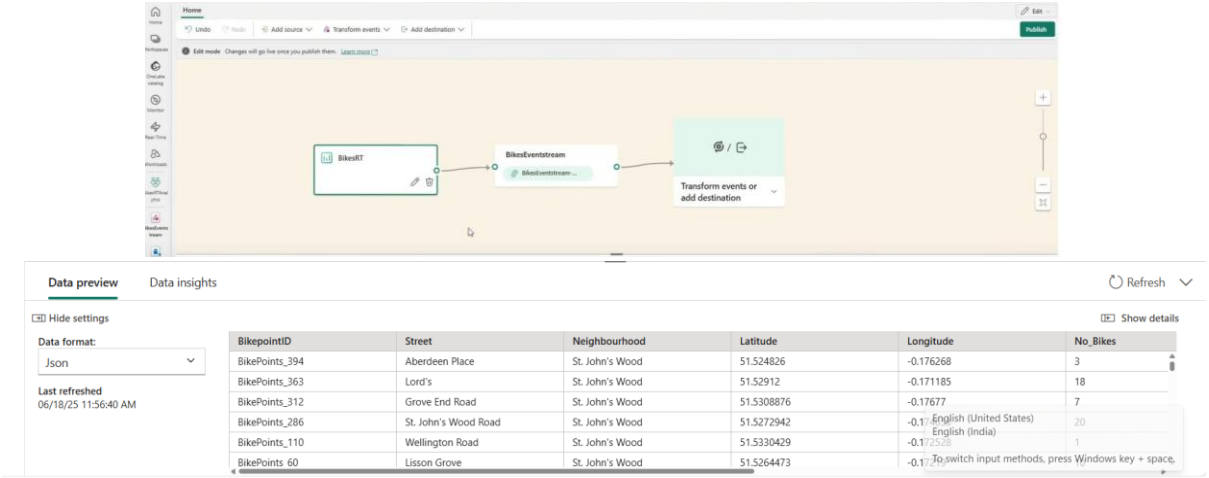
5. Once the Eventstream is created, you'll see a page like the one below. Here, you have three options to add data sources:

- **Connect Data Source** – Enables you to link external streaming sources such as Event Hubs, IoT Hub, or custom APIs to ingest real-time data.
- **Use Sample Data** – Offers built-in sample datasets (e.g., the Bikes dataset) for easy testing, learning, or prototyping without requiring an external connection.
- **Use Custom Point** – Allows you to define and configure a custom streaming input, giving you flexibility to pull data from specific endpoints or custom applications.



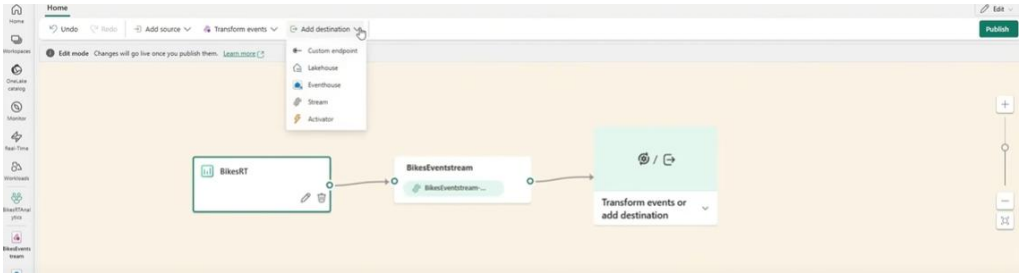
Here we've taken a Sample data which as Bikes Sample Data

6. After selecting the sample data source, we provide a name, choose the **Bicycle** option (which contains the Bikes dataset), and click **Add**. The Eventstream page will then appear as shown below.

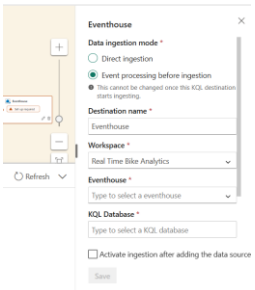


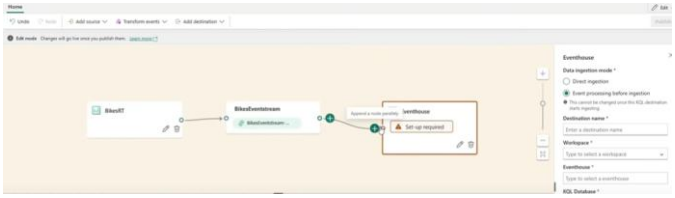
7. After creating a source we can add either transform event and below are the different operations which are available, we can **filter** the data by **filter operation**, **manage the fields** whether we want to select all the fields or add a field or delete a field, **Aggregate** this data by **Aggregate operation**, **Join** it from other data source, We can **Group by**, and we can **Union** which is basically apend.

8. We can add **Destination**, the **destination for eventstream** could be **Lakehouse, Eventhouse,** we can have another **derived stream** from a stream. We can also send a data directly to an **activator** or we can also send the data to a **custom endpoint**.

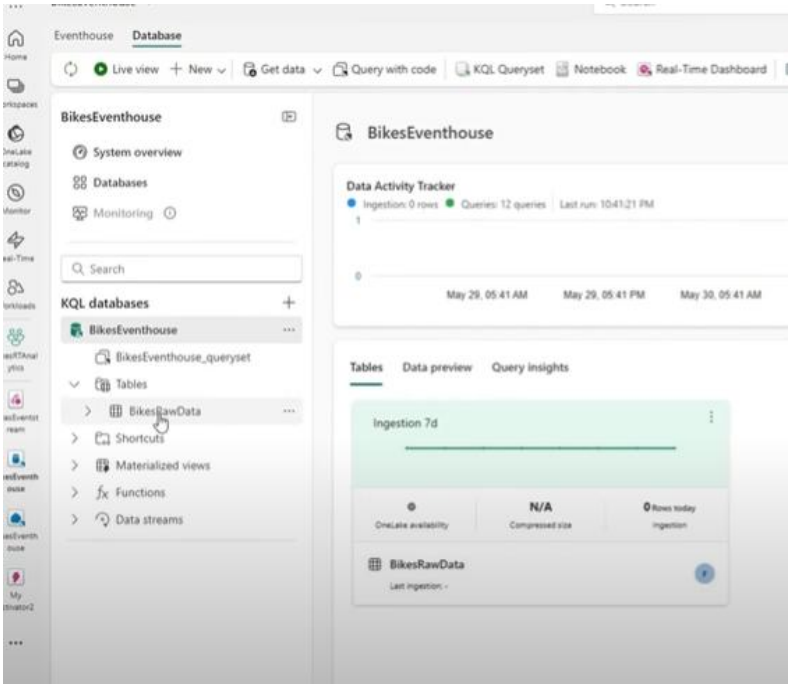


9. In this step, we'll route the data to our Eventhouse by connecting it to the selected data source. After filling in all the required details in the Eventhouse, click the **green "Publish"** button at the top-right corner to complete the setup.

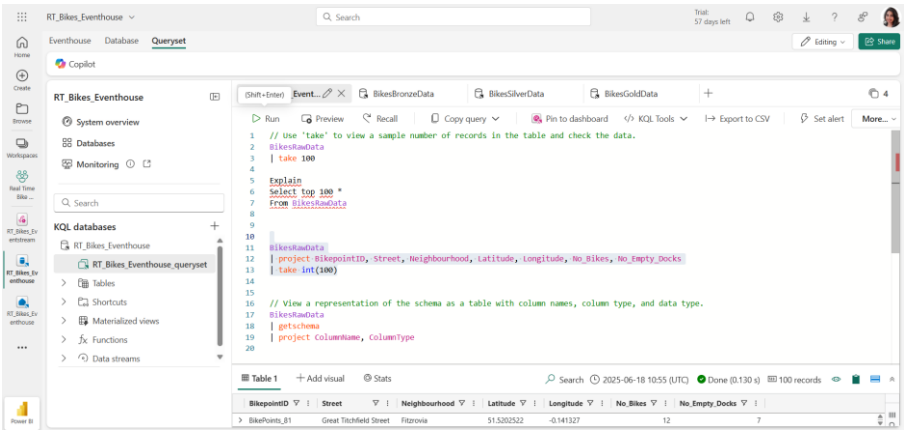




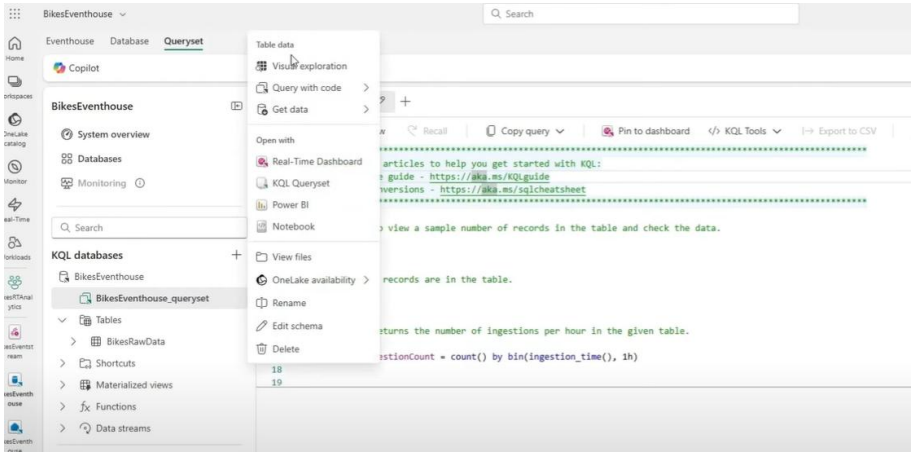
10. Once, it has been published, need to go to Workspace → Eventhouse → KQL Database → Refresh (Top left), then we can see the Table name, BikesRawData.



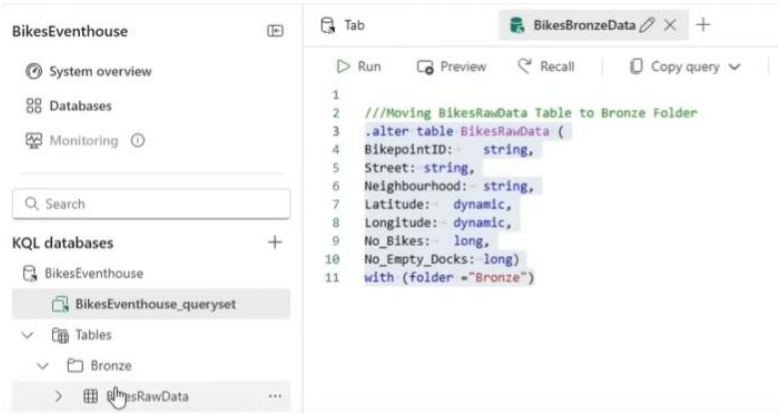
11. Once we have clicked KQL query set, we can run our KQL Queries here, it also Microsoft providing KQL Reference guide - https://aka.ms/KQLguide , we can referred a KQL queries from this link.



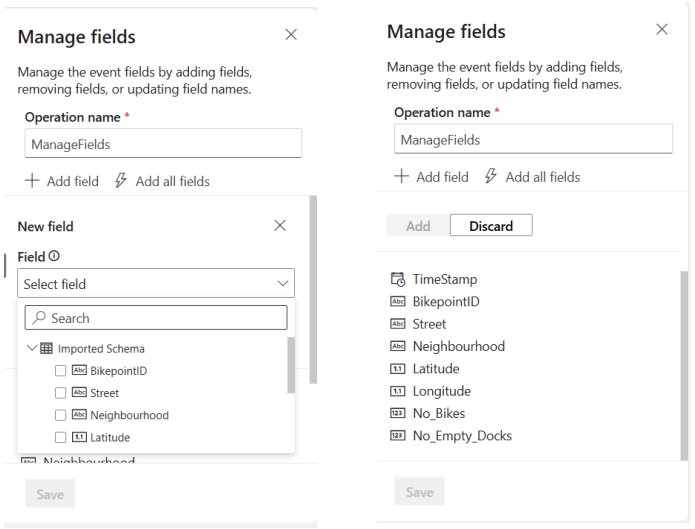
12. Probable see the data in, by right clicking the queryset the table data will appear, click→ Query with code → Show any 100 records (top 100), now the data will appear from eventstream and this is the KQL query  by run this we can see our data.

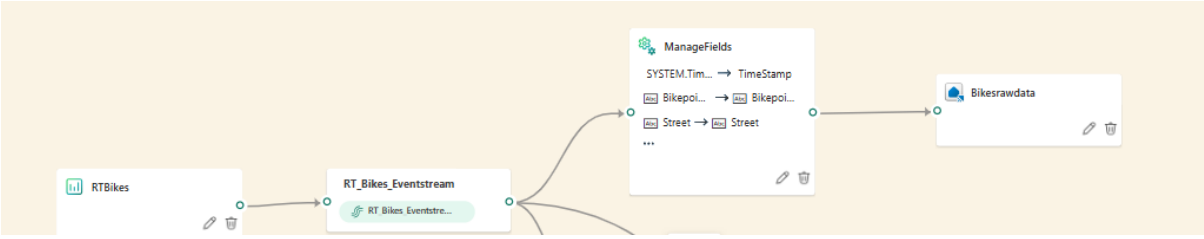


13. BikesRawData in a KQL database is a Bronze Layer Data, right now it is in Table, if we want it to store in folder structure under it, we will make it by KQL Query.

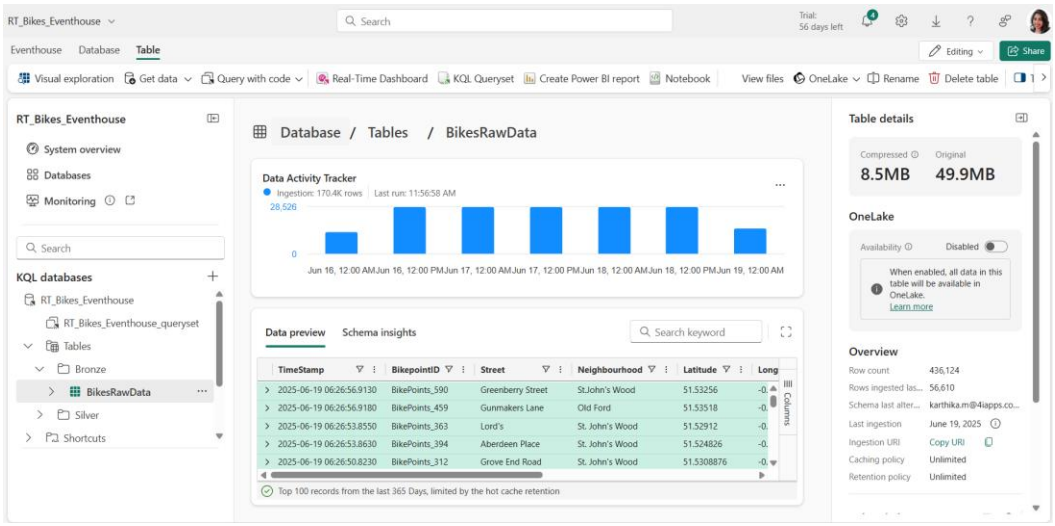


14. Since the dataset currently lacks a time field, adding one would be beneficial for report creation. To do this, go back to the Eventstream and edit it to include an inbuilt timestamp field. Click **Edit** at the top of the Eventstream. Under the **Transforming the Events** section, use the **Manage Fields** activity from the source connection. Once connected, click **Manage Fields** and then select **Add Fields** to modify the schema. In the new field section, choose fields from the **Imported Schema** (e.g., Bikepoint, Street, etc.) and add the **Inbuilt Timestamp**. After configuring everything, make sure to **Save** your changes, and **Refresh** the preview and **Publish** it.



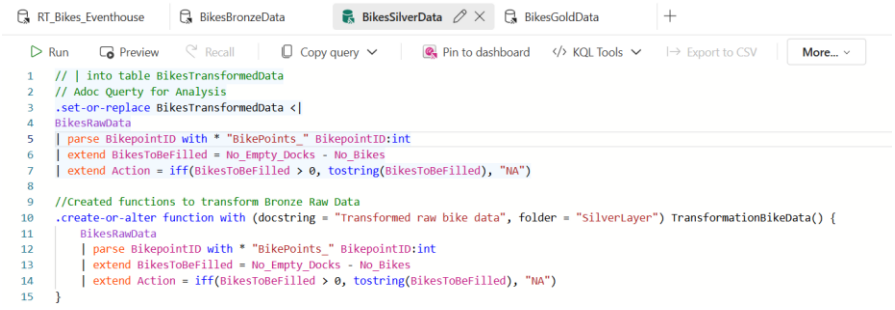


15. Go back to the Eventhouse and refresh the KQL database. Then, navigate to the table and click on the **Bronze** layer. The bronze data will now be visible—check the table to confirm that the timestamp field has been added.



16. Our goal is to determine how many bikes are needed to fill a station. This is calculated by subtracting the number of empty docks from the number of available bikes. If the result is positive, we want to take action; if it's negative, no action is required. We only need the **Bike Station ID**, and we want to remove the string prefix "BikePoints_" from it.

- To perform these transformations, we'll use the **parse** and **extend** operations. We'll go back to the KQL query set and add a new query for the **Silver Layer Transformation**. There are two approaches to this:
- **If the silver table does not exist** (which is the case now), we'll create it using the set-or-replace syntax and name it BikesTransformedData, using data from BikesRawData.
- **If the silver table already exists**, we could use the into operator to ingest data into the existing table by writing a query.
  Below is the syntax for the approach



In this query, our **source table** is BikesRawData. The goal is to transform this data for **ad-hoc analysis**.

🔧 **Step-by-Step Explanation of the Query:**

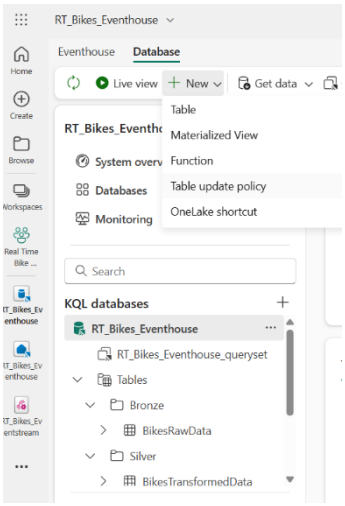
1. **Parsing the BikepointID**:
   - We're using the parse operator to extract the **numeric part** of the BikepointID field.
   - The original format is something like "BikePoints_123", and we only want the number 123.
   - We use the wildcard * to ignore the string part and extract the number as an int.
   - The extracted value is stored again in a new field named BikepointID with data type int.
2. **Adding a Column – BikesToBeFilled**:
   - Using the extend operator, we create a new column BikesToBeFilled, which is calculated as: `No_Empty_Docks - No_Bikes`
3. **Adding a Column – Action**:
   - Another extend is used to define an Action column.
   - If BikesToBeFilled is greater than 0, we keep the value as a string.
   - Otherwise, we set the action as "NA" (Not Available).
   - This helps indicate whether an action is needed or not.
4. **Ad-hoc Query Execution**:
   - This transformation is done using. set-or-replace BikesTransformedData to temporarily store the transformed data for analysis.
   - It's meant for **ad-hoc** or one-time use, unless automated further.
5. **Creating a Reusable Function**:
   - Instead of manually running this query every time, we define a function using create-or-alter function.
   - The function is named TransformationBikeData() and is saved under the folder SilverLayer.
   - Once this function is created successfully, it allows you to reuse the transformation logic automatically.Now, we will use the Update Policy on Bikesrawdata, now will create a new policy, We use **update policies** in Microsoft Fabric (KQL database) to **automate data transformations and flow** between tables—specifically from **raw (bronze) to transformed (silver)** layers.

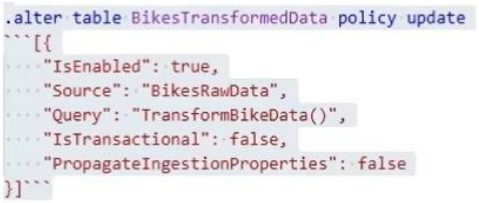
🔁 **Why Use an Update Policy?**

When real-time data is continuously ingested into a **source table** (e.g., BikesRawData), we often need to **transform** that data (e.g., parse fields, calculate values) and save the result into a **target table** (e.g., BikesTransformedData in the silver layer). Instead of running transformation queries manually, an **update policy** does this **automatically**.

🧠 **What It Does:**

- **Listens to a source table** (like your bronze layer).
- **Automatically executes a transformation query** whenever new data arrives.
- **Stores the result in a target table** (like your silver layer).
- Ensures real-time or near-real-time **data enrichment and flow** across layers.

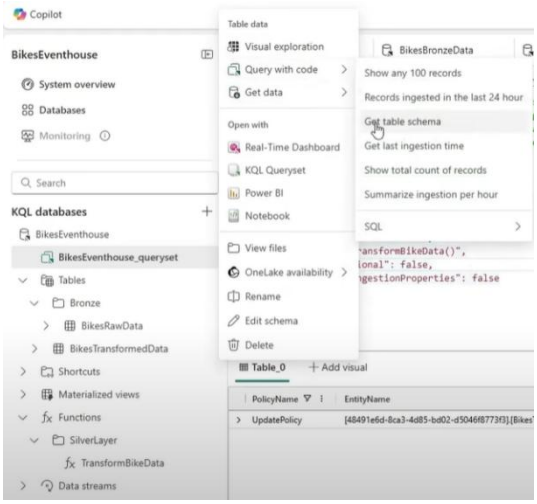





```
.alter table YOUR_TABLE_NAME policy update
[
  {
    "IsEnabled": true,                          // Enables the update policy
    "Source": "SOURCE_TABLE_NAME",              // Table that receives raw data
    "Query": "YOUR_TRANSFORMATION_QUERY",       // KQL logic to transform data
    "IsTransactional": false,                   // Whether transformation is transactional
    "PropagateIngestionProperties": false       // Whether to carry ingestion metadata
  }
]
```

19. Let's move the BikesTransformedData table into the **Silver Layer** folder. To do this, we first need to retrieve the **table schema** using the **Queryset**. In the **Silver Layer** query section, we will define the **Update Policy** using the transformation function we created earlier.

- Since we've modified some columns (e.g., added parsed fields or calculated values), the **Silver table schema** may differ from the **Bronze table**. Therefore, we must ensure the schema is updated accordingly to reflect these changes before applying the update policy and Silver Layer folder has been created, now we have bikes transform data in it.




```
// View a representation of the schema as a table with column names, column type, and data type.
c
| getschema
| project ColumnName, ColumnType

///Moving BikesRawData Table to Bronze Folder
.alter table BikesTransformedData (
TimeStamp: datetime,
BikepointID:    int ,
Street: string,
Neighbourhood:  string,
Latitude:   dynamic,
Longitude:  dynamic,
No_Bikes:   long,
No_Empty_Docks: long,
BikesToBeFilled:    long,
Action: string)
with (folder = "Silver")
```

20. Now, we're moving to the **Gold Layer**, where we will perform aggregations. For this, we'll create a **materialized view**—which is ideal for aggregation scenarios. One key advantage is that we don't need to explicitly define a function or an update policy; these are automatically handled in the background when working with materialized views for the Gold Layer. Let's open a new tab dedicated to the Gold Layer and use the following query to create the materialized view.

```
.create-or-alter materialized-view with (folder = "Gold") AggregatedData on table BikesTransformedData
{
    BikesTransformedData
    |summarize arg_max(TimeStamp,No_Bikes) by BikepointID
}
```

The syntax will use CREATE OR ALTER MATERIALIZED VIEW, placing it within the **Gold** folder. The materialized view will store **aggregated data** from the bike_transformed table. Our objective is to retrieve the **latest count of bikes available at each bike station**, grouped by bike_station_id and based on the **most recent timestamp**.

For visualization, we'll use a query that:

- Retrieves the latest bike count per station.
- Sorts the results by bike_point_id.

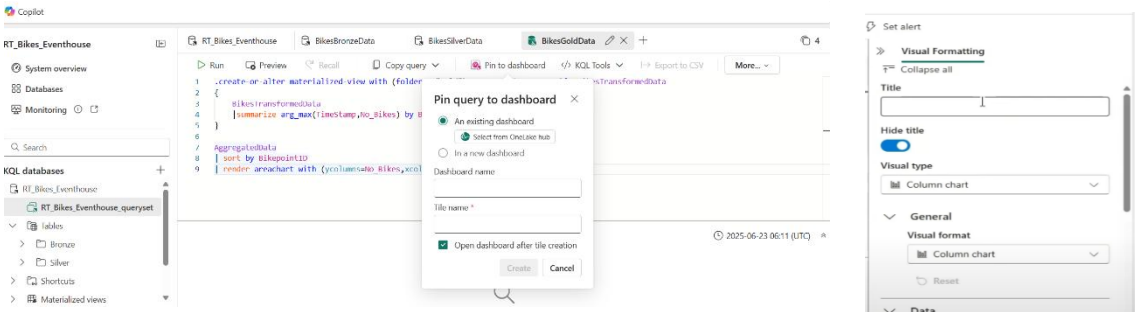
```
AggregatedData
| sort by BikepointID
| render areachart with (ycolumns=No_Bikes,xcolumn=BikepointID)
```

This query output can then be used to build a **column chart**, where each bar represents the number of bikes at the latest timestamp for a specific bike station. Sorting by bike_point_id allows for a clearer, organized visual representation of the data.

21. You can pin the visualization to a **Real-Time Dashboard** by clicking the **"Pin to dashboard"** option at the top. When you click it, a dialog box titled **"Pin query to dashboard"** will appear.
In this box:

- Select **"In a new dashboard"**
- Enter a **dashboard name** and a **tile name**
- Click **Create**

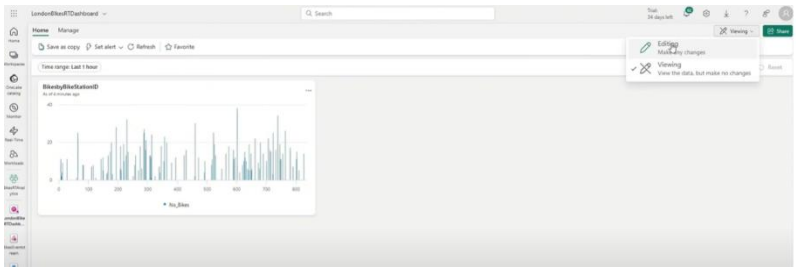
Once the dashboard is created, a **details panel** will appear on the left with **visual formatting options**. Fill in the necessary formatting details and click **Save**.
After completing these steps, your **Real-Time Dashboard** will be created and ready for use.



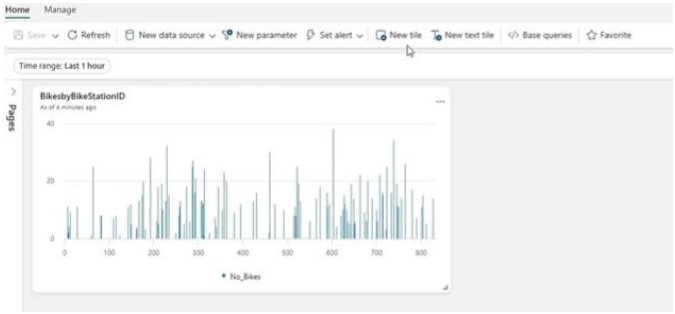
22. In the **Real-Time Dashboard**, we can create visualizations based on our data. You can switch from **view mode to edit mode** and add a **new tile**.

- For example, if there's a **critical neighborhood** you want to monitor—such as bike availability—you can use data from any layer available in the **Eventhouse**. Suppose we want to track high-priority data for a specific neighborhood, like **"Chelsea"**. You would write a query where neighborhood = "Chelsea" based on your business needs, then **run** the query and **apply the changes**.
- After executing the query, click on the **"Visual"** tab above the result table to begin designing your visualization. Fill out the **visual formatting settings** to complete the visual.
- Repeat this process for each additional visual you want to create:
    - Add a new tile
    - Write a query
    - Generate a visual based on that query

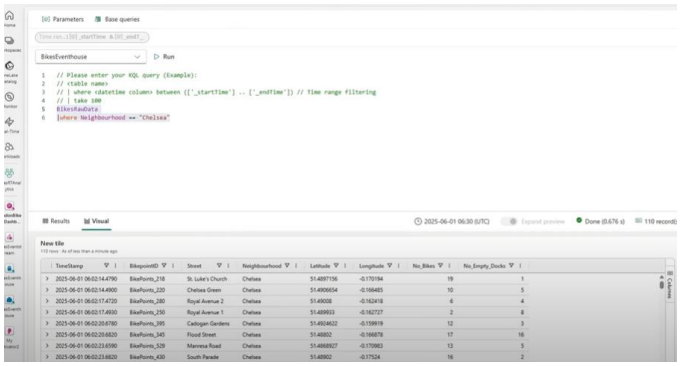
  This way, you can build a fully customized Real-Time Dashboard tailored to your data and business requirements.



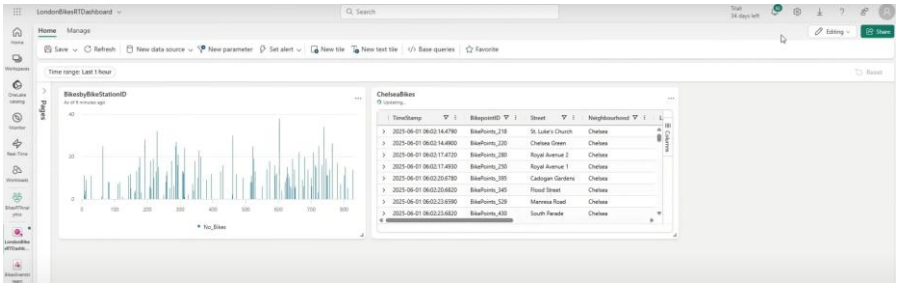
Real-Time Dashboard (Editing mode)



Real-Time Dashboard (New Tile to create a query for new visual)

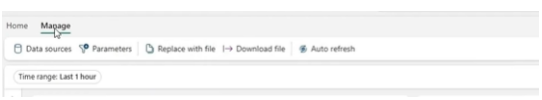


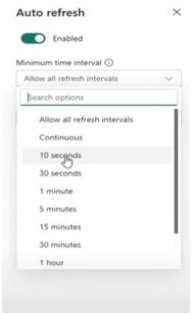
Real – Time Dashboard (Query Page to create a Visuals)

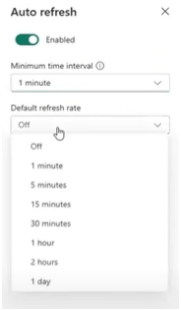


Real – Time Dashboard (Visual Page)

23. After creating the Real-Time Dashboard, you can configure the **auto-refresh** settings by going to the **Manage** section. There, you can specify how often you want the dashboard to refresh—whether it's **continuous** or at a fixed interval, such as **every 1 minute**. Simply choose your preferred option and click **Apply**.

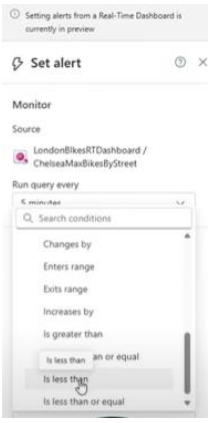






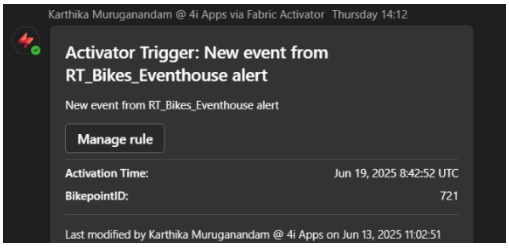
24. If you want to receive **alerts when the number of bikes falls below a certain threshold**, click **"Set Alert"** on the Home page. This will open the **Set Alert** panel on the right side of the screen.

- Fill in the required details, such as the condition, alert name, and delivery method, then click **Apply**. Once configured, the alert will notify you accordingly—whether you've chosen to receive it via **Microsoft Teams** or **Outlook**, it will be triggered in your selected platform, Finally, once the alerts have been successfully set up and created, click **Save** to save your **Real-Time Dashboard**.
-



Alert Panel

Microsoft Teams

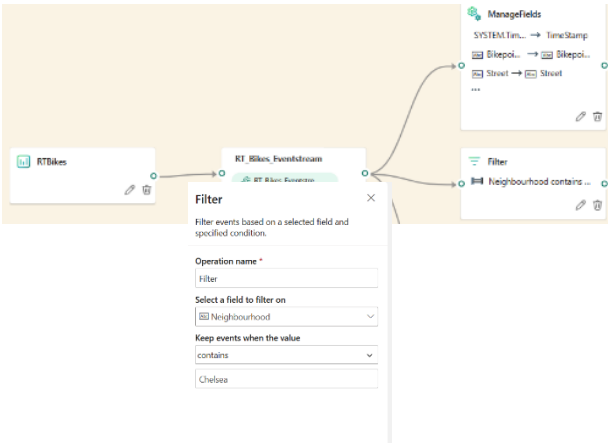


25. After completing the alert setup, return to the **Eventstream** to learn how to create a **derived stream**.

    For example, if you have **global data** but different teams—across countries or departments—require filtered data specific to their region, you can achieve this using **Eventstream**.

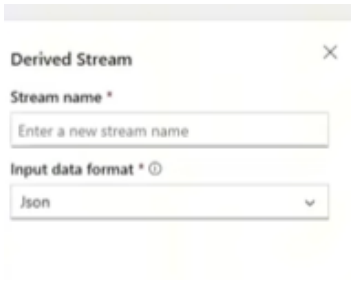
    Here's how:

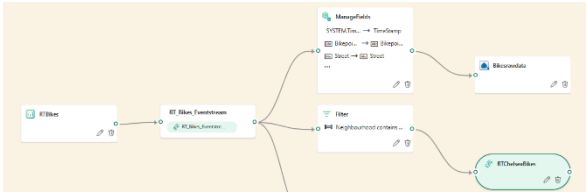
- Click **Edit** on the Eventstream.
- In the Eventstream panel, click **Transform Events** at the top.
- Add a **Filter** to the stream and connect it with stream source which as already connected with **Manage Fields** section.
- Configure the filter based on your needs. For instance, filter by the field **neighborhood**, and set the value to something like "Chelsea" to isolate that region's data.



Next, you must **create a destination** for this filtered data. Without a destination, the filter setup will show an error, as the stream must be fully connected.

- At the top of the Eventstream panel, go to **Transform Events**, click on **Stream**, and a stream configuration panel will appear.
- Fill in the **stream name** and connect it to the previously created **filter**.

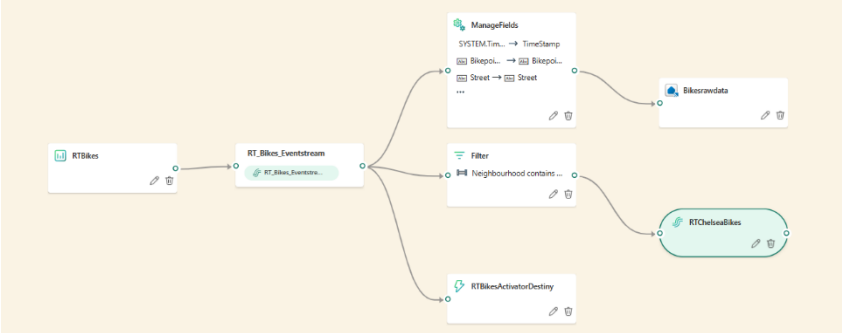




Additionally, if you need to set up a pipeline for **Data Activator**:

- In the same **Transform Events** section, choose **Activator**.
- A panel will open—fill in the required details, **connect it to the source**, and **save** the configuration.

Once done, you'll have a **complete end-to-end pipeline** set up, from source data to filtered stream and alert-based activation.



✅ Final Notes & Conclusion

With this end-to-end real-time data pipeline in Microsoft Fabric, we've successfully:

- Ingested live data using **Eventstream**
- Stored raw data in **Eventhouse (Bronze Layer)**
- Transformed and enriched it in the **Silver Layer** using functions and update policies
- Aggregated key metrics in the **Gold Layer** through materialized views
- Built interactive **Real-Time Dashboards** for visualization and decision-making
- Set up **alerts with Data Activator** to trigger notifications based on business conditions
- Created **derived streams** to distribute filtered data to specific teams or departments
- Enabled a fully connected, automated **real-time monitoring and analytics system**

This architecture not only supports **scalability and flexibility** but also ensures **real-time responsiveness**, helping organizations take **timely action** on data as it flows in.