

Medallion Architecture in Fabric Real-Time Intelligence

Summary

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.

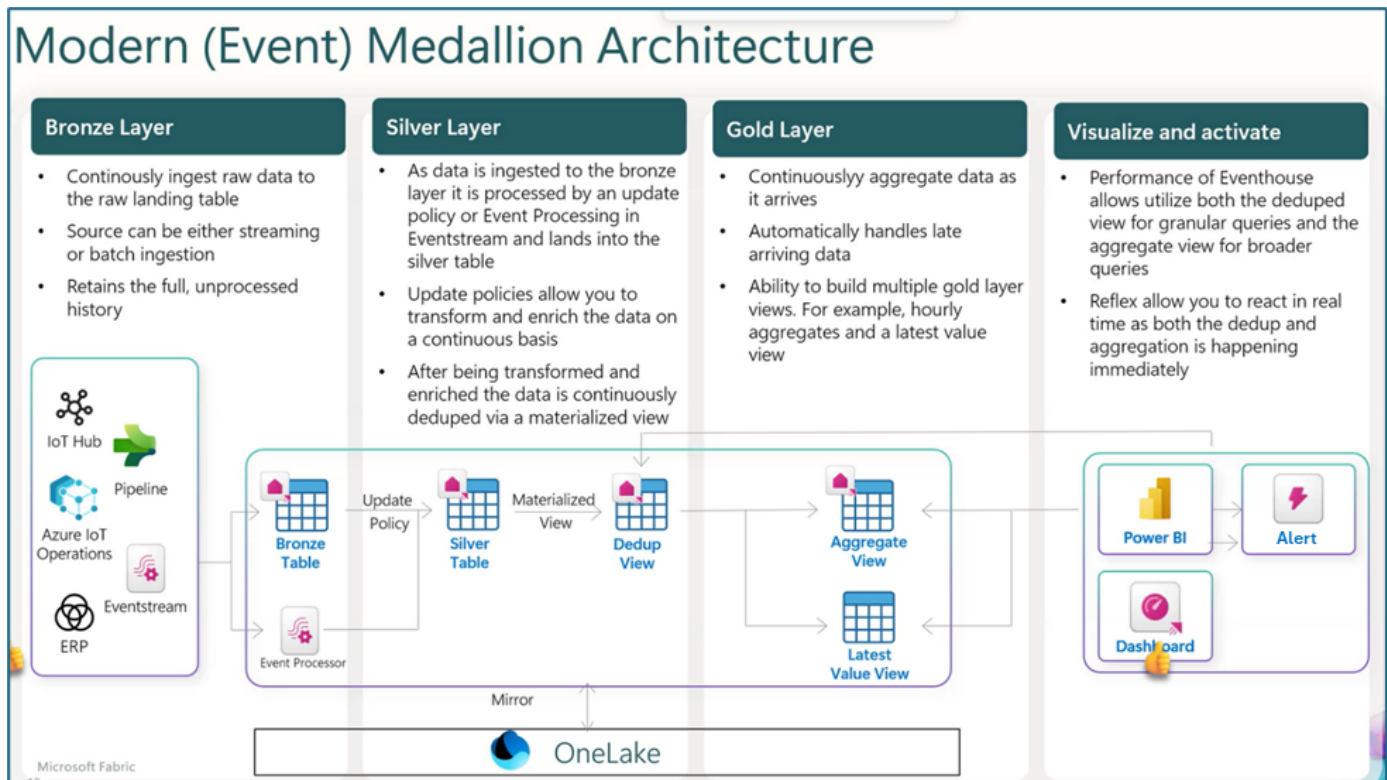


Figure 1 - Medallion Architecture in Fabric Real-Time Intelligence

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:

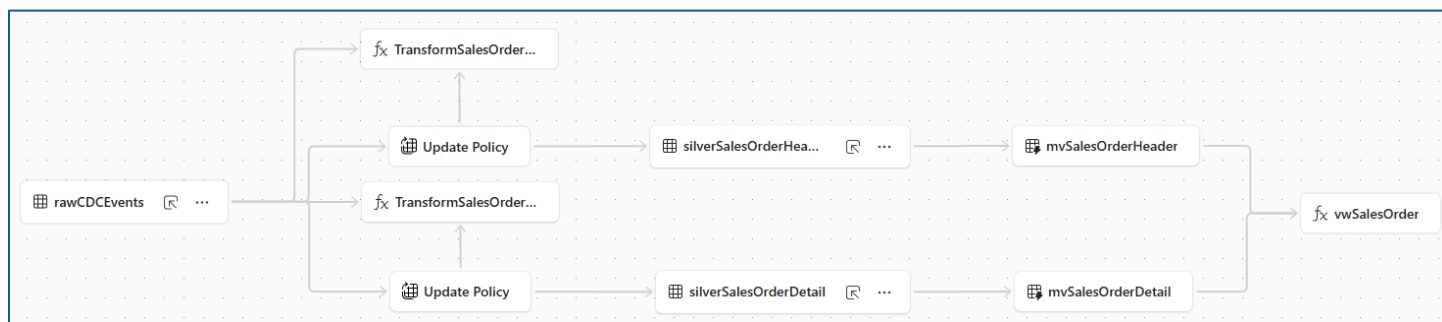


Figure 2 - KQL Database entity diagram view

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**.

The screenshot shows the 'Eventstream designer' interface. A pipeline is configured with three components: 'AzureSqlDb' (source), 'es_AdWorksLT1' (processor), and 'ehrawCDCEvents' (destination). Below the pipeline, there is a 'Test result' section showing a table with 'schema' and 'payload' columns. The 'payload' column contains JSON data representing sales order details. On the right side, there is a configuration panel for 'ehrawCDCEvents' with the following settings:

- Data ingestion mode:** Event processing before ingestion
- Destination name:** ehrawCDCEvents
- Workspace:** fabSC_AdWorksLT
- Eventhouse:** ehAdWCDC
- KQL Database:** ehAdWCDC
- KQL Destination table:** rawCDCEvents

A 'Save' button is located at the bottom of the configuration panel.

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).

```
10 // Bronze (Raw) layer.
11 // We've defined a table (rawCDCEvents) to capture CDC events from both tables. Note, in a production environment, you could choose to filter/push the raw events into multiple tables.
12 rawCDCEvents
13 | take 5;
14
```

Table 1

+ Add visual

Stats

Search

2025-04-16 21:10 (UTC)

Done (0.071 s)

5 records

schema	payload
> ["type":"struct","optional":false,"name":"es_1277e215-c64d-4...	["before":null,"after":{"SalesOrderID":71958,"SalesOrderDetailID":113486,"OrderQty":14,"ProductID":875,"UnitPrice":"AMuu","UnitPriceDiscount":"AMg=","LineTotal":"AA=","rowguid":"97f93c03-0c8d...
> ["type":"struct","optional":false,"name":"es_1277e215-c64d-4...	["before":null,"after":{"SalesOrderID":71958,"SalesOrderDetailID":113487,"OrderQty":3,"ProductID":872,"UnitPrice":"AMLs","UnitPriceDiscount":"AA=","LineTotal":"AA=","rowguid":"88006969-39f0-4...
> ["type":"struct","optional":false,"name":"es_1277e215-c64d-4...	["before":null,"after":{"SalesOrderID":71959,"SalesOrderDetailID":113489,"OrderQty":4,"ProductID":874,"UnitPrice":"ANK0","UnitPriceDiscount":"AA=","LineTotal":"AA=","rowguid":"836b5189-fbfd-4...
> ["type":"struct","optional":false,"name":"es_1277e215-c64d-4...	["before":null,"after":{"SalesOrderID":71958,"rowguid":"02823f71-9eb2-4381-b28f-49e95e6ba4aa","ModifiedDate":1744821078403,"RevisionNumber":1,"OrderDate":1744821078403,"DueDate":17455...
> ["type":"struct","optional":false,"name":"es_1277e215-c64d-4...	["before":{"SalesOrderID":71958,"rowguid":"02823f71-9eb2-4381-b28f-49e95e6ba4aa","ModifiedDate":1744821078403,"RevisionNumber":1,"OrderDate":1744821078403,"DueDate":1745512278403,"...

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.

```
68 // With these two queries, we can now create a table update policy to populate the silverSalesOrderHeader and silverSalesOrderDetail table.
69 // First, we define functions that encapsulate our working queries
70 .create-or-alter function TransformSalesOrderHeaderToSilver() {
71   rawCDCEvents
72   | where toString(payload.source.table) == "SalesOrderHeader"
73   | extend IngestedAt = ingestion_time()
74   | extend op = toString(payload.op)
75   | extend columns = if(op == "d", payload.before, payload.after)
76   | extend ModifiedDate = if(op == "d", IngestedAt, unixtime_milliseconds_todatetime(tolong(columns.ModifiedDate)))
77   | extend SalesOrderID = toint(columns.SalesOrderID), RevisionNumber = toint(columns.RevisionNumber),
78   |       OrderDate = unixtime_milliseconds_todatetime(tolong(columns.OrderDate)), DueDate = unixtime_milliseconds_todatetime(tolong(columns.DueDate)),
79   |       ShipDate = unixtime_milliseconds_todatetime(tolong(columns.ShipDate)), Status = toint(columns.Status),
80   |       OnlineOrderFlag = tobool(columns.OnlineOrderFlag), CustomerID = toint(columns.CustomerID), ShipToAddressID = toint(columns.ShipToAddressID),
81   |       Comment = toString(columns.Comment), ModifiedDate = ModifiedDate // unixtime_milliseconds_todatetime(tolong(columns.ModifiedDate))
82   | project op, SalesOrderID, RevisionNumber, OrderDate, DueDate, ShipDate, Status, OnlineOrderFlag, CustomerID, ShipToAddressID, Comment, ModifiedDate, IngestedAt;
83 }
84
85 .create-or-alter function TransformSalesOrderDetailToSilver() {
86   rawCDCEvents
```

Table 1

+ Add visual

Stats

Search

2025-04-16 21:18 (UTC)

Done (0.072 s)

24 records

op	SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	CustomerID	ShipToAddressID	Comment	ModifiedDate
> c	71,955	1	2025-03-27 16:06:07.5330	2025-04-04 16:06:07.5330	2025-03-30 16:06:07.5330	1	true	29,781	1,035	New Order Mar 27 2025 4:06PM	2025-03-27
> u	71,955	2	2025-03-27 16:06:07.5330	2025-04-04 16:06:07.5330	2025-03-30 16:06:07.5330	2	true	29,781	1,035	New Order Mar 27 2025 4:06PM	2025-03-27

Figure 5 - Function used in Table Update Function.

```
110 // Now, we create an update policy on each table
111 .alter table silverSalesOrderHeader policy update
112 '''{
113   "IsEnabled": true,
114   "Source": "rawCDCEvents",
115   "Query": "TransformSalesOrderHeaderToSilver()",
116   "IsTransactional": false,
117   "PropagateIngestionProperties": false
118 }'''
```

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).

```
148 // Silver layer (Part II - Dedup View)
149 // silverSalesOrderHeader and silverSalesOrderDetail contain a "running history" of all events for a Sales Order.
150 // We will use a Materialized View to dedup each table - producing a deduplicated, persisted view.
151 // .drop materialized-view mvSalesOrderHeader
152 // .drop materialized-view mvSalesOrderDetail
153 .create async materialized-view with (backfill=true) mvSalesOrderHeader on table silverSalesOrderHeader
154 {
155     silverSalesOrderHeader
156     | summarize arg_max(ModifiedDate, *) by SalesOrderID
157 }
158
```

Table 1

+

 Add visual

Ⓢ

 Stats

🔍

 Search

🕒

 2025-04-16 21:24 (UTC)

🟢

 Done (0.077 s)

📄

 1 records

👁

🗑

🔗

SalesOrderID	ModifiedDate	op	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	CustomerID	ShipToAddressID	Comment	IngestedAt
>	71,955	2025-03-27 16:06...	u	2	2025-03-27 16:06...	2025-04-04 16:06...	2025-03-30 16:06...	2	true	29,781	1,035	New Order Mar 2... 2025-03-27 16:06...

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 out any deleted records.

```
187 // We can save this view for downstream clients
188 .create-or-alter function
189 | with (view=true)
190 | vwSalesOrder() { mvSalesOrderDetail
191 | join kind=inner
192 | (mvSalesOrderHeader
193 | project-rename ['SOH.SalesOrderID'] = SalesOrderID, ['SOH.ModifiedDate'] = ModifiedDate, ['SOH.op'] = op, ['SOH.IngestedAt'] = IngestedAt) on ($left.SalesOrderID == $right['SOH.SalesOrderID'])
194 | where (op != "d")
195 | project op, ['SOH.SalesOrderID'], RevisionNumber, OrderDate, DueDate, ShipDate, Status, OnlineOrderFlag, CustomerID,
196 | ShipToAddressID, Comment, SalesOrderDetailID, ProductID, OrderQty, UnitPrice, LineItemTotal=OrderQty * UnitPrice, ModifiedDate, IngestedAt
197 | project-rename SalesOrderID=['SOH.SalesOrderID'] }
198
199 vwSalesOrder
200 | sort by SalesOrderID, SalesOrderDetailID
201 | where (IngestedAt > datetime_add("minute", int(-15), now()))
202 | take 10
203
204
```

op	SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	CustomerID	ShipToAddressID	Comment	SalesOrderDetailID
>	c	71,955	2	2025-03-27 16:06:07.5330	2025-04-04 16:06:07.5330	2025-03-30 16:06:07.5330	2	true	29,781	1,035	New Order Mar 27 2025 4:06PM
>	c	71,955	2	2025-03-27 16:06:07.5330	2025-04-04 16:06:07.5330	2025-03-30 16:06:07.5330	2	true	29,781	1,035	New Order Mar 27 2025 4:06PM
>	u	71,955	2	2025-03-27 16:06:07.5330	2025-04-04 16:06:07.5330	2025-03-30 16:06:07.5330	2	true	29,781	1,035	New Order Mar 27 2025 4:06PM

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.

```

01_NewOrders.sql - Microsoft.com (59)
-- This script can be used to create, and make changes to, a new Sales Order

-- We will use an existing Order (which contains 3 order detail line items) as our starting point
DECLARE @SalesOrderIDOrig int = 71923;

-- The first transaction creates a new SalesOrderHeader row - and 3 SalesOrderDetail rows
BEGIN TRAN;

    DECLARE @SalesOrderID int;
    SELECT @SalesOrderID = MAX(SalesOrderID) + 1 FROM SalesLT.SalesOrderHeader;

    DECLARE @OrderDate datetime = getdate();
  
```

NewSalesOrder										
SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	SalesOrderNumber	PurchaseOrderNumber	OrderType	SubOrderType
71959	1	2025-04-16 17:01:54.663	2025-04-24 17:01:54.663	2025-04-19 17:01:54.663	1	1	SO71959			

SalesOrderID	SalesOrderDetailID	OrderQty	ProductID	UnitPrice	UnitPriceDiscount	LineTotal	rowguid	ModifiedDate
71959	113488	1	870	2.994	0.00	2.994000	FAB7A10...	2025-04-16 17:01:54.663
71959	113489	4	874	5.394	0.00	21.576000	836B5189...	2025-04-16 17:01:54.663
71959	113490	14	875	5.2142	0.02	71.538824	FEC7BD5...	2025-04-16 17:01:54.663

UpdatedSalesOrder										
SalesOrderID	RevisionNumber	OrderDate	DueDate	ShipDate	Status	OnlineOrderFlag	SalesOrderNumber	PurchaseOrderNumber	OrderType	SubOrderType
71959	2	2025-04-16 17:01:54.663	2025-04-24 17:01:54.663	2025-04-19 17:01:54.663	2	1	SO71959			

SalesOrderID	SalesOrderDetailID	OrderQty	ProductID	UnitPrice	UnitPriceDiscount	LineTotal	rowguid	ModifiedDate
71959	113489	5	874	5.394	0.00	26.970000	836B5189...	2025-04-16 17:01:54.893
71959	113490	14	875	5.2142	0.02	71.538824	FEC7BD5...	2025-04-16 17:01:54.663
71959	113491	3	872	4.99	0.00	14.970000	04552B56...	2025-04-16 17:01:55.003

Figure 9 - New Sales Order generate 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.