Author: *Tyler Chessman* | See *github.com/tylerchessman/RTI_Medallion_Eg* for most recent updates.

Medallion Architecture in Fabric Real-Time Intelligence

Introduction

Building a multi-layer, medallion architecture using Fabric Real-Time Intelligence (RTI) requires a different approach compared to traditional data warehousing techniques. But even transactional source systems can be effectively processed in RTI. To demonstrate, we'll look at how sales orders (created in a relational database) can be continuously ingested and transformed through a RTI bronze, silver, and gold layer.

Pre-Requisites

Code samples and scripts are available in the GitHub repository https://github.com/tylerchessman/RTI_Medallion_Eg. To implement this example in your own environment, get started by creating the AdventureWorksLT sample database in Azure. See here for information on how to install the sample database. Then, enable Change Data Capture (CDC). The **00_CDCSetup_AdWorksLT.sql** script (located in the repository CDCSetup folder) can be used to enable and configure CDC.

You'll also need access to a workspace assigned to a Fabric-enabled capacity. Depending on your environment, you may be able to use a trial if an existing capacity is not available.

Medallion Architecture Primer

As big data and data lakes became popular, architectural patterns arose to help organize, process, and improve the quality of the data. The Medallion architecture defines three layers (bronze, silver, gold) where data can land and progress to serve different needs. The bronze (or raw) layer is typically used for initial ingestion of the data; data cleansing, validation, and transformation occurs in the silver layer – and additional modeling/aggregation takes place in a gold layer.

In Fabric RTI, a medallion architecture also contains these three layers. Data, whether it arrives in near real-time or in batches, is continuously processed and transformed along the bronze, silver, and gold layers – as shown in the following figure.

Author: *Tyler Chessman* | See *github.com/tylerchessman/RTI_Medallion_Eg* for most recent updates.

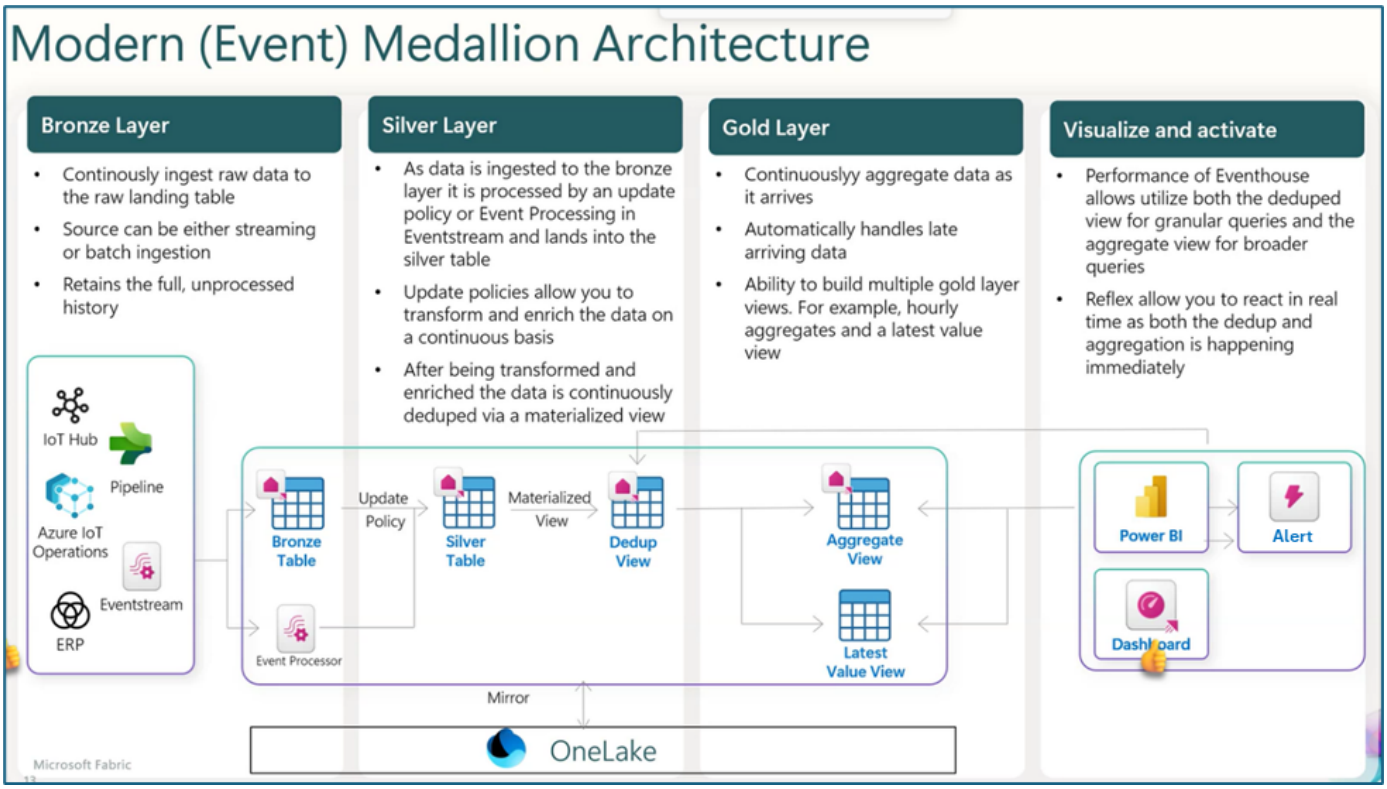

Figure 1 - Medallion Architecture in Fabric Real-Time Intelligence

With RTI, data isn't typically *updated* or *deleted* (as often occurs in a traditional data warehouse). Instead, all events are *inserted*. Different processes are used to reflect updates and/or filter out deleted records – but without overwriting the data.

To demonstrate, we start with a data source – the AdventureWorksLT database. Change Data Capture (CDC) is enabled for two tables – SalesOrderHeader and SalesOrderDetail. In Fabric, an eventstream ingests and pushes these CDC events into a bronze eventhouse table. Update policies populate (and maintain) silver tables – and materialized views remove any duplicates. Finally, a "latest value" view joins header and details into a single, denormalized gold layer.

Create the Fabric Eventhouse

Prior to connecting to the source database, we create an eventhouse in our Fabric workspace. An eventhouse contains one or more KQL databases; similar to SQL Server, a KQL database can contain multiple tables, views, and functions. It supports its own query language (also referred to as KQL). For those unfamiliar with KQL, SQL queries are also supported – and can be converted to KQL as needed.

Bonus tip: a KQL Database in Fabric has an entity diagram view, which shows the relationships among tables, functions, policies, views, etc. Peeking ahead, the entity diagram in our example will eventually look like the following:

Author: *Tyler Chessman* | See *github.com/tylerchessman/RTI_Medallion_Eg* for most recent updates.

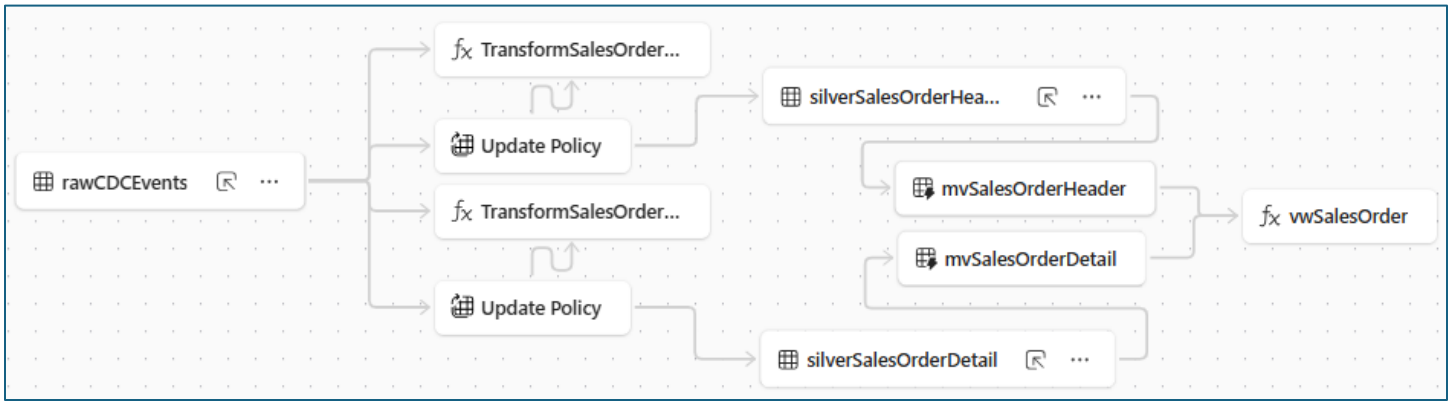

Figure 2 - KQL Database entity diagram view

Implement the Bronze Layer: Data Ingestion with an Eventstream

An Eventstream can connect to and process a variety of data sources (it can also expose an EventHub compatible endpoint so applications can push events directly to the stream). In this example, we connect to a CDC-enabled Azure SQL DB.

While an eventstream can also filter, join, split, or aggregate incoming events, our eventstream simply pushes the event data into a single landing table in the eventhouse – **rawCDCEvents.**

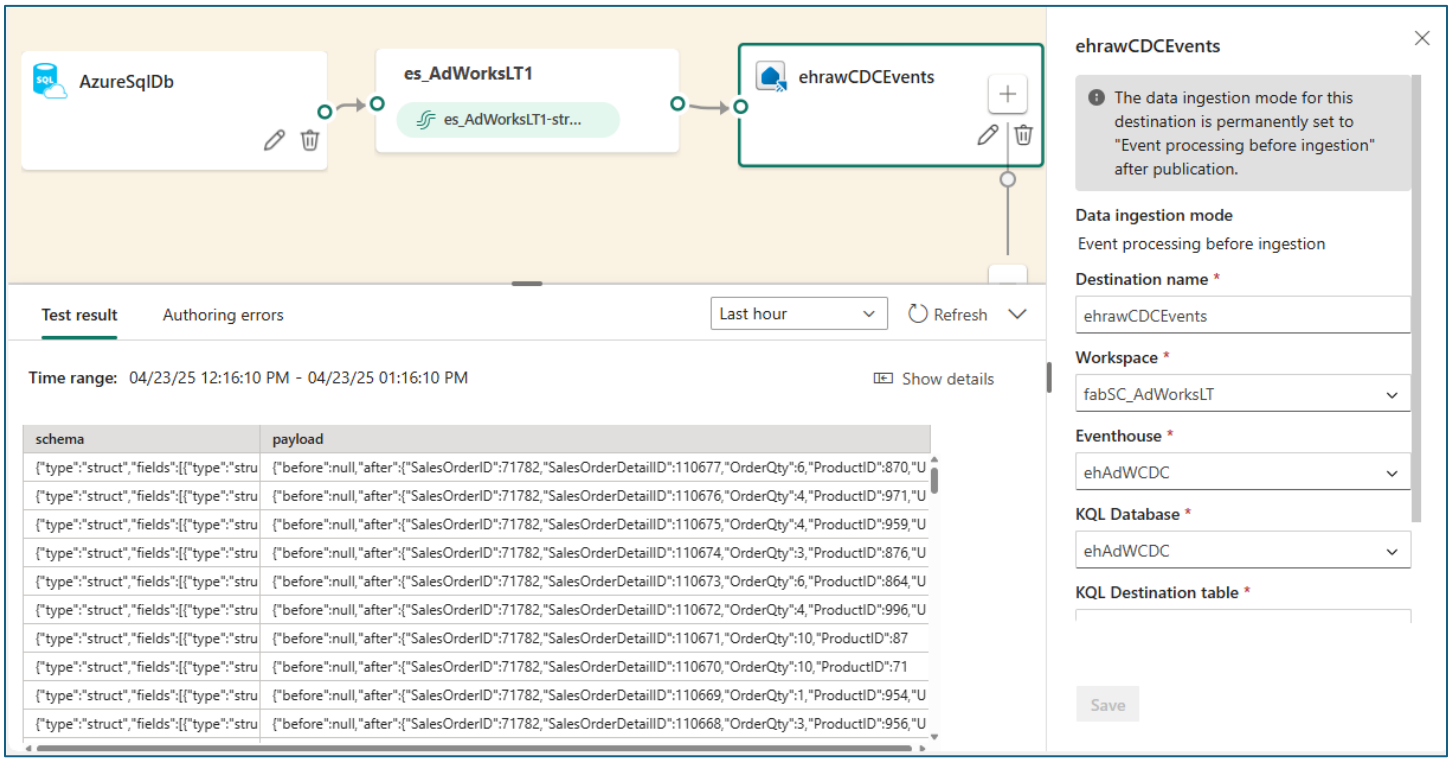

Figure 3- Eventstream designer

A couple of notes/tips when configuring the eventstream –

- When creating the source, and entering the initial list of Azure tables, do not include spaces between tables names (as of April 2025, there is a UI glitch that won't let you save if spaces are present).

- The designer can create the rawCDCEvents table on your behalf. Alternatively, you can create it ahead of time with a simple KQL Statement:
  .create table rawCDCEvents (schema:dynamic,payload:dynamic);

The Silver Layer: Table Update Policies and Materialized Views in the Eventhouse

The following query shows an example of records that are pushed into the **rawCDCEvents** table (the KQL folder in the GitHub repository contains all KQL queries and commands).

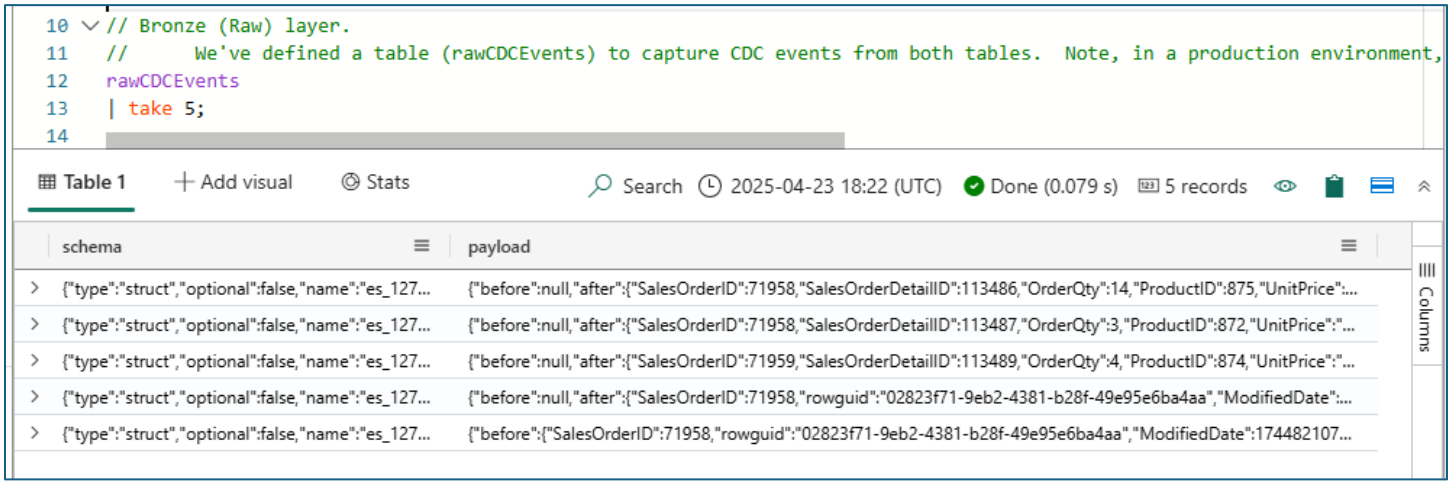

Figure 4 – A view of records in the rawCDCEvents table

The payload column is what we're interested in – a JSON representation of a CDC event. KQL has good JSON support, so we can parse each event into something suitable for light transformation. To do this, we define a table update policy. Specifically, a query is encapsulated into a function; this function is then referenced in an update policy. The policy also specifies a target table.

Think of an update policy like a trigger in a database table; the policy runs every time data is ingested; there isn't anything needed in terms of scheduling the update. And, though it is called an *update* policy, we are only appending *new* rows to the target table; there are no updates to *existing* rows. The following figures show the function and update policy used to populate the **silverSalesOrderHeader** table.

```
// With these two queries, we can now create a table update policy to populate the silverSalesOrderHeader and silverSale
//  First, we define functions that encapsulate our working queries
.create-or-alter function TransformSalesOrderHeaderToSilver() {
rawCDCEvents
| where tostring(payload.source.table) == "SalesOrderHeader"
| extend IngestedAt = ingestion_time()
| extend op = tostring(payload.op)
| extend columns = iif(op == "d", payload.before, payload.after)
| extend ModifiedDate = iif(op == "d", IngestedAt, unixtime_milliseconds_todatetime(tolong(columns.ModifiedDate)))
| extend SalesOrderID = toint(columns.SalesOrderID), RevisionNumber = toint(columns.RevisionNumber),
         OrderDate = unixtime_milliseconds_todatetime(tolong(columns.OrderDate)), DueDate = unixtime_milliseconds_todate
         ShipDate = unixtime_milliseconds_todatetime(tolong(columns.ShipDate)), Status = toint(columns.Status),
         OnlineOrderFlag = tobool(columns.OnlineOrderFlag), CustomerID = toint(columns.CustomerID), ShipToAddressID = to
         Comment =  tostring(columns.Comment), ModifiedDate = ModifiedDate // unixtime_milliseconds_todatetime(tolong(co
| project op, SalesOrderID, RevisionNumber, OrderDate, DueDate, ShipDate, Status, OnlineOrderFlag, CustomerID, ShipToAdd
}
```

Table 1 | Add visual | Stats | Search | 2025-04-23 18:26 (UTC) | Done (0.285 s) | 24 records

op	SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineO
c	71,958	1	2025-04-16 16:31:18.4030	2025-04-24 16:31:18.4030	2025-04-19 16:31:18.4030	1	true
u	71,958	2	2025-04-16 16:31:18.4030	2025-04-24 16:31:18.4030	2025-04-19 16:31:18.4030	2	true

Columns

Figure 5 - Function used in Table Update Function.

```
// Now, we create an update policy on each table
.alter table silverSalesOrderHeader policy update
```[{
    "IsEnabled": true,
    "Source": "rawCDCEvents",
    "Query": "TransformSalesOrderHeaderToSilver()",
    "IsTransactional": false,
    "PropagateIngestionProperties": false
}]```
```

*Figure 6 - Table Update Function*

Note that the silver tables can contain multiple rows for a given Order Header or Order Detail. For example, when a new Order Header is created, a row with an op (i.e. operation) value of **c** is ingested. Later, if the Order Header is updated, another row with an op value of **u** is ingested.

A Materialized View, using the summarize operation and arg_max() aggregation function, provides a persisted, deduplicated query that makes it easier for downstream users to analyze orders. Similar to a materialized view in a data warehouse, the view is maintained automatically (as new data arrives).
```

```
// Silver layer (Part II - Dedup View)
//  silverSalesOrderHeader and silverSalesOrderDetail contain a "running history" of all events for a Sales Order.
//    We will use a Materialized View to dedup each table - producing a deduplicated, persisted view.
//.drop materialized-view mvSalesOrderHeader
//.drop materialized-view mvSalesOrder
.create async materialized-view with (backfill=true) mvSalesOrderHeader  on table silverSalesOrderHeader
{
    silverSalesOrderHeader
    | summarize arg_max(ModifiedDate, *) by SalesOrderID
}
```

CreateMaterializedView: table()

Table 1 | Add visual | Stats | Search | 2025-04-23 18:31 (UTC) | Done (0.066 s) | 1 records

SalesOrderID	ModifiedDate	op	RevisionNumber	OrderDate	DueDate	ShipDate
71,958	2025-04-16 16:31:18.8970	u	2	2025-04-16 16:31:18.4030	2025-04-24 16:31:18.4030	2025-04-19 16:31:18.40

Figure 7 - Materialized View Definition

Gold Layer: The Latest (and Denormalized) View of Orders

Depending on the use case, creation of a gold layer may be done outside the eventhouse; for example, additional transformations may be done in a Power BI semantic model. For this scenario, we make use of a view to do a bit of final clean-up. Specifically, we join the two materialized views together – and filter our any deleted records.

```
// We can save this view for downstream clients
.create-or-alter function
    with (view=true)
    vwSalesOrder() { mvSalesOrderDetail
| join kind=inner
(mvSalesOrderHeader
| project-rename ['SOH.SalesOrderID']=SalesOrderID, ['SOH.ModifiedDate']=ModifiedDate, ['SOH.op']=op,
    ['SOH.IngestedAt']=IngestedAt) on ($left.SalesOrderID == $right.['SOH.SalesOrderID'])
| where (op != "d")
| project op, ['SOH.SalesOrderID'], RevisionNumber, OrderDate, DueDate, ShipDate, Status, OnlineOrderFlag, CustomerID,
    ShipToAddressID, Comment, SalesOrderDetailID, ProductID, OrderQty, UnitPrice, LineItemTotal=OrderQty * UnitPrice,
    ModifiedDate, IngestedAt
| project-rename SalesOrderID=['SOH.SalesOrderID'] }
```

Table 1 | Add visual | Stats | Search | 2025-04-23 18:36 (UTC) | Done (0.085 s) | 3 records

op	SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineO
c	71,958	2	2025-04-16 16:31:18.4030	2025-04-24 16:31:18.4030	2025-04-19 16:31:18.4030	2	true
c	71,958	2	2025-04-16 16:31:18.4030	2025-04-24 16:31:18.4030	2025-04-19 16:31:18.4030	2	true
u	71,958	2	2025-04-16 16:31:18.4030	2025-04-24 16:31:18.4030	2025-04-19 16:31:18.4030	2	true

Figure 8 - Latest Value View

Generate Test Data

In addition to the scripts needed to configure CDC, the SQL script **01_NewOrders.sql** (located in the CDCSetup folder) can be used to create a new Sales Order. Run this script (from a tool like SSMS or Azure Data Studio) to create new rows in the database – and then inspect the events being ingested into the eventhouse by running KQL queries. The SQL script itself is simple, but it covers all the basic operations – inserts, updates, and deletes.

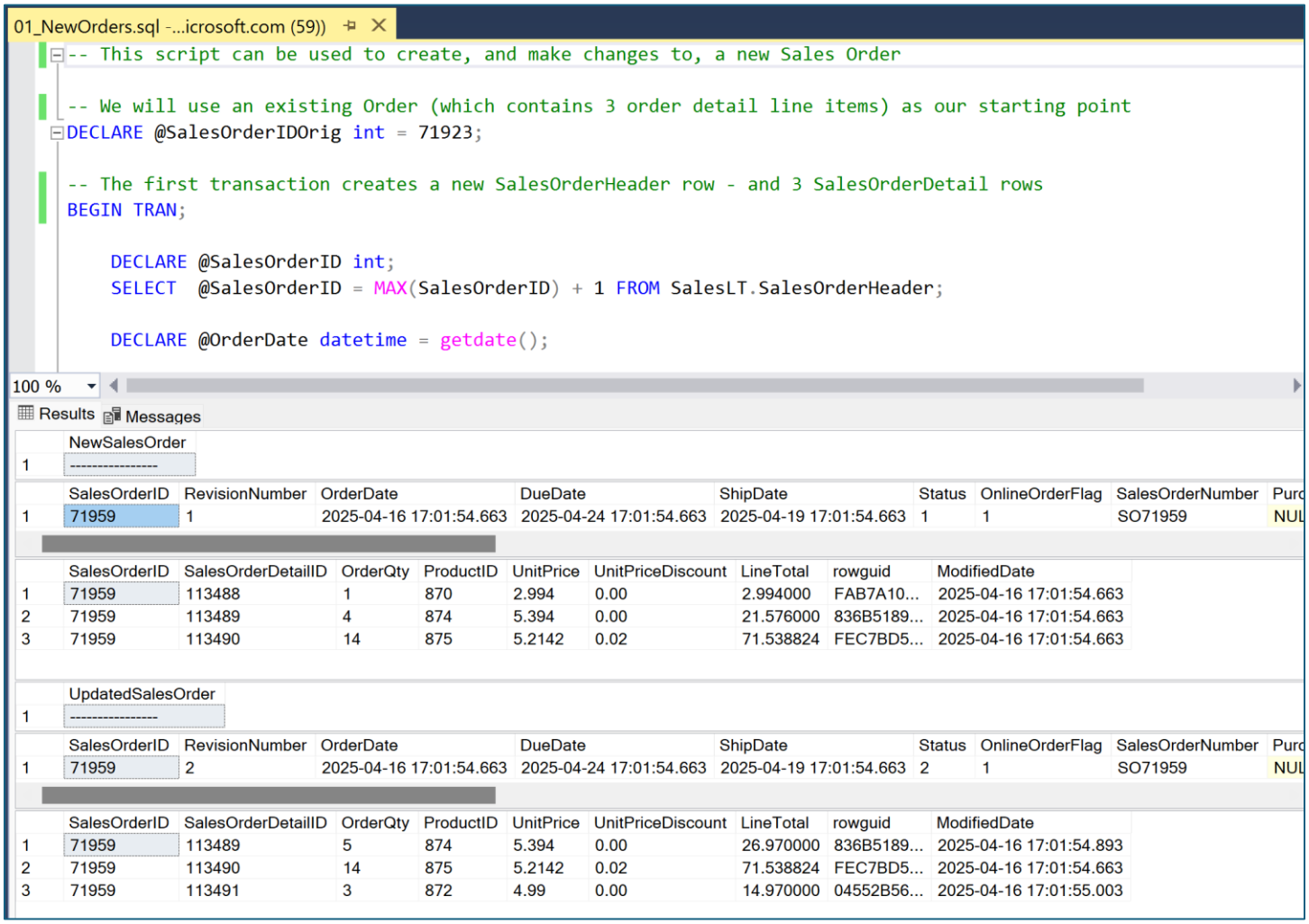

Figure 9 - New Sales Order generated (and then modified) with T-SQL script

Summary and Next Steps

Fabric RTI can continuously process transactional events - including updated and/or deleted records. While the approach differs from traditional data warehousing techniques, RTI can handle large volumes of data – while providing a full audit trail of changes for analysis and monitoring. I invite you to head over to the GitHub repository – and try recreating this sample in your environment.