

Contents

Stream Analytics Documentation

Overview

[What is Stream Analytics?](#)

[End-to-end solution patterns](#)

Quickstarts

[Create a job - Azure portal](#)

[Create a job - PowerShell](#)

[Create a job - Visual Studio](#)

[Create a job - Visual Studio Code](#)

Tutorials

[1 - Create / manage a job](#)

[2 - Run Azure Functions](#)

[3 - Run a Javascript UDF](#)

[4 - Deploy with CI/CD in Azure Pipelines](#)

[5 - Run C# UDF on IoT Edge](#)

Samples

[Code samples](#)

Concepts

[Choose a streaming analytics technology](#)

[Input types for a job](#)

[Inputs overview](#)

[Streaming data inputs](#)

[Reference data inputs](#)

[Output types for a job](#)

[Outputs overview](#)

[Output to Cosmos DB](#)

[Output to Azure SQL DB](#)

[Blob custom path patterns](#)

[States of a job](#)

[Window functions](#)

[Geospatial functions](#)

[Compatibility level](#)

[Common query patterns](#)

[Parse JSON and AVRO data](#)

[Time handling considerations](#)

[Checkpoint and replay](#)

[Error policy](#)

[How-to-guides](#)

[Manage](#)

[Update credentials](#)

[Configure event ordering policies](#)

[Configure alerts](#)

[Start a job](#)

[Test a job](#)

[View results on a dashboard](#)

[Stop or delete a job](#)

[Pair jobs for reliability](#)

[Authenticate with managed identity](#)

[Use SQL reference data](#)

[Build solutions](#)

[Twitter sentiment analysis](#)

[Real-time fraud detection](#)

[Run jobs on IoT Edge](#)

[Toll booth sensor data analysis](#)

[Run a JavaScript UDA](#)

[High-frequency trading](#)

[Process IoT streaming data](#)

[Threshold-based rules](#)

[Process Kafka events](#)

[Geospatial scenarios](#)

[Monitor](#)

[Monitor jobs- Azure portal](#)

[Monitor jobs- PowerShell](#)

[Monitor jobs- Azure .NET SDK](#)

[Monitor jobs- Visual Studio](#)

Scale

[Scale with streaming units](#)

[Scale with parallelization](#)

[Query parallelization](#)

[Increase throughput](#)

[Scale with ML functions](#)

Automate

[Using .NET SDK](#)

[Using Azure PowerShell](#)

[Using API for IoT Edge jobs](#)

Visual Studio

[Install tools](#)

[Test with sample data](#)

[Test with live data](#)

[View jobs in Visual Studio](#)

[Develop an edge job](#)

[Set up CI/CD pipeline](#)

[Write .NET UDF for edge](#)

Visual Studio Code

[Test with sample data](#)

[Set up CI/CD pipeline](#)

[Explore jobs](#)

Troubleshoot

[Input](#)

[Output](#)

[Query logic](#)

[Activity and diagnostic logs](#)

[Data errors](#)

[Integrate with machine learning](#)

[Sentiment analysis with ML models](#)

[Anomaly detection](#)

[Use REST APIs](#)

[Debug using job diagram](#)

[Stream Analytics Query Language](#)

[Built-in Functions](#)

[Built-in Functions Overview](#)

[Aggregate Functions](#)

[Aggregate Functions Overview](#)

[AVG](#)

[COUNT](#)

[Collect](#)

[CollectTOP](#)

[MAX](#)

[MIN](#)

[Percentile_Cont](#)

[Percentile_Disc](#)

[STDEV](#)

[STDEVP](#)

[SUM](#)

[TopOne](#)

[VAR](#)

[VARP](#)

[Analytic Functions](#)

[Analytic Functions Overview](#)

[AnomalyDetection_SpikeAndDip](#)

[AnomalyDetection_ChangePoint](#)

[ISFIRST](#)

[LAG](#)

[LAST](#)

[Array Functions](#)

[Array Functions Overview](#)

[GetArrayLength](#)

[GetArrayElement](#)

[GetArrayElements](#)

[Conversion Functions](#)

[Conversion Functions Overview](#)

[CAST](#)

[GetType](#)

[TRY_CAST](#)

[Date and Time Functions](#)

[Date and Time Functions Overview](#)

[DATEADD](#)

[DATEDIFF](#)

[DATENAME](#)

[DATEPART](#)

[DATETIMEFROMPARTS](#)

[DAY](#)

[MONTH](#)

[YEAR](#)

[GeoSpatial Functions](#)

[GeoSpatial Functions Overview](#)

[CreateLineString](#)

[CreatePoint](#)

[CreatePolygon](#)

[ST_DISTANCE](#)

[ST_OVERLAPS](#)

[ST_INTERSECTS](#)

[ST_WITHIN](#)

[Input Metadata Functions](#)

[Input Metadata Functions Overview](#)

[GetMetadataPropertyValue](#)

[Mathematical Functions](#)

[Mathematical Functions Overview](#)

[ABS](#)

[CEILING](#)

[EXP](#)

[FLOOR](#)

[POWER](#)

[ROUND](#)

[SIGN](#)

[SQUARE](#)

[SQRT](#)

[Record Functions](#)

[Record Functions Overview](#)

[GetRecordProperties](#)

[GetRecordPropertyValue](#)

[String Functions](#)

[String Functions Overview](#)

[CHARINDEX](#)

[CONCAT](#)

[LEN](#)

[LOWER](#)

[PATINDEX](#)

[REGEXMATCH](#)

[REPLACE](#)

[SUBSTRING](#)

[UPPER](#)

[Windowing Functions](#)

[Windowing Functions Overview](#)

[Hopping Window](#)

[Session Window](#)

[Sliding Window](#)

[Tumbling Window](#)

[Aggregate Windowing functions](#)

Data Types

[Data Types Overview](#)

[Parsing JSON and AVRO data](#)

Query Language Elements

[Query Language Elements Overview](#)

[APPLY](#)

[CASE](#)

[COALESCE](#)

[CREATE TABLE](#)

[FROM](#)

[GROUP BY](#)

[HAVING](#)

[INTO](#)

[JOIN](#)

[Reference Data JOIN](#)

[SELECT](#)

[UNION](#)

[WHERE](#)

[WITH](#)

Time Management

[Time Management Overview](#)

[System.Timestamp](#)

[TIMESTAMP BY](#)

[Time Skew Policies](#)

Event Delivery Guarantees

Reference

[Azure PowerShell](#)

[.NET](#)

[REST](#)

[Resource Manager template](#)

Resources

[Stream Analytics previews](#)

[Azure Roadmap](#)

[Blog](#)

[Feedback forum](#)

[Forum](#)

[Pricing](#)

[Pricing calculator](#)

[Service updates](#)

[Stack Overflow](#)

[Videos](#)

[Customer case studies](#)

[Whitepaper - Real-time event processing](#)

What is Azure Stream Analytics?

6/21/2019 • 6 minutes to read • [Edit Online](#)

Azure Stream Analytics is a real-time analytics and complex event-processing engine that is designed to analyze and process high volumes of fast streaming data from multiple sources simultaneously. Patterns and relationships can be identified in information extracted from a number of input sources including devices, sensors, clickstreams, social media feeds, and applications. These patterns can be used to trigger actions and initiate workflows such as creating alerts, feeding information to a reporting tool, or storing transformed data for later use. Also, Stream Analytics is available on Azure IoT Edge runtime, and supports the same exact language or syntax as cloud.

The following scenarios are examples of when you can use Azure Stream Analytics:

- Analyze real-time telemetry streams from IoT devices
- Web logs clickstream analytics
- Geospatial analytics for fleet management and driverless vehicles
- Remote monitoring and predictive maintenance of high value assets
- Real-time analytics on Point of Sale data for inventory control and anomaly detection

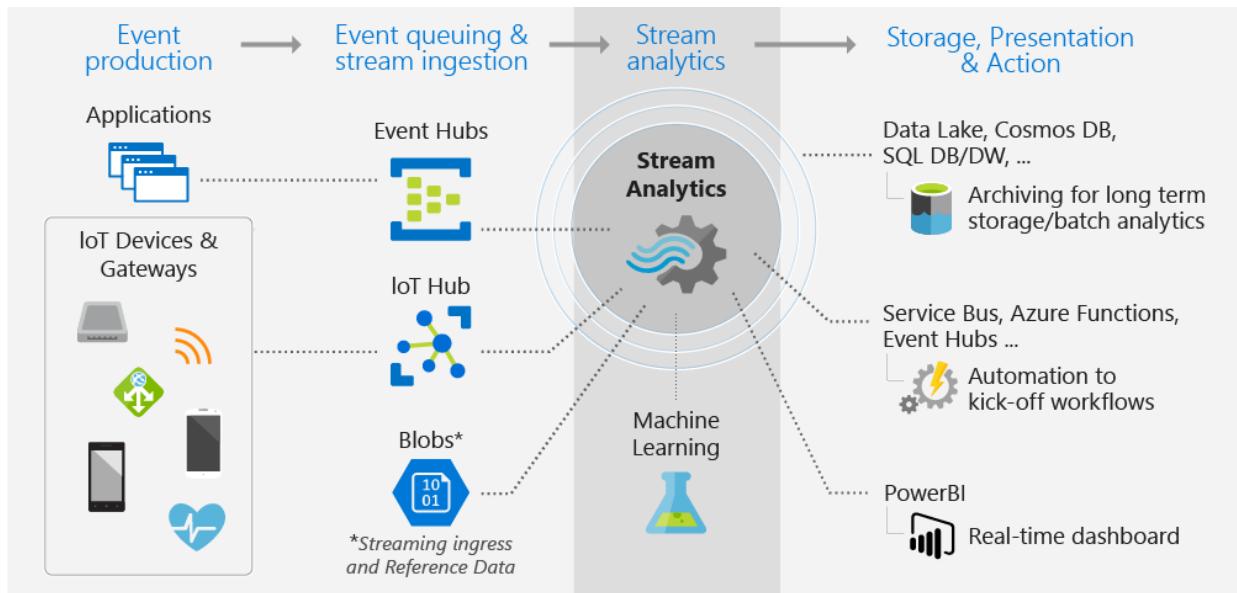
How does Stream Analytics work?

An Azure Stream Analytics job consists of an input, query, and an output. Stream Analytics ingests data from Azure Event Hubs, Azure IoT Hub, or Azure Blob Storage. The query, which is based on SQL query language, can be used to easily filter, sort, aggregate, and join streaming data over a period of time. You can also extend this SQL language with JavaScript and C# user defined functions (UDFs). You can easily adjust the event ordering options and duration of time windows when performing aggregation operations through simple language constructs and/or configurations.

Each job has an output for the transformed data, and you can control what happens in response to the information you've analyzed. For example, you can:

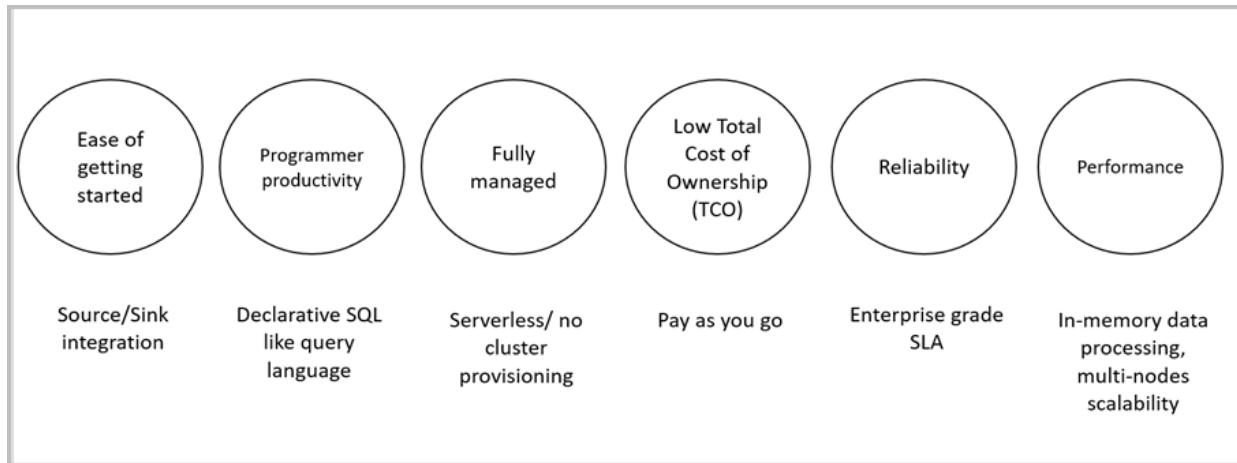
- Send data to services such as Azure Functions, Service Bus Topics or Queues to trigger communications or custom workflows downstream.
- Send data to a Power BI dashboard for real-time dashboarding.
- Store data in other Azure storage services to train a machine learning model based on historical data or perform batch analytics.

The following image shows how data is sent to Stream Analytics, analyzed, and sent for other actions like storage or presentation:



Key capabilities and benefits

Azure Stream Analytics is designed to be easy to use, flexible, reliable, and scalable to any job size. It is available across multiple Azure regions. The following image illustrates the key capabilities of Azure Stream Analytics:



Ease of getting started

Azure Stream Analytics is easy to start. It only takes a few clicks to connect to multiple sources and sinks, creating an end-to-end pipeline. Stream Analytics can connect to [Azure Event Hubs](#) and [Azure IoT Hub](#) for streaming data ingestion, as well as [Azure Blob storage](#) to ingest historical data. Job input can also include static or slow-changing reference data from Azure Blob storage or [SQL Database](#) that you can join to streaming data to perform lookup operations.

Stream Analytics can route job output to many storage systems such as [Azure Blob storage](#), [Azure SQL Database](#), [Azure Data Lake Store](#), and [Azure CosmosDB](#). You can run batch analytics on stored output with [Azure HDInsight](#), or you can send the output to another service, like Event Hubs for consumption or [Power BI](#) for real-time visualization.

For the entire list of Stream Analytics outputs, see [Understand outputs from Azure Stream Analytics](#).

Programmer productivity

Azure Stream Analytics uses a simple SQL-based query language that has been augmented with powerful temporal constraints to analyze data in motion. To define job transformations, you use a simple, declarative [Stream Analytics query language](#) that lets you author complex temporal queries and analytics using simple

SQL constructs. Because Stream Analytics query language is consistent to the SQL language, familiarity with SQL is sufficient to start creating jobs. You can also create jobs by using developer tools like Azure PowerShell, [Stream Analytics Visual Studio tools](#), the [Stream Analytics Visual Studio Code extension](#), or Azure Resource Manager templates. Using developer tools allow you to develop transformation queries offline and use the [CI/CD pipeline](#) to submit jobs to Azure.

The Stream Analytics query language offers a wide array of functions for analyzing and processing streaming data. This query language supports simple data manipulation, aggregation functions, and complex geospatial functions. You can edit queries in the portal and test them using sample data that is extracted from a live stream.

You can extend the capabilities of the query language by defining and invoking additional functions. You can define function calls in the Azure Machine Learning service to take advantage of Azure Machine Learning solutions, and integrate JavaScript or C# user-defined functions (UDFs) or user-defined aggregates to perform complex calculations as part a Stream Analytics query.

Fully managed

Azure Stream Analytics is a fully managed serverless (PaaS) offering on Azure. You don't have to provision any hardware or manage clusters to run your jobs. Azure Stream Analytics fully manages your job by setting up complex compute clusters in the cloud and taking care of the performance tuning necessary to run the job. Integration with Azure Event Hubs and Azure IoT Hub allows your job to ingest millions of events per second coming from a number of sources, to include connected devices, clickstreams, and log files. Using the partitioning feature of Event Hubs, you can partition computations into logical steps, each with the ability to be further partitioned to increase scalability.

Run in the cloud or on the intelligent edge

Azure Stream Analytics can run in the cloud, for large-scale analytics, or run on IoT Edge for ultra-low latency analytics. Azure Stream Analytics uses the same query language on both cloud and the edge, enabling developers to build truly hybrid architectures for stream processing.

Low total cost of ownership

As a cloud service, Stream Analytics is optimized for cost. There are no upfront costs involved - you only pay for the [streaming units you consume](#), and the amount of data processed. There is no commitment or cluster provisioning required, and you can scale the job up or down based on your business needs.

Mission-critical ready

Azure Stream Analytics is available across multiple regions worldwide and is designed to run mission-critical workloads by supporting reliability, security and compliance requirements.

Reliability

Azure Stream Analytics guarantees exactly-once event processing and at-least-once delivery of events, so events are never lost. Exactly-once processing is guaranteed with selected output as described in [Event Delivery Guarantees](#).

Azure Stream Analytics has built-in recovery capabilities in case the delivery of an event fails. Stream Analytics also provides built-in checkpoints to maintain the state of your job and provides repeatable results.

As a managed service, Stream Analytics guarantees event processing with a 99.9% availability at a minute level of granularity. For more information, see the [Stream Analytics SLA](#) page.

Security

In terms of security, Azure Stream Analytics encrypts all incoming and outgoing communications and supports TLS 1.2. Built-in checkpoints are also encrypted. Stream Analytics doesn't store the incoming data since all processing is done in-memory.

Compliance

Azure Stream Analytics follows multiple compliance certifications as described in the [overview of Azure compliance](#).

Performance

Stream Analytics can process millions of events every second and it can deliver results with ultra low latencies. It allows you to scale-up and scale-out to handle large real-time and complex event processing applications. Stream Analytics supports higher performance by partitioning, allowing complex queries to be parallelized and executed on multiple streaming nodes. Azure Stream Analytics is built on [Trill](#), a high-performance in-memory streaming analytics engine developed in collaboration with Microsoft Research.

Next steps

You now have an overview of Azure Stream Analytics. Next, you can dive deep and create your first Stream Analytics job:

- [Create a Stream Analytics job by using the Azure portal](#).
- [Create a Stream Analytics job by using Azure PowerShell](#).
- [Create a Stream Analytics job by using Visual Studio](#).
- [Create a Stream Analytics job by using Visual Studio Code](#).

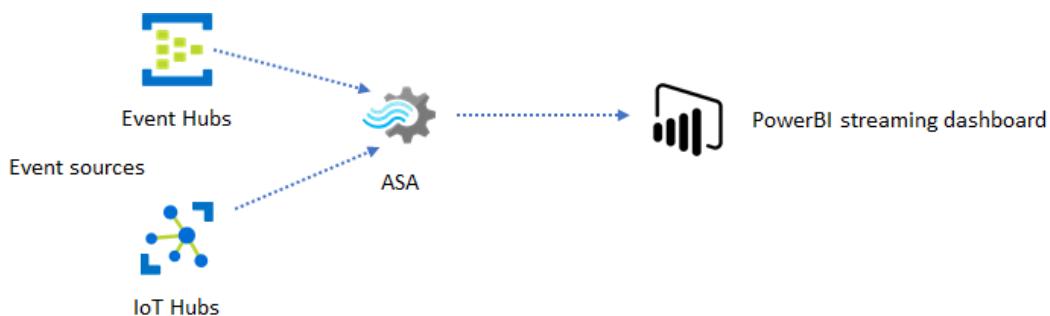
Azure Stream Analytics solution patterns

6/21/2019 • 13 minutes to read • [Edit Online](#)

Like many other services in Azure, Stream Analytics is best used with other services to create a larger end-to-end solution. This article discusses simple Azure Stream Analytics solutions and various architectural patterns. You can build on these patterns to develop more complex solutions. The patterns described in this article can be used in a wide variety of scenarios. Examples of scenario-specific patterns are available on [Azure solution architectures](#).

Create a Stream Analytics job to power real-time dashboarding experience

With Azure Stream Analytics, you can quickly stand up real-time dashboards and alerts. A simple solution ingests events from Event Hubs or IoT Hub, and [feeds the Power BI dashboard with a streaming data set](#). For more information, see the detailed tutorial [Analyze phone call data with Stream Analytics and visualize results in Power BI dashboard](#).

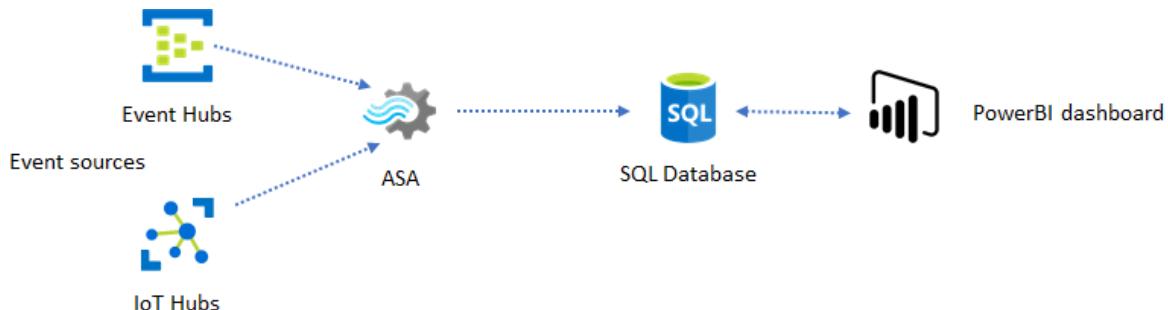


This solution can be built in just a few minutes from Azure portal. There is no extensive coding involved, and SQL language is used to express the business logic.

This solution pattern offers the lowest latency from the event source to the Power BI dashboard in a browser. Azure Stream Analytics is the only Azure service with this built-in capability.

Use SQL for dashboard

The Power BI dashboard offers low latency, but it cannot be used to produce full fledged Power BI reports. A common reporting pattern is to output your data to a SQL database first. Then use Power BI's SQL connector to query SQL for the latest data.



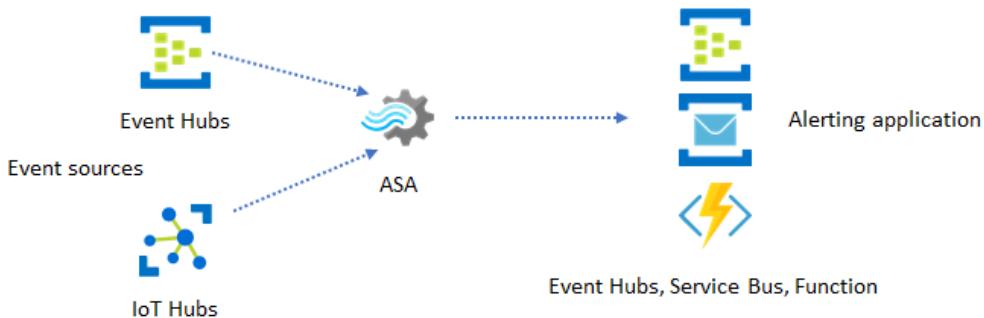
Using SQL database gives you more flexibility but at the expense of a slightly higher latency. This solution is optimal for jobs with latency requirements greater than one second. With this method, you can maximize Power BI capabilities to further slice and dice the data for reports, and much more visualization options. You also gain the flexibility of using other dashboard solutions, such as Tableau.

SQL is not a high throughput data store. The maximum throughput to a SQL database from Azure Stream Analytics is currently around 24 MB/s. If the event sources in your solution produce data at a higher rate, you need to use processing logic in Stream Analytics to reduce the output rate to SQL. Techniques such as filtering, windowed aggregates, pattern matching with temporal joins, and analytic functions can be used. The output rate to SQL can be further optimized using techniques described in [Azure Stream Analytics output to Azure SQL Database](#).

Incorporate real-time insights into your application with event messaging

The second most popular use of Stream Analytics is to generate real-time alerts. In this solution pattern, business logic in Stream Analytics can be used to detect [temporal and spatial patterns](#) or [anomalies](#), then produce alerting signals. However, unlike the dashboard solution where Stream Analytics uses Power BI as a preferred endpoint, a number of intermediate data sinks can be used. These sinks include Event Hubs, Service Bus, and Azure Functions. You, as the application builder, need to decide which data sink works best for your scenario.

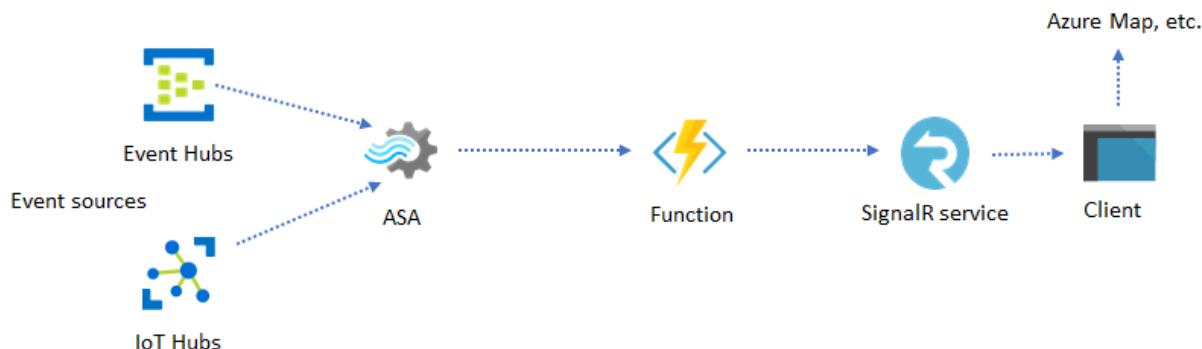
Downstream event consumer logic must be implemented to generate alerts in your existing business workflow. Because you can implement custom logic in Azure Functions, Azure Functions is the fastest way you can perform this integration. A tutorial for using Azure Function as the output for a Stream Analytics job can be found in [Run Azure Functions from Azure Stream Analytics jobs](#). Azure Functions also supports various types of notifications including text and email. Logic App may also be used for such integration, with Event Hubs between Stream Analytics and Logic App.



Event Hubs, on the other hand, offers the most flexible integration point. Many other services, like Azure Data Explorer and Time Series Insights can consume events from Event Hubs. Services can be connected directly to the Event Hubs sink from Azure Stream Analytics to complete the solution. Event Hubs is also the highest throughput messaging broker available on Azure for such integration scenarios.

Dynamic applications and websites

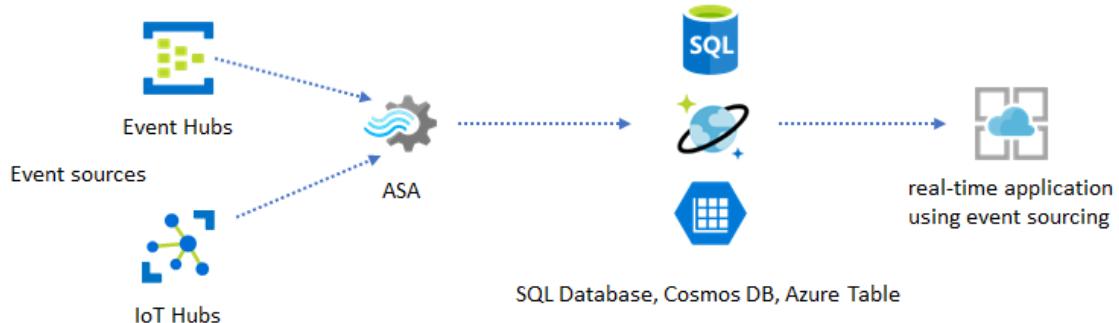
You can create custom real-time visualizations, such as dashboard or map visualization, using Azure Stream Analytics and Azure SignalR Service. Using SignalR, web clients can be updated and show dynamic content in real-time.



Incorporate real-time insights into your application through data stores

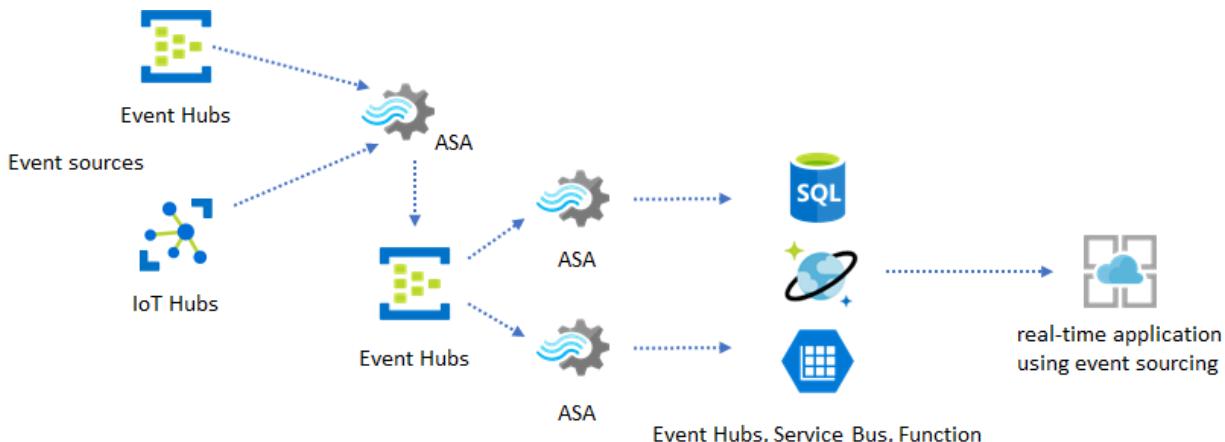
Most web services and web applications today use a request/response pattern to serve the presentation layer. The request/response pattern is simple to build and can be easily scaled with low response time using a stateless frontend and scalable stores, like Cosmos DB.

High data volume often creates performance bottlenecks in a CRUD-based system. The [event sourcing solution pattern](#) is used to address the performance bottlenecks. Temporal patterns and insights are also difficult and inefficient to extract from a traditional data store. Modern high-volume data driven applications often adopt a dataflow-based architecture. Azure Stream Analytics as the compute engine for data in motion is a linchpin in that architecture.



In this solution pattern, events are processed and aggregated into data stores by Azure Stream Analytics. The application layer interacts with data stores using the traditional request/response pattern. Because of Stream Analytics' ability to process a large number of events in real-time, the application is highly scalable without the need to bulk up the data store layer. The data store layer is essentially a materialized view in the system. [Azure Stream Analytics output to Azure Cosmos DB](#) describes how Cosmos DB is used as a Stream Analytics output.

In real applications where processing logic is complex and there is the need to upgrade certain parts of the logic independently, multiple Stream Analytics jobs can be composed together with Event Hubs as the intermediary event broker.



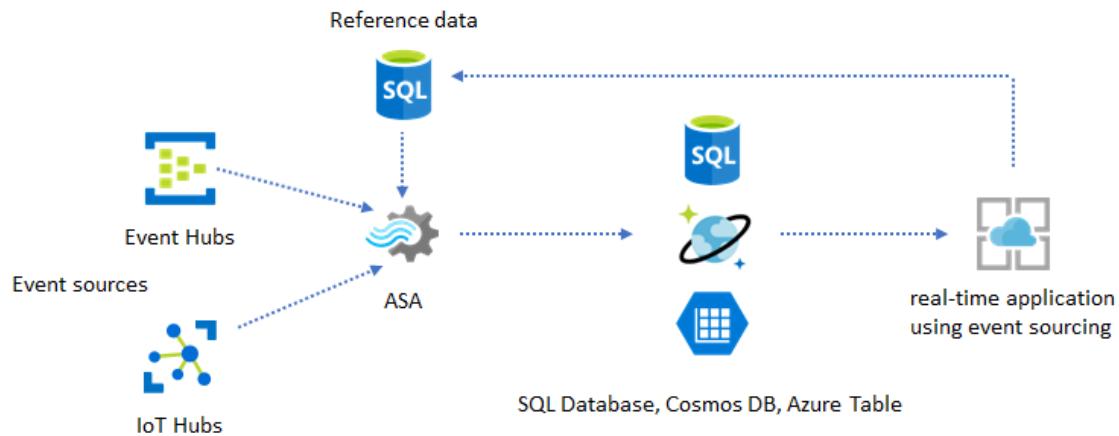
This pattern improves the resiliency and manageability of the system. However, even though Stream Analytics guarantees exactly once processing, there is a small chance that duplicate events may land in the intermediary Event Hubs. It's important for the downstream Stream Analytics job to dedupe events using logic keys in a lookback window. For more information on event delivery, see [Event Delivery Guarantees](#) reference.

Use reference data for application customization

The Azure Stream Analytics reference data feature is designed specifically for end-user customization like alerting threshold, processing rules, and [geofences](#). The application layer can accept parameter changes and store them in a SQL database. The Stream Analytics job periodically queries for changes from the database and makes the customization parameters accessible through a reference data join. For more information on how to use reference

data for application customization, see [SQL reference data](#) and [reference data join](#).

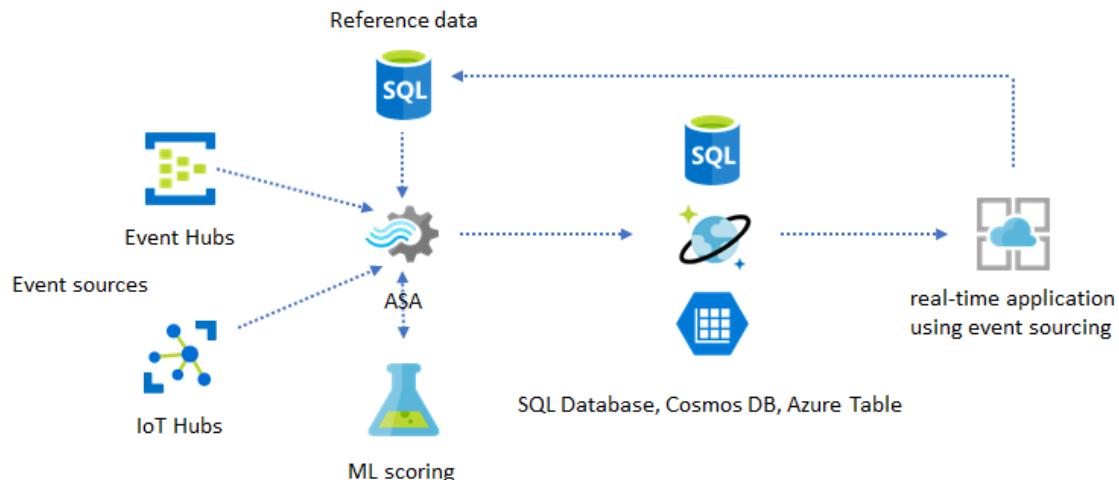
This pattern can also be used to implement a rules engine where the thresholds of the rules are defined from reference data. For more information on rules, see [Process configurable threshold-based rules in Azure Stream Analytics](#).



Add Machine Learning to your real-time insights

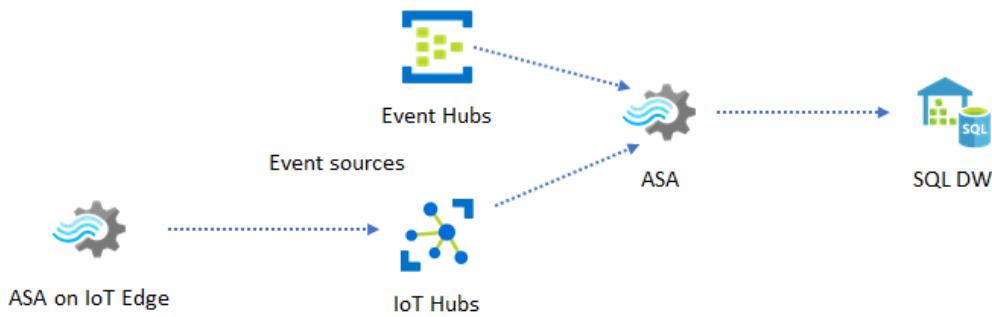
Azure Stream Analytics' built-in [Anomaly Detection model](#) is a convenient way to introduce Machine Learning to your real-time application. For a wider range of Machine Learning needs, see [Azure Stream Analytics integrates with Azure Machine Learning's scoring service](#).

For advanced users who want to incorporate online training and scoring into the same Stream Analytics pipeline, see this example of how do that with [linear regression](#).



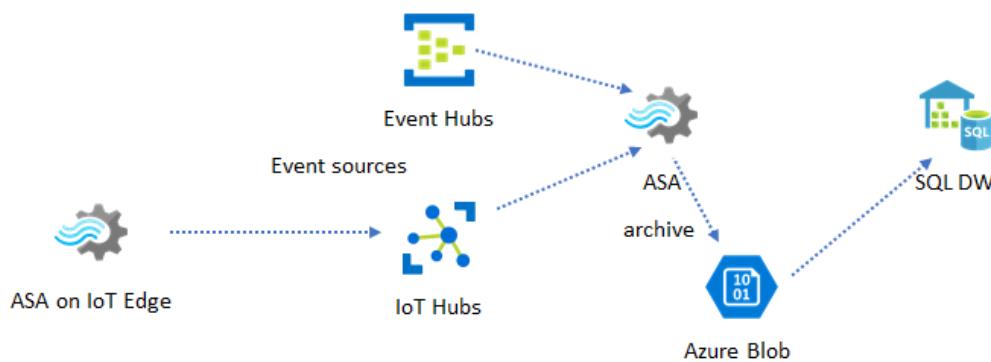
Near real-time data warehousing

Another common pattern is real-time data warehousing, also called streaming data warehouse. In addition to events arriving at Event Hubs and IoT Hub from your application, [Azure Stream Analytics running on IoT Edge](#) can be used to fulfill data cleansing, data reduction, and data store and forward needs. Stream Analytics running on IoT Edge can gracefully handle bandwidth limitation and connectivity issues in the system. The SQL output adapter can be used to output to SQL Data Warehouse; however, the maximum throughput is limited to 10 MB/s.



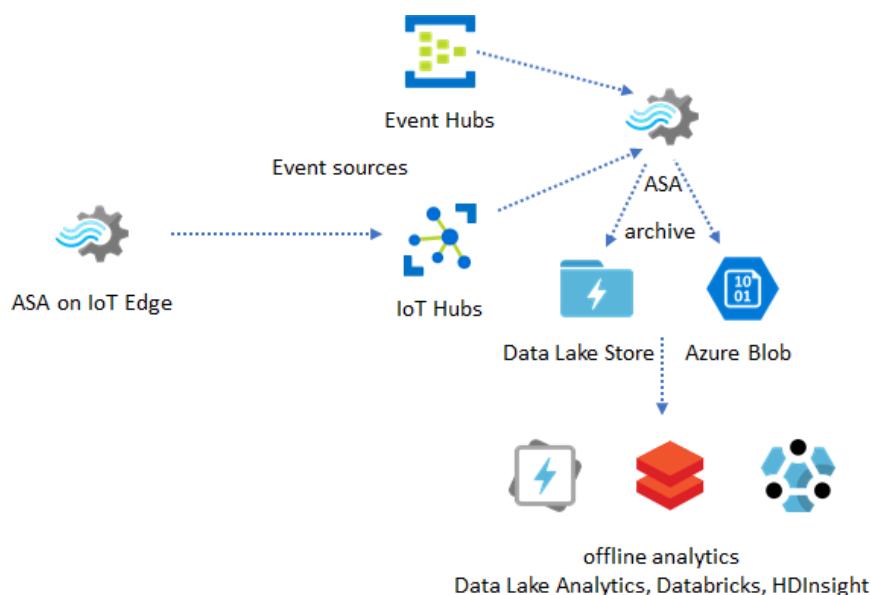
One way to improve the throughput with some latency tradeoff is to archive the events into Azure Blob storage, and then import them into SQL Data Warehouse with Polybase. You must manually stitch together output from Stream Analytics to blob storage and input from blob storage to SQL Data Warehouse by [archiving the data by timestamp](#) and importing periodically.

In this usage pattern, Azure Stream Analytics is used as a near real-time ETL engine. Newly arriving events are continuously transformed and stored for downstream analytics service consumption.



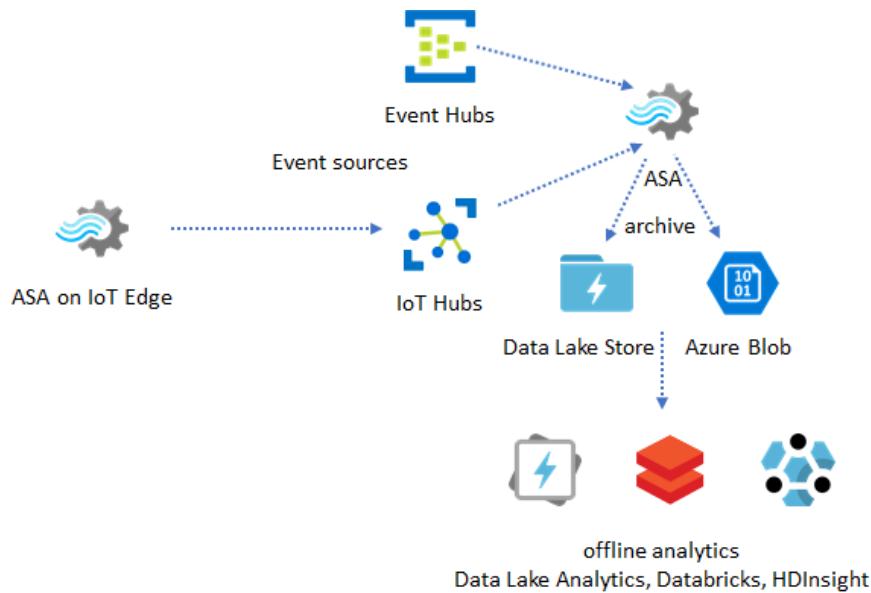
Archiving real-time data for analytics

Most data science and analytics activities still happen offline. Data can be archived by Azure Stream Analytics through Azure Data Lake Store Gen2 output and Parquet output formats. This capability removes the friction to feed data directly into Azure Data Lake Analytics, Azure Databricks, and Azure HDInsight. Azure Stream Analytics is used as a near real-time ETL engine in this solution. You can explore archived data in Data Lake using various compute engines.



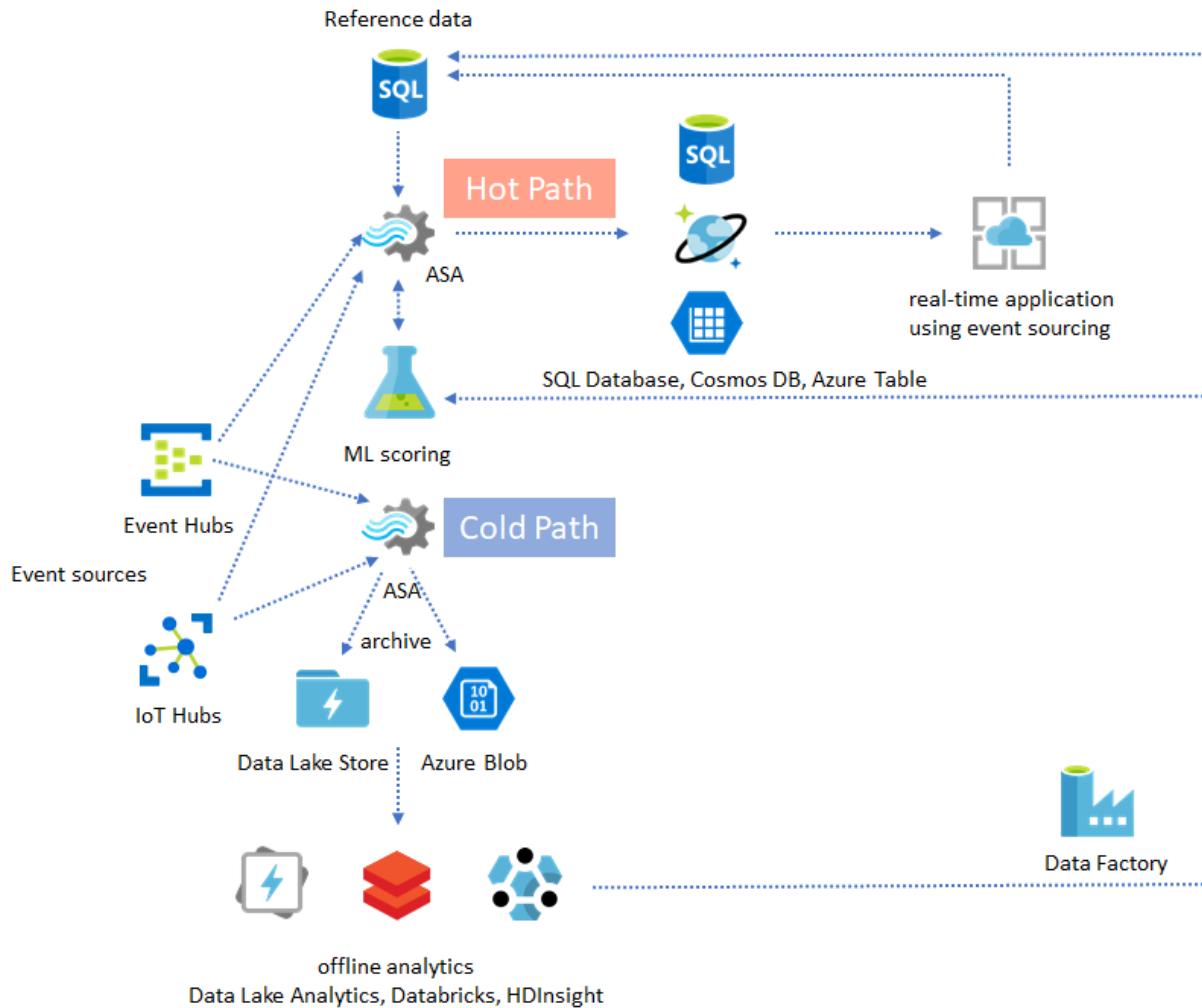
Use reference data for enrichment

Data enrichment is often a requirement for ETL engines. Azure Stream Analytics supports data enrichment with [reference data](#) from both SQL database and Azure Blob storage. Data enrichment can be done for data landing in both Azure Data Lake and SQL Data Warehouse.



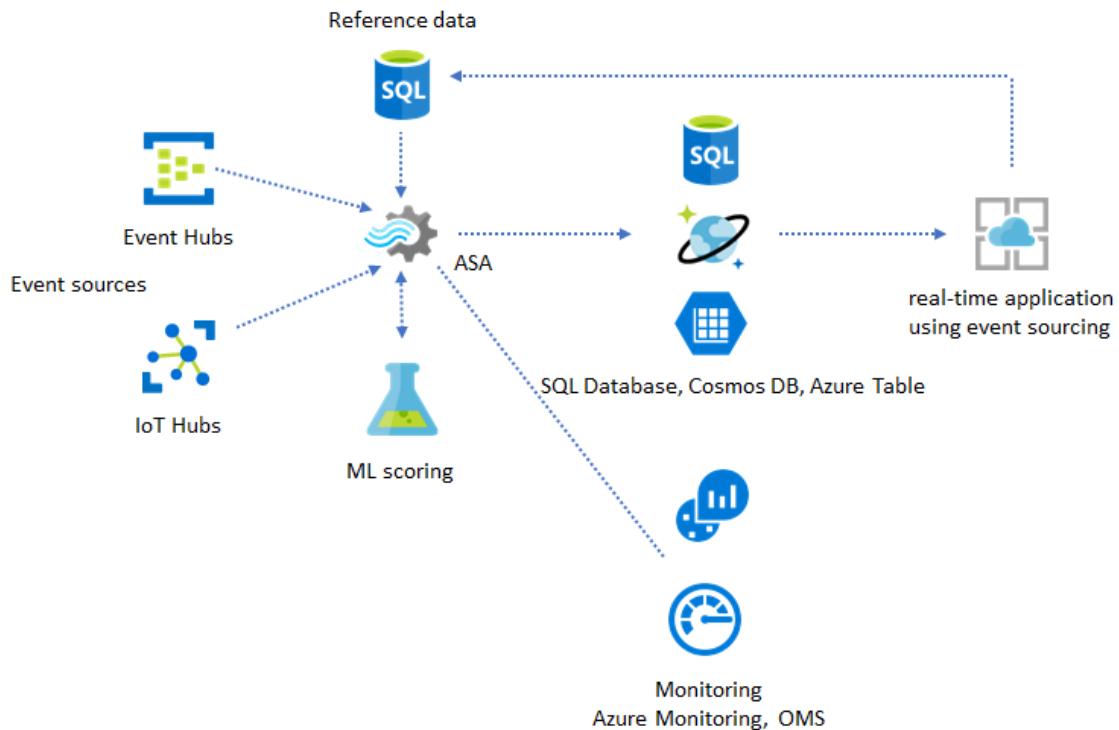
Operationalize insights from archived data

If you combine the offline analytics pattern with the near real-time application pattern, you can create a feedback loop. The feedback loop lets the application automatically adjust for changing patterns in the data. This feedback loop can be as simple as changing the threshold value for alerting, or as complex as retraining Machine Learning models. The same solution architecture can be applied to both ASA jobs running in the cloud and on IoT Edge.



How to monitor ASA jobs

An Azure Stream Analytics job can be run 24/7 to process incoming events continuously in real time. Its uptime guarantee is crucial to the health of the overall application. While Stream Analytics is the only streaming analytics service in the industry that offers a [99.9% availability guarantee](#), you may still incur some level of down time. Over the years, Stream Analytics has introduced metrics, logs, and job states to reflect the health of the jobs. All of them are surfaced through Azure Monitor service and can be further exported to OMS. For more information, see [Understand Stream Analytics job monitoring and how to monitor queries](#).



There are two key things to monitor:

- [Job failed state](#)

First and foremost, you need to make sure the job is running. Without the job in the running state, no new metrics or logs are generated. Jobs can change to a failed state for various reasons, including having a high SU utilization level (i.e., running out of resources).

- [Watermark delay metrics](#)

This metric reflects how far behind your processing pipeline is in wall clock time (seconds). Some of the delay is attributed to the inherent processing logic. As a result, monitoring the increasing trend is much more important than monitoring the absolute value. The steady state delay should be addressed by your application design, not by monitoring or alerts.

Upon failure, activity logs and [diagnostics logs](#) are the best places to begin looking for errors.

Build resilient and mission critical applications

Regardless of Azure Stream Analytics' SLA guarantee and how careful you run your end-to-end application, outages happen. If your application is mission critical, you need to be prepared for outages in order to recover gracefully.

For alerting applications, the most important thing is to detect the next alert. You may choose to restart the job from the current time when recovering, ignoring past alerts. The job start time semantics are by the first output time, not the first input time. The input is rewound backwards an appropriate amount of time to guarantee the first output at the specified time is complete and correct. You won't get partial aggregates and trigger alerts.

unexpectedly as a result.

You may also choose to start output from some amount of time in the past. Both Event Hubs and IoT Hub's retention policies hold a reasonable amount of data to allow processing from the past. The tradeoff is how fast you can catch up to the current time and start to generate timely new alerts. Data loses its value rapidly over time, so it's important to catch up to the current time quickly. There are two ways to catch up quickly:

- Provision more resources (SU) when catching up.
- Restart from current time.

Restarting from current the time is simple to do, with the tradeoff of leaving a gap during processing. Restarting this way might be OK for alerting scenarios, but can be problematic for dashboard scenarios and is a non-starter for archiving and data warehousing scenarios.

Provisioning more resources can speed up the process, but the effect of having a processing rate surge is complex.

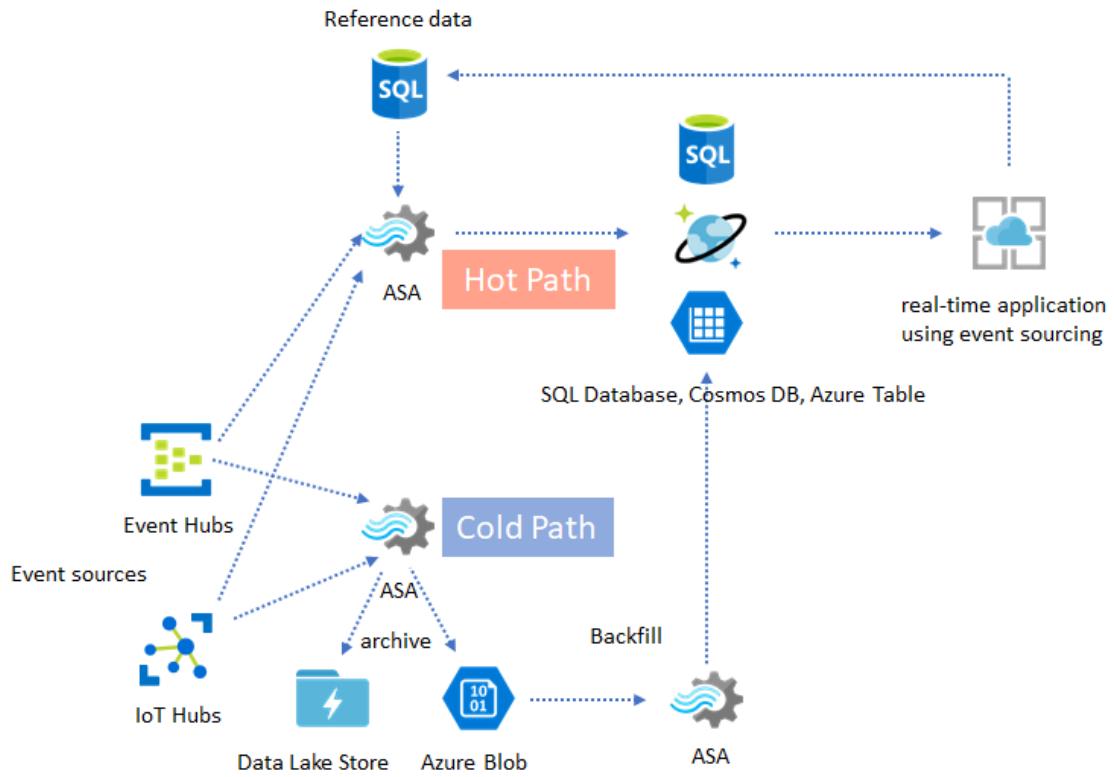
- Test that your job is scalable to a larger number of SUs. Not all queries are scalable. You need to make sure your query is [parallelized](#).
- Make sure there are enough partitions in the upstream Event Hubs or IoT Hub that you can add more Throughput Units (TUs) to scale the input throughput. Remember, each Event Hubs TU maxes out at an output rate of 2 MB/s.
- Make sure you have provisioned enough resources in the output sinks (i.e., SQL Database, Cosmos DB), so they don't throttle the surge in output, which can sometimes cause the system to lock up.

The most important thing is to anticipate the processing rate change, test these scenarios before going into production, and be ready to scale the processing correctly during failure recovery time.

In the extreme scenario that incoming events are all delayed, [it's possible all the delayed events are dropped](#) if you have applied a late arriving window to your job. The dropping of the events may appear to be a mysterious behavior at the beginning; however, considering Stream Analytics is a real-time processing engine, it expects incoming events to be close to the wall clock time. It has to drop events that violate these constraints.

Backfilling process

Fortunately, the previous data archiving pattern can be used to process these late events gracefully. The idea is that the archiving job processes incoming events in arrival time and archives events into the right time bucket in Azure Blob or Azure Data Lake Store with their event time. It doesn't matter how late an event arrives, it will never be dropped. It will always land in the right time bucket. During recovery, it's possible to reprocess the archived events and backfill the results to the store of choice.



The backfill process has to be done with an offline batch processing system, which most likely has a different programming model than Azure Stream Analytics. This means you have to re-implement the entire processing logic.

For backfilling, it's still important to at least temporarily provision more resource to the output sinks to handle higher throughput than the steady state processing needs.

SCENARIOS	RESTART FROM NOW ONLY	RESTART FROM LAST STOPPED TIME	RESTART FROM NOW + BACKFILL WITH ARCHIVED EVENTS
Dashboarding	Creates gap	OK for short outage	Use for long outage
Alerting	Acceptable	OK for short outage	Not necessary
Event sourcing app	Acceptable	OK for short outage	Use for long outage
Data warehousing	Data loss	Acceptable	Not necessary
Offline analytics	Data loss	Acceptable	Not necessary

Putting it all together

It's not hard to imagine that all the solution patterns mentioned above can be combined together in a complex end-to-end system. The combined system can include dashboards, alerting, event sourcing application, data warehousing, and offline analytics capabilities.

The key is to design your system in composable patterns, so each subsystem can be built, tested, upgraded, and recovered independently.

Next steps

You now have seen a variety of solution patterns using Azure Stream Analytics. Next, you can dive deep and create

your first Stream Analytics job:

- [Create a Stream Analytics job by using the Azure portal.](#)
- [Create a Stream Analytics job by using Azure PowerShell.](#)
- [Create a Stream Analytics job by using Visual Studio.](#)

Quickstart: Create a Stream Analytics job by using the Azure portal

6/4/2019 • 6 minutes to read • [Edit Online](#)

This quickstart shows you how to get started with creating a Stream Analytics job. In this quickstart, you define a Stream Analytics job that reads real-time streaming data and filters messages with a temperature greater than 27. Your Stream Analytics job will read data from an IoT Hub device, transform the data, and write the data back to a container in blob storage. The input data used in this quickstart is generated by a Raspberry Pi online simulator.

Before you begin

- If you don't have an Azure subscription, create a [free account](#).
- Sign in to the [Azure portal](#).

Prepare the input data

Before defining the Stream Analytics job, you should prepare the data, which is later configured as the job input. To prepare the input data required by the job, complete the following steps:

1. Sign in to the [Azure portal](#).
2. Select **Create a resource > Internet of Things > IoT Hub**.
3. In the **IoT Hub** pane, enter the following information:

SETTING	SUGGESTED VALUE	DESCRIPTION
Subscription	<Your subscription>	Select the Azure subscription that you want to use.
Resource group	asaquickstart-resourcegroup	Select Create New and enter a new resource-group name for your account.
Region	<Select the region that is closest to your users>	Select a geographic location where you can host your IoT Hub. Use the location that's closest to your users.
IoT Hub Name	MyASAIoTHub	Select a name for your IoT Hub.

The screenshot shows the Microsoft Azure portal with the URL <https://portal.azure.com/Microsoft.IoTHub>. On the left, there's a sidebar titled 'New' with a search bar for 'Search the Marketplace'. Below it is the 'Azure Marketplace' section with various service categories like Compute, Storage, and Internet of Things. The 'Internet of Things' category is highlighted with a red box. Within this category, the 'IoT Hub' service is selected and highlighted with a red box. The main right pane is titled 'IoT hub' and contains tabs for 'Basics', 'Size and scale', and 'Review + create'. Under 'PROJECT DETAILS', there are fields for 'Subscription' (selected), 'Resource Group' (with a 'Create new' button), 'Region' (set to 'West US'), and 'IoT Hub Name' (set to 'MyASAIoTHub'). All these fields are also highlighted with a red box.

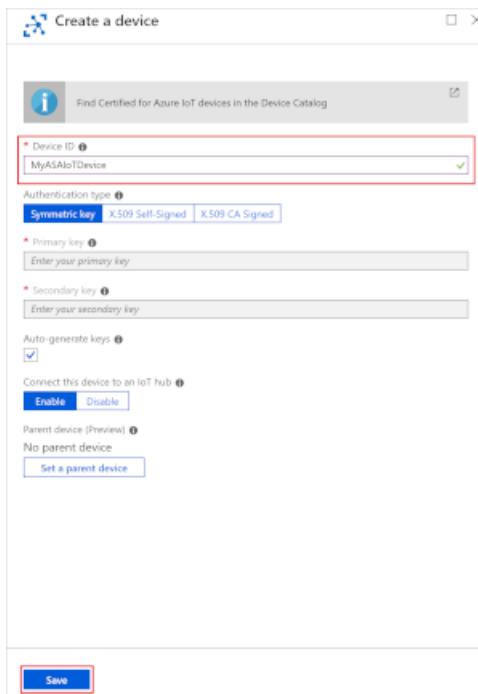
4. Select **Next: Set size and scale**.

5. Choose your **Pricing and scale tier**. For this quickstart, select the **F1 - Free** tier if it's still available on your subscription. For more information, see [IoT Hub pricing](#).

The screenshot shows the 'Size and scale' blade for creating an IoT hub. At the top, there are tabs for 'Basics', 'Size and scale' (which is selected and highlighted with a red box), and 'Review + create'. A note below says 'Each IoT Hub is provisioned with a certain number of units in a specific tier. The tier and number of units determine the maximum daily quota of messages that you can send.' In the 'SCALE TIER AND UNITS' section, there's a dropdown for 'Pricing and scale tier' with 'S1: Standard tier' selected (highlighted with a red box). Below it is a slider for 'Number of S1 IoT Hub units' set to 1, with a note: 'This determines your IoT Hub scale capability and can be changed as your need increases.' The main area lists features: 'Pricing and scale tier S1', 'Device-to-cloud-messages Enabled', 'Messages per day 400,000', 'Message routing Enabled', 'Cost per month <price>', 'Cloud-to-device commands Enabled', 'IoT Edge Enabled', and 'Device management Enabled'. At the bottom, there are buttons for 'Review + create', '< Previous: Basics', and 'Automation options'.

6. Select **Review + create**. Review your IoT Hub information and click **Create**. Your IoT Hub might take a few minutes to create. You can monitor the progress in the **Notifications** pane.

7. In your IoT Hub navigation menu, click **Add** under **IoT devices**. Add a **Device ID** and click **Save**.



8. Once the device is created, open the device from the **IoT devices** list. Copy the **Connection string -- primary key** and save it to a notepad to use later.

MODULE IDENTITY NAME	CONNECTION STATE	CONNECTION STATE LAST UPDATED	LAST ACTIVITY TIME
No module identities listed.			

Create blob storage

- From the upper left-hand corner of the Azure portal, select **Create a resource > Storage > Storage account**.
- In the **Create storage account** pane, enter a storage account name, location, and resource group. Choose the same location and resource group as the IoT Hub you created. Then click **Review + create** to create the account.

Create storage account

Basics Advanced Tags Review + create

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below. [Learn more](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription * Resource group Create new

INSTANCE DETAILS

The default deployment model is Resource Manager, which supports the latest Azure features. You may choose to deploy using the classic deployment model. [Choose classic deployment model](#)

* Storage account name * Location

Performance Standard Premium

Account kind

Replication

Access tier (default) Hot Cool

Review + create Previous Next: Advanced >

3. Once your storage account is created, select the **Blobs** tile on the **Overview** pane.

myasaquickstartstorage Storage account

Search (Ctrl+)

Overview Open in Explorer Move Delete Refresh

Resource group (change) <resource group>

Status Primary: Available

Location West US

Subscription (change) <subscription>

Subscription ID <subscription id>

Tags (change) Click here to add tags

Performance/Access tier Standard/Hot

Replication Locally-redundant storage (LRS)

Account kind StorageV2 (general purpose v2)

Services

Blobs REST-based object storage for unstructured data Explore data using OAuth preview Learn more

Files File shares that use the standard SMB 3.0 protocol Learn more

Tables Tabular data storage Learn more

Queues Effectively scale apps according to traffic Explore data using OAuth preview Learn more

Monitoring

Show data for last: 1 hour 6 hours 12 hours 1 day 7 days Account

Total egress Total ingress

4. From the **Blob Service** page, select **Container** and provide a name for your container, such as **container1**. Leave the **Public access level** as **Private (no anonymous access)** and select **OK**.

Container Refresh Delete

New container

* Name

Public access level

OK Cancel

Create a Stream Analytics job

1. Sign in to the Azure portal.
2. Select **Create a resource** in the upper left-hand corner of the Azure portal.

3. Select **Analytics > Stream Analytics job** from the results list.

4. Fill out the Stream Analytics job page with the following information:

SETTING	SUGGESTED VALUE	DESCRIPTION
Job name	MyASAJob	Enter a name to identify your Stream Analytics job. Stream Analytics job name can contain alphanumeric characters, hyphens, and underscores only and it must be between 3 and 63 characters long.
Subscription	<Your subscription>	Select the Azure subscription that you want to use for this job.
Resource group	asaquickstart-resourcegroup	Select the same resource group as your IoT Hub.
Location	<Select the region that is closest to your users>	Select geographic location where you can host your Stream Analytics job. Use the location that's closest to your users for better performance and to reduce the data transfer cost.
Streaming units	1	Streaming units represent the computing resources that are required to execute a job. By default, this value is set to 1. To learn about scaling streaming units, refer to understanding and adjusting streaming units article.
Hosting environment	Cloud	Stream Analytics jobs can be deployed to cloud or edge. Cloud allows you to deploy to Azure Cloud, and Edge allows you to deploy to an IoT Edge device.

5. Check the **Pin to dashboard** box to place your job on your dashboard and then select **Create**.
6. You should see a *Deployment in progress...* notification displayed in the top right of your browser window.

Configure job input

In this section, you will configure an IoT Hub device input to the Stream Analytics job. Use the IoT Hub you created in the previous section of the quickstart.

1. Navigate to your Stream Analytics job.
2. Select **Inputs > Add Stream input > IoT Hub**.
3. Fill out the **IoT Hub** page with the following values:

SETTING	SUGGESTED VALUE	DESCRIPTION
Input alias	IoTHubInput	Enter a name to identify the job's input.
Subscription	<Your subscription>	Select the Azure subscription that has the storage account you created. The storage account can be in the same or in a different subscription. This example assumes that you have created storage account in the same subscription.

SETTING	SUGGESTED VALUE	DESCRIPTION
IoT Hub	MyASAIoTHub	Enter the name of the IoT Hub you created in the previous section.

4. Leave other options to default values and select **Save** to save the settings.

Configure job output

1. Navigate to the Stream Analytics job that you created earlier.
2. Select **Outputs > Add > Blob storage**.
3. Fill out the **Blob storage** page with the following values:

SETTING	SUGGESTED VALUE	DESCRIPTION
Output alias	BlobOutput	Enter a name to identify the job's output.
Subscription	<Your subscription>	Select the Azure subscription that has the storage account you created. The storage account can be in the same or in a different subscription. This example assumes that you have created storage account in the same subscription.
Storage account	asaquickstartstorage	Choose or enter the name of the storage account. Storage account names are automatically detected if they are created in the same subscription.
Container	container1	Select the existing container that you created in your storage account.

- Leave other options to default values and select **Save** to save the settings.

The screenshot shows the Azure Stream Analytics job configuration interface. On the left, the navigation pane is open with the 'Outputs' section highlighted. In the main area, a 'Sink' section is displayed, with 'Event Hub' and 'Blob storage' listed. A red box highlights 'Blob storage'. To the right, a detailed configuration dialog for 'Blob storage' is shown. It includes fields for 'Output alias' (set to 'BlobOutput'), 'Storage account' (set to 'myasaquickstartstorage'), 'Container' (set to 'container1'), and a 'Save' button at the bottom.

Define the transformation query

1. Navigate to the Stream Analytics job that you created earlier.
2. Select **Query** and update the query as follows:

```
SELECT *
INTO BlobOutput
FROM IoTHubInput
HAVING Temperature > 27
```

3. In this example, the query reads the data from IoT Hub and copies it to a new file in the blob. Select **Save**.

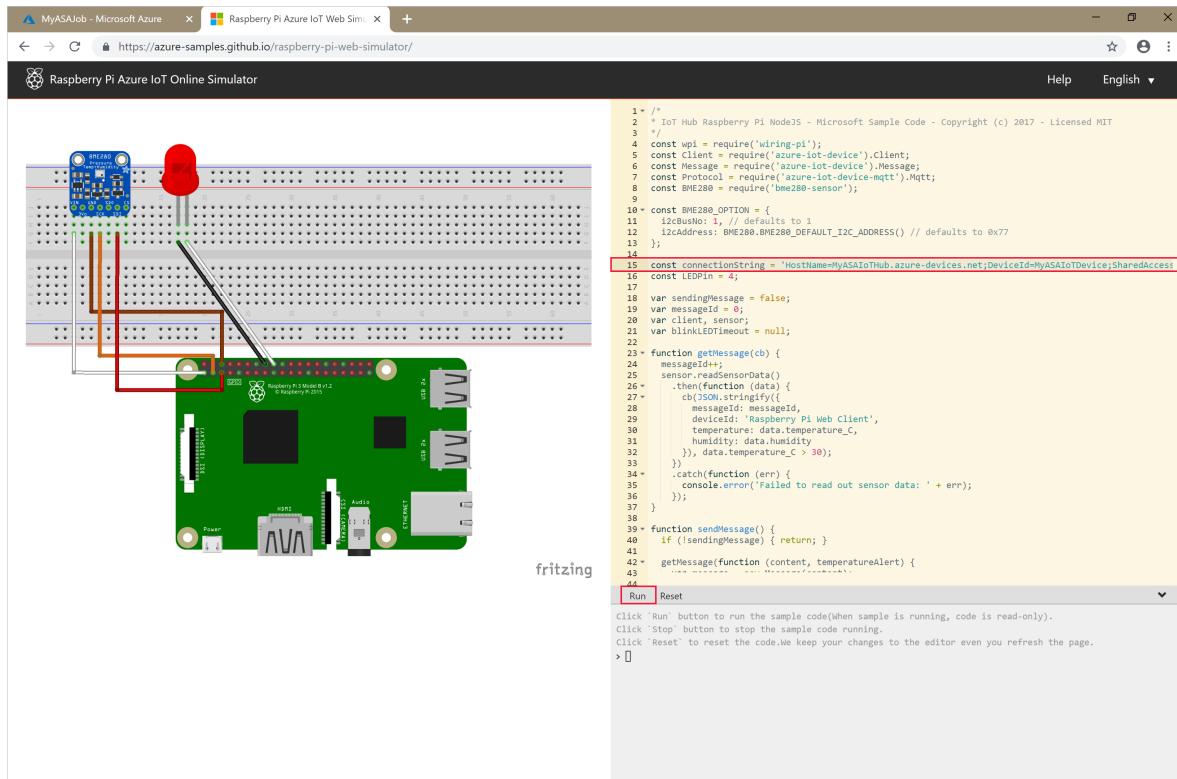
The screenshot shows the Azure Stream Analytics job configuration interface. The 'Query' section is highlighted in the navigation pane. The main area displays the transformation query code in a syntax-highlighted editor. The code is:

```
1 SELECT
2 *
3 INTO
4 BlobOutput
5 FROM
6 IoTHubInput
7 HAVING temperature > 27
```

A red box highlights the 'Save' button at the top of the editor. The status bar at the bottom indicates: 'Your query could be put in logs that are in a potentially different geography.' and 'Missing some language constructs? Let us know! (Powered by UserVoice - Privacy Policy)'

Run the IoT simulator

1. Open the [Raspberry Pi Azure IoT Online Simulator](#).
2. Replace the placeholder in Line 15 with the Azure IoT Hub device connection string you saved in a previous section.
3. Click **Run**. The output should show the sensor data and messages that are being sent to your IoT Hub.



Start the Stream Analytics job and check the output

1. Return to the job overview page and select **Start**.
2. Under **Start job**, select **Now**, for the **Job output start time** field. Then, select **Start** to start your job.
3. After few minutes, in the portal, find the storage account & the container that you have configured as output for the job. You can now see the output file in the container. The job takes a few minutes to start for the first time, after it is started, it will continue to run as the data arrives.

The screenshot shows the Azure Storage Explorer interface. On the left, there's a navigation tree with 'container1' selected. The main area has tabs for 'Overview', 'Snapshots', 'Edit blob' (which is highlighted with a red box), and 'Generate SAS'. The 'Edit blob' tab displays a JSON array of 30 objects, each representing a message with a messageId, deviceId, temperature, and humidity. The JSON is as follows:

```

[{"messageId": "775", "deviceId": "Raspberry Pi Web Client", "temperature": 27.56286555465898, "humidity": 77}, {"messageId": "776", "deviceId": "Raspberry Pi Web Client", "temperature": 31.76983389970918, "humidity": 64}, {"messageId": "777", "deviceId": "Raspberry Pi Web Client", "temperature": 30.173582241045128, "humidity": 61}, {"messageId": "778", "deviceId": "Raspberry Pi Web Client", "temperature": 29.70036239344644, "humidity": 70}, {"messageId": "779", "deviceId": "Raspberry Pi Web Client", "temperature": 29.537158745343632, "humidity": 73}, {"messageId": "780", "deviceId": "Raspberry Pi Web Client", "temperature": 30.11726573198575, "humidity": 66}, {"messageId": "783", "deviceId": "Raspberry Pi Web Client", "temperature": 27.237104885232633, "humidity": 63}, {"messageId": "786", "deviceId": "Raspberry Pi Web Client", "temperature": 30.541928099495646, "humidity": 66}, {"messageId": "789", "deviceId": "Raspberry Pi Web Client", "temperature": 30.46121575792222, "humidity": 61}, {"messageId": "791", "deviceId": "Raspberry Pi Web Client", "temperature": 31.687045217662682, "humidity": 75}, {"messageId": "792", "deviceId": "Raspberry Pi Web Client", "temperature": 29.0065290125080579, "humidity": 70}, {"messageId": "794", "deviceId": "Raspberry Pi Web Client", "temperature": 28.859692581436892, "humidity": 69}, {"messageId": "796", "deviceId": "Raspberry Pi Web Client", "temperature": 30.702890613043248, "humidity": 71}, {"messageId": "797", "deviceId": "Raspberry Pi Web Client", "temperature": 29.466947493481154, "humidity": 63}, {"messageId": "798", "deviceId": "Raspberry Pi Web Client", "temperature": 31.1188081842064, "humidity": 64}, {"messageId": "801", "deviceId": "Raspberry Pi Web Client", "temperature": 27.136869152806462, "humidity": 64}, {"messageId": "802", "deviceId": "Raspberry Pi Web Client", "temperature": 31.197715957953303, "humidity": 60}, {"messageId": "803", "deviceId": "Raspberry Pi Web Client", "temperature": 31.612204161725575, "humidity": 60}, {"messageId": "804", "deviceId": "Raspberry Pi Web Client", "temperature": 31.733574107399651, "humidity": 68}, {"messageId": "805", "deviceId": "Raspberry Pi Web Client", "temperature": 29.535226236689539, "humidity": 62}, {"messageId": "806", "deviceId": "Raspberry Pi Web Client", "temperature": 27.866822678967593, "humidity": 66}, {"messageId": "807", "deviceId": "Raspberry Pi Web Client", "temperature": 27.775068973730495, "humidity": 60}, {"messageId": "808", "deviceId": "Raspberry Pi Web Client", "temperature": 30.5594159223502, "humidity": 69.1}, {"messageId": "810", "deviceId": "Raspberry Pi Web Client", "temperature": 30.96622244545026, "humidity": 64}, {"messageId": "811", "deviceId": "Raspberry Pi Web Client", "temperature": 29.411526757613782, "humidity": 69}, {"messageId": "812", "deviceId": "Raspberry Pi Web Client", "temperature": 28.086233098841412, "humidity": 70}, {"messageId": "822", "deviceId": "Raspberry Pi Web Client", "temperature": 30.849464815252702, "humidity": 64}, {"messageId": "823", "deviceId": "Raspberry Pi Web Client", "temperature": 27.634944188044333, "humidity": 62}, {"messageId": "827", "deviceId": "Raspberry Pi Web Client", "temperature": 31.171862047724645, "humidity": 66}, {"messageId": "831", "deviceId": "Raspberry Pi Web Client", "temperature": 31.748161497954346, "humidity": 64}

```

Clean up resources

When no longer needed, delete the resource group, the streaming job, and all related resources. Deleting the job avoids billing the streaming units consumed by the job. If you're planning to use the job in future, you can stop it and restart it later when you need. If you are not going to continue to use this job, delete all resources created by this quickstart by using the following steps:

1. From the left-hand menu in the Azure portal, select **Resource groups** and then select the name of the resource you created.
2. On your resource group page, select **Delete**, type the name of the resource to delete in the text box, and then select **Delete**.

Next steps

In this quickstart, you deployed a simple Stream Analytics job using Azure portal. You can also deploy Stream Analytics jobs using [PowerShell](#), [Visual Studio](#), and [Visual Studio Code](#).

To learn about configuring other input sources and performing real-time detection, continue to the following article:

[Real-time fraud detection using Azure Stream Analytics](#)

Quickstart: Create a Stream Analytics job using Azure PowerShell

3/26/2019 • 9 minutes to read • [Edit Online](#)

The Azure PowerShell module is used to create and manage Azure resources using PowerShell cmdlets or scripts. This quickstart details using the Azure PowerShell module to deploy and run an Azure Stream Analytics job.

The example job reads streaming data from an IoT Hub device. The input data is generated by a Raspberry Pi online simulator. Next, the Stream Analytics job transforms the data using the Stream Analytics query language to filter messages with a temperature greater than 27°. Finally, it writes the resulting output events into a file in blob storage.

Before you begin

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

- If you don't have an Azure subscription, create a [free account](#).
- This quickstart requires the Azure PowerShell module. Run `Get-Module -ListAvailable Az` to find the version that is installed on your local machine. If you need to install or upgrade, see [Install Azure PowerShell module](#).
- Some IoT Hub actions are not supported by Azure PowerShell and must be completed using Azure CLI version 2.0.24 or later and the IoT extension for Azure CLI. [Install the Azure CLI](#) and use `az extension add --name azure-cli-iot-ext` to install the IoT extension.

Sign in to Azure

Sign in to your Azure subscription with the `Connect-AzAccount` command, and enter your Azure credentials in the pop-up browser:

```
# Connect to your Azure account
Connect-AzAccount
```

If you have more than one subscription, select the subscription you would like to use for this quickstart by running the following cmdlets. Make sure to replace `<your subscription name>` with the name of your subscription:

```
# List all available subscriptions.
Get-AzSubscription

# Select the Azure subscription you want to use to create the resource group and resources.
Get-AzSubscription -SubscriptionName "<your subscription name>" | Select-AzSubscription
```

Create a resource group

Create an Azure resource group with [New-AzResourceGroup](#). A resource group is a logical container into which Azure resources are deployed and managed.

```
$resourceGroup = "StreamAnalyticsRG"  
$location = "WestUS2"  
New-AzResourceGroup `  
    -Name $resourceGroup `  
    -Location $location
```

Prepare the input data

Before defining the Stream Analytics job, prepare the data that is configured as input to the job.

The following Azure CLI code block does many commands to prepare the input data required by the job. Review the sections to understand the code.

1. In your PowerShell window, run the [az login](#) command to sign in to your Azure account.

When you successfully sign in, Azure CLI returns a list of your subscriptions. Copy the subscription you're using for this quickstart and run the [az account set](#) command to select that subscription. Choose the same subscription you selected in the previous section with PowerShell. Make sure to replace `<your subscription name>` with the name of your subscription.

```
az login  
  
az account set --subscription "<your subscription>"
```

2. Create an IoT Hub using the [az iot hub create](#) command. This example creates an IoT Hub called **MyASAIoTHub**. Because IoT Hub names are unique, you need to come up with your own IoT Hub name. Set the SKU to F1 to use the free tier if it is available with your subscription. If not, choose the next lowest tier.

```
az iot hub create --name "<your IoT Hub name>" --resource-group $resourceGroup --sku S1
```

Once the IoT hub has been created, get the IoT Hub connection string using the [az iot hub show-connection-string](#) command. Copy the entire connection string and save it for when you add the IoT Hub as input to your Stream Analytics job.

```
az iot hub show-connection-string --hub-name "MyASAIoTHub"
```

3. Add a device to IoT Hub using the [az iot hub device-identity create](#) command. This example creates a device called **MyASAIoTDevice**.

```
az iot hub device-identity create --hub-name "MyASAIoTHub" --device-id "MyASAIoTDevice"
```

4. Get the device connection string using the [az iot hub device-identity show-connection-string](#) command. Copy the entire connection string and save it for when you create the Raspberry Pi simulator.

```
az iot hub device-identity show-connection-string --hub-name "MyASAIoTHub" --device-id "MyASAIoTDevice" `  
    --output table
```

Output example:

```
HostName=MyASAIoTHub.azure-
devices.net;DeviceId=MyASAIoTDevice;SharedAccessKey=a2mnUsg52+NIgYudxYYUNXI67r0JmNubmfVafojG8=
```

Create blob storage

The following Azure PowerShell code block uses commands to create blob storage that is used for job output. Review the sections to understand the code.

1. Create a standard general-purpose storage account using [New-AzStorageAccount](#) cmdlet. This example creates a storage account called **myasaquickstartstorage** with locally redundant storage(LRS) and blob encryption (enabled by default).
2. Retrieve the storage account context `$storageAccount.Context` that defines the storage account to be used. When working with storage accounts, you reference the context instead of repeatedly providing the credentials.
3. Create a storage container using [New-AzStorageContainer](#).
4. Copy the storage key that is outputted by the code, and save that key to create the streaming job's output later on.

```
$storageAccountName = "myasaquickstartstorage"
$storageAccount = New-AzStorageAccount ` 
    -ResourceGroupName $resourceGroup ` 
    -Name $storageAccountName ` 
    -Location $location ` 
    -SkuName Standard_LRS ` 
    -Kind Storage

$ctx = $storageAccount.Context
$containerName = "container1"

New-AzStorageContainer ` 
    -Name $containerName ` 
    -Context $ctx

$storageAccountKey = (Get-AzStorageAccountKey ` 
    -ResourceGroupName $resourceGroup ` 
    -Name $storageAccountName).Value[0]

Write-Host "The <storage account key> placeholder needs to be replaced in your output json files with this key value:"
Write-Host $storageAccountKey -ForegroundColor Cyan
```

Create a Stream Analytics job

Create a Stream Analytics job with [New-AzStreamAnalyticsJob](#) cmdlet. This cmdlet takes the job name, resource group name, and job definition as parameters. The job name can be any friendly name that identifies your job. It can have alphanumeric characters, hyphens, and underscores only and it must be between 3 and 63 characters long. The job definition is a JSON file that contains the properties required to create a job. On your local machine, create a file named `JobDefinition.json` and add the following JSON data to it:

```
{
  "location": "WestUS2",
  "properties": {
    "sku": {
      "name": "standard"
    },
    "eventsOutOfOrderPolicy": "adjust",
    "eventsOutOfOrderMaxDelayInSeconds": 10,
    "compatibilityLevel": 1.1
  }
}
```

Next, run the `New-AzStreamAnalyticsJob` cmdlet. Replace the value of `jobDefinitionFile` variable with the path where you've stored the job definition JSON file.

```
$jobName = "MyStreamingJob"
$jobDefinitionFile = "C:\JobDefinition.json"
New-AzStreamAnalyticsJob ` 
  -ResourceGroupName $resourceGroup ` 
  -File $jobDefinitionFile ` 
  -Name $jobName ` 
  -Force
```

Configure input to the job

Add an input to your job by using the `New-AzStreamAnalyticsInput` cmdlet. This cmdlet takes the job name, job input name, resource group name, and the job input definition as parameters. The job input definition is a JSON file that contains the properties required to configure the job's input. In this example, you'll create a blob storage as an input.

On your local machine, create a file named `JobInputDefinition.json` and add the following JSON data to it. Make sure to replace the value for `accesspolicykey` with the `sharedAccessKey` portion of the IoT Hub connection string you saved in a previous section.

```
{
  "properties": {
    "type": "Stream",
    "datasource": {
      "type": "Microsoft.Devices/IotHubs",
      "properties": {
        "iotHubNamespace": "MyASAIoTHub",
        "sharedAccessPolicyName": "iothubowner",
        "sharedAccessPolicyKey": "accesspolicykey",
        "endpoint": "messages/events",
        "consumerGroupName": "$Default"
      }
    },
    "compression": {
      "type": "None"
    },
    "serialization": {
      "type": "Json",
      "properties": {
        "encoding": "UTF8"
      }
    }
  },
  "name": "IoTHubInput",
  "type": "Microsoft.StreamAnalytics/streamingjobs/inputs"
}
```

Next, run the `New-AzStreamAnalyticsInput` cmdlet, make sure to replace the value of `jobDefinitionFile` variable with the path where you've stored the job input definition JSON file.

```
$jobInputName = "IoTHubInput"
$jobInputDefinitionFile = "C:\JobInputDefinition.json"
New-AzStreamAnalyticsInput ` 
    -ResourceGroupName $resourceGroup ` 
    -JobName $jobName ` 
    -File $jobInputDefinitionFile ` 
    -Name $jobInputName
```

Configure output to the job

Add an output to your job by using the [New-AzStreamAnalyticsOutput](#) cmdlet. This cmdlet takes the job name, job output name, resource group name, and the job output definition as parameters. The job output definition is a JSON file that contains the properties required to configure job's output. This example uses blob storage as output.

On your local machine, create a file named `JobOutputDefinition.json`, and add the following JSON data to it.

Make sure to replace the value for `accountKey` with your storage account's access key that is the value stored in `$storageAccountKey` value.

```
{
  "properties": {
    "datasource": {
      "type": "Microsoft.Storage/Blob",
      "properties": {
        "storageAccounts": [
          {
            "accountName": "asaquickstartstorage",
            "accountKey": "<storage account key>"
          }
        ],
        "container": "container1",
        "pathPattern": "output/",
        "dateFormat": "yyyy/MM/dd",
        "timeFormat": "HH"
      }
    },
    "serialization": {
      "type": "Json",
      "properties": {
        "encoding": "UTF8",
        "format": "LineSeparated"
      }
    }
  },
  "name": "BlobOutput",
  "type": "Microsoft.StreamAnalytics/streamingjobs/outputs"
}
```

Next, run the `New-AzStreamAnalyticsOutput` cmdlet. Make sure to replace the value of `jobOutputDefinitionFile` variable with the path where you have stored the job output definition JSON file.

```
$jobOutputName = "BlobOutput"
$jobOutputDefinitionFile = "C:\JobOutputDefinition.json"
New-AzStreamAnalyticsOutput ` 
    -ResourceGroupName $resourceGroup ` 
    -JobName $jobName ` 
    -File $jobOutputDefinitionFile ` 
    -Name $jobOutputName -Force
```

Define the transformation query

Add a transformation your job by using the [New-AzStreamAnalyticsTransformation](#) cmdlet. This cmdlet takes the job name, job transformation name, resource group name, and the job transformation definition as parameters.

On your local machine, create a file named `JobTransformationDefinition.json` and add the following JSON data to it. The JSON file contains a query parameter that defines the transformation query:

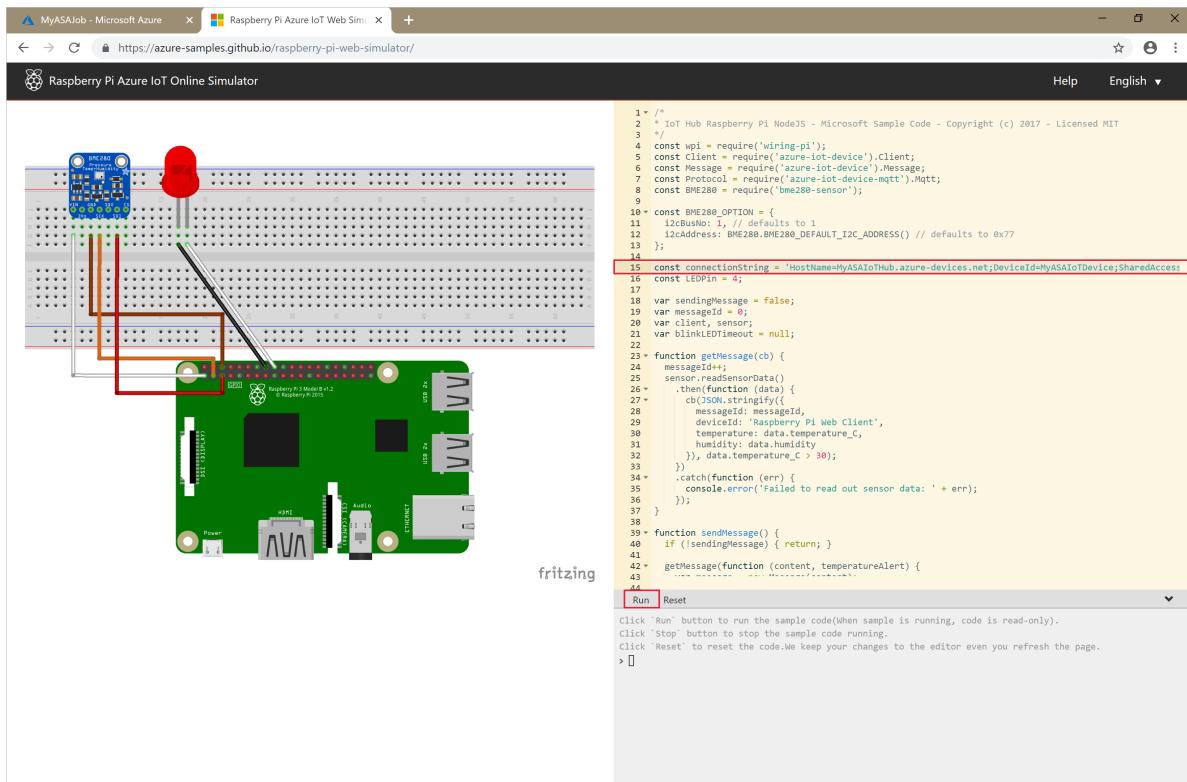
```
{
    "name": "MyTransformation",
    "type": "Microsoft.StreamAnalytics/streamingjobs/transformations",
    "properties": {
        "streamingUnits": 1,
        "script": null,
        "query": " SELECT * INTO BlobOutput FROM IoTHubInput HAVING Temperature > 27"
    }
}
```

Next run the `New-AzStreamAnalyticsTransformation` cmdlet. Make sure to replace the value of `jobTransformationDefinitionFile` variable with the path where you've stored the job transformation definition JSON file.

```
$jobTransformationName = "MyJobTransformation"
$jobTransformationDefinitionFile = "C:\JobTransformationDefinition.json"
New-AzStreamAnalyticsTransformation ` 
    -ResourceGroupName $resourceGroup ` 
    -JobName $jobName ` 
    -File $jobTransformationDefinitionFile ` 
    -Name $jobTransformationName -Force
```

Run the IoT simulator

1. Open the [Raspberry Pi Azure IoT Online Simulator](#).
2. Replace the placeholder in Line 15 with the entire Azure IoT Hub Device connection string you saved in a previous section.
3. Click **Run**. The output should show the sensor data and messages that are being sent to your IoT Hub.



Start the Stream Analytics job and check the output

Start the job by using the [Start-AzStreamAnalyticsJob](#) cmdlet. This cmdlet takes the job name, resource group name, output start mode, and start time as parameters. `OutputStartMode` accepts values of `JobStartTime`, `CustomTime`, or `LastOutputEventTime`. To learn more about what each of these values are referring to, see the [parameters](#) section in PowerShell documentation.

After you run the following cmdlet, it returns `True` as output if the job starts. In the storage container, an output folder is created with the transformed data.

```
Start-AzStreamAnalyticsJob ` 
-ResourceGroupName $resourceGroup ` 
-Name $jobName ` 
-OutputStartMode 'JobStartTime'
```

Clean up resources

When no longer needed, delete the resource group, the streaming job, and all related resources. Deleting the job avoids billing the streaming units consumed by the job. If you're planning to use the job in future, you can skip deleting it, and stop the job for now. If you aren't going to continue to use this job, delete all resources created by this quickstart by running the following cmdlet:

```
Remove-AzResourceGroup ` 
-Name $resourceGroup
```

Next steps

In this quickstart, you deployed a simple Stream Analytics job using PowerShell. You can also deploy Stream Analytics jobs using the [Azure portal](#) and [Visual Studio](#).

To learn about configuring other input sources and performing real-time detection, continue to the following article:

Quickstart: Create a Stream Analytics job by using the Azure Stream Analytics tools for Visual Studio

6/13/2019 • 6 minutes to read • [Edit Online](#)

This quickstart shows you how to create and run a Stream Analytics job using Azure Stream Analytics tools for Visual Studio. The example job reads streaming data from an IoT Hub device. You define a job that calculates the average temperature when over 27° and writes the resulting output events to a new file in blob storage.

Before you begin

- If you don't have an Azure subscription, create a [free account](#).
- Sign in to the [Azure portal](#).
- Install Visual Studio 2019, Visual Studio 2015, or Visual Studio 2013 Update 4. Enterprise (Ultimate/Premium), Professional, and Community editions are supported. Express edition is not supported.
- Follow the [installation instructions](#) to install Stream Analytics tools for Visual Studio.

Prepare the input data

Before defining the Stream Analytics job, you should prepare the data, which is later configured as the job input. To prepare the input data required by the job, complete the following steps:

1. Sign in to the [Azure portal](#).
2. Select **Create a resource > Internet of Things > IoT Hub**.
3. In the **IoT Hub** pane, enter the following information:

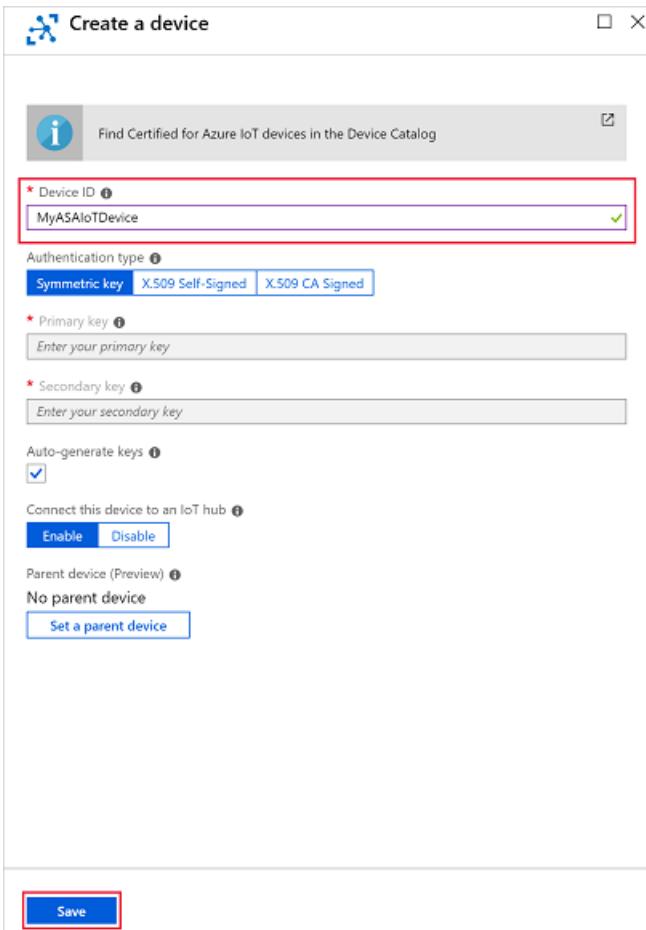
SETTING	SUGGESTED VALUE	DESCRIPTION
Subscription	<Your subscription>	Select the Azure subscription that you want to use.
Resource group	asaquickstart-resourcegroup	Select Create New and enter a new resource-group name for your account.
Region	<Select the region that is closest to your users>	Select a geographic location where you can host your IoT Hub. Use the location that's closest to your users.
IoT Hub Name	MyASAIoTHub	Select a name for your IoT Hub.

4. Select **Next: Set size and scale**.

5. Choose your **Pricing and scale tier**. For this quickstart, select the **F1 - Free** tier if it's still available on your subscription. If the free tier is unavailable, choose the lowest tier available. For more information, see [IoT Hub pricing](#).

6. Select **Review + create**. Review your IoT Hub information and click **Create**. Your IoT Hub might take a few minutes to create. You can monitor the progress in the **Notifications** pane.

7. In your IoT Hub navigation menu, click **Add** under **IoT devices**. Add a **Device ID** and click **Save**.



8. Once the device is created, open the device from the **IoT devices** list. Copy the **Connection string -- primary key** and save it to a notepad to use later.

Create blob storage

1. From the upper left-hand corner of the Azure portal, select **Create a resource > Storage > Storage account**.
2. In the **Create storage account** pane, enter a storage account name, location, and resource group.

Choose the same location and resource group as the IoT Hub you created. Then click **Review + create** to create the account.

The screenshot shows the Azure Marketplace on the left and the 'Create storage account' wizard on the right. In the Marketplace, the 'Storage' category is selected. In the wizard, the 'Subscription' dropdown is set to 'myasauickstartstorage' and the 'Resource group' dropdown is set to '(preview)'. The 'Storage account name' is 'myasauickstartstorage' and the 'Location' is 'West US'. The 'Performance' section shows 'Standard' selected. The 'Account kind' is 'StorageV2 (general purpose v2)', 'Replication' is 'Locally-redundant storage (LRS)', and 'Access tier (default)' is 'Hot'. The 'Review + create' button is highlighted at the bottom.

3. Once your storage account is created, select the **Blobs** tile on the **Overview** pane.

The screenshot shows the 'myasauickstartstorage' storage account overview. The left sidebar includes 'Overview' (highlighted with a red box), 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Events', 'Storage Explorer (preview)', 'Settings' (with 'Access keys', 'CORS', 'Configuration', 'Encryption', 'Shared access signature', 'Firewalls and virtual networks', 'Advanced Threat Protection ...', 'Static website (preview)', 'Properties', 'Locks', and 'Automation script'), 'Blob service' (with 'Blobs' and 'Custom domain'), and 'Container service' (with 'Containers' and 'File shares'). The main pane displays account details: Resource group '(change)', Status 'Primary: Available', Location 'West US', Subscription '(change)', and Account kind 'StorageV2 (general purpose v2)'. It also shows services: 'Blobs' (REST-based object storage for unstructured data), 'Files' (file shares that use the standard SMB 3.0 protocol), 'Tables' (tabular data storage), and 'Queues' (effectively scale apps according to traffic). Monitoring and network traffic data are also shown.

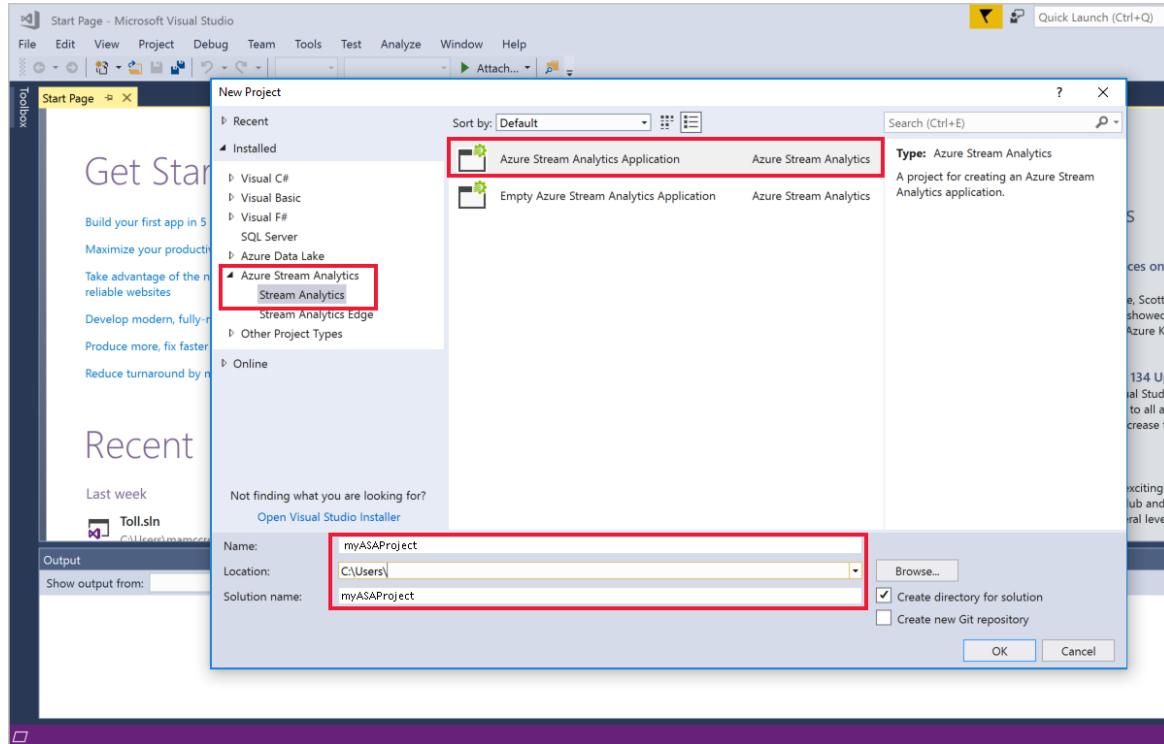
4. From the **Blob Service** page, select **Container** and provide a name for your container, such as *container1*. Leave the **Public access level** as **Private (no anonymous access)** and select **OK**.

The screenshot shows the 'New container' dialog. It has fields for 'Name' (set to 'container1') and 'Public access level' (set to 'Private (no anonymous access)'). At the bottom are 'OK' and 'Cancel' buttons, with 'OK' highlighted with a red box.

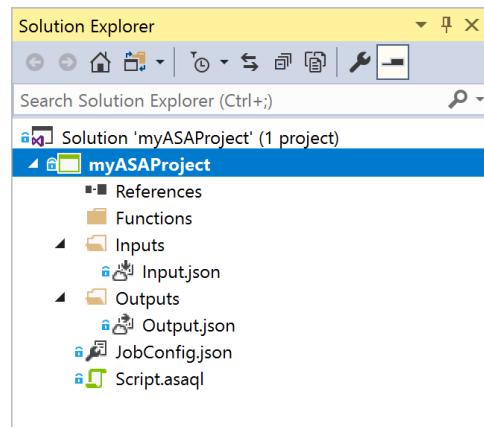
Create a Stream Analytics project

1. Start Visual Studio.

2. Select **File > New Project**.
3. In the templates list on the left, select **Stream Analytics**, and then select **Azure Stream Analytics Application**.
4. Input the project **Name**, **Location**, and **Solution name**, and select **OK**.



Notice the elements that are included in an Azure Stream Analytics project.



Choose the required subscription

1. In Visual Studio, on the **View** menu, select **Server Explorer**.
2. Right click on **Azure**, select **Connect to Microsoft Azure Subscription**, and then sign in with your Azure account.

Define input

1. In **Solution Explorer**, expand the **Inputs** node and double-click **Input.json**.
2. Fill out the **Stream Analytics Input Configuration** with the following values:

SETTING	SUGGESTED VALUE	DESCRIPTION
Input Alias	Input	Enter a name to identify the job's input.
Source Type	Data Stream	Choose the appropriate input source: Data Stream or Reference Data.
Source	IoT Hub	Choose the appropriate input source.
Resource	Choose data source from current account	Choose to enter data manually or select an existing account.
Subscription	<Your subscription>	Select the Azure subscription that has the IoT Hub you created.
IoT Hub	MyASAIoTHub	Choose or enter the name of your IoT Hub. IoT Hub names are automatically detected if they are created in the same subscription.

3. Leave other options to default values and select **Save** to save the settings.



Stream Analytics Input Configuration

Configure the Input properties for your Stream Analytics job.

* Input Alias	<input type="text" value="Input"/>
* Source Type	<input type="text" value="Data Stream"/>
* Source	<input type="text" value="IoT Hub"/>
* Resource	<input type="text" value="Choose data source from current account"/>
* Subscription	<input type="text"/>
* IoT Hub	<input type="text" value="MyASAIoTHub"/>
* Endpoint	<input type="text" value="Messaging"/>
* Shared Access Policy Name	<input type="text" value="iothubowner"/>
* Shared Access Policy Key	<input type="text" value="....."/>
* Consumer Group	<input type="text" value="\$Default"/>
* Event Serialization Format	<input type="text" value="Json"/>
Encoding	<input type="text" value="UTF8"/>
* Event Compression Type	<input type="text" value="None"/>
<input type="button" value="Save"/> <input type="button" value="Cancel"/>	

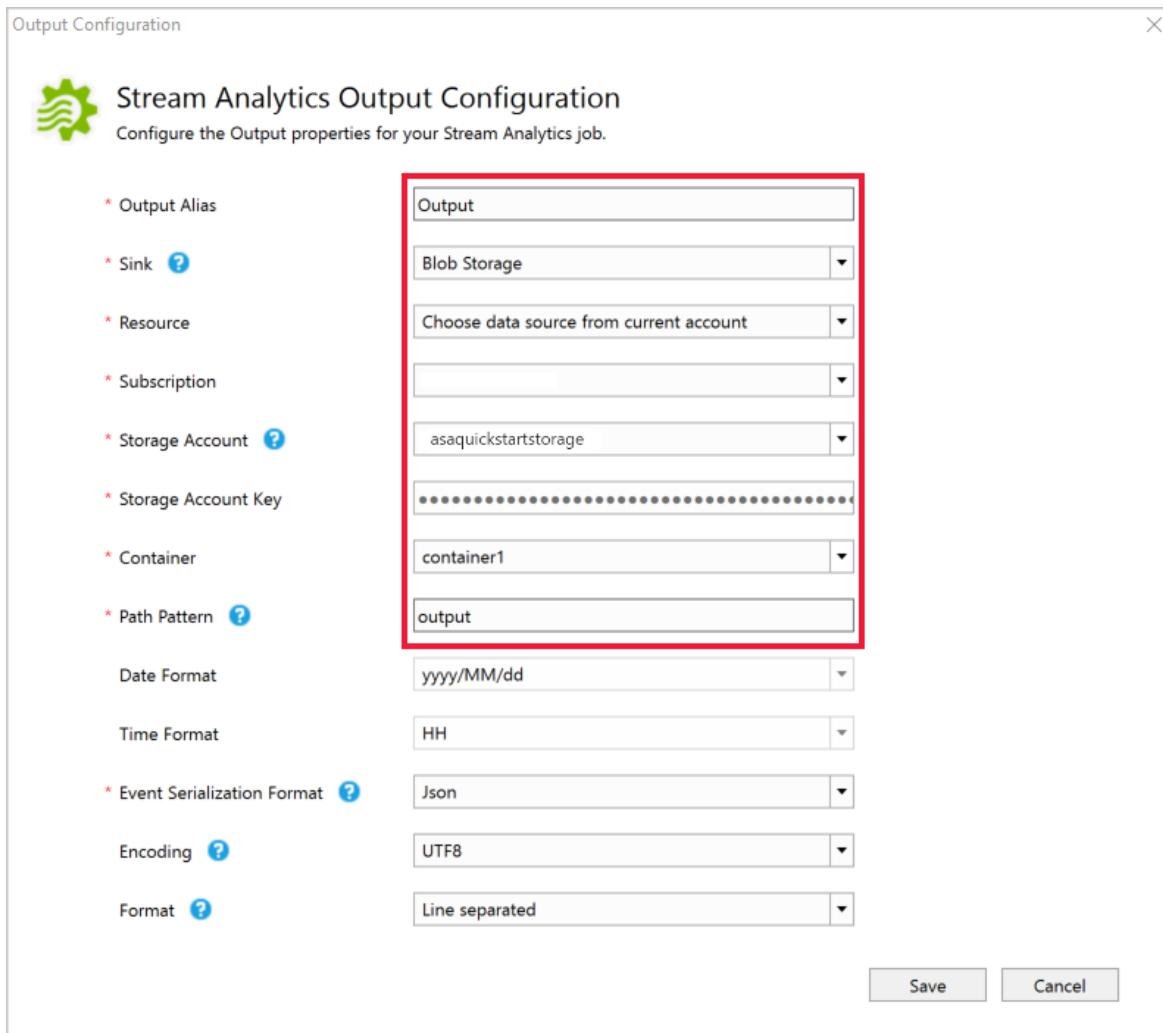
Define output

1. In **Solution Explorer**, expand the **Outputs** node and double-click **Output.json**.

2. Fill out the **Stream Analytics Output Configuration** with the following values:

SETTING	SUGGESTED VALUE	DESCRIPTION
Output Alias	Output	Enter a name to identify the job's output.
Sink	Blob Storage	Choose the appropriate sink.
Resource	Provide data source settings manually	Choose to enter data manually or select an existing account.
Subscription	<Your subscription>	Select the Azure subscription that has the storage account you created. The storage account can be in the same or in a different subscription. This example assumes that you have created storage account in the same subscription.
Storage Account	asaquickstartstorage	Choose or enter the name of the storage account. Storage account names are automatically detected if they are created in the same subscription.
Container	container1	Select the existing container that you created in your storage account.
Path Pattern	output	Enter the name of a file path to be created within the container.

3. Leave other options to default values and select **Save** to save the settings.



Define the transformation query

1. Open **Script.asaql** from **Solution Explorer** in Visual Studio.
2. Add the following query:

```
SELECT *
INTO BlobOutput
FROM IoTHubInput
HAVING Temperature > 27
```

Submit a Stream Analytics query to Azure

1. In the **Query Editor**, select **Submit To Azure** in the script editor.
2. Select **Create a New Azure Stream Analytics job** and enter a **Job Name**. Choose the **Subscription**, **Resource Group**, and **Location** you used at the beginning of the Quickstart.

Submit Job

Update existing Azure Stream Analytics job

Create a new Azure Stream Analytics job

* Job Name

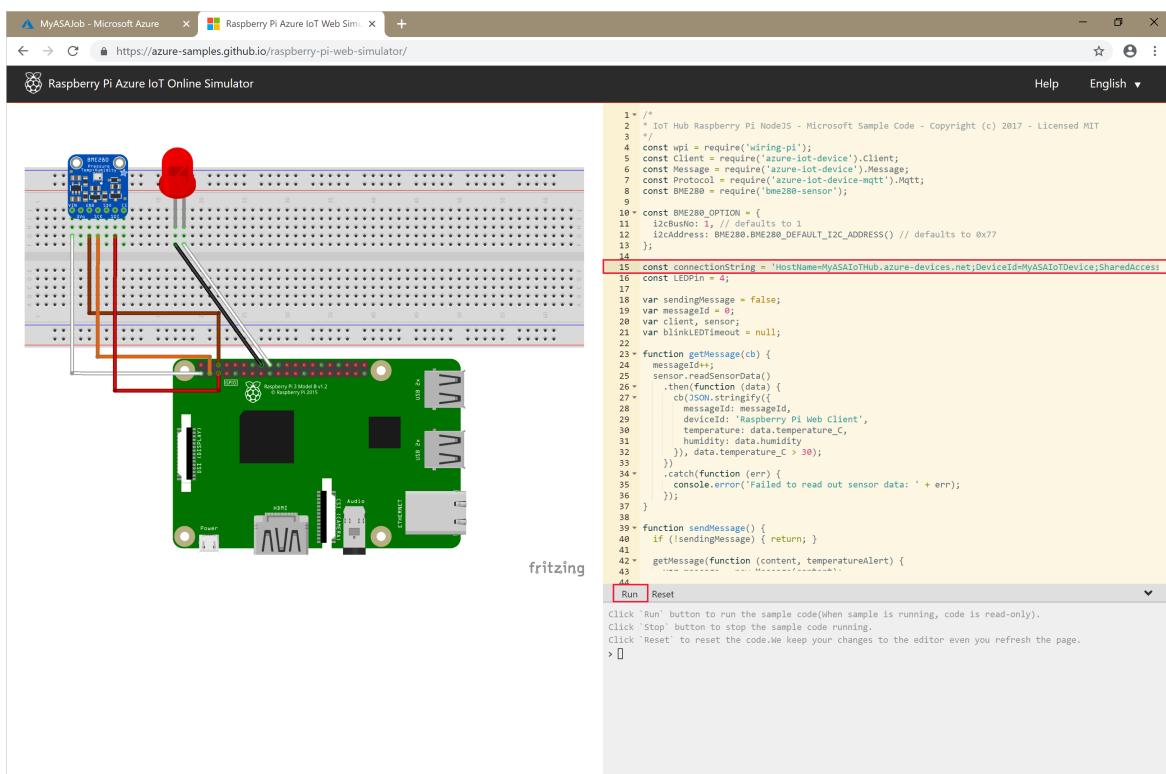
* Subscription

* Resource Group

* Location

Run the IoT simulator

1. Open the [Raspberry Pi Azure IoT Online Simulator](#) in a new browser tab or window.
2. Replace the placeholder in Line 15 with the Azure IoT Hub device connection string you saved in a previous section.
3. Click **Run**. The output should show the sensor data and messages that are being sent to your IoT Hub.



Start the Stream Analytics job and check output

1. When your job is created, the job view opens automatically. Select the green arrow button to start the job,

Stream Analytics : myASAJob X Script.asaql

Job Summary

Status	Created
Job Type	Cloud
Creation Time	6/18/2018 2:13:45 PM
Job Output Start Mode	N/A
Job Output Start Time	N/A
Last Output Time	N/A
Data Locale	en-US
Output Error Handling	Drop
Late Arrival Tolerance	5 Second(s)
Out of Order Tolerance	0 Second(s)
Out of Order Actions	Adjust
Compatibility Level	1.0
Stream Analytics Units	1

Job Metrics (Last 30 Mins)

Total Input Events	0
--------------------	---

2. Change the **Job output start mode** to **JobStartTime** and select **Start**.

Stream Analytics Start Job Configuration

Configure the Start Job properties for your Stream Analytics job.

* SU Allocation

* Job Output Start Mode JobStartTime

3. Note the job status has changed to **Running**, and there are input/output events. This may take a few minutes.

Status Running

Job Type Cloud

Creation Time 6/18/2018 2:13:45 PM

Job Output Start Mode CustomTime

Job Output Start Time 1/1/2018 2:15:03 PM

Last Output Time 6/18/2018 2:02:30 PM

Data Locale en-US

Output Error Handling Drop

Late Arrival Tolerance 5 Second(s)

Out of Order Tolerance 0 Second(s)

Out of Order Actions Adjust

Compatibility Level 1.0

Stream Analytics Units 1

Total Input Events 1.86 K

Input Event Size 164.57 KiB

Total Output Events 5

Late Input Events 1.86 K

Out of Order Events 0

Events Handled by Functions 0

- To view results, on the **View** menu, select **Cloud Explorer**, and navigate to the storage account in your resource group. Under **Blob Containers**, double-click **container1**, and then the **output** file path.

Name	Size	Last Modified (UTC)	Content Type	URL
0_33df09b67e3141618f2192d52e51aee9_1.json	527 bytes	6/18/2018 6:18:12 PM	application/octet-stream	https://.blob....

Clean up resources

When no longer needed, delete the resource group, the streaming job, and all related resources. Deleting the job avoids billing the streaming units consumed by the job. If you're planning to use the job in future, you can stop it and restart it later when you need. If you are not going to continue to use this job, delete all resources created by this quickstart by using the following steps:

- From the left-hand menu in the Azure portal, select **Resource groups** and then select the name of the resource you created.
- On your resource group page, select **Delete**, type the name of the resource to delete in the text box, and then select **Delete**.

Next steps

In this quickstart, you deployed a simple Stream Analytics job using Visual Studio. You can also deploy Stream Analytics jobs using the [Azure portal](#) and [PowerShell](#).

To learn about Azure Stream Analytics tools for Visual Studio, continue to the following article:

[Use Visual Studio to view Azure Stream Analytics jobs](#)

Quickstart: Create an Azure Stream Analytics cloud job in Visual Studio Code (Preview)

5/16/2019 • 7 minutes to read • [Edit Online](#)

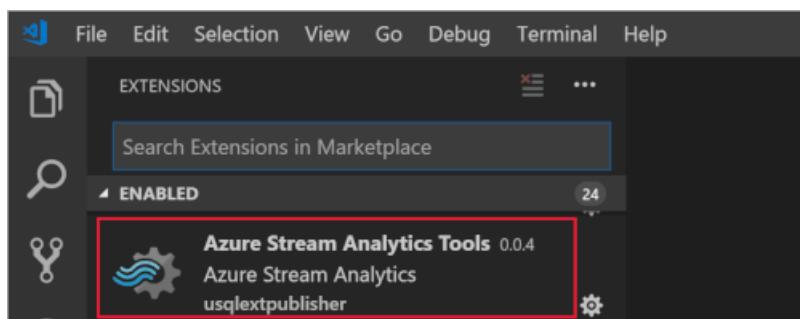
This quickstart shows you how to create and run a Stream Analytics job using the Azure Stream Analytics extension for Visual Studio Code. The example job reads streaming data from an IoT Hub device. You define a job that calculates the average temperature when over 27° and writes the resulting output events to a new file in blob storage.

Before you begin

- If you don't have an Azure subscription, create a [free account](#).
- Sign in to the [Azure portal](#).
- Install [Visual Studio Code](#).

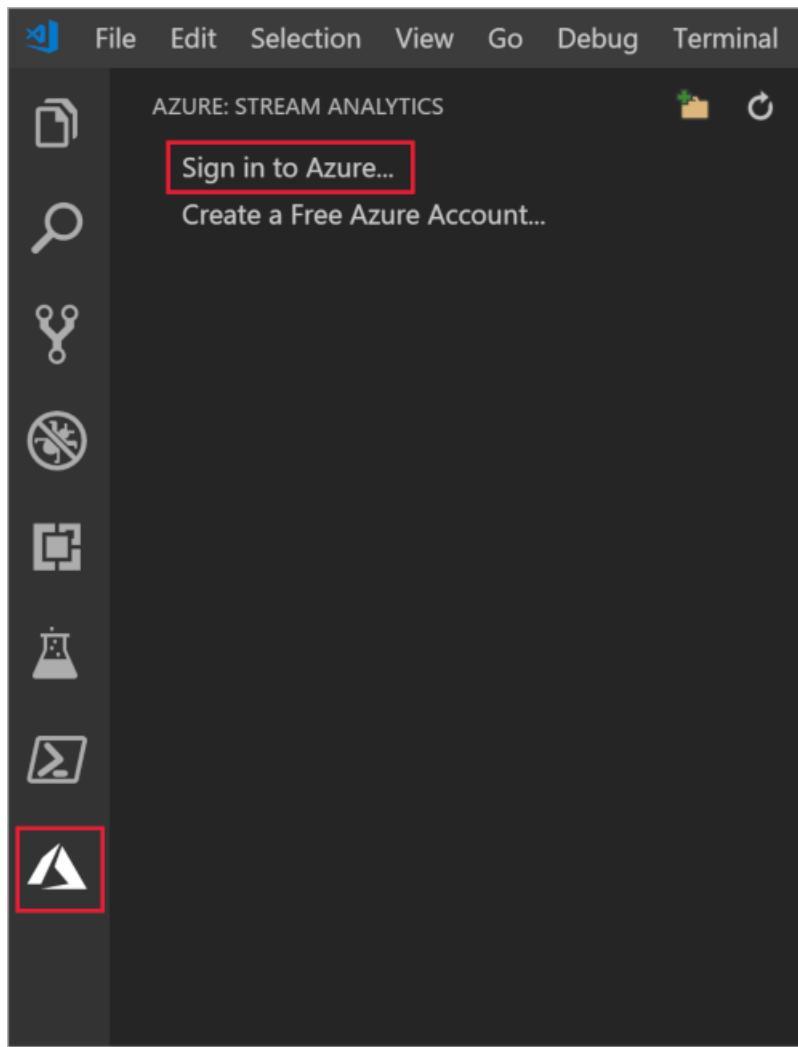
Install the Azure Stream Analytics extension

1. Open Visual Studio Code.
2. From **Extensions** on the left pane, search for **Stream Analytics** and select **Install** on the **Azure Stream Analytics** extension.
3. Once the extension is installed, verify that **Azure Stream Analytics Tools** is visible in your **Enabled Extensions**.



Activate the Azure Stream Analytics extension

1. Select the **Azure** icon on the VS Code activity bar. **Stream Analytics** will be visible in the side bar. Under **Stream Analytics**, select **Sign in to Azure**.



- When you are signed in, your Azure account name appears on the status bar in the lower-left corner of the VS Code window.

NOTE

Azure Stream Analytics tools will automatically sign in the next time if you don't sign out. If your account has two-factor authentication, it is recommended that you use phone authentication rather than using a PIN. If you have issues listing resources, signing out and signing in again usually helps. To sign out, enter the command `Azure: Sign Out`.

Prepare the input data

Before defining the Stream Analytics job, you should prepare the data, which is later configured as the job input. To prepare the input data required by the job, complete the following steps:

- Sign in to the [Azure portal](#).
- Select **Create a resource > Internet of Things > IoT Hub**.
- In the **IoT Hub** pane, enter the following information:

SETTING	SUGGESTED VALUE	DESCRIPTION
Subscription	<Your subscription>	Select the Azure subscription that you want to use.

SETTING	SUGGESTED VALUE	DESCRIPTION
Resource group	asaquickstart-resourcegroup	Select Create New and enter a new resource-group name for your account.
Region	<Select the region that is closest to your users>	Select a geographic location where you can host your IoT Hub. Use the location that's closest to your users.
IoT Hub Name	MyASAIoTHub	Select a name for your IoT Hub.

The screenshot shows the Microsoft Azure portal interface for creating a new IoT hub. On the left, the 'Azure Marketplace' sidebar lists various services, with 'Internet of Things' selected and highlighted with a red box. In the main content area, the 'IoT hub' service is selected and highlighted with a red box. The 'Basics' tab is selected, showing the configuration fields: 'Subscription' (selected), 'Resource Group' (selected), 'Region' (set to 'West US'), and 'IoT Hub Name' (set to 'MyASAIoTHub'). Both the 'Region' and 'IoT Hub Name' fields are also highlighted with a red box.

4. Select **Next: Set size and scale**.
5. Choose your **Pricing and scale tier**. For this quickstart, select the **F1 - Free** tier if it's still available on your subscription. If the free tier is unavailable, choose the lowest tier available. For more information, see [IoT Hub pricing](#).

- Select **Review + create**. Review your IoT Hub information and click **Create**. Your IoT Hub might take a few minutes to create. You can monitor the progress in the **Notifications** pane.
- In your IoT Hub navigation menu, click **Add** under **IoT devices**. Add a **Device ID** and click **Save**.

- Once the device is created, open the device from the **IoT devices** list. Copy the **Connection string -- primary key** and save it to a notepad to use later.

The screenshot shows the 'Device details' page for a device named 'MyASAIoTDevice'. It displays several connection strings and their copy buttons:

- Primary key: OreiHYCK0aLnrwdw4IdyenEwHXALOGPNnSdgGs35rouo=
- Secondary key: NahlmQFZH8CJLPCj6xUgRF4xCwqtj8gMvB01ao44=
- Connection string (primary key): HostName=MyASAIoTHub.azure-devices.net;DeviceId=MyASAIoTDevice;SharedAccessKey=OreiHYCK0aLnrwdw4IdyenEwHXALOGPNnSdgGs35rouo=
- Connection string (secondary key): HostName=MyASAIoTHub.azure-devices.net;DeviceId=MyASAIoTDevice;SharedAccessKey=NahlmQFZH8CJLPCj6xUgRF4xCwqtj8gMvB01ao44=

Below these, there's a 'Connect this device to an IoT hub' section with 'Enable' and 'Disable' buttons, and a 'Parent device (Preview)' section with a 'Set a parent device' button. The 'Module identities' tab is selected, showing a table with one row: 'No module identities listed.'

Create blob storage

- From the upper left-hand corner of the Azure portal, select **Create a resource > Storage > Storage account**.
- In the **Create storage account** pane, enter a storage account name, location, and resource group. Choose the same location and resource group as the IoT Hub you created. Then click **Review + create** to create the account.

The screenshot shows the 'Create storage account' wizard in the Azure portal. The 'Storage' category is selected in the sidebar. The main form is filled with the following details:

- Basics** tab selected.
- Subscription**: myasaunderstandingstorage
- Resource group**: myasaunderstandingstorage
- Storage account name**: myasaquickstartstorage
- Location**: West US
- Performance**: Standard
- Account kind**: StorageV2 (general purpose v2)
- Replication**: Locally-redundant storage (LRS)
- Access tier (default)**: Hot

At the bottom, there are 'Review + create' and 'Next: Advanced >' buttons.

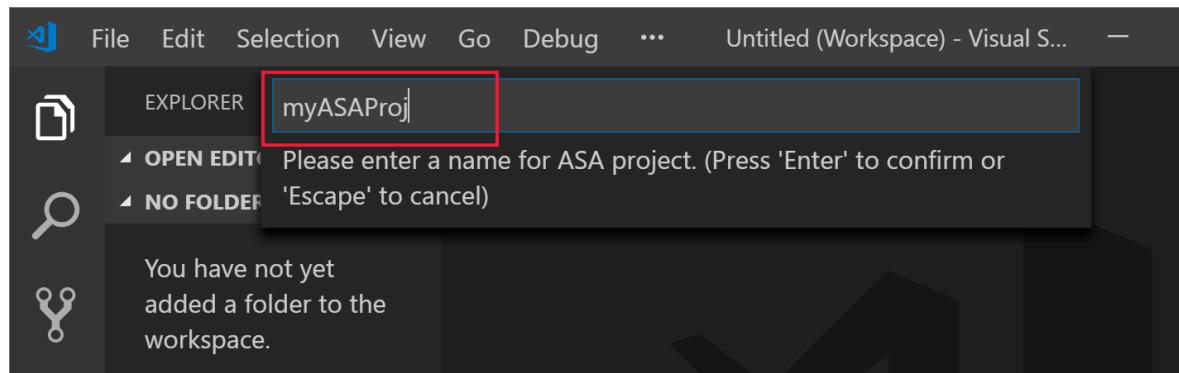
- Once your storage account is created, select the **Blobs** tile on the **Overview** pane.

- From the **Blob Service** page, select **Container** and provide a name for your container, such as **container1**. Leave the **Public access level** as **Private (no anonymous access)** and select **OK**.

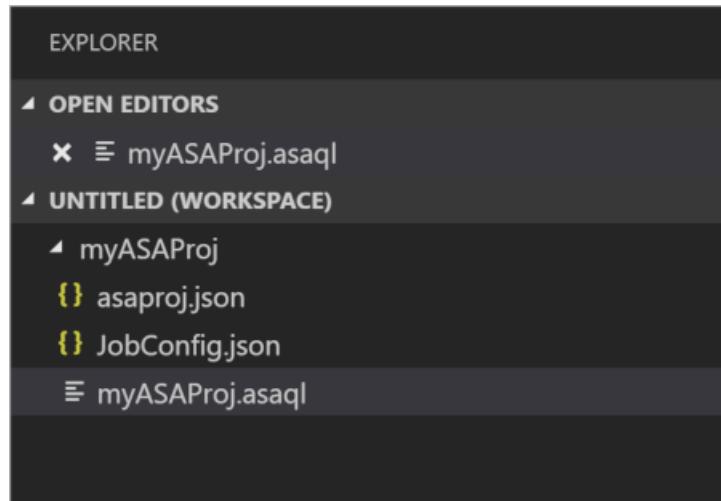
Create a Stream Analytics project

- In Visual Studio Code, press **Ctrl+Shift+P** to open the command palette. Then type **ASA** and select **ASA: Create New Project**.

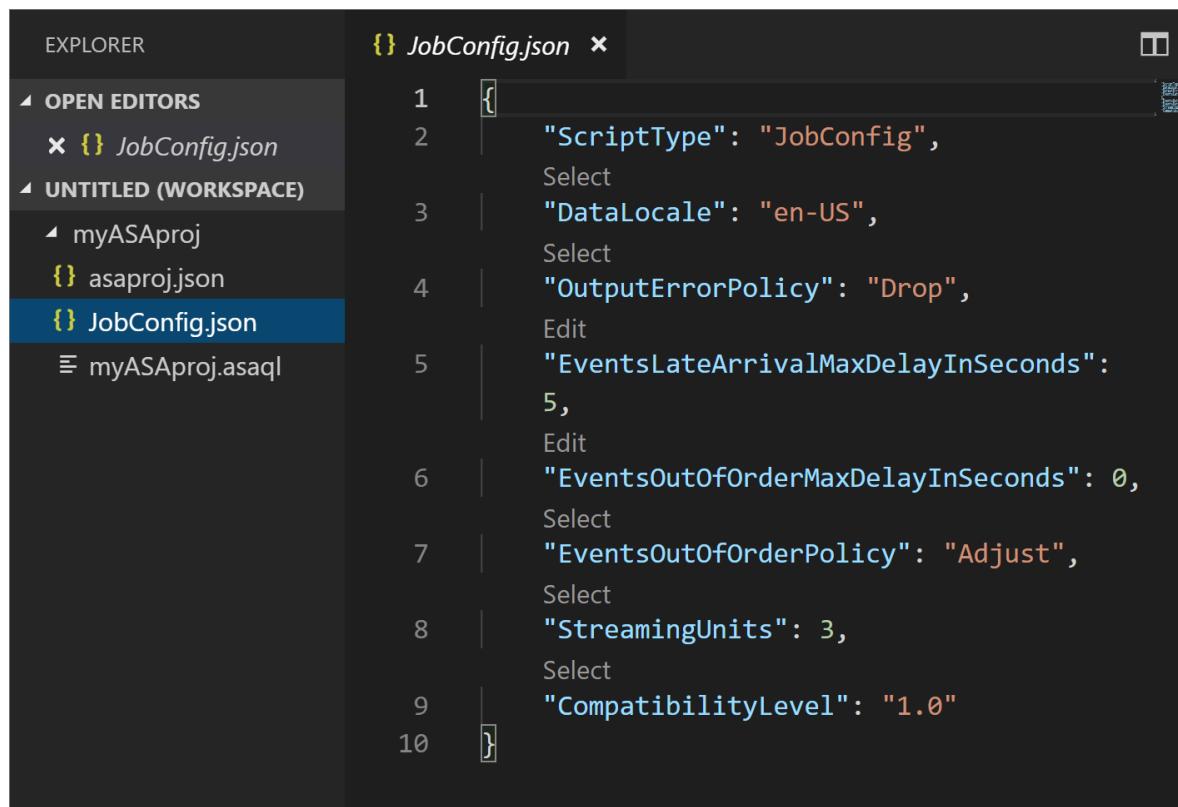
- Input your project name, like **myASAProj** and select a folder for your project.



3. The new project will be added to your workspace. An ASA project consists of the query script (***.asaql**), a **JobConfig.json** file, and an **asaproj.json** configuration file.



4. The **asaproj.json** configuration file contains the inputs, outputs, and job configuration file information needed for submitting the Stream Analytics job to Azure.

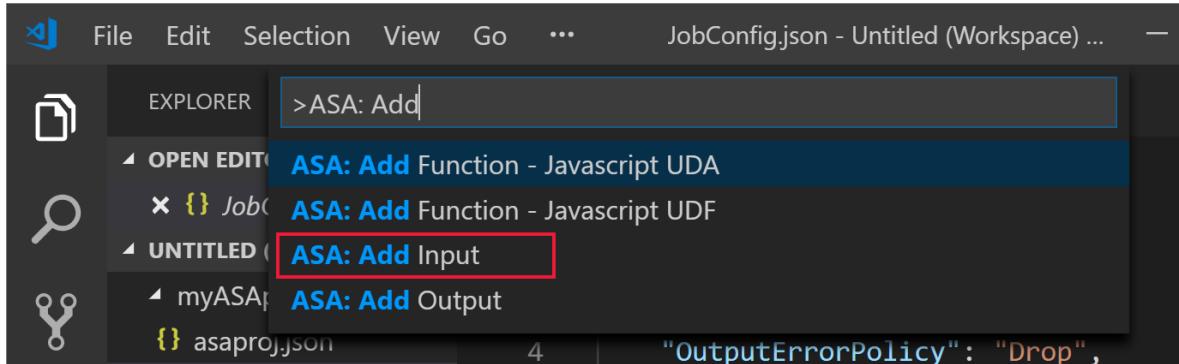


NOTE

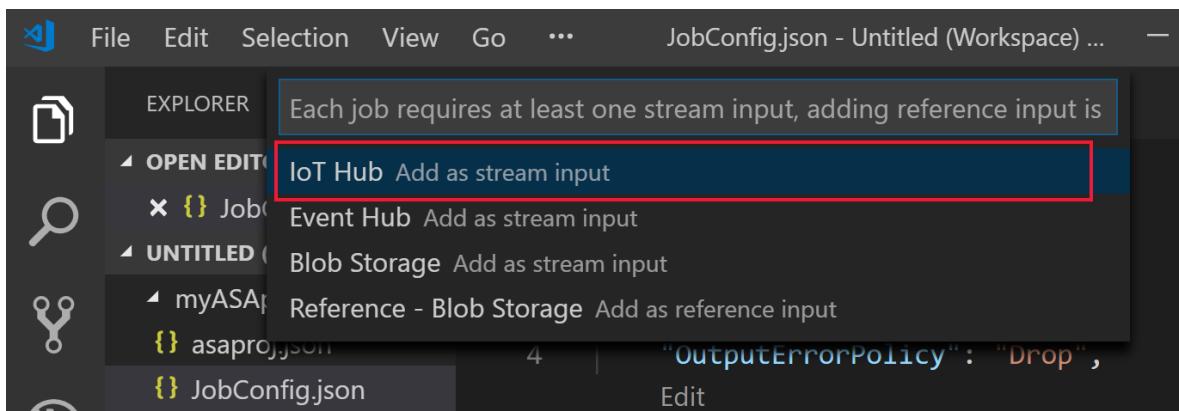
When adding inputs and outputs from the command palette, the corresponding paths will be added into **asaproj.json** automatically. If you add or remove inputs or outputs on disk directly, you need to manually add or remove them from **asaproj.json**. You can choose to put the inputs and outputs in one place then reference them in different jobs by specifying the paths in each **asaproj.json**.

Define an input

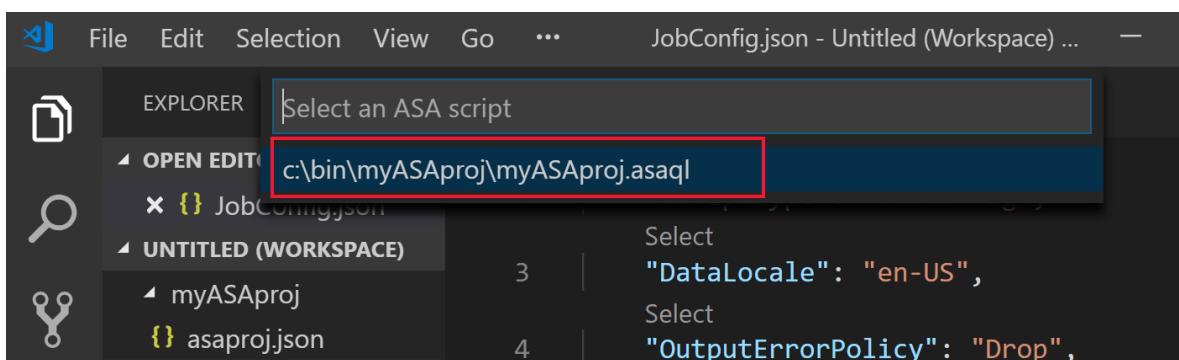
1. Select **Ctrl+Shift+P** to open the command palette and enter **ASA: Add Input**.



2. Choose **IoT Hub** for the input type.



3. Choose the ASA query script that will use the input. It should automatically populate with the file path to **myASAProj.asaql**.



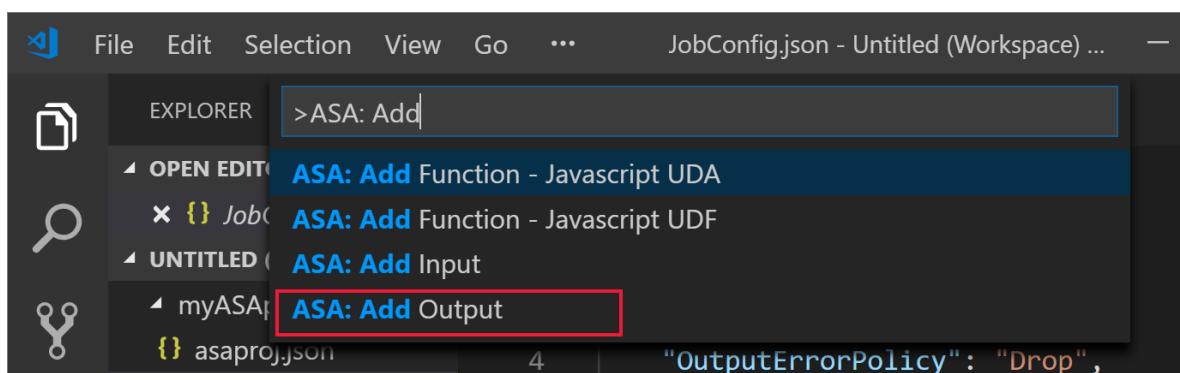
4. Enter the input file name as **iotHub.json**.

5. Edit **IoTHub.json** with the following values. Keep default values for fields not mentioned below. You can use the CodeLens to help you enter a string, select from a dropdown list, or change the text directly in the file.

SETTING	SUGGESTED VALUE	DESCRIPTION
Name	Input	Enter a name to identify the job's input.
IoTHubNamespace	MyASAIoTHub	Choose or enter the name of your IoT Hub. IoT Hub names are automatically detected if they are created in the same subscription.
EndPoint	Messaging	
SharedAccessPolicyName	iothubowner	

Define an output

1. Select **Ctrl+Shift+P** to open the command palette. Then, enter **ASA: Add Output**.



2. Choose **Blob Storage** for the Sink type.
3. Choose the ASA query script that will use this input.
4. Enter the output file name as **BlobStorage.json**.
5. Edit **BlobStorage.json** with the following values. Keep default values for fields not mentioned below. Use the CodeLens to help you enter a string or select from a dropdown list.

SETTING	SUGGESTED VALUE	DESCRIPTION
Name	Output	Enter a name to identify the job's output.
Storage Account	asaquickstartstorage	Choose or enter the name of your storage account. Storage account names are automatically detected if they are created in the same subscription.
Container	container1	Select the existing container that you created in your storage account.
Path Pattern	output	Enter the name of a file path to be created within the container.

Define the transformation query

1. Open **myASAProj.asaql** from your project folder.

2. Add the following query:

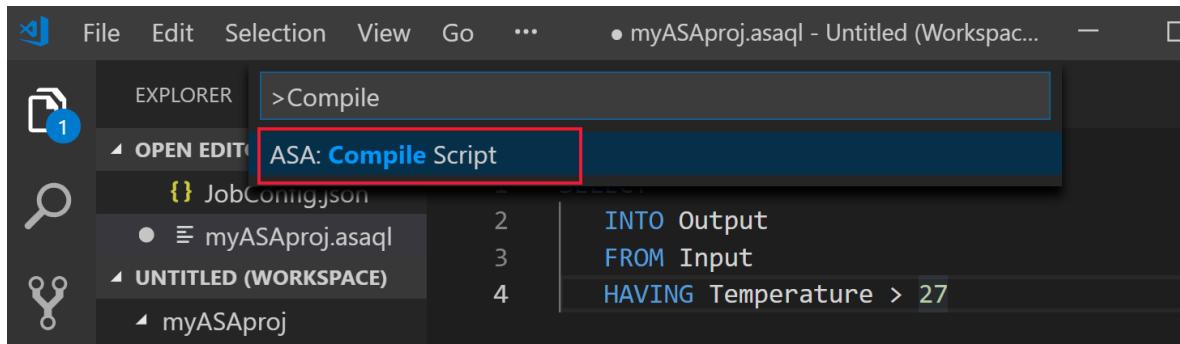
```
SELECT *
INTO Output
FROM Input
HAVING Temperature > 27
```

Compile the script

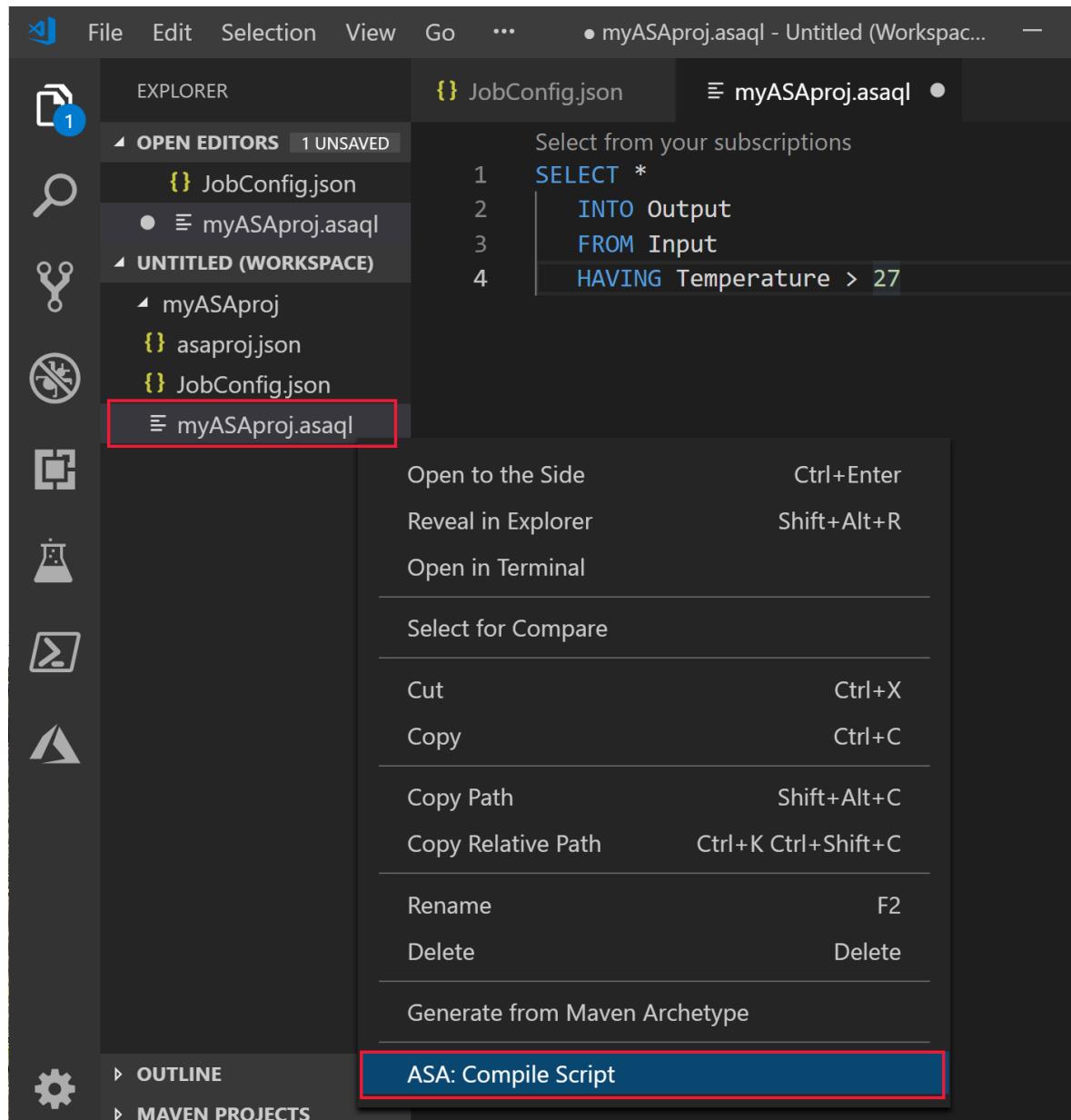
Script compilation does two things: check syntax and generate the Azure Resource Manager templates for autodeployment.

There are two ways to trigger script compilation:

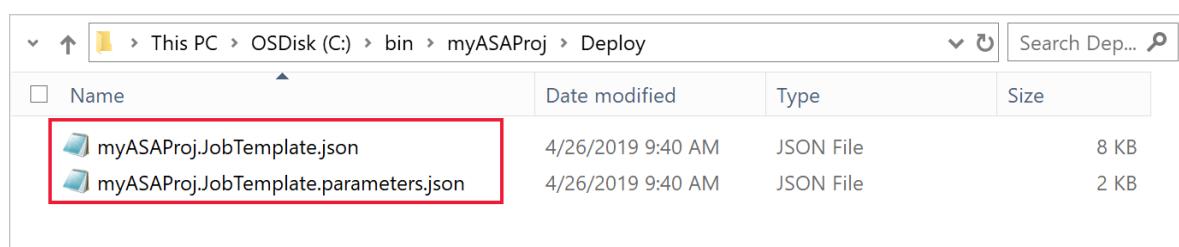
1. Select the script from workspace and then trigger compile from the command palette.



2. Right click on the script and select **ASA: compile script**.



3. After compilation, you can find the two generated Azure Resource Manager templates in **Deploy** folder of your project. These two files are used for autodeployment.



Submit a Stream Analytics job to Azure

1. In the script editor window of Visual Studio Code, select **Select from your subscriptions**.

≡ myASAProj.asaql × {} asaproj.json

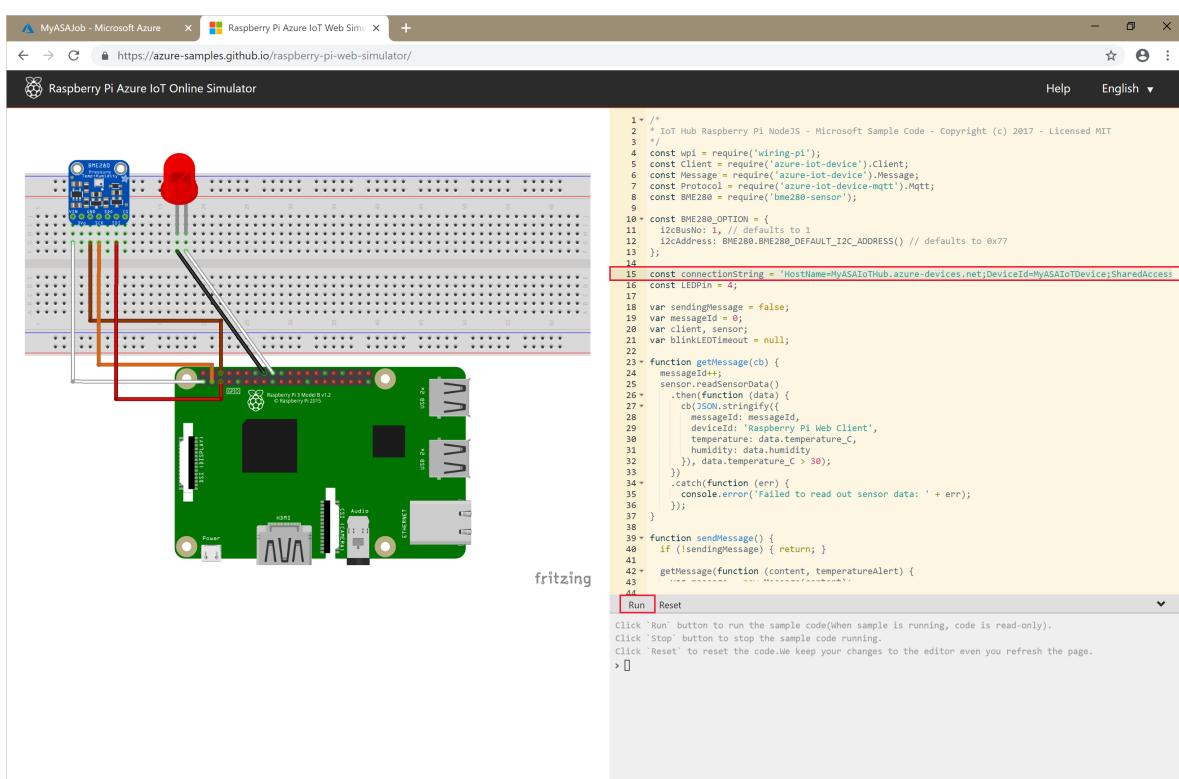
Select from your subscriptions

```
1 SELECT *
2 INTO Output
3 FROM Input
4 HAVING Temperature > 27
```

2. Select your subscription from the popup list.
3. Select a job**. Then choose Create a new job.
4. Enter your job name, **myASAjob** and then follow the instructions to choose the resource group and location.
5. Select **Submit to Azure**. The logs can be found in the output window.
6. When your job is created, you can see it in the Stream Analytics Explorer.

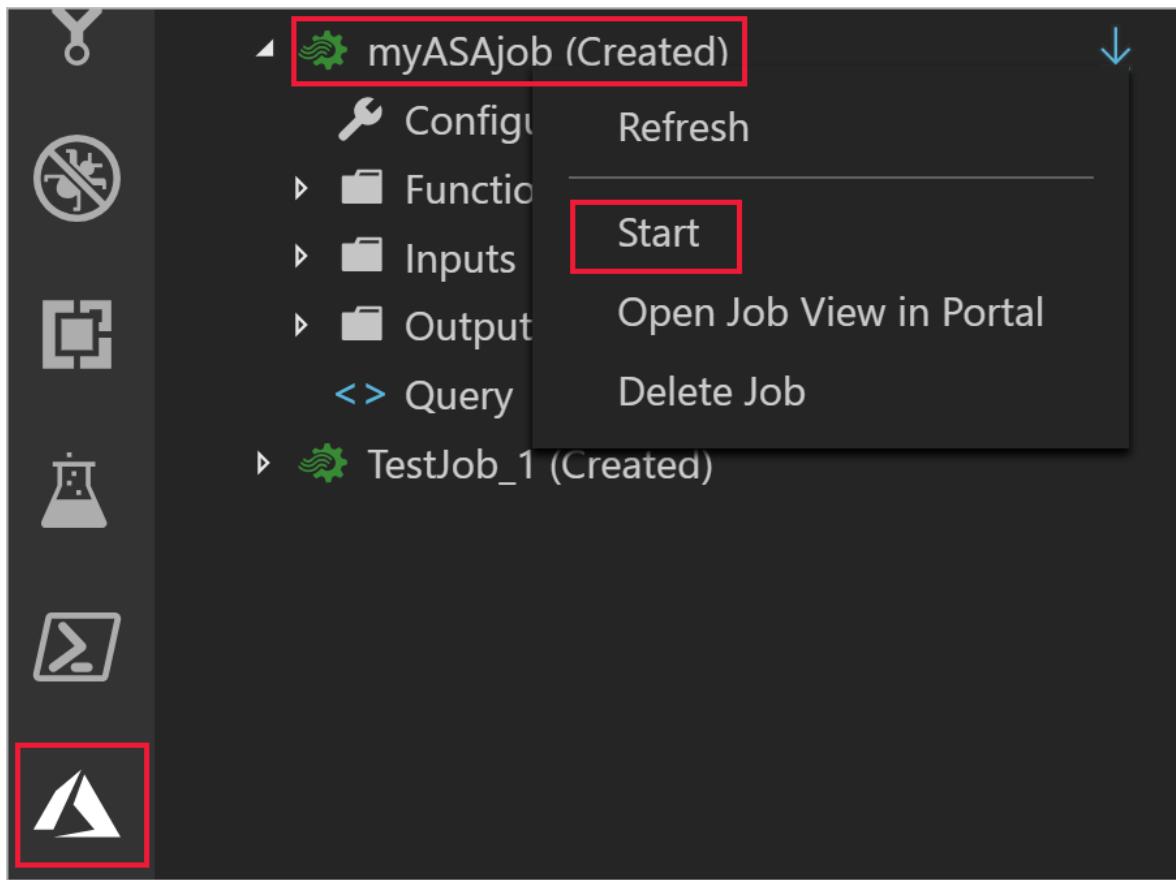
Run the IoT simulator

1. Open the [Raspberry Pi Azure IoT Online Simulator](#) in a new browser tab or window.
2. Replace the placeholder in Line 15 with the Azure IoT Hub device connection string you saved in a previous section.
3. Click **Run**. The output should show the sensor data and messages that are being sent to your IoT Hub.



Start the Stream Analytics job and check output

1. Open **Stream Analytics Explorer** in Visual Studio Code and find your job, **myASAJob**.
2. Right-click on the job name. Then, select **Start** from the context menu.



3. Choose **Now** in the popup window to start the job.
4. Note the job status has changed to **Running**. Right-click on the job name and choose **Open Job View in Portal** to see the input and output event metrics. This action may take a few minutes.
5. To view the results, open the blob storage in the Visual Studio Code extension or in the Azure portal.

Clean up resources

When no longer needed, delete the resource group, the streaming job, and all related resources. Deleting the job avoids billing the streaming units consumed by the job. If you're planning to use the job in future, you can stop it and restart it later when you need. If you are not going to continue to use this job, delete all resources created by this quickstart by using the following steps:

1. From the left-hand menu in the Azure portal, select **Resource groups** and then select the name of the resource group you created.
2. On your resource group page, select **Delete**, type the name of the resource to delete in the text box, and then select **Delete**.

Next steps

In this quickstart, you deployed a simple Stream Analytics job using Visual Studio Code. You can also deploy Stream Analytics jobs using the [Azure portal](#), [PowerShell](#), and Visual Studio (stream-analytics-quick-create-vs.md).

To learn about Azure Stream Analytics tools for Visual Studio, continue to the following article:

Use Visual Studio to view Azure Stream Analytics jobs

Analyze phone call data with Stream Analytics and visualize results in Power BI dashboard

6/4/2019 • 11 minutes to read • [Edit Online](#)

This tutorial teaches how to analyze phone call data using Azure Stream Analytics. The phone call data, generated by a client application, contains some fraudulent calls, which will be filtered by the Stream Analytics job.

In this tutorial, you learn how to:

- Generate sample phone call data and send it to Azure Event Hubs
- Create a Stream Analytics job
- Configure job input and output
- Define a query to filter fraudulent calls
- Test and start the job
- Visualize results in Power BI

Prerequisites

Before you start, do the following actions:

- If you don't have an Azure subscription, create a [free account](#).
- Sign in to the [Azure portal](#).
- Download the phone call event generator app [TelcoGenerator.zip](#) from the Microsoft Download Center or get the source code from [GitHub](#).
- You will need Power BI account.

Create an Azure Event Hub

Before Stream Analytics can analyze the fraudulent calls data stream, the data needs to be sent to Azure. In this tutorial, you will send data to Azure by using [Azure Event Hubs](#).

Use the following steps to create an Event Hub and send call data to that Event Hub:

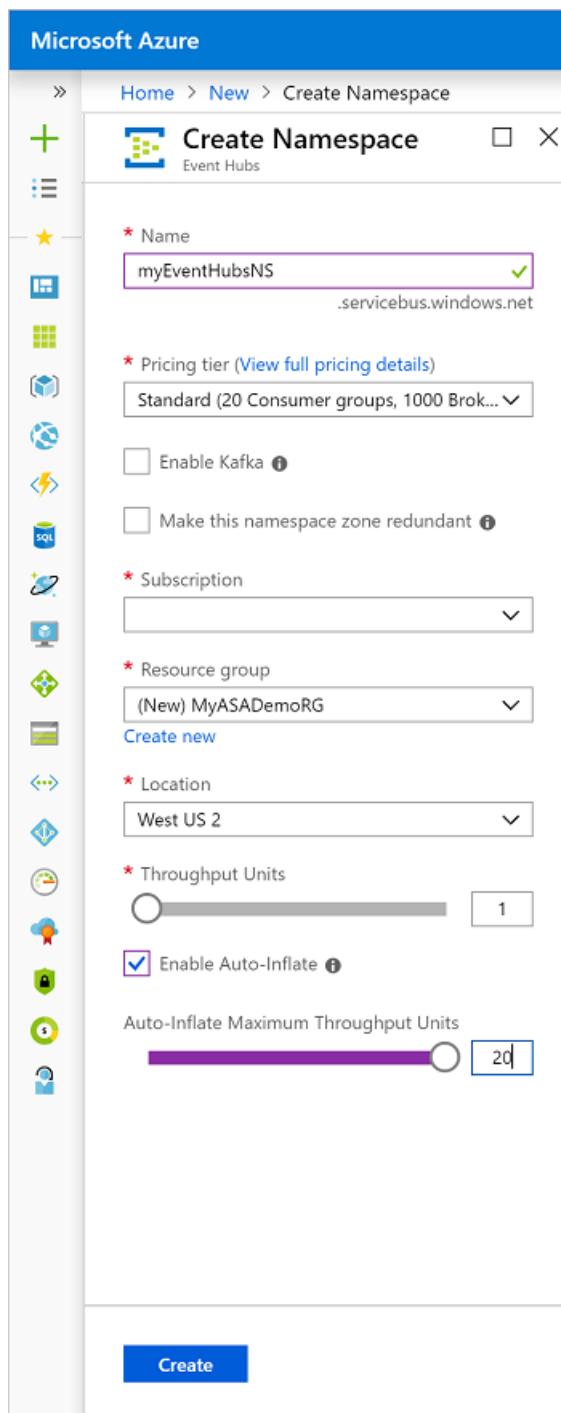
1. Sign in to the [Azure portal](#).
2. Select **Create a resource > Internet of Things > Event Hubs**.

The screenshot shows the Microsoft Azure portal's 'New' blade. On the left, there's a sidebar with various service categories like Get started, Recently created, Compute, Networking, Storage, Web, Mobile, Containers, Databases, Analytics, AI + Machine Learning, Internet of Things, Integration, Security, Identity, Developer tools, Management Tools, Software as a service (SaaS), and Blockchain. The 'Internet of Things' category is currently selected and highlighted with a red box. Within this category, the 'Event Hubs' service is also highlighted with a red box. Other services listed include Data Box Edge / Data Box Gateway (preview), IoT Hub, IoT Hub Device Provisioning Service, Time Series Insights, Stream Analytics job, Machine Learning service workspace (preview), Machine Learning Model Management (Retiring), Notification Hub, and HDInsight.

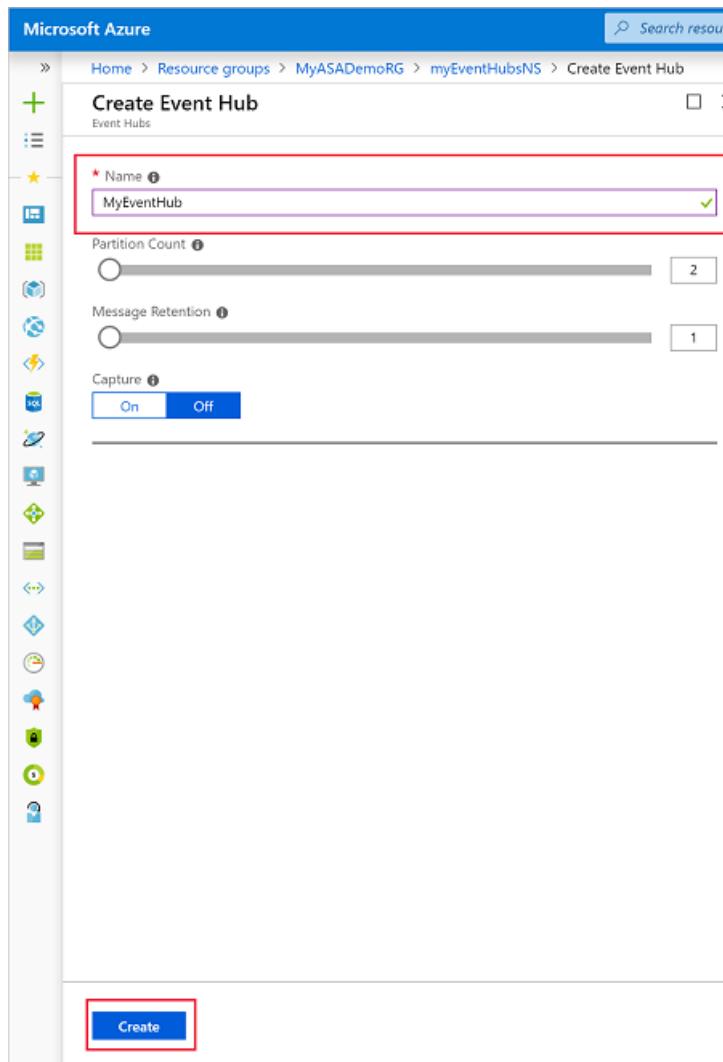
3. Fill out the **Create Namespace** pane with the following values:

SETTING	SUGGESTED VALUE	DESCRIPTION
Name	myEventHubsNS	A unique name to identify the event hub namespace.
Subscription	<Your subscription>	Select an Azure subscription where you want to create the event hub.
Resource group	MyASADemoRG	Select Create New and enter a new resource-group name for your account.
Location	West US2	Location where the event hub namespace can be deployed.

4. Use default options on the remaining settings and select **Create**.



5. When the namespace has finished deploying, go to **All resources** and find *myEventHubsNS* in the list of Azure resources. Select *myEventHubsNS* to open it.
6. Next select **+Event Hub** and enter the **Name** as *MyEventHub* or a different name of your choice. Use the default options on the remaining settings and select **Create**. Then wait for the deployment to succeed.



Grant access to the event hub and get a connection string

Before an application can send data to Azure Event Hubs, the event hub must have a policy that allows appropriate access. The access policy produces a connection string that includes authorization information.

1. Navigate to the event hub you created in the previous step, MyEventHub*. Select **Shared access policies** under **Settings**, and then select **+ Add**.
2. Name the policy **MyPolicy** and ensure **Manage** is checked. Then select **Create**.

The screenshot shows the 'Shared access policies' blade for the 'myeventhub' event hub. A new policy named 'MyPolicy' is being created. The 'POLICY' section shows 'Manage', 'Send', and 'Listen' checkboxes checked. The 'CLAIMS' section is empty. A red box highlights the 'Policy name' input field containing 'MyPolicy'. A blue box highlights the 'Create' button at the bottom right.

- Once the policy is created, select to open the policy, and find the **Connection string-primary key**. Select the blue **copy** button next to the connection string.

The screenshot shows the 'SAS Policy: MyPolicy' blade. It lists four keys: Primary key, Secondary key, Connection string-primary key, and Connection string-secondary key. Each key has a copy icon to its right. A red box highlights the 'MyPolicy' link in the left navigation bar. A blue box highlights the 'Copy' icon next to the Connection string-primary key value.

- Paste the connection string into a text editor. You need this connection string in the next section.

The connection string looks as follows:

```
Endpoint=sb://<Your event hub namespace>.servicebus.windows.net/;SharedAccessKeyName=<Your shared access policy name>;SharedAccessKey=<generated key>;EntityPath=<Your event hub name>
```

Notice that the connection string contains multiple key-value pairs separated with semicolons: **Endpoint**, **SharedAccessKeyName**, **SharedAccessKey**, and **EntityPath**.

Start the event generator application

Before you start the TelcoGenerator app, you should configure it to send data to the Azure Event Hubs you created earlier.

1. Extract the contents of [TelcoGenerator.zip](#) file.
2. Open the `TelcoGenerator\TelcoGenerator\telcodatagen.exe.config` file in a text editor of your choice (There is more than one .config file, so be sure that you open the right one.)
3. Update the `<appSettings>` element in the config file with the following details:
 - Set the value of the `EventHubName` key to the value of the EntityPath in the connection string.
 - Set the value of the `Microsoft.ServiceBus.ConnectionString` key to the connection string without the EntityPath value (don't forget to remove the semicolon that precedes it).
4. Save the file.
5. Next open a command window and change to the folder where you unzipped the TelcoGenerator application. Then enter the following command:

```
telcodatagen.exe 1000 0.2 2
```

This command takes the following parameters:

- Number of call data records per hour.
- Percentage of fraud probability, which is how often the app should simulate a fraudulent call. The value 0.2 means that about 20% of the call records will look fraudulent.
- Duration in hours, which is the number of hours that the app should run. You can also stop the app at any time by ending the process (**Ctrl+C**) at the command line.

After a few seconds, the app starts displaying phone call records on the screen as it sends them to the event hub. The phone call data contains the following fields:

RECORD	DEFINITION
CallrecTime	The timestamp for the call start time.
SwitchNum	The telephone switch used to connect the call. For this example, the switches are strings that represent the country/region of origin (US, China, UK, Germany, or Australia).
CallingNum	The phone number of the caller.
CallingIMSI	The International Mobile Subscriber Identity (IMSI). It's a unique identifier of the caller.
CalledNum	The phone number of the call recipient.
CalledIMSI	International Mobile Subscriber Identity (IMSI). It's a unique identifier of the call recipient.

Create a Stream Analytics job

Now that you have a stream of call events, you can create a Stream Analytics job that reads data from the event hub.

1. To create a Stream Analytics job, navigate to the [Azure portal](#).
2. Select **Create a resource > Internet of Things > Stream Analytics job**.
3. Fill out the **New Stream Analytics job** pane with the following values:

SETTING	SUGGESTED VALUE	DESCRIPTION
Job name	ASATutorial	A unique name to identify the event hub namespace.
Subscription	<Your subscription>	Select an Azure subscription where you want to create the job.
Resource group	MyASADemoRG	Select Use existing and enter a new resource-group name for your account.
Location	West US2	Location where the job can be deployed. It's recommended to place the job and the event hub in the same region for best performance and so that you don't pay to transfer data between regions.
Hosting environment	Cloud	Stream Analytics jobs can be deployed to cloud or edge. Cloud allows you to deploy to Azure Cloud, and Edge allows you to deploy to an IoT Edge device.
Streaming units	1	Streaming units represent the computing resources that are required to execute a job. By default, this value is set to 1. To learn about scaling streaming units, see understanding and adjusting streaming units article.

4. Use default options on the remaining settings, select **Create**, and wait for the deployment to succeed.

The screenshot shows the Azure portal interface for creating a new Stream Analytics job. On the left, there's a sidebar with a 'New' button and a search bar. Below it is the 'Azure Marketplace' section with various categories like Get started, Recently created, Compute, Networking, Storage, Web, Mobile, Containers, Databases, Analytics, AI + Machine Learning, Internet of Things, Integration, Security, Identity, Developer tools, Management Tools, Software as a service (SaaS), and Blockchain. The 'Internet of Things' category is highlighted with a red box. Within this category, the 'Stream Analytics job' option is also highlighted with a red box. The main pane on the right is titled 'New Stream Analytics job' and contains fields for 'Job name' (ASATutorial), 'Subscription' (SQL DB Content), 'Resource group' (MyASADemoRG), 'Location' (West US 2), 'Hosting environment' (Cloud), and 'Streaming units (1 to 120)' (set to 1). At the bottom of the configuration pane are 'Create' and 'Automation options' buttons.

Configure job input

The next step is to define an input source for the job to read data using the event hub you created in the previous section.

1. From the Azure portal, open the **All resources** pane, and find the *ASATutorial* Stream Analytics job.
2. In the **Job Topology** section of the Stream Analytics job pane, select the **Inputs** option.
3. Select **+ Add stream input** and **Event hub**. Fill out the pane with the following values:

SETTING	SUGGESTED VALUE	DESCRIPTION
Input alias	CallStream	Provide a friendly name to identify your input. Input alias can contain alphanumeric characters, hyphens, and underscores only and must be 3-63 characters long.
Subscription	<Your subscription>	Select the Azure subscription where you created the event hub. The event hub can be in same or a different subscription as the Stream Analytics job.

SETTING	SUGGESTED VALUE	DESCRIPTION
Event hub namespace	myEventHubsNS	Select the event hub namespace you created in the previous section. All the event hub namespaces available in your current subscription are listed in the dropdown.
Event Hub name	MyEventHub	Select the event hub you created in the previous section. All the event hubs available in your current subscription are listed in the dropdown.
Event Hub policy name	MyPolicy	Select the event hub shared access policy you created in the previous section. All the event hubs policies available in your current subscription are listed in the dropdown.

4. Use default options on the remaining settings and select **Save**.

The screenshot shows the Azure Stream Analytics job configuration interface. On the left, the 'Inputs' section is selected. A red box highlights the 'Add stream input' button. On the right, a modal dialog titled 'Event Hub' is open, showing the configuration for a new input. The 'Input alias' field is set to 'CallStream'. The 'Event Hub name' field is set to 'myeventhub'. Other fields like 'Event Hub namespace' and 'Event Hub policy name' are also visible. The 'Save' button at the bottom of the dialog is highlighted with a red box.

Configure job output

The last step is to define an output sink for the job where it can write the transformed data. In this tutorial, you output and visualize data with Power BI.

- From the Azure portal open **All resources** pane, and the **ASATutorial** Stream Analytics job.
- In the **Job Topology** section of the Stream Analytics job pane, select the **Outputs** option.
- Select **+ Add > Power BI**. Then fill the form with the following details and select **Authorize**:

SETTING	SUGGESTED VALUE
Output alias	MyPBOutput

SETTING	SUGGESTED VALUE
Dataset name	ASAdataset
Table name	ASATable

The screenshot shows the 'ASATutorial - Outputs' section in the Azure Stream Analytics job configuration. On the left, there's a sidebar with various settings like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, and a prominent 'Outputs' section which is highlighted with a red box. In the main area, there's a 'NAME' field set to 'Empty' and a 'SINK' field labeled 'Empty'. A large red box highlights the 'Add' button. To the right, a modal window titled 'Power BI' is open, showing fields for 'Output alias' (set to 'MyPBoutput'), 'Group workspace' (set to 'Authorize connection to load workspaces'), 'Dataset name' (set to 'ASAdataset'), and 'Table name' (set to 'ASATable'). The 'Authorize' button is also highlighted with a red box. Below the modal, there's a note about granting permanent access to Power BI and three options for revoking it.

- When you select **Authorize**, a pop-up window opens and you are asked to provide credentials to authenticate to your Power BI account. Once the authorization is successful, **Save** the settings.

Define a query to analyze input data

The next step is to create a transformation that analyzes data in real time. You define the transformation query by using [Stream Analytics Query Language](#). The query used in this tutorial detects fraudulent calls from the phone data.

In this example, fraudulent calls are made from the same user within five seconds but in separate locations. For example, the same user can't legitimately make a call from the US and Australia at the same time. To define the transformation query for your Stream Analytics job:

- From the Azure portal, open the **All resources** pane and navigate to the **ASATutorial** Stream Analytics job you created earlier.
- In the **Job Topology** section of the Stream Analytics job pane, select the **Query** option. The query window lists the inputs and outputs that are configured for the job, and lets you create a query to transform the input stream.
- Replace the existing query in the editor with the following query, which performs a self-join on a 5-second interval of call data:

```

SELECT System.Timestamp AS WindowEnd, COUNT(*) AS FraudulentCalls
INTO "MyPBIoutput"
FROM "CallStream" CS1 TIMESTAMP BY CallRecTime
JOIN "CallStream" CS2 TIMESTAMP BY CallRecTime
ON CS1.CallingIMSI = CS2.CallingIMSI
AND DATEDIFF(ss, CS1, CS2) BETWEEN 1 AND 5
WHERE CS1.SwitchNum != CS2.SwitchNum
GROUP BY TumblingWindow(Duration(second, 1))

```

To check for fraudulent calls, you can self-join the streaming data based on the `CallRecTime` value. You can then look for call records where the `CallingIMSI` value (the originating number) is the same, but the `SwitchNum` value (country/region of origin) is different. When you use a JOIN operation with streaming data, the join must provide some limits on how far the matching rows can be separated in time. Because the streaming data is endless, the time bounds for the relationship are specified within the **ON** clause of the join using the **DATEDIFF** function.

This query is just like a normal SQL join except for the **DATEDIFF** function. The **DATEDIFF** function used in this query is specific to Stream Analytics, and it must appear within the `ON...BETWEEN` clause.

4. Save the query.

The screenshot shows the Azure Stream Analytics job editor interface. On the left, there's a sidebar with various tabs: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Locks, Job topology, Inputs, Functions, and Query. The 'Query' tab is highlighted with a red box. The main area shows the query editor with the following code:

```

1 SELECT System.Timestamp AS WindowEnd, COUNT(*) AS FraudulentCalls
2 INTO "MyPBIoutput"
3 FROM "CallStream" CS1 TIMESTAMP BY CallRecTime
4 JOIN "CallStream" CS2 TIMESTAMP BY CallRecTime
5 ON CS1.CallingIMSI = CS2.CallingIMSI
6 AND DATEDIFF(ss, CS1, CS2) BETWEEN 1 AND 5
7 WHERE CS1.SwitchNum != CS2.SwitchNum
8 GROUP BY TumblingWindow(Duration(second, 1))
9

```

Test your query

You can test a query from the query editor using sample data. Run the following steps to test the query:

1. Make sure that the TelcoGenerator app is running and producing phone call records.
2. In the **Query** pane, select the dots next to the *CallStream* input and then select **Sample data from input**.
3. Set **Minutes** to 3 and select **OK**. Three minutes worth of data is then sampled from the input stream and you are notified when the sample data is ready. You can view the status of sampling from the notification bar.

The sample data is stored temporarily and is available while you have the query window open. If you close the query window, the sample data is discarded, and you'll have to create a new set of sample data if you want to test. Alternatively, you can use a sample data JSON file from [GitHub](#), and then upload that JSON file to use as sample data for the *CallStream* input.

```

1 SELECT System.Timestamp AS WindowEnd, COUNT(*) AS FraudulentCalls
2 INTO "MyPBIoutput"
3 FROM "CallStream" CS1 TIMESTAMP BY CallRecTime
4 JOIN "CallStream" CS2 TIMESTAMP BY CallRecTime
5 ON CS1.CallingIMSI = CS2.CallingIMSI
6 AND DATEDIFF(ss, CS1, CS2) BETWEEN 1 AND 5
7 WHERE CS1.SwitchNum != CS2.SwitchNum
8 GROUP BY TumblingWindow(Duration(second, 1))

```

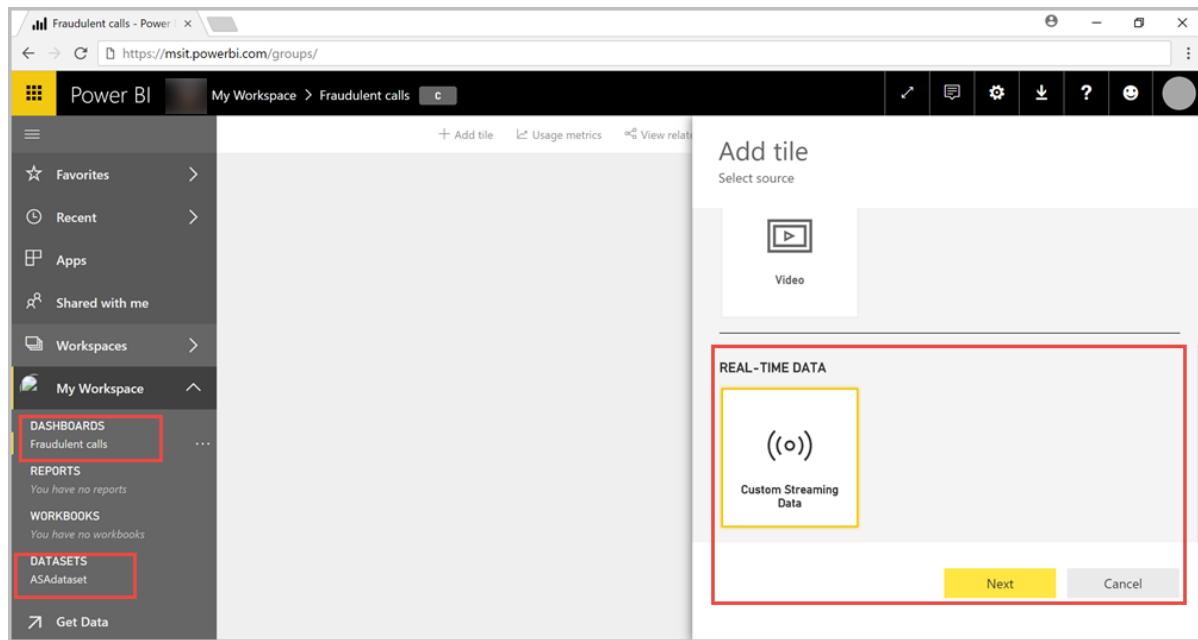
4. Select **Test** to test the query. You should see the following results:

WINDOWEND	FRAUDULENTCALLS
"2018-10-30T02:30:09.0000000Z"	1
"2018-10-30T02:30:10.0000000Z"	2
"2018-10-30T02:30:11.0000000Z"	2
"2018-10-30T02:30:13.0000000Z"	2
"2018-10-30T02:30:15.0000000Z"	2
"2018-10-30T02:30:16.0000000Z"	2
"2018-10-30T02:30:18.0000000Z"	5

Start the job and visualize output

- To start the job, navigate to the **Overview** pane of your job and select **Start**.
- Select **Now** for job output start time and select **Start**. You can view the job status in the notification bar.
- Once the job succeeds, navigate to **Power BI** and sign in with your work or school account. If the Stream Analytics job query is outputting results, the *ASAdataset* dataset you created exists under the **Datasets** tab.
- From your Power BI workspace, select + **Create** to create a new dashboard named *Fraudulent Calls*.
- At the top of the window, select **Add tile**. Then select **Custom Streaming Data** and **Next**. Choose the

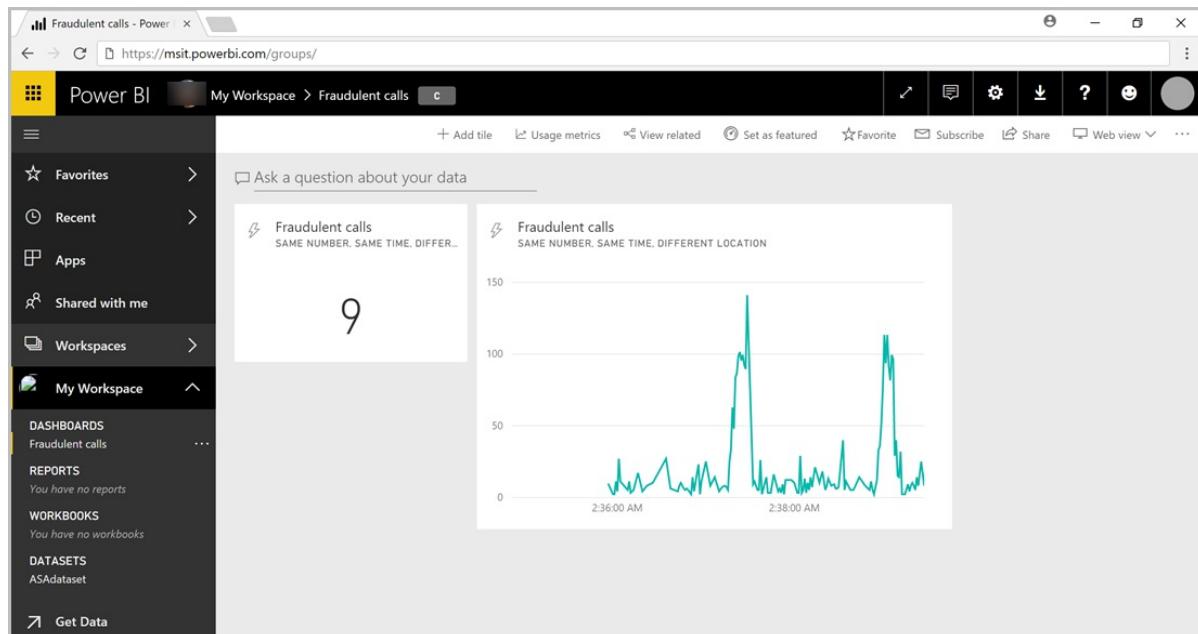
ASAdataset under **Your Datasets**. Select **Card** from the **Visualization type** dropdown, and add **fraudulent calls** to **Fields**. Select **Next** to enter a name for the tile, and then select **Apply** to create the tile.



6. Follow the step 5 again with the following options:

- When you get to Visualization Type, select Line chart.
- Add an axis and select **windowend**.
- Add a value and select **fraudulentcalls**.
- For **Time window to display**, select the last 10 minutes.

7. Your dashboard should look like the example below once both tiles are added. Notice that, if your event hub sender application and Streaming Analytics application are running, your Power BI dashboard periodically updates as new data arrives.



Embedding your Power BI Dashboard in a Web Application

For this part of the tutorial, you'll use a sample [ASP.NET](#) web application created by the Power BI team to embed your dashboard. For more information about embedding dashboards, see [embedding with Power BI](#) article.

To set up the application, go to the [PowerBI-Developer-Samples](#) GitHub repository and follow the instructions under the **User Owns Data** section (use the redirect and homepage URLs under the **integrate-dashboard-web-**

app subsection). Since we are using the Dashboard example, use the **integrate-dashboard-web-app** sample code located in the [GitHub repository](#). Once you've got the application running in your browser, follow these steps to embed the dashboard you created earlier into the web page:

1. Select **Sign in to Power BI**, which grants the application access to the dashboards in your Power BI account.
2. Select the **Get Dashboards** button, which displays your account's Dashboards in a table. Find the name of the dashboard you created earlier, **powerbi-embedded-dashboard**, and copy the corresponding **EmbedUrl**.
3. Finally, paste the **EmbedUrl** into the corresponding text field and select **Embed Dashboard**. You can now view the same dashboard embedded within a web application.

Next steps

In this tutorial, you created a simple Stream Analytics job, analyzed the incoming data, and presented results in a Power BI dashboard. To learn more about Stream Analytics jobs, continue to the next tutorial:

[Run Azure Functions within Stream Analytics jobs](#)

Tutorial: Run Azure Functions from Azure Stream Analytics jobs

6/13/2019 • 7 minutes to read • [Edit Online](#)

You can run Azure Functions from Azure Stream Analytics by configuring Functions as one of the output sinks to the Stream Analytics job. Functions are an event-driven, compute-on-demand experience that lets you implement code that is triggered by events occurring in Azure or third-party services. This ability of Functions to respond to triggers makes it a natural output to Stream Analytics jobs.

Stream Analytics invokes Functions through HTTP triggers. The Functions output adapter allows users to connect Functions to Stream Analytics, such that the events can be triggered based on Stream Analytics queries.

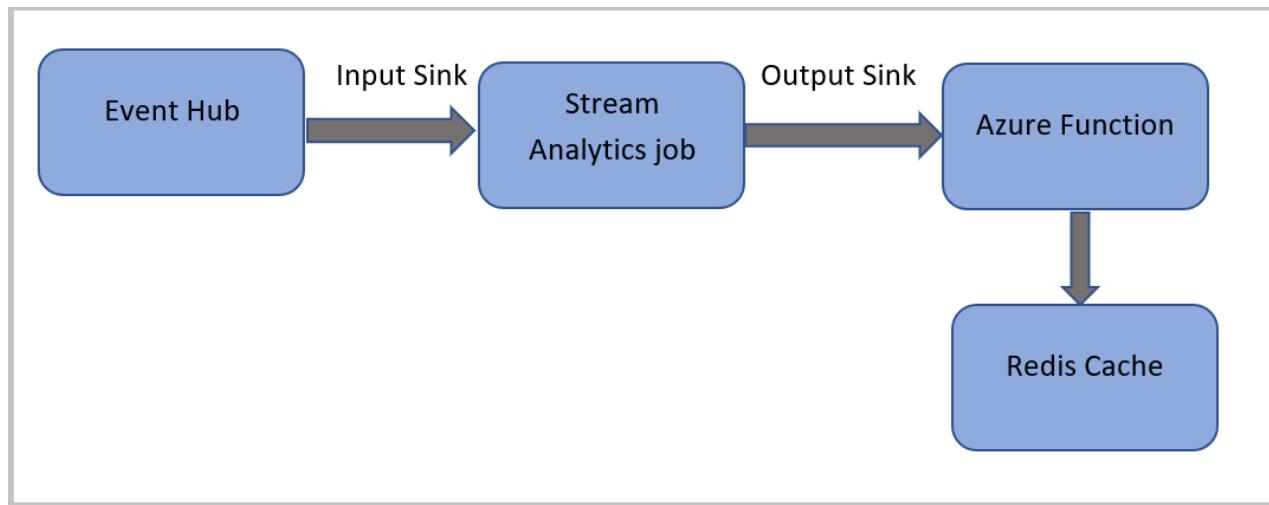
In this tutorial, you learn how to:

- Create and run a Stream Analytics job
- Create an Azure Cache for Redis instance
- Create an Azure Function
- Check Azure Cache for Redis for results

If you don't have an Azure subscription, create a [free account](#) before you begin.

Configure a Stream Analytics job to run a function

This section demonstrates how to configure a Stream Analytics job to run a function that writes data to Azure Cache for Redis. The Stream Analytics job reads events from Azure Event Hubs, and runs a query that invokes the function. This function reads data from the Stream Analytics job, and writes it to Azure Cache for Redis.



Create a Stream Analytics job with Event Hubs as input

Follow the [Real-time fraud detection](#) tutorial to create an event hub, start the event generator application, and create a Stream Analytics job. Skip the steps to create the query and the output. Instead, see the following sections to set up an Azure Functions output.

Create an Azure Cache for Redis instance

1. Create a cache in Azure Cache for Redis by using the steps described in [Create a cache](#).

2. After you create the cache, under **Settings**, select **Access Keys**. Make a note of the **Primary connection string**.

The screenshot shows the 'Access Keys' section of the Azure Cache for Redis settings. On the left, there's a sidebar with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (with 'Access keys' highlighted in red), and Advanced settings. The main area has sections for Primary and Secondary keys, each with a placeholder like '<primary key>' or '<secondary key>'. Below these are fields for 'Primary connection string (StackExchange.Redis)' and 'Secondary connection string (StackExchange.Redis)', both containing '<primary connection string>' and '<secondary connection string>' respectively. A red box highlights the 'Primary connection string' field.

Create a function in Azure Functions that can write data to Azure Cache for Redis

1. See the [Create a function app](#) section of the Functions documentation. This walks you through how to create a function app and an [HTTP-triggered function in Azure Functions](#), by using the CSharp language.
2. Browse to the **run.csx** function. Update it with the following code. Replace "**<your Azure Cache for Redis connection string goes here>**" with the Azure Cache for Redis primary connection string that you retrieved in the previous section.

```

using System;
using System.Net;
using System.Threading.Tasks;
using StackExchange.Redis;
using Newtonsoft.Json;
using System.Configuration;

public static async Task<HttpResponseMessage> Run(HttpRequestMessage req, TraceWriter log)
{
    log.Info($"C# HTTP trigger function processed a request. RequestUri={req.RequestUri}");

    // Get the request body
    dynamic dataArray = await req.Content.ReadAsAsync<object>();

    // Throw an HTTP Request Entity Too Large exception when the incoming batch(dataArray) is greater
    // than 256 KB. Make sure that the size value is consistent with the value entered in the Stream Analytics
    // portal.

    if (dataArray.ToString().Length > 262144)
    {
        return new HttpResponseMessage(HttpStatusCode.RequestEntityTooLarge);
    }
    var connection = ConnectionMultiplexer.Connect("<your Azure Cache for Redis connection string goes
here>");
    log.Info($"Connection string.. {connection}");

    // Connection refers to a property that returns a ConnectionMultiplexer
    IDatabase db = connection.GetDatabase();
    log.Info($"Created database {db}");

    log.Info($"Message Count {dataArray.Count}");

    // Perform cache operations using the cache object. For example, the following code block adds few
    integral data types to the cache
    for (var i = 0; i < dataArray.Count; i++)
    {
        string time = dataArray[i].time;
        string callingnum1 = dataArray[i].callingnum1;
        string key = time + " - " + callingnum1;
        db.StringSet(key, dataArray[i].ToString());
        log.Info($"Object put in database. Key is {key} and value is {dataArray[i].ToString()}");

        // Simple get of data types from the cache
        string value = db.StringGet(key);
        log.Info($"Database got: {value}");
    }

    return req.CreateResponse(HttpStatusCode.OK, "Got");
}

```

When Stream Analytics receives the "HTTP Request Entity Too Large" exception from the function, it reduces the size of the batches it sends to Functions. The following code ensures that Stream Analytics doesn't send oversized batches. Make sure that the maximum batch count and size values used in the function are consistent with the values entered in the Stream Analytics portal.

```

if (dataArray.ToString().Length > 262144)
{
    return new HttpResponseMessage(HttpStatusCode.RequestEntityTooLarge);
}

```

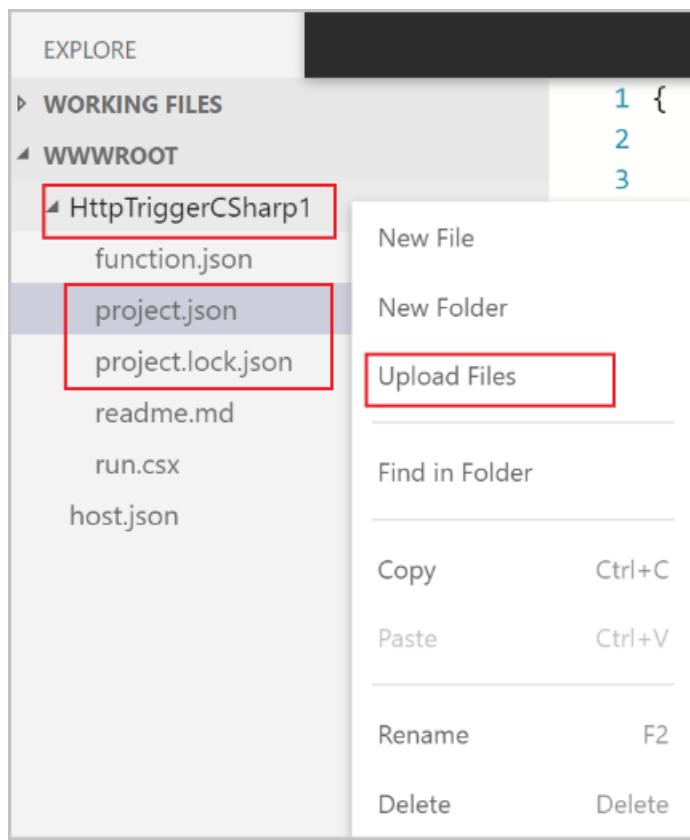
3. In a text editor of your choice, create a JSON file named **project.json**. Paste the following code, and save it on your local computer. This file contains the NuGet package dependencies required by the C# function.

```
{
  "frameworks": {
    "net46": {
      "dependencies": {
        "StackExchange.Redis": "1.1.603",
        "Newtonsoft.Json": "9.0.1"
      }
    }
  }
}
```

4. Go back to the Azure portal. From the **Platform features** tab, browse to your function. Under **Development Tools**, select **App Service Editor**.

The screenshot shows the Azure portal interface. The top navigation bar has 'Search' and 'Overview' tabs, with 'Platform features' currently selected. The main content area is titled 'Platform features' and contains a search bar. It is divided into several sections: **GENERAL SETTINGS** (Function app settings, Application settings, Properties, Backups, All settings), **NETWORKING** (Networking, SSL, Custom domains, Authentication / Authorization, Managed service identity, Push notifications), **API** (API definition, CORS), **APP SERVICE PLAN** (App Service plan, Quotas), **MONITORING** (Diagnostic logs, Log streaming, Process explorer), and **RESOURCE MANAGEMENT** (Activity log, Access control (IAM), Tags, Locks, Automation script). On the left, the sidebar shows 'Microsoft Azure Internal Consumption' and a tree view of 'Function Apps' under 'SAfun1'. The 'Functions' node is expanded, showing 'HttpTriggerCSharp1' with options to Integrate, Manage, and Monitor. Other nodes like 'Proxies' and 'Slots (preview)' are also listed.

5. In the App Service Editor, right-click your root directory, and upload the **project.json** file. After the upload is successful, refresh the page. You should now see an autogenerated file named **project.lock.json**. The autogenerated file contains references to the .dll files that are specified in the project.json file.



Update the Stream Analytics job with the function as output

1. Open your Stream Analytics job on the Azure portal.
2. Browse to your function, and select **Overview > Outputs > Add**. To add a new output, select **Azure Function** for the sink option. The Functions output adapter has the following properties:

PROPERTY NAME	DESCRIPTION
Output alias	A user-friendly name that you use in the job's query to reference the output.
Import option	You can use the function from the current subscription, or provide the settings manually if the function is located in another subscription.
Function App	Name of your Functions app.
Function	Name of the function in your Functions app (name of your run.csx function).
Max Batch Size	Sets the maximum size for each output batch which is sent to your function in bytes. By default, this value is set to 262,144 bytes (256 KB).
Max Batch Count	Specifies the maximum number of events in each batch that is sent to the function. The default value is 100. This property is optional.
Key	Allows you to use a function from another subscription. Provide the key value to access your function. This property is optional.

- Provide a name for the output alias. In this tutorial, it is named **saop1**, but you can use any name of your choice. Fill in other details.
- Open your Stream Analytics job, and update the query to the following. If you did not name your output sink **saop1**, remember to change it in the query.

```

SELECT
    System.Timestamp as Time, CS1.CallingIMSI, CS1.CallingNum as CallingNum1,
    CS2.CallingNum as CallingNum2, CS1.SwitchNum as Switch1, CS2.SwitchNum as Switch2
INTO saop1
FROM CallStream CS1 TIMESTAMP BY CallRecTime
    JOIN CallStream CS2 TIMESTAMP BY CallRecTime
        ON CS1.CallingIMSI = CS2.CallingIMSI AND DATEDIFF(ss, CS1, CS2) BETWEEN 1 AND 5
WHERE CS1.SwitchNum != CS2.SwitchNum

```

- Start the telcodatagen.exe application by running the following command in command line. The command uses the format `telcodatagen.exe [#NumCDRsPerHour] [SIM Card Fraud Probability] [#DurationHours]`.

```
telcodatagen.exe 1000 0.2 2
```

- Start the Stream Analytics job.

Check Azure Cache for Redis for results

- Browse to the Azure portal, and find your Azure Cache for Redis. Select **Console**.
- Use [Azure Cache for Redis commands](#) to verify that your data is in Azure Cache for Redis. (The command takes the format `Get {key}`.) For example:

Get "12/19/2017 21:32:24 - 123414732"

This command should print the value for the specified key:

```
>Get "12/19/2017 21:32:24 - 123414732"
"(\"time\": \"2017-12-19T21:32:24.14Z\", \"callingimsi\": \"466922702346260\", \"callingnum1\": \"123414732\", \"callingnum2\": \"123488173\", \"switch1\": \"US\", \"switch2\": \"UK\")"
>
```

Error handling and retries

In the event of a failure while sending events to Azure Functions, Stream Analytics retries to successfully complete the operation. However, there are some failures for which retries are not attempted and they are as follows:

- HttpRequestExceptions
- Request Entity Too Large (Http error code 413)
- ApplicationExceptions

Known issues

In the Azure portal, when you try to reset the Max Batch Size/ Max Batch Count value to empty (default), the value changes back to the previously entered value upon save. Manually enter the default values for these fields in this case.

The use of [Http routing](#) on your Azure Functions is currently not supported by Stream Analytics.

Clean up resources

When no longer needed, delete the resource group, the streaming job, and all related resources. Deleting the job avoids billing the streaming units consumed by the job. If you're planning to use the job in future, you can stop it and re-start it later when you need. If you are not going to continue to use this job, delete all resources created by this quickstart by using the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

In this tutorial, you have create a simple Stream Analytics job, that runs an Azure Function, to learn more about Stream Analytics jobs, continue to the next tutorial:

[Run JavaScript user-defined functions within Stream Analytics jobs](#)

Tutorial: Azure Stream Analytics JavaScript user-defined functions

6/21/2019 • 5 minutes to read • [Edit Online](#)

Azure Stream Analytics supports user-defined functions written in JavaScript. With the rich set of **String**, **RegExp**, **Math**, **Array**, and **Date** methods that JavaScript provides, complex data transformations with Stream Analytics jobs become easier to create.

In this tutorial, you learn how to:

- Define a JavaScript user-defined function
- Add the function to the portal
- Define a query that runs the function

If you don't have an Azure subscription, create a [free account](#) before you begin.

JavaScript user-defined functions

JavaScript user-defined functions support stateless, compute-only scalar functions that do not require external connectivity. The return value of a function can only be a scalar (single) value. After you add a JavaScript user-defined function to a job, you can use the function anywhere in the query, like a built-in scalar function.

Here are some scenarios where you might find JavaScript user-defined functions useful:

- Parsing and manipulating strings that have regular expression functions, for example, **Regexp_Replace()** and **Regexp_Extract()**
- Decoding and encoding data, for example, binary-to-hex conversion
- Performing mathematic computations with JavaScript **Math** functions
- Performing array operations like sort, join, find, and fill

Here are some things that you cannot do with a JavaScript user-defined function in Stream Analytics:

- Call out external REST endpoints, for example, performing reverse IP lookup or pulling reference data from an external source
- Perform custom event format serialization or deserialization on inputs/outputs
- Create custom aggregates

Although functions like **Date.GetDate()** or **Math.random()** are not blocked in the functions definition, you should avoid using them. These functions **do not** return the same result every time you call them, and the Azure Stream Analytics service does not keep a journal of function invocations and returned results. If a function returns different result on the same events, repeatability is not guaranteed when a job is restarted by you or by the Stream Analytics service.

Add a JavaScript user-defined function in the Azure portal

To create a simple JavaScript user-defined function under an existing Stream Analytics job, follow these steps:

NOTE

These steps work on the Stream Analytics jobs configured to run in the cloud. If your Stream Analytics job is configured to run on Azure IoT Edge, instead use Visual Studio and [write the user-defined function using C#](#).

1. In the Azure portal, find your Stream Analytics job.
2. Under the **Job topology** heading, select **Functions**. An empty list of functions appears.
3. To create a new user-defined function, select **+ Add**.
4. On the **New Function** blade, for **Function Type**, select **JavaScript**. A default function template appears in the editor.
5. For the **UDF alias**, enter **hex2Int**, and change the function implementation as follows:

```
// Convert Hex value to integer.  
function hex2Int(hexValue) {  
    return parseInt(hexValue, 16);  
}
```

6. Select **Save**. Your function appears in the list of functions.
7. Select the new **hex2Int** function, and check the function definition. All functions have a **UDF** prefix added to the function alias. You need to *include the prefix* when you call the function in your Stream Analytics query. In this case, you call **UDF.hex2Int**.

Call a JavaScript user-defined function in a query

1. In the query editor, under the **Job topology** heading, select **Query**.
2. Edit your query, and then call the user-defined function, like this:

```
SELECT  
    time,  
    UDF.hex2Int(offset) AS IntOffset  
INTO  
    output  
FROM  
    InputStream
```

3. To upload the sample data file, right-click the job input.
4. To test your query, select **Test**.

Supported JavaScript objects

Azure Stream Analytics JavaScript user-defined functions support standard, built-in JavaScript objects. For a list of these objects, see [Global Objects](#).

Stream Analytics and JavaScript type conversion

There are differences in the types that the Stream Analytics query language and JavaScript support. This table lists the conversion mappings between the two:

STREAM ANALYTICS	JAVASCRIPT
bigint	Number (JavaScript can only represent integers up to precisely 2^{53})
DateTime	Date (JavaScript only supports milliseconds)
double	Number
nvarchar(MAX)	String
Record	Object
Array	Array
NULL	Null

Here are JavaScript-to-Stream Analytics conversions:

JAVASCRIPT	STREAM ANALYTICS
Number	Bigint (if the number is round and between long.MinValue and long.MaxValue; otherwise, it's double)
Date	DateTime
String	nvarchar(MAX)
Object	Record
Array	Array
Null, Undefined	NULL
Any other type (for example, a function or error)	Not supported (results in runtime error)

JavaScript language is case sensitive and casing of the object fields in JavaScript code must match the casing of the fields in the incoming data. Please note that jobs with compatibility level 1.0 will convert fields from SQL SELECT statement to be lowercase. Under compatibility level 1.1 and higher, fields from SELECT statement will have the same casing as specified in the SQL query.

Troubleshooting

JavaScript runtime errors are considered fatal, and are surfaced through the Activity log. To retrieve the log, in the Azure portal, go to your job and select **Activity log**.

Other JavaScript user-defined function patterns

Write nested JSON to output

If you have a follow-up processing step that uses a Stream Analytics job output as input, and it requires a JSON format, you can write a JSON string to output. The next example calls the **JSON.stringify()** function to pack all name/value pairs of the input, and then write them as a single string value in output.

JavaScript user-defined function definition:

```
function main(x) {  
    return JSON.stringify(x);  
}
```

Sample query:

```
SELECT  
    DataString,  
    DataValue,  
    HexValue,  
    UDF.json_stringify(input) As InputEvent  
INTO  
    output  
FROM  
    input PARTITION BY PARTITIONID
```

Clean up resources

When no longer needed, delete the resource group, the streaming job, and all related resources. Deleting the job avoids billing the streaming units consumed by the job. If you're planning to use the job in future, you can stop it and re-start it later when you need. If you are not going to continue to use this job, delete all resources created by this quickstart by using the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Get help

For additional help, try our [Azure Stream Analytics forum](#).

Next steps

In this tutorial, you have created a Stream Analytics job that runs a simple JavaScript user-defined function. To learn more about Stream Analytics, continue to the real-time scenario articles:

[Real-time Twitter sentiment analysis in Azure Stream Analytics](#)

Tutorial: Deploy an Azure Stream Analytics job with CI/CD using Azure Pipelines

2/22/2019 • 5 minutes to read • [Edit Online](#)

This tutorial describes how to set up continuous integration and deployment for an Azure Stream Analytics job using Azure Pipelines.

In this tutorial, you learn how to:

- Add source control to your project
- Create a build pipeline in Azure Pipelines
- Create a release pipeline in Azure Pipelines
- Automatically deploy and upgrade an application

Prerequisites

Before you start, make sure you have the following:

- If you don't have an Azure subscription, create a [free account](#).
- Install [Visual Studio](#) and the **Azure development** or **Data Storage and Processing** workloads.
- Create a [Stream Analytics project in Visual Studio](#).
- Create an [Azure DevOps organization](#).

Configure NuGet package dependency

In order to do auto build and auto deployment on an arbitrary machine, you need to use the NuGet package `Microsoft.Azure.StreamAnalytics.CICD`. It provides the MSBuild, local run, and deployment tools that support the continuous integration and deployment process of Stream Analytics Visual Studio projects. For more information, see [Stream Analytics CI/CD tools](#).

Add **packages.config** to your project directory.

```
<?xml version="1.0" encoding="utf-8"?>
<packages>
<package id="Microsoft.Azure.StreamAnalytics.CICD" version="1.0.0" targetFramework="net452" />
</packages>
```

Share your Visual Studio solution to a new Azure Repos Git repo

Share your application source files to a project in Azure DevOps so you can generate builds.

1. Create a new local Git repo for your project by selecting **Add to Source Control**, then **Git** on the status bar in the lower right-hand corner of Visual Studio.
2. In the **Synchronization** view in **Team Explorer**, select the **Publish Git Repo** button under **Push to Azure DevOps Services**.

Team Explorer - Synchronization

Push | myRepo

Backup and share your code. Publish it to a Git service.

► Push to Visual Studio Team Services

 Team Services
Microsoft Corporation

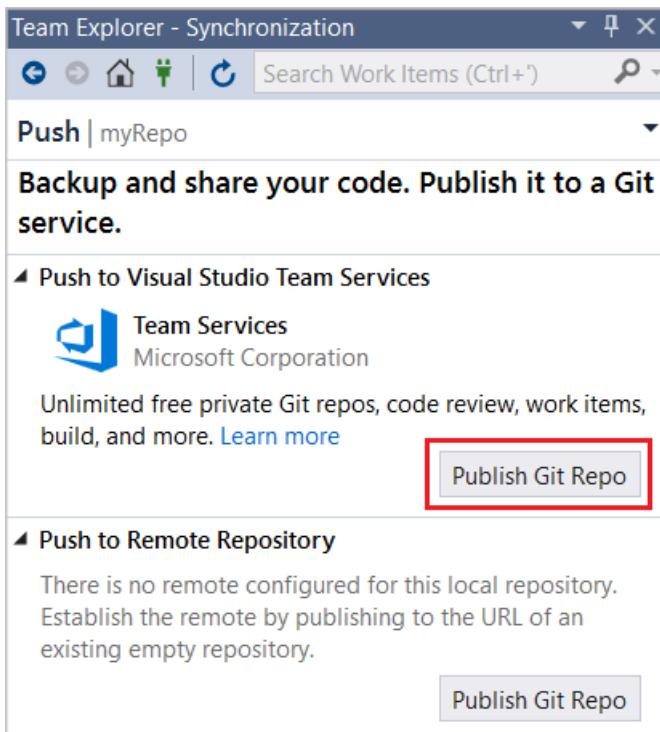
Unlimited free private Git repos, code review, work items, build, and more. [Learn more](#)

Publish Git Repo

► Push to Remote Repository

There is no remote configured for this local repository.
Establish the remote by publishing to the URL of an existing empty repository.

Publish Git Repo



3. Verify your email and select your organization in the **Azure DevOps Services Domain** drop-down. Enter your repository name and select **Publish repository**.

Team Explorer - Synchronization

Push | myRepo

Backup and share your code. Publish it to a Git service.

► Push to Visual Studio Team Services

 Team Services
Microsoft Corporation

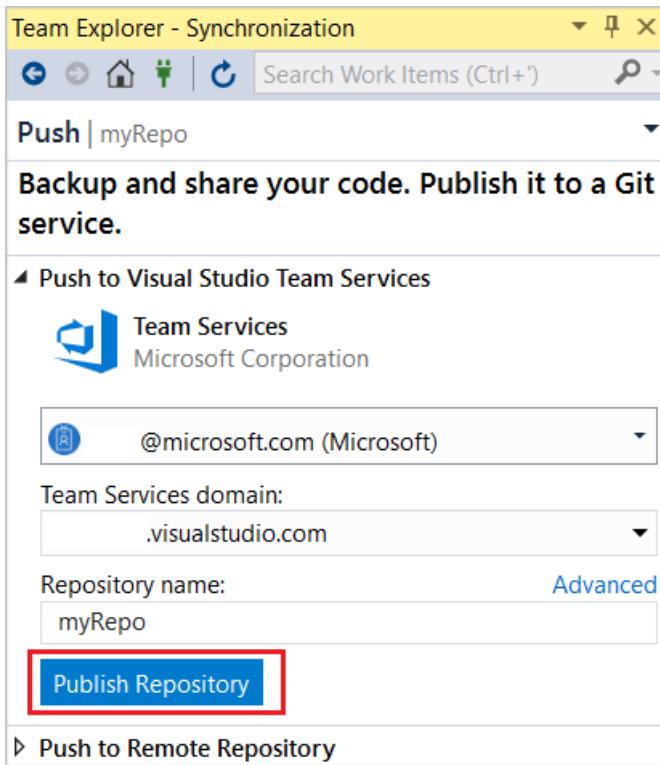
 @microsoft.com (Microsoft)

Team Services domain:
.visualstudio.com

Repository name: [Advanced](#)
myRepo

Publish Repository

► Push to Remote Repository



Publishing the repo creates a new project in your organization with the same name as the local repo. To create the repo in an existing project, click **Advanced** next to **Repository name**, and select a project. You can view your code in the browser by selecting **See it on the web**.

Configure continuous delivery with Azure DevOps

An Azure Pipelines build pipeline describes a workflow comprised of build steps that are executed sequentially. Learn more about [Azure Pipelines build pipelines](#).

An Azure Pipelines release pipeline describes a workflow that deploys an application package to a cluster. When used together, the build pipeline and release pipeline execute the entire workflow starting with source files and

ending with a running application in your cluster. Learn more about Azure Pipelines [release pipelines](#).

Create a build pipeline

Open a web browser and navigate to the project you just created in [Azure DevOps](#).

- Under the **Build & Release** tab, select **Builds**, and then **+New**. Select **Azure DevOps Services Git** and **Continue**.

Select a source

VSTS Git GitHub GitHub Enterprise Subversion Bitbucket® Cloud External Git

Team project: myRepo

Repository: myRepo

Default branch for manual and scheduled builds: master

Continue

- In **Select a template**, click **Empty Process** to start with an empty pipeline.

Select a template

Or start with an Empty process

Search

Featured

.NET Desktop

Build and test a .NET or Windows classic desktop solution.

Android

Build, test, sign, and align an Android APK.

Choose a template

- Under **Triggers**, enable continuous integration by checking **Enable continuous integration** trigger status. Select **Save and queue** to manually start a build.

Continuous integration

MyFirstProject Enabled

Enable continuous integration

Batch changes while a build is in progress

Branch filters

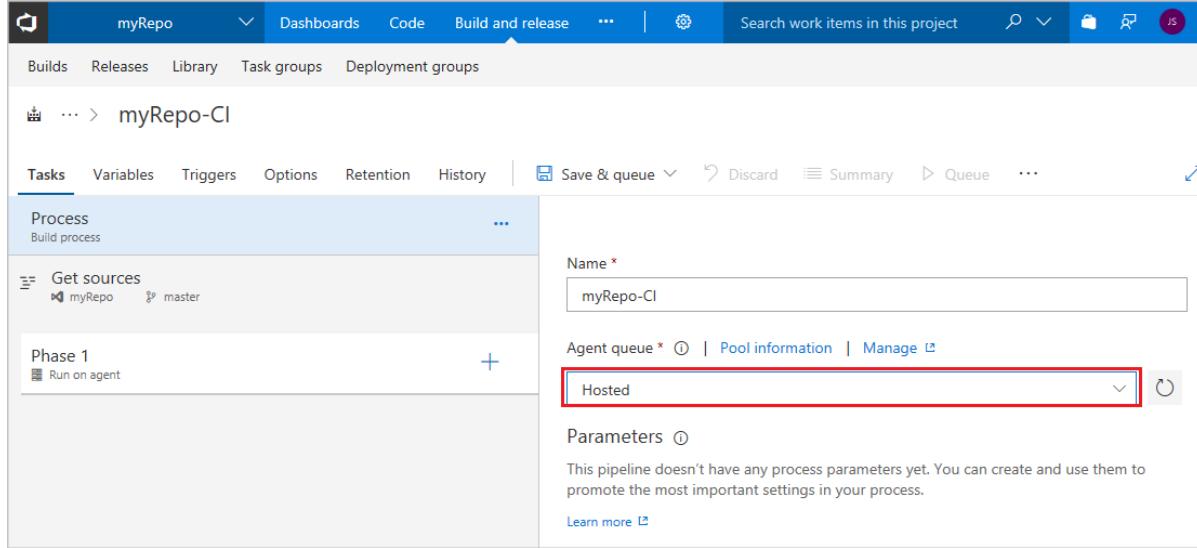
Type: Include Branch specification: master

Path filters

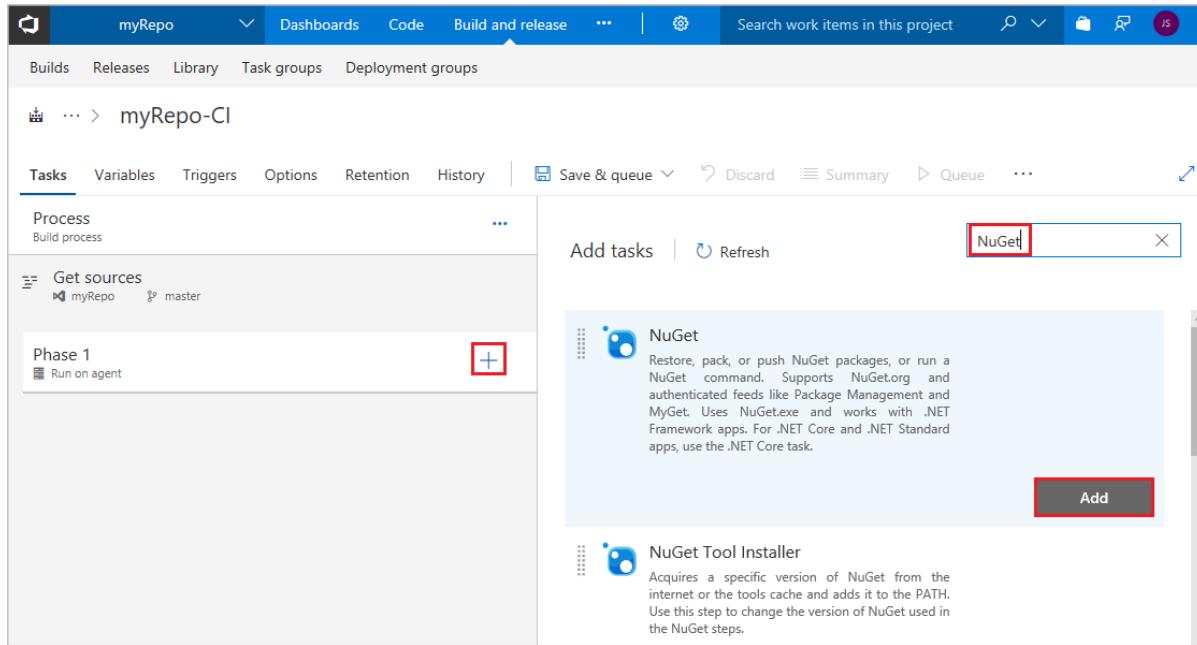
Tasks Variables Triggers Options Retention History Save & queue Discard Summary Queue ...

4. Builds are also triggered upon push or check-in. To check your build progress, switch to the **Builds** tab. Once you verify that the build executes successfully, you must define a release pipeline that deploys your application to a cluster. Right click on the ellipses next to your build pipeline and select **Edit**.

5. In **Tasks**, enter "Hosted" as the **Agent queue**.



6. In **Phase 1**, click **+** and add a **NuGet** task.



7. Expand **Advanced** and add `$(Build.SourcesDirectory)\packages` to **Destination directory**. Keep the remaining default NuGet configuration values.

The screenshot shows the 'myRepo / myRepo Team' dashboard in Azure DevOps. Under the 'Builds' tab, the 'myRepo-Cl' pipeline is selected. The 'Tasks' tab is active, showing a process step and a Phase 1 step. The 'NuGet restore' task is currently selected. On the right, its configuration pane is open, displaying fields like 'Display name' (NuGet restore), 'Command' (restore), and 'Path to solution, packages.config, or project.json' (**/*.sln). A red box highlights the 'Destination directory' field, which contains the value '\$(Build.SourcesDirectory)/packages'. Other settings shown include 'Feed(s) I select here' (selected), 'Use packages from NuGet.org' (checked), and 'Advanced' options.

8. In **Phase 1**, click **+** and add a **MSBuild** task.

The screenshot shows the same 'myRepo-Cl' pipeline configuration. The 'Add tasks' search bar at the top right is highlighted with a red box and contains the text 'MSBuild'. Below the search bar, the 'MSBuild' task card is displayed, featuring its icon, name, and brief description. A large blue 'Add' button is located at the bottom right of the task card, which is also highlighted with a red box.

9. Change the **MSBuild Arguments** to the following:

```
/p:CompilerTaskAssemblyFile="Microsoft.WindowsAzure.StreamAnalytics.Common.CompileService.dll"  
/p:ASATargetsFilePath="$(Build.SourcesDirectory)\packages\Microsoft.Azure.StreamAnalytics.CICD.1.0.0\bui  
ld\StreamAnalytics.targets"
```

10. In **Phase 1**, click **+** and add an **Azure Resource Group Deployment** task.

11. Expand **Azure Details** and fill out the configuration with the following:

SETTING	SUGGESTED VALUE
Subscription	Choose your subscription.
Action	Create or update resource group
Resource Group	Enter a resource group name.
Template	[Your solution path]\bin\Debug\Deploy\[Your project name].JobTemplate.json
Template parameters	[Your solution path]\bin\Debug\Deploy\[Your project name].JobTemplate.parameters.json

SETTING	SUGGESTED VALUE
Override template parameters	Type the template parameters to override in the textbox. Example, --storageName fabrikam --adminUsername \$(vmusername) --adminPassword \$(password) --azureKeyVaultName \$(fabrikamFibre). This property is optional, but your build will result in errors if key parameters are not overridden.

The screenshot shows the Azure DevOps Build Pipeline configuration for the 'myRepo-CI' pipeline. The pipeline structure is as follows:

- Get sources:** myRepo, master
- Phase 1:** Run on agent
 - NuGet restore** (NuGet)
 - Build solution **/*.sln** (MSBuild)
 - Azure Deployment:Create Or Update Res...** (Azure Resource Group Deployment)
 - Action:** Create or update resource group
 - Resource group:** (dropdown menu)
 - Location:** West US

The 'Azure Deployment' task is expanded, showing its configuration. It is set to 'Create or update resource group' in 'Action', 'West US' in 'Location', and uses a 'Linked artifact' for 'Template location'. The 'Template parameters' field contains '[Your solution path]\bin\Debug\Deploy\[ProjectName].JobTemplate.json'. The 'Override template parameters' field is empty.

12. Click **Save & Queue** to test the build pipeline.

The screenshot shows the Azure DevOps interface for a build pipeline named 'myRepo-CI'. The pipeline consists of several tasks:

- Get sources (myRepo, master branch)
- NuGet restore
- Build solution (**/*.sln)
- Azure Deployment:Create Or Update Resource Group (Azure Resource Group Deployment)

The 'Azure Deployment' task is currently selected. The 'Save & queue' button is highlighted with a red box. The 'Azure Details' section contains the following configuration:

- Azure subscription: Visual Studio Ultimate with MSDN (647a4e4a-cc7e-443c-aa0d-4248a4c5af)
- Action: Create or update resource group
- Resource group: Default-Storage-WestUS
- Location: West US
- Template location: Linked artifact
- Template: [Your solution path]\bin\Debug\Deploy\[ProjectName].JobTemplate.json
- Template parameters: [Your solution path]\bin\Debug\Deploy\[ProjectName].JobTemplate.json
- Override template parameters: (empty)

Failed build process

You may receive errors for null deployment parameters if you did not override template parameters in the **Azure Resource Group Deployment** task of your build pipeline. Return to the build pipeline and override the null parameters to resolve the error.

Build process failed

3 error(s) / 0 warning(s)

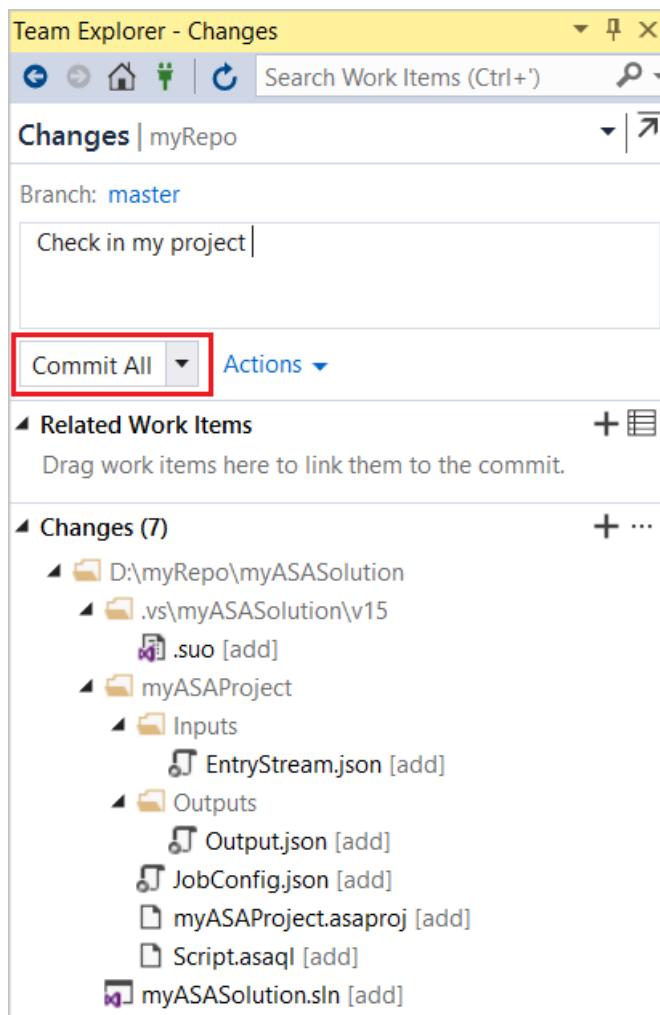
- The value of deployment parameter 'Input_Input_accountKey_asaehstorage' is null. Please specify the value or use the parameter reference. See <https://aka.ms/arm-deploy/#parameter-file> for details.
- Task failed while creating or updating the template deployment.

Commit and push changes to trigger a release

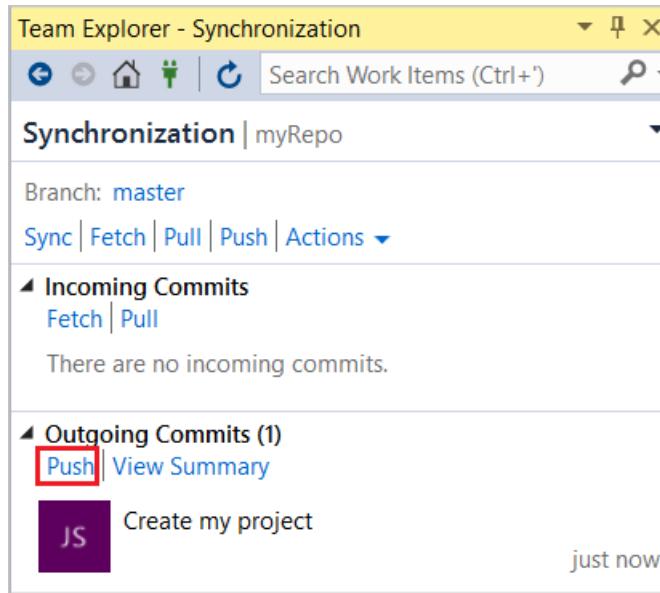
Verify that the continuous integration pipeline is functioning by checking in some code changes to Azure DevOps.

As you write your code, your changes are automatically tracked by Visual Studio. Commit changes to your local Git repository by selecting the pending changes icon from the status bar in the bottom right.

1. On the **Changes** view in Team Explorer, add a message describing your update and commit your changes.



2. Select the unpublished changes status bar icon or the Sync view in Team Explorer. Select **Push** to update your code in Azure DevOps.



Pushing the changes to Azure DevOps Services automatically triggers a build. When the build pipeline successfully completes, a release is automatically created and starts updating the job on the cluster.

Clean up resources

When no longer needed, delete the resource group, the streaming job, and all related resources. Deleting the job avoids billing the streaming units consumed by the job. If you're planning to use the job in future, you can stop it and re-start it later when you need. If you are not going to continue to use this job, delete all resources created by

this tutorial by using the following steps:

1. From the left-hand menu in the Azure portal, click **Resource groups** and then click the name of the resource you created.
2. On your resource group page, click **Delete**, type the name of the resource to delete in the text box, and then click **Delete**.

Next steps

To learn more about using Azure Stream Analytics tools for Visual Studio to set up a continuous integration and deployment process, continue to the set up CI/CD pipeline article:

[Continuously integrate and develop with Stream Analytics tools](#)

Tutorial: Write a C# user-defined function for Azure Stream Analytics Edge job (Preview)

12/7/2018 • 3 minutes to read • [Edit Online](#)

C# user-defined functions (UDFs) created in Visual Studio allow you to extend the Azure Stream Analytics query language with your own functions. You can reuse existing code (including DLLs) and use mathematical or complex logic with C#. There are three ways to implement UDFs: CodeBehind files in a Stream Analytics project, UDFs from a local C# project, or UDFs from an existing package from a storage account. This tutorial uses the CodeBehind method to implement a basic C# function. The UDF feature for Stream Analytics Edge jobs is currently in preview and shouldn't be used in production workloads.

In this tutorial, you learn how to:

- Create a C# user defined function using CodeBehind.
- Test your Stream Analytics Edge job locally.
- Publish your Edge job to Azure.

Prerequisites

Before you start, make sure you've completed the following prerequisites:

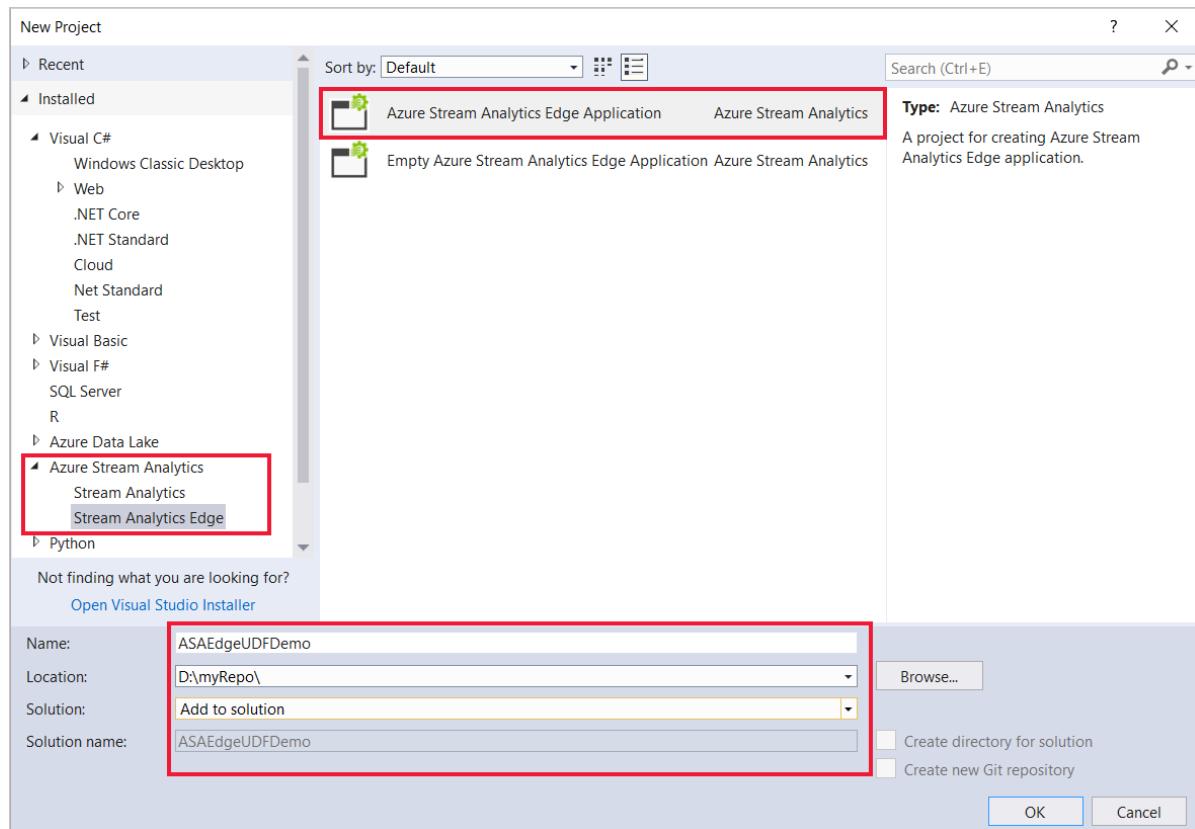
- If you don't have an Azure subscription, create a [free account](#).
- Install [Stream Analytics tools for Visual Studio](#) and the **Azure development or Data Storage and Processing** workloads.
- Take a look at the existing [Stream Analytics Edge development guide](#).

Create a container in your Azure Storage Account

The container you create will be used to store the compiled C# package and deploy the package to your IoT Edge device. Use a dedicated container for each Stream Analytics job. Reusing the same container for multiple Stream Analytics Edge jobs is not supported. If you already have a storage account with existing containers, you may use them. If not, you'll need to [create a new container](#).

Create a Stream Analytics Edge project in Visual Studio

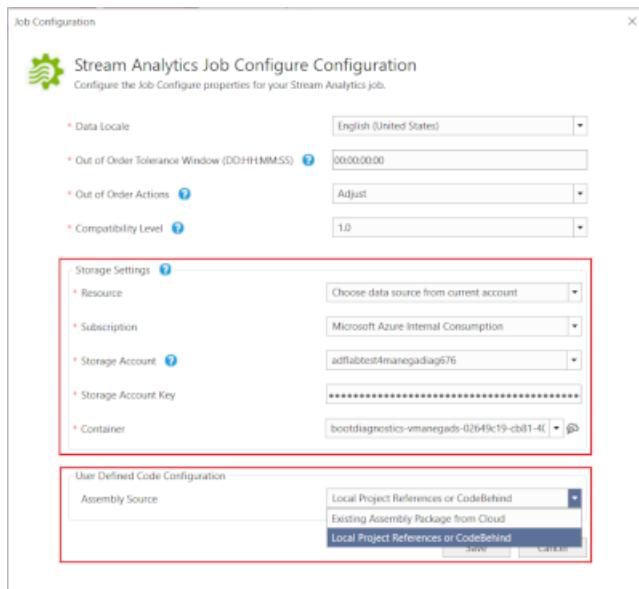
1. Start Visual Studio.
2. Select **File > New > Project**.
3. In the templates list on the left, select **Stream Analytics**, and then select **Azure Stream Analytics Edge Application**.
4. Input the project **Name**, **Location**, and **Solution name**, and select **OK**.



Configure assembly package path

1. Open Visual Studio and navigate to the **Solution Explorer**.
2. Double-click the job configuration file, `EdgeJobConfig.json`.
3. Expand the **User-Defined Code Configuration** section, and fill out the configuration with the following suggested values:

SETTING	SUGGESTED VALUE
Assembly Source	Local Project Reference or CodeBehind
Resource	Choose data from current account
Subscription	Choose your subscription.
Storage Account	Choose your storage account.
Container	Choose the container you created in your storage account.



Write a C# UDF with CodeBehind

A CodeBehind file is a C# file associated with a single ASA Edge query script. Visual Studio tools will automatically zip the CodeBehind file and upload it to your Azure storage account upon submission. All classes must be defined as *public* and all objects must be defined as *static public*.

1. In **Solution Explorer**, expand **Script.asql** to find the **Script.asaql.cs** CodeBehind file.
2. Replace the code with the following sample:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;

namespace ASAEdgeUDFDemo
{
    public class Class1
    {
        // Public static function
        public static Int64 SquareFunction(Int64 a)
        {
            return a * a;
        }
    }
}
```

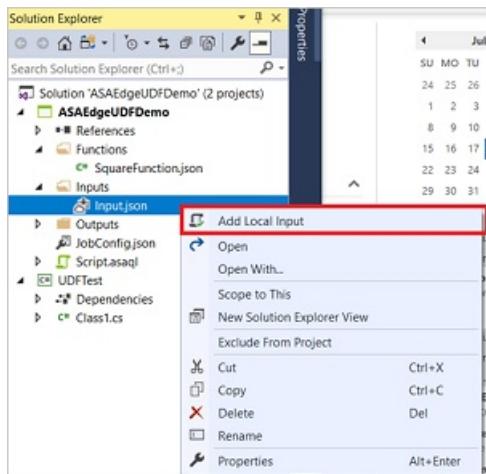
Implement the UDF

1. In **Solution Explorer**, open the **Script.asaql** file.
2. Replace the existing query with the following:

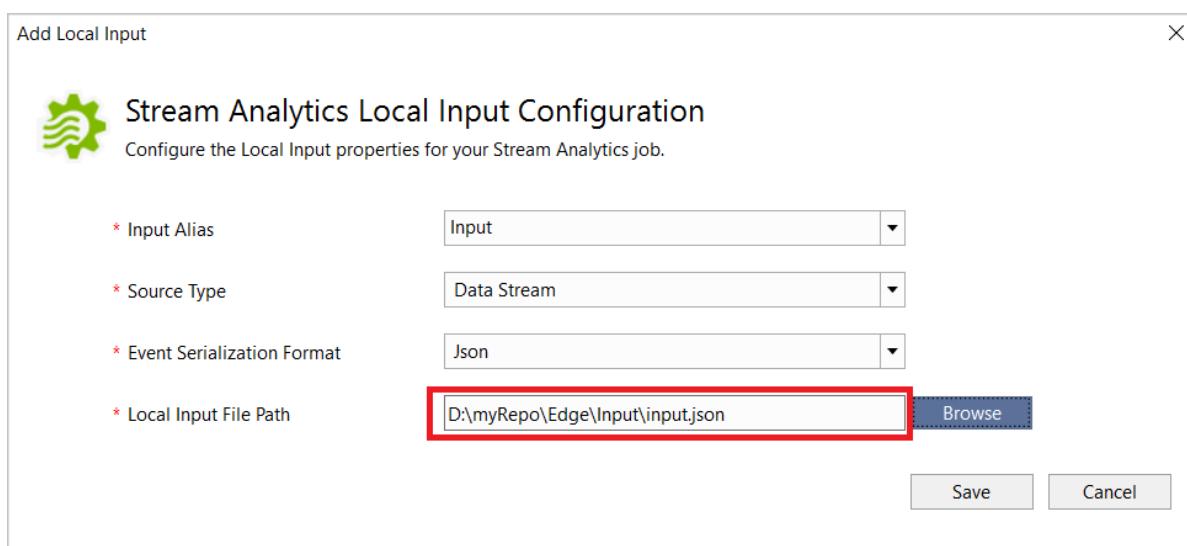
```
SELECT machine.temperature, udf.ASAEdgeUDFDemo_Class1_SquareFunction(try_cast(machine.temperature
as bigint))
INTO Output
FROM Input
```

Local testing

1. Download the Edge [temperature simulator sample data file](#).
2. In **Solution Explorer**, expand **Inputs**, right-click **Input.json**, and select **Add Local Input**.



3. Specify the local input file path for the sample data you downloaded and **Save**.



4. Click **Run Locally** in the script editor. Once the local run has successfully saved the output results, press any key to see the results in table format.

```
C:\WINDOWS\system32\cmd.exe
=====
Start local run =====
7/18/2018 4:07:17 PM : Info : There are 1 input(s) for job
7/18/2018 4:07:17 PM : Info : Input Alias : 'Input' , Input Type : 'Data Stream'
7/18/2018 4:07:17 PM : Info : Compiling script ...
7/18/2018 4:07:17 PM : Info : Current compatibility level for this job is 1.0
7/18/2018 4:07:18 PM : Info : Script compilation succeeded.
7/18/2018 4:07:18 PM : Info : Creating query inputs ...
7/18/2018 4:07:18 PM : Info : Current data locale for inputs of this job is en-US
7/18/2018 4:07:18 PM : Info : Detected several necessary inputs for running this script : input
7/18/2018 4:07:18 PM : Info : Query inputs created
7/18/2018 4:07:18 PM : Info : Executing query ...
7/18/2018 4:07:18 PM : Info : Extract assembly zip file 'C:\Users\sujie\AppData\Local\Temp\812a3892-4a73-490b-8898-90852
54450a6.zip' to 'D:\myRepo\Edge\ASAEdgeUDFDemo\ASALocalRun\2018-07-18-16-07-14\UserCustomCode\CLR'
7/18/2018 4:07:18 PM : Info : Query execute succeeded
7/18/2018 4:07:18 PM : Info : There are 1 output(s) for job
7/18/2018 4:07:19 PM : Info : Output file created D:\myRepo\Edge\ASAEdgeUDFDemo\ASALocalRun\2018-07-18-16-07-14\output.c
sv for : output
7/18/2018 4:07:19 PM : Info : Output file created D:\myRepo\Edge\ASAEdgeUDFDemo\ASALocalRun\2018-07-18-16-07-14\output.j
son for : output
=====
Local run successful : Output results saved to D:\myRepo\Edge\ASAEdgeUDFDemo\ASALocalRun\2018-07-18-16-0
7-14 =====
Press any key to continue . . .
```

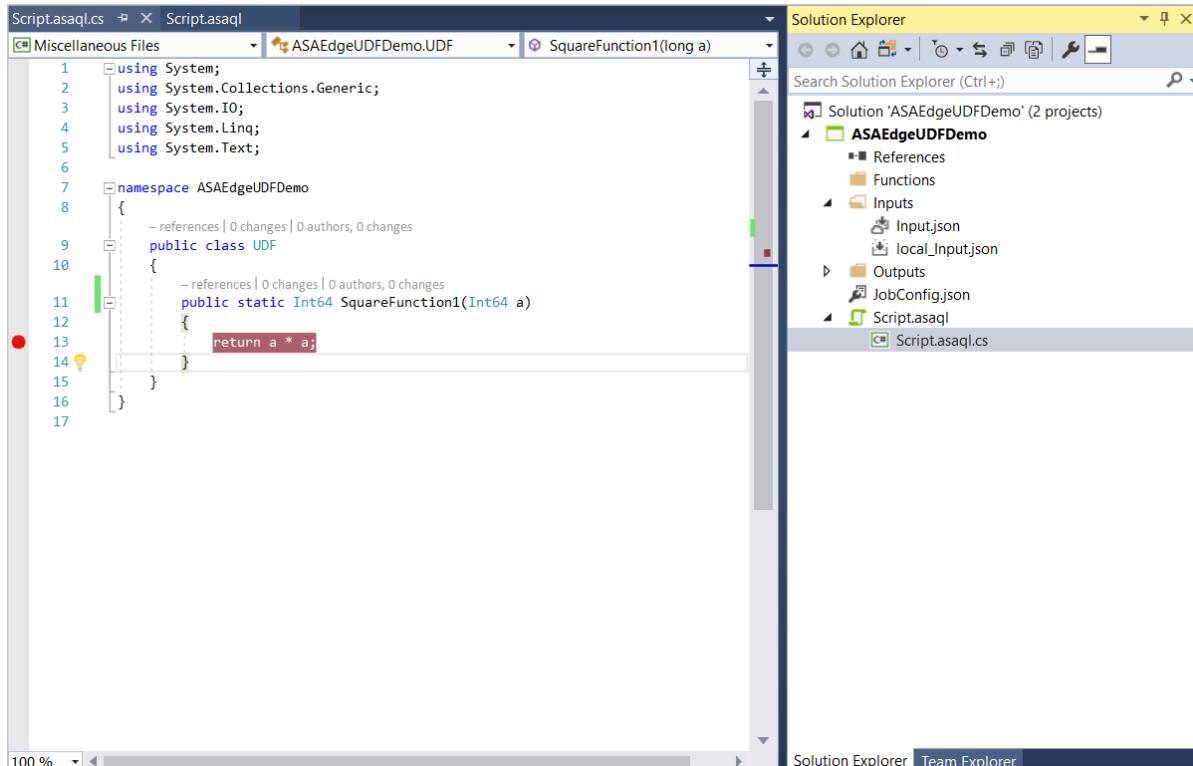
5. You can also select **Open Results Folder** to see the raw files in JSON and CSV format.



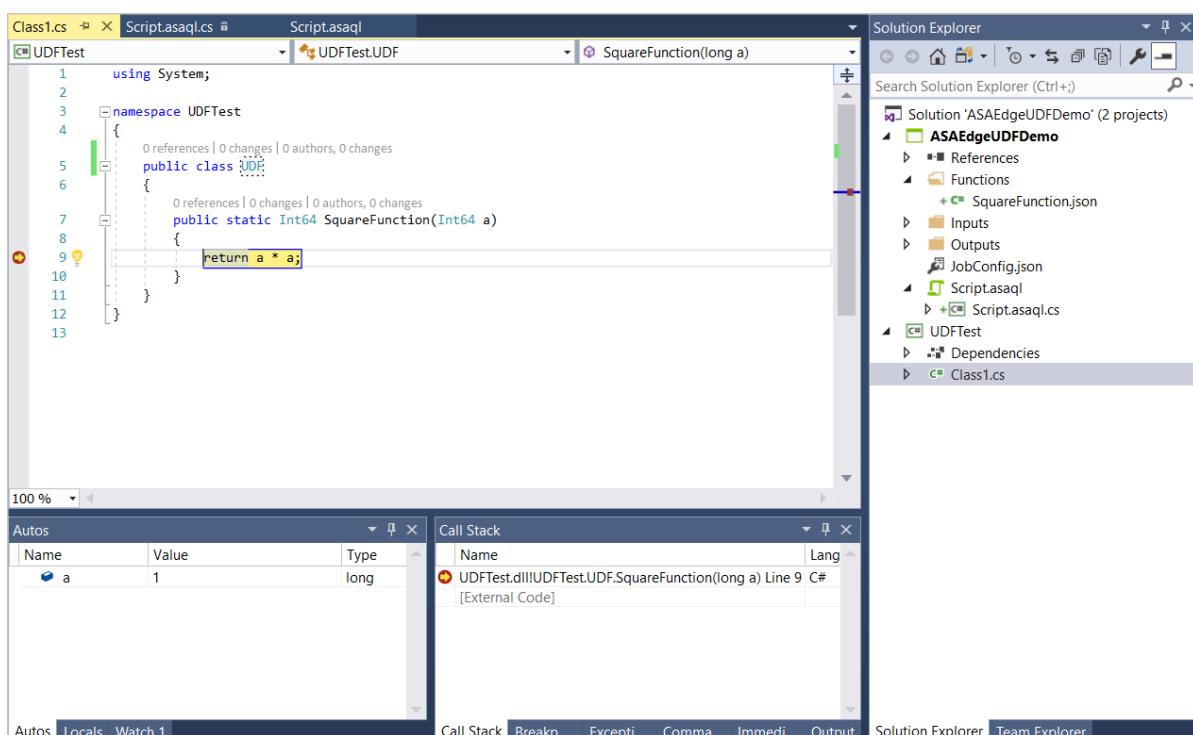
Debug a UDF

You can debug your C# UDF locally the same way you debug standard C# code.

- Add breakpoints in your C# function.

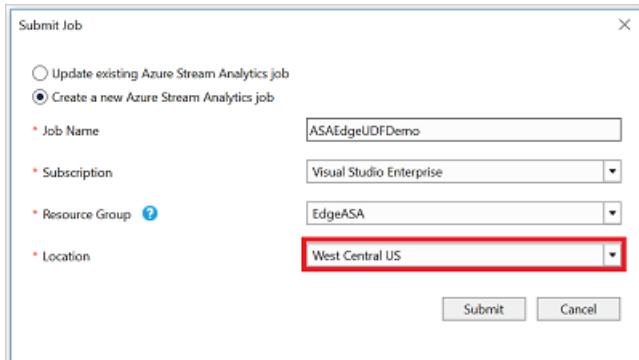


- Press **F5** to start debugging. The program will stop at your breakpoints as expected.



Publish your job to Azure

Once you've tested your query locally, select **Submit to Azure** in the script editor to publish the job to Azure.



Deploy to IoT Edge devices

Your Stream Analytics job is now ready to be deployed as an IoT Edge module. Follow the [IoT Edge quickstart](#) to create an IoT Hub, register an IoT Edge device, and install and start the IoT Edge runtime on your device. Then follow the [deploy the job](#) tutorial to deploy your Stream Analytics job as an IoT Edge module.

Next steps

In this tutorial, you created a simple C# user-defined function using CodeBehind, published your job to Azure, and deployed the job to IoT Edge devices using the IoT Hub portal.

To learn more about the different ways to use C# user-defined functions for Stream Analytics Edge jobs, continue to this article:

[Write C# functions for Azure Stream Analytics](#)

Choose a real-time analytics and streaming processing technology on Azure

5/23/2019 • 2 minutes to read • [Edit Online](#)

There are several services available for real-time analytics and streaming processing on Azure. This article provides the information you need to decide which technology is the best fit for your application.

When to use Azure Stream Analytics

Azure Stream Analytics is the recommended service for stream analytics on Azure. It's meant for a wide range of scenarios that include but aren't limited to:

- Dashboards for data visualization
- Real-time [alerts](#) from temporal and spatial patterns or anomalies
- Extract, Transform, Load (ETL)
- [Event Sourcing pattern](#)
- [IoT Edge](#)

Adding an Azure Stream Analytics job to your application is the fastest way to get streaming analytics up and running in Azure, using the SQL language you already know. Azure Stream Analytics is a job service, so you don't have to spend time managing clusters, and you don't have to worry about downtime with a 99.9% SLA at the job level. Billing is also done at the job level making startup costs low (one Streaming Unit), but scalable (up to 192 Streaming Units). It's much more cost effective to run a few Stream Analytics jobs than it is to run and maintain a cluster.

Azure Stream Analytics has a rich out-of-the-box experience. You can immediately take advantage of the following features without any additional setup:

- Built-in temporal operators, such as [windowed aggregates](#), temporal joins, and temporal analytic functions.
- Native Azure [input](#) and [output](#) adapters
- Support for slow changing [reference data](#) (also known as a lookup tables), including joining with geospatial reference data for geofencing.
- Integrated solutions, such as [Anomaly Detection](#)
- Multiple time windows in the same query
- Ability to compose multiple temporal operators in arbitrary sequences.
- Under 100-ms end-to-end latency from input arriving at Event Hubs, to output landing in Event Hubs, including the network delay from and to Event Hubs, at sustained high throughput

When to use other technologies

You need to input from or output to Kafka

Azure Stream Analytics doesn't have an Apache Kafka input or output adapter. If you have events landing in or need to send to Kafka and you don't have a requirement to run your own Kafka cluster, you can continue to use Stream Analytics by sending events to Event Hubs using the Event Hubs Kafka API without changing the event sender. If you do need to run your own Kafka cluster, you can use Spark Structured Streaming, which is fully supported on [Azure Databricks](#), or Storm on [Azure HDInsight](#).

You want to write UDFs, UDAFs, and custom deserializers in a language other than JavaScript or C#

Azure Stream Analytics supports user-defined functions (UDF) or user-defined aggregates (UDA) in JavaScript for cloud jobs and C# for IoT Edge jobs. C# user-defined deserializers are also supported. If you want to implement a deserializer, a UDF, or a UDA in other languages, such as Java or Python, you can use Spark Structured Streaming. You can also run the Event Hubs **EventProcessorHost** on your own virtual machines to do arbitrary streaming processing.

Your solution is in a multi-cloud or on-premises environment

Azure Stream Analytics is Microsoft's proprietary technology and is only available on Azure. If you need your solution to be portable across Clouds or on-premises, consider open-source technologies such as Spark Structured Streaming or Storm.

Next steps

- [Create a Stream Analytics job by using the Azure portal](#)
- [Create a Stream Analytics job by using Azure PowerShell](#)
- [Create a Stream Analytics job by using Visual Studio](#)
- [Create a Stream Analytics job by using Visual Studio Code](#)

Understand inputs for Azure Stream Analytics

6/21/2019 • 2 minutes to read • [Edit Online](#)

Azure Stream Analytics jobs connect to one or more data inputs. Each input defines a connection to an existing data source. Stream Analytics accepts data incoming from several kinds of event sources including Event Hubs, IoT Hub, and Blob storage. The inputs are referenced by name in the streaming SQL query that you write for each job. In the query, you can join multiple inputs to blend data or compare streaming data with a lookup to reference data, and pass the results to outputs.

Stream Analytics has first-class integration with three kinds of resources as inputs:

- [Azure Event Hubs](#)
- [Azure IoT Hub](#)
- [Azure Blob storage](#)

These input resources can live in the same Azure subscription as your Stream Analytics job, or from a different subscription.

You can use the [Azure portal](#), [Azure PowerShell](#), [.NET API](#), [REST API](#), and [Visual Studio](#) to create, edit, and test Stream Analytics job inputs.

Stream and reference inputs

As data is pushed to a data source, it's consumed by the Stream Analytics job and processed in real time. Inputs are divided into two types: data stream inputs and reference data inputs.

Data stream input

A data stream is an unbounded sequence of events over time. Stream Analytics jobs must include at least one data stream input. Event Hubs, IoT Hub, and Blob storage are supported as data stream input sources. Event Hubs are used to collect event streams from multiple devices and services. These streams might include social media activity feeds, stock trade information, or data from sensors. IoT Hubs are optimized to collect data from connected devices in Internet of Things (IoT) scenarios. Blob storage can be used as an input source for ingesting bulk data as a stream, such as log files.

For more information about streaming data inputs, see [Stream data as input into Stream Analytics](#)

Reference data input

Stream Analytics also supports input known as *reference data*. Reference data is either completely static or changes slowly. It is typically used to perform correlation and lookups. For example, you might join data in the data stream input to data in the reference data, much as you would perform a SQL join to look up static values. Azure Blob storage and Azure SQL Database are currently supported as input sources for reference data. Reference data source blobs have a limit of up to 300 MB in size, depending on the query complexity and allocated Streaming Units (see the [Size limitation](#) section of the reference data documentation for more details).

For more information about reference data inputs, see [Using reference data for lookups in Stream Analytics](#)

Next steps

[Quickstart: Create a Stream Analytics job by using the Azure portal](#)

Stream data as input into Stream Analytics

6/20/2019 • 10 minutes to read • [Edit Online](#)

Stream Analytics has first-class integration with Azure data streams as inputs from three kinds of resources:

- [Azure Event Hubs](#)
- [Azure IoT Hub](#)
- [Azure Blob storage](#)

These input resources can live in the same Azure subscription as your Stream Analytics job or a different subscription.

Compression

Stream Analytics supports compression across all data stream input sources. Supported compression types are: None, GZip, and Deflate compression. Support for compression is not available for reference data. If the input format is Avro data that is compressed, it's handled transparently. You don't need to specify compression type with Avro serialization.

Create, edit, or test inputs

You can use the [Azure portal](#), [Visual Studio](#), and [Visual Studio Code](#) to add and view or edit existing inputs on your streaming job. You can also test input connections and [test queries](#) from sample data from the Azure portal, [Visual Studio](#), and [Visual Studio Code](#). When you write a query, you list the input in the FROM clause. You can get the list of available inputs from the [Query](#) page in the portal. If you wish to use multiple inputs, you can `JOIN` them or write multiple `SELECT` queries.

Stream data from Event Hubs

Azure Event Hubs provides highly scalable publish-subscribe event ingestors. An event hub can collect millions of events per second so that you can process and analyze the massive amounts of data produced by your connected devices and applications. Together, Event Hubs and Stream Analytics provide an end-to-end solution for real-time analytics. Event Hubs lets you feed events into Azure in real-time, and Stream Analytics jobs can process those events in real-time. For example, you can send web clicks, sensor readings, or online log events to Event Hubs. You can then create Stream Analytics jobs to use Event Hubs as the input data streams for real-time filtering, aggregating, and correlation.

`EventEnqueuedUtcTime` is the timestamp of an event's arrival in an event hub and is the default timestamp of events coming from Event Hubs to Stream Analytics. To process the data as a stream using a timestamp in the event payload, you must use the [TIMESTAMP BY](#) keyword.

Event Hubs Consumer groups

You should configure each Stream Analytics event hub input to have its own consumer group. When a job contains a self-join or has multiple inputs, some inputs might be read by more than one reader downstream. This situation impacts the number of readers in a single consumer group. To avoid exceeding the Event Hubs limit of five readers per consumer group per partition, it's a best practice to designate a consumer group for each Stream Analytics job. There is also a limit of 20 consumer groups for a Standard tier event hub. For more information, see [Troubleshoot Azure Stream Analytics inputs](#).

Create an input from Event Hubs

The following table explains each property in the [New input](#) page in the Azure portal to stream data input from

an event hub:

PROPERTY	DESCRIPTION
Input alias	A friendly name that you use in the job's query to reference this input.
Subscription	Choose the subscription in which the Event hub resource exists.
Event Hub namespace	The Event Hub namespace is a container for a set of messaging entities. When you create a new event hub, you also create the namespace.
Event Hub name	The name of the event hub to use as input.
Event Hub policy name	The shared access policy that provides access to the Event Hub. Each shared access policy has a name, permissions that you set, and access keys. This option is automatically populated, unless you select the option to provide the Event Hub settings manually.
Event Hub consumer group (recommended)	It is highly recommended to use a distinct consumer group for each Stream Analytics job. This string identifies the consumer group to use to ingest data from the event hub. If no consumer group is specified, the Stream Analytics job uses the \$Default consumer group.
Event serialization format	The serialization format (JSON, CSV, or Avro) of the incoming data stream. Ensure the JSON format aligns with the specification and doesn't include leading 0 for decimal numbers.
Encoding	UTF-8 is currently the only supported encoding format.
Event compression type	The compression type used to read the incoming data stream, such as None (default), GZip, or Deflate.

When your data comes from an Event Hub stream input, you have access to the following metadata fields in your Stream Analytics query:

PROPERTY	DESCRIPTION
EventProcessedUtcTime	The date and time that the event was processed by Stream Analytics.
EventEnqueuedUtcTime	The date and time that the event was received by Event Hubs.
PartitionId	The zero-based partition ID for the input adapter.

For example, using these fields, you can write a query like the following example:

```
SELECT  
    EventProcessedUtcTime,  
    EventEnqueuedUtcTime,  
    PartitionId  
FROM Input
```

NOTE

When using Event Hub as an endpoint for IoT Hub Routes, you can access to the IoT Hub metadata using the [GetMetadataPropertyValue function](#).

Stream data from IoT Hub

Azure IoT Hub is a highly scalable publish-subscribe event ingestor optimized for IoT scenarios.

The default timestamp of events coming from an IoT Hub in Stream Analytics is the timestamp that the event arrived in the IoT Hub, which is `EventEnqueuedUtcTime`. To process the data as a stream using a timestamp in the event payload, you must use the [TIMESTAMP BY keyword](#).

IoT Hub Consumer groups

You should configure each Stream Analytics IoT Hub input to have its own consumer group. When a job contains a self-join or when it has multiple inputs, some input might be read by more than one reader downstream. This situation impacts the number of readers in a single consumer group. To avoid exceeding the Azure IoT Hub limit of five readers per consumer group per partition, it's a best practice to designate a consumer group for each Stream Analytics job.

Configure an IoT Hub as a data stream input

The following table explains each property in the [New input](#) page in the Azure portal when you configure an IoT Hub as a stream input.

PROPERTY	DESCRIPTION
Input alias	A friendly name that you use in the job's query to reference this input.
Subscription	Choose the subscription in which the IoT Hub resource exists.
IoT Hub	The name of the IoT Hub to use as input.
Endpoint	The endpoint for the IoT Hub.
Shared access policy name	The shared access policy that provides access to the IoT Hub. Each shared access policy has a name, permissions that you set, and access keys.
Shared access policy key	The shared access key used to authorize access to the IoT Hub. This option is automatically populated in unless you select the option to provide the IoT Hub settings manually.
Consumer group	It is highly recommended that you use a different consumer group for each Stream Analytics job. The consumer group is used to ingest data from the IoT Hub. Stream Analytics uses the \$Default consumer group unless you specify otherwise.

PROPERTY	DESCRIPTION
Event serialization format	The serialization format (JSON, CSV, or Avro) of the incoming data stream. Ensure the JSON format aligns with the specification and doesn't include leading 0 for decimal numbers.
Encoding	UTF-8 is currently the only supported encoding format.
Event compression type	The compression type used to read the incoming data stream, such as None (default), GZip, or Deflate.

When you use stream data from an IoT Hub, you have access to the following metadata fields in your Stream Analytics query:

PROPERTY	DESCRIPTION
EventProcessedUtcTime	The date and time that the event was processed.
EventEnqueuedUtcTime	The date and time that the event was received by the IoT Hub.
PartitionId	The zero-based partition ID for the input adapter.
IoTHub.MessageId	An ID that's used to correlate two-way communication in IoT Hub.
IoTHub.CorrelationId	An ID that's used in message responses and feedback in IoT Hub.
IoTHub.ConnectionDeviceId	The authentication ID used to send this message. This value is stamped on servicebound messages by the IoT Hub.
IoTHub.ConnectionDeviceGenerationId	The generation ID of the authenticated device that was used to send this message. This value is stamped on servicebound messages by the IoT Hub.
IoTHub.EnqueueTime	The time when the message was received by the IoT Hub.

Stream data from Blob storage

For scenarios with large quantities of unstructured data to store in the cloud, Azure Blob storage offers a cost-effective and scalable solution. Data in Blob storage is usually considered data at rest; however, blob data can be processed as a data stream by Stream Analytics.

Log processing is a commonly used scenario for using Blob storage inputs with Stream Analytics. In this scenario, telemetry data files have been captured from a system and need to be parsed and processed to extract meaningful data.

The default timestamp of Blob storage events in Stream Analytics is the timestamp that the blob was last modified, which is `BlobLastModifiedUtcTime`. To process the data as a stream using a timestamp in the event payload, you must use the [TIMESTAMP BY](#) keyword. A Stream Analytics job pulls data from Azure Blob storage input every second if the blob file is available. If the blob file is unavailable, there is an exponential backoff with a maximum time delay of 90 seconds.

CSV-formatted inputs require a header row to define fields for the data set, and all header row fields must be

unique.

NOTE

Stream Analytics does not support adding content to an existing blob file. Stream Analytics will view each file only once, and any changes that occur in the file after the job has read the data are not processed. Best practice is to upload all the data for a blob file at once and then add additional newer events to a different, new blob file.

Uploading a very large number of blobs at once might cause Stream Analytics to skip reading few blobs in rare cases. It is recommended to upload blobs at least 2 seconds apart to Blob storage. If this option is not feasible, you can use Event Hubs to stream large volumes of events.

Configure Blob storage as a stream input

The following table explains each property in the **New input** page in the Azure portal when you configure Blob storage as a stream input.

PROPERTY	DESCRIPTION
Input alias	A friendly name that you use in the job's query to reference this input.
Subscription	Choose the subscription in which the IoT Hub resource exists.
Storage account	The name of the storage account where the blob files are located.
Storage account key	The secret key associated with the storage account. This option is automatically populated in unless you select the option to provide the Blob storage settings manually.
Container	The container for the blob input. Containers provide a logical grouping for blobs stored in the Microsoft Azure Blob service. When you upload a blob to the Azure Blob storage service, you must specify a container for that blob. You can choose either Use existing container or Create new to have a new container created.
Path pattern (optional)	The file path used to locate the blobs within the specified container. Within the path, you can specify one or more instances of the following three variables: <code>{date}</code> , <code>{time}</code> , or <code>{partition}</code> Example 1: <code>cluster1/logs/{date}/{time}/{partition}</code> Example 2: <code>cluster1/logs/{date}</code> The <code>*</code> character is not an allowed value for the path prefix. Only valid Azure blob characters are allowed. No not include container names or file names.
Date format (optional)	If you use the date variable in the path, the date format in which the files are organized. Example: <code>YYYY/MM/DD</code>
Time format (optional)	If you use the time variable in the path, the time format in which the files are organized. Currently the only supported value is <code>HH</code> for hours.

PROPERTY	DESCRIPTION
Event serialization format	The serialization format (JSON, CSV, or Avro) of the incoming data stream. Ensure the JSON format aligns with the specification and doesn't include leading 0 for decimal numbers.
Encoding	For CSV and JSON, UTF-8 is currently the only supported encoding format.
Compression	The compression type used to read the incoming data stream, such as None (default), GZip, or Deflate.

When your data comes from a Blob storage source, you have access to the following metadata fields in your Stream Analytics query:

PROPERTY	DESCRIPTION
BlobName	The name of the input blob that the event came from.
EventProcessedUtcTime	The date and time that the event was processed by Stream Analytics.
BlobLastModifiedUtcTime	The date and time that the blob was last modified.
PartitionId	The zero-based partition ID for the input adapter.

For example, using these fields, you can write a query like the following example:

```
SELECT
    BlobName,
    EventProcessedUtcTime,
    BlobLastModifiedUtcTime
FROM Input
```

Next steps

[Quickstart: Create a Stream Analytics job by using the Azure portal](#)

Using reference data for lookups in Stream Analytics

6/21/2019 • 9 minutes to read • [Edit Online](#)

Reference data (also known as a lookup table) is a finite data set that is static or slowly changing in nature, used to perform a lookup or to augment your data streams. For example, in an IoT scenario, you could store metadata about sensors (which don't change often) in reference data and join it with real time IoT data streams. Azure Stream Analytics loads reference data in memory to achieve low latency stream processing. To make use of reference data in your Azure Stream Analytics job, you will generally use a [Reference Data Join](#) in your query.

Stream Analytics supports Azure Blob storage and Azure SQL Database as the storage layer for Reference Data. You can also transform and/or copy reference data to Blob storage from Azure Data Factory to use [any number of cloud-based and on-premises data stores](#).

Azure Blob storage

Reference data is modeled as a sequence of blobs (defined in the input configuration) in ascending order of the date/time specified in the blob name. It **only** supports adding to the end of the sequence by using a date/time **greater** than the one specified by the last blob in the sequence.

Configure blob reference data

To configure your reference data, you first need to create an input that is of type **Reference Data**. The table below explains each property that you will need to provide while creating the reference data input with its description:

PROPERTY NAME	DESCRIPTION
Input Alias	A friendly name that will be used in the job query to reference this input.
Storage Account	The name of the storage account where your blobs are located. If it's in the same subscription as your Stream Analytics Job, you can select it from the drop-down.
Storage Account Key	The secret key associated with the storage account. This gets automatically populated if the storage account is in the same subscription as your Stream Analytics job.
Storage Container	Containers provide a logical grouping for blobs stored in the Microsoft Azure Blob service. When you upload a blob to the Blob service, you must specify a container for that blob.
Path Pattern	<p>The path used to locate your blobs within the specified container. Within the path, you may choose to specify one or more instances of the following 2 variables: {date}, {time}</p> <p>Example 1: products/{date}/{time}/product-list.csv Example 2: products/{date}/product-list.csv Example 3: product-list.csv</p> <p>If the blob doesn't exist in the specified path, the Stream Analytics job will wait indefinitely for the blob to become available.</p>

PROPERTY NAME	DESCRIPTION
Date Format [optional]	If you have used {date} within the Path Pattern that you specified, then you can select the date format in which your blobs are organized from the drop-down of supported formats. Example: YYYY/MM/DD, MM/DD/YYYY, etc.
Time Format [optional]	If you have used {time} within the Path Pattern that you specified, then you can select the time format in which your blobs are organized from the drop-down of supported formats. Example: HH, HH/mm, or HH-mm.
Event Serialization Format	To make sure your queries work the way you expect, Stream Analytics needs to know which serialization format you're using for incoming data streams. For Reference Data, the supported formats are CSV and JSON.
Encoding	UTF-8 is the only supported encoding format at this time.

Static reference data

If your reference data is not expected to change, then support for static reference data is enabled by specifying a static path in the input configuration. Azure Stream Analytics picks up the blob from the specified path. {date} and {time} substitution tokens aren't required. Because reference data is immutable in Stream Analytics, overwriting a static reference data blob is not recommended.

Generate reference data on a schedule

If your reference data is a slowly changing data set, then support for refreshing reference data is enabled by specifying a path pattern in the input configuration using the {date} and {time} substitution tokens. Stream Analytics picks up the updated reference data definitions based on this path pattern. For example, a pattern of `sample/{date}/{time}/products.csv` with a date format of "**YYYY-MM-DD**" and a time format of "**HH-mm**" instructs Stream Analytics to pick up the updated blob `sample/2015-04-16/17-30/products.csv` at 5:30 PM on April 16th, 2015 UTC time zone.

Azure Stream Analytics automatically scans for refreshed reference data blobs at a one minute interval. If a blob with timestamp 10:30:00 is uploaded with a small delay (for example, 10:30:30), you will notice a small delay in Stream Analytics job referencing this blob. To avoid such scenarios, it is recommended to upload the blob earlier than the target effective time (10:30:00 in this example) to allow the Stream Analytics job enough time to discover and load it in memory and perform operations.

NOTE

Currently Stream Analytics jobs look for the blob refresh only when the machine time advances to the time encoded in the blob name. For example, the job will look for `sample/2015-04-16/17-30/products.csv` as soon as possible but no earlier than 5:30 PM on April 16th, 2015 UTC time zone. It will *never* look for a blob with an encoded time earlier than the last one that is discovered.

For example, once the job finds the blob `sample/2015-04-16/17-30/products.csv` it will ignore any files with an encoded date earlier than 5:30 PM April 16th, 2015 so if a late arriving `sample/2015-04-16/17-25/products.csv` blob gets created in the same container the job will not use it.

Likewise if `sample/2015-04-16/17-30/products.csv` is only produced at 10:03 PM April 16th, 2015 but no blob with an earlier date is present in the container, the job will use this file starting at 10:03 PM April 16th, 2015 and use the previous reference data until then.

An exception to this is when the job needs to re-process data back in time or when the job is first started. At start time the job is looking for the most recent blob produced before the job start time specified. This is done to ensure that there is a **non-empty** reference data set when the job starts. If one cannot be found, the job displays the following diagnostic:

```
Initializing input without a valid reference data blob for UTC time <start time> .
```

Azure Data Factory can be used to orchestrate the task of creating the updated blobs required by Stream Analytics to update reference data definitions. Data Factory is a cloud-based data integration service that orchestrates and automates the movement and transformation of data. Data Factory supports [connecting to a large number of cloud based and on-premises data stores](#) and moving data easily on a regular schedule that you specify. For more information and step by step guidance on how to set up a Data Factory pipeline to generate reference data for Stream Analytics which refreshes on a pre-defined schedule, check out this [GitHub sample](#).

Tips on refreshing blob reference data

1. Do not overwrite reference data blobs as they are immutable.
2. The recommended way to refresh reference data is to:
 - Use {date}/{time} in the path pattern
 - Add a new blob using the same container and path pattern defined in the job input
 - Use a date/time **greater** than the one specified by the last blob in the sequence.
3. Reference data blobs are **not** ordered by the blob's "Last Modified" time but only by the time and date specified in the blob name using the {date} and {time} substitutions.
4. To avoid having to list large number of blobs, consider deleting very old blobs for which processing will no longer be done. Please note that ASA might go have to reprocess a small amount in some scenarios like a restart.

Azure SQL Database

Azure SQL Database reference data is retrieved by your Stream Analytics job and is stored as a snapshot in memory for processing. The snapshot of your reference data is also stored in a container in a storage account that you specify in the configuration settings. The container is auto-created when the job starts. If the job is stopped or enters a failed state, the auto-created containers are deleted when the job is restarted.

If your reference data is a slowly changing data set, you need to periodically refresh the snapshot that is used in your job. Stream Analytics allows you to set a refresh rate when you configure your Azure SQL Database input connection. The Stream Analytics runtime will query your Azure SQL Database at the interval specified by the refresh rate. The fastest refresh rate supported is once per minute. For each refresh, Stream Analytics stores a new snapshot in the storage account provided.

Stream Analytics provides two options for querying your Azure SQL Database. A snapshot query is mandatory and must be included in each job. Stream Analytics runs the snapshot query periodically based on your refresh

interval and uses the result of the query (the snapshot) as the reference data set. The snapshot query should fit most scenarios, but if you run into performance issues with large data sets and fast refresh rates, you can use the delta query option. Queries that take more than 60 seconds to return reference data set will result in a timeout.

With the delta query option, Stream Analytics runs the snapshot query initially to get a baseline reference data set. After, Stream Analytics runs the delta query periodically based on your refresh interval to retrieve incremental changes. These incremental changes are continually applied to the reference data set to keep it updated. Using delta query may help reduce storage cost and network I/O operations.

Configure SQL Database reference

To configure your SQL Database reference data, you first need to create **Reference Data** input. The table below explains each property that you will need to provide while creating the reference data input with its description. For more information, see [Use reference data from a SQL Database for an Azure Stream Analytics job](#).

PROPERTY NAME	DESCRIPTION
Input alias	A friendly name that will be used in the job query to reference this input.
Subscription	Choose your subscription
Database	The Azure SQL Database that contains your reference data.
Username	The username associated with your Azure SQL Database.
Password	The password associated with your Azure SQL Database.
Refresh periodically	This option allows you to choose a refresh rate. Choosing "On" will allow you to specify the refresh rate in DD:HH:MM.
Snapshot query	This is the default query option that retrieves the reference data from your SQL Database.
Delta query	For advanced scenarios with large data sets and a short refresh rate, choose to add a delta query.

Size limitation

Stream Analytics supports reference data with **maximum size of 300 MB**. The 300 MB limit of maximum size of reference data is achievable only with simple queries. As the complexity of query increases to include stateful processing, such as windowed aggregates, temporal joins and temporal analytic functions, it is expected that the maximum supported size of reference data decreases. If Azure Stream Analytics cannot load the reference data and perform complex operations, the job will run out of memory and fail. In such cases, SU % Utilization metric will reach 100%.

NUMBER OF STREAMING UNITS	APPROX. MAX SIZE SUPPORTED (IN MB)
1	50
3	150
6 and beyond	300

Increasing number of Streaming Units of a job beyond 6 does not increase the maximum supported size of

reference data.

Support for compression is not available for reference data.

Next steps

[Quickstart: Create a Stream Analytics job by using the Azure portal](#)

Understand outputs from Azure Stream Analytics

6/21/2019 • 26 minutes to read • [Edit Online](#)

This article describes the types of outputs available for an Azure Stream Analytics job. Outputs let you store and save the results of the Stream Analytics job. By using the output data, you can do further business analytics and data warehousing of your data.

When you design your Stream Analytics query, refer to the name of the output by using the [INTO clause](#). You can use a single output per job, or multiple outputs per streaming job (if you need them) by providing multiple INTO clauses in the query.

To create, edit, and test Stream Analytics job outputs, you can use the [Azure portal](#), [Azure PowerShell](#), [.NET API](#), [REST API](#), and [Visual Studio](#).

Some outputs types support [partitioning](#). [Output batch sizes](#) vary to optimize throughput.

Azure Data Lake Storage Gen 1

Stream Analytics supports [Azure Data Lake Storage Gen 1](#). Azure Data Lake Storage is an enterprise-wide, hyperscale repository for big data analytic workloads. You can use Data Lake Storage to store data of any size, type, and ingestion speed for operational and exploratory analytics. Stream Analytics needs to be authorized to access Data Lake Storage.

Azure Data Lake Storage output from Stream Analytics is currently not available in the Azure China 21Vianet and Azure Germany (T-Systems International) regions.

The following table lists property names and their descriptions to configure your Data Lake Storage Gen 1 output.

PROPERTY NAME	DESCRIPTION
Output alias	A friendly name used in queries to direct the query output to Data Lake Store.
Subscription	The subscription that contains your Azure Data Lake Storage account.
Account name	The name of the Data Lake Store account where you're sending your output. You're presented with a drop-down list of Data Lake Store accounts that are available in your subscription.

PROPERTY NAME	DESCRIPTION
Path prefix pattern	<p>The file path that's used to write your files within the specified Data Lake Store account. You can specify one or more instances of the {date} and {time} variables:</p> <ul style="list-style-type: none"> • Example 1: folder1/logs/{date}/{time} • Example 2: folder1/logs/{date} <p>The time stamp of the created folder structure follows UTC and not local time.</p> <p>If the file path pattern doesn't contain a trailing slash (/), the last pattern in the file path is treated as a file name prefix.</p> <p>New files are created in these circumstances:</p> <ul style="list-style-type: none"> • Change in output schema • External or internal restart of a job
Date format	Optional. If the date token is used in the prefix path, you can select the date format in which your files are organized. Example: YYYY/MM/DD
Time format	Optional. If the time token is used in the prefix path, specify the time format in which your files are organized. Currently the only supported value is HH.
Event serialization format	The serialization format for output data. JSON, CSV, and Avro are supported.
Encoding	If you're using CSV or JSON format, an encoding must be specified. UTF-8 is the only supported encoding format at this time.
Delimiter	Applicable only for CSV serialization. Stream Analytics supports a number of common delimiters for serializing CSV data. Supported values are comma, semicolon, space, tab, and vertical bar.
Format	Applicable only for JSON serialization. Line separated specifies that the output is formatted by having each JSON object separated by a new line. Array specifies that the output is formatted as an array of JSON objects. This array is closed only when the job stops or Stream Analytics has moved on to the next time window. In general, it's preferable to use line-separated JSON, because it doesn't require any special handling while the output file is still being written to.
Authentication mode	You can authorize access to your Data Lake Storage account using Managed Identity or User token. Once you grant access, you can revoke access by changing the user account password, deleting the Data Lake Storage output for this job, or deleting the Stream Analytics job.

SQL Database

You can use [Azure SQL Database](#) as an output for data that's relational in nature or for applications that depend on content being hosted in a relational database. Stream Analytics jobs write to an existing table in SQL Database.

The table schema must exactly match the fields and their types in your job's output. You can also specify [Azure SQL Data Warehouse](#) as an output via the SQL Database output option. To learn about ways to improve write throughput, see the [Stream Analytics with Azure SQL Database as output](#) article.

The following table lists the property names and their description for creating a SQL Database output.

PROPERTY NAME	DESCRIPTION
Output alias	A friendly name used in queries to direct the query output to this database.
Database	The name of the database where you're sending your output.
Server name	The SQL Database server name.
Username	The username that has write access to the database. Stream Analytics supports only SQL authentication.
Password	The password to connect to the database.
Table	The table name where the output is written. The table name is case-sensitive. The schema of this table should exactly match the number of fields and their types that your job output generates.
Inherit partition scheme	An option for inheriting the partitioning scheme of your previous query step, to enable fully parallel topology with multiple writers to the table. For more information, see Azure Stream Analytics output to Azure SQL Database .
Max batch count	The recommended upper limit on the number of records sent with every bulk insert transaction.

NOTE

The Azure SQL Database offering is supported for a job output in Stream Analytics, but an Azure virtual machine running SQL Server with a database attached or in a SQL Azure Managed Instance is not supported yet. This is subject to change in future releases.

Blob storage

Azure Blob storage offers a cost-effective and scalable solution for storing large amounts of unstructured data in the cloud. For an introduction on Blob storage and its usage, see [Upload, download, and list blobs with the Azure portal](#).

The following table lists the property names and their descriptions for creating a blob output.

PROPERTY NAME	DESCRIPTION
Output alias	A friendly name used in queries to direct the query output to this blob storage.
Storage account	The name of the storage account where you're sending your output.

PROPERTY NAME	DESCRIPTION
Storage account key	The secret key associated with the storage account.
Storage container	A logical grouping for blobs stored in the Azure Blob service. When you upload a blob to the Blob service, you must specify a container for that blob.
Path pattern	<p>Optional. The file path pattern that's used to write your blobs within the specified container.</p> <p>In the path pattern, you can choose to use one or more instances of the date and time variables to specify the frequency that blobs are written: {date}, {time}</p> <p>You can use custom blob partitioning to specify one custom {field} name from your event data to partition blobs. The field name is alphanumeric and can include spaces, hyphens, and underscores. Restrictions on custom fields include the following:</p> <ul style="list-style-type: none"> • Field names aren't case-sensitive. For example, the service can't differentiate between column "ID" and column "id." • Nested fields are not permitted. Instead, use an alias in the job query to "flatten" the field. • Expressions can't be used as a field name. <p>This feature enables the use of custom date/time format specifier configurations in the path. Custom date and time formats must be specified one at a time, enclosed by the {datetime:<specifier>} keyword. Allowable inputs for <specifier> are yyyy, MM, M, dd, d, HH, H, mm, m, ss, or s. The {datetime:<specifier>} keyword can be used multiple times in the path to form custom date/time configurations.</p> <p>Examples:</p> <ul style="list-style-type: none"> • Example 1: cluster1/logs/{date}/{time} • Example 2: cluster1/logs/{date} • Example 3: cluster1/{client_id}/{date}/{time} • Example 4: cluster1/{datetime:ss}/{myField} where the query is: SELECT data.myField AS myField FROM Input; • Example 5: cluster1/year={datetime:yyyy}/month={datetime:MM}/day={datetime:dd} <p>The time stamp of the created folder structure follows UTC and not local time.</p> <p>File naming uses the following convention:</p> <p>{Path Prefix Pattern}/schemaHashcode_Guid_Number.extension</p> <p>Example output files:</p> <ul style="list-style-type: none"> • Myoutput/20170901/00/45434_gguid_1.csv • Myoutput/20170901/01/45434_gguid_1.csv <p>For more information about this feature, see Azure Stream Analytics custom blob output partitioning.</p>

PROPERTY NAME	DESCRIPTION
Date format	Optional. If the date token is used in the prefix path, you can select the date format in which your files are organized. Example: YYYY/MM/DD
Time format	Optional. If the time token is used in the prefix path, specify the time format in which your files are organized. Currently the only supported value is HH.
Event serialization format	Serialization format for output data. JSON, CSV, and Avro are supported.
Encoding	If you're using CSV or JSON format, an encoding must be specified. UTF-8 is the only supported encoding format at this time.
Delimiter	Applicable only for CSV serialization. Stream Analytics supports a number of common delimiters for serializing CSV data. Supported values are comma, semicolon, space, tab, and vertical bar.
Format	Applicable only for JSON serialization. Line separated specifies that the output is formatted by having each JSON object separated by a new line. Array specifies that the output is formatted as an array of JSON objects. This array is closed only when the job stops or Stream Analytics has moved on to the next time window. In general, it's preferable to use line-separated JSON, because it doesn't require any special handling while the output file is still being written to.

When you're using Blob storage as output, a new file is created in the blob in the following cases:

- If the file exceeds the maximum number of allowed blocks (currently 50,000). You might reach the maximum allowed number of blocks without reaching the maximum allowed blob size. For example, if the output rate is high, you can see more bytes per block, and the file size is larger. If the output rate is low, each block has less data, and the file size is smaller.
- If there's a schema change in the output, and the output format requires fixed schema (CSV and Avro).
- If a job is restarted, either externally by a user stopping it and starting it, or internally for system maintenance or error recovery.
- If the query is fully partitioned, and a new file is created for each output partition.
- If the user deletes a file or a container of the storage account.
- If the output is time partitioned by using the path prefix pattern, and a new blob is used when the query moves to the next hour.
- If the output is partitioned by a custom field, and a new blob is created per partition key if it does not exist.
- If the output is partitioned by a custom field where the partition key cardinality exceeds 8,000, and a new blob is created per partition key.

Event Hubs

The [Azure Event Hubs](#) service is a highly scalable publish-subscribe event ingestor. It can collect millions of events per second. One use of an event hub as output is when the output of a Stream Analytics job becomes the input of another streaming job.

You need a few parameters to configure data streams from event hubs as an output.

PROPERTY NAME	DESCRIPTION
Output alias	A friendly name used in queries to direct the query output to this event hub.
Event hub namespace	A container for a set of messaging entities. When you created a new event hub, you also created an event hub namespace.
Event hub name	The name of your event hub output.
Event hub policy name	The shared access policy, which you can create on the event hub's Configure tab. Each shared access policy has a name, permissions that you set, and access keys.
Event hub policy key	The shared access key that's used to authenticate access to the event hub namespace.
Partition key column	Optional. A column that contains the partition key for event hub output.
Event serialization format	The serialization format for output data. JSON, CSV, and Avro are supported.
Encoding	For CSV and JSON, UTF-8 is the only supported encoding format at this time.
Delimiter	Applicable only for CSV serialization. Stream Analytics supports a number of common delimiters for serializing data in CSV format. Supported values are comma, semicolon, space, tab, and vertical bar.
Format	Applicable only for JSON serialization. Line separated specifies that the output is formatted by having each JSON object separated by a new line. Array specifies that the output is formatted as an array of JSON objects. This array is closed only when the job stops or Stream Analytics has moved on to the next time window. In general, it's preferable to use line-separated JSON, because it doesn't require any special handling while the output file is still being written to.
Property columns	Optional. Comma-separated columns that need to be attached as user properties of the outgoing message instead of the payload. More info about this feature is in the section Custom metadata properties for output .

Power BI

You can use [Power BI](#) as an output for a Stream Analytics job to provide for a rich visualization experience of analysis results. You can use this capability for operational dashboards, report generation, and metric-driven reporting.

Power BI output from Stream Analytics is currently not available in the Azure China 21Vianet and Azure Germany (T-Systems International) regions.

The following table lists property names and their descriptions to configure your Power BI output.

PROPERTY NAME	DESCRIPTION
Output alias	Provide a friendly name that's used in queries to direct the query output to this Power BI output.
Group workspace	To enable sharing data with other Power BI users, you can select groups inside your Power BI account or choose My Workspace if you don't want to write to a group. Updating an existing group requires renewing the Power BI authentication.
Dataset name	Provide a dataset name that you want the Power BI output to use.
Table name	Provide a table name under the dataset of the Power BI output. Currently, Power BI output from Stream Analytics jobs can have only one table in a dataset.
Authorize connection	You need to authorize with Power BI to configure your output settings. Once you grant this output access to your Power BI dashboard, you can revoke access by changing the user account password, deleting the job output, or deleting the Stream Analytics job.

For a walkthrough of configuring a Power BI output and dashboard, see the [Azure Stream Analytics and Power BI tutorial](#).

NOTE

Don't explicitly create the dataset and table in the Power BI dashboard. The dataset and table are automatically populated when the job is started and the job starts pumping output into Power BI. If the job query doesn't generate any results, the dataset and table aren't created. If Power BI already had a dataset and table with the same name as the one provided in this Stream Analytics job, the existing data is overwritten.

Create a schema

Azure Stream Analytics creates a Power BI dataset and table schema for the user if they don't already exist. In all other cases, the table is updated with new values. Currently, only one table can exist within a dataset.

Power BI uses the first-in, first-out (FIFO) retention policy. Data will collect in a table until it hits 200,000 rows.

Convert a data type from Stream Analytics to Power BI

Azure Stream Analytics updates the data model dynamically at runtime if the output schema changes. Column name changes, column type changes, and the addition or removal of columns are all tracked.

This table covers the data type conversions from [Stream Analytics data types](#) to Power BI [Entity Data Model \(EDM\) types](#), if a Power BI dataset and table don't exist.

FROM STREAM ANALYTICS	TO POWER BI
bigint	Int64
nvarchar(max)	String
datetime	Datetime

FROM STREAM ANALYTICS	TO POWER BI
float	Double
Record array	String type, constant value "IRecord" or "IArray"

Update the schema

Stream Analytics infers the data model schema based on the first set of events in the output. Later, if necessary, the data model schema is updated to accommodate incoming events that might not fit into the original schema.

Avoid the `SELECT *` query to prevent dynamic schema update across rows. In addition to potential performance implications, it might result in uncertainty of the time taken for the results. Select the exact fields that need to be shown on the Power BI dashboard. Additionally, the data values should be compliant with the chosen data type.

PREVIOUS/CURRENT	INT64	STRING	DATETIME	DOUBLE
Int64	Int64	String	String	Double
Double	Double	String	String	Double
String	String	String	String	String
Datetime	String	String	Datetime	String

Table storage

[Azure Table storage](#) offers highly available, massively scalable storage, so that an application can automatically scale to meet user demand. Table storage is Microsoft's NoSQL key/attribute store, which you can use for structured data with fewer constraints on the schema. Azure Table storage can be used to store data for persistence and efficient retrieval.

The following table lists the property names and their descriptions for creating a table output.

PROPERTY NAME	DESCRIPTION
Output alias	A friendly name used in queries to direct the query output to this table storage.
Storage account	The name of the storage account where you're sending your output.
Storage account key	The access key associated with the storage account.
Table name	The name of the table. The table gets created if it doesn't exist.
Partition key	The name of the output column that contains the partition key. The partition key is a unique identifier for the partition within a table that forms the first part of an entity's primary key. It's a string value that can be up to 1 KB in size.

PROPERTY NAME	DESCRIPTION
Row key	The name of the output column that contains the row key. The row key is a unique identifier for an entity within a partition. It forms the second part of an entity's primary key. The row key is a string value that can be up to 1 KB in size.
Batch size	The number of records for a batch operation. The default (100) is sufficient for most jobs. See the Table Batch Operation spec for more details on modifying this setting.

Service Bus queues

[Service Bus queues](#) offer a FIFO message delivery to one or more competing consumers. Typically, messages are received and processed by the receivers in the temporal order in which they were added to the queue. Each message is received and processed by only one message consumer.

The following table lists the property names and their descriptions for creating a queue output.

PROPERTY NAME	DESCRIPTION
Output alias	A friendly name used in queries to direct the query output to this Service Bus queue.
Service Bus namespace	A container for a set of messaging entities.
Queue name	The name of the Service Bus queue.
Queue policy name	When you create a queue, you can also create shared access policies on the queue's Configure tab. Each shared access policy has a name, permissions that you set, and access keys.
Queue policy key	The shared access key that's used to authenticate access to the Service Bus namespace.
Event serialization format	The serialization format for output data. JSON, CSV, and Avro are supported.
Encoding	For CSV and JSON, UTF-8 is the only supported encoding format at this time.
Delimiter	Applicable only for CSV serialization. Stream Analytics supports a number of common delimiters for serializing data in CSV format. Supported values are comma, semicolon, space, tab, and vertical bar.
Format	Applicable only for JSON type. Line separated specifies that the output is formatted by having each JSON object separated by a new line. Array specifies that the output is formatted as an array of JSON objects.
Property columns	Optional. Comma-separated columns that need to be attached as user properties of the outgoing message instead of the payload. More info about this feature is in the section Custom metadata properties for output .

The number of partitions is [based on the Service Bus SKU and size](#). Partition key is a unique integer value for each partition.

Service Bus Topics

Service Bus queues provide a one-to-one communication method from sender to receiver. [Service Bus topics](#) provide a one-to-many form of communication.

The following table lists the property names and their descriptions for creating a Service Bus topic output.

PROPERTY NAME	DESCRIPTION
Output alias	A friendly name used in queries to direct the query output to this Service Bus topic.
Service Bus namespace	A container for a set of messaging entities. When you created a new event hub, you also created a Service Bus namespace.
Topic name	Topics are messaging entities, similar to event hubs and queues. They're designed to collect event streams from devices and services. When a topic is created, it's also given a specific name. The messages sent to a topic aren't available unless a subscription is created, so ensure there's one or more subscriptions under the topic.
Topic policy name	When you create a Service Bus topic, you can also create shared access policies on the topic's Configure tab. Each shared access policy has a name, permissions that you set, and access keys.
Topic policy key	The shared access key that's used to authenticate access to the Service Bus namespace.
Event serialization format	The serialization format for output data. JSON, CSV, and Avro are supported.
Encoding	If you're using CSV or JSON format, an encoding must be specified. UTF-8 is the only supported encoding format at this time.
Delimiter	Applicable only for CSV serialization. Stream Analytics supports a number of common delimiters for serializing data in CSV format. Supported values are comma, semicolon, space, tab, and vertical bar.
Property columns	Optional. Comma-separated columns that need to be attached as user properties of the outgoing message instead of the payload. More info about this feature is in the section Custom metadata properties for output .

The number of partitions is [based on the Service Bus SKU and size](#). The partition key is a unique integer value for each partition.

Azure Cosmos DB

[Azure Cosmos DB](#) is a globally distributed database service that offers limitless elastic scale around the globe, rich query, and automatic indexing over schema-agnostic data models. To learn about Azure Cosmos DB collection

options for Stream Analytics, see the [Stream Analytics with Azure Cosmos DB as output](#) article.

Azure Cosmos DB output from Stream Analytics is currently not available in the Azure China 21Vianet and Azure Germany (T-Systems International) regions.

NOTE

At this time, Azure Stream Analytics only supports connection to Azure Cosmos DB by using the SQL API. Other Azure Cosmos DB APIs are not yet supported. If you point Azure Stream Analytics to the Azure Cosmos DB accounts created with other APIs, the data might not be properly stored.

The following table describes the properties for creating an Azure Cosmos DB output.

PROPERTY NAME	DESCRIPTION
Output alias	An alias to refer this output in your Stream Analytics query.
Sink	Azure Cosmos DB.
Import option	Choose either Select Cosmos DB from your subscription or Provide Cosmos DB settings manually .
Account ID	The name or endpoint URI of the Azure Cosmos DB account.
Account key	The shared access key for the Azure Cosmos DB account.
Database	The Azure Cosmos DB database name.
Collection name	Name of the collection in Azure Cosmos DB. Azure Cosmos DB unlimited containers are the recommended approach for partitioning your data, as Azure Cosmos DB automatically scales partitions based on your workload.
Document ID	Optional. The name of the field in output events that's used to specify the primary key on which insert or update operations are based.

Azure Functions

Azure Functions is a serverless compute service that you can use to run code on-demand without having to explicitly provision or manage infrastructure. It lets you implement code that's triggered by events occurring in Azure or partner services. This ability of Azure Functions to respond to triggers makes it a natural output for Azure Stream Analytics. This output adapter enables users to connect Stream Analytics to Azure Functions, and run a script or piece of code in response to a variety of events.

Azure Functions output from Stream Analytics is currently not available in the Azure China 21Vianet and Azure Germany (T-Systems International) regions.

Azure Stream Analytics invokes Azure Functions via HTTP triggers. The Azure Functions output adapter is available with the following configurable properties:

PROPERTY NAME	DESCRIPTION
Function app	The name of your Azure Functions app.

PROPERTY NAME	DESCRIPTION
Function	The name of the function in your Azure Functions app.
Key	If you want to use an Azure Function from another subscription, you can do so by providing the key to access your function.
Max batch size	A property that lets you set the maximum size for each output batch that's sent to your Azure function. The input unit is in bytes. By default, this value is 262,144 bytes (256 KB).
Max batch count	A property that lets you specify the maximum number of events in each batch that's sent to Azure Functions. The default value is 100.

When Azure Stream Analytics receives a 413 ("http Request Entity Too Large") exception from an Azure function, it reduces the size of the batches that it sends to Azure Functions. In your Azure function code, use this exception to make sure that Azure Stream Analytics doesn't send oversized batches. Also, make sure that the maximum batch count and size values used in the function are consistent with the values entered in the Stream Analytics portal.

Also, in a situation where there's no event landing in a time window, no output is generated. As a result, the **computeResult** function isn't called. This behavior is consistent with the built-in windowed aggregate functions.

Custom metadata properties for output

You can attach query columns as user properties to your outgoing messages. These columns don't go into the payload. The properties are present in the form of a dictionary on the output message. *Key* is the column name and *value* is the column value in the properties dictionary. All Stream Analytics data types are supported except Record and Array.

Supported outputs:

- Service Bus queue
- Service Bus topic
- Event hub

In the following example, we add the two fields `DeviceId` and `DeviceStatus` to the metadata.

- Query: `select *, DeviceId, DeviceStatus from iotHubInput`
- Output configuration: `DeviceId,DeviceStatus`

Property columns ⓘ
 ✓

The following screenshot shows output message properties inspected in EventHub through [Service Bus Explorer](#).

Event Custom Properties	
Name	Value
DeviceId	Device25997937-399e-48e5-9ba5-59257ee6dc57
DeviceStatus	Healthy

Partitioning

The following table summarizes the partition support and the number of output writers for each output type:

OUTPUT TYPE	PARTITIONING SUPPORT	PARTITION KEY	NUMBER OF OUTPUT WRITERS
Azure Data Lake Store	Yes	Use {date} and {time} tokens in the path prefix pattern. Choose the date format, such as YYYY/MM/DD, DD/MM/YYYY, or MM-DD-YYYY. HH is used for the time format.	Follows the input partitioning for fully parallelizable queries .
Azure SQL Database	Yes, needs to enabled.	Based on the PARTITION BY clause in the query.	When Inherit Partitioning option is enabled, follows the input partitioning for fully parallelizable queries . To learn more about achieving better write throughput performance when you're loading data into Azure SQL Database, see Azure Stream Analytics output to Azure SQL Database .
Azure Blob storage	Yes	Use {date} and {time} tokens from your event fields in the path pattern. Choose the date format, such as YYYY/MM/DD, DD/MM/YYYY, or MM-DD-YYYY. HH is used for the time format. Blob output can be partitioned by a single custom event attribute {fieldname} or {datetime:<specifier>}.	Follows the input partitioning for fully parallelizable queries .

OUTPUT TYPE	PARTITIONING SUPPORT	PARTITION KEY	NUMBER OF OUTPUT WRITERS
Azure Event Hubs	Yes	Yes	<p>Varies depending on partition alignment.</p> <p>When the partition key for event hub output is equally aligned with the upstream (previous) query step, the number of writers is the same as the number of partitions in event hub output. Each writer uses the EventHubSender class to send events to the specific partition.</p> <p>When the partition key for event hub output is not aligned with the upstream (previous) query step, the number of writers is the same as the number of partitions in that prior step. Each writer uses the SendBatchAsync class in EventHubClient to send events to all the output partitions.</p>
Power BI	No	None	Not applicable.
Azure Table storage	Yes	Any output column.	Follows the input partitioning for fully parallelized queries .
Azure Service Bus topic	Yes	Automatically chosen. The number of partitions is based on the Service Bus SKU and size . The partition key is a unique integer value for each partition.	Same as the number of partitions in the output topic.
Azure Service Bus queue	Yes	Automatically chosen. The number of partitions is based on the Service Bus SKU and size . The partition key is a unique integer value for each partition.	Same as the number of partitions in the output queue.
Azure Cosmos DB	Yes	Use the {partition} token in the collection name pattern. The {partition} value is based on the PARTITION BY clause in the query.	Follows the input partitioning for fully parallelized queries .
Azure Functions	No	None	Not applicable.

If your output adapter is not partitioned, lack of data in one input partition will cause a delay up to the late arrival amount of time. In such cases, the output is merged to a single writer, which might cause bottlenecks in your pipeline. To learn more about late arrival policy, see [Azure Stream Analytics event order considerations](#).

Output batch size

Azure Stream Analytics uses variable-size batches to process events and write to outputs. Typically the Stream Analytics engine doesn't write one message at a time, and uses batches for efficiency. When the rate of both the incoming and outgoing events is high, Stream Analytics uses larger batches. When the egress rate is low, it uses smaller batches to keep latency low.

The following table explains some of the considerations for output batching:

OUTPUT TYPE	MAX MESSAGE SIZE	BATCH SIZE OPTIMIZATION
Azure Data Lake Store	See Data Lake Storage limits .	Use up to 4 MB per write operation.
Azure SQL Database	Configurable using Max batch count. 10,000 maximum and 100 minimum rows per single bulk insert by default. See Azure SQL limits .	Every batch is initially bulk inserted with maximum batch count. Batch is split in half (until minimum batch count) based on retryable errors from SQL.
Azure Blob storage	See Azure Storage limits .	The maximum blob block size is 4 MB. The maximum blob block count is 50,000.
Azure Event Hubs	256 KB or 1 MB per message. See Event Hubs limits .	When input/output partitioning isn't aligned, each event is packed individually in <code>EventData</code> and sent in a batch of up to the maximum message size. This also happens if custom metadata properties are used. When input/output partitioning is aligned, multiple events are packed into a single <code>EventData</code> instance, up to the maximum message size, and sent.
Power BI	See Power BI Rest API limits .	
Azure Table storage	See Azure Storage limits .	The default is 100 entities per single transaction. You can configure it to a smaller value as needed.
Azure Service Bus queue	256 KB per message for Standard tier, 1MB for Premium tier. See Service Bus limits .	Use a single event per message.
Azure Service Bus topic	256 KB per message for Standard tier, 1MB for Premium tier. See Service Bus limits .	Use a single event per message.
Azure Cosmos DB	See Azure Cosmos DB limits .	Batch size and write frequency are adjusted dynamically based on Azure Cosmos DB responses. There are no predetermined limitations from Stream Analytics.

OUTPUT TYPE	MAX MESSAGE SIZE	BATCH SIZE OPTIMIZATION
Azure Functions		<p>The default batch size is 262,144 bytes (256 KB).</p> <p>The default event count per batch is 100.</p> <p>The batch size is configurable and can be increased or decreased in the Stream Analytics output options.</p>

Next steps

[Quickstart: Create a Stream Analytics job by using the Azure portal](#)

Azure Stream Analytics output to Azure Cosmos DB

4/9/2019 • 6 minutes to read • [Edit Online](#)

Stream Analytics can target [Azure Cosmos DB](#) for JSON output, enabling data archiving and low-latency queries on unstructured JSON data. This document covers some best practices for implementing this configuration.

For those who are unfamiliar with Cosmos DB, take a look at [Azure Cosmos DB's learning path](#) to get started.

NOTE

At this time, Azure Stream Analytics only supports connection to Azure Cosmos DB using [SQL API](#). Other Azure Cosmos DB APIs are not yet supported. If you point Azure Stream Analytics to the Azure Cosmos DB accounts created with other APIs, the data might not be properly stored.

Basics of Cosmos DB as an output target

The Azure Cosmos DB output in Stream Analytics enables writing your stream processing results as JSON output into your Cosmos DB collection(s). Stream Analytics doesn't create collections in your database, instead requiring you to create them upfront. This is so that the billing costs of Cosmos DB collections are controlled by you, and so that you can tune the performance, consistency, and capacity of your collections directly using the [Cosmos DB APIs](#).

NOTE

You must add 0.0.0.0 to the list of allowed IPs from your Azure Cosmos DB firewall.

Some of the Cosmos DB collection options are detailed below.

Tune consistency, availability, and latency

To match your application requirements, Azure Cosmos DB allows you to fine-tune the database and collections and make trade-offs between consistency, availability, and latency. Depending on what levels of read consistency your scenario needs against read and write latency, you can choose a consistency level on your database account. Also by default, Azure Cosmos DB enables synchronous indexing on each CRUD operation to your collection. This is another useful option to control the write/read performance in Azure Cosmos DB. For more information, review the [change your database and query consistency levels](#) article.

Upserts from Stream Analytics

Stream Analytics integration with Azure Cosmos DB allows you to insert or update records in your collection based on a given Document ID column. This is also referred to as an *Upsert*.

Stream Analytics uses an optimistic upsert approach, where updates are only done when insert fails with a Document ID conflict. With Compatibility Level 1.0, this update is performed as a PATCH, so it enables partial updates to the document, that is, addition of new properties or replacing an existing property is performed incrementally. However, changes in the values of array properties in your JSON document result in the entire array getting overwritten, that is, the array isn't merged. With 1.2, upsert behavior is modified to insert or replace the document. This is described further in the Compatibility Level 1.2 section below.

If the incoming JSON document has an existing ID field, that field is automatically used as the Document ID

column in Cosmos DB and any subsequent writes are handled as such, leading to one of these situations:

- unique IDs lead to insert
- duplicate IDs and 'Document ID' set to 'ID' leads to upsert
- duplicate IDs and 'Document ID' not set leads to error, after the first document

If you want to save *all* documents including the ones with a duplicate ID, rename the ID field in your query (with the AS keyword) and let Cosmos DB create the ID field or replace the ID with another column's value (using the AS keyword or by using the 'Document ID' setting).

Data partitioning in Cosmos DB

Azure Cosmos DB [unlimited](#) containers are the recommended approach for partitioning your data, as Azure Cosmos DB automatically scales partitions based on your workload. When writing to unlimited containers, Stream Analytics uses as many parallel writers as the previous query step or input partitioning scheme.

NOTE

At this time, Azure Stream Analytics only supports unlimited collections with partition keys at the top level. For example, `/region` is supported. Nested partition keys (e.g. `/region/name`) are not supported.

For fixed Azure Cosmos DB collections, Stream Analytics allows no way to scale up or out once they're full. They have an upper limit of 10 GB and 10,000 RU/s throughput. To migrate the data from a fixed container to an unlimited container (for example, one with at least 1,000 RU/s and a partition key), you need to use the [data migration tool](#) or the [change feed library](#).

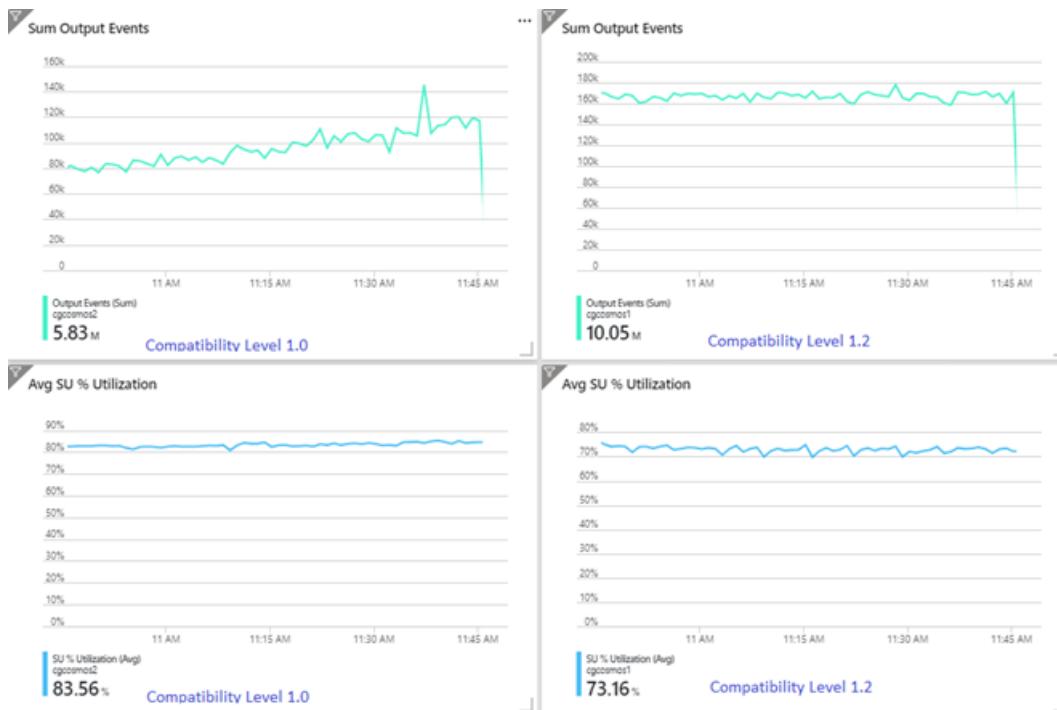
Writing to multiple fixed containers is being deprecated and is not the recommended approach for scaling out your Stream Analytics job. The article [Partitioning and scaling in Cosmos DB](#) provides further details.

Improved throughput with Compatibility Level 1.2

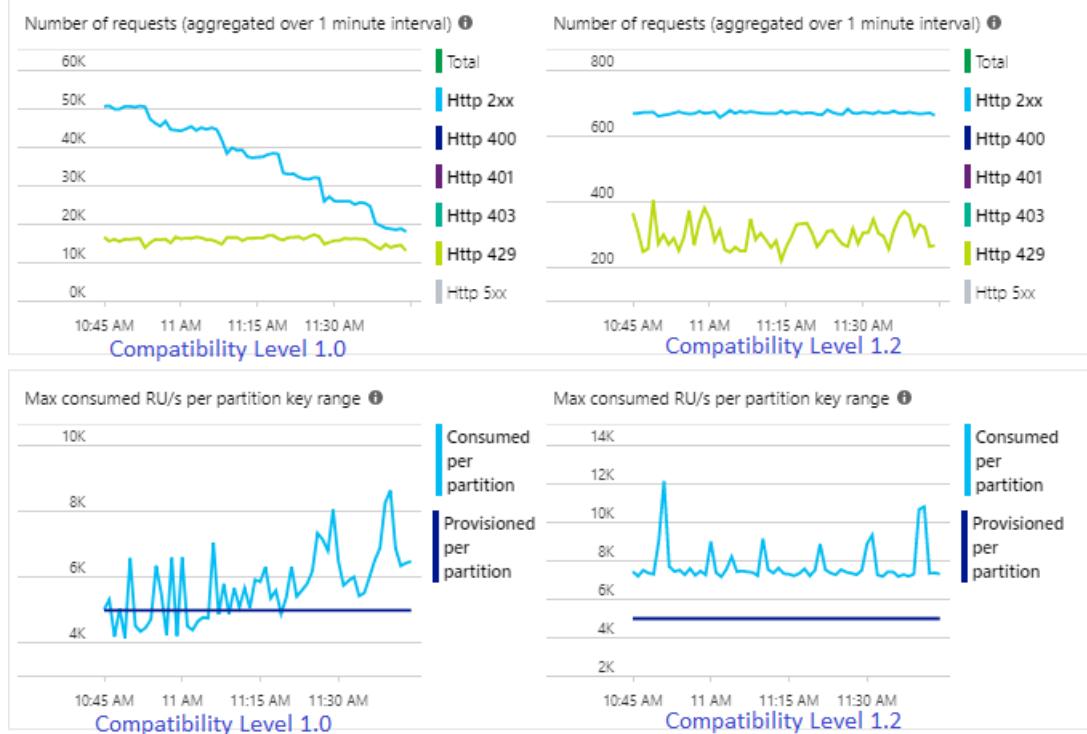
With Compatibility level 1.2, Stream Analytics supports native integration to bulk write into Cosmos DB. This enables writing effectively to Cosmos DB with maximizing throughput and efficiently handle throttling requests. The improved writing mechanism is available under a new compatibility level due to an upsert behavior difference. Prior to 1.2, the upsert behavior is to insert or merge the document. With 1.2, upserts behavior is modified to insert or replace the document.

Before 1.2, uses a custom stored procedure to bulk upsert documents per partition key into Cosmos DB, where a batch is written as a transaction. Even when a single record hits a transient error (throttling), the whole batch must be retried. This made scenarios with even reasonable throttling relatively slower. Following comparison shows how such jobs would behave with 1.2.

Below setup shows two identical Stream Analytics jobs reading from same input (event hub). Both Stream Analytics jobs are [fully partitioned](#) with a passthrough query and writing to identical CosmosDB collections. Metrics on the left is from the job configured with compatibility level 1.0 and the ones on the right is configured with 1.2. Cosmos DB collections partition key is a unique guid coming from the input event.



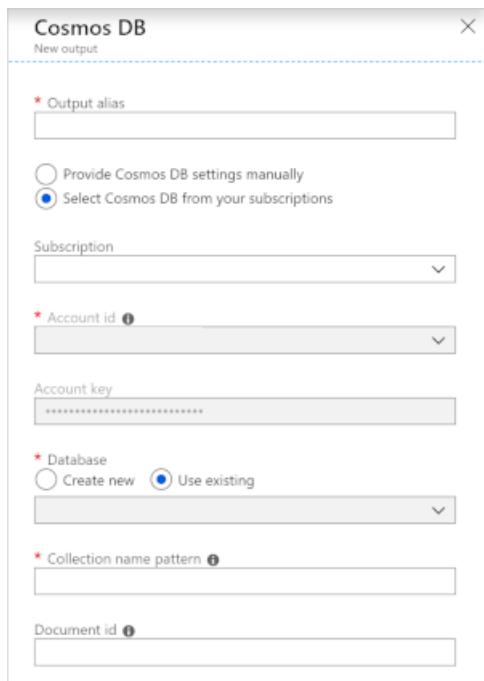
Incoming event rate in Event Hub is 2x higher than Cosmos DB collections (20K RUs) are configured to intake, so throttling is expected in Cosmos DB. However, the job with 1.2, is consistently writing at a higher throughput (Output Events/minute) and with a lower average SU% utilization. In your environment, this difference will depend on few more factors such as choice of event format, input event/message size, partition keys, query etc.



With 1.2, Stream Analytics is more intelligent in utilizing 100% of the available throughput in Cosmos DB with very few resubmissions from throttling/rate limiting. This provides a better experience for other workloads like queries running on the collection at the same time. In case you need to try out how ASA scales out with Cosmos DB as a sink for 1k to 10k messages/second, here is an [azure samples project](#) that lets you do that. Please note that Cosmos DB output throughput is identical with 1.0 and 1.1. Since 1.2 is currently not the default, you can [set compatibility level](#) for a Stream Analytics job by using portal or by using the [create job REST API call](#). It's *strongly recommended* to use Compatibility Level 1.2 in ASA with Cosmos DB.

Cosmos DB settings for JSON output

Creating Cosmos DB as an output in Stream Analytics generates a prompt for information as seen below. This section provides an explanation of the properties definition.



FIELD	DESCRIPTION
Output alias	An alias to refer this output in your ASA query.
Subscription	Choose the Azure subscription.
Account ID	The name or endpoint URI of the Azure Cosmos DB account.
Account key	The shared access key for the Azure Cosmos DB account.
Database	The Azure Cosmos DB database name.
Collection name pattern	The collection name for the collection to be used. MyCollection is a sample valid input - one collection named MyCollection must exist.
Document ID	Optional. The column name in output events used as the unique key on which insert or update operations must be based. If left empty, all events will be inserted, with no update option.

Azure Stream Analytics output to Azure SQL Database

6/23/2019 • 4 minutes to read • [Edit Online](#)

This article discusses tips to achieve better write throughput performance when you're loading data into SQL Azure Database using Azure Stream Analytics.

SQL output in Azure Stream Analytics supports writing in parallel as an option. This option allows for [fully parallel](#) job topologies, where multiple output partitions are writing to the destination table in parallel. Enabling this option in Azure Stream Analytics however may not be sufficient to achieve higher throughputs, as it depends significantly on your SQL Azure database configuration and table schema. The choice of indexes, clustering key, index fill factor, and compression have an impact on the time to load tables. For more information about how to optimize your SQL Azure database to improve query and load performance based on internal benchmarks, see [SQL database performance guidance](#). Ordering of writes is not guaranteed when writing in parallel to SQL Azure Database.

Here are some configurations within each service that can help improve overall throughput of your solution.

Azure Stream Analytics

- **Inherit Partitioning** – This SQL output configuration option enables inheriting the partitioning scheme of your previous query step or input. With this enabled, writing to a disk-based table and having a [fully parallel](#) topology for your job, expect to see better throughputs. This partitioning already automatically happens for many other [outputs](#). Table locking (TABLOCK) is also disabled for bulk inserts made with this option.

NOTE

When there are more than 8 input partitions, inheriting the input partitioning scheme might not be an appropriate choice. This upper limit was observed on a table with a single identity column and a clustered index. In this case, consider using `INTO 8` in your query, to explicitly specify the number of output writers. Based on your schema and choice of indexes, your observations may vary.

- **Batch Size** - SQL output configuration allows you to specify the maximum batch size in an Azure Stream Analytics SQL output based on the nature of your destination table/workload. Batch size is the maximum number of records that sent with every bulk insert transaction. In clustered columnstore indexes, batch sizes around [100K](#) allow for more parallelization, minimal logging, and locking optimizations. In disk-based tables, 10K (default) or lower may be optimal for your solution, as higher batch sizes may trigger lock escalation during bulk inserts.
- **Input Message Tuning** – If you've optimized using inherit partitioning and batch size, increasing the number of input events per message per partition helps further pushing up your write throughput. Input message tuning allows batch sizes within Azure Stream Analytics to be up to the specified Batch Size, thereby improving throughput. This can be achieved by using [compression](#) or increasing input message sizes in EventHub or Blob.

SQL Azure

- **Partitioned Table and Indexes** – Using a [partitioned](#) SQL table and partitioned indexes on the table with the same column as your partition key (for example, PartitionId) can significantly reduce contentions

among partitions during writes. For a partitioned table, you'll need to create a [partition function](#) and a [partition scheme](#) on the PRIMARY filegroup. This will also increase availability of existing data while new data is being loaded. Log IO limit may be hit based on number of partitions, which can be increased by upgrading the SKU.

- **Avoid unique key violations** – If you get [multiple key violation warning messages](#) in the Azure Stream Analytics Activity Log, ensure your job isn't impacted by unique constraint violations which are likely to happen during recovery cases. This can be avoided by setting the [IGNORE_DUP_KEY](#) option on your indexes.

Azure Data Factory and In-Memory Tables

- **In-Memory Table as temp table** – [In-Memory tables](#) allow for very high-speed data loads but data needs to fit in memory. Benchmarks show bulk loading from an in-memory table to a disk-based table is about 10 times faster than directly bulk inserting using a single writer into the disk-based table with an identity column and a clustered index. To leverage this bulk insert performance, set up a [copy job using Azure Data Factory](#) that copies data from the in-memory table to the disk-based table.

Avoiding Performance Pitfalls

Bulk inserting data is much faster than loading data with single inserts because the repeated overhead of transferring the data, parsing the insert statement, running the statement, and issuing a transaction record is avoided. Instead, a more efficient path is used into the storage engine to stream the data. The setup cost of this path is however much higher than a single insert statement in a disk-based table. The break-even point is typically around 100 rows, beyond which bulk loading is almost always more efficient.

If the incoming events rate is low, it can easily create batch sizes lower than 100 rows, which makes bulk insert inefficient and uses too much disk space. To work around this limitation, you can do one of these actions:

- Create an INSTEAD OF [trigger](#) to use simple insert for every row.
- Use an In-Memory temp table as described in the previous section.

Another such scenario occurs when writing into a non-clustered columnstore index (NCCI), where smaller bulk inserts can create too many segments, that can crash the index. In this case, the recommendation is to use a Clustered Columnstore index instead.

Summary

In summary, with the partitioned output feature in Azure Stream Analytics for SQL output, aligned parallelization of your job with a partitioned table in SQL Azure should give you significant throughput improvements.

Leveraging Azure Data Factory for orchestrating data movement from an In-Memory table into Disk-based tables can give order of magnitude throughput gains. If feasible, improving message density can also be a major factor in improving overall throughput.

Azure Stream Analytics custom blob output partitioning

5/15/2019 • 4 minutes to read • [Edit Online](#)

Azure Stream Analytics supports custom blob output partitioning with custom fields or attributes and custom DateTime path patterns.

Custom field or attributes

Custom field or input attributes improve downstream data-processing and reporting workflows by allowing more control over the output.

Partition key options

The partition key, or column name, used to partition input data may contain alphanumeric characters with hyphens, underscores, and spaces. It is not possible to use nested fields as a partition key unless used in conjunction with aliases. The partition key must be NVARCHAR(MAX).

Example

Suppose a job takes input data from live user sessions connected to an external video game service where ingested data contains a column **client_id** to identify the sessions. To partition the data by **client_id**, set the Blob Path Pattern field to include a partition token **{client_id}** in blob output properties when creating a job. As data with various **client_id** values flow through the Stream Analytics job, the output data is saved into separate folders based on a single **client_id** value per folder.



Similarly, if the job input was sensor data from millions of sensors where each sensor had a **sensor_id**, the Path Pattern would be **{sensor_id}** to partition each sensor data to different folders.

Using the REST API, the output section of a JSON file used for that request may look like the following:

```
{
  "outputs": [
    {
      "properties": {
        "datasource": {
          "type": "Microsoft.Storage/Blob",
          "properties": {
            "storageAccounts": [
              {
                "accountName": "<account-name>",
                "accountKey": "<account-key>"
              }
            ],
            "container": "clients",
            "pathPattern": "logs/{client_id}"
          }
        },
        "serialization": {
          "type": "Json",
          "properties": {
            "encoding": "UTF8",
            "format": "LineSeparated"
          }
        }
      },
      "name": "blob-output",
      "type": "Microsoft.StreamAnalytics/streamingjobs/outputs"
    }
  ]
}
```

Once the job starts running, the *clients* container may look like the following:

NAME	MODIFIED
0600000	
0600001	
0600002	
0600031	
0600032	
0600041	
0600042	
0640820	
0640821	
0640941	

Each folder may contain multiple blobs where each blob contains one or more records. In the above example, there is a single blob in a folder labelled "06000000" with the following contents:

```
[  
  {  
    "client_id": "0600000",  
    "name": "John",  
    "email": "johndoh@example.com",  
    "message": "message1",  
    "PartitionId": 0,  
    "EventEnqueuedUtcTime": "2018-04-03T23:28:05.8041216Z",  
    "EventProcessedUtcTime": "2018-04-03T23:28:05.8121147Z"  
  },  
  {  
    "client_id": "0600000",  
    "name": "John",  
    "email": "johndoh@example.com",  
    "message": "message2",  
    "PartitionId": 0,  
    "EventEnqueuedUtcTime": "2018-04-03T23:28:05.9061465Z",  
    "EventProcessedUtcTime": "2018-04-03T23:28:05.9061465Z"  
  },  
  {  
    "client_id": "0600000",  
    "name": "John",  
    "email": "johndoh@example.com",  
    "message": "message3",  
    "PartitionId": 0,  
    "EventEnqueuedUtcTime": "2018-04-03T23:28:05.9071453Z",  
    "EventProcessedUtcTime": "2018-04-03T23:28:05.9071453Z"  
  }  
]
```

Notice that each record in the blob has a **client_id** column matching the folder name since the column used to partition the output in the output path was **client_id**.

Limitations

- Only one custom partition key is permitted in the Path Pattern blob output property. All of the following Path Patterns are valid:
 - cluster1/{date}/{aFieldInMyData}
 - cluster1/{time}/{aFieldInMyData}
 - cluster1/{aFieldInMyData}
 - cluster1/{date}/{time}/{aFieldInMyData}
- Partition keys are case insensitive, so partition keys like "John" and "john" are equivalent. Also, expressions cannot be used as partition keys. For example, **{columnA + columnB}** does not work.
- When an input stream consists of records with a partition key cardinality under 8000, the records will be appended to existing blobs and only create new blobs when necessary. If the cardinality is over 8000 there is no guarantee existing blobs will be written to and new blobs won't be created for an arbitrary number of records with the same partition key.

Custom DateTime path patterns

Custom DateTime path patterns allow you to specify an output format that aligns with Hive Streaming conventions, giving Azure Stream Analytics the ability to send data to Azure HDInsight and Azure Databricks for

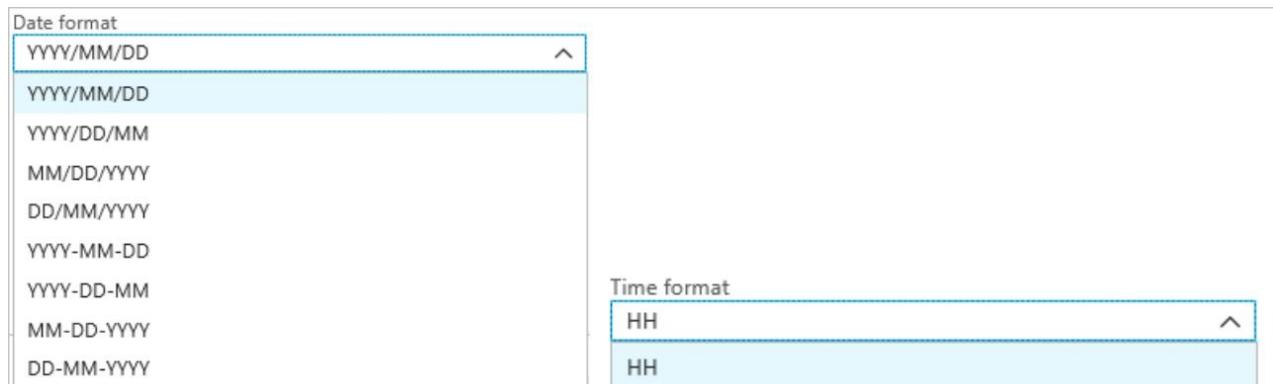
downstream processing. Custom DateTime path patterns are easily implemented using the `{datetime}` keyword in the Path Prefix field of your blob output, along with the format specifier. For example, `{datetime:yyyy}`.

Supported tokens

The following format specifier tokens can be used alone or in combination to achieve custom DateTime formats:

FORMAT SPECIFIER	DESCRIPTION	RESULTS ON EXAMPLE TIME 2018-01-02T10:06:08
<code>{datetime:yyyy}</code>	The year as a four-digit number	2018
<code>{datetime:MM}</code>	Month from 01 to 12	01
<code>{datetime:M}</code>	Month from 1 to 12	1
<code>{datetime:dd}</code>	Day from 01 to 31	02
<code>{datetime:d}</code>	Day from 1 to 12	2
<code>{datetime:HH}</code>	Hour using the 24-hour format, from 00 to 23	10
<code>{datetime:mm}</code>	Minutes from 00 to 24	06
<code>{datetime:m}</code>	Minutes from 0 to 24	6
<code>{datetime:ss}</code>	Seconds from 00 to 60	08

If you do not wish to use custom DateTime patterns, you can add the `{date}` and/or `{time}` token to the Path Prefix to generate a dropdown with built-in DateTime formats.



Extensibility and restrictions

You can use as many tokens, `{datetime:<specifier>}`, as you like in the path pattern until you reach the Path Prefix character limit. Format specifiers can't be combined within a single token beyond the combinations already listed by the date and time dropdowns.

For a path partition of `logs/MM/dd`:

VALID EXPRESSION	INVALID EXPRESSION
<code>logs/{datetime:MM}/{datetime:dd}</code>	<code>logs/{datetime:MM/dd}</code>

You may use the same format specifier multiple times in the Path Prefix. The token must be repeated each time.

Hive Streaming conventions

Custom path patterns for blob storage can be used with the Hive Streaming convention, which expects folders to be labeled with `column=` in the folder name.

For example, `year={datetime:yyyy}/month={datetime:MM}/day={datetime:dd}/hour={datetime:HH}`.

Custom output eliminates the hassle of altering tables and manually adding partitions to port data between Azure Stream Analytics and Hive. Instead, many folders can be added automatically using:

```
MSCK REPAIR TABLE while hive.exec.dynamic.partition true
```

Example

Create a storage account, a resource group, a Stream Analytics job, and an input source according to the [Azure Stream Analytics Azure Portal](#) quickstart guide. Use the same sample data used in the quickstart guide, also available on [GitHub](#).

Create a blob output sink with the following configuration:

The screenshot shows the configuration for a blob output sink. The settings are as follows:

- Path pattern:** `year={datetime:yyyy}/month={datetime:MM}/day={datetime:dd}`
- Date format:** `YYYY/MM/DD`
- Time format:** `HH`
- * Event serialization format:** `JSON`
- Encoding:** `UTF-8`
- Format:** `Line separated`

The full path pattern is as follows:

```
year={datetime:yyyy}/month={datetime:MM}/day={datetime:dd}
```

When you start the job, a folder structure based on the path pattern is created in your blob container. You can drill down to the day level.



Next steps

- Understand outputs from Azure Stream Analytics

Azure Stream Analytics job states

6/21/2019 • 2 minutes to read • [Edit Online](#)

A Stream Analytics job could be in one of four states at any given time. You can find the state of your job on your Stream Analytics job's Overview page in the Azure portal.

STATE	DESCRIPTION	RECOMMENDED ACTIONS
Running	Your job is running on Azure reading events coming from the defined input sources, processing them and writing the results to the configured output sinks.	It is a best practice to track your job's performance by monitoring key metrics .
Stopped	Your job is stopped and does not process events.	NA
Degraded	There might be intermittent issues with your input and output connections. These errors are called transient errors which might make your job enter a Degraded state. Stream Analytics will immediately try to recover from such errors and return to a Running state (within few minutes). These errors could happen due to network issues, availability of other Azure resources, deserialization errors etc. Your job's performance may be impacted when job is in degraded state.	You can look at the diagnostic or activity logs to learn more about the cause of these transient errors. In cases such as deserialization errors, it is recommended to take corrective action to ensure events aren't malformed. If the job keeps reaching the resource utilization limit, try to increase the SU number or parallelize your job . In other cases where you cannot take any action, Stream Analytics will try to recover to a <i>Running</i> state. You can use watermark delay metric to understand if these transient errors are impacting your job's performance.
Failed	Your job encountered a critical error resulting in a failed state. Events aren't read and processed. Runtime errors are a common cause for jobs ending up in a failed state.	You can configure alerts so that you get notified when job goes to Failed state. You can debug using activity and diagnostic logs to identify root cause and address the issue.

Next steps

- [Setup alerts for Azure Stream Analytics jobs](#)
- [Metrics available in Stream Analytics](#)
- [Troubleshoot using activity and diagnostic logs](#)

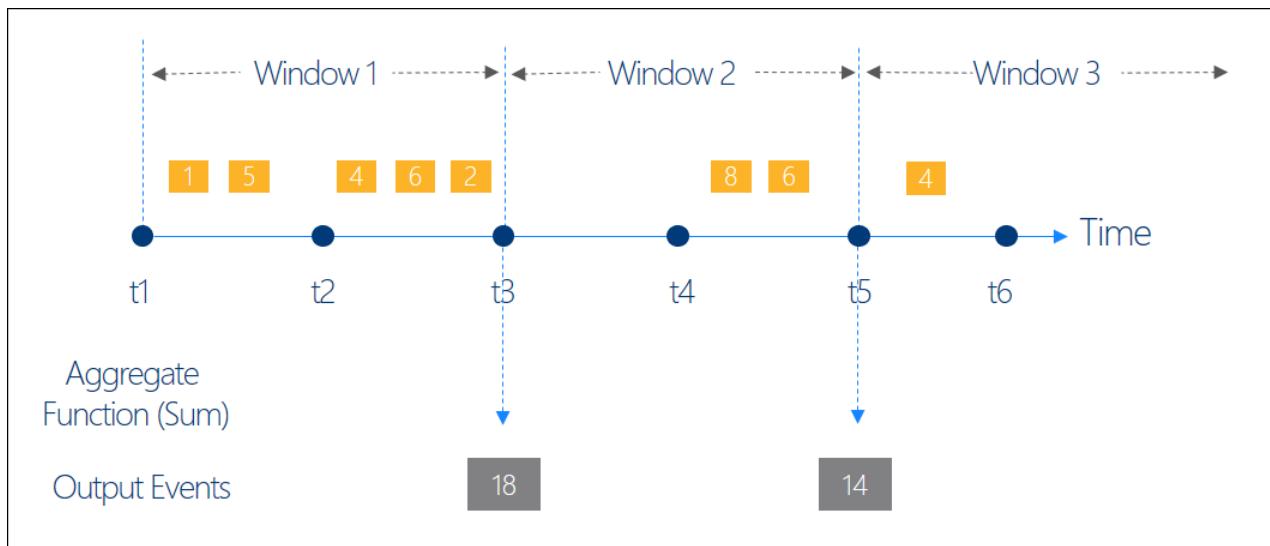
Introduction to Stream Analytics windowing functions

6/13/2019 • 2 minutes to read • [Edit Online](#)

In time-streaming scenarios, performing operations on the data contained in temporal windows is a common pattern. Stream Analytics has native support for windowing functions, enabling developers to author complex stream processing jobs with minimal effort.

There are four kinds of temporal windows to choose from: **Tumbling**, **Hopping**, **Sliding**, and **Session** windows. You use the window functions in the **GROUP BY** clause of the query syntax in your Stream Analytics jobs. You can also aggregate events over multiple windows using the **Windows()** function.

All the **windowing** operations output results at the **end** of the window. The output of the window will be single event based on the aggregate function used. The output event will have the time stamp of the end of the window and all window functions are defined with a fixed length.

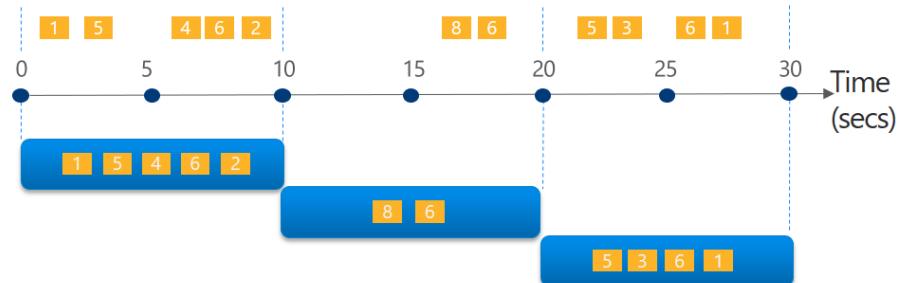


Tumbling window

Tumbling window functions are used to segment a data stream into distinct time segments and perform a function against them, such as the example below. The key differentiators of a Tumbling window are that they repeat, do not overlap, and an event cannot belong to more than one tumbling window.

Tell me the count of tweets per time zone every 10 seconds

A 10-second Tumbling Window



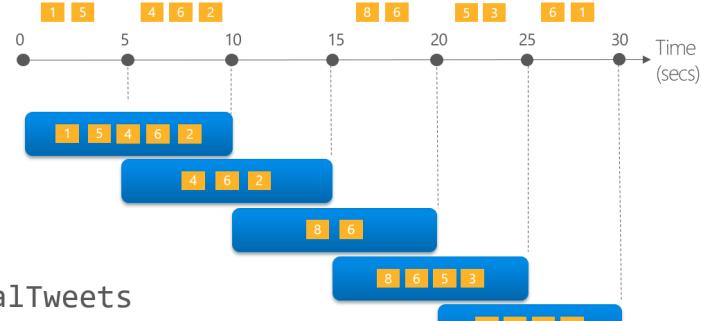
```
SELECT TimeZone, COUNT(*) AS Count
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY TimeZone, TumblingWindow(second,10)
```

Hopping window

Hopping window functions hop forward in time by a fixed period. It may be easy to think of them as Tumbling windows that can overlap, so events can belong to more than one Hopping window result set. To make a Hopping window the same as a Tumbling window, specify the hop size to be the same as the window size.

Every 5 seconds give me the count of tweets over the last 10 seconds

A 10-second Hopping Window with a 5-second "Hop"

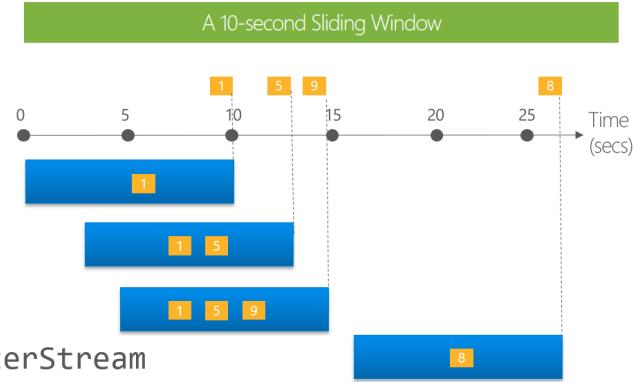


```
SELECT Topic, COUNT(*) AS TotalTweets
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY Topic, HoppingWindow(second, 10 , 5)
```

Sliding window

Sliding window functions, unlike Tumbling or Hopping windows, produce an output **only** when an event occurs. Every window will have at least one event and the window continuously moves forward by an ϵ (epsilon). Like hopping windows, events can belong to more than one sliding window.

Give me the count of tweets for all topics which are tweeted more than 10 times in the last 10 seconds

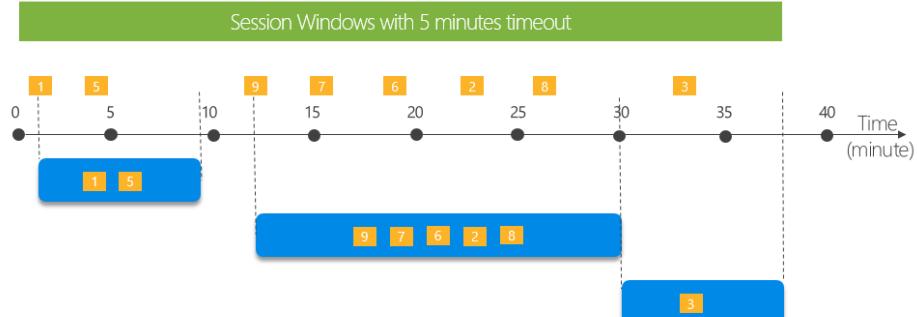


```
SELECT Topic, COUNT(*) FROM TwitterStream
TIMESTAMP BY CreatedAt
GROUP BY Topic, SlidingWindow(second, 10)
HAVING COUNT(*) > 10
```

Session window

Session window functions group events that arrive at similar times, filtering out periods of time where there is no data. It has three main parameters: timeout, maximum duration, and partitioning key (optional).

Tell me the count of tweets that occur within 5 minutes to each other.



```
SELECT Topic, COUNT(*)
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY Topic, SessionWindow(minute, 5, 10)
```

A session window begins when the first event occurs. If another event occurs within the specified timeout from the last ingested event, then the window extends to include the new event. Otherwise if no events occur within the timeout, then the window is closed at the timeout.

If events keep occurring within the specified timeout, the session window will keep extending until maximum duration is reached. The maximum duration checking intervals are set to be the same size as the specified max duration. For example, if the max duration is 10, then the checks on if the window exceed maximum duration will happen at t = 0, 10, 20, 30, etc.

When a partition key is provided, the events are grouped together by the key and session window is applied to each group independently. This partitioning is useful for cases where you need different session windows for different users or devices.

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Introduction to Stream Analytics geospatial functions

12/7/2018 • 3 minutes to read • [Edit Online](#)

Geospatial functions in Azure Stream Analytics enable real-time analytics on streaming geospatial data. With just a few lines of code, you can develop a production grade solution for complex scenarios.

Examples of scenarios that can benefit from geospatial functions include:

- Ride-sharing
- Fleet management
- Asset tracking
- Geo-fencing
- Phone tracking across cell sites

Stream Analytics Query Language has seven built-in geospatial functions: **CreateLineString**, **CreatePoint**, **CreatePolygon**, **ST_DISTANCE**, **ST_OVERLAPS**, **ST_INTERSECTS**, and **ST_WITHIN**.

CreateLineString

The `CreateLineString` function accepts points and returns a GeoJSON LineString, which can be plotted as a line on a map. You must have at least two points to create a LineString. The LineString points will be connected in order.

The following query uses `CreateLineString` to create a LineString using three points. The first point is created from streaming input data, while the other two are created manually.

```
SELECT
    CreateLineString(CreatePoint(input.latitude, input.longitude), CreatePoint(10.0, 10.0), CreatePoint(10.5,
10.5))
FROM input
```

Input example

LATITUDE	LONGITUDE
3.0	-10.2
-87.33	20.2321

Output example

```
{"type" : "LineString", "coordinates" : [ [-10.2, 3.0], [10.0, 10.0], [10.5, 10.5] ]}

{"type" : "LineString", "coordinates" : [ [20.2321, -87.33], [10.0, 10.0], [10.5, 10.5] ]}
```

To learn more, visit the [CreateLineString](#) reference.

CreatePoint

The `CreatePoint` function accepts a latitude and longitude and returns a GeoJSON point, which can be plotted on a map. Your latitudes and longitudes must be a **float** datatype.

The following example query uses `CreatePoint` to create a point using latitudes and longitudes from streaming

input data.

```
SELECT
    CreatePoint(input.latitude, input.longitude)
FROM input
```

Input example

LATITUDE	LONGITUDE
3.0	-10.2
-87.33	20.2321

Output example

```
{"type": "Point", "coordinates": [-10.2, 3.0]}

{"type": "Point", "coordinates": [20.2321, -87.33]}
```

To learn more, visit the [CreatePoint](#) reference.

CreatePolygon

The `CreatePolygon` function accepts points and returns a GeoJSON polygon record. The order of points must follow right-hand ring orientation, or counter-clockwise. Imagine walking from one point to another in the order they were declared. The center of the polygon would be to your left the entire time.

The following example query uses `CreatePolygon` to create a polygon from three points. The first two points are created manually, and the last point is created from input data.

```
SELECT
    CreatePolygon(CreatePoint(input.latitude, input.longitude), CreatePoint(10.0, 10.0), CreatePoint(10.5,
10.5), CreatePoint(input.latitude, input.longitude))
FROM input
```

Input example

LATITUDE	LONGITUDE
3.0	-10.2
-87.33	20.2321

Output example

```
{"type": "Polygon", "coordinates": [[[ -10.2, 3.0], [10.0, 10.0], [10.5, 10.5], [-10.2, 3.0]]]

 {"type": "Polygon", "coordinates": [[[ 20.2321, -87.33], [10.0, 10.0], [10.5, 10.5], [20.2321, -87.33]]]}
```

To learn more, visit the [CreatePolygon](#) reference.

ST_DISTANCE

The `ST_DISTANCE` function returns the distance between two points in meters.

The following query uses `ST_DISTANCE` to generate an event when a gas station is less than 10 km from the car.

```
SELECT Cars.Location, Station.Location
FROM Cars c
JOIN Station s ON ST_DISTANCE(c.Location, s.Location) < 10 * 1000
```

To learn more, visit the [ST_DISTANCE](#) reference.

ST_OVERLAPS

The `ST_OVERLAPS` function compares two polygons. If the polygons overlap, the function returns a 1. The function returns 0 if the polygons don't overlap.

The following query uses `ST_OVERLAPS` to generate an event when a building is within a possible flooding zone.

```
SELECT Building.Polygon, Building.Polygon
FROM Building b
JOIN Flooding f ON ST_OVERLAPS(b.Polygon, b.Polygon)
```

The following example query generates an event when a storm is heading towards a car.

```
SELECT Cars.Location, Storm.Course
FROM Cars c, Storm s
JOIN Storm s ON ST_OVERLAPS(c.Location, s.Course)
```

To learn more, visit the [ST_OVERLAPS](#) reference.

ST_INTERSECTS

The `ST_INTERSECTS` function compares two LineString. If the LineString intersect, then the function returns 1. The function returns 0 if the LineString don't intersect.

The following example query uses `ST_INTERSECTS` to determine if a paved road intersects a dirt road.

```
SELECT
    ST_INTERSECTS(input.pavedRoad, input.dirtRoad)
FROM input
```

Input example

DATACENTERAREA	STORMAREA
{"type": "LineString", "coordinates": [[-10.0, 0.0], [0.0, 0.0], [10.0, 0.0]]}	{"type": "LineString", "coordinates": [[0.0, 10.0], [0.0, 0.0], [0.0, -10.0]]}
{"type": "LineString", "coordinates": [[-10.0, 0.0], [0.0, 0.0], [10.0, 0.0]]}	{"type": "LineString", "coordinates": [[-10.0, 10.0], [0.0, 10.0], [10.0, 10.0]]}

Output example

1

0

To learn more, visit the [ST_INTERSECTS](#) reference.

ST_WITHIN

The `ST_WITHIN` function determines whether a point or polygon is within a polygon. If the polygon contains the point or polygon, the function will return 1. The function will return 0 if the point or polygon isn't located within the declared polygon.

The following example query uses `ST_WITHIN` to determine whether the delivery destination point is within the given warehouse polygon.

```
SELECT  
    ST_WITHIN(input.deliveryDestination, input.warehouse)  
FROM input
```

Input example

DELIVERYDESTINATION	WAREHOUSE
{"type": "Point", "coordinates": [76.6, 10.1]}	{"type": "Polygon", "coordinates": [[0.0, 0.0], [10.0, 0.0], [10.0, 10.0], [0.0, 10.0], [0.0, 0.0]]}
{"type": "Point", "coordinates": [15.0, 15.0]}	{"type": "Polygon", "coordinates": [[10.0, 10.0], [20.0, 10.0], [20.0, 20.0], [10.0, 20.0], [10.0, 10.0]]}

Output example

```
0  
1
```

To learn more, visit the [ST_WITHIN](#) reference.

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Compatibility level for Azure Stream Analytics jobs

5/6/2019 • 4 minutes to read • [Edit Online](#)

This article describes the compatibility level option in Azure Stream Analytics. Stream Analytics is a managed service, with regular feature updates, and performance improvements. Most of the service's runtimes updates are automatically made available to end users.

However, some new functionality in the service may introduce a major change, such as a change in the behavior of an existing job, or a change in the way data is consumed in running jobs. You can keep your existing Stream Analytics jobs running without major changes by leaving the compatibility level setting lowered. When you are ready for the latest runtime behaviors, you can opt-in by raising the compatibility level.

Choose a compatibility level

Compatibility level controls the runtime behavior of a stream analytics job.

Azure Stream Analytics currently supports three compatibility levels:

- 1.0 - Previous behavior
- 1.1 - Default behavior
- 1.2 (preview) - newest behavior with most recent improvements in evaluation

The original 1.0 compatibility level was introduced during general availability of Azure Stream Analytics several years ago.

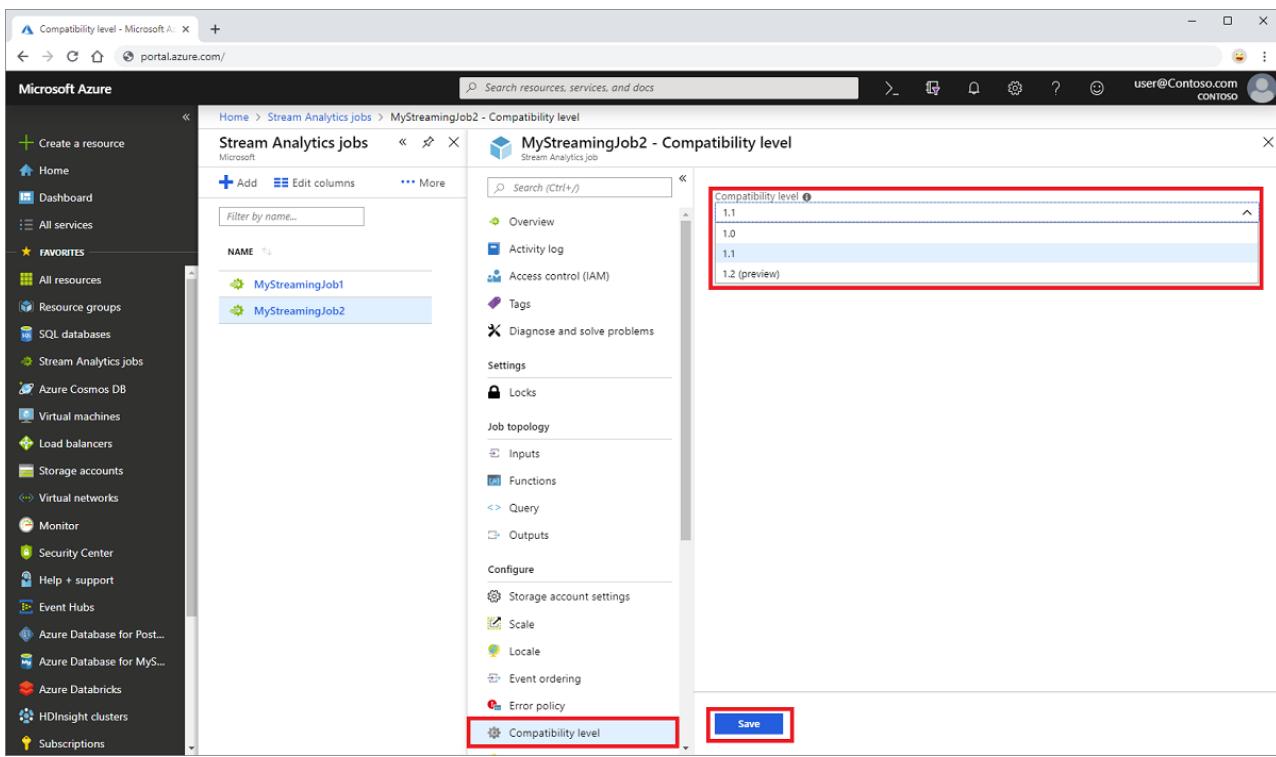
When you create a new Stream Analytics job, it's a best practice to create it by using the latest compatibility level. Start your job design relying upon the latest behaviors, to avoid added change and complexity later on.

Set the compatibility level

You can set the compatibility level for a Stream Analytics job in the Azure portal or by using the [create job REST API call](#).

To update the compatibility level of the job in the Azure portal:

1. Use the [Azure portal](#) to locate to your Stream Analytics job.
2. **Stop** the job before updating the compatibility level. You can't update the compatibility level if your job is in a running state.
3. Under the **Configure** heading, select **Compatibility level**.
4. Choose the compatibility level value that you want.
5. Select **Save** at the bottom of the page.



When you update the compatibility level, the T-SQL compiler validates the job with the syntax that corresponds to the selected compatibility level.

Compatibility level 1.2 (Preview)

The following major changes are introduced in compatibility level 1.2:

Geospatial functions

Previous levels: Azure Stream Analytics used Geography calculations.

1.2 level: Azure Stream Analytics allows you to compute Geometric projected geo coordinates. There's no change in the signature of the geospatial functions. However, their semantics is slightly different, allowing more precise computation than before.

Azure Stream Analytics supports geospatial reference data indexing. Reference Data containing geospatial elements can be indexed for a faster join computation.

The updated geospatial functions bring the full expressiveness of Well Known Text (WKT) geospatial format. You can specify other geospatial components that weren't previously supported with GeoJson.

For more information, see [Updates to geospatial features in Azure Stream Analytics – Cloud and IoT Edge](#).

Parallel query execution for input sources with multiple partitions

Previous levels: Azure Stream Analytics queries required the use of PARTITION BY clause to parallelize query processing across input source partitions.

1.2 level: If query logic can be parallelized across input source partitions, Azure Stream Analytics creates separate query instances and runs computations in parallel.

Native Bulk API integration with CosmosDB output

Previous levels: The upsert behavior was *insert or merge*.

1.2 level: Native Bulk API integration with CosmosDB output maximizes throughput and efficiently handles throttling requests. For more information, see [the Azure Stream Analytics output to Azure Cosmos DB page](#).

The upsert behavior is *insert or replace*.

DateOffset when writing to SQL output

Previous levels: `DateOffset` types were adjusted to UTC.

1.2 level: `DateOffset` is no longer adjusted.

Strict validation of prefix of functions

Previous levels: There was no strict validation of function prefixes.

1.2 level: Azure Stream Analytics has a strict validation of function prefixes. Adding a prefix to a built-in function causes an error. For example, `myprefix.ABS(...)` isn't supported.

Adding a prefix to built-in aggregates also results in error. For example, `myprefix.SUM(...)` isn't supported.

Using the prefix "system" for any user-defined functions results in error.

Disallow Array and Object as key properties in Cosmos DB output adapter

Previous levels: Array and Object types were supported as a key property.

1.2 level: Array and Object types are no longer supported as a key property.

Compatibility level 1.1

The following major changes are introduced in compatibility level 1.1:

Service Bus XML format

1.0 level: Azure Stream Analytics used `DataContractSerializer`, so the message content included XML tags. For example:

```
@\u0006string\b3http://schemas.microsoft.com/2003/10/Serialization/\u0001{ "SensorId":"1",  
"Temperature":64}\u0001
```

1.1 level: The message content contains the stream directly with no additional tags. For example:

```
{ "SensorId":"1", "Temperature":64}
```

Persisting case-sensitivity for field names

1.0 level: Field names were changed to lower case when processed by the Azure Stream Analytics engine.

1.1 level: case-sensitivity is persisted for field names when they are processed by the Azure Stream Analytics engine.

NOTE

Persisting case-sensitivity isn't yet available for Stream Analytic jobs hosted by using Edge environment. As a result, all field names are converted to lowercase if your job is hosted on Edge.

FloatNaNDeserializationDisabled

1.0 level: `CREATE TABLE` command did not filter events with NaN (Not-a-Number. For example, `Infinity`, `-Infinity`) in a FLOAT column type because they are out of the documented range for these numbers.

1.1 level: `CREATE TABLE` allows you to specify a strong schema. The Stream Analytics engine validates that the data conforms to this schema. With this model, the command can filter events with NaN values.

Disable automatic upcast for datetime strings in JSON

1.0 level: The JSON parser would automatically upcast string values with date/time/zone information to `DateTime` type and then convert it to UTC. This behavior resulted in losing the timezone information.

1.1 level: There is no more automatically upcast of string values with date/time/zone information to `DateTime` type. As a result, timezone information is kept.

Next steps

- [Troubleshoot Azure Stream Analytics inputs](#)
- [Stream Analytics Resource health](#)

Query examples for common Stream Analytics usage patterns

6/21/2019 • 13 minutes to read • [Edit Online](#)

Queries in Azure Stream Analytics are expressed in a SQL-like query language. The language constructs are documented in the [Stream Analytics query language reference](#) guide.

The query design can express simple pass-through logic to move event data from one input stream into an output data store, or it can do rich pattern matching and temporal analysis to calculate aggregates over various time windows as in the [Build an IoT solution by using Stream Analytics](#) guide. You can join data from multiple inputs to combine streaming events, and you can do lookups against static reference data to enrich the event values. You can also write data to multiple outputs.

This article outlines solutions to several common query patterns based on real-world scenarios.

Work with complex Data Types in JSON and AVRO

Azure Stream Analytics supports processing events in CSV, JSON and Avro data formats.

Both JSON and Avro may contain complex types such as nested objects (records) or arrays. For more information on working with these complex data types, refer to the [Parsing JSON and AVRO data](#) article.

Query example: Convert data types

Description: Define the types of properties on the input stream. For example, the car weight is coming on the input stream as strings and needs to be converted to **INT** to perform **SUM**.

Input:

MAKE	TIME	WEIGHT
Honda	2015-01-01T00:00:01.0000000Z	"1000"
Honda	2015-01-01T00:00:02.0000000Z	"2000"

Output:

MAKE	WEIGHT
Honda	3000

Solution:

```
SELECT
    Make,
    SUM(CAST(Weight AS BIGINT)) AS Weight
FROM
    Input TIMESTAMP BY Time
GROUP BY
    Make,
    TumblingWindow(second, 10)
```

Explanation: Use a **CAST** statement in the **Weight** field to specify its data type. See the list of supported data types in [Data types \(Azure Stream Analytics\)](#).

Query example: Use LIKE/NOT LIKE to do pattern matching

Description: Check that a field value on the event matches a certain pattern. For example, check that the result returns license plates that start with A and end with 9.

Input:

MAKE	LICENSEPLATE	TIME
Honda	ABC-123	2015-01-01T00:00:01.0000000Z
Toyota	AAA-999	2015-01-01T00:00:02.0000000Z
Nissan	ABC-369	2015-01-01T00:00:03.0000000Z

Output:

MAKE	LICENSEPLATE	TIME
Toyota	AAA-999	2015-01-01T00:00:02.0000000Z
Nissan	ABC-369	2015-01-01T00:00:03.0000000Z

Solution:

```
SELECT
  *
FROM
  Input TIMESTAMP BY Time
WHERE
  LicensePlate LIKE 'A%9'
```

Explanation: Use the **LIKE** statement to check the **LicensePlate** field value. It should start with the letter A, then have any string of zero or more characters, and then end with the number 9.

Query example: Specify logic for different cases/values (CASE statements)

Description: Provide a different computation for a field, based on a particular criterion. For example, provide a string description for how many cars of the same make passed, with a special case for 1.

Input:

MAKE	TIME
Honda	2015-01-01T00:00:01.0000000Z
Toyota	2015-01-01T00:00:02.0000000Z
Toyota	2015-01-01T00:00:03.0000000Z

Output:

CARS PASSED	TIME
1 Honda	2015-01-01T00:00:10.0000000Z
2 Toyotas	2015-01-01T00:00:10.0000000Z

Solution:

```

SELECT
    CASE
        WHEN COUNT(*) = 1 THEN CONCAT('1 ', Make)
        ELSE CONCAT(CAST(COUNT(*) AS NVARCHAR(MAX)), ' ', Make, 's')
    END AS CarsPassed,
    System.TimeStamp() AS Time
FROM
    Input TIMESTAMP BY Time
GROUP BY
    Make,
    TumblingWindow(second, 10)

```

Explanation: The **CASE** expression compares an expression to a set of simple expressions to determine the result. In this example, vehicle makes with a count of 1 returned a different string description than vehicle makes with a count other than 1.

Query example: Send data to multiple outputs

Description: Send data to multiple output targets from a single job. For example, analyze data for a threshold-based alert and archive all events to blob storage.

Input:

MAKE	TIME
Honda	2015-01-01T00:00:01.0000000Z
Honda	2015-01-01T00:00:02.0000000Z
Toyota	2015-01-01T00:00:01.0000000Z
Toyota	2015-01-01T00:00:02.0000000Z
Toyota	2015-01-01T00:00:03.0000000Z

Output1:

MAKE	TIME
Honda	2015-01-01T00:00:01.0000000Z
Honda	2015-01-01T00:00:02.0000000Z
Toyota	2015-01-01T00:00:01.0000000Z
Toyota	2015-01-01T00:00:02.0000000Z

MAKE	TIME
Toyota	2015-01-01T00:00:03.0000000Z

Output2:

MAKE	TIME	COUNT
Toyota	2015-01-01T00:00:10.0000000Z	3

Solution:

```

SELECT
    *
INTO
    ArchiveOutput
FROM
    Input TIMESTAMP BY Time

SELECT
    Make,
    System.TimeStamp() AS Time,
    COUNT(*) AS [Count]
INTO
    AlertOutput
FROM
    Input TIMESTAMP BY Time
GROUP BY
    Make,
    TumblingWindow(second, 10)
HAVING
    [Count] >= 3
  
```

Explanation: The **INTO** clause tells Stream Analytics which of the outputs to write the data to from this statement. The first query is a pass-through of the data received to an output named **ArchiveOutput**. The second query does some simple aggregation and filtering, and it sends the results to a downstream alerting system, **AlertOutput**.

Note that you can also reuse the results of the common table expressions (CTEs) (such as **WITH** statements) in multiple output statements. This option has the added benefit of opening fewer readers to the input source.

For example:

```

WITH AllRedCars AS (
    SELECT
        *
    FROM
        Input TIMESTAMP BY Time
    WHERE
        Color = 'red'
)
SELECT * INTO HondaOutput FROM AllRedCars WHERE Make = 'Honda'
SELECT * INTO ToyotaOutput FROM AllRedCars WHERE Make = 'Toyota'
  
```

Query example: Count unique values

Description: Count the number of unique field values that appear in the stream within a time window. For example, how many unique makes of cars passed through the toll booth in a 2-second window?

Input:

MAKE	TIME
Honda	2015-01-01T00:00:01.0000000Z
Honda	2015-01-01T00:00:02.0000000Z
Toyota	2015-01-01T00:00:01.0000000Z
Toyota	2015-01-01T00:00:02.0000000Z
Toyota	2015-01-01T00:00:03.0000000Z

Output:

COUNTMAKE	TIME
2	2015-01-01T00:00:02.000Z
1	2015-01-01T00:00:04.000Z

Solution:

```

SELECT
    COUNT(DISTINCT Make) AS CountMake,
    System.TIMESTAMP() AS TIME
FROM Input TIMESTAMP BY TIME
GROUP BY
    TumblingWindow(second, 2)

```

Explanation: **COUNT(DISTINCT Make)** returns the number of distinct values in the **Make** column within a time window.

Query example: Determine if a value has changed

Description: Look at a previous value to determine if it is different than the current value. For example, is the previous car on the toll road the same make as the current car?

Input:

MAKE	TIME
Honda	2015-01-01T00:00:01.0000000Z
Toyota	2015-01-01T00:00:02.0000000Z

Output:

MAKE	TIME
Toyota	2015-01-01T00:00:02.0000000Z

Solution:

```

SELECT
    Make,
    Time
FROM
    Input TIMESTAMP BY Time
WHERE
    LAG(Make, 1) OVER (LIMIT DURATION(minute, 1)) <> Make

```

Explanation: Use **LAG** to peek into the input stream one event back and get the **Make** value. Then compare it to the **Make** value on the current event and output the event if they are different.

Query example: Find the first event in a window

Description: Find the first car in every 10-minute interval.

Input:

LICENSEPLATE	MAKE	TIME
DXE 5291	Honda	2015-07-27T00:00:05.0000000Z
YZK 5704	Ford	2015-07-27T00:02:17.0000000Z
RMV 8282	Honda	2015-07-27T00:05:01.0000000Z
YHN 6970	Toyota	2015-07-27T00:06:00.0000000Z
VFE 1616	Toyota	2015-07-27T00:09:31.0000000Z
QYF 9358	Honda	2015-07-27T00:12:02.0000000Z
MDR 6128	BMW	2015-07-27T00:13:45.0000000Z

Output:

LICENSEPLATE	MAKE	TIME
DXE 5291	Honda	2015-07-27T00:00:05.0000000Z
QYF 9358	Honda	2015-07-27T00:12:02.0000000Z

Solution:

```

SELECT
    LicensePlate,
    Make,
    Time
FROM
    Input TIMESTAMP BY Time
WHERE
    IsFirst(minute, 10) = 1

```

Now let's change the problem and find the first car of a particular make in every 10-minute interval.

LICENSEPLATE	MAKE	TIME
DXE 5291	Honda	2015-07-27T00:00:05.0000000Z
YZK 5704	Ford	2015-07-27T00:02:17.0000000Z
YHN 6970	Toyota	2015-07-27T00:06:00.0000000Z
QYF 9358	Honda	2015-07-27T00:12:02.0000000Z
MDR 6128	BMW	2015-07-27T00:13:45.0000000Z

Solution:

```

SELECT
    LicensePlate,
    Make,
    Time
FROM
    Input TIMESTAMP BY Time
WHERE
    IsFirst(minute, 10) OVER (PARTITION BY Make) = 1
  
```

Query example: Find the last event in a window

Description: Find the last car in every 10-minute interval.

Input:

LICENSEPLATE	MAKE	TIME
DXE 5291	Honda	2015-07-27T00:00:05.0000000Z
YZK 5704	Ford	2015-07-27T00:02:17.0000000Z
RMV 8282	Honda	2015-07-27T00:05:01.0000000Z
YHN 6970	Toyota	2015-07-27T00:06:00.0000000Z
VFE 1616	Toyota	2015-07-27T00:09:31.0000000Z
QYF 9358	Honda	2015-07-27T00:12:02.0000000Z
MDR 6128	BMW	2015-07-27T00:13:45.0000000Z

Output:

LICENSEPLATE	MAKE	TIME
VFE 1616	Toyota	2015-07-27T00:09:31.0000000Z
MDR 6128	BMW	2015-07-27T00:13:45.0000000Z

Solution:

```

WITH LastInWindow AS
(
    SELECT
        MAX(Time) AS LastEventTime
    FROM
        Input TIMESTAMP BY Time
    GROUP BY
        TumblingWindow(minute, 10)
)
SELECT
    Input.LicensePlate,
    Input.Make,
    Input.Time
FROM
    Input TIMESTAMP BY Time
    INNER JOIN LastInWindow
    ON DATEDIFF(minute, Input, LastInWindow) BETWEEN 0 AND 10
    AND Input.Time = LastInWindow.LastEventTime

```

Explanation: There are two steps in the query. The first one finds the latest time stamp in 10-minute windows. The second step joins the results of the first query with the original stream to find the events that match the last time stamps in each window.

Query example: Detect the absence of events

Description: Check that a stream has no value that matches a certain criterion. For example, have 2 consecutive cars from the same make entered the toll road within the last 90 seconds?

Input:

MAKE	LICENSEPLATE	TIME
Honda	ABC-123	2015-01-01T00:00:01.0000000Z
Honda	AAA-999	2015-01-01T00:00:02.0000000Z
Toyota	DEF-987	2015-01-01T00:00:03.0000000Z
Honda	GHI-345	2015-01-01T00:00:04.0000000Z

Output:

MAKE	TIME	CURRENTCARLICENSEPLATE	FIRSTCARLICENSEPLATE	FIRSTCARTIME
Honda	2015-01-01T00:00:02.0000000Z	AAA-999	ABC-123	2015-01-01T00:00:01.0000000Z

Solution:

```

SELECT
    Make,
    Time,
    LicensePlate AS CurrentCarLicensePlate,
    LAG(LicensePlate, 1) OVER (LIMIT DURATION(second, 90)) AS FirstCarLicensePlate,
    LAG(Time, 1) OVER (LIMIT DURATION(second, 90)) AS FirstCarTime
FROM
    Input TIMESTAMP BY Time
WHERE
    LAG(Make, 1) OVER (LIMIT DURATION(second, 90)) = Make

```

Explanation: Use **LAG** to peek into the input stream one event back and get the **Make** value. Compare it to the **MAKE** value in the current event, and then output the event if they are the same. You can also use **LAG** to get data about the previous car.

Query example: Detect the duration between events

Description: Find the duration of a given event. For example, given a web clickstream, determine the time spent on a feature.

Input:

USER	FEATURE	EVENT	TIME
user@location.com	RightMenu	Start	2015-01-01T00:00:01.0000000Z
user@location.com	RightMenu	End	2015-01-01T00:00:08.0000000Z

Output:

USER	FEATURE	DURATION
user@location.com	RightMenu	7

Solution:

```

SELECT
    [user],
    feature,
    DATEDIFF(
        second,
        LAST(Time) OVER (PARTITION BY [user], feature LIMIT DURATION(hour, 1) WHEN Event = 'start'),
        Time) as duration
FROM input TIMESTAMP BY Time
WHERE
    Event = 'end'

```

Explanation: Use the **LAST** function to retrieve the last **TIME** value when the event type was **Start**. The **LAST** function uses **PARTITION BY [user]** to indicate that the result is computed per unique user. The query has a 1-hour maximum threshold for the time difference between **Start** and **Stop** events, but is configurable as needed (**LIMIT DURATION(hour, 1)**).

Query example: Detect the duration of a condition

Description: Find out how long a condition occurred. For example, suppose that a bug resulted in all cars having

an incorrect weight (above 20,000 pounds), and the duration of that bug must be computed.

Input:

MAKE	TIME	WEIGHT
Honda	2015-01-01T00:00:01.0000000Z	2000
Toyota	2015-01-01T00:00:02.0000000Z	25000
Honda	2015-01-01T00:00:03.0000000Z	26000
Toyota	2015-01-01T00:00:04.0000000Z	25000
Honda	2015-01-01T00:00:05.0000000Z	26000
Toyota	2015-01-01T00:00:06.0000000Z	25000
Honda	2015-01-01T00:00:07.0000000Z	26000
Toyota	2015-01-01T00:00:08.0000000Z	2000

Output:

STARTFAULT	ENDFAULT
2015-01-01T00:00:02.000Z	2015-01-01T00:00:07.000Z

Solution:

```
WITH SelectPreviousEvent AS
(
SELECT
*, 
LAG([time]) OVER (LIMIT DURATION(hour, 24)) as previousTime,
LAG([weight]) OVER (LIMIT DURATION(hour, 24)) as previousWeight
FROM input TIMESTAMP BY [time]
)

SELECT
LAG(time) OVER (LIMIT DURATION(hour, 24) WHEN previousWeight < 20000 ) [StartFault],
previousTime [EndFault]
FROM SelectPreviousEvent
WHERE
[weight] < 20000
AND previousWeight > 20000
```

Explanation: Use **LAG** to view the input stream for 24 hours and look for instances where **StartFault** and **StopFault** are spanned by the weight < 20000.

Query example: Fill missing values

Description: For the stream of events that have missing values, produce a stream of events with regular intervals. For example, generate an event every 5 seconds that reports the most recently seen data point.

Input:

T	VALUE
"2014-01-01T06:01:00"	1
"2014-01-01T06:01:05"	2
"2014-01-01T06:01:10"	3
"2014-01-01T06:01:15"	4
"2014-01-01T06:01:30"	5
"2014-01-01T06:01:35"	6

Output (first 10 rows):

WINDOWEND	LASTEVENT.T	LASTEVENT.VALUE
2014-01-01T14:01:00.000Z	2014-01-01T14:01:00.000Z	1
2014-01-01T14:01:05.000Z	2014-01-01T14:01:05.000Z	2
2014-01-01T14:01:10.000Z	2014-01-01T14:01:10.000Z	3
2014-01-01T14:01:15.000Z	2014-01-01T14:01:15.000Z	4
2014-01-01T14:01:20.000Z	2014-01-01T14:01:15.000Z	4
2014-01-01T14:01:25.000Z	2014-01-01T14:01:15.000Z	4
2014-01-01T14:01:30.000Z	2014-01-01T14:01:30.000Z	5
2014-01-01T14:01:35.000Z	2014-01-01T14:01:35.000Z	6
2014-01-01T14:01:40.000Z	2014-01-01T14:01:35.000Z	6
2014-01-01T14:01:45.000Z	2014-01-01T14:01:35.000Z	6

Solution:

```

SELECT
    System.Timestamp() AS windowEnd,
    TopOne() OVER (ORDER BY t DESC) AS lastEvent
FROM
    input TIMESTAMP BY t
GROUP BY HOPPINGWINDOW(second, 300, 5)

```

Explanation: This query generates events every 5 seconds and outputs the last event that was received previously. The [Hopping window](#) duration determines how far back the query looks to find the latest event (300 seconds in this example).

Query example: Correlate two event types within the same stream

Description: Sometimes alerts need to be generated based on multiple event types that occurred in a certain time

range. For example, in an IoT scenario for home ovens, an alert must be generated when the fan temperature is less than 40 and the maximum power during the last 3 minutes is less than 10.

Input:

TIME	DEVICEID	SENSORNAME	VALUE
"2018-01-01T16:01:00"	"Oven1"	"temp"	120
"2018-01-01T16:01:00"	"Oven1"	"power"	15
"2018-01-01T16:02:00"	"Oven1"	"temp"	100
"2018-01-01T16:02:00"	"Oven1"	"power"	15
"2018-01-01T16:03:00"	"Oven1"	"temp"	70
"2018-01-01T16:03:00"	"Oven1"	"power"	15
"2018-01-01T16:04:00"	"Oven1"	"temp"	50
"2018-01-01T16:04:00"	"Oven1"	"power"	15
"2018-01-01T16:05:00"	"Oven1"	"temp"	30
"2018-01-01T16:05:00"	"Oven1"	"power"	8
"2018-01-01T16:06:00"	"Oven1"	"temp"	20
"2018-01-01T16:06:00"	"Oven1"	"power"	8
"2018-01-01T16:07:00"	"Oven1"	"temp"	20
"2018-01-01T16:07:00"	"Oven1"	"power"	8
"2018-01-01T16:08:00"	"Oven1"	"temp"	20
"2018-01-01T16:08:00"	"Oven1"	"power"	8

Output:

EVENTTIME	DEVICEID	TEMP	ALERTMESSAGE	MAXPOWERDURINGLAST3MINS
"2018-01-01T16:05:00"	"Oven1"	30	"Short circuit heating elements"	15
"2018-01-01T16:06:00"	"Oven1"	20	"Short circuit heating elements"	15
"2018-01-01T16:07:00"	"Oven1"	20	"Short circuit heating elements"	15

Solution:

```

WITH max_power_during_last_3_mins AS (
    SELECT
        System.TimeStamp() AS windowTime,
        deviceId,
        max(value) as maxPower
    FROM
        input TIMESTAMP BY t
    WHERE
        sensorName = 'power'
    GROUP BY
        deviceId,
        SlidingWindow(minute, 3)
)

SELECT
    t1.t AS eventTime,
    t1.deviceId,
    t1.value AS temp,
    'Short circuit heating elements' as alertMessage,
    t2.maxPower AS maxPowerDuringLast3mins
    INTO resultsr

FROM input t1 TIMESTAMP BY t
JOIN max_power_during_last_3_mins t2
    ON t1.deviceId = t2.deviceId
    AND t1.t = t2.windowTime
    AND DATEDIFF(minute,t1,t2) between 0 and 3

WHERE
    t1.sensorName = 'temp'
    AND t1.value <= 40
    AND t2.maxPower > 10

```

Explanation: The first query `max_power_during_last_3_mins`, uses the [Sliding window](#) to find the max value of the power sensor for every device, during the last 3 minutes. The second query is joined to the first query to find the power value in the most recent window relevant for the current event. And then, provided the conditions are met, an alert is generated for the device.

Query example: Process events independent of Device Clock Skew (substreams)

Description: Events can arrive late or out of order due to clock skews between event producers, clock skews between partitions, or network latency. In the following example, the device clock for TollID 2 is five seconds behind TollID 1, and the device clock for TollID 3 is ten seconds behind TollID 1.

Input:

LICENSEPLATE	MAKE	TIME	TOLLID
DXE 5291	Honda	2015-07-27T00:00:01.0000000Z	1
YHN 6970	Toyota	2015-07-27T00:00:05.0000000Z	1
QYF 9358	Honda	2015-07-27T00:00:01.0000000Z	2

LICENSEPLATE	MAKE	TIME	TOLLID
GXF 9462	BMW	2015-07-27T00:00:04.0000000Z	2
VFE 1616	Toyota	2015-07-27T00:00:10.0000000Z	1
RMV 8282	Honda	2015-07-27T00:00:03.0000000Z	3
MDR 6128	BMW	2015-07-27T00:00:11.0000000Z	2
YZK 5704	Ford	2015-07-27T00:00:07.0000000Z	3

Output:

TOLLID	COUNT
1	2
2	2
1	1
3	1
2	1
3	1

Solution:

```

SELECT
    TollId,
    COUNT(*) AS Count
FROM input
    TIMESTAMP BY Time OVER TollId
GROUP BY TUMBLINGWINDOW(second, 5), TollId

```

Explanation: The [TIMESTAMP BY OVER](#) clause looks at each device timeline separately using substreams. The output events for each TollID are generated as they are computed, meaning that the events are in order with respect to each TollID instead of being reordered as if all devices were on the same clock.

Query example: Remove duplicate events in a window

Description: When performing an operation such as calculating averages over events in a given time window, duplicate events should be filtered. In the following example, the second event is a duplicate of the first.

Input:

DEVICEID	TIME	ATTRIBUTE	VALUE
1	2018-07-27T00:00:01.0000000Z	Temperature	50
1	2018-07-27T00:00:01.0000000Z	Temperature	50
2	2018-07-27T00:00:01.0000000Z	Temperature	40
1	2018-07-27T00:00:05.0000000Z	Temperature	60
2	2018-07-27T00:00:05.0000000Z	Temperature	50
1	2018-07-27T00:00:10.0000000Z	Temperature	100

Output:

AVERAGEVALUE	DEVICEID
70	1
45	2

Solution:

```

With Temp AS (
    SELECT
        COUNT(DISTINCT Time) AS CountTime,
        Value,
        DeviceId
    FROM
        Input TIMESTAMP BY Time
    GROUP BY
        Value,
        DeviceId,
        SYSTEM.TIMESTAMP()
)

SELECT
    AVG(Value) AS AverageValue, DeviceId
INTO Output
FROM Temp
GROUP BY DeviceId,TumblingWindow(minute, 5)

```

Explanation: `COUNT(DISTINCT Time)` returns the number of distinct values in the Time column within a time window. You can then use the output of this step to compute the average per device by discarding duplicates.

Geofencing and geospatial queries

Azure Stream Analytics provides built-in geospatial functions that can be used to implement scenarios such as fleet management, ride sharing, connected cars, and asset tracking. Geospatial data can be ingested in either GeoJSON or WKT formats as part of event stream or reference data. For more information, refer to the [Geofencing and](#)

[geospatial aggregation scenarios with Azure Stream Analytics](#) article.

Language extensibility through JavaScript and C#

Azure Stream Analytics query language can be extended with custom functions written in JavaScript or C# languages. For more information see the following articles:

- [Azure Stream Analytics JavaScript user-defined functions](#)
- [Azure Stream Analytics JavaScript user-defined aggregates](#)
- [Develop .NET Standard user-defined functions for Azure Stream Analytics Edge jobs](#)

Get help

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Parse JSON and Avro data in Azure Stream Analytics

6/21/2019 • 2 minutes to read • [Edit Online](#)

Azure Stream Analytics support processing events in CSV, JSON, and Avro data formats. Both JSON and Avro data can be structured and contain some complex types such as nested objects (records) and arrays.

Record data types

Record data types are used to represent JSON and Avro arrays when corresponding formats are used in the input data streams. These examples demonstrate a sample sensor, which is reading input events in JSON format. Here is example of a single event:

```
{  
    "DeviceId" : "12345",  
    "Location" :  
    {  
        "Lat": 47,  
        "Long": 122  
    },  
    "SensorReadings" :  
    {  
        "Temperature" : 80,  
        "Humidity" : 70,  
        "CustomSensor01" : 5,  
        "CustomSensor02" : 99,  
        "SensorMetadata" :  
        {  
            "Manufacturer": "ABC",  
            "Version": "1.2.45"  
        }  
    }  
}
```

Access nested fields in known schema

Use dot notation (.) to easily access nested fields directly from your query. For example, this query selects the Latitude and Longitude coordinates under the Location property in the preceding JSON data. The dot notation can be used to navigate multiple levels as shown below.

```
SELECT  
    DeviceID,  
    Location.Lat,  
    Location.Long,  
    SensorReadings.SensorMetadata.Version  
FROM input
```

Select all properties

You can select all the properties of a nested record using '*' wildcard. Consider the following example:

```
SELECT input.Location.*  
FROM input
```

The result is:

```
{  
    "Lat" : 47,  
    "Long" : 122  
}
```

Access nested fields when property name is a variable

Use the [GetRecordPropertyValue](#) function if the property name is a variable.

For example, imagine a sample data stream needs to be joined with reference data containing thresholds for each device sensor. A snippet of such reference data is shown below.

```
{  
    "DeviceId" : "12345",  
    "SensorName" : "Temperature",  
    "Value" : 75  
}
```

```
SELECT  
    input.DeviceID,  
    thresholds.SensorName  
FROM input      -- stream input  
JOIN thresholds -- reference data input  
ON  
    input.DeviceId = thresholds.DeviceId  
WHERE  
    GetRecordPropertyValue(input.SensorReadings, thresholds.SensorName) > thresholds.Value  
    -- the where statement selects the property value coming from the reference data
```

Convert record fields into separate events

To convert record fields into separate events, use the [APPLY](#) operator together with the [GetRecordProperties](#) function. For example, if the previous example had several records for SensorReading, the following query could be used to extract them into different events:

```
SELECT  
    event.DeviceID,  
    sensorReading.PropertyName,  
    sensorReading.PropertyValue  
FROM input as event  
CROSS APPLY GetRecordProperties(event.SensorReadings) AS sensorReading
```

Array data types

Array data types are an ordered collection of values. Some typical operations on array values are detailed below. These examples assume the input events have a property named "arrayField" that is an array datatype.

These examples use the functions [GetArrayElement](#), [GetArrayElements](#), [GetArrayLength](#), and the [APPLY](#) operator.

Working with a specific array element

Select array element at a specified index (selecting the first array element):

```
SELECT  
    GetArrayElement(arrayField, 0) AS firstElement  
FROM input
```

Select array length

```
SELECT
    GetArrayLength(arrayField) AS arrayLength
FROM input
```

Convert array elements into separate events

Select all array element as individual events. The [APPLY](#) operator together with the [GetArrayElements](#) built-in function extracts all array elements as individual events:

```
SELECT
    arrayElement.ArrayIndex,
    arrayElement.ArrayValue
FROM input as event
CROSS APPLY GetArrayElements(event.arrayField) AS arrayElement
```

See Also

[Data Types in Azure Stream Analytics](#)

Understand time handling in Azure Stream Analytics

4/23/2019 • 14 minutes to read • [Edit Online](#)

In this article, we discuss how you can make design choices to solve practical time handling problems in the Azure Stream Analytics service. Time handling design decisions are closely related to event ordering factors.

Background time concepts

To better frame the discussion, let's define some background concepts:

- **Event time:** The time when the original event happened. For example, when a moving car on the highway approaches a toll booth.
- **Processing time:** The time when the event reaches the processing system and is observed. For example, when a toll booth sensor sees the car and the computer system takes a few moments to process the data.
- **Watermark:** An event time marker that indicates up to what point events have been ingressed to the streaming processor. Watermarks let the system indicate clear progress on ingesting the events. By the nature of streams, the incoming event data never stops, so watermarks indicate the progress to a certain point in the stream.

The watermark concept is important. Watermarks allow Stream Analytics to determine when the system can produce complete, correct, and repeatable results that don't need to be retracted. The processing can be done in a guaranteed way that's predictable and repeatable. For example, if a recount needs to be done for some error handling condition, watermarks are safe starting and ending points.

As additional resources on this subject, see Tyler Akidau's blog posts [Streaming 101](#) and [Streaming 102](#).

Choosing the best starting time

Stream Analytics gives users two choices for picking event time:

1. Arrival time

Arrival time is assigned at the input source when the event reaches the source. You can access arrival time by using the **EventEnqueuedUtcTime** property for Event Hubs inputs, **IoTHub.EnqueueTime** property for IoT Hub, and using the **BlobProperties.LastModified** property for blob input.

Using arrival time is the default behavior, and best used for data archiving scenarios, where there's no temporal logic necessary.

2. Application time (also named Event Time)

Application time is assigned when the event is generated, and it's part of the event payload. To process events by application time, use the **Timestamp by** clause in the select query. If the **Timestamp by** clause is absent, events are processed by arrival time.

It's important to use a timestamp in the payload when temporal logic is involved. That way, delays in the source system or in the network can be accounted for.

How time progresses in Azure Stream Analytics

When using application time, the time progression is based on the incoming events. It's difficult for the stream processing system to know if there are no events, or if events are delayed. For this reason, Azure Stream Analytics

generates heuristic watermarks in the following ways for each input partition:

1. Whenever there's any incoming event, the watermark is the largest event time we have seen so far minus the out-of-order tolerance window size.
2. Whenever there is no incoming event, the watermark is the current estimated arrival time (the elapsed time on behind the scenes VM processing the events from last time an input event is seen plus that input event's arrival time) minus the late arrival tolerance window.

The arrival time can only be estimated, because the real arrival time is generated on the input event broker, such as Event Hubs, and not the Azure Stream Analytics VM processing the events.

The design serves two additional purposes, besides generating watermarks:

1. The system generates results in a timely fashion with or without incoming events.

You have control over how timely they want to see the output results. In the Azure portal, on the **Event ordering** page of your Stream Analytics job, you can configure the **Out of order events** setting. When configuring that setting, consider the trade-off of timeliness with tolerance of out-of-order events in the event stream.

The late arrival tolerance window is important to keep generating watermarks, even in the absence of incoming events. At times, there may be a period where no incoming events come in, such as when an event input stream is sparse. That problem is exacerbated by the use of multiple partitions in the input event broker.

Streaming data processing systems without a late arrival tolerance window may suffer from delayed outputs when inputs are sparse and multiple partitions are used.

2. The system behavior has to be repeatable. Repeatability is an important property of a streaming data processing system.

The watermark is derived from arrival time and application time. Both are persisted in the event broker, and thus repeatable. In the case the arrival time has to be estimated in the absence of events, Azure Stream Analytics journals the estimated arrival time for repeatability during replay for the purpose of failure recovery.

Notice that when you choose to use **arrival time** as the event time, there is no need to configure the out-of-order tolerance and late arrival tolerance. Since **arrival time** is guaranteed to be monotonically increasing in the input event broker, Azure Stream Analytics simply disregards the configurations.

Late arriving events

By definition of late arrival tolerance window, for each incoming event, Azure Stream Analytics compares the **event time** with the **arrival time**; if the event time is outside of the tolerance window, you can configure the system to either drop the event or adjust the event's time to be within the tolerance.

Consider that after watermarks are generated, the service can potentially receive events with event time lower than the watermark. You can configure the service to either **drop** those events, or **adjust** the event's time to the watermark value.

As a part of the adjustment, the event's **System.Timestamp** is set to the new value, but the **event time** field itself is not changed. This adjustment is the only situation where an event's **System.Timestamp** can be different from the value in the event time field, and may cause unexpected results to be generated.

Handling time variation with substreams

The heuristic watermark generation mechanism described here works well in most of the cases where time is

mostly synchronized between the various event senders. However, in real life, especially in many IoT scenarios, the system has little control over the clock on the event senders. The event senders could be all sorts of devices in the field, perhaps on different versions of hardware and software.

Instead of using a watermark global to all events in an input partition, Stream Analytics has another mechanism called substreams to help you. You can utilize substreams in your job by writing a job query that uses the **TIMESTAMP BY** clause and the keyword **OVER**. To designate the substream, provide a key column name after the **OVER** keyword, such as a `deviceid`, so that system applies time policies by that column. Each substream gets its own independent watermark. This mechanism is useful to allow timely output generation, when dealing with large clock skews or network delays among event senders.

Substreams are a unique solution provided by Azure Stream Analytics, and are not offered by other streaming data processing systems. Stream Analytics applies the late arrival tolerance window to incoming events when substreams are used. The default setting (5 seconds) is likely too small for devices with divergent timestamps. We recommend that you start with 5 minutes, and make adjustments according to their device clock skew pattern.

Early arriving events

You may have noticed another concept called early arrival window, that looks like the opposite of late arrival tolerance window. This window is fixed at 5 minutes, and serves a different purpose from late arrival one.

Because Azure Stream Analytics guarantees it always generates complete results, you can only specify **job start time** as the first output time of the job, not the input time. The job start time is required so that the complete window is processed, not just from the middle of the window.

Stream Analytics then derives the starting time from the query specification. However, because input event broker is only indexed by arrival time, the system has to translate the starting event time to arrival time. The system can start processing events from that point in the input event broker. With the early arriving window limit, the translation is straightforward. It's starting event time minus the 5-minute early arriving window. This calculation also means that the system drops all events that are seen having event time 5 minutes greater than arrival time.

This concept is used to ensure the processing is repeatable no matter where you start to output from. Without such a mechanism, it would not be possible to guarantee repeatability, as many other streaming systems claim they do.

Side effects of event ordering time tolerances

Stream Analytics jobs have several **Event ordering** options. Two can be configured in the Azure portal: the **Out of order events** setting (out-of-order tolerance), and the **Events that arrive late** setting (late arrival tolerance). The **early arrival** tolerance is fixed and cannot be adjusted. These time policies are used by Stream Analytics to provide strong guarantees. However, these settings do have some sometimes unexpected implications:

1. Accidentally sending events that are too early.

Early events should not be outputted normally. It's possible that early events are sent to the output if sender's clock is running too fast though. All early arriving events are dropped, so you will not see any of them from the output.

2. Sending old events to Event Hubs to be processed by Azure Stream Analytics.

While old events may seem harmless at first, because of the application of the late arrival tolerance, the old events may be dropped. If the events are too old, the **System.Timestamp** value is altered during event ingestion. Due to this behavior, currently Azure Stream Analytics is more suited for near-real-time event processing scenarios, instead of historical event processing scenarios. You can set the **Events that arrive late** time to the largest possible value (20 days) to work around this behavior in some cases.

3. Outputs seem to be delayed.

The first watermark is generated at the calculated time: the **maximum event time** the system has observed so far, minus the out-of-order tolerance window size. By default, the out-of-order tolerance is configured to zero (00 minutes and 00 seconds). When you set it to a higher, non-zero time value, the streaming job's first output is delayed by that value of time (or greater) due to the first watermark time that is calculated.

4. Inputs are sparse.

When there is no input in a given partition, the watermark time is calculated as the **arrival time** minus the late arrival tolerance window. As a result, if input events are infrequent and sparse, the output can be delayed by that amount of time. The default **Events that arrive late** value is 5 seconds. You should expect to see some delay when sending input events one at a time, for example. The delays can get worse, when you set **Events that arrive late** window to a large value.

5. **System.Timestamp** value is different from the time in the **event time** field.

As described previously, the system adjusts event time by the out-of-order tolerance or late arrival tolerance windows. The **System.Timestamp** value of the event is adjusted, but not the **event time** field.

Metrics to observe

You can observe a number of the Event ordering time tolerance effects through [Stream Analytics job metrics](#). The following metrics are relevant:

METRIC	DESCRIPTION
Out-of-Order Events	Indicates the number of events received out of order, that were either dropped or given an adjusted timestamp. This metric is directly impacted by the configuration of the Out of order events setting on the Event ordering page on the job in the Azure portal.
Late Input Events	Indicates the number of events arriving late from the source. This metric includes events that have been dropped or have had their timestamp was adjusted. This metric is directly impacted by the configuration of the Events that arrive late setting in the Event ordering page on the job in the Azure portal.
Early Input Events	Indicates the number of events arriving early from the source that have either been dropped, or their timestamp has been adjusted if they are beyond 5 minutes early.
Watermark Delay	Indicates the delay of the streaming data processing job. See more information in the following section.

Watermark Delay details

The **Watermark delay** metric is computed as the wall clock time of the processing node minus the largest watermark it has seen so far. For more information, see the [watermark delay blog post](#).

There can be several reasons this metric value is larger than 0 under normal operation:

1. Inherent processing delay of the streaming pipeline. Normally this delay is nominal.
2. The out-of-order tolerance window introduced delay, because watermark is reduced by the size of the tolerance window.

3. The late arrival window introduced delay, because watermark is reduced by the size the tolerance window.
4. Clock skew of the processing node generating the metric.

There are a number of other resource constraints that can cause the streaming pipeline to slow down. The watermark delay metric can rise due to:

1. Not enough processing resources in Stream Analytics to handle the volume of input events. To scale up resources, see [Understand and adjust Streaming Units](#).
2. Not enough throughput within the input event brokers, so they are throttled. For possible solutions, see [Automatically scale up Azure Event Hubs throughput units](#).
3. Output sinks are not provisioned with enough capacity, so they are throttled. The possible solutions vary widely based on the flavor of output service being used.

Output event frequency

Azure Stream Analytics uses watermark progress as the only trigger to produce output events. Because the watermark is derived from input data, it is repeatable during failure recovery and also in user initiated reprocessing.

When using [windowed aggregates](#), the service only produces outputs at the end of the windows. In some cases, users may want to see partial aggregates generated from the windows. Partial aggregates are not supported currently in Azure Stream Analytics.

In other streaming solutions, output events could be materialized at various trigger points, depending on external circumstances. It's possible in some solutions that the output events for a given time window could be generated multiple times. As the input values are refined, the aggregate results become more accurate. Events could be speculated at first, and revised over time. For example, when a certain device is offline from the network, an estimated value could be used by a system. Later on, the same device comes online to the network. Then the actual event data could be included in the input stream. The output results from processing that time window produces more accurate output.

Illustrated example of watermarks

The following images illustrate how watermarks progress in different circumstances.

This table shows the example data that is charted below. Notice that the event time and the arrival time vary, sometimes matching and sometimes not.

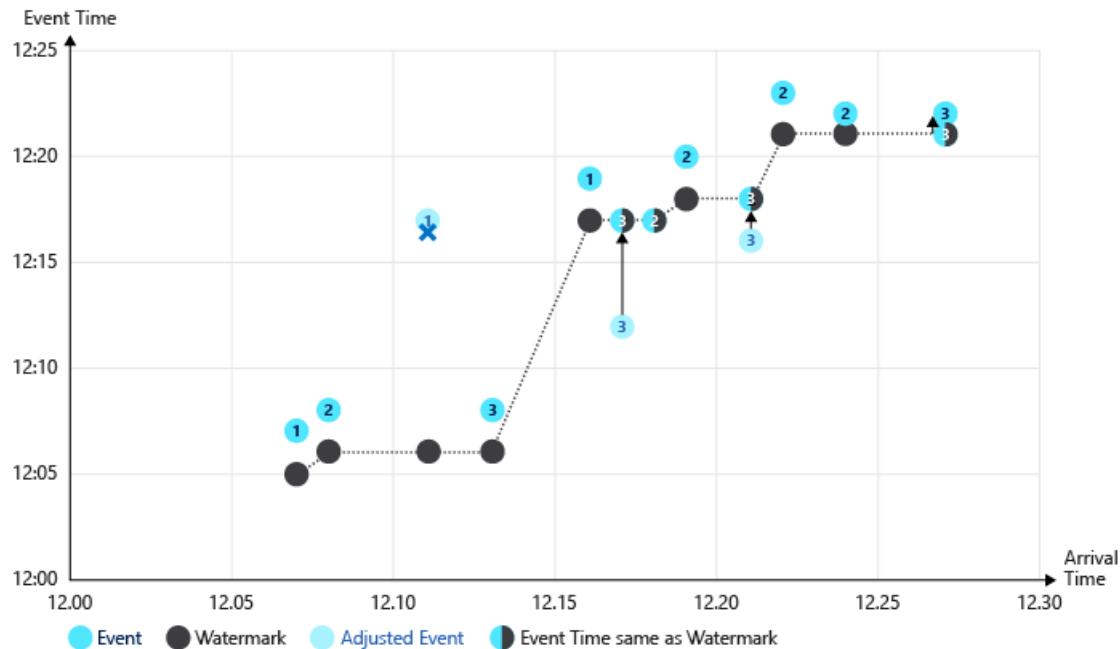
EVENT TIME	ARRIVAL TIME	DEVICEID
12:07	12:07	device1
12:08	12:08	device2
12:17	12:11	device1
12:08	12:13	device3
12:19	12:16	device1
12:12	12:17	device3
12:17	12:18	device2

EVENT TIME	ARRIVAL TIME	DEVICEID
12:20	12:19	device2
12:16	12:21	device3
12:23	12:22	device2
12:22	12:24	device2
12:21	12:27	device3

In this illustration, the following tolerances are used:

- Early arrival windows is 5 minutes
- Late arriving window is 5 minutes
- Reorder window is 2 minutes

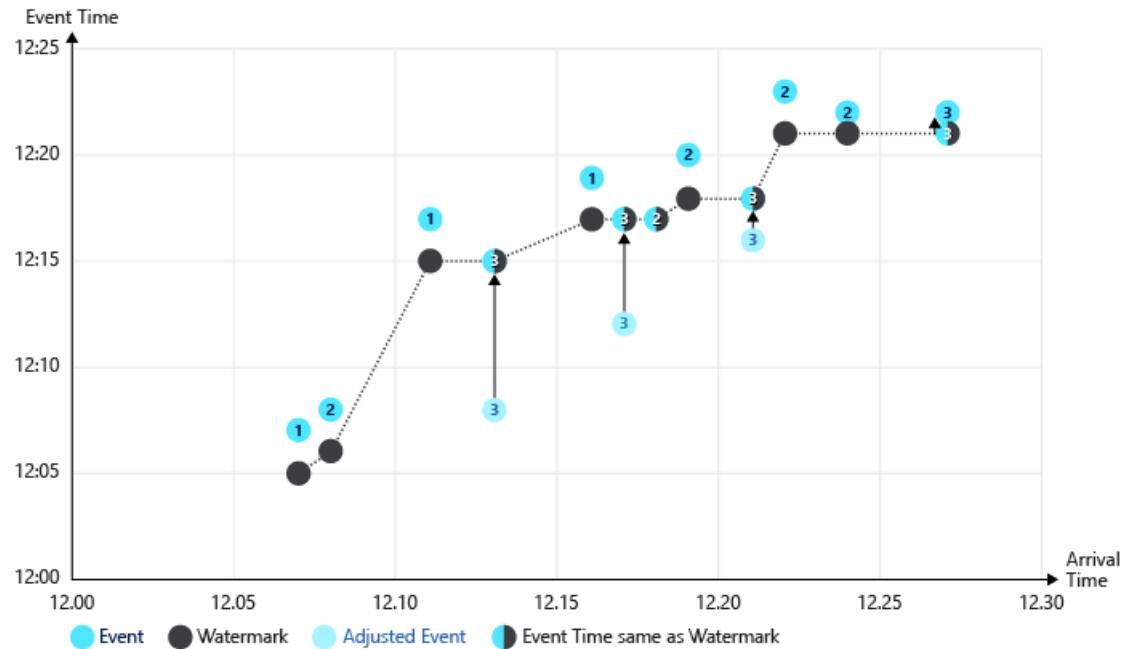
1. Illustration of watermark progressing through these events:



Notable processes illustrated in the preceding graphic:

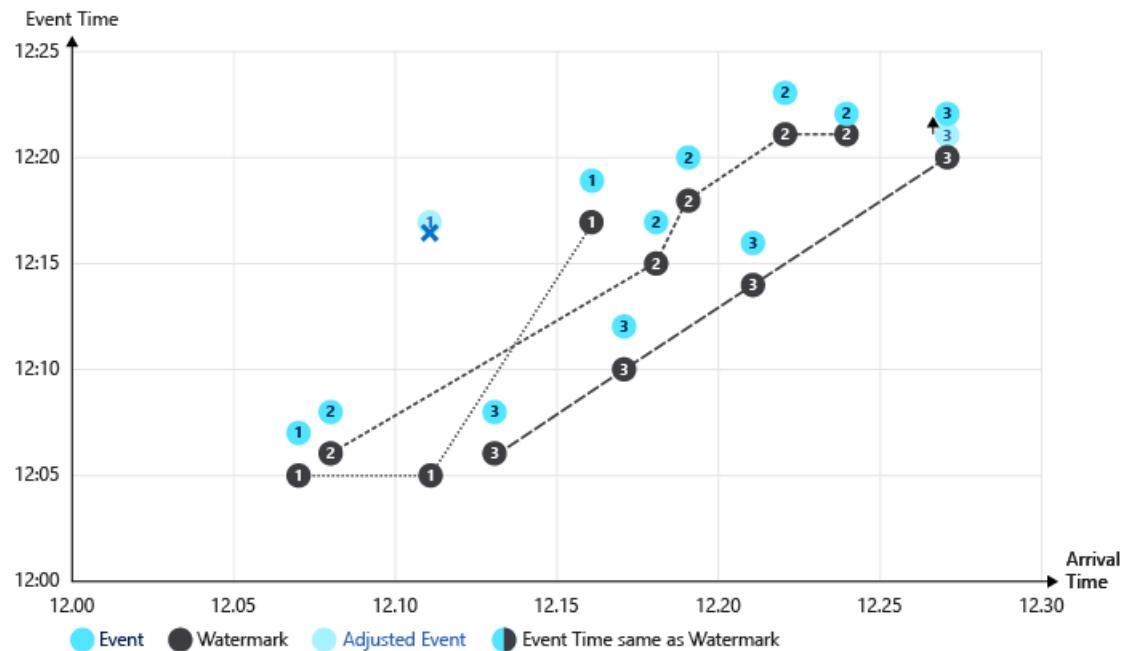
- a. The first event (device1), and second event (device2) have aligned times and are processed without adjustments. The watermark progresses on each event.
- b. When the third event (device1) is processed, the arrival time (12:11) precedes the event time (12:17). The event arrived 6 minutes early, so the event is dropped due to the 5-minute early arrival tolerance. The watermark doesn't progress in this case of an early event.
- c. The fourth event (device3), and fifth event (device1) have aligned times and are processed without adjustment. The watermark progresses on each event.
- d. When the sixth event (device3) is processed, the arrival time (12:17) and the event time (12:12) is below the watermark level. The event time is adjusted to the water mark level (12:17).
- e. When the ninth event (device3) is processed, the arrival time (12:27) is 6 minutes ahead of the event time (12:21). The late arrival policy is applied. The event time is adjusted (12:22), which is above the watermark (12:21) so no further adjustment is applied.

2. Second illustration of watermark progressing without an early arrival policy:



In this example, no early arrival policy is applied. Outlier events that arrive early raise the watermark significantly. Notice the third event (deviceld1 at time 12:11) is not dropped in this scenario, and the watermark is raised to 12:15. The fourth event time is adjusted forward 7 minutes (12:08 to 12:15) as a result.

3. In the final illustration, substreams are used (OVER the Deviceld). Multiple watermarks are tracked, one per stream. There are fewer events with their times adjusted as a result.



Next steps

- [Azure Stream Analytics event order considerations](#)
- [Stream Analytics job metrics](#)

Checkpoint and replay concepts in Azure Stream Analytics jobs

12/7/2018 • 4 minutes to read • [Edit Online](#)

This article describes the internal checkpoint and replay concepts in Azure Stream Analytics, and the impact those have on job recovery. Each time a Stream Analytics job runs, state information is maintained internally. That state information is saved in a checkpoint periodically. In some scenarios, the checkpoint information is used for job recovery if a job failure or upgrade occurs. In other circumstances, the checkpoint cannot be used for recovery, and a replay is necessary.

Stateful query logic in temporal elements

One of the unique capability of Azure Stream Analytics job is to perform stateful processing, such as windowed aggregates, temporal joins, and temporal analytic functions. Each of these operators keeps state information when the job runs. The maximum window size for these query elements is seven days.

The temporal window concept appears in several Stream Analytics query elements:

1. Windowed aggregates (GROUP BY of Tumbling, Hopping, and Sliding windows)
2. Temporal joins (JOIN with DATEDIFF)
3. Temporal analytic functions (ISFIRST, LAST, and LAG with LIMIT DURATION)

Job recovery from node failure, including OS upgrade

Each time a Stream Analytics job runs, internally it is scaled out to do work across multiple worker nodes. Each worker node's state is checkpointed every few minutes, which helps the system recover if a failure occurs.

At times, a given worker node may fail, or an Operating System upgrade can occur for that worker node. To recover automatically, Stream Analytics acquires a new healthy node, and the prior worker node's state is restored from the latest available checkpoint. To resume the work, a small amount of replay is necessary to restore the state from the time when the checkpoint is taken. Usually, the restore gap is only a few minutes. When enough Streaming Units are selected for the job, the replay should be completed quickly.

In a fully parallel query, the time it takes to catch up after a worker node failure is proportional to:

[the input event rate] x [the gap length] / [number of processing partitions]

If you ever observe significant processing delay because of node failure and OS upgrade, consider making the query fully parallel, and scale the job to allocate more Streaming Units. For more information, see [Scale an Azure Stream Analytics job to increase throughput](#).

Current Stream Analytics does not show a report when this kind of recovery process is taking place.

Job recovery from a service upgrade

Microsoft occasionally upgrades the binaries that run the Stream Analytics jobs in the Azure service. At these times, users' running jobs are upgraded to newer version and the job restarts automatically.

Currently, the recovery checkpoint format is not preserved between upgrades. As a result, the state of the streaming query must be restored entirely using replay technique. In order to allow Stream Analytics jobs to replay the exact same input from before, it's important to set the retention policy for the source data to at least the

window sizes in your query. Failing to do so may result in incorrect or partial results during service upgrade, since the source data may not be retained far enough back to include the full window size.

In general, the amount of replay needed is proportional to the size of the window multiplied by the average event rate. As an example, for a job with an input rate of 1000 events per second, a window size greater than one hour is considered to have a large replay size. Up to one hour of data may need to be re-processed to initialize the state so it can produce full and correct results, which may cause delayed output (no output) for some extended period.

Queries with no windows or other temporal operators, like `JOIN` or `LAG`, would have zero replay.

Estimate replay catch-up time

To estimate the length of the delay due to a service upgrade, you can follow this technique:

1. Load the input Event Hub with sufficient data to cover the largest window size in your query, at expected event rate. The events' timestamp should be close to the wall clock time throughout that period of time, as if it's a live input feed. For example, if you have a 3-day window in your query, send events to Event Hub for three days, and continue to send events.
2. Start the job using **Now** as the start time.
3. Measure the time between the start time and when the first output is generated. The time is rough how much delay the job would incur during a service upgrade.
4. If the delay is too long, try to partition your job and increase number of SUs, so the load is spread out to more nodes. Alternatively, consider reducing the window sizes in your query, and perform further aggregation or other stateful processing on the output produced by the Stream Analytics job in the downstream sink (for example, using Azure SQL database).

For general service stability concern during upgrade of mission critical jobs, consider running duplicate jobs in paired Azure regions. For more information, see [Guarantee Stream Analytics job reliability during service updates](#).

Job recovery from a user initiated stop and start

To edit the Query syntax on a streaming job, or to adjust inputs and outputs, the job needs to be stopped to make the changes and upgrade the job design. In such scenarios, when a user stops the streaming job, and starts it again, the recovery scenario is similar to service upgrade.

Checkpoint data cannot be used for a user initiated job restart. To estimate the delay of output during such a restart, use the same procedure as described in the previous section, and apply similar mitigation if the delay is too long.

Next steps

For more information on reliability and scalability, see these articles:

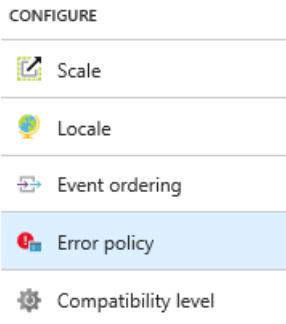
- [Tutorial: Set up alerts for Azure Stream Analytics jobs](#)
- [Scale an Azure Stream Analytics job to increase throughput](#)
- [Guarantee Stream Analytics job reliability during service updates](#)

Azure Stream Analytics output error policy

12/7/2018 • 2 minutes to read • [Edit Online](#)

This article describes the output data error handling policies that can be configured in Azure Stream Analytics.

Output data error handling policies apply only to data conversion errors that occur when the output event produced by a Stream Analytics job does not conform to the schema of the target sink. You can configure this policy by choosing either **Retry** or **Drop**. In the Azure portal, while in a Stream Analytics job, under **Configure**, select **Error Policy** to make your selection.



Retry

When an error occurs, Azure Stream Analytics retries writing the event indefinitely until the write succeeds. There is no timeout for retries. Eventually all subsequent events are blocked from processing by the event that is retrying. This option is the default output error handling policy.

Drop

Azure Stream Analytics will drop any output event that results in a data conversion error. The dropped events cannot be recovered for reprocessing later.

All transient errors (for example, network errors) are retried regardless of the output error handling policy configuration.

Next steps

[Troubleshooting guide for Azure Stream Analytics](#)

Rotate login credentials for inputs and outputs of a Stream Analytics Job

6/21/2019 • 3 minutes to read • [Edit Online](#)

Whenever you regenerate credentials for an input or output of a Stream Analytics job, you should update the job with new credentials. You must stop the job before updating the credentials, you can't replace the credentials while the job is running. To reduce the lag between stopping and restarting the job, Stream Analytics supports resuming a job from its last output. This topic describes the process of rotating the login credentials and restarting the job with new credentials.

Regenerate new credentials and update your job with the new credentials

In this section, we will walk you through regenerating credentials for Blob Storage, Event Hubs, SQL Database, and Table Storage.

Blob storage/Table storage

1. Sign in to the Azure portal > browse the storage account that you used as input/output for the Stream Analytics job.
2. From the settings section, open **Access keys**. Between the two default keys (key1, key2), pick the one that is not used by your job and regenerate it:

The screenshot shows the 'Access keys' page for a storage account named 'asaiot'. On the left, there's a sidebar with options like Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. Under SETTINGS, 'Access keys' is selected and highlighted in blue. The main pane displays instructions for using access keys to authenticate applications. It shows two default keys: 'key1' and 'key2'. The 'key1' row is currently selected, with its value partially visible as '9q5adeti6a2U4jQMDuQnxqINU+Cf5D9K4I...'. Below the key values are 'CONNECTION STRING' fields and 'Regenerate' buttons. The 'key2' row shows a different key value and its connection string. A red box highlights the 'Regenerate' button for 'key2'.

3. Copy the newly generated key.
4. From the Azure portal, browse your Stream Analytics job > select **Stop** and wait for the job to stop.
5. Locate the Blob/Table storage input/output for which you want to update credentials.
6. Find the **Storage Account Key** field and paste your newly generated key > click **Save**.
7. A connection test will automatically start when you save your changes, you can view it from the notifications tab. There are two notifications- one corresponds to saving the update and other corresponds to testing the connection:

The screenshot shows the Azure Notifications panel with two entries:

- Successful connection test** at 1:56 PM: Connection to input 'ip2' succeeded.
- Edited input** at 1:56 PM: Edited input 'ip2' in Streaming job 'TSASA'.

8. Proceed to [start your job from the last stopped time](#) section.

Event hubs

1. Sign in to the Azure portal > browse the Event Hub that you used as input/output for the Stream Analytics job.
2. From the settings section, open **Shared access policies** and select the required access policy. Between the **Primary Key** and **Secondary Key**, pick the one that is not used by your job and regenerate it:

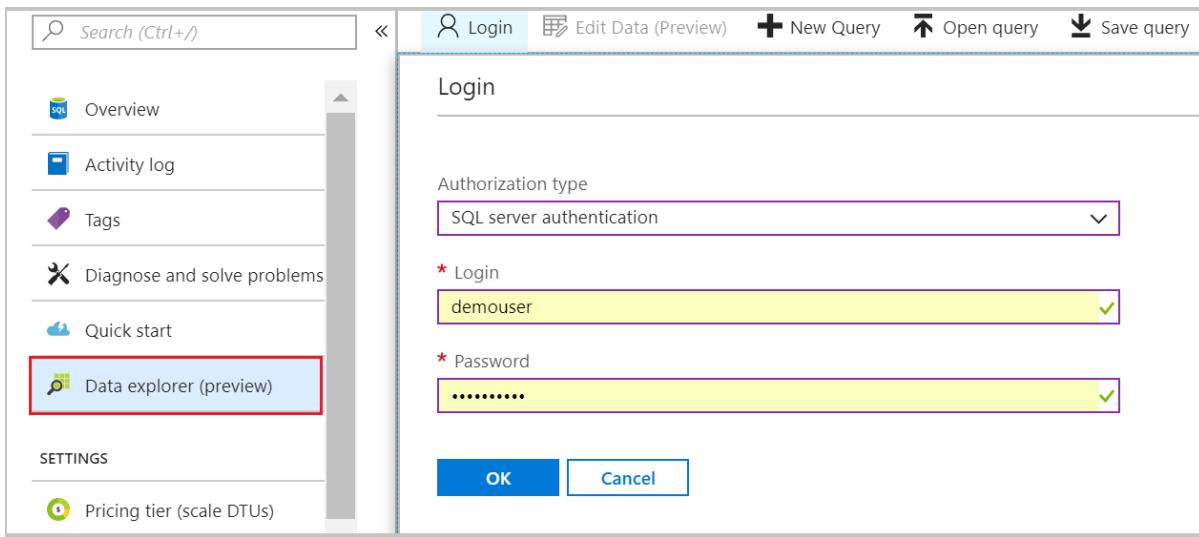
The screenshot shows the 'Shared access policies' page for an Event Hub named 'Ehub11'. The left sidebar shows options like Overview, Access control (IAM), Tags, and Shared access policies (which is selected and highlighted with a red box). The main area lists a single policy named 'RootManageSharedAccessKey'. On the right, there are buttons for Save, Discard, Delete, and More. Under the 'More' button, two options are listed: 'Regenerate Primary Keys' and 'Regenerate Secondary Keys', with 'Regenerate Secondary Keys' highlighted with a red box. Below these are fields for Primary key, Secondary key, Connection string-primary key, and Connection string-secondary key, each with a download icon.

3. Copy the newly generated key.
4. From the Azure portal, browse your Stream Analytics job > select **Stop** and wait for the job to stop.
5. Locate the Event hubs input/output for which you want to update credentials.
6. Find the **Event Hub Policy Key** field and paste your newly generated key > click **Save**.
7. A connection test will automatically start when you save your changes, make sure that it has successfully passed.
8. Proceed to [start your job from the last stopped time](#) section.

SQL Database

You need to connect to the SQL database to update the login credentials of an existing user. You can update credentials by using Azure portal or a client-side tool such as SQL Server Management Studio. This section demonstrates the process of updating credentials by using Azure portal.

1. Sign in to the Azure portal > browse the SQL database that you used as output for the Stream Analytics job.
2. From **Data explorer**, login/connect to your database > select Authorization type as **SQL server authentication** > type in your **Login** and **Password** details > Select **Ok**.



3. In the query tab, alter the password for one of your user's by running the following query (make sure to replace <user_name> with your username and <new_password> with your new password):

```
Alter user `<user_name>` WITH PASSWORD = '<new_password>'  
Alter role db_owner Add member `<user_name>`
```

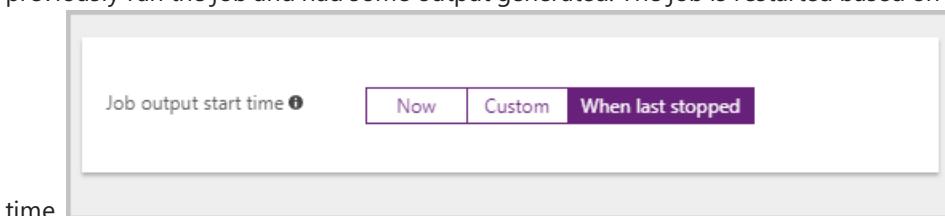
4. Make a note of the new password.
5. From the Azure portal, browse your Stream Analytics job > select **Stop** and wait for the job to stop.
6. Locate the SQL database output for which you want to rotate credentials. Update the password and save changes.
7. A connection test will automatically start when you save your changes, make sure that it has successfully passed.
8. Proceed to [start your job from the last stopped time](#) section.

Power BI

1. Sign in to the Azure portal > browse your Stream Analytics job > select **Stop** and wait for the job to stop.
2. Locate the Power BI output for which you want to renew credentials > click the **Renew authorization** (you should see a success message) > **Save** the changes.
3. A connection test will automatically start when you save your changes, make sure it has successfully passed.
4. Proceed to [start your job from the last stopped time](#) section.

Start your job from the last stopped time

1. Navigate to the job's **Overview** pane > select **Start** to start the job.
2. Select **When last stopped** > click **Start**. Note that the "When last stopped" option only appears if you previously ran the job and had some output generated. The job is restarted based on the last output value's



Next steps

- [Introduction to Azure Stream Analytics](#)

- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Configuring event ordering policies for Azure Stream Analytics

6/19/2019 • 4 minutes to read • [Edit Online](#)

This article describes how to setup and use late arrival and out-of-order event policies in Azure Stream Analytics. These policies are applied only when you use the **TIMESTAMP BY** clause in your query.

Event time and Arrival Time

Your Stream Analytics job can process events based on either *event time* or *arrival time*. **Event/application time** is the timestamp present in event payload (when the event was generated). **Arrival time** is the timestamp when the event was received at the input source (Event Hubs/IoT Hub/Blob storage).

By default, Stream Analytics processes events by *arrival time*, but you can choose to process events by *event time* by using **TIMESTAMP BY** clause in your query. Late arrival and out-of-order policies are applicable only if you process events by event time. Consider latency and correctness requirements for your scenario when configuring these settings.

What is late arrival policy?

Sometimes events arrive late because of various reasons. For example, an event that arrives 40 seconds late will have event time = 00:10:00 and arrival time = 00:10:40. If you set the late arrival policy to 15 seconds, any event that arrives later than 15 seconds will either be dropped (not processed by Stream Analytics) or have their event time adjusted. In the example above, as the event arrived 40 seconds late (more than policy set), its event time will be adjusted to the maximum of late arrival policy 00:10:25 (arrival time - late arrival policy value). Default late arrival policy is 5 seconds.

What is out-of-order policy?

Event may arrive out of order as well. After event time is adjusted based on late arrival policy, you can also choose to automatically drop or adjust events that are out-of-order. If you set this policy to 8 seconds, any events that arrive out of order but within the 8-second window are reordered by event time. Events that arrive later will be either dropped or adjusted to the maximum out-of-order policy value. Default out-of-order policy is 0 seconds.

Adjust or Drop events

If events arrive late or out-of-order based on the policies you have configured, you can either drop such events (not processed by Stream Analytics) or have their event time adjusted.

Let us see an example of these policies in action.

Late arrival policy: 15 seconds

Out-of-order policy: 8 seconds

EVENT NO.	EVENT TIME	ARRIVAL TIME	SYSTEM.TIMESTAMP	EXPLANATION
-----------	------------	--------------	------------------	-------------

Event No.	Event Time	Arrival Time	System.Timestamp	Explanation
1	00:10:00	00:10:40	00:10:25	Event arrived late and outside tolerance level. So event time gets adjusted to maximum late arrival tolerance.
2	00:10:30	00:10:41	00:10:30	Event arrived late but within tolerance level. So event time does not get adjusted.
3	00:10:42	00:10:42	00:10:42	Event arrived on time. No adjustment needed.
4	00:10:38	00:10:43	00:10:38	Event arrived out-of-order but within the tolerance of 8 seconds. So, event time does not get adjusted. For analytics purposes, this event will be considered as preceding event number 4.
5	00:10:35	00:10:45	00:10:37	Event arrived out-of-order and outside tolerance of 8 seconds. So, event time is adjusted to maximum of out-of-order tolerance.

Can these settings delay output of my job?

Yes. By default, out-of-order policy is set to zero (00 minutes and 00 seconds). If you change the default, your job's first output is delayed by this value (or greater).

If one of the partitions of your inputs doesn't receive events, you should expect your output to be delayed by the late arrival policy value. To learn why, read the InputPartition error section below.

I see LateInputEvents messages in my activity log

These messages are shown to inform you that events have arrived late and are either dropped or adjusted according to your configuration. You can ignore these messages if you have configured late arrival policy appropriately.

Example of this message is:

```
{"message Time":"2019-02-04 17:11:52Z","error":null, "message":"First Occurred: 02/04/2019 17:11:48 | Resource Name: ASAjob | Message: Source 'ASAjob' had 24 data errors of kind 'LateInputEvent' between processing times '2019-02-04T17:10:49.7250696Z' and '2019-02-04T17:11:48.7563961Z'. Input event with application timestamp '2019-02-04T17:05:51.6050000' and arrival time '2019-02-04T17:10:44.3090000' was sent later than configured tolerance.", "type":"DiagnosticMessage", "correlation ID":"49efa148-4asd-4fe0-869d-a40ba4d7ef3b"}
```

I see InputPartitionNotProgressing in my activity log

Your input source (Event Hub/IoT Hub) likely has multiple partitions. Azure Stream Analytics produces output for time stamp t1 only after all the partitions that are combined are at least at time t1. For example, assume that the query reads from an event hub partition that has two partitions. One of the partitions, P1, has events until time t1. The other partition, P2, has events until time t1 + x. Output is then produced until time t1. But if there's an explicit Partition by PartitionId clause, both the partitions progress independently.

When multiple partitions from the same input stream are combined, the late arrival tolerance is the maximum amount of time that every partition waits for new data. If there is one partition in your Event Hub, or if IoT Hub doesn't receive inputs, the timeline for that partition doesn't progress until it reaches the late arrival tolerance threshold. This delays your output by the late arrival tolerance threshold. In such cases, you may see the following message:

```
{"message Time": "2/3/2019 8:54:16 PM UTC", "message": "Input Partition [2] does not have additional data for more than [5] minute(s). Partition will not progress until either events arrive or late arrival threshold is met.", "type": "InputPartitionNotProgressing", "correlation ID": "2328d411-52c7-4100-ba01-1e860c757fc2"}
```

This message to inform you that at least one partition in your input is empty and will delay your output by the late arrival threshold. To overcome this, it is recommended you either:

1. Ensure all partitions of your Event Hub/IoT Hub receive input.
2. Use Partition by PartitionID clause in your query.

Next steps

- [Time handling considerations](#)
- [Metrics available in Stream Analytics](#)

Set up alerts for Azure Stream Analytics jobs

6/21/2019 • 2 minutes to read • [Edit Online](#)

It's important to monitor your Azure Stream Analytics job to ensure the job is running continuously without any problems. This article describes how to set up alerts for common scenarios that should be monitored.

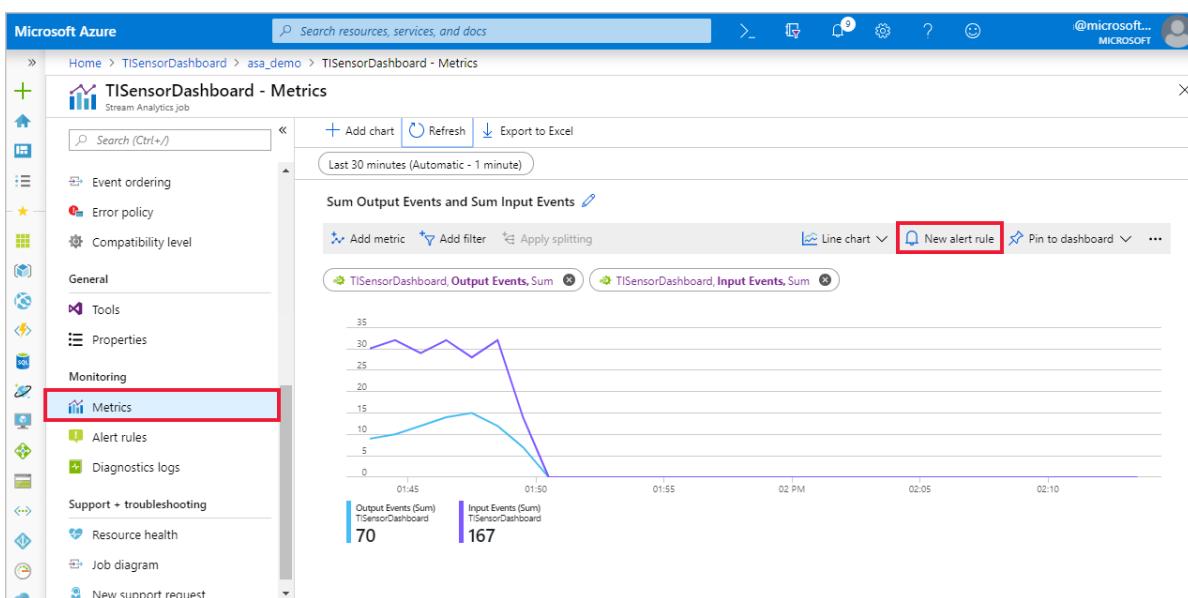
You can define rules on metrics from Operation Logs data through the portal, as well as [programmatically](#).

Set up alerts in the Azure portal

Get alerted when a job stops unexpectedly

The following example demonstrates how to set up alerts for when your job enters a failed state. This alert is recommended for all jobs.

1. In the Azure portal, open the Stream Analytics job you want to create an alert for.
2. On the **Job** page, navigate to the **Monitoring** section.
3. Select **Metrics**, and then **New alert rule**.



4. Your Stream Analytics job name should automatically appear under **RESOURCE**. Click **Add condition**, and select **All Administrative operations** under **Configure signal logic**.

The screenshot shows the 'Create rule' interface with the 'Configure signal logic' dialog open. The dialog lists various administrative operations under 'Input Sources Received':

Input Sources Received	Metric	Platform
All Administrative operations	Activity Log	Administrative
Delete Stream Analytics Job (streamingjobs)	Activity Log	Administrative
Read Stream Analytics Job (streamingjobs)	Activity Log	Administrative
Start Stream Analytics Job (streamingjobs)	Activity Log	Administrative
Stop Stream Analytics Job (streamingjobs)	Activity Log	Administrative
Write Stream Analytics Job (streamingjobs)	Activity Log	Administrative
All Security operations	Activity Log	Security
Delete Stream Analytics Job (streamingjobs)	Activity Log	Security
Read Stream Analytics Job (streamingjobs)	Activity Log	Security
Start Stream Analytics Job (streamingjobs)	Activity Log	Security
Stop Stream Analytics Job (streamingjobs)	Activity Log	Security
Write Stream Analytics Job (streamingjobs)	Activity Log	Security

5. Under **Configure signal logic**, change **Event Level** to **All** and change **Status** to **Failed**. Leave **Event initiated by** blank and select **Done**.

The screenshot shows the 'Configure signal logic' dialog with the 'Alert logic' section highlighted. The 'Event Level' dropdown is set to 'All', the 'Status' dropdown is set to 'Failed', and the 'Event initiated by' input field is empty.

6. Select an existing action group or create a new group. In this example, a new action group called **TIDashboardGroupActions** was created with an **Emails** action that sends an email to users with the **Owner** Azure Resource Manager Role.

The screenshot shows two dialogs: 'Add action group' and 'Email/SMS/Push/Voice'.

Add action group:

- Action group name: TIDashboardGroupActions
- Short name: TIDashboard
- Subscription: Azure Big Data Analytics Content Team Subscription
- Resource group: asa_demo

Email/SMS/Push/Voice:

- Name: Emails
- Email: email@example.com
- Checkboxes: Email Azure Resource Manager Role (selected), Owner
- SMS: Country code 1, Phone number 1234567890
- Carrier charges may apply.
- Azure app Push Notifications
- *Push notifications only supported for Service Health alerts
- Learn about the connecting to your Azure resources using the Azure app.
- Email: email@example.com
- This is the email you use to log into your Azure account.
- Voice

7. The **RESOURCE**, **CONDITION**, and **ACTION GROUPS** should each have an entry. Note that in order for the alerts to fire, the conditions defined need to be met. For example, you can measure a metric's average value of over the last 15 minutes, every 5 minutes.

The screenshot shows the 'Create rule' interface under 'Rules management'. It is divided into three main sections: **RESOURCE**, **HIERARCHY**, **CONDITION**, and **ACTION GROUPS**.

- RESOURCE:** A tree view showing 'TISensorDashboard' selected. A red box highlights the entire section.
- HIERARCHY:** Shows a folder icon and a link to 'asa_demo...'.
- CONDITION:** A list containing a single item: 'Whenever the Administrative Activity Log "All Administrative operations" has "any" level wi...'. A red box highlights this item.
- ACTION GROUPS:** A table with one row. Column headers are 'ACTION GROUP NAME' and 'ACTION GROUP TYPE'. The row contains 'TIDashboardGroupActions' and '1 Email Azure Resource Manager Role'. A red box highlights this row.

Add an **Alert rule name**, **Description**, and your **Resource Group** to the **ALERT DETAILS** and click **Create alert rule** to create the rule for your Stream Analytics job.

The screenshot shows the 'Create rule' interface under 'Rules management' with the 'ALERT DETAILS' section highlighted by a red box.

ALERT DETAILS:

- Alert rule name:** Sample ASA Alert
- Description:** Alerts when a job enters Failed state
- Save alert to resource group:** asa_demo

Below these fields are two buttons: **Yes** and **No** for 'Enable rule upon creation'. A note below states: 'It can take up to 5 minutes for an Activity log alert rule to become active.'

At the bottom is a large blue button labeled **Create alert rule**.

Scenarios to monitor

The following alerts are recommended for monitoring the performance of your Stream Analytics job. These metrics should be evaluated every minute over the last 5-minute period.

Metric	Condition	Time Aggregation	Threshold	Corrective Actions
SU% Utilization	Greater than	Maximum	80	There are multiple factors that increase SU% Utilization. You can scale with query parallelization or increase the number of streaming units. For more information, see Leverage query parallelization in Azure Stream Analytics .
Runtime errors	Greater than	Total	0	Examine the activity or diagnostic logs and make appropriate changes to the inputs, query, or outputs.
Watermark delay	Greater than	Maximum	When average value of this metric over the last 15 minutes is greater than late arrival tolerance (in seconds). If you have not modified the late arrival tolerance, the default is set to 5 seconds.	Try increasing the number of SUs or parallelizing your query. For more information on SUs, see Understand and adjust Streaming Units . For more information on parallelizing your query, see Leverage query parallelization in Azure Stream Analytics .
Input deserialization errors	Greater than	Total	0	Examine the activity or diagnostic logs and make appropriate changes to the input. For more information on diagnostic logs, see Troubleshoot Azure Stream Analytics using diagnostics logs

Get help

For more detail on configuring alerts in the Azure portal, see [Receive alert notifications](#).

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)

- [Azure Stream Analytics Management REST API Reference](#)

How to start an Azure Stream Analytics job

4/4/2019 • 2 minutes to read • [Edit Online](#)

You can start your Azure Stream Analytics job using the Azure portal, Visual Studio, and PowerShell. When you start a job, you select a time for the job to start creating output. Azure portal, Visual Studio, and PowerShell each have different methods for setting the start time. Those methods are described below.

Start options

The three following options are available to start a job. Note that all the times mentioned below are the ones specified in **TIMESTAMP BY**. If **TIMESTAMP BY** is not specified, arrival time will be used.

- **Now:** Makes the starting point of the output event stream the same as when the job is started. If a temporal operator is used (e.g. time window, LAG or JOIN), Azure Stream Analytics will automatically look back at the data in the input source. For instance, if you start a job “Now” and if your query uses a 5-minutes Tumbling Window, Azure Stream Analytics will seek data from 5 minutes ago in the input. The first possible output event would have a timestamp equal to or greater than the current time, and ASA guarantees that all input events that may logically contribute to the output has been accounted for. For example, no partial windowed aggregates are generated. It’s always the complete aggregated value.
- **Custom:** You can choose the starting point of the output. Similarly to the **Now** option, Azure Stream Analytics will automatically read the data prior to this time if a temporal operator is used
- **When last stopped.** This option is available when the job was previously started, but was stopped manually or failed. When choosing this option Azure Stream Analytics will use the last output time to restart the job so no data is lost. Similarly to previous options, Azure Stream Analytics will automatically read the data prior to this time if a temporal operator is used. Since several input partitions may have different time, the earliest stop time of all partitions is used, as a result some duplicates may be seen in the output. More information about exactly-once processing are available on the page [Event Delivery Guarantees](#).

Azure portal

Navigate to your job in the Azure portal and select **Start** on the overview page. Select a **Job output start time** and then select **Start**.

Choose one of the options for **Job output start time**. The options are *Now*, *Custom*, and, if the job was previously run, *When last stopped*. See above for more information about these options.

Visual Studio

In the job view, select the green arrow button to start the job. Set the **Job Output Start Mode** and select **Start**. The job status will change to **Running**.

There are three options for **Job Output Start Mode**: *JobStartTime*, *CustomTime*, and *LastOutputEventTime*. If this property is absent, the default is *JobStartTime*. See above for more information about these options.

PowerShell

Use the following cmdlet to start your job using PowerShell:

```
Start-AzStreamAnalyticsJob ` 
    -ResourceGroupName $resourceGroup ` 
    -Name $jobName ` 
    -OutputStartMode 'JobStartTime'
```

There are three options for **OutputStartMode**: *JobStartTime*, *CustomTime*, and *LastOutputEventTime*. If this property is absent, the default is *JobStartTime*. See above for more information about these options.

For more information on the `Start-AzStreamAnalyticsJob` cmdlet, view the [Start-AzStreamAnalyticsJob reference](#).

Next steps

- [Quickstart: Create a Stream Analytics job by using the Azure portal](#)
- [Quickstart: Create a Stream Analytics job using Azure PowerShell](#)
- [Quickstart: Create a Stream Analytics job by using the Azure Stream Analytics tools for Visual Studio](#)

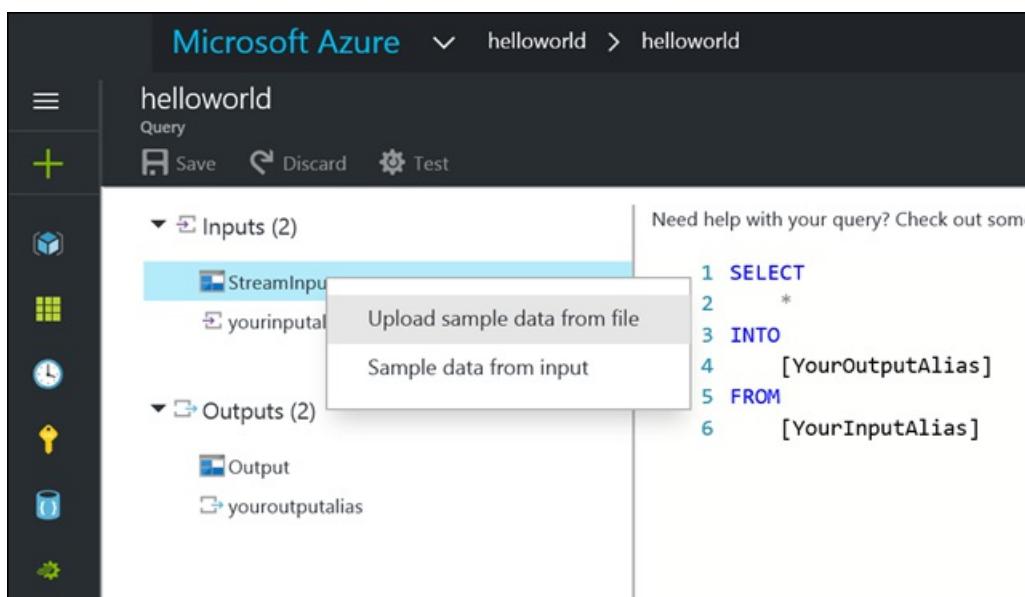
Test a Stream Analytics query with sample data

6/23/2019 • 2 minutes to read • [Edit Online](#)

By using Azure Stream Analytics, you can sample data from inputs or upload sample data to test queries in the Azure portal without starting or stopping a job.

Upload sample data and test the query

1. Sign in to the Azure portal.
2. Locate your existing Stream Analytics job and select it.
3. On the Stream Analytics job page, under the **Job Topology** heading, select **Query** to open the Query editor window.
4. To test your query you can then either sample data from a live input, or upload from a file. The data must be serialized in JSON, CSV or AVRO. Sample input must be encoded in UTF-8 and not compressed. Only comma (,) delimiter is supported for testing CSV input on portal.
 - a. Using live input: right-click on any of your inputs. Then select **Sample data from input**. In the next screen, you can set the duration of the sample.
 - b. Using file: right-click on any of your inputs. Then select **Upload sample data from file**.



5. After the sampling or upload is complete, select **Test** to test this query against the sample data you have provided.

The screenshot shows the Microsoft Azure Stream Analytics interface. At the top, there's a navigation bar with 'Microsoft Azure', 'helloworld > helloworld'. Below it is a toolbar with icons for Save, Discard, and Test. The 'Test' button is highlighted with a red box. On the left, there's a sidebar with icons for various operations like Create, Delete, and Edit. The main area shows a query editor with sections for Inputs (1) and Outputs (1). Under Inputs, there's a StreamInput node. Under Outputs, there's an Output node. To the right of the inputs, there's a code preview window with the following SQL-like query:

```

1 SELECT *
2 INTO Output
3 FROM StreamInput
4
5 SELECT [device-id] AS Device, AVG(temp) AS AverageTemp
6 INTO HighAvgTempOutput
7 FROM StreamInput
8 GROUP BY TumblingWindow(minute, 1), [device-id] HAVING AverageTemp > 100
9
10

```

6. If you need the test output for later use, the output of your query is displayed in the browser with a link to the download results.
7. Iteratively modify your query and test it again to see how the output changes.

This screenshot shows the Microsoft Azure Stream Analytics interface with a more complex query. It has two outputs: 'Output' and 'highavgtempoutput'. The 'highavgtempoutput' tab is active, showing the results of the query. The results table has columns 'DEVICE' and 'AVERAGETEMP', with one row showing DEVICE as 2 and AVERAGETEMP as 120.

DEVICE	AVERAGETEMP
2	120

When you use multiple outputs in a query, the results are shown on separate tabs, and you can easily toggle between them.

8. After you verify the results shown in the browser, **Save** your query. Then **Start** the job, and let it process the incoming events.

Next steps

[Azure Stream Analytics Query Language Reference](#)

Stream Analytics and Power BI: A real-time analytics dashboard for streaming data

6/13/2019 • 7 minutes to read • [Edit Online](#)

Azure Stream Analytics enables you to take advantage of one of the leading business intelligence tools, [Microsoft Power BI](#). In this article, you learn how to create business intelligence tools by using Power BI as an output for your Azure Stream Analytics jobs. You also learn how to create and use a real-time dashboard.

This article continues from the Stream Analytics [real-time fraud detection](#) tutorial. It builds on the workflow created in that tutorial and adds a Power BI output so that you can visualize fraudulent phone calls that are detected by a Streaming Analytics job.

You can watch [a video](#) that illustrates this scenario.

Prerequisites

Before you start, make sure you have the following:

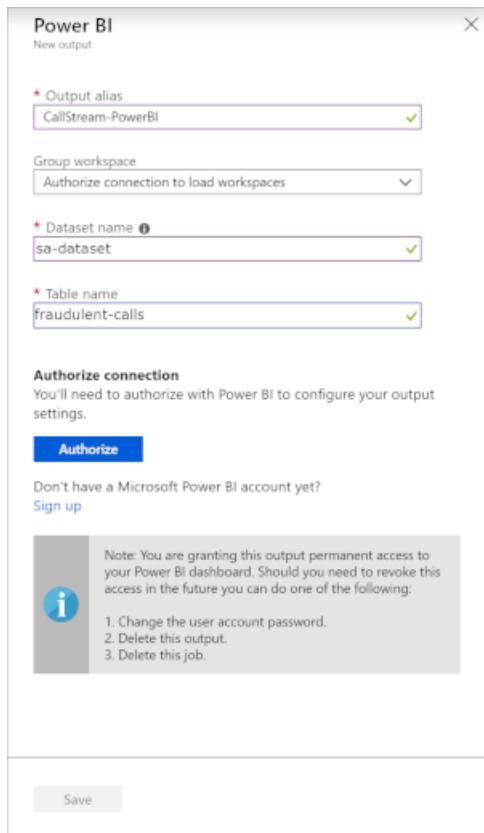
- An Azure account.
- An account for Power BI. You can use a work account or a school account.
- A completed version of the [real-time fraud detection](#) tutorial. The tutorial includes an app that generates fictitious telephone-call metadata. In the tutorial, you create an event hub and send the streaming phone call data to the event hub. You write a query that detects fraudulent calls (calls from the same number at the same time in different locations).

Add Power BI output

In the real-time fraud detection tutorial, the output is sent to Azure Blob storage. In this section, you add an output that sends information to Power BI.

1. In the Azure portal, open the Streaming Analytics job that you created earlier. If you used the suggested name, the job is named `sa_fraudetection_job_demo`.
2. On the left menu, select **Outputs** under **Job topology**. Then, select **+ Add** and choose **Power BI** from the dropdown menu.
3. Select **+ Add > Power BI**. Then fill the form with the following details and select **Authorize**:

SETTING	SUGGESTED VALUE
Output alias	CallStream-PowerBI
Dataset name	sa-dataset
Table name	fraudulent-calls



WARNING

If Power BI has a dataset and table that have the same names as the ones that you specify in the Stream Analytics job, the existing ones are overwritten. We recommend that you do not explicitly create this dataset and table in your Power BI account. They are automatically created when you start your Stream Analytics job and the job starts pumping output into Power BI. If your job query doesn't return any results, the dataset and table are not created.

4. When you select **Authorize**, a pop-up window opens and you are asked to provide credentials to authenticate to your Power BI account. Once the authorization is successful, **Save** the settings.
5. Click **Create**.

The dataset is created with the following settings:

- **defaultRetentionPolicy: BasicFIFO** - Data is FIFO, with a maximum of 200,000 rows.
- **defaultMode: pushStreaming** - The dataset supports both streaming tiles and traditional report-based visuals (also known as push).

Currently, you can't create datasets with other flags.

For more information about Power BI datasets, see the [Power BI REST API](#) reference.

Write the query

1. Close the **Outputs** blade and return to the job blade.
2. Click the **Query** box.
3. Enter the following query. This query is similar to the self-join query you created in the fraud-detection tutorial. The difference is that this query sends results to the new output you created (`CallStream-PowerBI`).

NOTE

If you did not name the input `CallStream` in the fraud-detection tutorial, substitute your name for `CallStream` in the **FROM** and **JOIN** clauses in the query.

```
/* Our criteria for fraud:  
Calls made from the same caller to two phone switches in different locations (for example, Australia and  
Europe) within five seconds */  
  
SELECT System.Timestamp AS WindowEnd, COUNT(*) AS FraudulentCalls  
INTO "CallStream-PowerBI"  
FROM "CallStream" CS1 TIMESTAMP BY CallRecTime  
JOIN "CallStream" CS2 TIMESTAMP BY CallRecTime  
  
/* Where the caller is the same, as indicated by IMSI (International Mobile Subscriber Identity) */  
ON CS1.CallingIMSI = CS2.CallingIMSI  
  
/* ...and date between CS1 and CS2 is between one and five seconds */  
AND DATEDIFF(ss, CS1, CS2) BETWEEN 1 AND 5  
  
/* Where the switch location is different */  
WHERE CS1.SwitchNum != CS2.SwitchNum  
GROUP BY TumblingWindow(Duration(second, 1))
```

4. Click **Save**.

Test the query

This section is optional, but recommended.

1. If the TelcoStreaming app is not currently running, start it by following these steps:
 - Open Command Prompt.
 - Go to the folder where the telcogenerator.exe and modified telcodatagen.exe.config files are.
 - Run the following command:
`telcodatagen.exe 1000 .2 2`
2. On the **Query** page for your Stream Analytics job, click the dots next to the `CallStream` input and then select **Sample data from input**.
3. Specify that you want three minutes' worth of data and click **OK**. Wait until you're notified that the data has been sampled.
4. Click **Test** and review the results.

Run the job

1. Make sure the TelcoStreaming app is running.
2. Navigate to the **Overview** page for your Stream Analytics job and select **Start**.

The screenshot shows the Microsoft Azure Stream Analytics job 'sa_fraudetection_job_demo'. At the top, there's a search bar labeled 'Search (Ctrl+ /)'. Below it are three buttons: 'Start' (highlighted with a red box), 'Stop', and 'Delete'. A status indicator at the bottom says 'Created'.

Your Streaming Analytics job starts looking for fraudulent calls in the incoming stream. The job also creates the dataset and table in Power BI and starts sending data about the fraudulent calls to them.

Create the dashboard in Power BI

1. Go to [Powerbi.com](#) and sign in with your work or school account. If the Stream Analytics job query outputs results, you see that your dataset is already created:

The screenshot shows the Power BI workspace. On the left, there's a sidebar with 'Favorites', 'Recent', and 'Apps'. Below that is a search bar for 'Shared with me'. Under 'Workspaces', 'My Workspace' is selected. In the main area, there are sections for 'DASHBOARDS' (with a note 'You have no dashboards'), 'REPORTS' (with a note 'You have no reports'), 'WORKBOOKS' (with a note 'You have no workbooks'), and 'DATASETS'. The 'DATASETS' section contains one item: 'sa-dataset', which is highlighted with a red box.

2. In your workspace, click **+ Create**.

The screenshot shows the Power BI workspace with a different layout. At the top, there's a toolbar with icons for 'New', 'My Workspace', 'Search', 'Settings', 'Down arrow', 'Help', and a user profile. Below the toolbar, there's a search bar labeled 'Search content...'. At the bottom of the screen, there are tabs for 'Dashboards' (which is underlined in yellow), 'Reports', 'Workbooks', and 'Datasets'. On the far right of the main area, there's a button labeled '+ Create' which is highlighted with a red box.

3. Create a new dashboard and name it **Fraudulent Calls**.

Create dashboard

Dashboard name

Create **Cancel**

4. At the top of the window, click **Add tile**, select **CUSTOM STREAMING DATA**, and then click **Next**.

Add tile

Select source

MEDIA



WEB CONTENT



IMAGE



TEXT BOX



VIDEO

REAL-TIME DATA



CUSTOM STREAMING DATA

Next **Cancel**

5. Under **YOUR DATASETS**, select your dataset and then click **Next**.

Add a custom streaming data tile

Choose a streaming dataset

+ Add streaming dataset

YOUR DATASETS

sa-dataset

6. Under **Visualization Type**, select **Card**, and then in the **Fields** list, select **fraudulentcalls**.

Add a custom streaming data tile

Choose a streaming dataset > Visualization design

Visualization Type

Card



Fields

fraudulentcalls



+ Add value

[Manage datasets](#)

Back

Next

Cancel

7. Click **Next**.

8. Fill in tile details like a title and subtitle.

Tile details

* Required

Details

Display title and subtitle

Title

Subtitle

Functionality

Set custom link

Link type

External link

Link to a dashboard or report in the current workspace

URL *

Open custom link in the same tab?

Yes

No

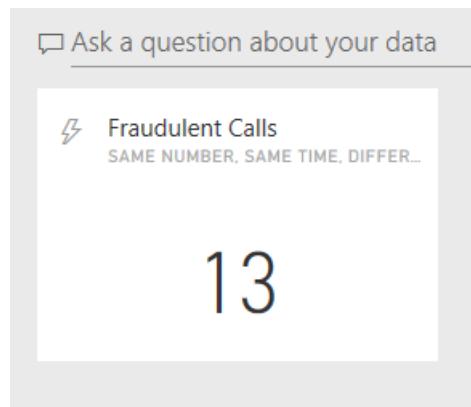
[Restore default](#)

[Technical Details](#)

[Back](#) [Apply](#) [Cancel](#)

9. Click **Apply**.

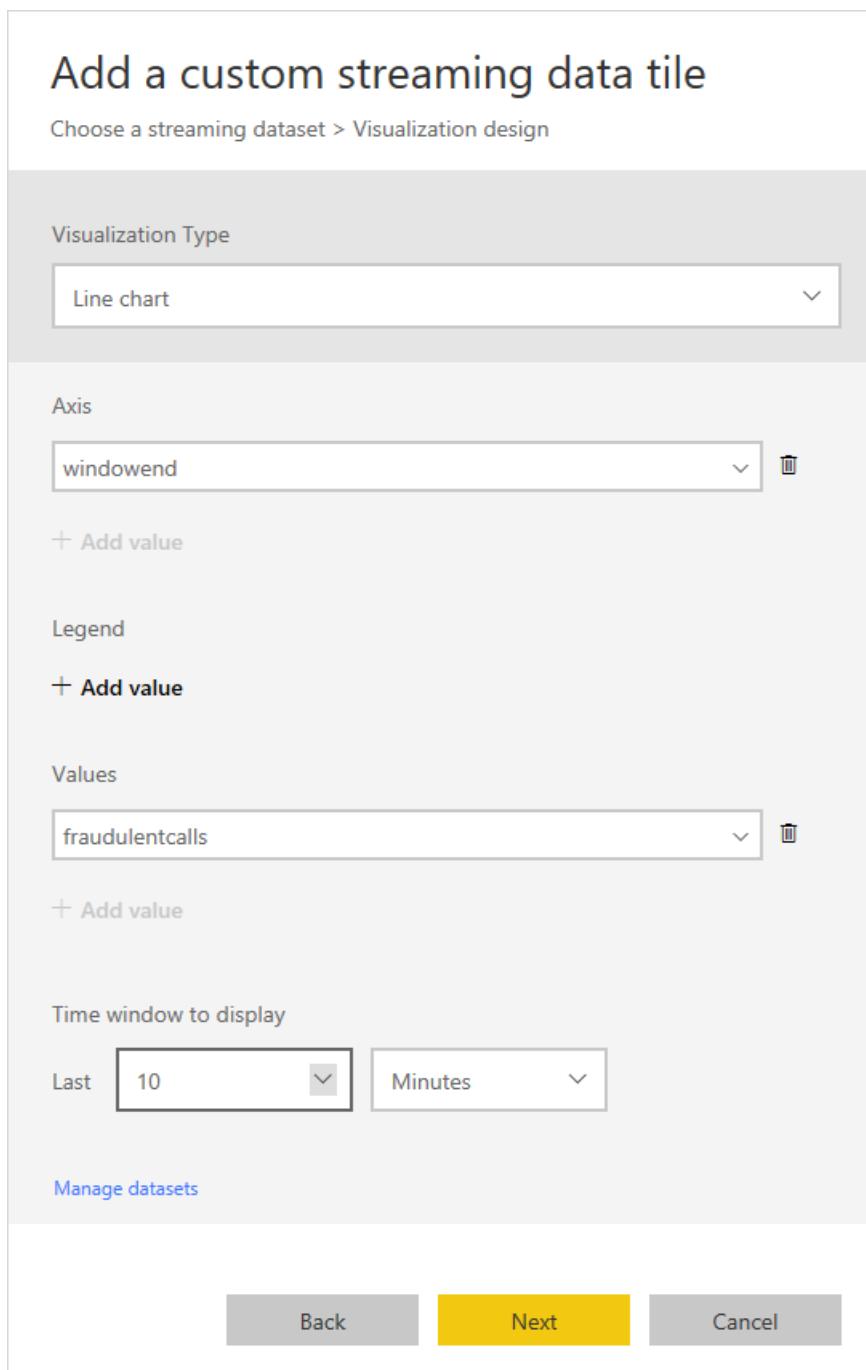
Now you have a fraud counter!



10. Follow the steps again to add a tile (starting with step 4). This time, do the following:

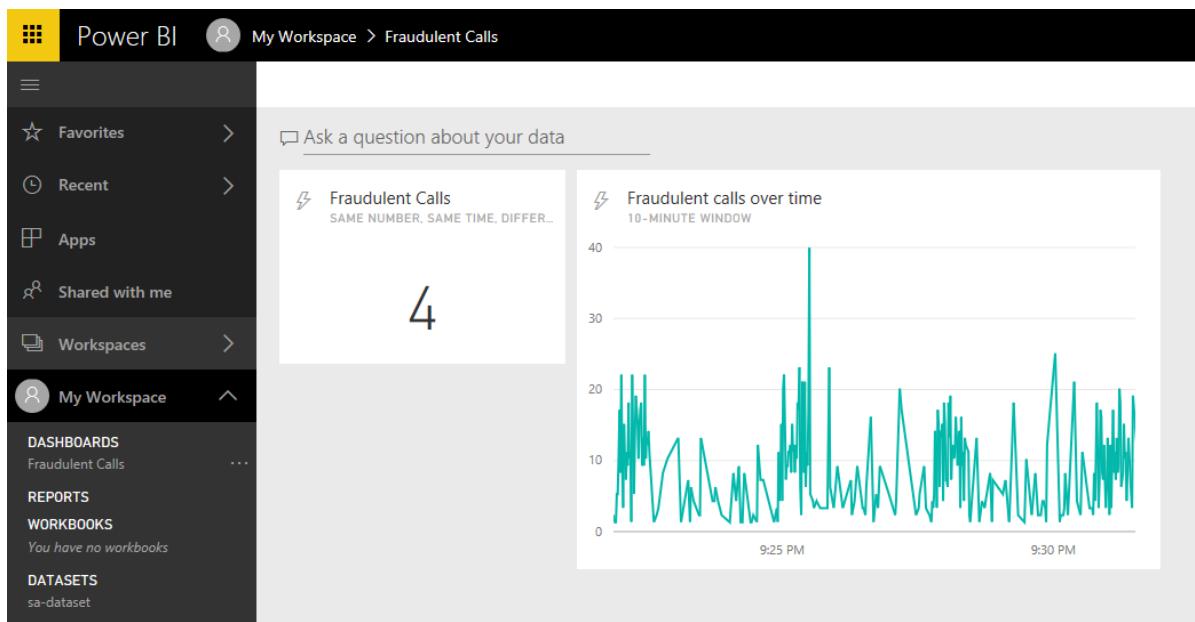
- When you get to **Visualization Type**, select **Line chart**.

- Add an axis and select **windowend**.
- Add a value and select **fraudulentcalls**.
- For **Time window to display**, select the last 10 minutes.



11. Click **Next**, add a title and subtitle, and click **Apply**.

The Power BI dashboard now gives you two views of data about fraudulent calls as detected in the streaming data.



Learn more about Power BI

This tutorial demonstrates how to create only a few kinds of visualizations for a dataset. Power BI can help you create other customer business intelligence tools for your organization. For more ideas, see the following resources:

- For another example of a Power BI dashboard, watch the [Getting Started with Power BI](#) video.
- For more information about configuring Streaming Analytics job output to Power BI and using Power BI groups, review the [Power BI](#) section of the [Stream Analytics outputs](#) article.
- For information about using Power BI generally, see [Dashboards in Power BI](#).

Learn about limitations and best practices

Currently, Power BI can be called roughly once per second. Streaming visuals support packets of 15 KB. Beyond that, streaming visuals fail (but push continues to work). Because of these limitations, Power BI lends itself most naturally to cases where Azure Stream Analytics does a significant data load reduction. We recommend using a Tumbling window or Hopping window to ensure that data push is at most one push per second, and that your query lands within the throughput requirements.

You can use the following equation to compute the value to give your window in seconds:

$$\left\lceil \frac{\text{EntityCount} * 60 * 60}{\text{Throughput}} \right\rceil$$

For example:

- You have 1,000 devices sending data at one-second intervals.
- You are using the Power BI Pro SKU that supports 1,000,000 rows per hour.
- You want to publish the amount of average data per device to Power BI.

As a result, the equation becomes:

$$\left\lceil \frac{\text{EntityCount} * 60 * 60}{\text{Throughput}} \right\rceil = \left\lceil \frac{1,000 * 60 * 60}{1,000,000} \right\rceil = [3.6] = 4$$

Given this configuration, you can change the original query to the following:

```
SELECT  
    MAX(hmdt) AS hmdt,  
    MAX(temp) AS temp,  
    System.TimeStamp AS time,  
    dspl  
INTO "CallStream-PowerBI"  
FROM  
    Input TIMESTAMP BY time  
GROUP BY  
    TUMBLINGWINDOW(ss,4),  
    dspl
```

Renew authorization

If the password has changed since your job was created or last authenticated, you need to reauthenticate your Power BI account. If Azure Multi-Factor Authentication is configured on your Azure Active Directory (Azure AD) tenant, you also need to renew Power BI authorization every two weeks. If you don't renew, you could see symptoms such as a lack of job output or an [Authenticate user error](#) in the operation logs.

Similarly, if a job starts after the token has expired, an error occurs and the job fails. To resolve this issue, stop the job that's running and go to your Power BI output. To avoid data loss, select the **Renew authorization** link, and then restart your job from the **Last Stopped Time**.

After the authorization has been refreshed with Power BI, a green alert appears in the authorization area to reflect that the issue has been resolved.

Get help

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics query language reference](#)
- [Azure Stream Analytics Management REST API reference](#)

Stop or delete your Azure Stream Analytics job

6/21/2019 • 2 minutes to read • [Edit Online](#)

Azure Stream Analytics jobs can be easily stopped or deleted through the Azure portal, Azure PowerShell, Azure SDK for .Net, or REST API. A Stream Analytics job cannot be recovered once it has been deleted.

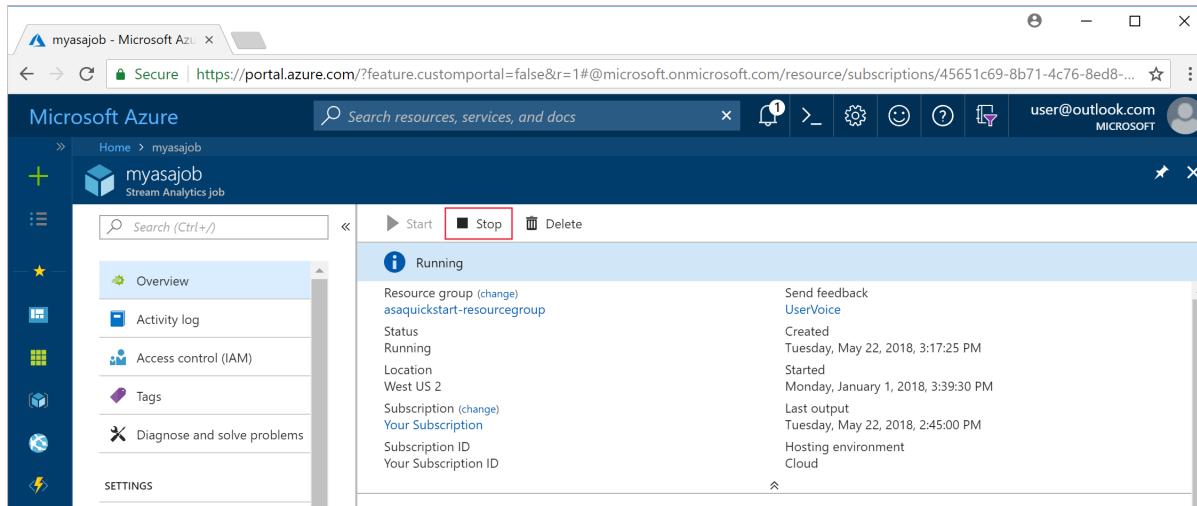
NOTE

When you stop your Stream Analytics job, the data persists only in the input and output storage, such as Event Hubs or Azure SQL Database. If you are required to remove data from Azure, be sure to follow the removal process for the input and output resources of your Stream Analytics job.

Stop a job in Azure portal

When you stop a job, the resources are deprovisioned and it stops processing events. Charges related to this job are also stopped. However all your configuration are kept and you can restart the job later

1. Sign in to the [Azure portal](#).
2. Locate your running Stream Analytics job and select it.
3. On the Stream Analytics job page, select **Stop** to stop the job.

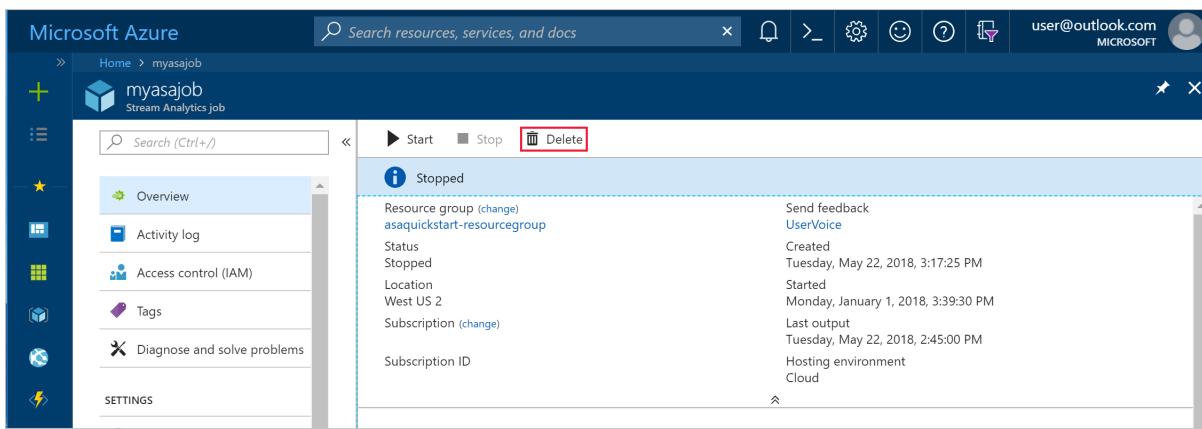


Delete a job in Azure portal

WARNING

A Stream Analytics job cannot be recovered once it has been deleted.

1. Sign in to the Azure portal.
2. Locate your existing Stream Analytics job and select it.
3. On the Stream Analytics job page, select **Delete** to delete the job.



Stop or delete a job using PowerShell

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

To stop a job using PowerShell, use the [Stop-AzStreamAnalyticsJob](#) cmdlet. To delete a job using PowerShell, use the [Remove-AzStreamAnalyticsJob](#) cmdlet.

Stop or delete a job using Azure SDK for .NET

To stop a job using Azure SDK for .NET, use the [StreamingJobsOperationsExtensions.BeginStop](#) method. To delete a job using Azure SDK for .NET, [StreamingJobsOperationsExtensions.BeginDelete](#) method.

Stop or delete a job using REST API

To stop a job using REST API, refer to the [Stop](#) method. To delete a job using REST API, refer to the [Delete](#) method.

Guarantee Stream Analytics job reliability during service updates

6/21/2019 • 2 minutes to read • [Edit Online](#)

Part of being a fully managed service is the capability to introduce new service functionality and improvements at a rapid pace. As a result, Stream Analytics can have a service update deploy on a weekly (or more frequent) basis. No matter how much testing is done there is still a risk that an existing, running job may break due to the introduction of a bug. If you are running mission critical jobs, these risks need to be avoided. You can reduce this risk by following Azure's **paired region** model.

How do Azure paired regions address this concern?

Stream Analytics guarantees jobs in paired regions are updated in separate batches. As a result there is a sufficient time gap between the updates to identify potential issues and remediate them.

With the exception of Central India (whose paired region, South India, does not have Stream Analytics presence), the deployment of an update to Stream Analytics would not occur at the same time in a set of paired regions. Deployments in multiple regions **in the same group** may occur **at the same time**.

The article on **availability and paired regions** has the most up-to-date information on which regions are paired.

It is recommended to deploy identical jobs to both paired regions. You should then [monitor these jobs](#) to get notified when something unexpected happens. If one of these jobs ends up in a [Failed state](#) after a Stream Analytics service update, you can contact customer support to help identify the root cause. You should also fail over any downstream consumers to the healthy job output.

Next steps

- [Introduction to Stream Analytics](#)
- [Get started with Stream Analytics](#)
- [Scale Stream Analytics jobs](#)
- [Stream Analytics query language reference](#)
- [Stream Analytics management REST API reference](#)

Authenticate Stream Analytics to Azure Data Lake Storage Gen1 using managed identities

4/11/2019 • 5 minutes to read • [Edit Online](#)

Azure Stream Analytics supports managed identity authentication with Azure Data Lake Storage (ADLS) Gen1 output. The identity is a managed application registered in Azure Active Directory that represents a given Stream Analytics job, and can be used to authenticate to a targeted resource. Managed identities eliminate the limitations of user-based authentication methods, like needing to reauthenticate due to password changes or user token expirations that occur every 90 days. Additionally, managed identities help with the automation of Stream Analytics job deployments that output to Azure Data Lake Storage Gen1.

This article shows you three ways to enable managed identity for an Azure Stream Analytics job that outputs to an Azure Data Lake Storage Gen1 through the Azure portal, Azure Resource Manager template deployment, and Azure Stream Analytics tools for Visual Studio.

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

Azure portal

1. Start by creating a new Stream Analytics job or by opening an existing job in Azure portal. From the menu bar located on the left side of the screen, select **Managed Identity** located under **Configure**.

The screenshot shows the Azure Stream Analytics job configuration page for 'AsaJobWithIdentity'. On the left, there's a sidebar with various navigation links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Locks, Job topology, Inputs, Functions, Query, Outputs, Configure, Storage account settings, Scale, Locale, Event ordering, Error policy, Compatibility level, and Managed Identity. The 'Managed Identity' link is highlighted with a red box.

2. Select **Use System-assigned Managed Identity** from the window that appears on the right. Click **Save** to a service principal for the identity of the Stream Analytics job in Azure Active Directory. The life cycle of the newly created identity will be managed by Azure. When the Stream Analytics job is deleted, the associated identity (that is, the service principal) is automatically deleted by Azure.

When the configuration is saved, the Object ID (OID) of the service principal is listed as the Principal ID as shown below:

The screenshot shows a configuration dialog for 'Use System-assigned Managed Identity (preview)'. It has a checked checkbox labeled 'Use System-assigned Managed Identity (preview)' and a text input field labeled 'Principal ID' containing the value '294c7a3b-0a2b-4c3b-8ffc-b5a61c8896ea'. There is also a small blue 'Save' button icon.

The service principal has the same name as the Stream Analytics job. For example, if the name of your job is **MyASAJob**, the name of the service principal created is also **MyASAJob**.

3. In the output properties window of the ADLS Gen1 output sink, click the Authentication mode drop-down and select **Managed Identity **.
4. Fill out the rest of the properties. To learn more about creating an ADLS output, see [Create a Data lake Store output with stream analytics](#). When you are finished, click **Save**.

Data Lake Storage Gen1

New output

* Output alias

Provide Data Lake Storage Gen1 settings manually
 Select Data Lake Storage Gen1 from your subscriptions

Subscription

Account name

* Path prefix pattern
Example: cluster1/logs/{date}/{time}

Date format

Time format

* Event serialization format

Encoding

Format

Authentication mode

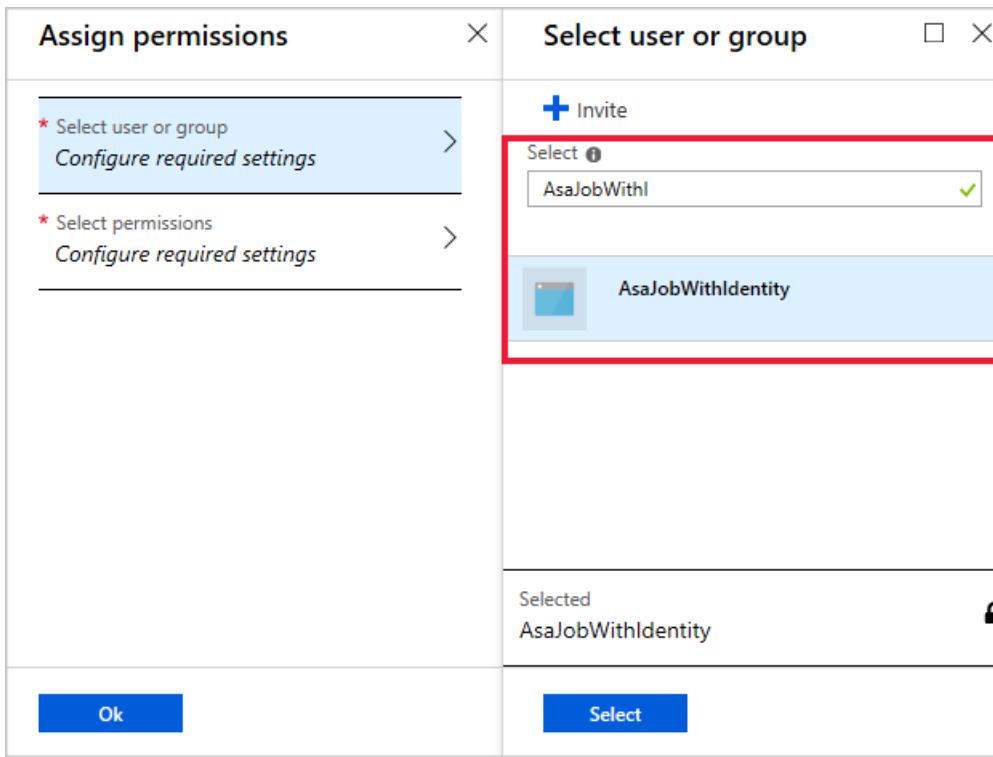
5. Navigate to the Overview page of your ADLS Gen1 and click on **Data explorer**.

The screenshot shows the 'adlsmsidemo' Data Lake Storage Gen1 overview page. At the top, there's a navigation bar with 'Home > adlsmsidemo'. Below it is a summary card for 'adlsmsidemo' showing a blue lightning bolt icon, the storage type 'Data Lake Storage Gen1', a search bar, and a 'Delete' button. To the right of the delete button is a red box highlighting the 'Data explorer' button. Below the summary card is a section titled 'Essentials' with a blue folder icon and the text 'adlsmsidemo Pay-as-You-Go'. At the bottom of the page are links for 'Overview' and 'Activity log'.

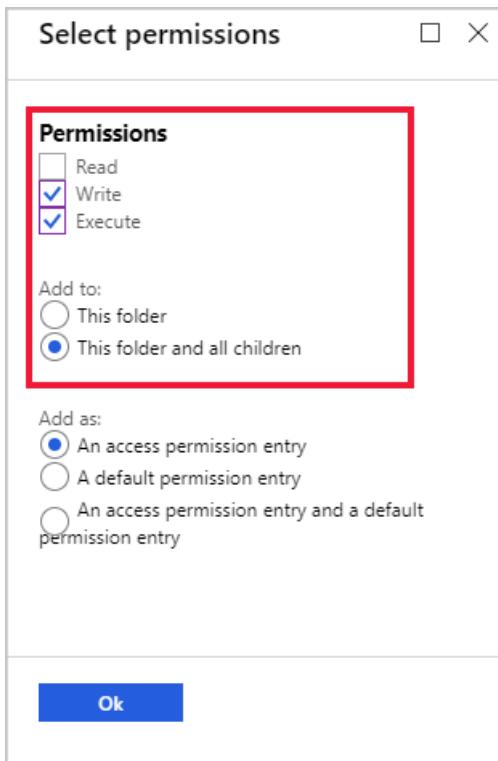
6. In the Data explorer pane, select **Access** and click **Add** in the Access pane.

The screenshot shows the 'adlsmsidemo' Data Explorer access pane. At the top, there's a header with 'adlsmsidemo Data Lake Storage Gen1' and various navigation buttons like 'Filter', 'New folder', 'Upload', 'Access' (which is highlighted with a red box), 'Rename folder', 'Folder properties', 'Delete folder', and 'Refresh'. Below the header is a table with one row labeled 'adlsmsidemo'. At the bottom of the pane is a section titled 'Your permission' with a 'Save' button.

7. In the text box on the **Select user or group** pane, type the name of the service principal. Remember that the name of the service principal is also the name of the corresponding Stream Analytics job. As you begin typing the principal name, it will appear below the text box. Choose the desired service principal name and click **Select**.



8. In the **Permissions** pane, check the **Write** and **Execute** permissions and assign it to **This Folder and all children**. Then click **Ok**.



9. The service principal is listed under **Assigned Permissions** on the **Access** pane as shown below. You can now go back and start your Stream Analytics job.

Access

/ (Folder)

+ Add Save Discard Advanced

Successfully assigned permissions to AsaJobWithIdentity
191 succeeded, 0 failed.

Your permissions
effective permissions on this folder are: Read,Write,Execute.

You have superuser privileges on this account.

Owners	Read	Write	Execute
--------	------	-------	---------

Assigned permissions

AsaJobWithIdentity	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
--------------------	--------------------------	-------------------------------------	-------------------------------------

To learn more about Data Lake Storage Gen1 file system permissions, see [Access Control in Azure Data Lake Storage Gen1](#).

Stream Analytics tools for Visual Studio

1. In JobConfig.json, set **Use System-assigned Identity** to **True**.

Job Configuration X

Stream Analytics Job Configure Configuration

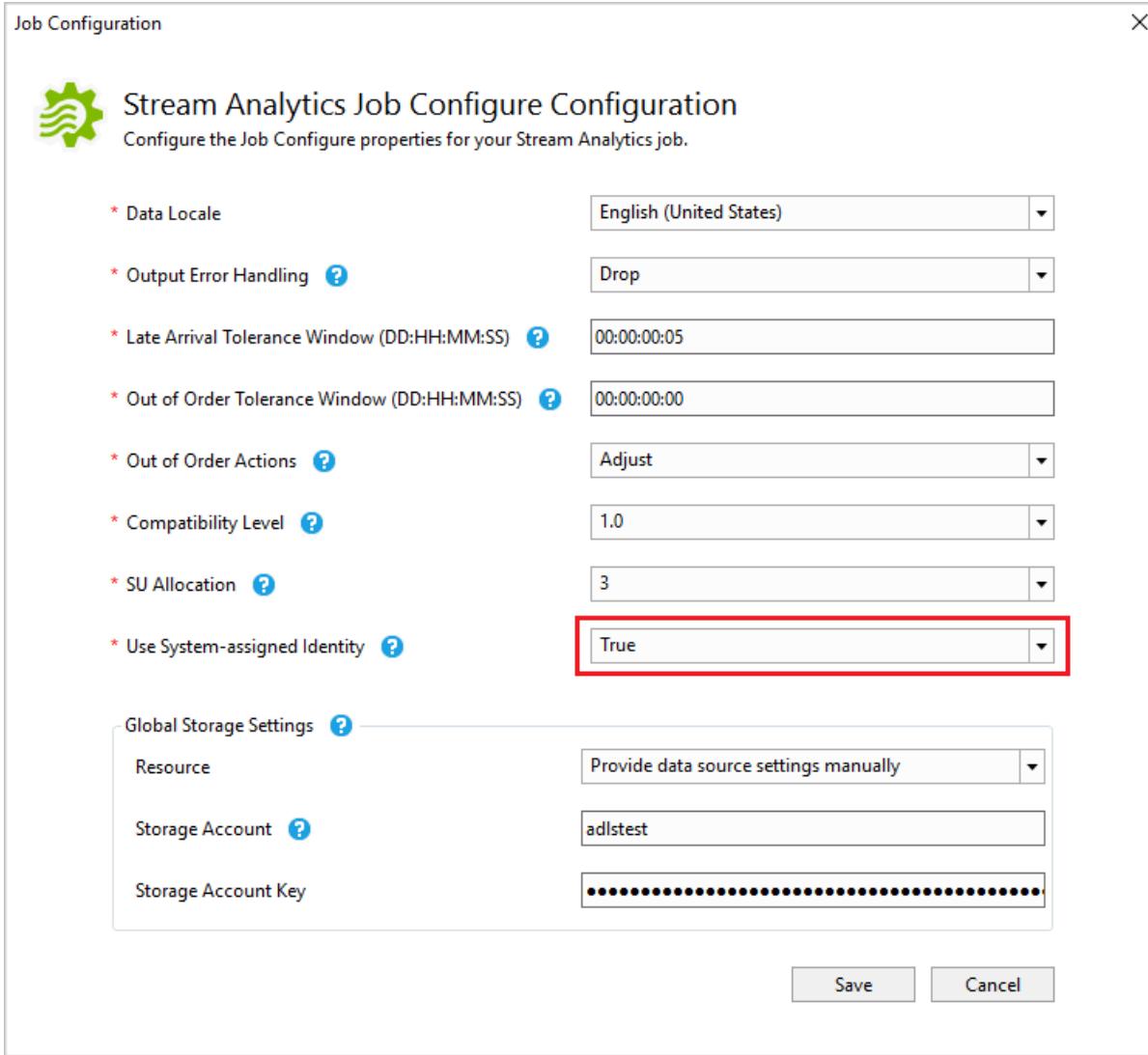
Configure the Job Configure properties for your Stream Analytics job.

* Data Locale	English (United States)
* Output Error Handling	Drop
* Late Arrival Tolerance Window (DD:HH:MM:SS)	00:00:00:05
* Out of Order Tolerance Window (DD:HH:MM:SS)	00:00:00:00
* Out of Order Actions	Adjust
* Compatibility Level	1.0
* SU Allocation	3
* Use System-assigned Identity	True

Global Storage Settings ?

Resource	Provide data source settings manually
Storage Account	adltest
Storage Account Key	*****

Save Cancel



2. In the output properties window of the ADLS Gen1 output sink, click the Authentication mode drop-down and select **Managed Identity **.

Output Configuration X

Stream Analytics Output Configuration

Configure the Output properties for your Stream Analytics job.

! Data Lake Storage Gen1 now supports Managed Identity based authentication. Please enable "Use System-assigned Identity" in job configuration for this job in order to use it.

* Output Alias	Output1
* Sink ?	Data Lake Store Gen1
* Resource	Choose data source from current account
* Subscription	<your subscription>
* Account Name	<your account name>
* Path Prefix Pattern ?	/
Date Format	yyyy/MM/dd
Time Format	HH
Authentication Mode	Managed Identity (preview)
* Event Serialization Format ?	Json
Encoding ?	UTF8
Format ?	Line separated

Save **Cancel**

3. Fill out the rest of the properties, and click **Save**.

4. Click **Submit to Azure** in the query editor.

When you submit the job, the tools do two things:

- Automatically creates a service principal for the identity of the Stream Analytics job in Azure Active Directory. The life cycle of the newly created identity will be managed by Azure. When the Stream Analytics job is deleted, the associated identity (that is, the service principal) is automatically deleted by Azure.
- Automatically set **Write** and **Execute** permissions for the ADLS Gen1 prefix path used in the job and assign it to this folder and all children.

5. You can generate the Resource Manager templates with the following property using [Stream Analytics CI.CD Nuget package](#) version 1.5.0 or above on a build machine (outside of Visual Studio). Follow the Resource Manager template deployment steps in the next section to get the service principal and grant access to the service principal via PowerShell.

Resource Manager template deployment

1. You can create a `Microsoft.StreamAnalytics/streamingjobs` resource with a managed identity by including the following property in the resource section of your Resource Manager template:

```
"Identity": {  
    "Type": "SystemAssigned",  
},
```

This property tells Azure Resource Manager to create and manage the identity for your Azure Stream Analytics job.

Sample job

```
{  
    "Name": "AsaJobWithIdentity",  
    "Type": "Microsoft.StreamAnalytics/streamingjobs",  
    "Location": "West US",  
    "Identity": {  
        "Type": "SystemAssigned",  
    },  
    "properties": {  
        "sku": {  
            "name": "standard"  
        },  
        "outputs": [  
            {  
                "name": "string",  
                "properties":{  
                    "datasource": {  
                        "type": "Microsoft.DataLake/Accounts",  
                        "properties": {  
                            "accountName": "myDataLakeAccountName",  
                            "filePathPrefix": "cluster1/logs/{date}/{time}",  
                            "dateFormat": "YYYY/MM/DD",  
                            "timeFormat": "HH",  
                            "authenticationMode": "Msi"  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

Sample job response

```
{  
    "Name": "mySAJob",  
    "Type": "Microsoft.StreamAnalytics/streamingjobs",  
    "Location": "West US",  
    "Identity": {  
        "Type": "SystemAssigned",  
        "principalId": "GUID",  
        "tenantId": "GUID",  
    },  
    "properties": {  
        "sku": {  
            "name": "standard"  
        },  
    }  
}
```

Take note of the Principal ID from the job response to grant access to the required ADLS resource.

The **Tenant ID** is the ID of the Azure Active Directory tenant where the service principal is created. The service principal is created in the Azure tenant that is trusted by the subscription.

The **Type** indicates the type of managed identity as explained in types of managed identities. Only the System Assigned type is supported.

2. Provide Access to the service principal using PowerShell. To give access to the service principal via PowerShell, execute the following command:

```
Set-AzDataLakeStoreItemAclEntry -AccountName <accountName> -Path <Path> -AceType User -Id <PrinicpalId>
-Permissions <Permissions>
```

The **PrincipalId** is the Object ID of the service principal and is listed on the portal screen once the service principal is created. If you created the job using a Resource Manager template deployment, the Object ID is listed in the Identity property of the job response.

Example

```
PS > Set-AzDataLakeStoreItemAclEntry -AccountName "adlsmssidemo" -Path / -AceType
User -Id 14c6fd67-d9f5-4680-a394-cd7df1f9bacf -Permissions WriteExecute
```

To learn more about the above PowerShell command, refer to the [Set-AzDataLakeStoreItemAclEntry](#) documentation.

Limitations

This feature doesn't support the following:

1. **Multi-tenant access:** The Service principal created for a given Stream Analytics job will reside on the Azure Active Directory tenant on which the job was created, and cannot be used against a resource that resides on a different Azure Active Directory tenant. Therefore, you can only use MSI on ADLS Gen 1 resources that are within the same Azure Active Directory tenant as your Azure Stream Analytics job.
2. **User Assigned Identity:** is not supported. This means the user is not able to enter their own service principal to be used by their Stream Analytics job. The service principal is generated by Azure Stream Analytics.

Next steps

- [Create a Data lake Store output with stream analytics](#)
- [Test Stream Analytics queries locally with Visual Studio](#)
- [Test live data locally using Azure Stream Analytics tools for Visual Studio](#)

Use reference data from a SQL Database for an Azure Stream Analytics job (Preview)

5/29/2019 • 7 minutes to read • [Edit Online](#)

Azure Stream Analytics supports Azure SQL Database as a source of input for reference data. You can use SQL Database as reference data for your Stream Analytics job in the Azure portal and in Visual Studio with Stream Analytics tools. This article demonstrates how to do both methods.

Azure portal

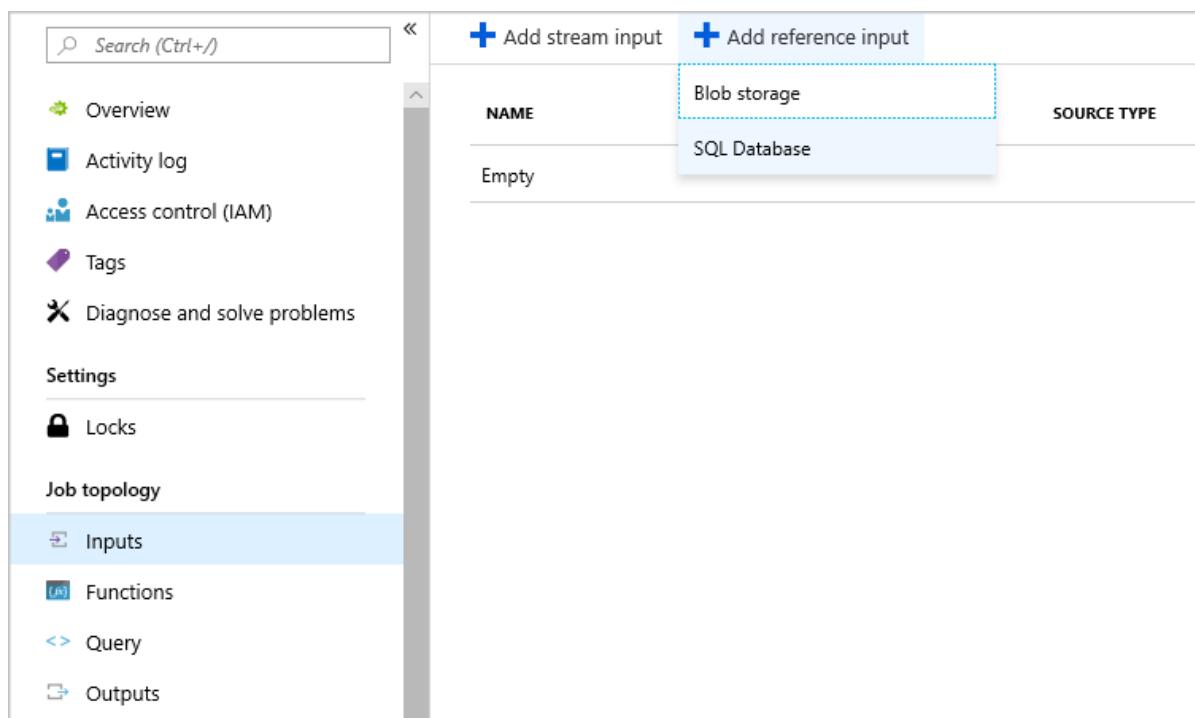
Use the following steps to add Azure SQL Database as a reference input source using the Azure portal:

Portal prerequisites

1. Create a Stream Analytics job.
2. Create a storage account to be used by the Stream Analytics job.
3. Create your Azure SQL Database with a data set to be used as reference data by the Stream Analytics job.

Define SQL Database reference data input

1. In your Stream Analytics job, select **Inputs** under **Job topology**. Click **Add reference input** and choose **SQL Database**.



The screenshot shows the Azure Stream Analytics Job Topology interface. On the left, there's a navigation sidebar with links like Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. Below that are sections for Settings, Locks, and Job topology, with Inputs selected. The main area is titled 'Add stream input' and 'Add reference input'. Under 'Add reference input', there's a table with two rows: 'NAME' (containing 'Empty') and 'SOURCE TYPE' (containing 'SQL Database'). A blue dashed box highlights the 'Empty' row. At the bottom right of the main area, there's a 'Next Step' button.

2. Fill out the Stream Analytics Input Configurations. Choose the database name, server name, username and password. If you want your reference data input to refresh periodically, choose "On" to specify the refresh rate in DD:HH:MM. If you have large data sets with a short refresh rate, you can use a [delta query](#).

SQL Database

New input

A storage account is needed to store the snapshots of the reference data from Azure SQL Database. Click here to add a storage account for your Stream Analytics job. →

* Input alias

Provide SQL Database settings manually
 Select SQL Database from your subscriptions

Subscription

Database

Server name

* Username

* Password

Refresh periodically

Delta query

Days Hours Minutes

Snapshot query

```

1  SELECT Id, Name, FullName
2  From dbo.Device
3  FOR SYSTEM_TIME AS OF @snapshotTime --Optional, available if table Device is temporal

```

Save  The selected resource and the stream analytics job are located in different regions. You will be billed to move data between regions.

- Test the snapshot query in the SQL query editor. For more information, see [Use the Azure portal's SQL query editor to connect and query data](#)

Specify storage account in Job config

Navigate to **Storage account settings** under **Configure** and select **Add storage account**.

sidsadbash - Storage account settings
Stream Analytics job

Search (Ctrl+/

This storage account is used for storing content related to your stream analytics job, such as SQL reference data snapshots.

Storage account settings

Add storage account

Overview

Activity log

Access control (IAM)

Tags

Diagnose and solve problems

Settings

Locks

Job topology

Inputs

Functions

Query

Outputs

Configure

- Storage account settings**
- Scale**
- Locale**
- Event ordering**
- Error policy**
- Compatibility level**

Start the job

Once you have configured other inputs, outputs, and query, you can start the Stream Analytics job.

Tools for Visual Studio

Use the following steps to add Azure SQL Database as a reference input source using Visual Studio:

Visual Studio prerequisites

1. [Install the Stream Analytics tools for Visual Studio](#). The following versions of Visual Studio are supported:
 - Visual Studio 2015
 - Visual Studio 2019
2. Become familiar with the [Stream Analytics tools for Visual Studio](#) quickstart.
3. Create a storage account.

Create a SQL Database table

Use SQL Server Management Studio to create a table to store your reference data. See [Design your first Azure SQL database using SSMS](#) for details.

The example table used in the following example was created from the following statement:

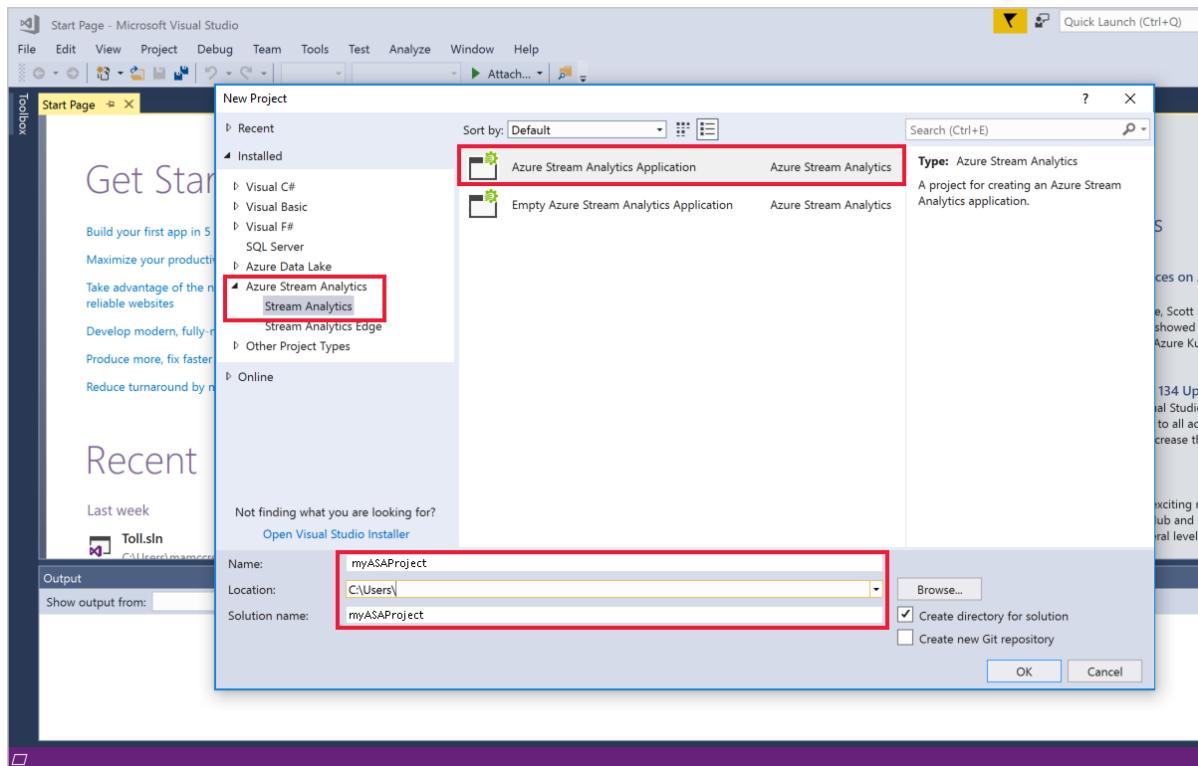
```
create table chemicals(Id Bigint,Name Nvarchar(max),FullName Nvarchar(max));
```

Choose your subscription

1. In Visual Studio, on the **View** menu, select **Server Explorer**.
2. Right click on **Azure**, select **Connect to Microsoft Azure Subscription**, and sign in with your Azure account.

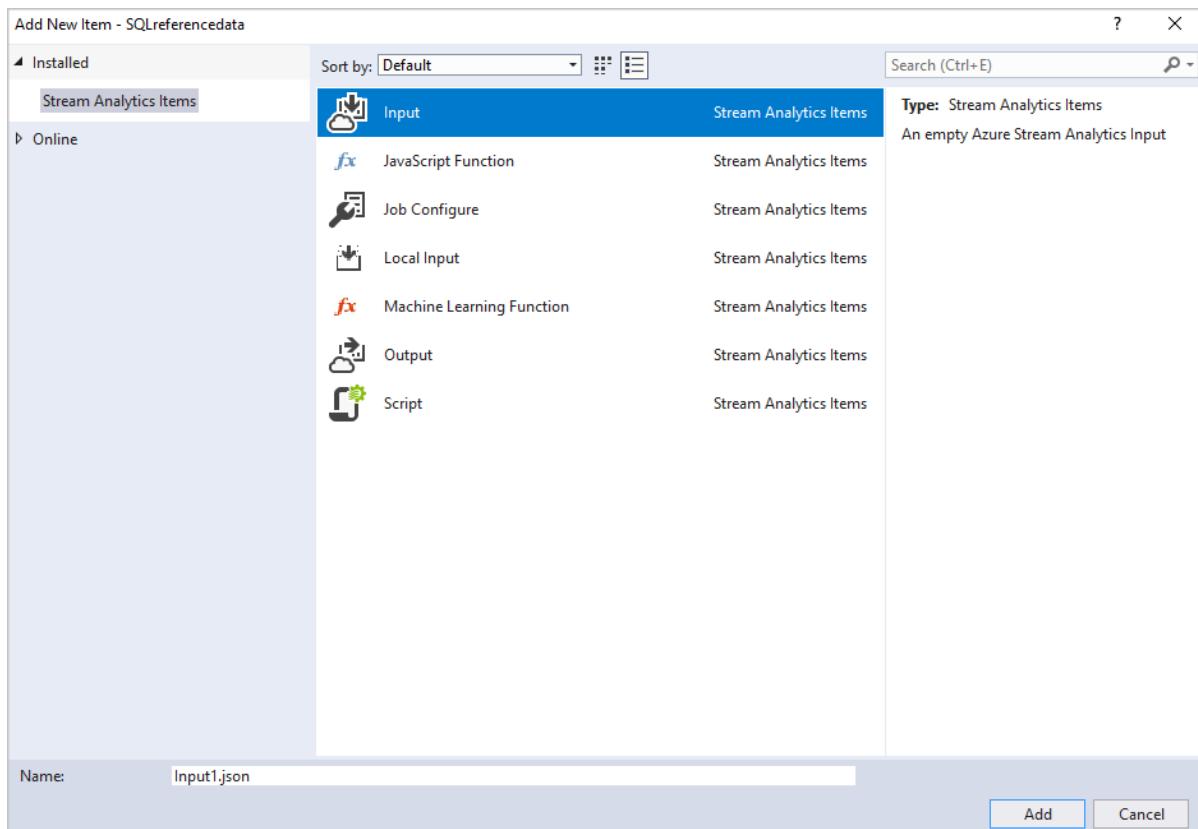
Create a Stream Analytics project

1. Select **File > New Project**.
2. In the templates list on the left, select **Stream Analytics**, and then select **Azure Stream Analytics Application**.
3. Enter the project **Name**, **Location**, and **Solution name**, and select **OK**.



Define SQL Database reference data input

1. Create a new input.



2. Double-click **Input.json** in the **Solution Explorer**.

3. Fill out the **Stream Analytics Input Configuration**. Choose the database name, server name, refresh type and refresh rate. Specify the refresh rate in the format DD:HH:MM .

Input Configuration X

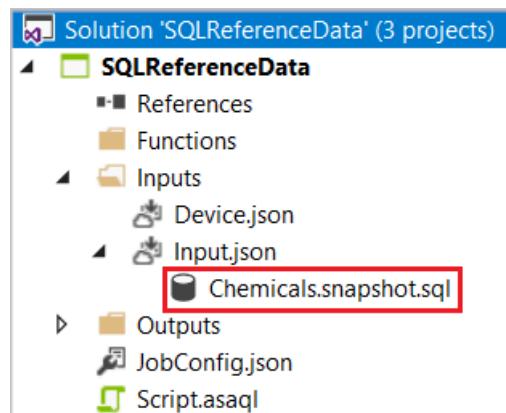
Stream Analytics Input Configuration

Configure the Input properties for your Stream Analytics job.

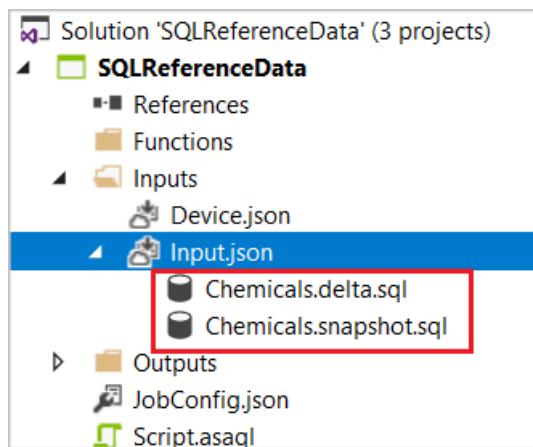
! A storage account is needed to store the snapshots of the reference data from Azure SQL database. Please configure 'Global storage account' in the JobConfig file.

* Input Alias	Chemicals
* Source Type ?	Reference data
* Source ?	SQL Database
* Resource	Provide data source settings manually
* Database	SqlreferenceDataForASA
* Server Name	sqlreferencedata.database.windows.net
* User Name	guizha
* Password	*****
* Refresh Type ?	<div style="border: 1px solid red; padding: 5px;"> Execute only once Execute only once (selected) Execute periodically Execute periodically with delta query </div>
Refresh Rate	<input type="button" value="Save"/> <input type="button" value="Cancel"/>

If you choose "Execute only once" or "Execute periodically", one SQL CodeBehind file named **[Input Alias].snapshot.sql** is generated in the project under the **Input.json** file node.



If you choose "Refresh Periodically with Delta", two SQL CodeBehind files will be generated: **[Input Alias].snapshot.sql** and **[Input Alias].delta.sql**.



4. Open the SQL file in the editor and write the SQL query.
5. If you are using Visual Studio 2019, and you have installed SQL Server Data tools, you can test the query by clicking **Execute**. A wizard window will pop up to help you connect to the SQL database and the query result will appear in the window at the bottom.

Specify storage account

Open **JobConfig.json** to specify the storage account for storing SQL reference snapshots.

Job Configuration

Stream Analytics Job Configure Configuration

Configure the Job Configure properties for your Stream Analytics job.

* Data Locale English (United States)

* Output Error Handling Drop

* Late Arrival Tolerance Window (DD:HH:MM:SS) 00:00:00:05

* Out of Order Tolerance Window (DD:HH:MM:SS) 00:00:00:00

* Out of Order Actions Adjust

* Compatibility Level 1.0

* SU Allocation 3

* Use System-assigned Identity False

Global Storage Settings

Resource Choose data source from current account

Subscription

Storage Account

Storage Account Key [REDACTED]

Save Cancel

Test locally and deploy to Azure

Before deploying the job to Azure, you can test the query logic locally against live input data. For more

information on this feature, see [Test live data locally using Azure Stream Analytics tools for Visual Studio \(Preview\)](#). When you're done testing, click **Submit to Azure**. Reference the [Create a Stream Analytics using the Azure Stream Analytics tools for Visual Studio](#) quickstart to learn how to start the job.

Delta query

When using the delta query, [temporal tables in Azure SQL Database](#) are recommended.

1. Create a temporal table in Azure SQL Database.

```
CREATE TABLE DeviceTemporal
(
    [DeviceId] int NOT NULL PRIMARY KEY CLUSTERED
    , [GroupDeviceId] nvarchar(100) NOT NULL
    , [Description] nvarchar(100) NOT NULL
    , [ValidFrom] datetime2 (0) GENERATED ALWAYS AS ROW START
    , [ValidTo] datetime2 (0) GENERATED ALWAYS AS ROW END
    , PERIOD FOR SYSTEM_TIME (ValidFrom, ValidTo)
)
WITH (SYSTEM_VERSIONING = ON (HISTORY_TABLE = dbo.DeviceHistory)); -- DeviceHistory table will be
used in Delta query
```

2. Author the snapshot query.

Use the **@snapshotTime** parameter to instruct the Stream Analytics runtime to obtain the reference data set from SQL database temporal table valid at the system time. If you don't provide this parameter, you risk obtaining an inaccurate base reference data set due to clock skews. An example of full snapshot query is shown below:

```
SELECT DeviceId, GroupDeviceId, [Description]
FROM dbo.DeviceTemporal
FOR SYSTEM_TIME AS OF @snapshotTime
```

3. Author the delta query.

This query retrieves all of the rows in your SQL database that were inserted or deleted within a start time, **@deltaStartTime**, and an end time **@deltaEndTime**. The delta query must return the same columns as the snapshot query, as well as the column **operation**. This column defines if the row is inserted or deleted between **@deltaStartTime** and **@deltaEndTime**. The resulting rows are flagged as **1** if the records were inserted, or **2** if deleted.

For records that were updated, the temporal table does bookkeeping by capturing an insertion and deletion operation. The Stream Analytics runtime will then apply the results of the delta query to the previous snapshot to keep the reference data up to date. An example of delta query is show below:

```
SELECT DeviceId, GroupDeviceId, Description, 1 as _operation_
FROM dbo.DeviceTemporal
WHERE ValidFrom BETWEEN @deltaStartTime AND @deltaEndTime -- records inserted
UNION
SELECT DeviceId, GroupDeviceId, Description, 2 as _operation_
FROM dbo.DeviceHistory -- table we created in step 1
WHERE ValidTo BETWEEN @deltaStartTime AND @deltaEndTime -- record deleted
```

Note that Stream Analytics runtime may periodically run the snapshot query in addition to the delta query to store checkpoints.

Test your query

It is important to verify that your query is returning the expected dataset that the Stream Analytics job will use as reference data. To test your query, go to Input under Job Topology section on portal. You can then select Sample Data on your SQL Database Reference input. After the sample becomes available, you can download the file and check to see if the data being returned is as expected. If you want to optimize your development and test iterations, it is recommended to use the [Stream Analytics tools for Visual Studio](#). You can also use any other tool of your preference to first ensure the query is returning the right results from your Azure SQL Database and then use that in your Stream Analytics job.

FAQs

Will I incur additional cost by using SQL reference data input in Azure Stream Analytics?

There are no additional [cost per streaming unit](#) in the Stream Analytics job. However, the Stream Analytics job must have an associated Azure storage account. The Stream Analytics job queries the SQL DB (during job start and refresh interval) to retrieve the reference data set and stores that snapshot in the storage account. Storing these snapshots will incur additional charges detailed in the [pricing page](#) for Azure storage account.

How do I know reference data snapshot is being queried from SQL DB and used in the Azure Stream Analytics job?

There are two metrics filtered by Logical Name (under Metrics Azure Portal) which you can use to monitor the health of the SQL database reference data input.

- InputEvents: This metric measures the number of records loaded in from the SQL database reference data set.
- InputEventBytes: This metric measures the size of the reference data snapshot loaded in memory of the Stream Analytics job.

The combination of both of these metrics can be used to infer if the job is querying the SQL database to fetch the reference data set and then loading it to memory.

Will I require a special type of Azure SQL Database?

Azure Stream Analytics will work with any type of Azure SQL Database. However, it is important to understand that the refresh rate set for your reference data input could impact your query load. To use the delta query option, it is recommended to use temporal tables in Azure SQL Database.

Why does Azure Stream Analytics store snapshots in Azure Storage account?

Stream Analytics guarantees exactly once event processing and at least once delivery of events. In cases where transient issues impact your job, a small amount of replay is necessary to restore state. To enable replay, it is required to have these snapshots stored in an Azure Storage account. For more information on checkpoint replay, see [Checkpoint and replay concepts in Azure Stream Analytics jobs](#).

Next steps

- [Using reference data for lookups in Stream Analytics](#)
- [Quickstart: Create a Stream Analytics job by using the Azure Stream Analytics tools for Visual Studio](#)
- [Test live data locally using Azure Stream Analytics tools for Visual Studio \(Preview\)](#)

Real-time Twitter sentiment analysis in Azure Stream Analytics

4/19/2019 • 13 minutes to read • [Edit Online](#)

IMPORTANT

Twitter application creation is no longer available through [apps.twitter.com](#). This tutorial is in the process of being updated to include the new Twitter API.

Learn how to build a sentiment analysis solution for social media analytics by bringing real-time Twitter events into Azure Event Hubs. You can then write an Azure Stream Analytics query to analyze the data and either store the results for later use or use a dashboard and [Power BI](#) to provide insights in real time.

Social media analytics tools help organizations understand trending topics. Trending topics are subjects and attitudes that have a high volume of posts in social media. Sentiment analysis, which is also called *opinion mining*, uses social media analytics tools to determine attitudes toward a product, idea, and so on.

Real-time Twitter trend analysis is a great example of an analytics tool, because the hashtag subscription model enables you to listen to specific keywords (hashtags) and develop sentiment analysis of the feed.

Scenario: Social media sentiment analysis in real time

A company that has a news media website is interested in gaining an advantage over its competitors by featuring site content that is immediately relevant to its readers. The company uses social media analysis on topics that are relevant to readers by doing real-time sentiment analysis of Twitter data.

To identify trending topics in real time on Twitter, the company needs real-time analytics about the tweet volume and sentiment for key topics. In other words, the need is a sentiment analysis analytics engine that's based on this social media feed.

Prerequisites

In this tutorial, you use a client application that connects to Twitter and looks for tweets that have certain hashtags (which you can set). In order to run the application and analyze the tweets using Azure Streaming Analytics, you must have the following:

- An Azure subscription
- A Twitter account
- A Twitter application, and the [OAuth access token](#) for that application. We provide high-level instructions for how to create a Twitter application later.
- The TwitterWPFCClient application, which reads the Twitter feed. To get this application, download the [TwitterWPFCClient.zip](#) file from GitHub and then unzip the package into a folder on your computer. If you want to see the source code and run the application in a debugger, you can get the source code from [GitHub](#).

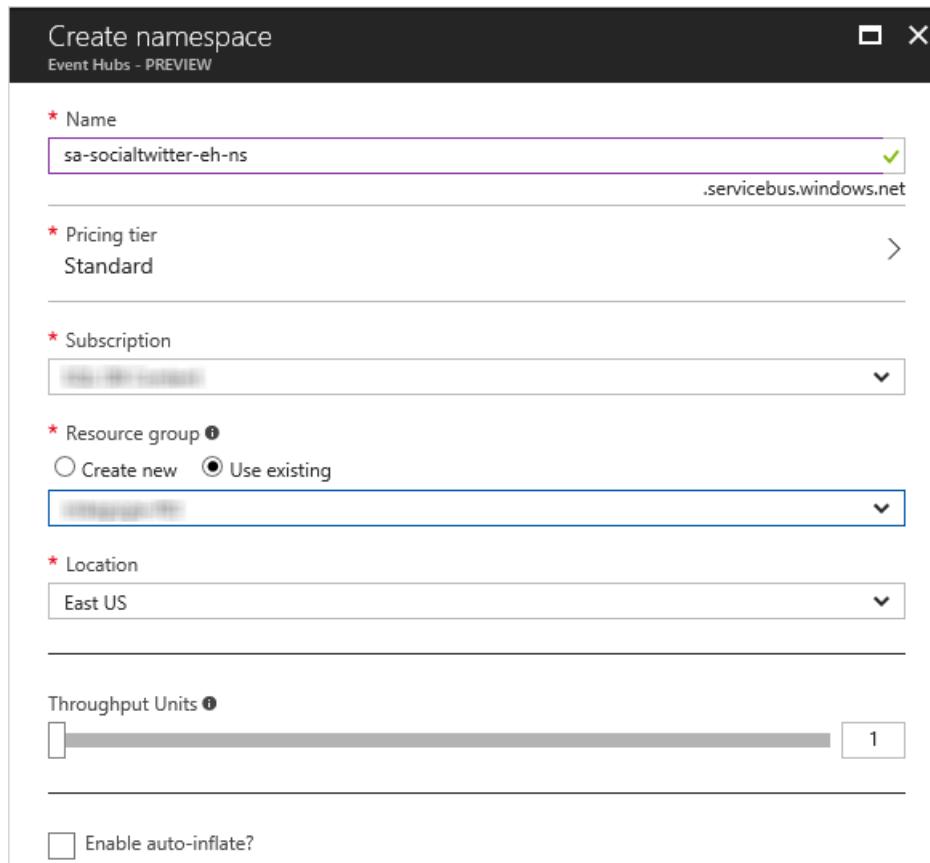
Create an event hub for Streaming Analytics input

The sample application generates events and pushes them to an Azure event hub. Azure event hubs are the preferred method of event ingestion for Stream Analytics. For more information, see the [Azure Event Hubs documentation](#).

Create an event hub namespace and event hub

In this procedure, you first create an event hub namespace, and then you add an event hub to that namespace. Event hub namespaces are used to logically group related event bus instances.

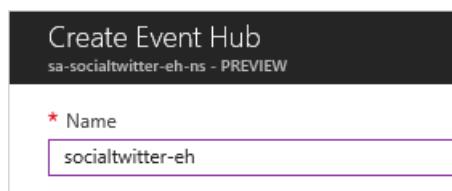
1. Log in to the Azure portal and click **Create a resource > Internet of Things > Event Hub**.
2. In the **Create namespace** blade, enter a namespace name such as `<yourusername>-socialtwitter-eh-ns`. You can use any name for the namespace, but the name must be valid for a URL and it must be unique across Azure.
3. Select a subscription and create or choose a resource group, then click **Create**.



4. When the namespace has finished deploying, find the event hub namespace in your list of Azure resources.
5. Click the new namespace, and in the namespace blade, click **+ Event Hub**.



6. Name the new event hub `socialtwitter-eh`. You can use a different name. If you do, make a note of it, because you need the name later. You don't need to set any other options for the event hub.



7. Click **Create**.

Grant access to the event hub

Before a process can send data to an event hub, the event hub must have a policy that allows appropriate access. The access policy produces a connection string that includes authorization information.

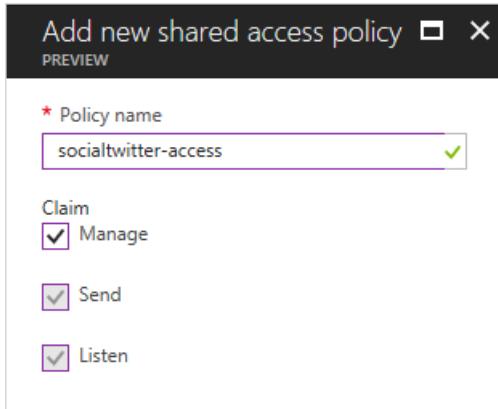
1. In the event namespace blade, click **Event Hubs** and then click the name of your new event hub.

2. In the event hub blade, click **Shared access policies** and then click **+ Add**.

NOTE

Make sure you're working with the event hub, not the event hub namespace.

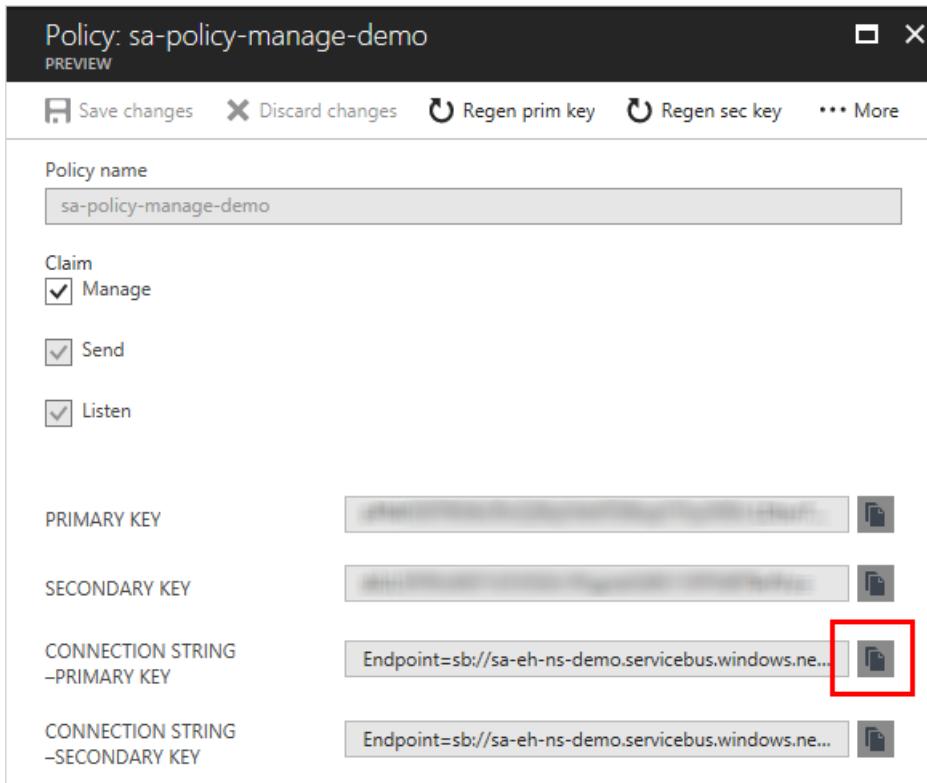
3. Add a policy named `socialtwitter-access` and for **Claim**, select **Manage**.



4. Click **Create**.

5. After the policy has been deployed, click it in the list of shared access policies.

6. Find the box labeled **CONNECTION STRING-PRIMARY KEY** and click the copy button next to the connection string.



7. Paste the connection string into a text editor. You need this connection string for the next section, after you make some small edits to it.

The connection string looks like this:

```
Endpoint=sb://YOURNAME-socialtwitter-eh-ns.servicebus.windows.net/;SharedAccessKeyName=socialtwitter-access;SharedAccessKey=Gw2NFZw6r...FxKbXaC2op6a0ZsPkI=;EntityPath=socialtwitter-eh
```

Notice that the connection string contains multiple key-value pairs, separated with semicolons: `Endpoint`, `SharedAccessKeyName`, `SharedAccessKey`, and `EntityPath`.

NOTE

For security, parts of the connection string in the example have been removed.

8. In the text editor, remove the `EntityPath` pair from the connection string (don't forget to remove the semicolon that precedes it). When you're done, the connection string looks like this:

```
Endpoint=sb://YOURNAME-socialtwitter-eh-ns.servicebus.windows.net/;SharedAccessKeyName=socialtwitter-access;SharedAccessKey=Gw2NFZw6r...FxKbXaC2op6a0ZsPkI=
```

Configure and start the Twitter client application

The client application gets tweet events directly from Twitter. In order to do so, it needs permission to call the Twitter Streaming APIs. To configure that permission, you create an application in Twitter, which generates unique credentials (such as an OAuth token). You can then configure the client application to use these credentials when it makes API calls.

Create a Twitter application

If you do not already have a Twitter application that you can use for this tutorial, you can create one. You must already have a Twitter account.

NOTE

The exact process in Twitter for creating an application and getting the keys, secrets, and token might change. If these instructions don't match what you see on the Twitter site, refer to the Twitter developer documentation.

1. Go to the [Twitter application management page](#).
2. Create a new application.
 - For the website URL, specify a valid URL. It does not have to be a live site. (You can't specify just `localhost`.)
 - Leave the callback field blank. The client application you use for this tutorial doesn't require callbacks.

Create an application

Application Details

Name *

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

3. Optionally, change the application's permissions to read-only.
4. When the application is created, go to the **Keys and Access Tokens** page.
5. Click the button to generate an access token and access token secret.

Keep this information handy, because you will need it in the next procedure.

NOTE

The keys and secrets for the Twitter application provide access to your Twitter account. Treat this information as sensitive, the same as you do your Twitter password. For example, don't embed this information in an application that you give to others.

Configure the client application

We've created a client application that connects to Twitter data using [Twitter's Streaming APIs](#) to collect tweet events about a specific set of topics. The application uses the [Sentiment140](#) open source tool, which assigns the following sentiment value to each tweet:

- 0 = negative
- 2 = neutral
- 4 = positive

After the tweet events have been assigned a sentiment value, they are pushed to the event hub that you created earlier.

Before the application runs, it requires certain information from you, like the Twitter keys and the event hub connection string. You can provide the configuration information in these ways:

- Run the application, and then use the application's UI to enter the keys, secrets, and connection string. If you do this, the configuration information is used for your current session, but it isn't saved.
- Edit the application's .config file and set the values there. This approach persists the configuration information, but it also means that this potentially sensitive information is stored in plain text on your computer.

The following procedure documents both approaches.

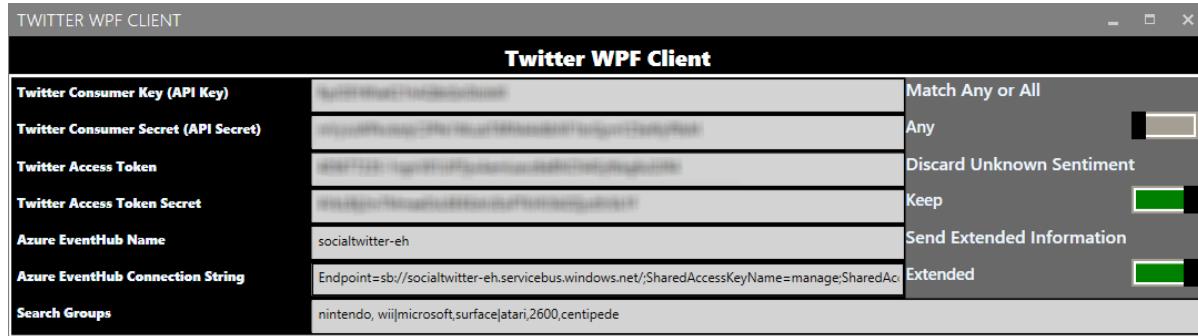
1. Make sure you've downloaded and unzipped the [TwitterWPFClient.zip](#) application, as listed in the

prerequisites.

2. To set the values at run time (and only for the current session), run the `TwitterWPFClient.exe` application.

When the application prompts you, enter the following values:

- The Twitter Consumer Key (API Key).
- The Twitter Consumer Secret (API Secret).
- The Twitter Access Token.
- The Twitter Access Token Secret.
- The connection string information that you saved earlier. Make sure that you use the connection string that you removed the `EntityPath` key-value pair from.
- The Twitter keywords that you want to determine sentiment for.



3. To set the values persistently, use a text editor to open the `TwitterWpfClient.exe.config` file. Then in the `<appSettings>` element, do this:

- Set `oauth_consumer_key` to the Twitter Consumer Key (API Key).
- Set `oauth_consumer_secret` to the Twitter Consumer Secret (API Secret).
- Set `oauth_token` to the Twitter Access Token.
- Set `oauth_token_secret` to the Twitter Access Token Secret.

Later in the `<appSettings>` element, make these changes:

- Set `EventHubName` to the event hub name (that is, to the value of the entity path).
- Set `EventHubNameConnectionString` to the connection string. Make sure that you use the connection string that you removed the `EntityPath` key-value pair from.

The `<appSettings>` section looks like the following example. (For clarity and security, we wrapped some lines and removed some characters.)

```

<appSettings>
    <!-- Twitter Streaming API settings. Replace Twitter OAuth settings with your own -->
    <add key="oauth_consumer_key" value="3K55dHKT6NY1oI38TNHd1xV0" />
    <add key="oauth_consumer_secret" value="nd3tKKPDnIAVifgVjnr...PytED10aCtaHHPP4b7" />
    <add key="oauth_token" value="11168372-MryjFocZx2V0...wsQXFV0xL3DA0kefeBcgwGGu" />
    <add key="oauth_token_secret" value="oIDCGt5U1r6d6122yx...VHVQf13dQnD9Af0GS" />

    <!-- Turn False for Troubleshooting, while false will not send to event hub-->
    <add key="AzureOn" value="true" />

    <!--Mark Rowe - Extended Functionality-->
    <!--If Sentiment Can't figure it out.. Kill it from data -->
    <add key="clear_all_with_undefined_sentiment" value="true" />

    <!--Extended Information adds "Tweet Text" and "Author"-->
    <add key="send_extended_information" value="true" />

    <!--Match Mode
        Valid Values:
        "Any"      // must match any keyword.
        "Measure" // -> TODO: Same as Any but calculates how many.
        "All"      // must match all keywords in the Search Group
    -->
    <add key="match_mode" value="any" />

    <!-- New functionality
        | (pipe) separates Search Groups
        , (comma) separates Search Terms
    -->
    <add key="twitter_keywords" value="nintendo, wii|microsoft,surface|atari,2600,centipede" />

    <!-- Service Bus specific app settings for messaging connections -->
    <add key="EventHubName" value="socialtwitter-eh" />
    <add key="EventHubConnectionString"
        value="Endpoint=sb://sa-socialtwitter-eh-ns.servicebus.windows.net;/SharedAccessKeyName=socialtwitter-access;SharedAccessKey=N+Yr1/zKnxwLd...bQys=" />
</appSettings>

```

4. If you didn't already start the application, run TwitterWpfClient.exe now.
5. Click the green start button to collect social sentiment. You see Tweet events with the **CreatedAt**, **Topic**, and **SentimentScore** values being sent to your event hub.

```

6/29/2017 5:37:16 PM microsoft 2 RT @Office365_Tech: Today is the day! Join us for a Microsoft Stream AMA in two hours. https://
6/29/2017 5:37:15 PM surface 2 @tkadlec The Surface Book builds my prod app ~25% faster which is extremely crazy
6/29/2017 5:37:14 PM microsoft 2 #vacature Developer Microsoft Dynamics AX in Breda https://t.co/7lukH3zRpV #ictvacature #tech
6/29/2017 5:37:13 PM microsoft 2 Microsoft's New IFTTT Integrations Let You Back up Photos to OneDrive
6/29/2017 5:37:12 PM microsoft 4 RT : Thrilled to announce our partnership with @Microsoft to bring innovative LI so
6/29/2017 5:37:11 PM microsoft 2 RT @Practical365: Controlling Third Party Cloud Storage Access for Microsoft Teams https://t.co

```

NOTE

If you see errors, and you don't see a stream of tweets displayed in the lower part of the window, double-check the keys and secrets. Also check the connection string (make sure that it does not include the `EntityPath` key and value.)

Create a Stream Analytics job

Now that tweet events are streaming in real time from Twitter, you can set up a Stream Analytics job to analyze these events in real time.

1. In the Azure portal, click **Create a resource** > **Internet of Things** > **Stream Analytics job**.
2. Name the job `socialtwitter-sa-job` and specify a subscription, resource group, and location.

It's a good idea to place the job and the event hub in the same region for best performance and so that you don't pay to transfer data between regions.

New Stream Analytics Job

* Job name
socialtwitter-sa-job

* Subscription
[dropdown]

* Resource group ⓘ
 Create new Use existing
[dropdown]

* Location
East US

3. Click **Create**.

The job is created and the portal displays job details.

Specify the job input

1. In your Stream Analytics job, under **Job Topology** in the middle of the job blade, click **Inputs**.

2. In the **Inputs** blade, click **+ Add** and then fill out the blade with these values:

- **Input alias:** Use the name `TwitterStream`. If you use a different name, make a note of it because you need it later.
- **Source type:** Select **Data stream**.
- **Source:** Select **Event hub**.
- **Import option:** Select **Use event hub from current subscription**.
- **Service bus namespace:** Select the event hub namespace that you created earlier (`<yourusername>-socialtwitter-eh-ns`).
- **Event hub:** Select the event hub that you created earlier (`socialtwitter-eh`).
- **Event hub policy name:** Select the access policy that you created earlier (`socialtwitter-access`).

New input

* Input alias
TwitterStream ✓

* Source Type ⓘ
Data stream

* Source ⓘ
Event hub

* Import option
Use event hub from current subscription

Service bus namespace
sa-socialtwitter-eh-ns

Event hub name
socialtwitter-eh

Event hub policy name
socialtwitter-access

Event hub consumer group ⓘ

* Event serialization format ⓘ
JSON

Encoding ⓘ
UTF-8

3. Click **Create**.

Specify the job query

Stream Analytics supports a simple, declarative query model that describes transformations. To learn more about the language, see the [Azure Stream Analytics Query Language Reference](#). This tutorial helps you author and test several queries over Twitter data.

To compare the number of mentions among topics, you can use a [Tumbling window](#) to get the count of mentions by topic every five seconds.

1. Close the **Inputs** blade if you haven't already.
2. In the **Overview** blade, click **Edit Query** near the top right of the Query box. Azure lists the inputs and outputs that are configured for the job, and lets you create a query that lets you transform the input stream as it is sent to the output.
3. Make sure that the TwitterWpfClient application is running.
4. In the **Query** blade, click the dots next to the `TwitterStream` input and then select **Sample data from input**.

This opens a blade that lets you specify how much sample data to get, defined in terms of how long to read the input stream.

- Set **Minutes** to 3 and then click **OK**.

Azure samples 3 minutes' worth of data from the input stream and notifies you when the sample data is ready. (This takes a short while.)

The sample data is stored temporarily and is available while you have the query window open. If you close the query window, the sample data is discarded, and you have to create a new set of sample data.

- Change the query in the code editor to the following:

```
SELECT System.Timestamp as Time, Topic, COUNT(*)
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY TUMBLINGWINDOW(s, 5), Topic
```

If didn't use `TwitterStream` as the alias for the input, substitute your alias for `TwitterStream` in the query.

This query uses the **TIMESTAMP BY** keyword to specify a timestamp field in the payload to be used in the temporal computation. If this field isn't specified, the windowing operation is performed by using the time that each event arrived at the event hub. Learn more in the "Arrival Time vs Application Time" section of [Stream Analytics Query Reference](#).

This query also accesses a timestamp for the end of each window by using the **System.Timestamp** property.

- Click **Test**. The query runs against the data that you sampled.
- Click **Save**. This saves the query as part of the Streaming Analytics job. (It doesn't save the sample data.)

Experiment using different fields from the stream

The following table lists the fields that are part of the JSON streaming data. Feel free to experiment in the query editor.

JSON PROPERTY	DEFINITION
CreatedAt	The time that the tweet was created

JSON PROPERTY	DEFINITION
Topic	The topic that matches the specified keyword
SentimentScore	The sentiment score from Sentiment140
Author	The Twitter handle that sent the tweet
Text	The full body of the tweet

Create an output sink

You have now defined an event stream, an event hub input to ingest events, and a query to perform a transformation over the stream. The last step is to define an output sink for the job.

In this tutorial, you write the aggregated tweet events from the job query to Azure Blob storage. You can also push your results to Azure SQL Database, Azure Table storage, Event Hubs, or Power BI, depending on your application needs.

Specify the job output

1. In the **Job Topology** section, click the **Output** box.
2. In the **Outputs** blade, click **+ Add** and then fill out the blade with these values:
 - **Output alias:** Use the name `TwitterStream-Output`.
 - **Sink:** Select **Blob storage**.
 - **Import options:** Select **Use blob storage from current subscription**.
 - **Storage account.** Select **Create a new storage account**.
 - **Storage account** (second box). Enter `YOURNAMEsa`, where `YOURNAME` is your name or another unique string. The name can use only lowercase letters and numbers, and it must be unique across Azure.
 - **Container.** Enter `socialtwitter`. The storage account name and container name are used together to provide a URI for the blob storage, like this:

```
http://YOURNAMEsa.blob.core.windows.net/socialtwitter/...
```

New output

* Output alias
TwitterStream-Output

* Sink ⓘ
Blob storage

* Import option
Use blob storage from current subscription

Storage account
Create a new storage account

* Storage account ⓘ
twittersa

* Container
socialtwitter

Path pattern ⓘ
[empty]

Date format
YYYY/MM/DD

Time format
HH

* Event serialization format ⓘ
JSON

Encoding ⓘ
UTF-8

Format ⓘ
Line separated

3. Click **Create**.

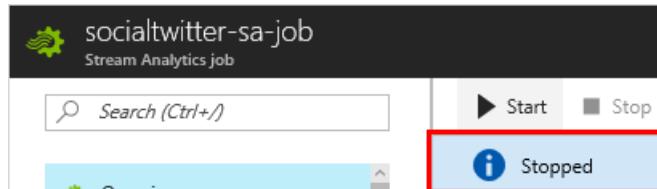
Azure creates the storage account and generates a key automatically.

4. Close the **Outputs** blade.

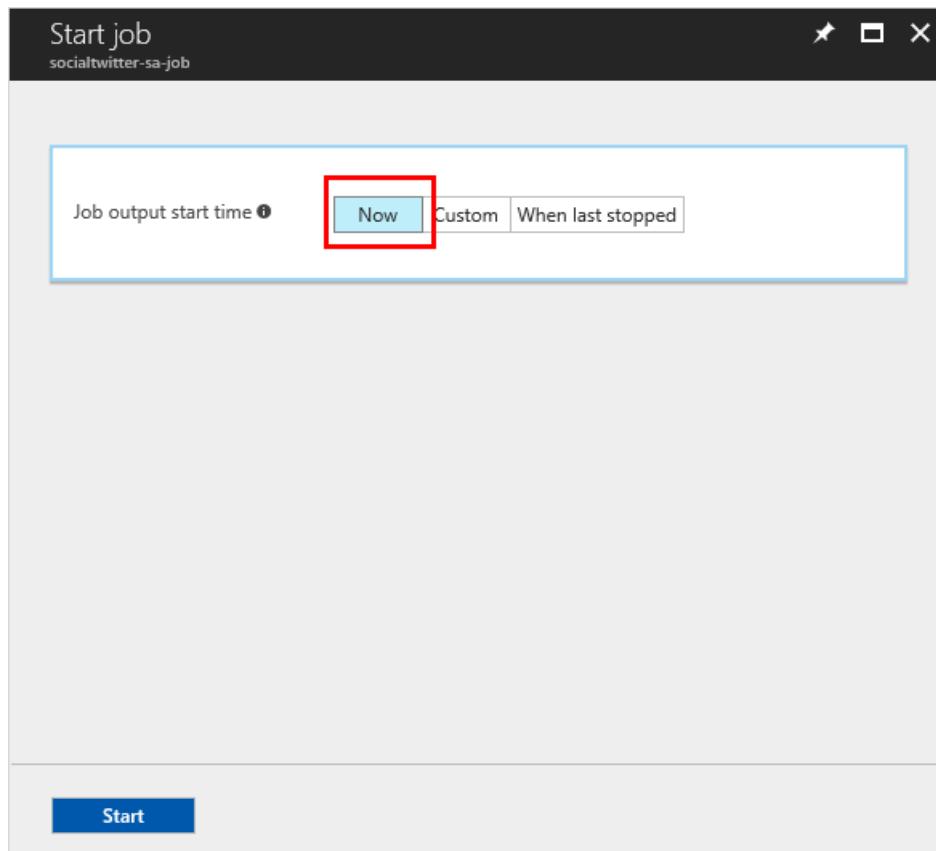
Start the job

A job input, query, and output are specified. You are ready to start the Stream Analytics job.

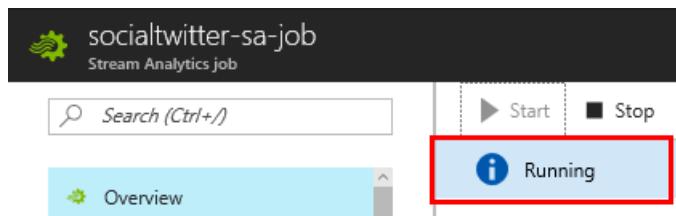
1. Make sure that the TwitterWpfClient application is running.
2. In the job blade, click **Start**.



3. In the **Start job** blade, for **Job output start time**, select **Now** and then click **Start**.



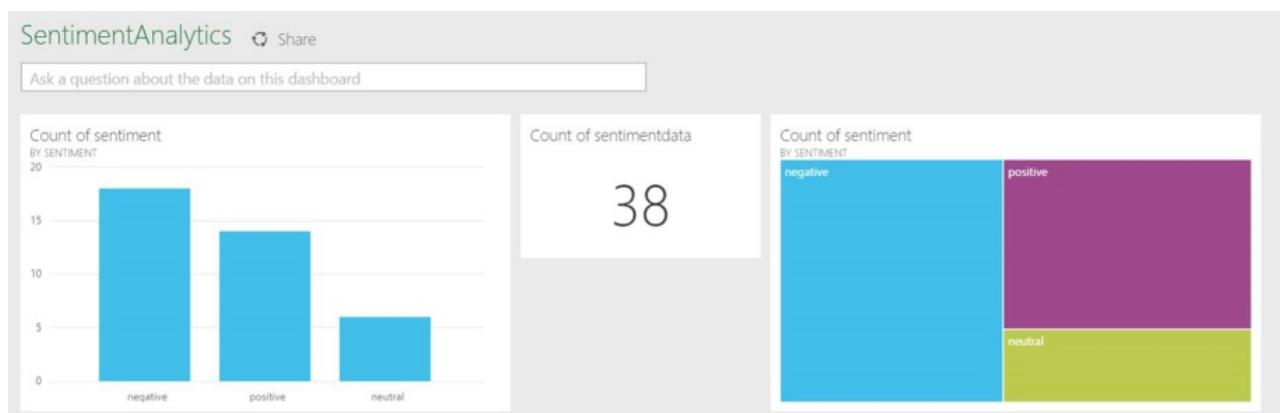
Azure notifies you when the job has started, and in the job blade, the status is displayed as **Running**.



View output for sentiment analysis

After your job has started running and is processing the real-time Twitter stream, you can view the output for sentiment analysis.

You can use a tool like [Azure Storage Explorer](#) or [Azure Explorer](#) to view your job output in real time. From here, you can use [Power BI](#) to extend your application to include a customized dashboard like the one shown in the following screenshot:



Create another query to identify trending topics

Another query you can use to understand Twitter sentiment is based on a [Sliding Window](#). To identify trending

topics, you look for topics that cross a threshold value for mentions in a specified amount of time.

For the purposes of this tutorial, you check for topics that are mentioned more than 20 times in the last 5 seconds.

1. In the job blade, click **Stop** to stop the job.
2. In the **Job Topology** section, click the **Query** box.
3. Change the query to the following:

```
SELECT System.Timestamp as Time, Topic, COUNT(*) as Mentions
FROM TwitterStream TIMESTAMP BY CreatedAt
GROUP BY SLIDINGWINDOW(s, 5), topic
HAVING COUNT(*) > 20
```

4. Click **Save**.
5. Make sure that the TwitterWpfClient application is running.
6. Click **Start** to restart the job using the new query.

Get support

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Get started using Azure Stream Analytics: Real-time fraud detection

6/13/2019 • 17 minutes to read • [Edit Online](#)

This tutorial provides an end-to-end illustration of how to use Azure Stream Analytics. You learn how to:

- Bring streaming events into an instance of Azure Event Hubs. In this tutorial, you'll use an app that simulates a stream of mobile-phone metadata records.
- Write SQL-like Stream Analytics queries to transform data, aggregating information or looking for patterns. You will see how to use a query to examine the incoming stream and look for calls that might be fraudulent.
- Send the results to an output sink (storage) that you can analyze for additional insights. In this case, you'll send the suspicious call data to Azure Blob storage.

This tutorial uses the example of real-time fraud detection based on phone-call data. The technique illustrated is also suited for other types of fraud detection, such as credit card fraud or identity theft.

Scenario: Telecommunications and SIM fraud detection in real time

A telecommunications company has a large volume of data for incoming calls. The company wants to detect fraudulent calls in real time so that they can notify customers or shut down service for a specific number. One type of SIM fraud involves multiple calls from the same identity around the same time but in geographically different locations. To detect this type of fraud, the company needs to examine incoming phone records and look for specific patterns—in this case, for calls made around the same time in different countries/regions. Any phone records that fall into this category are written to storage for subsequent analysis.

Prerequisites

In this tutorial, you'll simulate phone-call data by using a client app that generates sample phone call metadata. Some of the records that the app produces look like fraudulent calls.

Before you start, make sure you have the following:

- An Azure account.
- The call-event generator app, [TelcoGenerator.zip](#), which can be downloaded from the Microsoft Download Center. Unzip this package into a folder on your computer. If you want to see the source code and run the app in a debugger, you can get the app source code from [GitHub](#).

NOTE

Windows might block the downloaded .zip file. If you can't unzip it, right-click the file and select **Properties**. If you see the "This file came from another computer and might be blocked to help protect this computer" message, select the **Unblock** option and then click **Apply**.

If you want to examine the results of the Streaming Analytics job, you also need a tool for viewing the contents of an Azure Blob Storage container. If you use Visual Studio, you can use [Azure Tools for Visual Studio](#) or [Visual Studio Cloud Explorer](#). Alternatively, you can install standalone tools like [Azure Storage Explorer](#) or [Cerulean](#).

Create an Azure Event Hubs to ingest events

To analyze a data stream, you *ingest* it into Azure. A typical way to ingest data is to use [Azure Event Hubs](#), which lets you ingest millions of events per second and then process and store the event information. For this tutorial, you will create an event hub and then have the call-event generator app send call data to that event hub. For more about event hubs, see the [Azure Service Bus documentation](#).

NOTE

For a more detailed version of this procedure, see [Create an Event Hubs namespace and an event hub using the Azure portal](#).

Create a namespace and event hub

In this procedure, you first create an event hub namespace, and then you add an event hub to that namespace. Event hub namespaces are used to logically group related event bus instances.

1. Log in to the Azure portal and click **Create a resource** > **Internet of Things** > **Event Hub**.
2. In the **Create namespace** pane, enter a namespace name such as `<yourname>-eh-ns-demo`. You can use any name for the namespace, but the name must be valid for a URL and it must be unique across Azure.
3. Select a subscription and create or choose a resource group, then click **Create**.

Name
asa-eh-ns-demo

Pricing tier (View full pricing details)
Standard (20 Consumer groups, 1000 Bro...)

Enable Kafka

Make this namespace zone redundant

Subscription

Resource group
 Create new Use existing
asa-eh-ns-rg

Location
East US

Throughput Units
1

Enable Auto-Inflate

Create

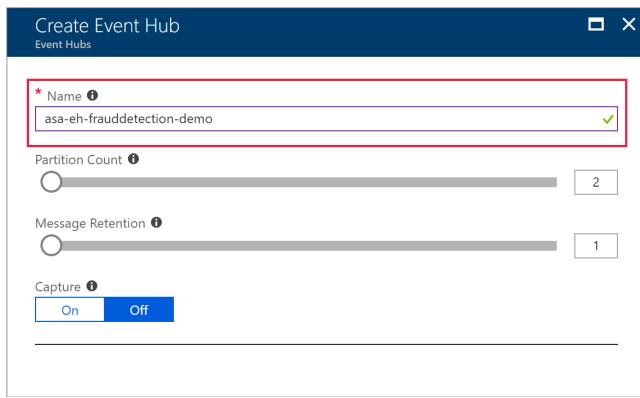
4. When the namespace has finished deploying, find the event hub namespace in your list of Azure resources.
5. Click the new namespace, and in the namespace pane, click **Event Hub**.

Resource group (change)	Status	Location	Subscription (change)
asa-eh-ns-rg	Active	East US	asa-eh-ns-demo
Subscription ID	Provisioning state	Created	Updated
	Succeeded	Tuesday, July 3, 2018, 15:14:50 EDT	Tuesday, July 3, 2018, 15:15:16 EDT
Connection Strings			

NAMESPACE CONTENTS: 0 EVENT HUBS | PRICING TIER: STANDARD | NAMESPACE STATUS: ACTIVE

Show metrics data for the last: 1 hour | 6 hours | 12 hours | 1 day | 7 days | 30 days

6. Name the new event hub **asa-eh-fraudetection-demo**. You can use a different name. If you do, make a note of it, because you need the name later. You don't need to set any other options for the event hub right now.



7. Click **Create**.

Grant access to the event hub and get a connection string

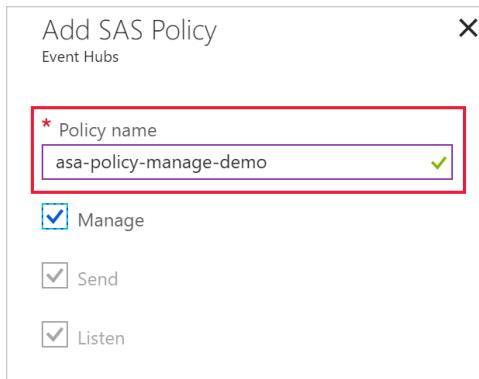
Before a process can send data to an event hub, the event hub must have a policy that allows appropriate access. The access policy produces a connection string that includes authorization information.

1. In the event namespace pane, click **Event Hubs** and then click the name of your new event hub.
2. In the event hub pane, click **Shared access policies** and then click **+ Add**.

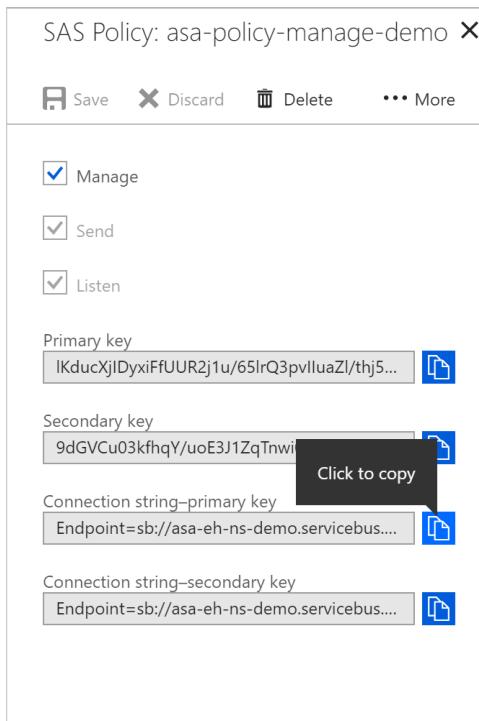
NOTE

Make sure you're working with the event hub, not the event hub namespace.

3. Add a policy named `sa-policy-manage-demo` and for **Claim**, select **Manage**.



4. Click **Create**.
5. After the policy has been deployed, click it in the list of shared access policies.
6. Find the box labeled **CONNECTION STRING-PRIMARY KEY** and click the copy button next to the connection string.



- Paste the connection string into a text editor. You need this connection string for the next section, after you make some small edits to it.

The connection string looks like this:

```
Endpoint=sb://YOURNAME-eh-ns-demo.servicebus.windows.net/;SharedAccessKeyName=asa-policy-manage-demo;SharedAccessKey=Gw2NFZwU1Di+rxA2T+6hJYAtFExKRXaC2oSQa0ZsPkJ=;EntityPath=asa-eh-frauddetection-demo
```

Notice that the connection string contains multiple key-value pairs, separated with semicolons:

`Endpoint`, `SharedAccessKeyName`, `SharedAccessKey`, and `EntityPath`.

Configure and start the event generator application

Before you start the TelcoGenerator app, you must configure it so that it will send call records to the event hub you created.

Configure the TelcoGenerator app

- In the editor where you copied the connection string, make a note of the `EntityPath` value, and then remove the `EntityPath` pair (don't forget to remove the semicolon that precedes it).
- In the folder where you unzipped the TelcoGenerator.zip file, open the `telcodatagen.exe.config` file in an editor. (There is more than one .config file, so be sure that you open the right one.)
- In the `<appSettings>` element:

- Set the value of the `EventHubName` key to the event hub name (that is, to the value of the entity path).
- Set the value of the `Microsoft.ServiceBus.ConnectionString` key to the connection string.

The `<appSettings>` section will look like the following example. (For clarity, the lines are wrapped and some characters have been removed from the authorization token.)

```

<appSettings>
    <!-- Service Bus specific app settings for messaging connections -->
    <add key="EventHubName"
        value="sa-eh-fraudetection-demo"/>
    <add key="Microsoft.ServiceBus.ConnectionString"
        value="Endpoint=sb://sa-eh-ns-demo.servicebus.windows.net/;SharedAccessKeyName=sa-policy-manage-demo;SharedAccessKey=oPMh1MFF1...Dtxn7o="/>
</appSettings>

```

4. Save the file.

Start the app

1. Open a command window and change to the folder where the TelcoGenerator app is unzipped.

2. Enter the following command:

```
telcodatagen.exe 1000 0.2 2
```

The parameters are:

- * Number of CDRs per hour.
- * SIM Card Fraud Probability: How often, as a percentage of all calls, that the app should simulate a fraudulent call. The value 0.2 means that about 20% of the call records will look fraudulent.
- * Duration in hours. The number of hours that the app should run. You can also stop the app any time by pressing Ctrl+C at the command line.

After a few seconds, the app starts displaying phone call records on the screen as it sends them to the event hub.

Some of the key fields that you will be using in this real-time fraud detection application are the following:

RECORD	DEFINITION
CallrecTime	The timestamp for the call start time.
SwitchNum	The telephone switch used to connect the call. For this example, the switches are strings that represent the country/region of origin (US, China, UK, Germany, or Australia).
CallingNum	The phone number of the caller.
CallingIMSI	The International Mobile Subscriber Identity (IMSI). This is the unique identifier of the caller.
CalledNum	The phone number of the call recipient.
CalledIMSI	International Mobile Subscriber Identity (IMSI). This is the unique identifier of the call recipient.

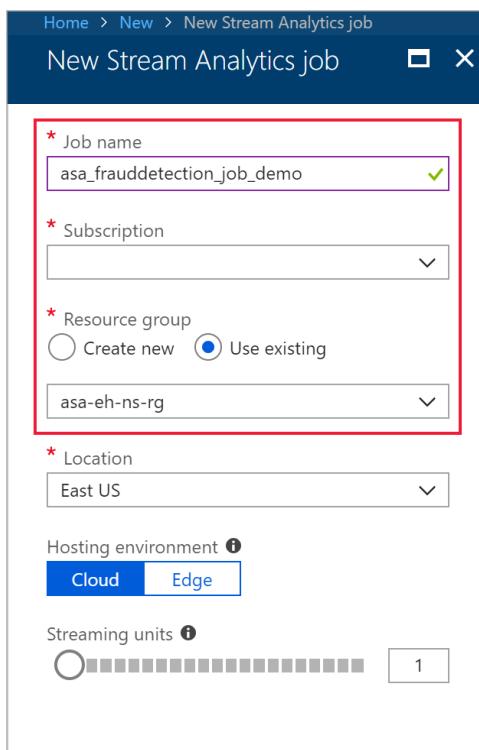
Create a Stream Analytics job to manage streaming data

Now that you have a stream of call events, you can set up a Stream Analytics job. The job will read data from the event hub that you set up.

Create the job

1. In the Azure portal, click **Create a resource** > **Internet of Things** > **Stream Analytics job**.
2. Name the job `asa_fraudetection_job_demo`, specify a subscription, resource group, and location.

It's a good idea to place the job and the event hub in the same region for best performance and so that you don't pay to transfer data between regions.



3. Click **Create**.

The job is created and the portal displays job details. Nothing is running yet, though—you have to configure the job before it can be started.

Configure job input

1. In the dashboard or the **All resources** pane, find and select the `asa_frauddetection_job_demo` Stream Analytics job.
2. In the **Overview** section of the Stream Analytics job pane, click the **Input** box.

```
1 SELECT
2 *
3 INTO
4   [YourOutputAlias]
5 FROM
6   [YourInputAlias]
```

3. Click **Add stream input** and select **Event Hub**. Then fill the New input page with the following information:

Setting	Suggested Value	Description
Input alias	CallStream	Enter a name to identify the job's input.
Subscription	<Your subscription>	Select the Azure subscription that has the Event Hub you created.
Event Hub namespace	asa-eh-ns-demo	Enter the name of the Event Hub namespace.
Event Hub name	asa-eh-fraudetection-demo	Select the name of your Event Hub.
Event Hub policy name	asa-policy-manage-demo	Select the access policy that you created earlier.

Event Hub
New input

* Input alias
CallStream

Provide Event Hub settings manually
 Select Event Hub from your subscriptions

Subscription

* Event Hub namespace ⓘ
asa-eh-ns-demo

* Event Hub name ⓘ
 Create new Use existing
asa-eh-fraudetection-demo

* Event Hub policy name ⓘ
asa-policy-manage-demo

Event Hub policy key

Event Hub consumer group ⓘ

* Event serialization format ⓘ
JSON

Encoding ⓘ
UTF-8

Event compression type ⓘ
None

Save

4. Click **Create**.

Create queries to transform real-time data

At this point, you have a Stream Analytics job set up to read an incoming data stream. The next step is to create a query that analyzes the data in real time. Stream Analytics supports a simple, declarative query model that describes transformations for real-time processing. The queries use a SQL-like language that has some extensions specific to Stream Analytics.

A simple query might just read all the incoming data. However, you often create queries that look for specific data or for relationships in the data. In this section of the tutorial, you create and test several queries to learn a few ways in which you can transform an input stream for analysis.

The queries you create here will just display the transformed data to the screen. In a later section, you'll configure an output sink and a query that writes the transformed data to that sink.

To learn more about the language, see the [Azure Stream Analytics Query Language Reference](#).

Get sample data for testing queries

The TelcoGenerator app is sending call records to the event hub, and your Stream Analytics job is configured to read from the event hub. You can use a query to test the job to make sure that it's reading correctly. To test a query in the Azure console, you need sample data. For this walkthrough, you'll extract sample data from the stream that's coming into the event hub.

1. Make sure that the TelcoGenerator app is running and producing call records.
2. In the portal, return to the Streaming Analytics job pane. (If you closed the pane, search for `asa_fraudetection_job_demo` in the **All resources** pane.)
3. Click the **Query** box. Azure lists the inputs and outputs that are configured for the job, and lets you create a query that lets you transform the input stream as it is sent to the output.
4. In the **Query** pane, click the dots next to the `CallStream` input and then select **Sample data from input**.

The screenshot shows the Azure Stream Analytics job pane for 'asa_fraudetection_job_demo'. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, SETTINGS (Locks), JOB TOPOLOGY (Inputs, Functions), and a bottom section for Local Time (UTC-07:00). The main area has tabs for Save, Discard, and Test. Under 'Inputs (1)', 'CallStream' is selected. A context menu is open over 'CallStream' with options: 'Upload sample data from file' and 'Sample data from input' (which is highlighted with a red box). The status bar at the bottom says '1 SELECT'.

5. Set **Minutes** to 3 and then click **OK**.

The screenshot shows the 'Sample data from input' configuration window. It has fields for 'Start time' (set to 2017-06-13 at 2:15:27 PM) and 'Duration' (set to 3 minutes). Below the duration, there are dropdowns for Days (0), Hours (0), Minutes (3), and Seconds (0).

Azure samples 3 minutes' worth of data from the input stream and notifies you when the sample data is ready. (This takes a short while.)

The sample data is stored temporarily and is available while you have the query window open. If you close the query window, the sample data is discarded, and you'll have to create a new set of sample data.

As an alternative, you can get a json file that has sample data in it [from GitHub](#), and then upload that json file

to use as sample data for the `CallStream` input.

Test using a pass-through query

If you want to archive every event, you can use a pass-through query to read all the fields in the payload of the event.

1. In the query window, enter this query:

```
SELECT  
    *  
FROM  
    CallStream
```

NOTE

As with SQL, keywords are not case-sensitive, and whitespace is not significant.

In this query, `CallStream` is the alias that you specified when you created the input. If you used a different alias, use that name instead.

2. Click **Test**.

The Stream Analytics job runs the query against the sample data and displays the output at the bottom of the window. The results indicate that the Event Hub and the Streaming Analytics job are configured correctly. (As noted, later you'll create an output sink that the query can write data to.)

Results														
output														
Generated the Following:														
• output with 73 rows.														
RECOR...	SYSTEM...	FILENUM	SWITCH...	CALLIN...	CALLIN...	CALLED...	CALLEDI...	DATES	TIMES	TIMETY...	CALLPE...	CALLIN...	CALLED...	
"MO"	"d0"	"862"	"Austra..."	"23453..."	"46692..."	"01234..."	"46692..."	"20170..."	null	1	528	null	null	
"MO"	"d0"	"831"	"China"	"23452..."	"46692..."	"78904..."	"46692..."	"20170..."	null	2	38	null	null	
"MO"	"d0"	"875"	"China"	"34562..."	"46692..."	"23456..."	"46692..."	"20170..."	null	2	310	null	null	
"MO"	"d0"	"877"	"US"	"78906..."	"46692..."	"78900..."	"46692..."	"20170..."	null	1	38	null	null	
"MO"	"d0"	"836"	"Austra..."	"67894..."	"46692..."	"01230..."	"46692..."	"20170..."	null	1	336	null	null	

The exact number of records you see will depend on how many records were captured in your 3-minute sample.

Reduce the number of fields using a column projection

In many cases, your analysis doesn't need all the columns from the input stream. You can use a query to project a smaller set of returned fields than in the pass-through query.

1. Change the query in the code editor to the following:

```
SELECT CallRecTime, SwitchNum, CallingIMSI, CallingNum, CalledNum  
FROM  
    CallStream
```

2. Click **Test** again.

Results

output

Generated the Following:

- output with 25 rows.

[Download results](#)

CALLRECTIME	SWITCHNUM	CALLINGIMSI
"2017-06-15T00:00:15.0000000Z"	"China"	"466922000696024"
"2017-06-14T23:52:52.0000000Z"	"Germany"	"466923200779222"
"2017-06-14T23:52:53.0000000Z"	"US"	"466922702341485"
"0001-01-01T00:00:00.0000000"	"Australia"	"466922200432822"
"2017-06-14T23:52:54.0000000Z"	"US"	"466921602131264"
"2017-06-14T23:52:54.0000000Z"	"US"	"466921602131264"

Count incoming calls by region: Tumbling window with aggregation

Suppose you want to count the number of incoming calls per region. In streaming data, when you want to perform aggregate functions like counting, you need to segment the stream into temporal units (since the data stream itself is effectively endless). You do this using a Streaming Analytics [window function](#). You can then work with the data inside that window as a unit.

For this transformation, you want a sequence of temporal windows that don't overlap—each window will have a discrete set of data that you can group and aggregate. This type of window is referred to as a *Tumbling window*. Within the Tumbling window, you can get a count of the incoming calls grouped by `SwitchNum`, which represents the country/region where the call originated.

1. Change the query in the code editor to the following:

```
```SQL
SELECT
 System.Timestamp as WindowEnd, SwitchNum, COUNT(*) as CallCount
FROM
 CallStream TIMESTAMP BY CallRecTime
GROUP BY TUMBLINGWINDOW(s, 5), SwitchNum
```
```

This query uses the `Timestamp By` keyword in the `FROM` clause to specify which timestamp field in the input stream to use to define the Tumbling window. In this case, the window divides the data into segments by the `callRecTime` field in each record. (If no field is specified, the windowing operation uses the time that each event arrives at the event hub. See "Arrival Time Vs Application Time" in [Stream Analytics Query Language Reference](#).

The projection includes `System.Timestamp`, which returns a timestamp for the end of each window.

To specify that you want to use a Tumbling window, you use the [TUMBLINGWINDOW](#) function in the `GROUP BY` clause. In the function, you specify a time unit (anywhere from a microsecond to a day) and a window size (how many units). In this example, the Tumbling window consists of 5-second intervals, so you will get a count by country/region for every 5 seconds' worth of calls.

2. Click **Test** again. In the results, notice that the timestamps under **WindowEnd** are in 5-second increments.

Results

output

Generated the Following:

- output with 13 rows.

[Download results](#)

| WINDOWEND | SWITCHNUM | CALLCOUNT |
|--------------------------------|-------------|-----------|
| "0001-01-01T00:00:00.0000000Z" | "UK" | 1 |
| "0001-01-01T00:00:00.0000000Z" | "Australia" | 1 |
| "2017-06-14T23:52:55.0000000Z" | "US" | 3 |
| "2017-06-14T23:52:55.0000000Z" | "Germany" | 2 |
| "2017-06-14T23:53:00.0000000Z" | "China" | 3 |
| "2017-06-14T23:53:00.0000000Z" | "UK" | 3 |

Detect SIM fraud using a self-join

For this example, consider fraudulent usage to be calls that originate from the same user but in different locations within 5 seconds of one another. For example, the same user can't legitimately make a call from the US and Australia at the same time.

To check for these cases, you can use a self-join of the streaming data to join the stream to itself based on the `callRecTime` value. You can then look for call records where the `CallingIMSI` value (the originating number) is the same, but the `SwitchNum` value (country/region of origin) is not the same.

When you use a join with streaming data, the join must provide some limits on how far the matching rows can be separated in time. (As noted earlier, the streaming data is effectively endless.) The time bounds for the relationship are specified inside the `ON` clause of the join, using the `DATEDIFF` function. In this case, the join is based on a 5-second interval of call data.

1. Change the query in the code editor to the following:

```
```SQL
SELECT System.Timestamp as Time,
 CS1.CallingIMSI,
 CS1.CallingNum as CallingNum1,
 CS2.CallingNum as CallingNum2,
 CS1.SwitchNum as Switch1,
 CS2.SwitchNum as Switch2
 FROM CallStream CS1 TIMESTAMP BY CallRecTime
 JOIN CallStream CS2 TIMESTAMP BY CallRecTime
 ON CS1.CallingIMSI = CS2.CallingIMSI
 AND DATEDIFF(ss, CS1, CS2) BETWEEN 1 AND 5
 WHERE CS1.SwitchNum != CS2.SwitchNum
```

```

This query is like any SQL join except for the `DATEDIFF` function in the join. This version of `DATEDIFF` is specific to Streaming Analytics, and it must appear in the `ON...BETWEEN` clause. The parameters are a time unit (seconds in this example) and the aliases of the two sources for the join. This is different from the standard SQL `DATEDIFF` function.

The `WHERE` clause includes the condition that flags the fraudulent call: the originating switches are not the same.

2. Click **Test** again.

Results

output

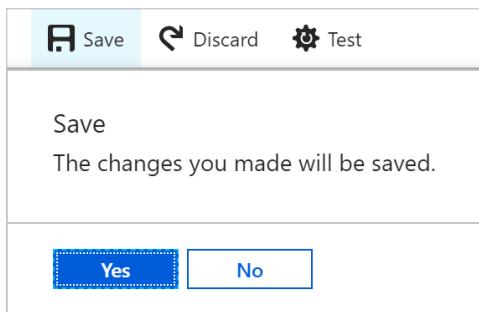
Generated the Following:

- output with 6 rows.

Download results

| TIME | CALLINGIMSI | CALLINGNUM1 | CALLINGNUM2 | SWITCH1 | SWITCH2 |
|--------------------------------|-------------------|-------------|-------------|-------------|-------------|
| "2017-06-15T00:00:29.0000000Z" | "466921402237651" | "345663427" | "012349301" | "Germany" | "Australia" |
| "2017-06-15T00:00:31.0000000Z" | "466921402237651" | "012349301" | "567815697" | "Australia" | "China" |
| "2017-06-15T00:00:31.0000000Z" | "466921402237651" | "345663427" | "567815697" | "Germany" | "China" |
| "2017-06-15T00:00:33.0000000Z" | "466923100098619" | "012315165" | "345607796" | "Australia" | "UK" |
| "2017-06-15T00:00:33.0000000Z" | "466922200432822" | "012304256" | "789065257" | "China" | "Australia" |
| "2017-06-15T00:00:34.0000000Z" | "466921602343040" | "789058195" | "789059831" | "Australia" | "US" |

3. Click **Save** to save the self-join query as part of the Streaming Analytics job. (It doesn't save the sample data.)



Create an output sink to store transformed data

You've defined an event stream, an event hub input to ingest events, and a query to perform a transformation over the stream. The last step is to define an output sink for the job—that is, a place to write the transformed stream to.

You can use many resources as output sinks—a SQL Server database, table storage, Data Lake storage, Power BI, and even another event hub. For this tutorial, you'll write the stream to Azure Blob Storage, which is a typical choice for collecting event information for later analysis, since it accommodates unstructured data.

If you have an existing blob storage account, you can use that. For this tutorial, you will learn how to create a new storage account.

Create an Azure Blob Storage account

1. From the upper left-hand corner of the Azure portal, select **Create a resource > Storage > Storage account**. Fill out the Storage account job page with **Name** set to "asaehstorage", **Location** set to "East US", **Resource group** set to "asa-eh-ns-rg" (host the storage account in the same resource group as the Streaming job for increased performance). The remaining settings can be left to their default values.

2. In the Azure portal, return to the Streaming Analytics job pane. (If you closed the pane, search for `asa_fraudetection_job_demo` in the **All resources** pane.)
3. In the **Job Topology** section, click the **Output** box.
4. In the **Outputs** pane, click **Add** and select **Blob storage**. Then fill out the New output page with the following information:

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|-----------------|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Output alias | CallStream-FraudulentCalls | Enter a name to identify the job's output. |
| Subscription | <Your subscription> | Select the Azure subscription that has the storage account you created. The storage account can be in the same or in a different subscription. This example assumes that you have created storage account in the same subscription. |
| Storage account | asaehstorage | Enter the name of the storage account you created. |

| SETTING | SUGGESTED VALUE | DESCRIPTION |
|-----------|--------------------------|-----------------------------------------------|
| Container | asa-fraudulentcalls-demo | Choose Create new and enter a container name. |

Blob storage

New output

* Output alias
CallStream-FraudulentCalls

Provide Blob storage settings manually
 Select Blob storage from your subscriptions

Subscription

* Storage account i
asaehstorage

* Storage account key

* Container
 Create new
 Use existing
asa-fraudulentcalls-demo

Path pattern i

Date format
YYYY/MM/DD

Time format
HH

* Event serialization format i
JSON

Encoding i
UTF-8

Format i
Line separated

Save

5. Click **Save**.

Start the Streaming Analytics job

The job is now configured. You've specified an input (the event hub), a transformation (the query to look for fraudulent calls), and an output (blob storage). You can now start the job.

1. Make sure the TelcoGenerator app is running.
2. In the job pane, click **Start**. In the **Start job** pane, for Job output start time, select **Now**.

```

1 SELECT System.Timestamp as Time,
2     CS1.CallingIMSI,
3     CS1.CallingNum as CallingNum1,
4     CS2.CallingNum as CallingNum2,
5     CS1.SwitchNum as Switch1,
6     CS2.SwitchNum as Switch2
7 FROM CallStream CS1 TIMESTAMP BY CallTime
8 JOIN CallStream CS2 TIMESTAMP BY CallTime
9 ON CS1.CallingIMSI = CS2.CallingIMSI
10 AND DATEDIFF(ss, CS1, CS2) BETWEEN
11 WHERE CS1.SwitchNum != CS2.SwitchNum

```

Examine the transformed data

You now have a complete Streaming Analytics job. The job is examining a stream of phone call metadata, looking for fraudulent phone calls in real time, and writing information about those fraudulent calls to storage.

To complete this tutorial, you might want to look at the data being captured by the Streaming Analytics job. The data is being written to Azure Blob Storage in chunks (files). You can use any tool that reads Azure Blob Storage. As noted in the Prerequisites section, you can use Azure extensions in Visual Studio, or you can use a tool like [Azure Storage Explorer](#) or [Cerulean](#).

When you examine the contents of a file in blob storage, you see something like the following:

```

1 {"time": "2017-06-15T00:26:23.187000Z", "callingimsi": "466922702341485", "callingnum1": "123423210", "callingnum2": "789002532", "switch1": "UK", "switch2": "Australia"}
2 {"time": "2017-06-15T00:26:56.891000Z", "callingimsi": "466923101048691", "callingnum1": "789056161", "callingnum2": "345636784", "switch1": "US", "switch2": "UK"}
3 {"time": "2017-06-15T00:27:01.184000Z", "callingimsi": "466921602343040", "callingnum1": "012309823", "callingnum2": "345686532", "switch1": "China", "switch2": "UK"}
4 {"time": "2017-06-15T00:27:01.521000Z", "callingimsi": "466923300507919", "callingnum1": "678930584", "callingnum2": "234540532", "switch1": "Germany", "switch2": "US"}
5 {"time": "2017-06-15T00:27:02.215000Z", "callingimsi": "466923306236137", "callingnum1": "678923578", "callingnum2": "678998343", "switch1": "Australia", "switch2": "China"}
6 {"time": "2017-06-15T00:27:02.657000Z", "callingimsi": "466922202679249", "callingnum1": "345604483", "callingnum2": "678992838", "switch1": "Germany", "switch2": "UK"}
7 {"time": "2017-06-15T00:27:02.886000Z", "callingimsi": "466922202546859", "callingnum1": "123400455", "callingnum2": "234526505", "switch1": "UK", "switch2": "US"}
8 {"time": "2017-06-15T00:27:03.081000Z", "callingimsi": "466922000696024", "callingnum1": "678988672", "callingnum2": "234595583", "switch1": "UK", "switch2": "China"}
9 {"time": "2017-06-15T00:27:03.315000Z", "callingimsi": "466923300507919", "callingnum1": "234540532", "callingnum2": "345624591", "switch1": "US", "switch2": "UK"}
10 {"time": "2017-06-15T00:27:03.316000Z", "callingimsi": "466923300507919", "callingnum1": "678930584", "callingnum2": "345624591", "switch1": "Germany", "switch2": "UK"}
11 {"time": "2017-06-15T00:27:03.449000Z", "callingimsi": "466920400352400", "callingnum1": "678906229", "callingnum2": "67898865", "switch1": "Australia", "switch2": "Germany"}
12 {"time": "2017-06-15T00:27:04.012000Z", "callingimsi": "466920400352400", "callingnum1": "678988964", "callingnum2": "678906229", "switch1": "Germany", "switch2": "China"}
13 {"time": "2017-06-15T00:27:04.012000Z", "callingimsi": "466920400352400", "callingnum1": "678906229", "callingnum2": "67898865", "switch1": "Australia", "switch2": "China"}
14 {"time": "2017-06-15T00:27:04.333000Z", "callingimsi": "466920400352400", "callingnum1": "678983965", "callingnum2": "234578014", "switch1": "Australia", "switch2": "China"}
15 {"time": "2017-06-15T00:27:04.458000Z", "callingimsi": "466923200408045", "callingnum1": "567820504", "callingnum2": "456775135", "switch1": "US", "switch2": "Germany"}

```

Clean up resources

There are additional articles that continue with the fraud-detection scenario and use the resources you've created in this tutorial. If you want to continue, see the suggestions under **Next steps**.

However, if you're done and you don't need the resources you've created, you can delete them so that you don't incur unnecessary Azure charges. In that case, we suggest that you do the following:

1. Stop the Streaming Analytics job. In the **Jobs** pane, click **Stop** at the top.
2. Stop the Telco Generator app. In the command window where you started the app, press **Ctrl+C**.
3. If you created a new blob storage account just for this tutorial, delete it.
4. Delete the Streaming Analytics job.
5. Delete the event hub.
6. Delete the event hub namespace.

Get support

For further assistance, try the [Azure Stream Analytics forum](#).

Next steps

You can continue this tutorial with the following article:

- [Stream Analytics and Power BI: A real-time analytics dashboard for streaming data](#). This article shows you how to send the TelCo output of the Stream Analytics job to Power BI for real-time visualization and analysis.

For more information about Stream Analytics in general, see these articles:

- [Introduction to Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

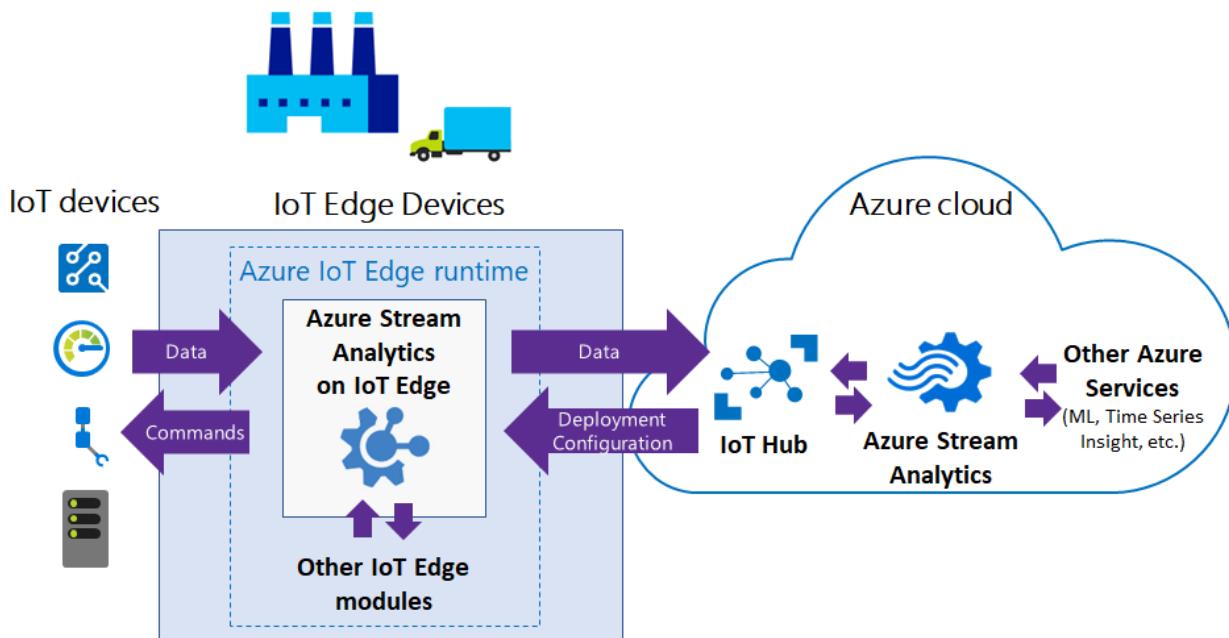
Azure Stream Analytics on IoT Edge

4/3/2019 • 9 minutes to read • [Edit Online](#)

Azure Stream Analytics (ASA) on IoT Edge empowers developers to deploy near-real-time analytical intelligence closer to IoT devices so that they can unlock the full value of device-generated data. Azure Stream Analytics is designed for low latency, resiliency, efficient use of bandwidth, and compliance. Enterprises can now deploy control logic close to the industrial operations and complement Big Data analytics done in the cloud.

Azure Stream Analytics on IoT Edge runs within the [Azure IoT Edge](#) framework. Once the job is created in ASA, you can deploy and manage it using IoT Hub.

Scenarios



- **Low-latency command and control:** For example, manufacturing safety systems must respond to operational data with ultra-low latency. With ASA on IoT Edge, you can analyze sensor data in near real-time, and issue commands when you detect anomalies to stop a machine or trigger alerts.
- **Limited connectivity to the cloud:** Mission critical systems, such as remote mining equipment, connected vessels, or offshore drilling, need to analyze and react to data even when cloud connectivity is intermittent. With ASA, your streaming logic runs independently of the network connectivity and you can choose what you send to the cloud for further processing or storage.
- **Limited bandwidth:** The volume of data produced by jet engines or connected cars can be so large that data must be filtered or pre-processed before sending it to the cloud. Using ASA, you can filter or aggregate the data that needs to be sent to the cloud.
- **Compliance:** Regulatory compliance may require some data to be locally anonymized or aggregated before being sent to the cloud.

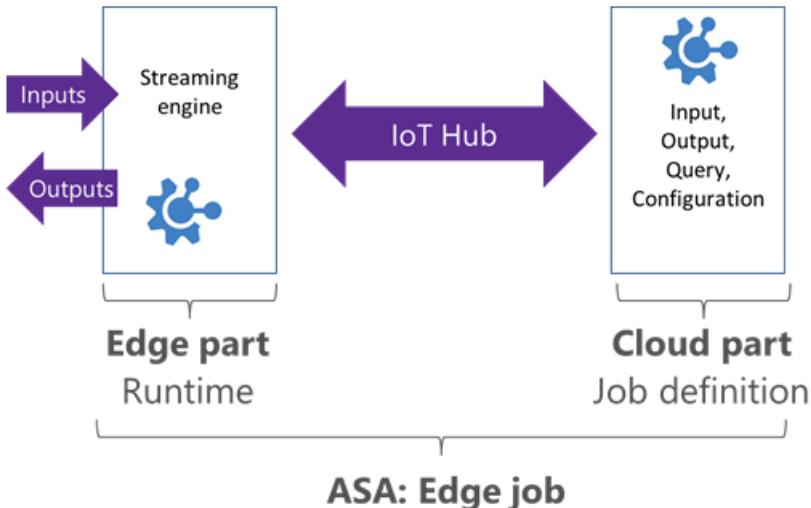
Edge jobs in Azure Stream Analytics

What is an "edge" job?

ASA Edge jobs run in containers deployed to [Azure IoT Edge devices](#). They are composed of two parts:

1. A cloud part that is responsible for job definition: users define inputs, output, query, and other settings (out of order events, etc.) in the cloud.
2. A module running on your IoT devices. It contains the ASA engine and receives the job definition from the cloud.

ASA uses IoT Hub to deploy edge jobs to device(s). More information about [IoT Edge deployment can be seen here](#).



Installation instructions

The high-level steps are described in the following table. More details are given in the following sections.

| | STEP | NOTES |
|---|----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | Create a storage container | Storage containers are used to save your job definition where they can be accessed by your IoT devices.
You can reuse any existing storage container. |
| 2 | Create an ASA edge job | Create a new job, select Edge as hosting environment .
These jobs are created/managed from the cloud, and run on your own IoT Edge devices. |
| 3 | Setup your IoT Edge environment on your device(s) | Instructions for Windows or Linux . |
| 4 | Deploy ASA on your IoT Edge device(s) | ASA job definition is exported to the storage container created earlier. |

You can follow [this step-by-step tutorial](#) to deploy your first ASA job on IoT Edge. The following video should help you understand the process to run a Stream Analytics job on an IoT edge device:

Create a storage container

A storage container is required in order to export the ASA compiled query and the job configuration. It is used to configure the ASA Docker image with your specific query.

1. Follow [these instructions](#) to create a storage account from the Azure portal. You can keep all default options to use this account with ASA.

2. In the newly created storage account, create a blob storage container:

- Click on **Blobs**, then **+ Container**.
- Enter a name and keep the container as **Private**.

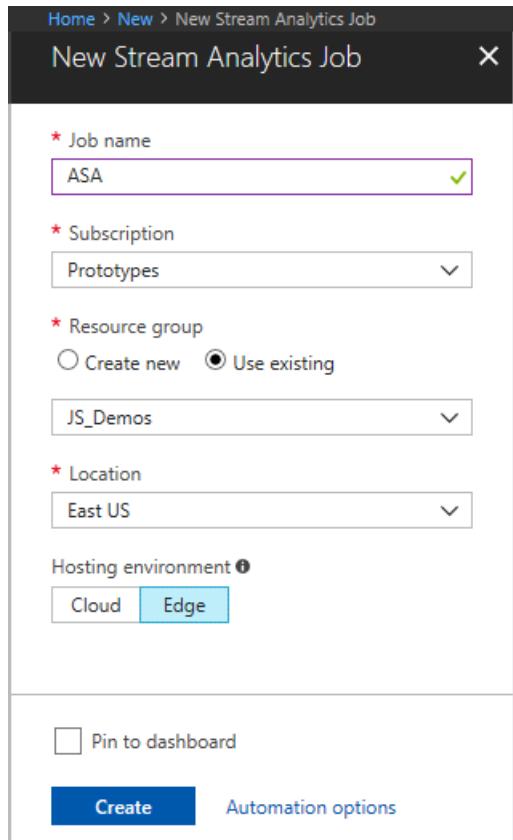
Create an ASA Edge job

NOTE

This tutorial focuses on ASA job creation using Azure portal. You can also use Visual Studio plugin to create an ASA Edge job

1. From the Azure portal, create a new "Stream Analytics job". [Direct link to create a new ASA job here.](#)

2. In the creation screen, select **Edge** as **hosting environment** (see the following picture)



3. Job Definition

- Define Input Stream(s).** Define one or several input streams for your job.
- Define Reference data (optional).
- Define Output Stream(s).** Define one or several outputs streams for your job.
- Define query.** Define the ASA query in the cloud using the inline editor. The compiler automatically checks the syntax enabled for ASA edge. You can also test your query by uploading sample data.

4. Set the storage container information in the **IoT Edge settings** menu.

5. Set optional settings

- Event ordering.** You can configure out-of-order policy in the portal. Documentation is available [here](#).
- Locale.** Set the internalization format.

NOTE

When a deployment is created, ASA exports the job definition to a storage container. This job definition remain the same during the duration of a deployment. As a consequence, if you want to update a job running on the edge, you need to edit the job in ASA, and then create a new deployment in IoT Hub.

Set up your IoT Edge environment on your device(s)

Edge jobs can be deployed on devices running Azure IoT Edge. For this, you need to follow these steps:

- Create an IoT Hub.
- Install Docker and IoT Edge runtime on your edge devices.
- Set your devices as **IoT Edge devices** in IoT Hub.

These steps are described in the IoT Edge documentation for [Windows](#) or [Linux](#).

Deployment ASA on your IoT Edge device(s)

[Add ASA to your deployment](#)

- In the Azure portal, open IoT Hub, navigate to **IoT Edge** and click on the device you want to target for this deployment.
- Select **Set modules**, then select **+ Add** and choose **Azure Stream Analytics Module**.
- Select the subscription and the ASA Edge job that you created. Click **Save**.

The screenshot shows the 'Edge Deployment' interface in the Azure portal. It's a three-step wizard:

- Step 1: Set Modules**: Shows a list of 'Azure Service IoT Edge Modules'. A button 'Import Azure Service IoT Edge Module' is visible.
- Step 2: Specify Routes (optional)**: Shows a table for IoT Edge Modules with columns NAME, VERSION, and STATUS. A button 'Add IoT Edge Module' is visible.
- Step 3: Review Template (optional)**: Shows configuration for 'Stream Analytics - Edge Job'. It includes dropdowns for 'Subscription' (set to 'Azure IoT client team subscription') and 'Edge job' (set to 'JSASAEEdge').

At the bottom, there are 'Previous' and 'Next' buttons, and a large blue 'Save' button.

NOTE

During this step, ASA creates a folder named "EdgeJobs" in the storage container (if it does not exist already). For each deployment, a new subfolder is created in the "EdgeJobs" folder. In order to deploy your job to edge devices, ASA creates a shared access signature (SAS) for the job definition file. The SAS key is securely transmitted to the IoT Edge devices using device twin. The expiration of this key is three years from the day of its creation.

For more information about IoT Edge deployments, see to [this page](#).

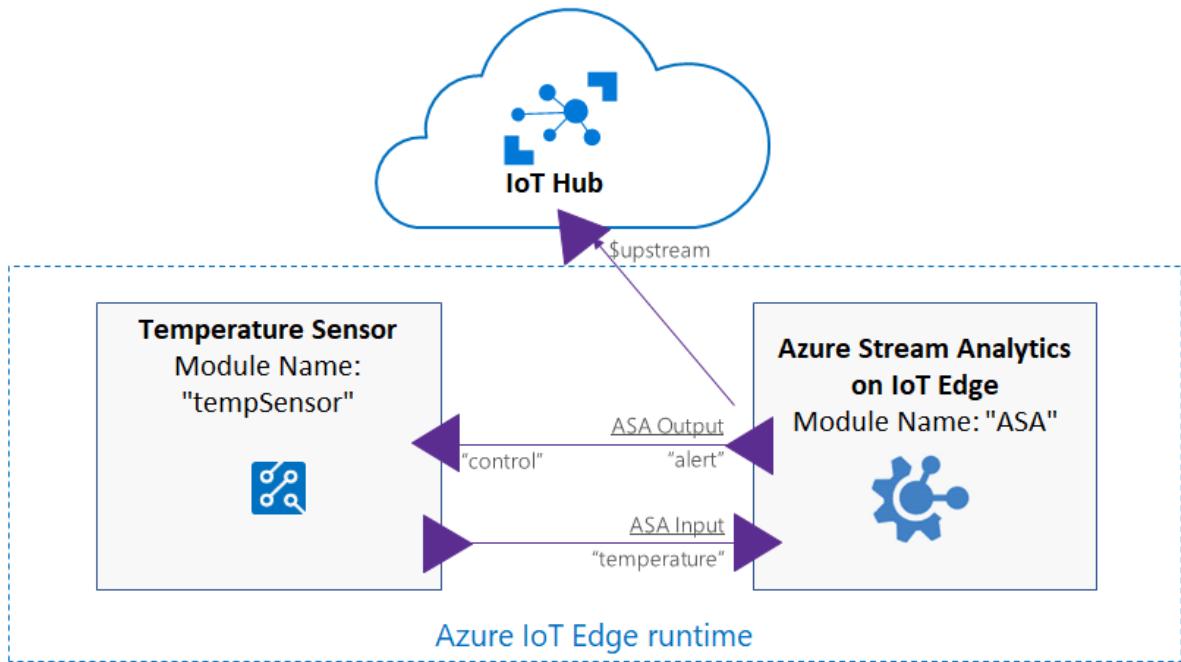
Configure routes

IoT Edge provides a way to declaratively route messages between modules, and between modules and IoT Hub. The full syntax is described [here](#). Names of the inputs and outputs created in the ASA job can be used as endpoints for routing.

Example

```
{  
    "routes": {  
        "sensorToAsa": "FROM /messages/modules/tempSensor/* INTO  
BrokeredEndpoint(\"/modules/ASA/inputs/temperature\"),  
        "alertsToCloud": "FROM /messages/modules/ASA/* INTO $upstream",  
        "alertsToReset": "FROM /messages/modules/ASA/* INTO  
BrokeredEndpoint(\"/modules/tempSensor/inputs/control\")"  
    }  
}
```

This example shows the routes for the scenario described in the following picture. It contains an edge job called "**ASA**", with an input named "**temperature**" and an output named "**alert**".



This example defines the following routes:

- Every message from the **tempSensor** is sent to the module named **ASA** to the input named **temperature**,
- All outputs of **ASA** module are sent to the IoT Hub linked to this device (**\$upstream**),
- All outputs of **ASA** module are sent to the **control** endpoint of the **tempSensor**.

Technical information

Current limitations for IoT Edge jobs compared to cloud jobs

The goal is to have parity between IoT Edge jobs and cloud jobs. Most SQL query language features are supported, enabling to run the same logic on both cloud and IoT Edge. However the following features are not yet supported for edge jobs:

- User-defined functions (UDF) in JavaScript. UDF are available in [C# for IoT Edge jobs \(preview\)](#).
- User-defined aggregates (UDA).
- Azure ML functions
- Using more than 14 aggregates in a single step.
- AVRO format for input/output. At this time, only CSV and JSON are supported.
- The following SQL operators:
 - PARTITION BY
 - GetMetadataPropertyValue

Runtime and hardware requirements

To run ASA on IoT Edge, you need devices that can run [Azure IoT Edge](#).

ASA and Azure IoT Edge use **Docker** containers to provide a portable solution that runs on multiple host operating systems (Windows, Linux).

ASA on IoT Edge is made available as Windows and Linux images, running on both x86-64 or ARM (Advanced RISC Machines) architectures.

Input and output

Input and Output Streams

ASA Edge jobs can get inputs and outputs from other modules running on IoT Edge devices. To connect from and to specific modules, you can set the routing configuration at deployment time. More information is described on [the IoT Edge module composition documentation](#).

For both inputs and outputs, CSV and JSON formats are supported.

For each input and output stream you create in your ASA job, a corresponding endpoint is created on your deployed module. These endpoints can be used in the routes of your deployment.

At present, the only supported stream input and stream output types are Edge Hub. Reference input supports reference file type. Other outputs can be reached using a cloud job downstream. For example, a Stream Analytics job hosted in Edge sends output to Edge Hub, which can then send output to IoT Hub. You can use a second cloud hosted Azure Stream Analytics job with input from IoT Hub and output to Power BI or another output type.

Reference data

Reference data (also known as a lookup table) is a finite data set that is static or slow changing in nature. It is used to perform a lookup or to correlate with your data stream. To make use of reference data in your Azure Stream Analytics job, you will generally use a [Reference Data JOIN](#) in your query. For more information, see the [Using reference data for lookups in Stream Analytics](#).

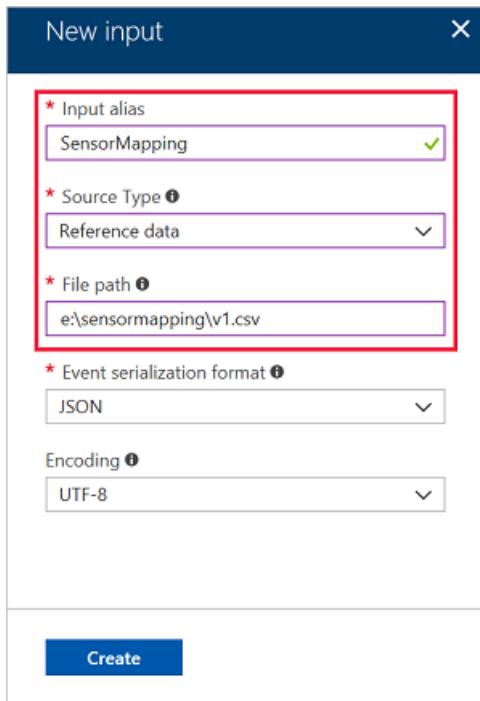
Only local reference data is supported. When a job is deployed to IoT Edge device, it loads reference data from the user defined file path.

To create a job with reference data on Edge:

1. Create a new input for your job.
2. Choose **Reference data** as the **Source Type**.
3. Have a reference data file ready on the device. For a Windows container, put the reference data file on the local drive and share the local drive with the Docker container. For a Linux container, create a Docker volume and populate the data file to the volume.

4. Set the file path. For Windows Host OS and Windows container, use the absolute path:

`E:\<PathToFile>\v1.csv`. For a Windows Host OS and Linux container or a Linux OS and Linux container, use the path in the volume: `<VolumeName>/file1.txt`.



The reference data on IoT Edge update is triggered by a deployment. Once triggered, the ASA module picks the updated data without stopping the running job.

There are two ways to update the reference data:

- Update reference data path in your ASA job from Azure portal.
- Update the IoT Edge deployment.

License and third-party notices

- [Azure Stream Analytics on IoT Edge license](#).
- [Third-party notice for Azure Stream Analytics on IoT Edge](#).

Get help

For further assistance, try the [Azure Stream Analytics forum](#).

Next steps

- [More information on Azure IoT Edge](#)
- [ASA on IoT Edge tutorial](#)
- [Develop Stream Analytics Edge jobs using Visual Studio tools](#)
- [Implement CI/CD for Stream Analytics using APIs](#)

Build an IoT solution by using Stream Analytics

1/15/2019 • 12 minutes to read • [Edit Online](#)

Introduction

In this solution, you learn how to use Azure Stream Analytics to get real-time insights from your data. Developers can easily combine streams of data, such as click-streams, logs, and device-generated events, with historical records or reference data to derive business insights. As a fully managed, real-time stream computation service that's hosted in Microsoft Azure, Azure Stream Analytics provides built-in resiliency, low latency, and scalability to get you up and running in minutes.

After completing this solution, you are able to:

- Familiarize yourself with the Azure Stream Analytics portal.
- Configure and deploy a streaming job.
- Articulate real-world problems and solve them by using the Stream Analytics query language.
- Develop streaming solutions for your customers by using Stream Analytics with confidence.
- Use the monitoring and logging experience to troubleshoot issues.

Prerequisites

You need the following prerequisites to complete this solution:

- An [Azure subscription](#)

Scenario introduction: "Hello, Toll!"

A toll station is a common phenomenon. You encounter them on many expressways, bridges, and tunnels across the world. Each toll station has multiple toll booths. At manual booths, you stop to pay the toll to an attendant. At automated booths, a sensor on top of each booth scans an RFID card that's affixed to the windshield of your vehicle as you pass the toll booth. It is easy to visualize the passage of vehicles through these toll stations as an event stream over which interesting operations can be performed.



Incoming data

This solution works with two streams of data. Sensors installed in the entrance and exit of the toll stations produce

the first stream. The second stream is a static lookup dataset that has vehicle registration data.

Entry data stream

The entry data stream contains information about cars as they enter toll stations. The exit data events are live streamed into an Event Hub queue from a Web App included in the sample app.

| TOLLID | ENTRYTIME | LICENSEPLATE | STATE | MAKE | MODEL | VEHICLETYPE | VEHICLEWEIGHT | TOLL | TAG |
|--------|--------------------------|--------------|-------|--------|---------|-------------|---------------|------|------------|
| 1 | 2014-09-10 12:01:00 .000 | JNB 7001 | NY | Honda | CRV | 1 | 0 | 7 | |
| 1 | 2014-09-10 12:02:00 .000 | YXZ 1001 | NY | Toyota | Camry | 1 | 0 | 4 | 123456 789 |
| 3 | 2014-09-10 12:02:00 .000 | ABC 1004 | CT | Ford | Taurus | 1 | 0 | 5 | 456789 123 |
| 2 | 2014-09-10 12:03:00 .000 | XYZ 1003 | CT | Toyota | Corolla | 1 | 0 | 4 | |
| 1 | 2014-09-10 12:03:00 .000 | BNJ 1007 | NY | Honda | CRV | 1 | 0 | 5 | 789123 456 |
| 2 | 2014-09-10 12:05:00 .000 | CDE 1007 | NJ | Toyota | 4x4 | 1 | 0 | 6 | 321987 654 |

Here is a short description of the columns:

| COLUMN | DESCRIPTION |
|--------------|--------------------------------------------------------------------|
| TollID | The toll booth ID that uniquely identifies a toll booth |
| EntryTime | The date and time of entry of the vehicle to the toll booth in UTC |
| LicensePlate | The license plate number of the vehicle |
| State | A state in United States |
| Make | The manufacturer of the automobile |
| Model | The model number of the automobile |
| VehicleType | Either 1 for passenger vehicles or 2 for commercial vehicles |

| COLUMN | DESCRIPTION |
|------------|-----------------------------------------------------------------------------------------------|
| WeightType | Vehicle weight in tons; 0 for passenger vehicles |
| Toll | The toll value in USD |
| Tag | The e-Tag on the automobile that automates payment; blank where the payment was done manually |

Exit data stream

The exit data stream contains information about cars leaving the toll station. The exit data events are live streamed into an Event Hub queue from a Web App included in the sample app.

| TOLLID | EXITTIME | LICENSEPLATE |
|--------|------------------------------|--------------|
| 1 | 2014-09-10T12:03:00.0000000Z | JNB 7001 |
| 1 | 2014-09-10T12:03:00.0000000Z | YXZ 1001 |
| 3 | 2014-09-10T12:04:00.0000000Z | ABC 1004 |
| 2 | 2014-09-10T12:07:00.0000000Z | XYZ 1003 |
| 1 | 2014-09-10T12:08:00.0000000Z | BNJ 1007 |
| 2 | 2014-09-10T12:07:00.0000000Z | CDE 1007 |

Here is a short description of the columns:

| COLUMN | DESCRIPTION |
|--------------|-----------------------------------------------------------------|
| TollID | The toll booth ID that uniquely identifies a toll booth |
| ExitTime | The date and time of exit of the vehicle from toll booth in UTC |
| LicensePlate | The license plate number of the vehicle |

Commercial vehicle registration data

The solution uses a static snapshot of a commercial vehicle registration database. This data is saved as a JSON file into Azure blob storage, included in the sample.

| LICENSEPLATE | REGISTRATIONID | EXPIRED |
|--------------|----------------|---------|
| SVT 6023 | 285429838 | 1 |
| XLZ 3463 | 362715656 | 0 |
| BAC 1005 | 876133137 | 1 |
| RIV 8632 | 992711956 | 0 |
| SNY 7188 | 592133890 | 0 |

| LICENSEPLATE | REGISTRATIONID | EXPIRED |
|--------------|----------------|---------|
| ELH 9896 | 678427724 | 1 |

Here is a short description of the columns:

| COLUMN | DESCRIPTION |
|----------------|-----------------------------------------------------------------------------------------------------------|
| LicensePlate | The license plate number of the vehicle |
| RegistrationId | The vehicle's registration ID |
| Expired | The registration status of the vehicle: 0 if vehicle registration is active, 1 if registration is expired |

Set up the environment for Azure Stream Analytics

To complete this solution, you need a Microsoft Azure subscription. If you do not have an Azure account, you can [request a free trial version](#).

Be sure to follow the steps in the "Clean up your Azure account" section at the end of this article so that you can make the best use of your Azure credit.

Deploy the sample

There are several resources that can easily be deployed in a resource group together with a few clicks. The solution definition is hosted in GitHub repository at <https://github.com/Azure/azure-stream-analytics/tree/master/Samples/TollApp>.

Deploy the TollApp template in the Azure portal

1. To deploy the TollApp environment to Azure, use this link to [Deploy TollApp Azure Template](#).
2. Sign in to the Azure portal if prompted.
3. Choose the subscription in which the various resources are billed.
4. Specify a new resource group, with a unique name, for example `MyTollBooth`.
5. Select an Azure location.
6. Specify an **Interval** as a number of seconds. This value is used in the sample web app, for how frequently to send data into Event Hub.
7. **Check** to agree to the terms and conditions.
8. Select **Pin to dashboard** so that you can easily locate the resources later on.
9. Select **Purchase** to deploy the sample template.
10. After a few moments, a notification appears to confirm the **Deployment succeeded**.

Review the Azure Stream Analytics TollApp resources

1. Log in to the Azure portal
2. Locate the Resource Group that you named in the previous section.
3. Verify that the following resources are listed in the resource group:

- One Cosmos DB Account
- One Azure Stream Analytics Job
- One Azure Storage Account
- One Azure Event Hub
- Two Web Apps

Examine the sample TollApp job

1. Starting from the resource group in the previous section, select the Stream Analytics streaming job starting with the name **tollapp** (name contains random characters for uniqueness).
2. On the **Overview** page of the job, notice the **Query** box to view the query syntax.

```
SELECT TollId, System.Timestamp AS WindowEnd, COUNT(*) AS Count
INTO CosmosDB
FROM EntryStream TIMESTAMP BY EntryTime
GROUP BY TUMBLINGWINDOW(minute, 3), TollId
```

To paraphrase the intent of the query, let's say that you need to count the number of vehicles that enter a toll booth. Because a highway toll booth has a continuous stream of vehicles entering, those entrance events are analogous to a stream that never stops. To quantify the stream, you have to define a "period of time" to measure over. Let's refine the question further, to "How many vehicles enter a toll booth every three minutes?" This is commonly referred to as the tumbling count.

As you can see, Azure Stream Analytics uses a query language that's like SQL and adds a few extensions to specify time-related aspects of the query. For more details, read about [Time Management](#) and [Windowing](#) constructs used in the query.

3. Examine the Inputs of the TollApp sample job. Only the EntryStream input is used in the current query.
 - **EntryStream** input is an Event Hub connection that queues data representing each time a car enters a tollbooth on the highway. A web app that is part of the sample is creating the events, and that data is queued in this Event Hub. Note that this input is queried in the FROM clause of the streaming query.
 - **ExitStream** input is an Event Hub connection that queues data representing each time a car exits a tollbooth on the highway. This streaming input is used in later variations of the query syntax.
 - **Registration** input is an Azure Blob storage connection, pointing to a static registration.json file, used for lookups as needed. This reference data input is used in later variations of the query syntax.
4. Examine the Outputs of the TollApp sample job.
 - **Cosmos DB** output is a Cosmos database collection that receives the output sink events. Note that this output is used in INTO clause of the streaming query.

Start the TollApp streaming job

Follow these steps to start the streaming job:

1. On the **Overview** page of the job, select **Start**.
2. On the **Start job** pane, select **Now**.
3. After a few moments, once the job is running, on the **Overview** page of the streaming job, view the **Monitoring** graph. The graph should show several thousand input events, and tens of output events.

Review the CosmosDB output data

1. Locate the resource group that contains the TollApp resources.
2. Select the Azure Cosmos DB Account with the name pattern **tollapp<random>-cosmos**.
3. Select the **Data Explorer** heading to open the Data Explorer page.
4. Expand the **tollAppDatabase > tollAppCollection > Documents**.
5. In the list of ids, several docs are shown once the output is available.
6. Select each id to review the JSON document. Notice each tollid, windowend time, and the count of cars from that window.
7. After an additional three minutes, another set of four documents is available, one document per tollid.

Report total time for each car

The average time that's required for a car to pass through the toll helps to assess the efficiency of the process and the customer experience.

To find the total time, join the EntryTime stream with the ExitTime stream. Join the two input streams on the equal matching TollId and LicencePlate columns. The **JOIN** operator requires you to specify temporal leeway that describes the acceptable time difference between the joined events. Use the **DATEDIFF** function to specify that events should be no more than 15 minutes from each other. Also apply the **DATEDIFF** function to exit and entry times to compute the actual time that a car spends in the toll station. Note the difference of the use of **DATEDIFF** when it's used in a **SELECT** statement rather than a **JOIN** condition.

```
SELECT EntryStream.TollId, EntryStream.EntryTime, ExitStream.ExitTime, EntryStream.LicensePlate, DATEDIFF
(minute, EntryStream.EntryTime, ExitStream.ExitTime) AS DurationInMinutes
INTO CosmosDB
FROM EntryStream TIMESTAMP BY EntryTime
JOIN ExitStream TIMESTAMP BY ExitTime
ON (EntryStream.TollId= ExitStream.TollId AND EntryStream.LicensePlate = ExitStream.LicensePlate)
AND DATEDIFF (minute, EntryStream, ExitStream ) BETWEEN 0 AND 15
```

To update the TollApp streaming job query syntax:

1. On the **Overview** page of the job, select **Stop**.
2. Wait a few moments for the notification that the job has stopped.
3. Under the **JOB TOPOLOGY** heading, select < > **Query**
4. Paste the adjusted streaming SQL query.
5. Select **Save** to save the query. Confirm **Yes** to save the changes.
6. On the **Overview** page of the job, select **Start**.
7. On the **Start job** pane, select **Now**.

Review the total time in the output

Repeat the steps in the preceding section to review the CosmosDB output data from the streaming job. Review the latest JSON documents.

For example, this document shows an example car with a certain license plate, the entrytime and exit time, and the DATEDIFF calculated durationinminutes field showing the toll booth duration as two minutes:

```
{
    "tollid": 4,
    "entrytime": "2018-04-05T06:51:39.0491173Z",
    "exittime": "2018-04-05T06:53:09.0491173Z",
    "licenseplate": "JVR 9425",
    "durationinminutes": 2,
    "id": "ff52eb25-d580-7566-2879-1f52bba6601e",
    "_rid": "+8E4AI1DZgBjAAAAAAA==",
    "_self": "dbs/+8E4AA==/colls/+8E4AI1DZgA=/docs/+8E4AI1DZgBjAAAAAAA==/",
    "_etag": "\"ad02f6b8-0000-0000-0000-5ac5c8330000\"",
    "_attachments": "attachments/",
    "_ts": 1522911283
}
```

Report vehicles with expired registration

Azure Stream Analytics can use static snapshots of reference data to join with temporal data streams. To demonstrate this capability, use the following sample query. The Registration input is a static blob json file that lists the expirations of license tags. By joining on the license plate, the reference data is compared to each vehicle passing through the toll both.

If a commercial vehicle is registered with the toll company, it can pass through the toll booth without being stopped for inspection. Use the registration lookup table to identify all commercial vehicles that have expired registrations.

```
SELECT EntryStream.EntryTime, EntryStream.LicensePlate, EntryStream.TollId, Registration.RegistrationId
INTO CosmosDB
FROM EntryStream TIMESTAMP BY EntryTime
JOIN Registration
ON EntryStream.LicensePlate = Registration.LicensePlate
WHERE Registration.Expired = '1'
```

1. Repeat the steps in the preceding section to update the TollApp streaming job query syntax.
2. Repeat the steps in the preceding section to review the CosmosDB output data from the streaming job.

Example output:

```
{
    "entrytime": "2018-04-05T08:01:28.0252168Z",
    "licenseplate": "GMT 3221",
    "tollid": 1,
    "registrationid": "763220582",
    "id": "47db0535-9716-4eb2-db58-de7886966cbf",
    "_rid": "y+F8AJ9QWACSAQAAAAAAA==",
    "_self": "dbs/y+F8AA==/colls/y+F8AJ9QWAA=/docs/y+F8AJ9QWACSAQAAAAAAA==/",
    "_etag": "\"88007d8d-0000-0000-0000-5ac5d7e20000\"",
    "_attachments": "attachments/",
    "_ts": 1522915298
}
```

Scale out the job

Azure Stream Analytics is designed to elastically scale so that it can handle large volumes of data. The Azure Stream Analytics query can use a **PARTITION BY** clause to tell the system that this step scales out. **PartitionId** is a special column that the system adds to match the partition ID of the input (event hub).

To scale out the query to partitions, edit the query syntax to the following code:

```

SELECT TollId, System.Timestamp AS WindowEnd, COUNT(*)AS Count
INTO CosmosDB
FROM EntryStream
TIMESTAMP BY EntryTime
PARTITION BY PartitionId
GROUP BY TUMBLINGWINDOW(minute,3), TollId, PartitionId

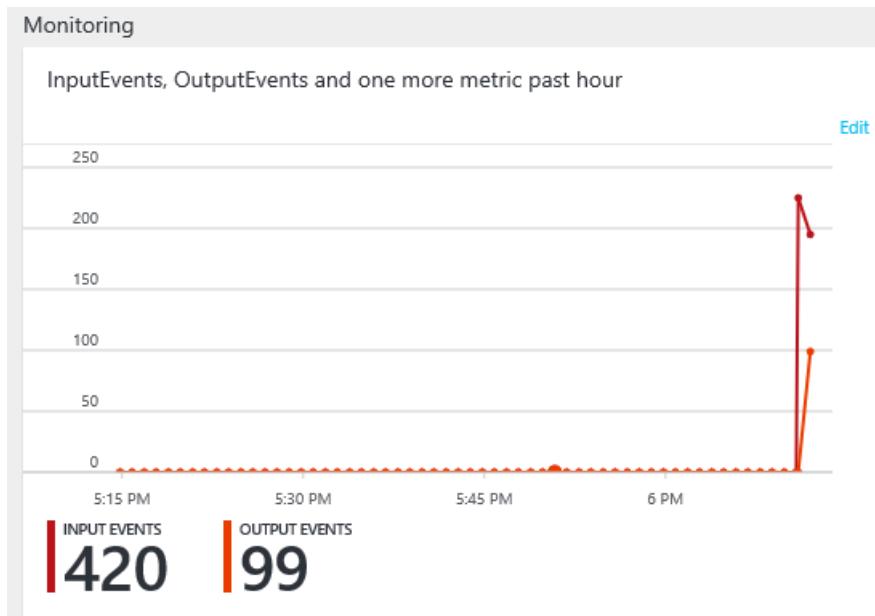
```

To scale up the streaming job to more streaming units:

1. **Stop** the current job.
2. Update the query syntax in the < > **Query** page, and save the changes.
3. Under the CONFIGURE heading on the streaming job, select **Scale**.
4. Slide the **Streaming units** slider from 1 to 6. Streaming units define the amount of compute power that the job can receive. Select **Save**.
5. **Start** the streaming job to demonstrate the additional scale. Azure Stream Analytics distributes work across more compute resources and achieve better throughput, partitioning the work across resources using the column designated in the PARTITION BY clause.

Monitor the job

The **MONITOR** area contains statistics about the running job. First-time configuration is needed to use the storage account in the same region (name toll like the rest of this document).



You can access **Activity Logs** from the job dashboard **Settings** area as well.

Clean up the TollApp resources

1. Stop the Stream Analytics job in the Azure portal.
2. Locate the resource group that contains eight resources related to the TollApp template.
3. Select **Delete resource group**. Type the name of the resource group to confirm deletion.

Conclusion

This solution introduced you to the Azure Stream Analytics service. It demonstrated how to configure inputs and

outputs for the Stream Analytics job. Using the Toll Data scenario, the solution explained common types of problems that arise in the space of data in motion and how they can be solved with simple SQL-like queries in Azure Stream Analytics. The solution described SQL extension constructs for working with temporal data. It showed how to join data streams, how to enrich the data stream with static reference data, and how to scale out a query to achieve higher throughput.

Although this solution provides a good introduction, it is not complete by any means. You can find more query patterns using the SAQL language at [Query examples for common Stream Analytics usage patterns](#).

Azure Stream Analytics JavaScript user-defined aggregates (Preview)

1/28/2019 • 5 minutes to read • [Edit Online](#)

Azure Stream Analytics supports user-defined aggregates (UDA) written in JavaScript, it enables you to implement complex stateful business logic. Within UDA you have full control of the state data structure, state accumulation, state decumulation, and aggregate result computation. The article introduces the two different JavaScript UDA interfaces, steps to create a UDA, and how to use UDA with window-based operations in Stream Analytics query.

JavaScript user-defined aggregates

A user-defined aggregate is used on top of a time window specification to aggregate over the events in that window and produce a single result value. There are two types of UDA interfaces that Stream Analytics supports today, AccumulateOnly and AccumulateDeaccumulate. Both types of UDA can be used by Tumbling Window, Hopping Window, and Sliding Window. AccumulateDeaccumulate UDA performs better than AccumulateOnly UDA when used together with Hopping Window and Sliding Window. You choose one of the two types based on the algorithm you use.

AccumulateOnly aggregates

AccumulateOnly aggregates can only accumulate new events to its state, the algorithm does not allow deaccumulation of values. Choose this aggregate type when deaccumulate an event information from the state value is impossible to implement. Following is the JavaScript template for AccumulateOnly aggregates:

```
// Sample UDA which state can only be accumulated.
function main() {
    this.init = function () {
        this.state = 0;
    }

    this.accumulate = function (value, timestamp) {
        this.state += value;
    }

    this.computeResult = function () {
        return this.state;
    }
}
```

AccumulateDeaccumulate aggregates

AccumulateDeaccumulate aggregates allow deaccumulation of a previous accumulated value from the state, for example, remove a key-value pair from a list of event values, or subtract a value from a state of sum aggregate. Following is the JavaScript template for AccumulateDeaccumulate aggregates:

```

// Sample UDA which state can be accumulated and deaccumulated.
function main() {
    this.init = function () {
        this.state = 0;
    }

    this.accumulate = function (value, timestamp) {
        this.state += value;
    }

    this.deaccumulate = function (value, timestamp) {
        this.state -= value;
    }

    this.deaccumulateState = function (otherState){
        this.state -= otherState.state;
    }

    this.computeResult = function () {
        return this.state;
    }
}

```

UDA - JavaScript function declaration

Each JavaScript UDA is defined by a Function object declaration. Following are the major elements in a UDA definition.

Function alias

Function alias is the UDA identifier. When called in Stream Analytics query, always use UDA alias together with a "uda." prefix.

Function type

For UDA, the function type should be **Javascript UDA**.

Output type

A specific type that Stream Analytics job supported, or "Any" if you want to handle the type in your query.

Function name

The name of this Function object. The function name should literally match the UDA alias (preview behavior, we are considering support anonymous function when GA).

Method - init()

The init() method initializes state of the aggregate. This method is called when window starts.

Method – accumulate()

The accumulate() method calculates the UDA state based on the previous state and the current event values. This method is called when an event enters a time window (TUMBLINGWINDOW, HOPPINGWINDOW, or SLIDINGWINDOW).

Method – deaccumulate()

The deaccumulate() method recalculates state based on the previous state and the current event values. This method is called when an event leaves a SLIDINGWINDOW.

Method – deaccumulateState()

The deaccumulateState() method recalculates state based on the previous state and the state of a hop. This method is called when a set of events leave a HOPPINGWINDOW.

Method – computeResult()

The computeResult() method returns aggregate result based on the current state. This method is called at the end of a time window (TUMBLINGWINDOW, HOPPINGWINDOW, and SLIDINGWINDOW).

JavaScript UDA supported input and output data types

For JavaScript UDA data types, refer to section **Stream Analytics and JavaScript type conversion** of [Integrate JavaScript UDFs](#).

Adding a JavaScript UDA from the Azure portal

Below we walk through the process of creating a UDA from Portal. The example we use here is computing time weighted averages.

Now let's create a JavaScript UDA under an existing ASA job by following steps.

1. Log on to Azure portal and locate your existing Stream Analytics job.
2. Then click on functions link under **JOB TOPOLOGY**.
3. Click on the **Add** icon to add a new function.
4. On the New Function view, select **JavaScript UDA** as the Function Type, then you see a default UDA template show up in the editor.
5. Fill in "TWA" as the UDA alias and change the function implementation as the following:

```

// Sample UDA which calculate Time-Weighted Average of incoming values.
function main() {
    this.init = function () {
        this.totalValue = 0.0;
        this.totalWeight = 0.0;
    }

    this.accumulate = function (value, timestamp) {
        this.totalValue += value.level * value.weight;
        this.totalWeight += value.weight;
    }

    // Uncomment below for AccumulateDeaccumulate implementation
    /*
    this.deaccumulate = function (value, timestamp) {
        this.totalValue -= value.level * value.weight;
        this.totalWeight -= value.weight;
    }

    this.deaccumulateState = function (otherState){
        this.state -= otherState.state;
        this.totalValue -= otherState.totalValue;
        this.totalWeight -= otherState.totalWeight;
    }
    */

    this.computeResult = function () {
        if(this.totalValue == 0) {
            result = 0;
        }
        else {
            result = this.totalValue/this.totalWeight;
        }
        return result;
    }
}

```

6. Once you click the "Save" button, your UDA shows up on the function list.
7. Click on the new function "TWA", you can check the function definition.

Calling JavaScript UDA in ASA query

In Azure portal and open your job, edit the query and call TWA() function with a mandate prefix "uda.". For example:

```

WITH value AS
(
    SELECT
        NoiseLevelDB as level,
        DurationSecond as weight
    FROM
        [YourInputAlias] TIMESTAMP BY EntryTime
)
SELECT
    System.Timestamp as ts,
    uda.TWA(value) as NoseDoseTWA
FROM value
GROUP BY TumblingWindow(minute, 5)

```

Testing query with UDA

Create a local JSON file with below content, upload the file to Stream Analytics job, and test above query.

```
[  
    {"EntryTime": "2017-06-10T05:01:00-07:00", "NoiseLevelDB": 80, "DurationSecond": 22.0},  
    {"EntryTime": "2017-06-10T05:02:00-07:00", "NoiseLevelDB": 81, "DurationSecond": 37.8},  
    {"EntryTime": "2017-06-10T05:02:00-07:00", "NoiseLevelDB": 85, "DurationSecond": 26.3},  
    {"EntryTime": "2017-06-10T05:03:00-07:00", "NoiseLevelDB": 95, "DurationSecond": 13.7},  
    {"EntryTime": "2017-06-10T05:03:00-07:00", "NoiseLevelDB": 88, "DurationSecond": 10.3},  
    {"EntryTime": "2017-06-10T05:05:00-07:00", "NoiseLevelDB": 103, "DurationSecond": 5.5},  
    {"EntryTime": "2017-06-10T05:06:00-07:00", "NoiseLevelDB": 99, "DurationSecond": 23.0},  
    {"EntryTime": "2017-06-10T05:07:00-07:00", "NoiseLevelDB": 108, "DurationSecond": 1.76},  
    {"EntryTime": "2017-06-10T05:07:00-07:00", "NoiseLevelDB": 79, "DurationSecond": 17.9},  
    {"EntryTime": "2017-06-10T05:08:00-07:00", "NoiseLevelDB": 83, "DurationSecond": 27.1},  
    {"EntryTime": "2017-06-10T05:09:00-07:00", "NoiseLevelDB": 91, "DurationSecond": 17.1},  
    {"EntryTime": "2017-06-10T05:09:00-07:00", "NoiseLevelDB": 115, "DurationSecond": 7.9},  
    {"EntryTime": "2017-06-10T05:09:00-07:00", "NoiseLevelDB": 80, "DurationSecond": 28.3},  
    {"EntryTime": "2017-06-10T05:10:00-07:00", "NoiseLevelDB": 55, "DurationSecond": 18.2},  
    {"EntryTime": "2017-06-10T05:10:00-07:00", "NoiseLevelDB": 93, "DurationSecond": 25.8},  
    {"EntryTime": "2017-06-10T05:11:00-07:00", "NoiseLevelDB": 83, "DurationSecond": 11.4},  
    {"EntryTime": "2017-06-10T05:12:00-07:00", "NoiseLevelDB": 89, "DurationSecond": 7.9},  
    {"EntryTime": "2017-06-10T05:15:00-07:00", "NoiseLevelDB": 112, "DurationSecond": 3.7},  
    {"EntryTime": "2017-06-10T05:15:00-07:00", "NoiseLevelDB": 93, "DurationSecond": 9.7},  
    {"EntryTime": "2017-06-10T05:18:00-07:00", "NoiseLevelDB": 96, "DurationSecond": 3.7},  
    {"EntryTime": "2017-06-10T05:20:00-07:00", "NoiseLevelDB": 108, "DurationSecond": 0.99},  
    {"EntryTime": "2017-06-10T05:20:00-07:00", "NoiseLevelDB": 113, "DurationSecond": 25.1},  
    {"EntryTime": "2017-06-10T05:22:00-07:00", "NoiseLevelDB": 110, "DurationSecond": 5.3}  
]
```

Get help

For additional help, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics query language reference](#)
- [Azure Stream Analytics management REST API reference](#)

High-frequency trading simulation with Stream Analytics

12/7/2018 • 12 minutes to read • [Edit Online](#)

The combination of SQL language and JavaScript user-defined functions (UDFs) and user-defined aggregates (UDAs) in Azure Stream Analytics enables users to perform advanced analytics. Advanced analytics might include online machine learning training and scoring, as well as stateful process simulation. This article describes how to perform linear regression in an Azure Stream Analytics job that does continuous training and scoring in a high-frequency trading scenario.

High-frequency trading

The logical flow of high-frequency trading is about:

1. Getting real-time quotes from a security exchange.
2. Building a predictive model around the quotes, so we can anticipate the price movement.
3. Placing buy or sell orders to make money from the successful prediction of the price movements.

As a result, we need:

- A real-time quote feed.
- A predictive model that can operate on the real-time quotes.
- A trading simulation that demonstrates the profit or loss of the trading algorithm.

Real-time quote feed

IEX offers free [real-time bid and ask quotes](#) by using socket.io. A simple console program can be written to receive real-time quotes and push to Azure Event Hubs as a data source. The following code is a skeleton of the program.

The code omits error handling for brevity. You also need to include SocketIoClientDotNet and

WindowsAzure.ServiceBus NuGet packages in your project.

```
using Quobject.SocketIoClientDotNet.Client;
using Microsoft.ServiceBus.Messaging;
var symbols = "msft,fb,amzn,goog";
var eventHubClient = EventHubClient.CreateFromConnectionString(connectionString, eventHubName);
var socket = IO.Socket("https://ws-api.iextrading.com/1.0/tops");
socket.On(Socket.EVENT_MESSAGE, (message) =>
{
    eventHubClient.Send(new EventData(Encoding.UTF8.GetBytes((string)message)));
});
socket.On(Socket.EVENT_CONNECT, () =>
{
    socket.Emit("subscribe", symbols);
});
```

Here are some generated sample events:

```
{
  "symbol": "MSFT", "marketPercent": 0.03246, "bidSize": 100, "bidPrice": 74.8, "askSize": 300, "askPrice": 74.83, "volume": 70572, "lastSalePrice": 74.825, "lastSaleSize": 100, "lastSaleTime": 1506953355123, "lastUpdated": 1506953357170, "sector": "software/services", "securityType": "commonstock"}, {"symbol": "GOOG", "marketPercent": 0.04825, "bidSize": 114, "bidPrice": 870, "askSize": 0, "askPrice": 0, "volume": 11240, "lastSalePrice": 959.47, "lastSaleSize": 60, "lastSaleTime": 1506953317571, "lastUpdated": 1506953357633, "sector": "software/services", "securityType": "commonstock"}, {"symbol": "MSFT", "marketPercent": 0.03244, "bidSize": 100, "bidPrice": 74.8, "askSize": 100, "askPrice": 74.83, "volume": 70572, "lastSalePrice": 74.825, "lastSaleSize": 100, "lastSaleTime": 1506953355123, "lastUpdated": 1506953359118, "sector": "software/services", "securityType": "commonstock"}, {"symbol": "FB", "marketPercent": 0.01211, "bidSize": 100, "bidPrice": 169.9, "askSize": 100, "askPrice": 170.67, "volume": 39042, "lastSalePrice": 170.67, "lastSaleSize": 100, "lastSaleTime": 1506953351912, "lastUpdated": 1506953359641, "sector": "software/services", "securityType": "commonstock"}, {"symbol": "GOOG", "marketPercent": 0.04795, "bidSize": 100, "bidPrice": 959.19, "askSize": 0, "askPrice": 0, "volume": 11240, "lastSalePrice": 959.47, "lastSaleSize": 60, "lastSaleTime": 1506953317571, "lastUpdated": 1506953360949, "sector": "software/services", "securityType": "commonstock"}, {"symbol": "FB", "marketPercent": 0.0121, "bidSize": 100, "bidPrice": 169.9, "askSize": 100, "askPrice": 170.7, "volume": 39042, "lastSalePrice": 170.67, "lastSaleSize": 100, "lastSaleTime": 1506953351912, "lastUpdated": 1506953362205, "sector": "software/services", "securityType": "commonstock"}, {"symbol": "GOOG", "marketPercent": 0.04795, "bidSize": 114, "bidPrice": 870, "askSize": 0, "askPrice": 0, "volume": 11240, "lastSalePrice": 959.47, "lastSaleSize": 60, "lastSaleTime": 1506953317571, "lastUpdated": 1506953362629, "sector": "software/services", "securityType": "commonstock"}}
```

NOTE

The time stamp of the event is **lastUpdated**, in epoch time.

Predictive model for high-frequency trading

For the purpose of demonstration, we use a linear model described by Darryl Shen in [his paper](#).

Volume order imbalance (VOI) is a function of current bid/ask price and volume, and bid/ask price and volume from the last tick. The paper identifies the correlation between VOI and future price movement. It builds a linear model between the past 5 VOI values and the price change in the next 10 ticks. The model is trained by using previous day's data with linear regression.

The trained model is then used to make price change predictions on quotes in the current trading day in real time. When a large enough price change is predicted, a trade is executed. Depending on the threshold setting, thousands of trades can be expected for a single stock during a trading day.

$$OI_t = \delta V_t^B - \delta V_t^A$$

$$\delta V_t^B = \begin{cases} 0, & P_t^B < P_{t-1}^B \\ V_t^B - V_{t-1}^B, & P_t^B = P_{t-1}^B \\ V_t^B, & P_t^B > P_{t-1}^B \end{cases}, \quad \delta V_t^A = \begin{cases} V_t^A, & P_t^A < P_{t-1}^A \\ V_t^A - V_{t-1}^A, & P_t^A = P_{t-1}^A \\ 0, & P_t^A > P_{t-1}^A \end{cases}$$

Now, let's express the training and prediction operations in an Azure Stream Analytics job.

First, the inputs are cleaned up. Epoch time is converted to datetime via **DATEADD.TRY_CAST** is used to coerce data types without failing the query. It's always a good practice to cast input fields to the expected data types, so there is no unexpected behavior in manipulation or comparison of the fields.

```

WITH
typeconvertedquotes AS (
    /* convert all input fields to proper types */
    SELECT
        System.Timestamp AS lastUpdated,
        symbol,
        DATEADD(millisecond, CAST(lastSaleTime as bigint), '1970-01-01T00:00:00Z') AS lastSaleTime,
        TRY_CAST(bidSize as bigint) AS bidSize,
        TRY_CAST(bidPrice as float) AS bidPrice,
        TRY_CAST(askSize as bigint) AS askSize,
        TRY_CAST(askPrice as float) AS askPrice,
        TRY_CAST(volume as bigint) AS volume,
        TRY_CAST(lastSaleSize as bigint) AS lastSaleSize,
        TRY_CAST(lastSalePrice as float) AS lastSalePrice
    FROM quotes TIMESTAMP BY DATEADD(millisecond, CAST(lastUpdated as bigint), '1970-01-01T00:00:00Z')
),
timefilteredquotes AS (
    /* filter between 7am and 1pm PST, 14:00 to 20:00 UTC */
    /* clean up invalid data points */
    SELECT * FROM typeconvertedquotes
    WHERE DATEPART(hour, lastUpdated) >= 14 AND DATEPART(hour, lastUpdated) < 20 AND bidSize > 0 AND askSize > 0
    AND bidPrice > 0 AND askPrice > 0
),

```

Next, we use the **LAG** function to get values from the last tick. One hour of **LIMIT DURATION** value is arbitrarily chosen. Given the quote frequency, it's safe to assume that you can find the previous tick by looking back one hour.

```

shiftedquotes AS (
    /* get previous bid/ask price and size in order to calculate VOI */
    SELECT
        symbol,
        (bidPrice + askPrice)/2 AS midPrice,
        bidPrice,
        bidSize,
        askPrice,
        askSize,
        LAG(bidPrice) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS bidPricePrev,
        LAG(bidSize) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS bidSizePrev,
        LAG(askPrice) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS askPricePrev,
        LAG(askSize) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS askSizePrev
    FROM timefilteredquotes
),

```

We can then compute VOI value. We filter out the null values if the previous tick doesn't exist, just in case.

```

currentPriceAndVOI AS (
    /* calculate VOI */
    SELECT
        symbol,
        midPrice,
        (CASE WHEN (bidPrice < bidPricePrev) THEN 0
            ELSE (CASE WHEN (bidPrice = bidPricePrev) THEN (bidSize - bidSizePrev) ELSE bidSize END)
            END) -
        (CASE WHEN (askPrice < askPricePrev) THEN askSize
            ELSE (CASE WHEN (askPrice = askPricePrev) THEN (askSize - askSizePrev) ELSE 0 END)
            END) AS VOI
    FROM shiftedquotes
    WHERE
        bidPrice IS NOT NULL AND
        bidSize IS NOT NULL AND
        askPrice IS NOT NULL AND
        askSize IS NOT NULL AND
        bidPricePrev IS NOT NULL AND
        bidSizePrev IS NOT NULL AND
        askPricePrev IS NOT NULL AND
        askSizePrev IS NOT NULL
),

```

Now, we use **LAG** again to create a sequence with 2 consecutive VOI values, followed by 10 consecutive mid-price values.

```

shiftedPriceAndShiftedVOI AS (
    /* get 10 future prices and 2 previous VOIs */
    SELECT
        symbol,
        midPrice AS midPrice10,
        LAG(midPrice, 1) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS midPrice9,
        LAG(midPrice, 2) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS midPrice8,
        LAG(midPrice, 3) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS midPrice7,
        LAG(midPrice, 4) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS midPrice6,
        LAG(midPrice, 5) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS midPrice5,
        LAG(midPrice, 6) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS midPrice4,
        LAG(midPrice, 7) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS midPrice3,
        LAG(midPrice, 8) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS midPrice2,
        LAG(midPrice, 9) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS midPrice1,
        LAG(midPrice, 10) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS midPrice,
        LAG(VOI, 10) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS VOI1,
        LAG(VOI, 11) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS VOI2
    FROM currentPriceAndVOI
),

```

We then reshape the data into inputs for a two-variable linear model. Again, we filter out the events where we don't have all the data.

```

modelInput AS (
    /* create feature vector, x being VOI, y being delta price */
    SELECT
        symbol,
        (midPrice1 + midPrice2 + midPrice3 + midPrice4 + midPrice5 + midPrice6 + midPrice7 + midPrice8 + midPrice9 +
        midPrice10)/10.0 - midPrice AS y,
        VOI1 AS x1,
        VOI2 AS x2
    FROM shiftedPriceAndShiftedVOI
    WHERE
        midPrice1 IS NOT NULL AND
        midPrice2 IS NOT NULL AND
        midPrice3 IS NOT NULL AND
        midPrice4 IS NOT NULL AND
        midPrice5 IS NOT NULL AND
        midPrice6 IS NOT NULL AND
        midPrice7 IS NOT NULL AND
        midPrice8 IS NOT NULL AND
        midPrice9 IS NOT NULL AND
        midPrice10 IS NOT NULL AND
        midPrice IS NOT NULL AND
        VOI1 IS NOT NULL AND
        VOI2 IS NOT NULL
),

```

Because Azure Stream Analytics doesn't have a built-in linear regression function, we use **SUM** and **AVG** aggregates to compute the coefficients for the linear model.

$$y = b_1x_1 + b_2x_2 + a$$

$$b_1 = \frac{\sum x_2^2 \sum x_1 y - \sum x_1 x_2 \sum x_2 y}{\sum x_1^2 \sum x_2^2 - \sum x_1 x_2 \sum x_1 x_2}$$

$$b_2 = \frac{\sum x_1^2 \sum x_2 y - \sum x_1 x_2 \sum x_1 y}{\sum x_1^2 \sum x_2^2 - \sum x_1 x_2 \sum x_1 x_2}$$

$$a = \frac{\sum y}{n} - b_1 \frac{\sum x_1}{n} - b_2 \frac{\sum x_2}{n}$$

```

modelagg AS (
    /* get aggregates for linear regression calculation,
    http://faculty.cas.usf.edu/mbrannick/regression/Reg2IV.html */
    SELECT
        symbol,
        SUM(x1 * x1) AS x1x1,
        SUM(x2 * x2) AS x2x2,
        SUM(x1 * y) AS x1y,
        SUM(x2 * y) AS x2y,
        SUM(x1 * x2) AS x1x2,
        AVG(y) AS avgy,
        AVG(x1) AS avgx1,
        AVG(x2) AS avgx2
    FROM modelInput
    GROUP BY symbol, TumblingWindow(hour, 24, -4)
),
modelparams AS (
    /* calculate b1 and b2 for the linear model */
    SELECT
        symbol,
        (x2x2 * x1y - x1x2 * x2y)/(x1x1 * x2x2 - x1x2 * x1x2) AS b1,
        (x1x1 * x2y - x1x2 * x1y)/(x1x1 * x2x2 - x1x2 * x1x2) AS b2,
        avgy,
        avgx1,
        avgx2
    FROM modelagg
),
model AS (
    /* calculate a for the linear model */
    SELECT
        symbol,
        avgy - b1 * avgx1 - b2 * avgx2 AS a,
        b1,
        b2
    FROM modelparams
),

```

To use the previous day's model for current event's scoring, we want to join the quotes with the model. But instead of using **JOIN**, we **UNION** the model events and quote events. Then we use **LAG** to pair the events with previous day's model, so we can get exactly one match. Because of the weekend, we have to look back three days. If we used a straightforward **JOIN**, we would get three models for every quote event.

```

shiftedVOI AS (
    /* get two consecutive VOIs */
    SELECT
        symbol,
        midPrice,
        VOI AS VOI1,
        LAG(VOI, 1) OVER (PARTITION BY symbol LIMIT DURATION(hour, 1)) AS VOI2
    FROM currentPriceAndVOI
),
VOIAndModel AS (
    /* combine VOIs and models */
    SELECT
        'voi' AS type,
        symbol,
        midPrice,
        VOI1,
        VOI2,
        0.0 AS a,
        0.0 AS b1,
        0.0 AS b2
    FROM shiftedVOI
    UNION
    SELECT
        'model' AS type,
        symbol,
        0.0 AS midPrice,
        0 AS VOI1,
        0 AS VOI2,
        a,
        b1,
        b2
    FROM model
),
VOIANDModelJoined AS (
    /* match VOIs with the latest model within 3 days (72 hours, to take the weekend into account) */
    SELECT
        symbol,
        midPrice,
        VOI1 as x1,
        VOI2 as x2,
        LAG(a, 1) OVER (PARTITION BY symbol LIMIT DURATION(hour, 72) WHEN type = 'model') AS a,
        LAG(b1, 1) OVER (PARTITION BY symbol LIMIT DURATION(hour, 72) WHEN type = 'model') AS b1,
        LAG(b2, 1) OVER (PARTITION BY symbol LIMIT DURATION(hour, 72) WHEN type = 'model') AS b2
    FROM VOIAndModel
    WHERE type = 'voi'
),

```

Now, we can make predictions and generate buy/sell signals based on the model, with a 0.02 threshold value. A trade value of 10 is buy. A trade value of -10 is sell.

```

prediction AS (
    /* make prediction if there is a model */
    SELECT
        symbol,
        midPrice,
        a + b1 * x1 + b2 * x2 AS efpc
    FROM VOIANDModelJoined
    WHERE
        a IS NOT NULL AND
        b1 IS NOT NULL AND
        b2 IS NOT NULL AND
        x1 IS NOT NULL AND
        x2 IS NOT NULL
),
tradeSignal AS (
    /* generate buy/sell signals */
    SELECT
        DateAdd(hour, -7, System.Timestamp) AS time,
        symbol,
        midPrice,
        efpc,
        CASE WHEN (efpc > 0.02) THEN 10 ELSE (CASE WHEN (efpc < -0.02) THEN -10 ELSE 0 END) END AS trade,
        DATETIMEFROMPARTS(DATEPART(year, System.Timestamp), DATEPART(month, System.Timestamp), DATEPART(day, System.Timestamp), 0, 0, 0, 0) as date
    FROM prediction
),

```

Trading simulation

After we have the trading signals, we want to test how effective the trading strategy is, without trading for real.

We achieve this test by using a UDA, with a hopping window, hopping every one minute. The additional grouping on date and the having clause allow the window only accounts for events that belong to the same day. For a hopping window across two days, the **GROUP BY** date separates the grouping into previous day and current day. The **HAVING** clause filters out the windows that are ending on the current day but grouping on the previous day.

```

simulation AS
(
    /* perform trade simulation for the past 7 hours to cover an entire trading day, and generate output every minute */
    SELECT
        DateAdd(hour, -7, System.Timestamp) AS time,
        symbol,
        date,
        uda.TradeSimulation(tradeSignal) AS s
    FROM tradeSignal
    GROUP BY HoppingWindow(minute, 420, 1), symbol, date
    Having DateDiff(day, date, time) < 1 AND DATEPART(hour, time) < 13
)

```

The JavaScript UDA initializes all accumulators in the `init` function, computes the state transition with every event added to the window, and returns the simulation results at the end of the window. The general trading process is to:

- Buy stock when a buy signal is received and there is no stock holding.
- Sell stock when a sell signal is received and there is stock holding.
- Short if there is no stock holding.

If there's a short position, and a buy signal is received, we buy to cover. We never hold or short 10 shares of a stock in this simulation. The transaction cost is a flat \$8.

```
function main() {
  var TRADE_COST = 8.0;
  var SHARES = 10;
  this.init = function () {
    this.own = false;
    this.pos = 0;
    this.pnl = 0.0;
    this.tradeCosts = 0.0;
    this.buyPrice = 0.0;
    this.sellPrice = 0.0;
    this.buySize = 0;
    this.sellSize = 0;
    this.buyTotal = 0.0;
    this.sellTotal = 0.0;
  }
  this.accumulate = function (tradeSignal, timestamp) {
    if(!this.own && tradeSignal.trade == 10) {
      // Buy to open
      this.own = true;
      this.pos = 1;
      this.buyPrice = tradeSignal.midprice;
      this.tradeCosts += TRADE_COST;
      this.buySize += SHARES;
      this.buyTotal += SHARES * tradeSignal.midprice;
    } else if(!this.own && tradeSignal.trade == -10) {
      // Sell to open
      this.own = true;
      this.pos = -1;
      this.sellPrice = tradeSignal.midprice;
      this.tradeCosts += TRADE_COST;
      this.sellSize += SHARES;
      this.sellTotal += SHARES * tradeSignal.midprice;
    } else if(this.own && this.pos == 1 && tradeSignal.trade == -10) {
      // Sell to close
      this.own = false;
      this.pos = 0;
      this.sellPrice = tradeSignal.midprice;
      this.tradeCosts += TRADE_COST;
      this.pnl += (this.sellPrice - this.buyPrice)*SHARES - 2*TRADE_COST;
      this.sellSize += SHARES;
      this.sellTotal += SHARES * tradeSignal.midprice;
      // Sell to open
      this.own = true;
      this.pos = -1;
      this.sellPrice = tradeSignal.midprice;
      this.tradeCosts += TRADE_COST;
      this.sellSize += SHARES;
      this.sellTotal += SHARES * tradeSignal.midprice;
    } else if(this.own && this.pos == -1 && tradeSignal.trade == 10) {
      // Buy to close
      this.own = false;
      this.pos = 0;
      this.buyPrice = tradeSignal.midprice;
      this.tradeCosts += TRADE_COST;
      this.pnl += (this.sellPrice - this.buyPrice)*SHARES - 2*TRADE_COST;
      this.buySize += SHARES;
      this.buyTotal += SHARES * tradeSignal.midprice;
      // Buy to open
      this.own = true;
      this.pos = 1;
      this.buyPrice = tradeSignal.midprice;
      this.tradeCosts += TRADE_COST;
      this.buySize += SHARES;
      this.buyTotal += SHARES * tradeSignal.midprice;
    }
  }
  this.computeResult = function () {
    var result = {
      "pnl": this.pnl,
```

```

        "buySize": this.buySize,
        "sellSize": this.sellSize,
        "buyTotal": this.buyTotal,
        "sellTotal": this.sellTotal,
        "tradeCost": this.tradeCost
    };
    return result;
}
}

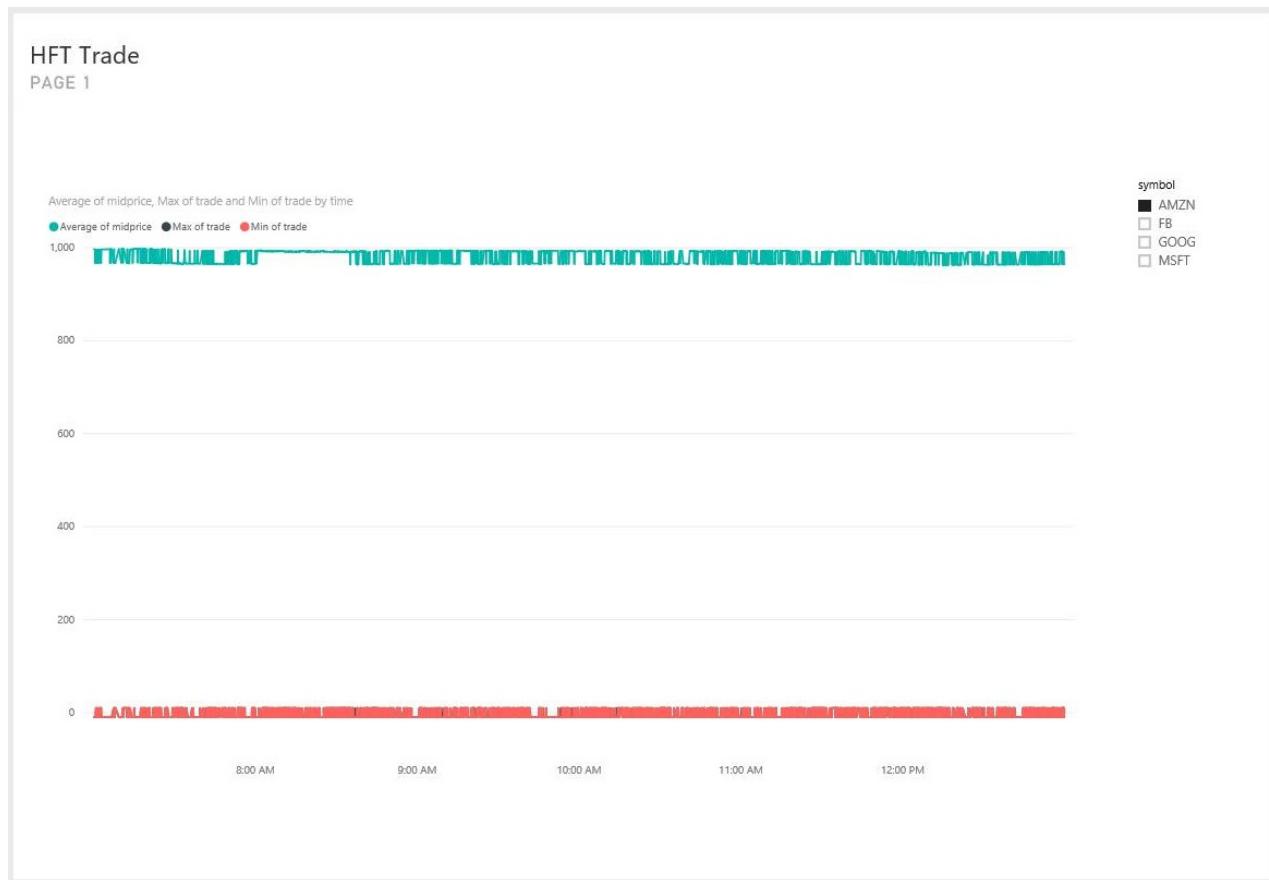
```

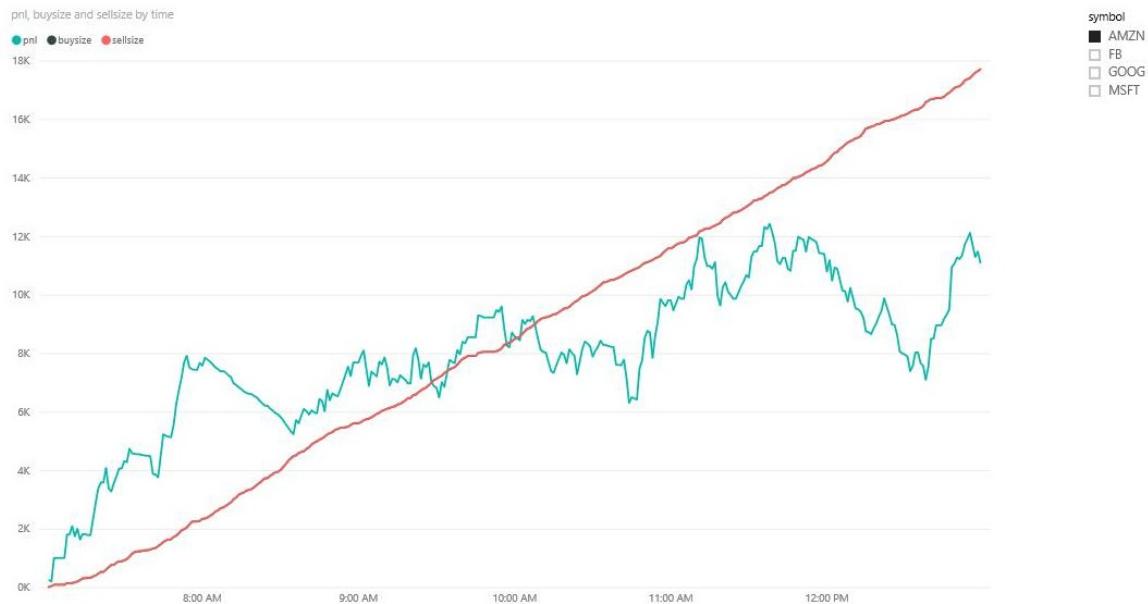
Finally, we output to the Power BI dashboard for visualization.

```

SELECT * INTO tradeSignalDashboard FROM tradeSignal /* output tradeSignal to PBI */
SELECT
    symbol,
    time,
    date,
    TRY_CAST(s.pnl as float) AS pnl,
    TRY_CAST(s.buySize as bigint) AS buySize,
    TRY_CAST(s.sellSize as bigint) AS sellSize,
    TRY_CAST(s.buyTotal as float) AS buyTotal,
    TRY_CAST(s.sellTotal as float) AS sellTotal
    INTO pnlDashboard
FROM simulation /* output trade simulation to PBI */

```





Summary

We can implement a realistic high-frequency trading model with a moderately complex query in Azure Stream Analytics. We have to simplify the model from five input variables to two, because of the lack of a built-in linear regression function. But for a determined user, algorithms with higher dimensions and sophistication can possibly be implemented as JavaScript UDA as well.

It's worth noting that most of the query, other than the JavaScript UDA, can be tested and debugged in Visual Studio through [Azure Stream Analytics tools for Visual Studio](#). After the initial query was written, the author spent less than 30 minutes testing and debugging the query in Visual Studio.

Currently, the UDA cannot be debugged in Visual Studio. We are working on enabling that with the ability to step through JavaScript code. In addition, note that the fields reaching the UDA have lowercase names. This was not an obvious behavior during query testing. But with Azure Stream Analytics compatibility level 1.1, we preserve the field name casing so the behavior is more natural.

I hope this article serves as an inspiration for all Azure Stream Analytics users, who can use our service to perform advanced analytics in near real time, continuously. Let us know any feedback you have to make it easier to implement queries for advanced analytics scenarios.

Get started with Azure Stream Analytics to process data from IoT devices

6/13/2019 • 5 minutes to read • [Edit Online](#)

In this tutorial, you learn how to create stream-processing logic to gather data from Internet of Things (IoT) devices. We will use a real-world, Internet of Things (IoT) use case to demonstrate how to build your solution quickly and economically.

Prerequisites

- [Azure subscription](#)
- Sample query and data files downloadable from [GitHub](#)

Scenario

Contoso, which is a company in the industrial automation space, has completely automated its manufacturing process. The machinery in this plant has sensors that are capable of emitting streams of data in real time. In this scenario, a production floor manager wants to have real-time insights from the sensor data to look for patterns and take actions on them. We will use the Stream Analytics Query Language (SAQL) over the sensor data to find interesting patterns from the incoming stream of data.

Here data is being generated from a Texas Instruments sensor tag device. The payload of the data is in JSON format and looks like the following:

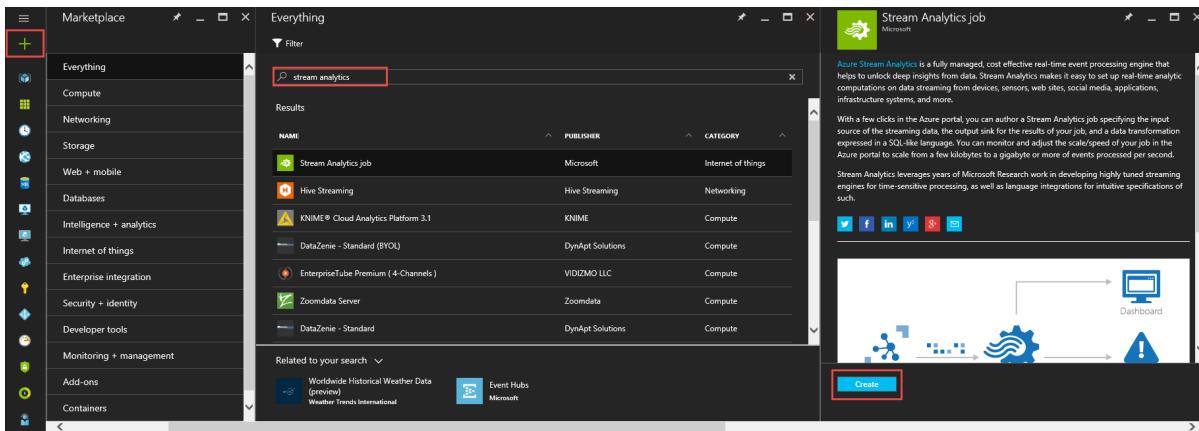
```
{  
    "time": "2016-01-26T20:47:53.0000000",  
    "dspl": "sensorE",  
    "temp": 123,  
    "hmdt": 34  
}
```

In a real-world scenario, you could have hundreds of these sensors generating events as a stream. Ideally, a gateway device would run code to push these events to [Azure Event Hubs](#) or [Azure IoT Hubs](#). Your Stream Analytics job would ingest these events from Event Hubs and run real-time analytics queries against the streams. Then, you could send the results to one of the [supported outputs](#).

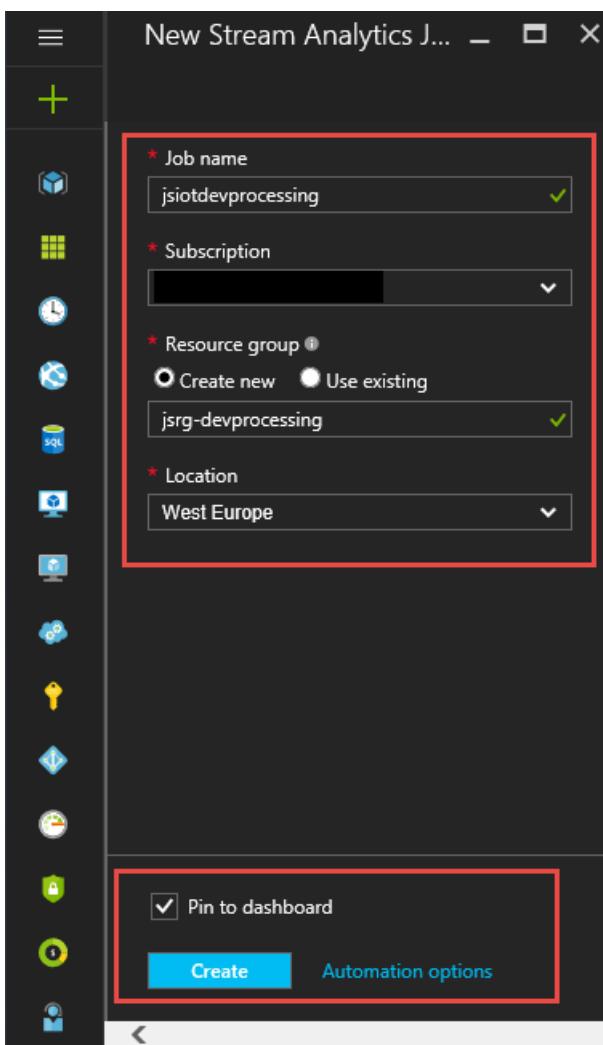
For ease of use, this getting started guide provides a sample data file, which was captured from real sensor tag devices. You can run queries on the sample data and see results. In subsequent tutorials, you will learn how to connect your job to inputs and outputs and deploy them to the Azure service.

Create a Stream Analytics job

1. In the [Azure portal](#), click the plus sign and then type **STREAM ANALYTICS** in the text window to the right. Then select **Stream Analytics job** in the results list.



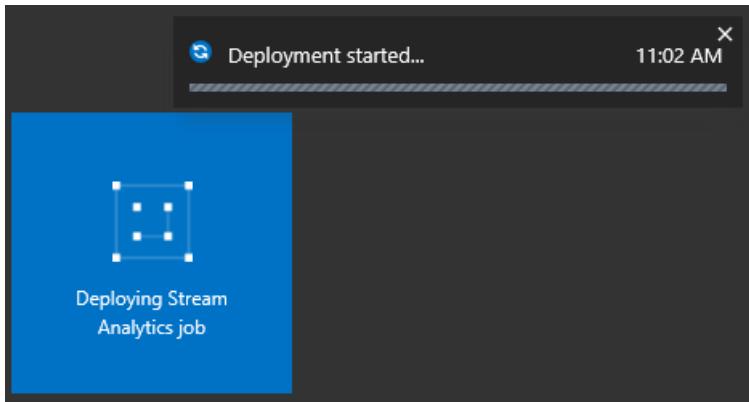
2. Enter a unique job name and verify the subscription is the correct one for your job. Then either create a new resource group or select an existing one on your subscription.
3. Then select a location for your job. For speed of processing and reduction of cost in data transfer selecting the same location as the resource group and intended storage account is recommended.



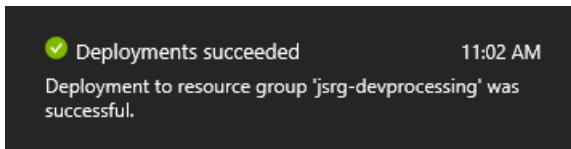
NOTE

You should create this storage account only once per region. This storage will be shared across all Stream Analytics jobs that are created in that region.

4. Check the box to place your job on your dashboard and then click **CREATE**.

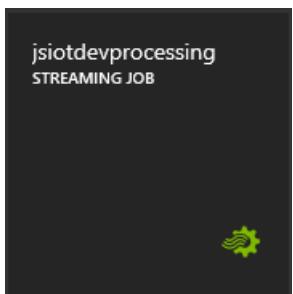


5. You should see a 'Deployment started...' displayed in the top right of your browser window. Soon it will change to a completed window as shown below.



Create an Azure Stream Analytics query

After your job is created it's time to open it and build a query. You can easily access your job by clicking the tile for it.



In the **Job Topology** pane click the **QUERY** box to go to the Query Editor. The **QUERY** editor allows you to enter a T-SQL query that performs the transformation over the incoming event data.

The screenshot shows the Azure Stream Analytics job 'jsiotdevprocessing' in the 'Created' state. The 'Job Topology' section indicates 0 inputs, 1 query (highlighted with a red box), and 0 outputs.

Query: Archive your raw data

The simplest form of query is a pass-through query that archives all input data to its designated output. Download the sample data file from [GitHub](#) to a location on your computer.

1. Paste the query from the PassThrough.txt file.

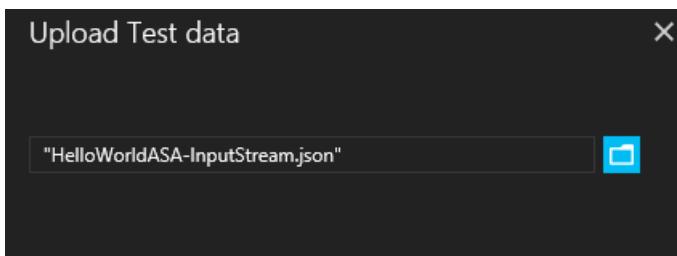
The screenshot shows the Stream Analytics Query editor with the following details:

- Inputs:** 1 item, 'yourinputalias'
- Outputs:** 1 item, 'youroutputalias'
- Code Editor:** Displays a sample query:

```
1 SELECT
2 *
3 INTO
4 [YourOutputAlias]
5 FROM
6 [YourInputAlias]
```

2. Click the three dots next to your input and select **Upload sample data from file** box.

3. A pane opens on the right as a result, in it select the HelloWorldASA-InputStream.json data file from your downloaded location and click **OK** at the bottom of the pane.



4. Then click the **Test** gear in the top left area of the window and process your test query against the sample dataset. A results window will open below your query as the processing is complete.

Results

output

Generated the Following:

- output with 1860 rows.

Download results

| TIME | DSPL | TEMP | HMDT |
|-----------------------------|---------|------|------|
| 2016-01-26T20:47:53.0000000 | sensorE | 123 | 34 |
| 2016-01-26T20:47:54.0000000 | sensorC | 77 | 49 |
| 2016-01-26T20:47:55.0000000 | sensorA | 128 | 64 |
| 2016-01-26T20:47:56.0000000 | sensorE | 73 | 45 |
| 2016-01-26T20:47:57.0000000 | sensorA | 134 | 68 |
| 2016-01-26T20:47:58.0000000 | sensorE | 140 | 62 |
| 2016-01-26T20:47:59.0000000 | sensorD | 100 | 59 |
| 2016-01-26T20:48:00.0000000 | sensorE | 125 | 41 |
| 2016-01-26T20:48:01.0000000 | sensorD | 99 | 35 |

Query: Filter the data based on a condition

Let's try to filter the results based on a condition. We would like to show results for only those events that come from "sensorA." The query is in the Filtering.txt file.

jsiotdevprocessing

Query

Save Discard Test

Inputs (1)

inputstream ...

Outputs (1)

output

Need help with your query? Check out some of the most common Stream Analytics query patterns [here](#).

```
1 SELECT
2     time,
3     dspl AS SensorName,
4     temp AS Temperature,
5     hmdt AS Humidity
6 INTO
7     output
8 FROM
9     InputStream
10 WHERE dspl='sensorA'
11 |
```

Your query could be put in logs that are in a potentially different geography.
Missing some language constructs? [Let us know!](#) (Powered by [UserVoice](#) - [Privacy Policy](#))

Note that the case-sensitive query compares a string value. Click the **Test** gear again to execute the query. The query should return 389 rows out of 1860 events.

Results

output

Generated the Following:

- output with 389 rows.

[Download results](#)

Query: Alert to trigger a business workflow

Let's make our query more detailed. For every type of sensor, we want to monitor average temperature per 30-second window and display results only if the average temperature is above 100 degrees. We will write the following query and then click **Test** to see the results. The query is in the ThresholdAlerting.txt file.

Need help with your query? Check out some of the most common Stream Analytics query patterns [here](#).

```
1 SELECT
2     System.Timestamp AS OutputTime,
3     dspl AS SensorName,
4     Avg(temp) AS AvgTemperature
5 INTO
6     output
7 FROM
8     InputStream TIMESTAMP BY time
9 GROUP BY TumblingWindow(second,30),dspl
10 HAVING Avg(temp)>100
```

Your query could be put in logs that are in a potentially different geography.
Missing some language constructs? [Let us know!](#) (Powered by [UserVoice](#) - [Privacy Policy](#))

Results

output

Generated the Following:

- output with 245 rows.

You should now see results that contain only 245 rows and names of sensors where the average temperate is greater than 100. This query groups the stream of events by **dspl**, which is the sensor name, over a **Tumbling Window** of 30 seconds. Temporal queries must state how we want time to progress. By using the **TIMESTAMP**

BY clause, we have specified the **OUTPUTTIME** column to associate times with all temporal calculations. For detailed information, read the MSDN articles about [Time Management](#) and [Windowing functions](#).

Query: Detect absence of events

How can we write a query to find a lack of input events? Let's find the last time that a sensor sent data and then did not send events for the next 5 seconds. The query is in the AbsenceOfEvent.txt file.

Need help with your query? Check out some of the most common Stream Analytics query patterns [here](#).

```
1 SELECT
2     t1.time,
3     t1.dspl AS SensorName
4 INTO
5     output
6 FROM
7     InputStream t1 TIMESTAMP BY time
8 LEFT OUTER JOIN InputStream t2 TIMESTAMP BY time
9 ON
10    t1.dspl=t2.dspl AND
11    DATEDIFF(second,t1,t2) BETWEEN 1 and 5
12 WHERE t2.dspl IS NULL
```

Your query could be put in logs that are in a potentially different geography.
Missing some language constructs? [Let us know!](#) (Powered by UserVoice - Privacy Policy)

Results

output

Generated the Following:

- output with 582 rows.

Here we use a **LEFT OUTER** join to the same data stream (self-join). For an **INNER** join, a result is returned only when a match is found. For a **LEFT OUTER** join, if an event from the left side of the join is unmatched, a row that has **NULL** for all the columns of the right side is returned. This technique is very useful to find an absence of events. See our MSDN documentation for more information about [JOIN](#).

Conclusion

The purpose of this tutorial is to demonstrate how to write different Stream Analytics Query Language queries and see results in the browser. However, this is just getting started. You can do so much more with Stream Analytics. Stream Analytics supports a variety of inputs and outputs and can even use functions in Azure Machine Learning to make it a robust tool for analyzing data streams. You can start to explore more about Stream Analytics by using our [learning map](#). For more information about how to write queries, read the article about [common query patterns](#).

Process configurable threshold-based rules in Azure Stream Analytics

2/22/2019 • 6 minutes to read • [Edit Online](#)

This article describes how to use reference data to achieve an alerting solution that uses configurable threshold-based rules in Azure Stream Analytics.

Scenario: Alerting based on adjustable rule thresholds

You may need to produce an alert as output when incoming streamed events have reached a certain value, or when an aggregated value based on the incoming streamed events exceeds a certain threshold. It's simple to set up a Stream Analytics query that compares value to a static threshold that is fixed and predetermined. A fixed threshold can be hard-coded into the streaming query syntax using simple numerical comparisons (greater than, less than, and equality).

In some cases, the threshold values need to be more easily configurable without editing the query syntax each time that a threshold value changes. In other cases, you may need numerous devices or users processed by the same query with each of them having a different threshold values on each kind of device.

This pattern can be used to dynamically configure thresholds, selectively choose which kind of device the threshold applies by filtering the input data, and selectively choose which fields to include in the output.

Recommended design pattern

Use a reference data input to a Stream Analytics job as a lookup of the alert thresholds:

- Store the threshold values in the reference data, one value per key.
- Join the streaming data input events to the reference data on the key column.
- Use the keyed value from the reference data as the threshold value.

Example data and query

In the example, alerts are generated when the aggregate of data streaming in from devices in a minute-long window matches the stipulated values in the rule supplied as reference data.

In the query, for each deviceId, and each metricName under the deviceId, you can configure from 0 to 5 dimensions to GROUP BY. Only the events having the corresponding filter values are grouped. Once grouped, windowed aggregates of Min, Max, Avg, are calculated over a 60-second tumbling window. Filters on the aggregated values are then calculated as per the configured threshold in the reference, to generate the alert output event.

As an example, assume there is a Stream Analytics job that has a reference data input named **rules**, and streaming data input named **metrics**.

Reference data

This example reference data shows how a threshold-based rule could be represented. A JSON file holds the reference data and is saved into Azure blob storage, and that blob storage container is used as a reference data input named **rules**. You could overwrite this JSON file and replace the rule configuration as time goes on, without stopping or starting the streaming job.

- The example rule is used to represent an adjustable alert when CPU exceeds (average is greater than or equal to) the value `90` percent. The `value` field is configurable as needed.
- Notice the rule has an **operator** field, which is dynamically interpreted in the query syntax later on `AVGGREATEROEQUAL`.
- The rule filters the data on a certain dimension key `2` with value `c1`. Other fields are empty string, indicating not to filter the input stream by those event fields. You could set up additional CPU rules to filter other matching fields as needed.
- Not all columns are to be included in the output alert event. In this case, `includedDim` key number `2` is turned on `TRUE` to represent that field number 2 of event data in the stream will be included in the qualifying output events. The other fields are not included in the alert output, but the field list can be adjusted.

```
{
  "ruleId": 1234,
  "deviceId" : "978648",
  "metricName": "CPU",
  "alertName": "hot node AVG CPU over 90",
  "operator" : "AVGGREATEROEQUAL",
  "value": 90,
  "includeDim": {
    "0": "FALSE",
    "1": "FALSE",
    "2": "TRUE",
    "3": "FALSE",
    "4": "FALSE"
  },
  "filter": {
    "0": "",
    "1": "",
    "2": "C1",
    "3": "",
    "4": ""
  }
}
```

Example streaming query

This example Stream Analytics query joins the **rules** reference data from the example above, to an input stream of data named **metrics**.

```

WITH transformedInput AS
(
    SELECT
        dim0 = CASE rules.includeDim.[0] WHEN 'TRUE' THEN metrics.custom.dimensions.[0].value ELSE NULL END,
        dim1 = CASE rules.includeDim.[1] WHEN 'TRUE' THEN metrics.custom.dimensions.[1].value ELSE NULL END,
        dim2 = CASE rules.includeDim.[2] WHEN 'TRUE' THEN metrics.custom.dimensions.[2].value ELSE NULL END,
        dim3 = CASE rules.includeDim.[3] WHEN 'TRUE' THEN metrics.custom.dimensions.[3].value ELSE NULL END,
        dim4 = CASE rules.includeDim.[4] WHEN 'TRUE' THEN metrics.custom.dimensions.[4].value ELSE NULL END,
        metric = metrics.metric.value,
        metricName = metrics.metric.name,
        deviceId = rules.deviceId,
        ruleId = rules.ruleId,
        alertName = rules.alertName,
        ruleOperator = rules.operator,
        ruleValue = rules.value
    FROM
        metrics
        timestamp by eventTime
    JOIN
        rules
    ON metrics.deviceId = rules.deviceId AND metrics.metric.name = rules.metricName
    WHERE
        (rules.filter.[0] = '' OR metrics.custom.filters.[0].value = rules.filter.[0]) AND
        (rules.filter.[1] = '' OR metrics.custom.filters.[1].value = rules.filter.[1]) AND
        (rules.filter.[2] = '' OR metrics.custom.filters.[2].value = rules.filter.[2]) AND
        (rules.filter.[3] = '' OR metrics.custom.filters.[3].value = rules.filter.[3]) AND
        (rules.filter.[4] = '' OR metrics.custom.filters.[4].value = rules.filter.[4])
)
SELECT
    System.Timestamp as time,
    transformedInput.deviceId as deviceId,
    transformedInput.ruleId as ruleId,
    transformedInput.metricName as metric,
    transformedInput.alertName as alert,
    AVG(metric) as avg,
    MIN(metric) as min,
    MAX(metric) as max,
    dim0, dim1, dim2, dim3, dim4
FROM
    transformedInput
GROUP BY
    transformedInput.deviceId,
    transformedInput.ruleId,
    transformedInput.metricName,
    transformedInput.alertName,
    dim0, dim1, dim2, dim3, dim4,
    ruleOperator,
    ruleValue,
    TumblingWindow(second, 60)
HAVING
(
    (ruleOperator = 'AVGGREATEROEQUAL' AND avg(metric) >= ruleValue) OR
    (ruleOperator = 'AVGEQUALORLESS' AND avg(metric) <= ruleValue)
)

```

Example streaming input event data

This example JSON data represents the **metrics** input data that is used in the above streaming query.

- Three example events are listed within the 1-minute timespan, value `T14:50`.
- All three have the same `deviceId` value `978648`.
- The CPU metric values vary in each event, `98`, `95`, `80` respectively. Only the first two example events exceed the CPU alert rule established in the rule.

- The includeDim field in the alert rule was key number 2. The corresponding key 2 field in the example events is named `NodeName`. The three example events have values `N024`, `N024`, and `N014` respectively. In the output, you see only the node `N024` as that is the only data that matches the alert criteria for high CPU. `N014` does not meet the high CPU threshold.
- The alert rule is configured with a `filter` only on key number 2, which corresponds to the `cluster` field in the sample events. The three example events all have value `C1` and match the filter criteria.

```
{
  "eventTime": "2018-04-30T14:50:23.1324132Z",
  "deviceId": "978648",
  "custom": {
    "dimensions": {
      "0": {
        "name": "NodeType",
        "value": "N1"
      },
      "1": {
        "name": "Cluster",
        "value": "C1"
      },
      "2": {
        "name": "NodeName",
        "value": "N024"
      }
    },
    "filters": {
      "0": {
        "0": {
          "name": "application",
          "value": "A1"
        },
        "1": {
          "name": "deviceType",
          "value": "T1"
        },
        "2": {
          "name": "cluster",
          "value": "C1"
        },
        "3": {
          "name": "nodeType",
          "value": "N1"
        }
      }
    },
    "metric": {
      "name": "CPU",
      "value": 98,
      "count": 1.0,
      "min": 98,
      "max": 98,
      "stdDev": 0.0
    }
  }
}

{
  "eventTime": "2018-04-30T14:50:24.1324138Z",
  "deviceId": "978648",
  "custom": {
    "dimensions": {
      "0": {
        "name": "NodeType",
        "value": "N2"
      },
      "1": {
        "name": "Cluster",
        "value": "C1"
      }
    }
  }
}
```

```
        "2": {
            "name": "NodeName",
            "value": "N024"
        },
        "filters": {
            "0": {
                "name": "application",
                "value": "A1"
            },
            "1": {
                "name": "deviceType",
                "value": "T1"
            },
            "2": {
                "name": "cluster",
                "value": "C1"
            },
            "3": {
                "name": "nodeType",
                "value": "N2"
            }
        }
    },
    "metric": {
        "name": "CPU",
        "value": 95,
        "count": 1,
        "min": 95,
        "max": 95,
        "stdDev": 0
    }
}
{
    "eventTime": "2018-04-30T14:50:37.1324130Z",
    "deviceId": "978648",
    "custom": {
        "dimensions": {
            "0": {
                "name": "NodeType",
                "value": "N3"
            },
            "1": {
                "name": "Cluster",
                "value": "C1"
            },
            "2": {
                "name": "NodeName",
                "value": "N014"
            }
        },
        "filters": {
            "0": {
                "name": "application",
                "value": "A1"
            },
            "1": {
                "name": "deviceType",
                "value": "T1"
            },
            "2": {
                "name": "cluster",
                "value": "C1"
            },
            "3": {
                "name": "nodeType",
                "value": "N3"
            }
        }
    }
}
```

```
        },
        "metric": {
            "name": "CPU",
            "value": 80,
            "count": 1,
            "min": 80,
            "max": 80,
            "stdDev": 0
        }
    }
```

Example output

This example output JSON data shows a single alert event was produced based on the CPU threshold rule defined in the reference data. The output event contains the name of the alert as well as the aggregated (average, min, max) of the fields considered. The output event data includes field key number 2 `NodeName` value `N024` due to the rule configuration. (The JSON was altered to show line breaks for readability.)

```
{"time":"2018-05-01T02:03:00.000000Z","deviceid":"978648","ruleid":1234,"metric":"CPU",
"alert":"hot node AVG CPU over 90","avg":96.5,"min":95.0,"max":98.0,
"dim0":null,"dim1":null,"dim2":"N024","dim3":null,"dim4":null}
```

Process Apache Kafka for Event Hubs events using Stream analytics

3/6/2019 • 5 minutes to read • [Edit Online](#)

This article shows how to stream data into Kafka-enabled Event Hubs and process it with Azure Stream Analytics. It walks you through the following steps:

1. Create a Kafka enabled Event Hubs namespace.
2. Create a Kafka client that sends messages to the event hub.
3. Create a Stream Analytics job that copies data from the event hub into an Azure blob storage.

You do not need to change your protocol clients or run your own clusters when you use the Kafka endpoint exposed by an event hub. Azure Event Hubs supports [Apache Kafka version 1.0](#). and above.

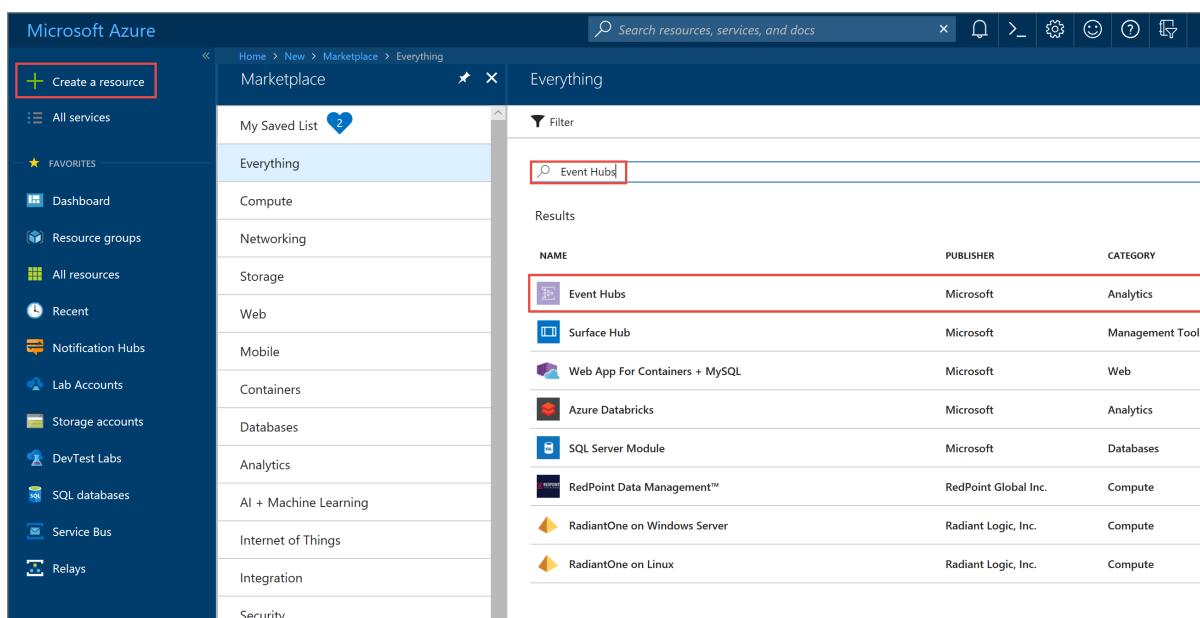
Prerequisites

To complete this quickstart, make sure you have the following prerequisites:

- An Azure subscription. If you do not have one, create a [free account](#) before you begin.
- [Java Development Kit \(JDK\) 1.7+](#).
- [Download](#) and [install](#) a Maven binary archive.
- [Git](#)
- An **Azure Storage account**. If you don't have one, [create one](#) before proceeding further. The Stream Analytics job in this walkthrough stores the output data in an Azure blob storage.

Create a Kafka enabled Event Hubs namespace

1. Sign in to the [Azure portal](#), and click **Create a resource** at the top left of the screen.
2. Search for **Event Hubs** and select the options shown here:

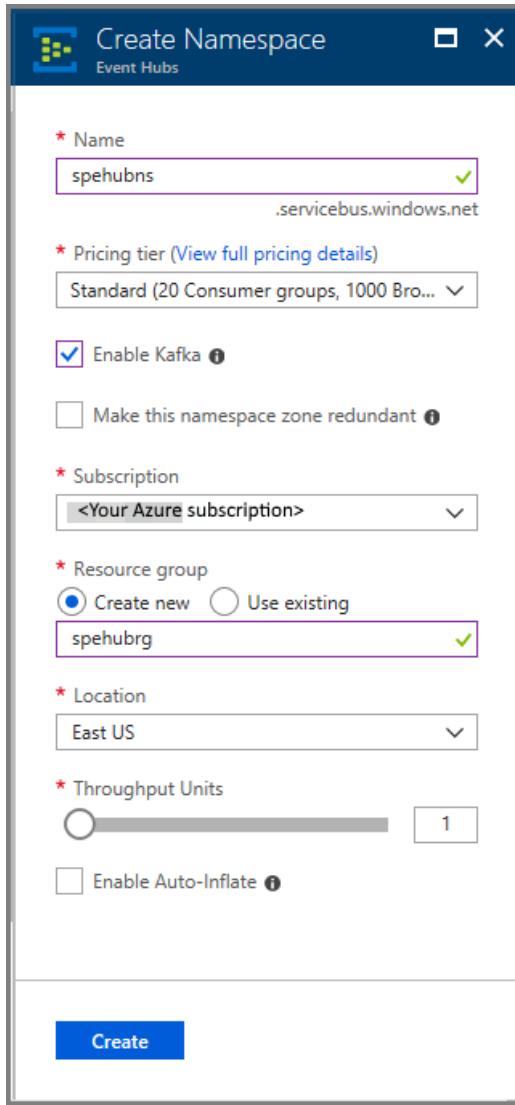


The screenshot shows the Microsoft Azure Marketplace search interface. A red box highlights the 'Create a resource' button in the top-left corner of the sidebar. The search bar at the top contains the text 'Event Hubs'. The search results table has columns for NAME, PUBLISHER, and CATEGORY. The first result, 'Event Hubs' by Microsoft under the Analytics category, is highlighted with a red border. Other results include Surface Hub, Web App For Containers + MySQL, Azure Databricks, SQL Server Module, RedPoint Data Management™, RadiantOne on Windows Server, and RadiantOne on Linux.

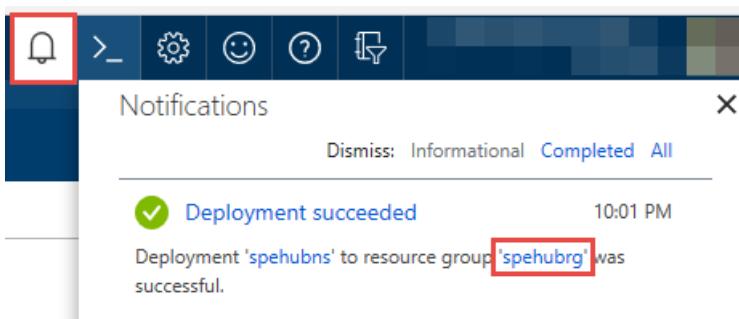
| NAME | PUBLISHER | CATEGORY |
|--------------------------------|----------------------|------------------|
| Event Hubs | Microsoft | Analytics |
| Surface Hub | Microsoft | Management Tools |
| Web App For Containers + MySQL | Microsoft | Web |
| Azure Databricks | Microsoft | Analytics |
| SQL Server Module | Microsoft | Databases |
| RedPoint Data Management™ | RedPoint Global Inc. | Compute |
| RadiantOne on Windows Server | Radiant Logic, Inc. | Compute |
| RadiantOne on Linux | Radiant Logic, Inc. | Compute |

3. On the **Event Hubs** page, select **Create**.
4. On the **Create Namespace** page, do the following actions:

- a. Provide a unique **name** for the namespace.
- b. Select a **pricing tier**.
- c. Select **Enable Kafka**. This step is an **important** step.
- d. Select your **subscription** in which you want the event hub namespace to be created.
- e. Create a new **resource group** or select an existing resource group.
- f. Select a **location**.
- g. Click **Create**.



5. In the **notification message**, select the **resource group name**.



6. Select the **event hub namespace** in the resource group.
7. Once the namespace is created, select **Shared access policies** under **SETTINGS**.

8. You can choose the default **RootManageSharedAccessKey**, or add a new policy. Click the policy name and copy the **connection string**. You use the connection string to configure the Kafka client.

You can now stream events from your applications that use the Kafka protocol into Event Hubs.

Send messages with Kafka in Event Hubs

1. Clone the [Azure Event Hubs for Kafka repository](#) to your machine.
2. Navigate to the folder: `azure-event-hubs-for-kafka/quickstart/java/producer`.
3. Update the configuration details for the producer in `src/main/resources/producer.config`. Specify the **name** and **connection string** for the **event hub namespace**.

```
bootstrap.servers={EVENT HUB NAMESPACE}.servicebus.windows.net:9093
security.protocol=SASL_SSL
sasl.mechanism=PLAIN
sasl.jaas.config=org.apache.kafka.common.security.plain.PlainLoginModule required
username="$ConnectionString" password="{CONNECTION STRING for EVENT HUB NAMESPACE}";
```

4. Navigate to `azure-event-hubs-for-kafka/quickstart/java/producer/src/main/java/com/example/app`, and open **TestDataReporter.java** file in an editor of your choice.
5. Comment out the following line of code:

```
//final ProducerRecord<Long, String> record = new ProducerRecord<Long, String>(TOPIC, time,  
"Test Data " + i);
```

6. Add the following line of code in place of the commented code:

```
final ProducerRecord<Long, String> record = new ProducerRecord<Long, String>(TOPIC, time, {"  
\\"eventData\\": \"Test Data " + i + "\"});
```

This code sends the event data in **JSON** format. When you configure input for a Stream Analytics job, you specify JSON as the format for the input data.

7. **Run the producer** and stream into Kafka-enabled Event Hubs. On a Windows machine, when using a **Node.js command prompt**, switch to the `azure-event-hubs-for-kafka/quickstart/java/producer` folder before running these commands.

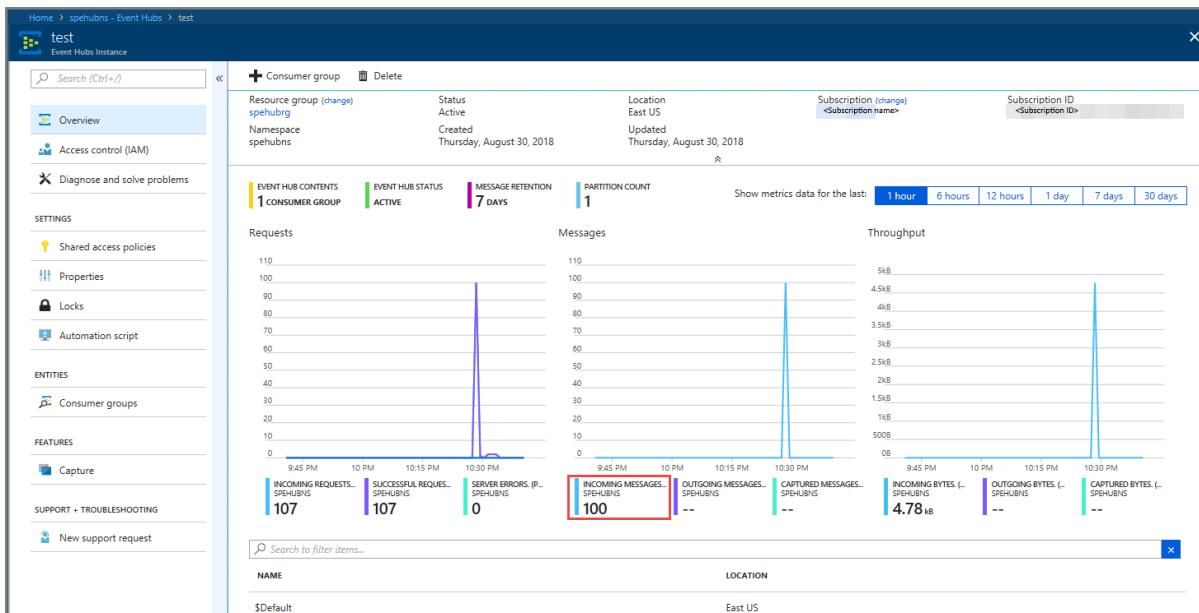
```
mvn clean package  
mvn exec:java -Dexec.mainClass="TestProducer"
```

Verify that event hub receives the data

1. Select **Event Hubs** under **ENTITIES**. Confirm that you see an event hub named **test**.

| NAME | STATUS | MESSAGE RETENTION | PARTITION COUNT |
|------|--------|-------------------|-----------------|
| test | Active | 7 | 1 |

2. Confirm that you see messages coming in to the event hub.



Process event data using a Stream Analytics job

In this section, you create an Azure Stream Analytics job. The Kafka client sends events to the event hub. You create a Stream Analytics job that takes event data as input and outputs it to an Azure blob storage. If you don't have an **Azure Storage account**, [create one](#).

The query in the Stream Analytics job passes through the data without performing any analytics. You can create a query that transforms the input data to produce output data in a different format or with gained insights.

Create a Stream Analytics job

1. Select + **Create a resource** in the [Azure portal](#).
2. Select **Analytics** in the **Azure Marketplace** menu, and select **Stream Analytics job**.
3. On the **New Stream Analytics** page, do the following actions:
 - a. Enter a **name** for the job.
 - b. Select your **subscription**.
 - c. Select **Create new** for the **resource group** and enter the name. You can also **use an existing** resource group.
 - d. Select a **location** for the job.
 - e. Select **Create** to create the job.

Home > New > New Stream Analytics job

New Stream Analytics job

* Job name
spstreamanalyticsjob ✓

* Subscription
<Your Azure subscription>

* Resource group
 Create new Use existing
spsajobrg ✓

* Location
East US

Hosting environment Cloud Edge

Streaming units (1 to 120) 3

Create [Automation options](#)

Configure job input

1. In the notification message, select **Go to resource** to see the **Stream Analytics job** page.
2. Select **Inputs** in the **JOB TOPOLOGY** section on the left menu.
3. Select **Add stream input**, and then select **Event Hub**.

Home > spstreamanalyticsjob - Inputs

spstreamanalyticsjob - Inputs

Stream Analytics job

Search (Ctrl+)

+ Add stream input **+ Add reference input**

| SOURCE TYPE | SOURCE |
|--------------|--------|
| Event Hub | |
| IoT Hub | |
| Blob storage | |

Inputs Inputs

SETTINGS

LOCK

JOB TOPOLOGY

- Inputs** Inputs
- Functions
- Query
- Outputs

4. On the **Event Hub input** configuration page, do the following actions:
 - a. Specify an **alias** for the input.
 - b. Select your **Azure subscription**.

c. Select the **event hub namespace** you created earlier.

d. Select **test** for the **event hub**.

e. Select **Save**.

The screenshot shows the 'Event Hub' configuration dialog box. It has a title bar 'Event Hub' and a sub-header 'New input'. The form contains the following fields:

- Input alias:** myeventhub (highlighted with a purple border)
- Provide Event Hub settings manually:** (radio button)
- Select Event Hub from your subscriptions:** (radio button, selected)
- Subscription:** <Your Azure subscription>
- Event Hub namespace:** spehubns
- Event Hub name:** Create new (radio button) test (selected)
- Event Hub policy name:** RootManageSharedAccessKey
- Event Hub policy key:** (redacted)
- Event Hub consumer group:** (redacted)
- Event serialization format:** JSON
- Encoding:** UTF-8
- Event compression type:** None

At the bottom is a blue 'Save' button.

Configure job output

1. Select **Outputs** in the **JOB TOPOLOGY** section on the menu.
2. Select **+ Add** on the toolbar, and select **Blob storage**
3. On the Blob storage output settings page, do the following actions:
 - a. Specify an **alias** for the output.
 - b. Select your Azure **subscription**.
 - c. Select your **Azure Storage account**.
 - d. Enter a **name for the container** that stores the output data from the Stream Analytics query.
 - e. Select **Save**.

Define a query

After you have a Stream Analytics job setup to read an incoming data stream, the next step is to create a transformation that analyzes data in real time. You define the transformation query by using [Stream Analytics Query Language](#). In this walkthrough, you define a query that passes through the data without performing any transformation.

1. Select **Query**.
2. In the query window, replace `[YourOutputAlias]` with the output alias you created earlier.
3. Replace `[YourInputAlias]` with the input alias you created earlier.
4. Select **Save** on the toolbar.

```

1 SELECT
2 *
3 INTO
4 myblobstorage
5 FROM
6 myeventhub
    
```

Run the Stream Analytics job

1. Select **Overview** on the left menu.
2. Select **Start**.

Home > spstreamanalyticsjob

spstreamanalyticsjob

Stream Analytics Job

Search (Ctrl+)

Start Stop Delete

Overview

Resource group (change) spsajobrg

Status Stopped

Location East US

Subscription (change)

Subscription ID [REDACTED]

Send feedback UserVoice

Created Wednesday, August 29, 2018 10:58:57 PM

Started Wednesday, August 29, 2018 11:22:56 PM

Last output Wednesday, August 29, 2018 11:27:24 PM

Hosting environment Cloud

Inputs

1 myeventhub

Outputs

1 myblobstorage

Query

```

1 SELECT
2 *
3 INTO
4 myblobstorage
5 FROM
6 myeventhub

```

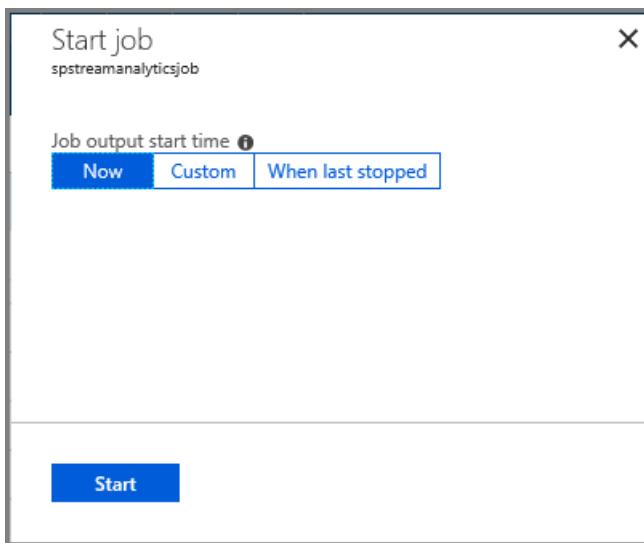
Monitoring

100
90
80
70
60
50

Resource utilization

100%
90%
80%
70%
60%
50%

3. On the **Start job** page, select **Start**.



4. Wait until the status of the job changes from **Starting** to **running**.

Home > spstreamanalyticsjob

spstreamanalyticsjob

Stream Analytics Job

Search (Ctrl+)

Start Stop Delete

Running

Resource group (change) spsajobrg

Status Running

Location East US

Send feedback UserVoice

Created Wednesday, August 29, 2018 10:58:57 PM

Started Thursday, August 30, 2018 12:43:26 PM

Test the scenario

1. Run the **Kafka producer** again to send events to the event hub.

```
mvn exec:java -Dexec.mainClass="TestProducer"
```

2. Confirm that you see **output data** is generated in the **Azure blob storage**. You see a JSON file in the container with 100 rows that look like the following sample rows:

```
{"eventData":"Test Data 0","EventProcessedUtcTime":"2018-08-30T03:27:23.1592910Z","PartitionId":0,"EventEnqueuedUtcTime":"2018-08-30T03:27:22.9220000Z"}  
{"eventData":"Test Data 1","EventProcessedUtcTime":"2018-08-30T03:27:23.3936511Z","PartitionId":0,"EventEnqueuedUtcTime":"2018-08-30T03:27:22.9220000Z"}  
{"eventData":"Test Data 2","EventProcessedUtcTime":"2018-08-30T03:27:23.3936511Z","PartitionId":0,"EventEnqueuedUtcTime":"2018-08-30T03:27:22.9220000Z"}
```

The Azure Stream Analytics job received input data from the event hub and stored it in the Azure blob storage in this scenario.

Next steps

In this article, you learned how to stream into Kafka-enabled Event Hubs without changing your protocol clients or running your own clusters. To learn more about Event Hubs and Event Hubs for Kafka, see the following topic:

- [Learn about Event Hubs](#)
- [Event Hubs for Apache Kafka](#)
- [How to create Kafka enabled Event Hubs](#)
- [Stream into Event Hubs from your Kafka applications](#)
- [Mirror a Kafka broker in a Kafka-enabled event hub](#)
- [Connect Apache Spark to a Kafka-enabled event hub](#)
- [Connect Apache Flink to a Kafka-enabled event hub](#)
- [Integrate Kafka Connect with a Kafka-enabled event hub](#)
- [Connect Akka Streams to a Kafka-enabled event hub](#)
- [Explore samples on our GitHub](#)

Geofencing and geospatial aggregation scenarios with Azure Stream Analytics

4/1/2019 • 4 minutes to read • [Edit Online](#)

With built-in geospatial functions, you can use Azure Stream Analytics to build applications for scenarios such as fleet management, ride sharing, connected cars, and asset tracking.

Geofencing

Azure Stream Analytics supports low latency real-time geofencing computations in the cloud and on the IoT Edge runtime.

Geofencing scenario

A manufacturing company needs to track assets on their buildings. They equipped every device with a GPS and want to receive notifications if a device leaves a certain area.

Reference data used in this example has the geofence information for the buildings and the devices that are allowed in each of the buildings. Remember that reference data could either be static or slow changing. Static reference data is used for this scenario. A stream of data continuously emits the device ID and its current position.

Define geofences in reference data

A geofence can be defined using a GeoJSON object. For jobs with compatibility version 1.2 and higher, geofences can also be defined using Well Known Text (WKT) as `NVARCHAR(MAX)`. WKT is an Open Geospatial Consortium (OGC) standard that is used to represent spatial data in a textual format.

The built-in geospatial functions can use defined geofences to find out if an element is in or out of a specific geofence polygon.

The following table is an example of geofence reference data that could be stored in Azure blob storage or an Azure SQL table. Every site is represented by a geospatial polygon, and every device is associated with an allowed site ID.

| SITEID | SITENAME | GEOFENCE | ALLOWEDDEVICEID |
|--------|----------|----------|-----------------|
|--------|----------|----------|-----------------|

| SITEID | SITENAME | GEOFENCE | ALLOWEDDEVICEID |
|--------|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|
| 1 | "Redmond Building 41" | "POLYGON((-122.1337357922017
47.63782998329432,-122.13373042778369
47.637634793257305,-122.13346757130023
47.637642022530954,-122.13348902897235
47.637508280806806,-122.13361777500506
47.637508280806806,-122.13361241058703
47.63732393354484,-122.13265754417773
47.63730947490855,-122.13266290859576
47.637519124743164,-122.13302232460376
47.637515510097955,-122.13301696018573
47.63764925180358,-122.13272728161212
47.63764925180358,-122.13274873928424
47.63784082716388,-122.13373579220172
47.63782998329432))" | "B" |
| 2 | "Redmond Building 40" | "POLYGON((-122.1336154507967
47.6366745947009,-122.13361008637867
47.636483015064535,-122.13349206918201
47.636479400347675,-122.13349743360004
47.63636372927573,-122.13372810357532
47.63636372927573,-122.13373346799335
47.63617576323771,-122.13263912671528
47.63616491902258,-122.13264985555134
47.63635649982525,-122.13304682248554
47.636367344000604,-122.13305218690357
47.63650831807564,-122.13276250832996
47.636497473929516,-122.13277323716602
47.63668543881025,-122.1336154507967
47.6366745947009))" | "A" |

| SITEID | SITENAME | GEOFENCE | ALLOWEDDEVICEID |
|--------|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------|
| 3 | "Redmond Building 22" | "POLYGON((-122.13611660248233 47.63758544698554,-122.13635263687564 47.6374083293018,-122.13622389084293 47.63733603619712,-122.13622389084293 47.63717699101473,-122.13581619507266 47.63692757827657,-122.13559625393344 47.637046862778135,-122.13569281345798 47.637144458985965,-122.13570890671207 47.637314348246214,-122.13611660248233 47.63758544698554))" | "C" |

Generate alerts with geofence

Devices can emit their ID and location every minute through a stream called `DeviceStreamInput`. The following table is a stream of input.

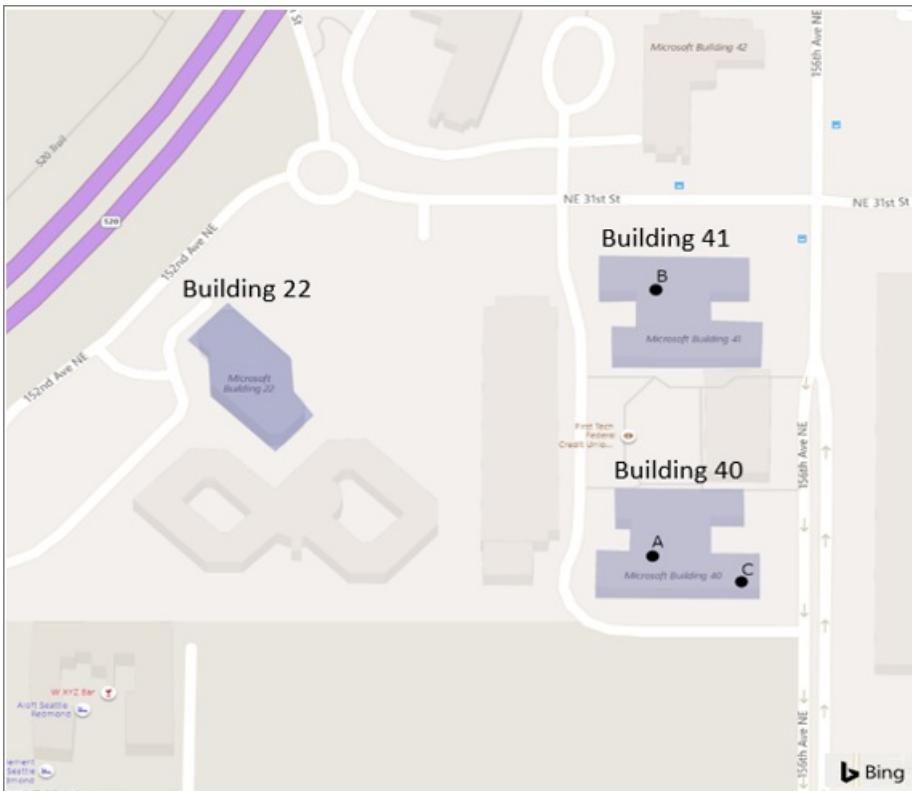
| DEVICEID | GEOPOSITION |
|----------|-------------------------------------------------|
| "A" | "POINT(-122.13292341559497 47.636318374032726)" |
| "B" | "POINT(-122.13338475554553 47.63743531308874)" |
| "C" | "POINT(-122.13354001095752 47.63627622505007)" |

You can write a query that joins the device stream with the geofence reference data and generates an alert every time a device is outside of an allowed building.

```

SELECT DeviceStreamInput.DeviceID, SiteReferenceInput.SiteID, SiteReferenceInput.SiteName
INTO Output
FROM DeviceStreamInput
JOIN SiteReferenceInput
ON st_within(DeviceStreamInput.GeoPosition, SiteReferenceInput.Geofence) = 0
WHERE DeviceStreamInput.DeviceID = SiteReferenceInput.AllowedDeviceID
    
```

The following image represents the geofences. You can see where the devices are in accordance to the stream data input.



Device "C" is located inside building ID 2, which is not allowed according to the reference data. This device should be located inside building ID 3. Running this job will generate an alert for this specific violation.

Site with multiple allowed devices

If a site allows multiple devices, an array of device IDs can be defined in `AllowedDeviceID` and a User-Defined Function can be used on the `WHERE` clause to verify if the stream device ID matches any device ID in that list. For more information, view the [Javascript UDF tutorial](#) for cloud jobs and the [C# UDF tutorial](#) for edge jobs.

Geospatial aggregation

Azure Stream Analytics supports low latency real-time geospatial aggregation in the cloud and on the IoT Edge runtime.

Geospatial aggregation scenario

A cab company wants to build a real-time application to guide their cab drivers looking for ride towards the areas of the cities currently experiencing higher demand.

The company stores logical regions of the city as reference data. Each region is defined by a `RegionID`, `RegionName`, and `Geofence`.

Define the geofences

The following table is an example of geofence reference data that could be stored in Azure blob storage or an Azure SQL table. Every region is represented by a geospatial polygon, which is used to correlate with the requests coming from streaming data.

These polygons are for reference only and do not represent actual city logical or physical separations.

| REGIONID | REGIONNAME | GEOFENCE |
|----------|------------|----------|
|----------|------------|----------|

| REGIONID | REGIONNAME | GEOFENCE |
|----------|-------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1 | "SoHo" | "POLYGON((-74.00279525078275
40.72833625216264,-
74.00547745979765
40.721929158663244,-
74.00125029839018
40.71893680218994,-
73.9957785919998
40.72521409075776,-
73.9972377137039
40.72557184584898,-
74.00279525078275
40.72833625216264))" |
| 2 | "Chinatown" | "POLYGON((-73.99712367114876
40.71281582267133,-
73.9901070123658
40.71336881907936,-
73.99023575839851
40.71452359088633,-
73.98976368961189
40.71554823078944,-
73.99551434573982
40.717337246783735,-
73.99480624255989
40.718491949759304,-
73.99652285632942
40.719109951574,-
73.99776740131233
40.7168005470334,-
73.99903340396736
40.71727219249899,-
74.00193018970344
40.71938642421256,-
74.00409741458748
40.71688186545551,-
74.00051398334358
40.71517415773184,-
74.0004281526551
40.714377212470005,-
73.99849696216438
40.713450141693166,-
73.99748845157478
40.71405192594819,-
73.99712367114876
40.71281582267133))" |
| 3 | "Tribeca" | "POLYGON((-74.01091641815208
40.72583120006787,-
74.01338405044578
40.71436586362705,-
74.01370591552757
40.713617702123415,-
74.00862044723533
40.711308107057235,-
74.00194711120628
40.7194238654018,-
74.01091641815208
40.72583120006787))" |

Aggregate data over a window of time

The following table contains streaming data of "rides."

| USERID | FROMLOCATION | TOLOCATION | TRIPREQUESTEDTIME |
|--------|---------------------------------------------------|---------------------------------------------------|------------------------|
| "A" | "POINT(-74.00726861389182
40.71610611981975)" | "POINT(-73.98615095917779
40.703107386025835)" | "2019-03-12T07:00:00Z" |
| "B" | "POINT(-74.00249841021645
40.723827238895666)" | "POINT(-74.01160699942085
40.71378884930115)" | "2019-03-12T07:01:00Z" |
| "C" | "POINT(-73.99680120565864
40.716439898624024)" | "POINT(-73.98289663412544
40.72582343969828)" | "2019-03-12T07:02:00Z" |
| "D" | "POINT(-74.00741090068288
40.71615626755086)" | "POINT(-73.97999843120539
40.73477895807408)" | "2019-03-12T07:03:00Z" |

The following query joins the device stream with the geofence reference data and calculates the number of requests per region on a time window of 15 minutes every minute.

```
SELECT count(*) as NumberOfRequests, RegionsRefDataInput.RegionName
FROM UserRequestStreamDataInput
JOIN RegionsRefDataInput
ON st_within(UserRequestStreamDataInput.FromLocation, RegionsRefDataInput.Geofence) = 1
GROUP BY RegionsRefDataInput.RegionName, hoppingwindow(minute, 1, 15)
```

This query outputs a count of requests every minute for the last 15 minutes by each region within the city. This information can be displayed easily by Power BI dashboard, or can be broadcasted to all drivers as SMS text messages through integration with services like Azure functions.

The image below illustrates the output of the query to Power BI dashboard.



Next steps

- [Introduction to Stream Analytics geospatial functions](#)

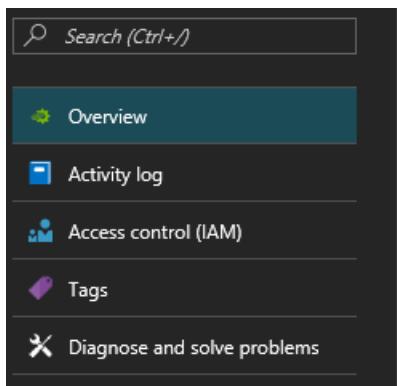
- GeoSpatial Functions (Azure Stream Analytics)

Understand Stream Analytics job monitoring and how to monitor queries

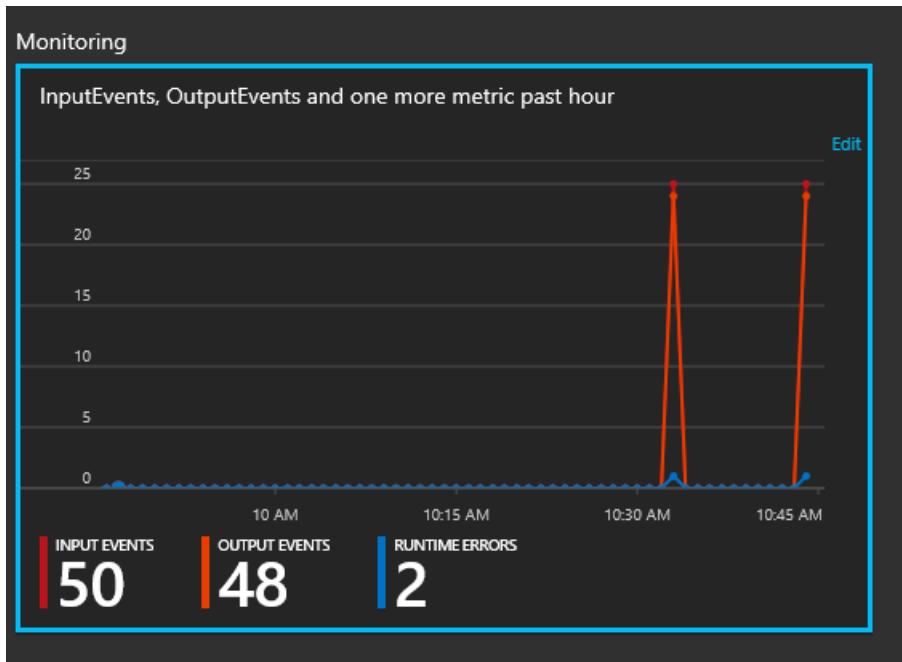
6/23/2019 • 3 minutes to read • [Edit Online](#)

Introduction: The monitor page

The Azure portal surfaces key performance metrics that can be used to monitor and troubleshoot your query and job performance. To see these metrics, browse to the Stream Analytics job you are interested in seeing metrics for and view the **Monitoring** section on the Overview page.



The window will appear as shown:



Metrics available for Stream Analytics

| METRIC | DEFINITION |
|--------|------------|
|--------|------------|

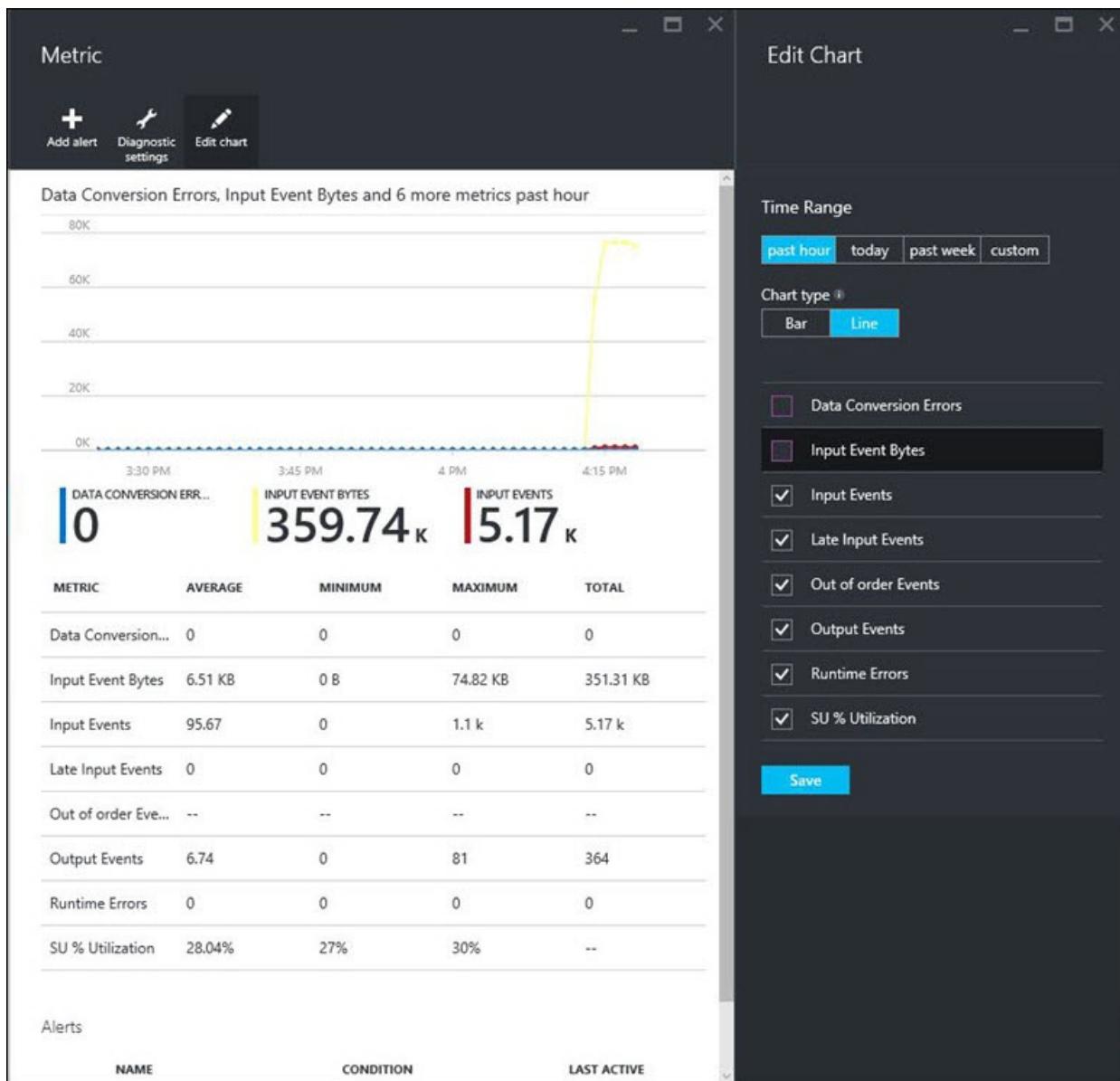
| METRIC | DEFINITION |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Backlogged Input Events | Number of input events that are backlogged. A non-zero value for this metric implies that your job isn't able to keep up with the number of incoming events. If this value is slowly increasing or consistently non-zero, you should scale out your job. You can learn more by visiting Understand and adjust Streaming Units . |
| Data Conversion Errors | Number of output events that could not be converted to the expected output schema. Error policy can be changed to 'Drop' to drop events that encounter this scenario. |
| Early Input Events | Events whose application timestamp is earlier than their arrival time by more than 5 minutes. |
| Failed Function Requests | Number of failed Azure Machine Learning function calls (if present). |
| Function Events | Number of events sent to the Azure Machine Learning function (if present). |
| Function Requests | Number of calls to the Azure Machine Learning function (if present). |
| Input Deserialization Errors | Number of input events that could not be deserialized. |
| Input Event Bytes | Amount of data received by the Stream Analytics job, in bytes. This can be used to validate that events are being sent to the input source. |
| Input Events | Number of records serialized from the input events. This count does not include incoming events that result in serialization errors. |
| Input Sources Received | Number of messages received by the job. For Event Hub, a message is a single EventData. For Blob, a message is a single blob. Please note that Input Sources are counted before serialization. If there are serialization errors, input sources can be greater than input events. Otherwise, it can be less than or equal to input events since each message can contain multiple events. |
| Late Input Events | Events that arrived later than the configured late arrival tolerance window. Learn more about Azure Stream Analytics event order considerations . |
| Out-of-Order Events | Number of events received out of order that were either dropped or given an adjusted timestamp, based on the Event Ordering Policy. This can be impacted by the configuration of the Out of Order Tolerance Window setting. |
| Output Events | Amount of data sent by the Stream Analytics job to the output target, in number of events. |
| Runtime Errors | Total number of errors related to query processing (excluding errors found while ingesting events or outputting results) |

| METRIC | DEFINITION |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SU % Utilization | The utilization of the Streaming Unit(s) assigned to a job from the Scale tab of the job. Should this indicator reach 80%, or above, there is high probability that event processing may be delayed or stopped making progress. |
| Watermark Delay | The maximum watermark delay across all partitions of all outputs in the job. |

You can use these metrics to [monitor the performance of your Stream Analytics job](#).

Customizing Monitoring in the Azure portal

You can adjust the type of chart, metrics shown, and time range in the Edit Chart settings. For details, see [How to Customize Monitoring](#).



Latest output

Another interesting data point to monitor your job is the time of the last output, shown in the Overview page. This time is the application time (i.e. the time using the timestamp from the event data) of the latest output of your job.

Get help

For further assistance, try our [Azure Stream Analytics forum](#)

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Monitor and manage Stream Analytics jobs with Azure PowerShell cmdlets

4/4/2019 • 11 minutes to read • [Edit Online](#)

Learn how to monitor and manage Stream Analytics resources with Azure PowerShell cmdlets and powershell scripting that execute basic Stream Analytics tasks.

NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

Prerequisites for running Azure PowerShell cmdlets for Stream Analytics

- Create an Azure Resource Group in your subscription. The following is a sample Azure PowerShell script. For Azure PowerShell information, see [Install and configure Azure PowerShell](#);

Azure PowerShell 0.9.8:

```
# Log in to your Azure account
Add-AzureAccount

# Select the Azure subscription you want to use to create the resource group if you have more than one
# subscription on your account.
Select-AzureSubscription -SubscriptionName <subscription name>
# If Stream Analytics has not been registered to the subscription, remove remark symbol below (#)to run the
#Register-AzureProvider cmdlet to register the provider namespace.
#Register-AzureProvider -Force -ProviderNamespace 'Microsoft.StreamAnalytics'
# Create an Azure resource group
New-AzureResourceGroup -Name <YOUR RESOURCE GROUP NAME> -Location <LOCATION>
```

Azure PowerShell 1.0:

```
# Log in to your Azure account
Connect-AzAccount

# Select the Azure subscription you want to use to create the resource group.
Get-AzSubscription -SubscriptionName "your sub" | Select-AzSubscription
# If Stream Analytics has not been registered to the subscription, remove remark symbol below (#)to run the
#Register-AzureProvider cmdlet to register the provider namespace.
#Register-AzResourceProvider -Force -ProviderNamespace 'Microsoft.StreamAnalytics'
# Create an Azure resource group
New-AzResourceGroup -Name <YOUR RESOURCE GROUP NAME> -Location <LOCATION>
```

NOTE

Stream Analytics jobs created programmatically do not have monitoring enabled by default. You can manually enable monitoring in the Azure Portal by navigating to the job's Monitor page and clicking the Enable button or you can do this programmatically by following the steps located at [Azure Stream Analytics - Monitor Stream Analytics Jobs Programmatically](#).

Azure PowerShell cmdlets for Stream Analytics

The following Azure PowerShell cmdlets can be used to monitor and manage Azure Stream Analytics jobs. Note that Azure PowerShell has different versions. **In the examples listed the first command is for Azure PowerShell 0.9.8, the second command is for Azure PowerShell 1.0.** The Azure PowerShell 1.0 commands will always have "Az" in the command.

Get-AzureStreamAnalyticsJob | Get-AzStreamAnalyticsJob

Lists all Stream Analytics jobs defined in the Azure subscription or specified resource group, or gets job information about a specific job within a resource group.

Example 1

Azure PowerShell 0.9.8:

```
Get-AzureStreamAnalyticsJob
```

Azure PowerShell 1.0:

```
Get-AzStreamAnalyticsJob
```

This PowerShell command returns information about all the Stream Analytics jobs in the Azure subscription.

Example 2

Azure PowerShell 0.9.8:

```
Get-AzureStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US
```

Azure PowerShell 1.0:

```
Get-AzStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US
```

This PowerShell command returns information about all the Stream Analytics jobs in the resource group StreamAnalytics-Default-Central-US.

Example 3

Azure PowerShell 0.9.8:

```
Get-AzureStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -Name StreamingJob
```

Azure PowerShell 1.0:

```
Get-AzStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -Name StreamingJob
```

This PowerShell command returns information about the Stream Analytics job StreamingJob in the resource group StreamAnalytics-Default-Central-US.

Get-AzureStreamAnalyticsInput | Get-AzStreamAnalyticsInput

Lists all of the inputs that are defined in a specified Stream Analytics job, or gets information about a specific input.

Example 1

Azure PowerShell 0.9.8:

```
Get-AzureStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob
```

Azure PowerShell 1.0:

```
Get-AzStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob
```

This PowerShell command returns information about all the inputs defined in the job StreamingJob.

Example 2

Azure PowerShell 0.9.8:

```
Get-AzureStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name EntryStream
```

Azure PowerShell 1.0:

```
Get-AzStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name EntryStream
```

This PowerShell command returns information about the input named EntryStream defined in the job StreamingJob.

Get-AzureStreamAnalyticsOutput | Get-AzStreamAnalyticsOutput

Lists all of the outputs that are defined in a specified Stream Analytics job, or gets information about a specific output.

Example 1

Azure PowerShell 0.9.8:

```
Get-AzureStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob
```

Azure PowerShell 1.0:

```
Get-AzStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob
```

This PowerShell command returns information about the outputs defined in the job StreamingJob.

Example 2

Azure PowerShell 0.9.8:

```
Get-AzureStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name Output
```

Azure PowerShell 1.0:

```
Get-AzStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name Output
```

This PowerShell command returns information about the output named Output defined in the job StreamingJob.

Get-AzureStreamAnalyticsQuota | Get-AzStreamAnalyticsQuota

Gets information about the quota of streaming units in a specified region.

Example 1

Azure PowerShell 0.9.8:

```
Get-AzureStreamAnalyticsQuota -Location "Central US"
```

Azure PowerShell 1.0:

```
Get-AzStreamAnalyticsQuota -Location "Central US"
```

This PowerShell command returns information about the quota and usage of streaming units in the Central US region.

Get-AzureStreamAnalyticsTransformation | Get-AzStreamAnalyticsTransformation

Gets information about a specific transformation defined in a Stream Analytics job.

Example 1

Azure PowerShell 0.9.8:

```
Get-AzureStreamAnalyticsTransformation -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name StreamingJob
```

Azure PowerShell 1.0:

```
Get-AzStreamAnalyticsTransformation -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name StreamingJob
```

This PowerShell command returns information about the transformation called StreamingJob in the job StreamingJob.

New-AzureStreamAnalyticsInput | New-AzStreamAnalyticsInput

Creates a new input within a Stream Analytics job, or updates an existing specified input.

The name of the input can be specified in the .json file or on the command line. If both are specified, the name on the command line must be the same as the one in the file.

If you specify an input that already exists and do not specify the –Force parameter, the cmdlet will ask whether or not to replace the existing input.

If you specify the –Force parameter and specify an existing input name, the input will be replaced without confirmation.

For detailed information on the JSON file structure and contents, refer to the [Create Input \(Azure Stream Analytics\)](#) section of the [Stream Analytics Management REST API Reference Library](#).

Example 1

Azure PowerShell 0.9.8:

```
New-AzureStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -File "C:\Input.json"
```

Azure PowerShell 1.0:

```
New-AzStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -File "C:\Input.json"
```

This PowerShell command creates a new input from the file Input.json. If an existing input with the name specified in the input definition file is already defined, the cmdlet will ask whether or not to replace it.

Example 2

Azure PowerShell 0.9.8:

```
New-AzureStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -File "C:\Input.json" -Name EntryStream
```

Azure PowerShell 1.0:

```
New-AzStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -File "C:\Input.json" -Name EntryStream
```

This PowerShell command creates a new input in the job called EntryStream. If an existing input with this name is already defined, the cmdlet will ask whether or not to replace it.

Example 3

Azure PowerShell 0.9.8:

```
New-AzureStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -File "C:\Input.json" -Name EntryStream -Force
```

Azure PowerShell 1.0:

```
New-AzStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -File "C:\Input.json" -Name EntryStream -Force
```

This PowerShell command replaces the definition of the existing input source called EntryStream with the definition from the file.

New-AzureStreamAnalyticsJob | New-AzStreamAnalyticsJob

Creates a new Stream Analytics job in Microsoft Azure, or updates the definition of an existing specified job.

The name of the job can be specified in the json file or on the command line. If both are specified, the name on the command line must be the same as the one in the file.

If you specify a job name that already exists and do not specify the –Force parameter, the cmdlet will ask whether or not to replace the existing job.

If you specify the –Force parameter and specify an existing job name, the job definition will be replaced without confirmation.

For detailed information on the JSON file structure and contents, refer to the [Create Stream Analytics Job](#) section

of the [Stream Analytics Management REST API Reference Library](#).

Example 1

Azure PowerShell 0.9.8:

```
New-AzureStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -File "C:\JobDefinition.json"
```

Azure PowerShell 1.0:

```
New-AzStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -File "C:\JobDefinition.json"
```

This PowerShell command creates a new job from the definition in JobDefinition.json. If an existing job with the name specified in the job definition file is already defined, the cmdlet will ask whether or not to replace it.

Example 2

Azure PowerShell 0.9.8:

```
New-AzureStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -File "C:\JobDefinition.json" -Name StreamingJob -Force
```

Azure PowerShell 1.0:

```
New-AzStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -File "C:\JobDefinition.json" -Name StreamingJob -Force
```

This PowerShell command replaces the job definition for StreamingJob.

New-AzureStreamAnalyticsOutput | New-AzStreamAnalyticsOutput

Creates a new output within a Stream Analytics job, or updates an existing output.

The name of the output can be specified in the json file or on the command line. If both are specified, the name on the command line must be the same as the one in the file.

If you specify an output that already exists and do not specify the –Force parameter, the cmdlet will ask whether or not to replace the existing output.

If you specify the –Force parameter and specify an existing output name, the output will be replaced without confirmation.

For detailed information on the JSON file structure and contents, refer to the [Create Output \(Azure Stream Analytics\)](#) section of the [Stream Analytics Management REST API Reference Library](#).

Example 1

Azure PowerShell 0.9.8:

```
New-AzureStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -File "C:\Output.json" -JobName StreamingJob -Name output
```

Azure PowerShell 1.0:

```
New-AzStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -File "C:\Output.json" -  
JobName StreamingJob -Name output
```

This PowerShell command creates a new output called "output" in the job StreamingJob. If an existing output with this name is already defined, the cmdlet will ask whether or not to replace it.

Example 2

Azure PowerShell 0.9.8:

```
New-AzureStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -File "C:\Output.json" -  
JobName StreamingJob -Name output -Force
```

Azure PowerShell 1.0:

```
New-AzStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -File "C:\Output.json" -  
JobName StreamingJob -Name output -Force
```

This PowerShell command replaces the definition for "output" in the job StreamingJob.

New-AzureStreamAnalyticsTransformation | New-AzStreamAnalyticsTransformation

Creates a new transformation within a Stream Analytics job, or updates the existing transformation.

The name of the transformation can be specified in the json file or on the command line. If both are specified, the name on the command line must be the same as the one in the file.

If you specify a transformation that already exists and do not specify the –Force parameter, the cmdlet will ask whether or not to replace the existing transformation.

If you specify the –Force parameter and specify an existing transformation name, the transformation will be replaced without confirmation.

For detailed information on the JSON file structure and contents, refer to the [Create Transformation \(Azure Stream Analytics\)](#) section of the [Stream Analytics Management REST API Reference Library](#).

Example 1

Azure PowerShell 0.9.8:

```
New-AzureStreamAnalyticsTransformation -ResourceGroupName StreamAnalytics-Default-Central-US -File  
"C:\Transformation.json" -JobName StreamingJob -Name StreamingJobTransform
```

Azure PowerShell 1.0:

```
New-AzStreamAnalyticsTransformation -ResourceGroupName StreamAnalytics-Default-Central-US -File  
"C:\Transformation.json" -JobName StreamingJob -Name StreamingJobTransform
```

This PowerShell command creates a new transformation called StreamingJobTransform in the job StreamingJob. If an existing transformation is already defined with this name, the cmdlet will ask whether or not to replace it.

Example 2

Azure PowerShell 0.9.8:

```
New-AzureStreamAnalyticsTransformation -ResourceGroupName StreamAnalytics-Default-Central-US -File "C:\Transformation.json" -JobName StreamingJob -Name StreamingJobTransform -Force
```

Azure PowerShell 1.0:

```
New-AzStreamAnalyticsTransformation -ResourceGroupName StreamAnalytics-Default-Central-US -File "C:\Transformation.json" -JobName StreamingJob -Name StreamingJobTransform -Force
```

This PowerShell command replaces the definition of StreamingJobTransform in the job StreamingJob.

Remove-AzureStreamAnalyticsInput | Remove-AzStreamAnalyticsInput

Asynchronously deletes a specific input from a Stream Analytics job in Microsoft Azure.

If you specify the –Force parameter, the input will be deleted without confirmation.

Example 1

Azure PowerShell 0.9.8:

```
Remove-AzureStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name EventStream
```

Azure PowerShell 1.0:

```
Remove-AzStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name EventStream
```

This PowerShell command removes the input EventStream in the job StreamingJob.

Remove-AzureStreamAnalyticsJob | Remove-AzStreamAnalyticsJob

Asynchronously deletes a specific Stream Analytics job in Microsoft Azure.

If you specify the –Force parameter, the job will be deleted without confirmation.

Example 1

Azure PowerShell 0.9.8:

```
Remove-AzureStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -Name StreamingJob
```

Azure PowerShell 1.0:

```
Remove-AzStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -Name StreamingJob
```

This PowerShell command removes the job StreamingJob.

Remove-AzureStreamAnalyticsOutput | Remove-AzStreamAnalyticsOutput

Asynchronously deletes a specific output from a Stream Analytics job in Microsoft Azure.

If you specify the –Force parameter, the output will be deleted without confirmation.

Example 1

Azure PowerShell 0.9.8:

```
Remove-AzureStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -  
Name Output
```

Azure PowerShell 1.0:

```
Remove-AzStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -  
Name Output
```

This PowerShell command removes the output Output in the job StreamingJob.

Start-AzureStreamAnalyticsJob | Start-AzStreamAnalyticsJob

Asynchronously deploys and starts a Stream Analytics job in Microsoft Azure.

Example 1

Azure PowerShell 0.9.8:

```
Start-AzureStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -Name StreamingJob -  
OutputStartTime CustomTime -OutputStartTime 2012-12-12T12:12:12Z
```

Azure PowerShell 1.0:

```
Start-AzStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -Name StreamingJob -  
OutputStartTime CustomTime -OutputStartTime 2012-12-12T12:12:12Z
```

This PowerShell command starts the job StreamingJob with a custom output start time set to December 12, 2012, 12:12:12 UTC.

Stop-AzureStreamAnalyticsJob | Stop-AzStreamAnalyticsJob

Asynchronously stops a Stream Analytics job from running in Microsoft Azure and de-allocates resources that were being used. The job definition and metadata will remain available within your subscription through both the Azure portal and management APIs, such that the job can be edited and restarted. You will not be charged for a job in the stopped state.

Example 1

Azure PowerShell 0.9.8:

```
Stop-AzureStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -Name StreamingJob
```

Azure PowerShell 1.0:

```
Stop-AzStreamAnalyticsJob -ResourceGroupName StreamAnalytics-Default-Central-US -Name StreamingJob
```

This PowerShell command stops the job StreamingJob.

Test-AzureStreamAnalyticsInput | Test-AzStreamAnalyticsInput

Tests the ability of Stream Analytics to connect to a specified input.

Example 1

Azure PowerShell 0.9.8:

```
Test-AzureStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name EntryStream
```

Azure PowerShell 1.0:

```
Test-AzStreamAnalyticsInput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name EntryStream
```

This PowerShell command tests the connection status of the input EntryStream in StreamingJob.

Test-AzureStreamAnalyticsOutput | Test-AzStreamAnalyticsOutput

Tests the ability of Stream Analytics to connect to a specified output.

Example 1

Azure PowerShell 0.9.8:

```
Test-AzureStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name Output
```

Azure PowerShell 1.0:

```
Test-AzStreamAnalyticsOutput -ResourceGroupName StreamAnalytics-Default-Central-US -JobName StreamingJob -Name Output
```

This PowerShell command tests the connection status of the output Output in StreamingJob.

Get support

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Programmatically create a Stream Analytics job monitor

5/29/2019 • 3 minutes to read • [Edit Online](#)

This article demonstrates how to enable monitoring for a Stream Analytics job. Stream Analytics jobs that are created via REST APIs, Azure SDK, or PowerShell do not have monitoring enabled by default. You can manually enable it in the Azure portal by going to the job's Monitor page and clicking the Enable button or you can automate this process by following the steps in this article. The monitoring data will show up in the Metrics area of the Azure portal for your Stream Analytics job.

Prerequisites

Before you begin this process, you must have the following prerequisites:

- Visual Studio 2019 or 2015
- [Azure .NET SDK](#) downloaded and installed
- An existing Stream Analytics job that needs to have monitoring enabled

Create a project

1. Create a Visual Studio C# .NET console application.
2. In the Package Manager Console, run the following commands to install the NuGet packages. The first one is the Azure Stream Analytics Management .NET SDK. The second one is the Azure Monitor SDK that will be used to enable monitoring. The last one is the Azure Active Directory client that will be used for authentication.

```
Install-Package Microsoft.Azure.Management.StreamAnalytics
Install-Package Microsoft.Azure.Insights -Pre
Install-Package Microsoft.IdentityModel.Clients.ActiveDirectory
```

3. Add the following appSettings section to the App.config file.

```
<appSettings>
<!--CSM Prod related values-->
<add key="ResourceGroupName" value="RESOURCE GROUP NAME" />
<add key="JobName" value="YOUR JOB NAME" />
<add key="StorageAccountName" value="YOUR STORAGE ACCOUNT"/>
<add key="ActiveDirectoryEndpoint" value="https://login.microsoftonline.com/" />
<add key="ResourceManagerEndpoint" value="https://management.azure.com/" />
<add key="WindowsManagementUri" value="https://management.core.windows.net/" />
<add key="AsaClientId" value="1950a258-227b-4e31-a9cf-717495945fc2" />
<add key="RedirectUri" value="urn:ietf:wg:oauth:2.0:oob" />
<add key="SubscriptionId" value="YOUR AZURE SUBSCRIPTION ID" />
<add key="ActiveDirectoryTenantId" value="YOUR TENANT ID" />
</appSettings>
```

Replace values for *SubscriptionId* and *ActiveDirectoryTenantId* with your Azure subscription and tenant IDs. You can get these values by running the following PowerShell cmdlet:

```
Get-AzureAccount
```

4. Add the following using statements to the source file (Program.cs) in the project.

```
using System;
using System.Configuration;
using System.Threading;
using Microsoft.Azure;
using Microsoft.Azure.Management.Insights;
using Microsoft.Azure.Management.Insights.Models;
using Microsoft.Azure.Management.StreamAnalytics;
using Microsoft.Azure.Management.StreamAnalytics.Models;
using Microsoft.IdentityModel.Clients.ActiveDirectory;
```

5. Add an authentication helper method.

```
public static string GetAuthorizationHeader()
{
    AuthenticationResult result = null;
    var thread = new Thread(() =>
    {
        try
        {
            var context = new AuthenticationContext(
                ConfigurationManager.AppSettings["ActiveDirectoryEndpoint"] +
                ConfigurationManager.AppSettings["ActiveDirectoryTenantId"]);
            result = context.AcquireToken(
                resource: ConfigurationManager.AppSettings["WindowsManagementUri"],
                clientId: ConfigurationManager.AppSettings["AsaClientId"],
                redirectUri: new Uri(ConfigurationManager.AppSettings["RedirectUri"]),
                promptBehavior: PromptBehavior.Always);
        }
        catch (Exception threadEx)
        {
            Console.WriteLine(threadEx.Message);
        }
    });
    thread.SetApartmentState(ApartmentState.STA);
    thread.Name = "AcquireTokenThread";
    thread.Start();
    thread.Join();

    if (result != null)
    {
        return result.AccessToken;
    }
    throw new InvalidOperationException("Failed to acquire token");
}
```

Create management clients

The following code will set up the necessary variables and management clients.

```

string resourceGroupName = "<YOUR AZURE RESOURCE GROUP NAME>";
string streamAnalyticsJobName = "<YOUR STREAM ANALYTICS JOB NAME>";

// Get authentication token
TokenCloudCredentials aadTokenCredentials =
    new TokenCloudCredentials(
        ConfigurationManager.AppSettings["SubscriptionId"],
        GetAuthorizationHeader());

Uri resourceManagerUri = new
Uri(ConfigurationManager.AppSettings["ResourceManagerEndpoint"]);

// Create Stream Analytics and Insights management client
StreamAnalyticsManagementClient streamAnalyticsClient = new
StreamAnalyticsManagementClient(aadTokenCredentials, resourceManagerUri);
InsightsManagementClient insightsClient = new
InsightsManagementClient(aadTokenCredentials, resourceManagerUri);

```

Enable monitoring for an existing Stream Analytics job

The following code enables monitoring for an **existing** Stream Analytics job. The first part of the code performs a GET request against the Stream Analytics service to retrieve information about the particular Stream Analytics job. It uses the *ID* property (retrieved from the GET request) as a parameter for the Put method in the second half of the code, which sends a PUT request to the Insights service to enable monitoring for the Stream Analytics job.

WARNING

If you have previously enabled monitoring for a different Stream Analytics job, either through the Azure portal or programmatically via the below code, **we recommend that you provide the same storage account name that you used when you previously enabled monitoring.**

The storage account is linked to the region that you created your Stream Analytics job in, not specifically to the job itself.

All Stream Analytics jobs (and all other Azure resources) in that same region share this storage account to store monitoring data. If you provide a different storage account, it might cause unintended side effects in the monitoring of your other Stream Analytics jobs or other Azure resources.

The storage account name that you use to replace <YOUR STORAGE ACCOUNT NAME> in the following code should be a storage account that is in the same subscription as the Stream Analytics job that you are enabling monitoring for.

```

// Get an existing Stream Analytics job
JobGetParameters jobGetParameters = new JobGetParameters()
{
    PropertiesToExpand = "inputs,transformation,outputs"
};
JobGetResponse jobGetResponse = streamAnalyticsClient.StreamingJobs.Get(resourceGroupName,
streamAnalyticsJobName, jobGetParameters);

// Enable monitoring
ServiceDiagnosticSettingsPutParameters insightPutParameters = new ServiceDiagnosticSettingsPutParameters()
{
    Properties = new ServiceDiagnosticSettings()
    {
        StorageAccountName = "<YOUR STORAGE ACCOUNT NAME>"
    }
};
insightsClient.ServiceDiagnosticSettingsOperations.Put(jobGetResponse.Job.Id, insightPutParameters);

```

Get support

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Monitor and manage Stream Analytics jobs with Visual Studio

12/7/2018 • 2 minutes to read • [Edit Online](#)

This article demonstrates how to monitor your Stream Analytics job in Visual Studio. Azure Stream Analytics tools for Visual Studio provides a monitoring experience similar to Azure portal without having to leave the IDE. You can begin to monitor a job as soon as you **Submit to Azure** from your **Script.asaql**, or you can monitor existing jobs regardless of how they were created.

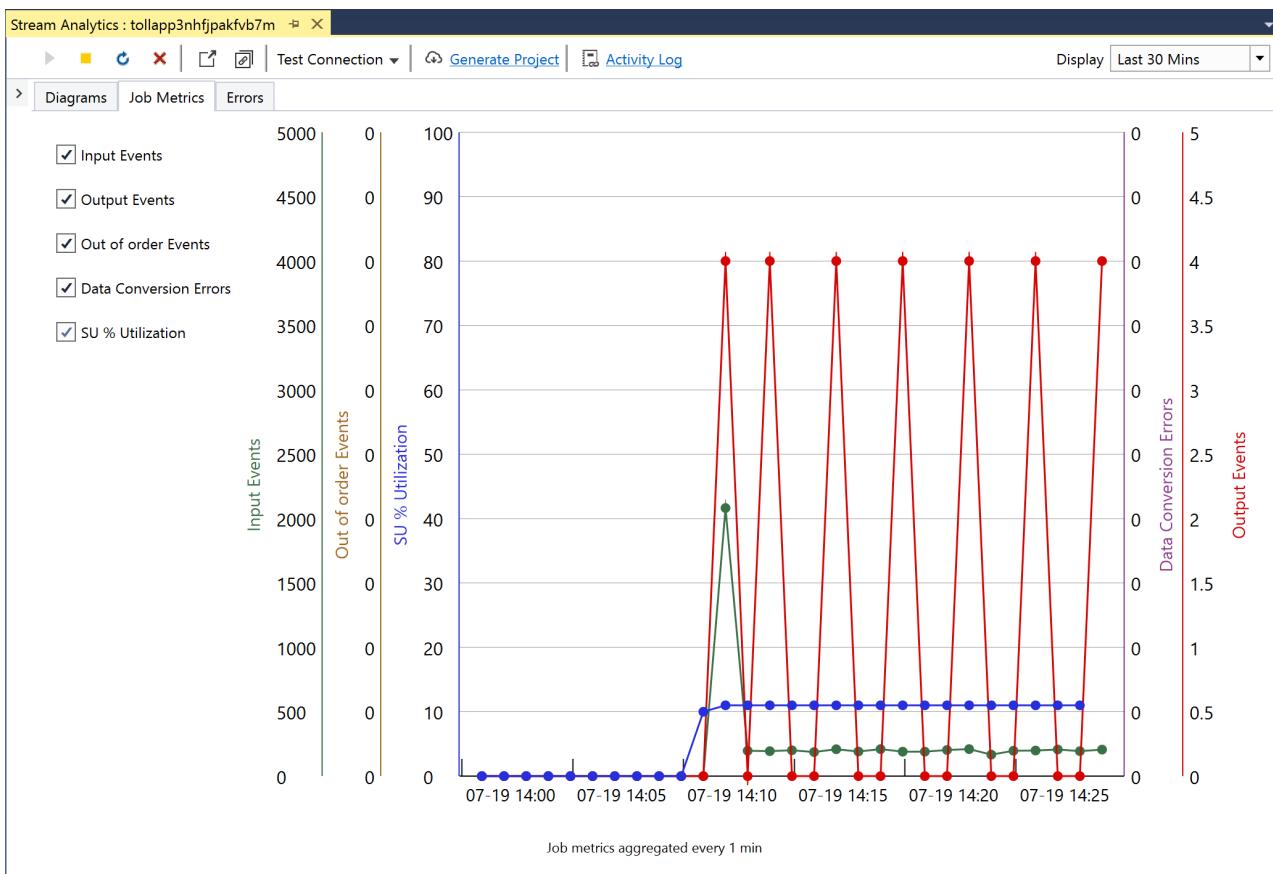
Job summary

The **Job Summary** and **Job Metrics** give a quick snapshot of your job. At a glance, you can determine a job's status and event information.]

Job Summary	
Status	Running
Job Type	Cloud
Creation Time	7/19/2018 11:52:31 AM
Job Output Start Mode	CustomTime
Job Output Start Time	7/19/2018 2:08:18 PM
Last Output Time	7/19/2018 2:27:00 PM
Data Locale	en-US
Output Error Handling	Retry
Late Arrival Tolerance	5 Second(s)
Out of Order Tolerance	5 Second(s)
Out of Order Actions	Adjust
Compatibility Level	1.0
Stream Analytics Units	1
Job Metrics (Last 30 Mins)	
Total Input Events	5.43 K
Input Event Size	1.07 MiB
Total Output Events	28
Late Input Events	0
Out of Order Events	0
Events Handled by Functions	0
External Function Calls	0
Failed External Function Calls	0
Data Conversion Errors	0

Job metrics

You can collapse the **Job Summary** and click the **Job Metrics** tab to view a graph with important metrics. Check and un-check metrics types to add and remove them from the graph.



Error monitoring

You can also monitor errors by clicking on the **Errors** tab.

ERROR(S)

<input checked="" type="button"/> 8/7/2018 9:50:43 PM	Diagnostic: Diagnostic Error
<input checked="" type="button"/> 8/7/2018 9:50:39 PM	Serialize Events

DETAILS

EventTimestamp	8/7/2018 9:50:43 PM
Level	Error
Description	N/A
OperationName	Diagnostic: Diagnostic Error
Status	Failed
Message Time	2018-08-07 21:50:43Z
Error	N/A
Message	First Occurred: 08/07/2018 21:50:39 Resource Name: Input Message: Fields referenced in query not found in input events Fields Expected: DATE Fields found: TIME, DSPL, TEMP, HMDT Missing fields: DATE
Type	DiagnosticMessage
Correlation ID	c9daa038-e9ae-41b0-85d3-c6135c262b2b

Get support

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Create an Azure Stream Analytics job with Visual Studio](#)
- [Install Azure Stream Analytics tools for Visual Studio](#)

Understand and adjust Streaming Units

6/21/2019 • 8 minutes to read • [Edit Online](#)

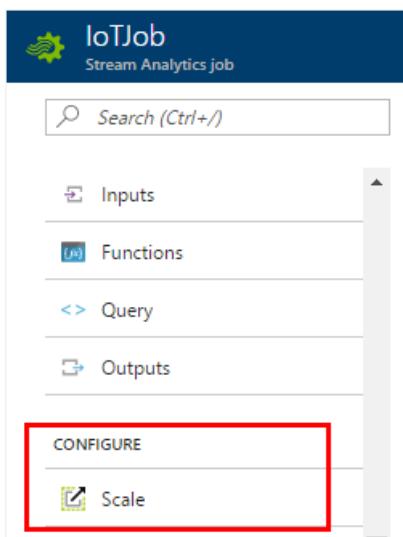
Streaming Units (SUs) represents the computing resources that are allocated to execute a Stream Analytics job. The higher the number of SUs, the more CPU and memory resources are allocated for your job. This capacity lets you focus on the query logic and abstracts the need to manage the hardware to run your Stream Analytics job in a timely manner.

To achieve low latency stream processing, Azure Stream Analytics jobs perform all processing in memory. When running out of memory, the streaming job fails. As a result, for a production job, it's important to monitor a streaming job's resource usage, and make sure there is enough resource allocated to keep the jobs running 24/7.

The SU % utilization metric, which ranges from 0% to 100%, describes the memory consumption of your workload. For a streaming job with minimal footprint, this metric is usually between 10% to 20%. If SU% utilization is low and input events get backlogged, your workload likely requires more compute resources, which requires you to increase the number of SUs. It's best to keep the SU metric below 80% to account for occasional spikes. Microsoft recommends setting an alert on 80% SU Utilization metric to prevent resource exhaustion. For more information, see [Tutorial: Set up alerts for Azure Stream Analytics jobs](#).

Configure Stream Analytics Streaming Units (SUs)

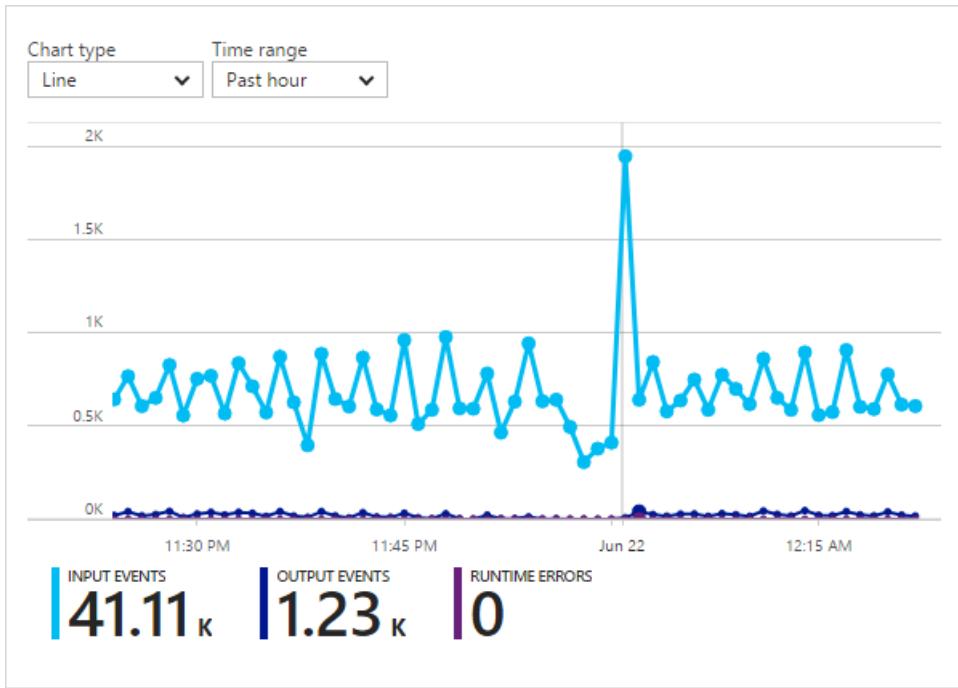
1. Sign in to [Azure portal](#)
2. In the list of resources, find the Stream Analytics job that you want to scale and then open it.
3. In the job page, under the **Configure** heading, select **Scale**.



4. Use the slider to set the SUs for the job. Notice that you are limited to specific SU settings.

Monitor job performance

Using the Azure portal, you can track the throughput of a job:



Calculate the expected throughput of the workload. If the throughput is less than expected, tune the input partition, tune the query, and add SUs to your job.

How many SUs are required for a job?

Choosing the number of required SUs for a particular job depends on the partition configuration for the inputs and the query that's defined within the job. The **Scale** page allows you to set the right number of SUs. It is a best practice to allocate more SUs than needed. The Stream Analytics processing engine optimizes for latency and throughput at the cost of allocating additional memory.

In general, the best practice is to start with 6 SUs for queries that don't use **PARTITION BY**. Then determine the sweet spot by using a trial and error method in which you modify the number of SUs after you pass representative amounts of data and examine the SU% Utilization metric. The maximum number of streaming units that can be used by a Stream Analytics job depends on the number of steps in the query defined for the job and the number of partitions in each step. You can learn more about the limits [here](#).

For more information about choosing the right number of SUs, see this page: [Scale Azure Stream Analytics jobs to increase throughput](#)

NOTE

Choosing how many SUs are required for a particular job depends on the partition configuration for the inputs and on the query defined for the job. You can select up to your quota in SUs for a job. By default, each Azure subscription has a quota of up to 500 SUs for all the analytics jobs in a specific region. To increase SUs for your subscriptions beyond this quota, contact [Microsoft Support](#). Valid values for SUs per job are 1, 3, 6, and up in increments of 6.

Factors that increase SU% utilization

Temporal (time-oriented) query elements are the core set of stateful operators provided by Stream Analytics. Stream Analytics manages the state of these operations internally on user's behalf, by managing memory consumption, checkpointing for resiliency, and state recovery during service upgrades. Even though Stream Analytics fully manages the states, there are a number of best practice recommendations that users should consider.

Note that a job with complex query logic could have high SU% utilization even when it is not continuously receiving input events. This can happen after a sudden spike in input and output events. The job might continue

to maintain state in memory if the query is complex.

SU% utilization may suddenly drop to 0 for a short period before coming back to expected levels. This happens due to transient errors or system initiated upgrades. Increasing number of streaming units for a job might not reduce SU% Utilization if your query is not [fully parallel](#).

Stateful query logic in temporal elements

One of the unique capability of Azure Stream Analytics job is to perform stateful processing, such as windowed aggregates, temporal joins, and temporal analytic functions. Each of these operators keeps state information. The maximum window size for these query elements is seven days.

The temporal window concept appears in several Stream Analytics query elements:

1. Windowed aggregates: GROUP BY of Tumbling, Hopping, and Sliding windows
2. Temporal joins: JOIN with DATEDIFF function
3. Temporal analytic functions: ISFIRST, LAST, and LAG with LIMIT DURATION

The following factors influence the memory used (part of streaming units metric) by Stream Analytics jobs:

Windowed aggregates

The memory consumed (state size) for a windowed aggregate is not always directly proportional to the window size. Instead, the memory consumed is proportional to the cardinality of the data, or the number of groups in each time window.

For example, in the following query, the number associated with `clusterid` is the cardinality of the query.

```
SELECT count(*)  
FROM input  
GROUP BY clusterid, tumblingwindow (minutes, 5)
```

In order to mitigate any issues caused by high cardinality in the previous query, you can send events to Event Hub partitioned by `clusterid`, and scale out the query by allowing the system to process each input partition separately using **PARTITION BY** as shown in the example below:

```
SELECT count(*)  
FROM input PARTITION BY PartitionId  
GROUP BY PartitionId, clusterid, tumblingwindow (minutes, 5)
```

Once the query is partitioned out, it is spread out over multiple nodes. As a result, the number of `clusterid` values coming into each node is reduced thereby reducing the cardinality of the group by operator.

Event Hub partitions should be partitioned by the grouping key to avoid the need for a reduce step. For more information, see [Event Hubs overview](#).

Temporal joins

The memory consumed (state size) of a temporal join is proportional to the number of events in the temporal wiggle room of the join, which is event input rate multiply by the wiggle room size. In other words, the memory consumed by joins is proportional to the DateDiff time range multiplied by average event rate.

The number of unmatched events in the join affect the memory utilization for the query. The following query is looking to find the ad impressions that generate clicks:

```
SELECT clicks.id
FROM clicks
INNER JOIN impressions ON impressions.id = clicks.id AND DATEDIFF(hour, impressions, clicks) between 0 AND 10.
```

In this example, it is possible that lots of ads are shown and few people click on it and it is required to keep all the events in the time window. Memory consumed is proportional to the window size and event rate.

To remediate this, send events to Event Hub partitioned by the join keys (id in this case), and scale out the query by allowing the system to process each input partition separately using **PARTITION BY** as shown:

```
SELECT clicks.id
FROM clicks PARTITION BY PartitionId
INNER JOIN impressions PARTITION BY PartitionId
ON impression.PartitionId = clicks.PartitionId AND impressions.id = clicks.id AND DATEDIFF(hour, impressions, clicks) between 0 AND 10
```

Once the query is partitioned out, it is spread out over multiple nodes. As a result the number of events coming into each node is reduced thereby reducing the size of the state kept in the join window.

Temporal analytic functions

The memory consumed (state size) of a temporal analytic function is proportional to the event rate multiply by the duration. The memory consumed by analytic functions is not proportional to the window size, but rather partition count in each time window.

The remediation is similar to temporal join. You can scale out the query using **PARTITION BY**.

Out of order buffer

User can configure the out of order buffer size in the Event Ordering configuration pane. The buffer is used to hold inputs for the duration of the window, and reorder them. The size of the buffer is proportional to the event input rate multiply by the out of order window size. The default window size is 0.

To remediate overflow of the out of order buffer, scale out query using **PARTITION BY**. Once the query is partitioned out, it is spread out over multiple nodes. As a result, the number of events coming into each node is reduced thereby reducing the number of events in each reorder buffer.

Input partition count

Each input partition of a job input has a buffer. The larger number of input partitions, the more resource the job consumes. For each streaming unit, Azure Stream Analytics can process roughly 1 MB/s of input. Therefore, you can optimize by matching the number of Stream Analytics streaming units with the number of partitions in your Event Hub.

Typically, a job configured with one streaming unit is sufficient for an Event Hub with two partitions (which is the minimum for Event Hub). If the Event Hub has more partitions, your Stream Analytics job consumes more resources, but not necessarily uses the extra throughput provided by Event Hub.

For a job with 6 streaming units, you may need 4 or 8 partitions from the Event Hub. However, avoid too many unnecessary partitions since that causes excessive resource usage. For example, an Event Hub with 16 partitions or larger in a Stream Analytics job that has 1 streaming unit.

Reference data

Reference data in ASA are loaded into memory for fast lookup. With the current implementation, each join operation with reference data keeps a copy of the reference data in memory, even if you join with the same reference data multiple times. For queries with **PARTITION BY**, each partition has a copy of the reference data, so the partitions are fully decoupled. With the multiplier effect, memory usage can quickly get very high if you join with reference data multiple times with multiple partitions.

Use of UDF functions

When you add a UDF function, Azure Stream Analytics loads the JavaScript runtime into memory. This will affect the SU%.

Next steps

- [Create parallelizable queries in Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs to increase throughput](#)

Leverage query parallelization in Azure Stream Analytics

6/21/2019 • 10 minutes to read • [Edit Online](#)

This article shows you how to take advantage of parallelization in Azure Stream Analytics. You learn how to scale Stream Analytics jobs by configuring input partitions and tuning the analytics query definition. As a prerequisite, you may want to be familiar with the notion of Streaming Unit described in [Understand and adjust Streaming Units](#).

What are the parts of a Stream Analytics job?

A Stream Analytics job definition includes inputs, a query, and output. Inputs are where the job reads the data stream from. The query is used to transform the data input stream, and the output is where the job sends the job results to.

A job requires at least one input source for data streaming. The data stream input source can be stored in an Azure event hub or in Azure blob storage. For more information, see [Introduction to Azure Stream Analytics](#) and [Get started using Azure Stream Analytics](#).

Partitions in sources and sinks

Scaling a Stream Analytics job takes advantage of partitions in the input or output. Partitioning lets you divide data into subsets based on a partition key. A process that consumes the data (such as a Stream Analytics job) can consume and write different partitions in parallel, which increases throughput.

Inputs

All Azure Stream Analytics input can take advantage of partitioning:

- EventHub (need to set the partition key explicitly with PARTITION BY keyword)
- IoT Hub (need to set the partition key explicitly with PARTITION BY keyword)
- Blob storage

Outputs

When you work with Stream Analytics, you can take advantage of partitioning in the outputs:

- Azure Data Lake Storage
- Azure Functions
- Azure Table
- Blob storage (can set the partition key explicitly)
- Cosmos DB (need to set the partition key explicitly)
- Event Hubs (need to set the partition key explicitly)
- IoT Hub (need to set the partition key explicitly)
- Service Bus
- SQL and SQL Data Warehouse with optional partitioning: see more information on the [Output to Azure SQL Database page](#).

Power BI doesn't support partitioning. However you can still partition the input as described in [this section](#)

For more information about partitions, see the following articles:

- [Event Hubs features overview](#)
- [Data partitioning](#)

Embarrassingly parallel jobs

An *embarrassingly parallel* job is the most scalable scenario we have in Azure Stream Analytics. It connects one partition of the input to one instance of the query to one partition of the output. This parallelism has the following requirements:

1. If your query logic depends on the same key being processed by the same query instance, you must make sure that the events go to the same partition of your input. For Event Hubs or IoT Hub, this means that the event data must have the **PartitionKey** value set. Alternatively, you can use partitioned senders. For blob storage, this means that the events are sent to the same partition folder. If your query logic does not require the same key to be processed by the same query instance, you can ignore this requirement. An example of this logic would be a simple select-project-filter query.
2. Once the data is laid out on the input side, you must make sure that your query is partitioned. This requires you to use **PARTITION BY** in all the steps. Multiple steps are allowed, but they all must be partitioned by the same key. Under compatibility level 1.0 and 1.1, the partitioning key must be set to **PartitionId** in order for the job to be fully parallel. For jobs with compatibility level 1.2 and higher, custom column can be specified as Partition Key in the input settings and the job will be parallelized automatically even without PARTITION BY clause.
3. Most of our output can take advantage of partitioning, however if you use an output type that doesn't support partitioning your job won't be fully parallel. Refer to the [output section](#) for more details.
4. The number of input partitions must equal the number of output partitions. Blob storage output can support partitions and inherits the partitioning scheme of the upstream query. When a partition key for Blob storage is specified, data is partitioned per input partition thus the result is still fully parallel. Here are examples of partition values that allow a fully parallel job:
 - 8 event hub input partitions and 8 event hub output partitions
 - 8 event hub input partitions and blob storage output
 - 8 event hub input partitions and blob storage output partitioned by a custom field with arbitrary cardinality
 - 8 blob storage input partitions and blob storage output
 - 8 blob storage input partitions and 8 event hub output partitions

The following sections discuss some example scenarios that are embarrassingly parallel.

Simple query

- Input: Event hub with 8 partitions
- Output: Event hub with 8 partitions

Query:

```
SELECT TollBoothId
FROM Input1 Partition By PartitionId
WHERE TollBoothId > 100
```

This query is a simple filter. Therefore, we don't need to worry about partitioning the input that is being sent to the event hub. Notice that jobs with compatibility level before 1.2 must include **PARTITION BY PartitionId** clause, so it fulfills requirement #2 from earlier. For the output, we need to configure the event hub output in the job to have the partition key set to **PartitionId**. One last check is to make sure that the number of input partitions is equal to the number of output partitions.

Query with a grouping key

- Input: Event hub with 8 partitions
- Output: Blob storage

Query:

```
SELECT COUNT(*) AS Count, TollBoothId  
FROM Input1 Partition By PartitionId  
GROUP BY TumblingWindow(minute, 3), TollBoothId, PartitionId
```

This query has a grouping key. Therefore, the events grouped together must be sent to the same Event Hub partition. Since in this example we group by TollBoothID, we should be sure that TollBoothID is used as the partition key when the events are sent to Event Hub. Then in ASA, we can use **PARTITION BY PartitionId** to inherit from this partition scheme and enable full parallelization. Since the output is blob storage, we don't need to worry about configuring a partition key value, as per requirement #4.

Example of scenarios that are *not* embarrassingly parallel

In the previous section, we showed some embarrassingly parallel scenarios. In this section, we discuss scenarios that don't meet all the requirements to be embarrassingly parallel.

Mismatched partition count

- Input: Event hub with 8 partitions
- Output: Event hub with 32 partitions

In this case, it doesn't matter what the query is. If the input partition count doesn't match the output partition count, the topology isn't embarrassingly parallel.⁺ However we can still get some level of parallelization.

Query using non-partitioned output

- Input: Event hub with 8 partitions
- Output: Power BI

Power BI output doesn't currently support partitioning. Therefore, this scenario is not embarrassingly parallel.

Multi-step query with different PARTITION BY values

- Input: Event hub with 8 partitions
- Output: Event hub with 8 partitions

Query:

```
WITH Step1 AS (  
    SELECT COUNT(*) AS Count, TollBoothId, PartitionId  
    FROM Input1 Partition By PartitionId  
    GROUP BY TumblingWindow(minute, 3), TollBoothId, PartitionId  
)  
  
SELECT SUM(Count) AS Count, TollBoothId  
FROM Step1 Partition By TollBoothId  
GROUP BY TumblingWindow(minute, 3), TollBoothId
```

As you can see, the second step uses **TollBoothId** as the partitioning key. This step is not the same as the first step, and it therefore requires us to do a shuffle.

The preceding examples show some Stream Analytics jobs that conform to (or don't) an embarrassingly parallel topology. If they do conform, they have the potential for maximum scale. For jobs that don't fit one of these profiles, scaling guidance will be available in future updates. For now, use the general guidance in the following

sections.

Compatibility level 1.2 - Multi-step query with different PARTITION BY values

- Input: Event hub with 8 partitions
- Output: Event hub with 8 partitions

Query:

```
WITH Step1 AS (
    SELECT COUNT(*) AS Count, TollBoothId
    FROM Input1
    GROUP BY TumblingWindow(minute, 3), TollBoothId
)

SELECT SUM(Count) AS Count, TollBoothId
FROM Step1
GROUP BY TumblingWindow(minute, 3), TollBoothId
```

Compatibility level 1.2 enables parallel query execution by default. For example, query from the previous section will be partitioned as long as "TollBoothId" column is set as input Partition Key. PARTITION BY PartitionId clause is not required.

Calculate the maximum streaming units of a job

The total number of streaming units that can be used by a Stream Analytics job depends on the number of steps in the query defined for the job and the number of partitions for each step.

Steps in a query

A query can have one or many steps. Each step is a subquery defined by the **WITH** keyword. The query that is outside the **WITH** keyword (one query only) is also counted as a step, such as the **SELECT** statement in the following query:

Query:

```
WITH Step1 AS (
    SELECT COUNT(*) AS Count, TollBoothId
    FROM Input1 Partition By PartitionId
    GROUP BY TumblingWindow(minute, 3), TollBoothId, PartitionId
)
SELECT SUM(Count) AS Count, TollBoothId
FROM Step1
GROUP BY TumblingWindow(minute, 3), TollBoothId
```

This query has two steps.

NOTE

This query is discussed in more detail later in the article.

Partition a step

Partitioning a step requires the following conditions:

- The input source must be partitioned.
- The **SELECT** statement of the query must read from a partitioned input source.
- The query within the step must have the **PARTITION BY** keyword.

When a query is partitioned, the input events are processed and aggregated in separate partition groups, and outputs events are generated for each of the groups. If you want a combined aggregate, you must create a second non-partitioned step to aggregate.

Calculate the max streaming units for a job

All non-partitioned steps together can scale up to six streaming units (SUs) for a Stream Analytics job. In addition to this, you can add 6 SUs for each partition in a partitioned step. You can see some **examples** in the table below.

QUERY	MAX SUs FOR THE JOB
<ul style="list-style-type: none">The query contains one step.The step is not partitioned.	6
<ul style="list-style-type: none">The input data stream is partitioned by 16.The query contains one step.The step is partitioned.	96 (6 * 16 partitions)
<ul style="list-style-type: none">The query contains two steps.Neither of the steps is partitioned.	6
<ul style="list-style-type: none">The input data stream is partitioned by 3.The query contains two steps. The input step is partitioned and the second step is not.The SELECT statement reads from the partitioned input.	24 (18 for partitioned steps + 6 for non-partitioned steps)

Examples of scaling

The following query calculates the number of cars within a three-minute window going through a toll station that has three tollbooths. This query can be scaled up to six SUs.

```
SELECT COUNT(*) AS Count, TollBoothId  
FROM Input1  
GROUP BY TumblingWindow(minute, 3), TollBoothId, PartitionId
```

To use more SUs for the query, both the input data stream and the query must be partitioned. Since the data stream partition is set to 3, the following modified query can be scaled up to 18 SUs:

```
SELECT COUNT(*) AS Count, TollBoothId  
FROM Input1 Partition By PartitionId  
GROUP BY TumblingWindow(minute, 3), TollBoothId, PartitionId
```

When a query is partitioned, the input events are processed and aggregated in separate partition groups. Output events are also generated for each of the groups. Partitioning can cause some unexpected results when the **GROUP BY** field is not the partition key in the input data stream. For example, the **TollBoothId** field in the previous query is not the partition key of **Input1**. The result is that the data from TollBooth #1 can be spread in multiple partitions.

Each of the **Input1** partitions will be processed separately by Stream Analytics. As a result, multiple records of the car count for the same tollbooth in the same Tumbling window will be created. If the input partition key can't be changed, this problem can be fixed by adding a non-partition step to aggregate values across partitions, as in the following example:

```
WITH Step1 AS (
    SELECT COUNT(*) AS Count, TollBoothId
    FROM Input1 Partition By PartitionId
    GROUP BY TumblingWindow(minute, 3), TollBoothId, PartitionId
)

SELECT SUM(Count) AS Count, TollBoothId
FROM Step1
GROUP BY TumblingWindow(minute, 3), TollBoothId
```

This query can be scaled to 24 SUs.

NOTE

If you are joining two streams, make sure that the streams are partitioned by the partition key of the column that you use to create the joins. Also make sure that you have the same number of partitions in both streams.

Get help

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Scale an Azure Stream Analytics job to increase throughput

12/17/2018 • 6 minutes to read • [Edit Online](#)

This article shows you how to tune a Stream Analytics query to increase throughput for Streaming Analytics jobs. You can use the following guide to scale your job to handle higher load and take advantage of more system resources (such as more bandwidth, more CPU resources, more memory). As a prerequisite, you may need to read the following articles:

- [Understand and adjust Streaming Units](#)
- [Create parallelizable jobs](#)

Case 1 – Your query is inherently fully parallelizable across input partitions

If your query is inherently fully parallelizable across input partitions, you can follow the following steps:

1. Author your query to be embarrassingly parallel by using **PARTITION BY** keyword. See more details in the Embarrassingly parallel jobs section [on this page](#).
2. Depending on output types used in your query, some output may either be not parallelizable, or need further configuration to be embarrassingly parallel. For example, SQL, SQL DW, and PowerBI outputs are not parallelizable. Outputs are always merged before sending to the output sink. Blobs, Tables, ADLS, Service Bus, and Azure Function are automatically parallelized. CosmosDB and Event Hub needs to have the PartitionKey configuration set to match with the **PARTITION BY** field (usually PartitionId). For Event Hub, also pay extra attention to match the number of partitions for all inputs and all outputs to avoid cross-over between partitions.
3. Run your query with **6 SU** (which is the full capacity of a single computing node) to measure maximum achievable throughput, and if you are using **GROUP BY**, measure how many groups (cardinality) the job can handle. General symptoms of the job hitting system resource limits are the following.
 - SU % utilization metric is over 80%. This indicates memory usage is high. The factors contributing to the increase of this metric are described [here](#).
 - Output timestamp is falling behind with respect to wall clock time. Depending on your query logic, the output timestamp may have a logic offset from the wall clock time. However, they should progress at roughly the same rate. If the output timestamp is falling further and further behind, it's an indicator that the system is overworking. It can be a result of downstream output sink throttling, or high CPU utilization. We don't provide CPU utilization metric at this time, so it can be difficult to differentiate the two.
 - If the issue is due to sink throttling, you may need to increase the number of output partitions (and also input partitions to keep the job fully parallelizable), or increase the amount of resources of the sink (for example number of Request Units for CosmosDB).
 - In job diagram, there is a per partition backlog event metric for each input. If the backlog event metric keeps increasing, it's also an indicator that the system resource is constrained (either because of output sink throttling, or high CPU).
4. Once you have determined the limits of what a 6 SU job can reach, you can extrapolate linearly the processing capacity of the job as you add more SUs, assuming you don't have any data skew that makes certain partition "hot."

NOTE

Choose the right number of Streaming Units: Because Stream Analytics creates a processing node for each 6 SU added, it's best to make the number of nodes a divisor of the number of input partitions, so the partitions can be evenly distributed across the nodes. For example, you have measured your 6 SU job can achieve 4 MB/s processing rate, and your input partition count is 4. You can choose to run your job with 12 SU to achieve roughly 8 MB/s processing rate, or 24 SU to achieve 16 MB/s. You can then decide when to increase SU number for the job to what value, as a function of your input rate.

Case 2 - If your query is not embarrassingly parallel.

If your query is not embarrassingly parallel, you can follow the following steps.

1. Start with a query with no **PARTITION BY** first to avoid partitioning complexity, and run your query with 6 SU to measure maximum load as in [Case 1](#).
2. If you can achieve your anticipated load in term of throughput, you are done. Alternatively, you may choose to measure the same job running at 3 SU and 1 SU, to find out the minimum number of SU that works for your scenario.
3. If you can't achieve the desired throughput, try to break your query into multiple steps if possible, if it doesn't have multiple steps already, and allocate up to 6 SU for each step in the query. For example if you have 3 steps, allocate 18 SU in the "Scale" option.
4. When running such a job, Stream Analytics puts each step on its own node with dedicated 6 SU resources.
5. If you still haven't achieved your load target, you can attempt to use **PARTITION BY** starting from steps closer to the input. For **GROUP BY** operator that may not be naturally partitionable, you can use the local/global aggregate pattern to perform a partitioned **GROUP BY** followed by a non-partitioned **GROUP BY**. For example, if you want to count how many cars going through each toll booth every 3 minutes, and the volume of the data is beyond what can be handled by 6 SU.

Query:

```
WITH Step1 AS (
    SELECT COUNT(*) AS Count, TollBoothId, PartitionId
    FROM Input1 Partition By PartitionId
    GROUP BY TumblingWindow(minute, 3), TollBoothId, PartitionId
)
SELECT SUM(Count) AS Count, TollBoothId
FROM Step1
GROUP BY TumblingWindow(minute, 3), TollBoothId
```

In the query above, you are counting cars per toll booth per partition, and then adding the count from all partitions together.

Once partitioned, for each partition of the step, allocate up to 6 SU, each partition having 6 SU is the maximum, so each partition can be placed on its own processing node.

NOTE

If your query cannot be partitioned, adding additional SU in a multi-steps query may not always improve throughput. One way to gain performance is to reduce volume on the initial steps using local/global aggregate pattern, as described above in step 5.

Case 3 - You are running lots of independent queries in a job.

For certain ISV use cases, where it's more cost-efficient to process data from multiple tenants in a single job, using separate inputs and outputs for each tenant, you may end up running quite a few (for example 20) independent queries in a single job. The assumption is each such subquery's load is relatively small. In this case, you can follow the following steps.

1. In this case, do not use **PARTITION BY** in the query
2. Reduce the input partition count to the lowest possible value of 2 if you are using Event Hub.
3. Run the query with 6 SU. With expected load for each subquery, add as many such subqueries as possible, until the job is hitting system resource limits. Refer to [Case 1](#) for the symptoms when this happens.
4. Once you are hitting the subquery limit measured above, start adding the subquery to a new job. The number of jobs to run as a function of the number of independent queries should be fairly linear, assuming you don't have any load skew. You can then forecast how many 6 SU jobs you need to run as a function of the number of tenants you would like to serve.
5. When using reference data join with such queries, union the inputs together before joining with the same reference data. Then, split out the events if necessary. Otherwise, each reference data join keeps a copy of reference data in memory, likely blowing up the memory usage unnecessarily.

NOTE

How many tenants to put in each job? This query pattern often has a large number of subqueries, and results in very large and complex topology. The controller of the job may not be able to handle such a large topology. As a rule of thumb, stay under 40 tenants for 1 SU job, and 60 tenants for 3 SU and 6 SU jobs. When you are exceeding the capacity of the controller, the job will not start successfully.

Get help

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Scale your Stream Analytics job with Azure Machine Learning Studio functions

6/24/2019 • 7 minutes to read • [Edit Online](#)

This article discusses how to efficiently scale Azure Stream Analytics jobs that use Azure Machine Learning functions. For information on how to scale Stream Analytics jobs in general see the article [Scaling jobs](#).

What is an Azure Machine Learning function in Stream Analytics?

A Machine Learning function in Stream Analytics can be used like a regular function call in the Stream Analytics query language. Behind the scenes, however, these function calls are actually Azure Machine Learning Web Service requests.

You can improve the throughput of Machine Learning web service requests by "batching" multiple rows together in the same web service API call. This grouping is called a mini-batch. For more information, see [Azure Machine Learning Studio Web Services](#). Support for Azure Machine Learning Studio in Stream Analytics is in preview.

Configure a Stream Analytics job with Machine Learning functions

There are two parameters to configure the Machine Learning function used by your Stream Analytics job:

- Batch size of the Machine Learning function calls.
- The number of Streaming Units (SUs) provisioned for the Stream Analytics job.

To determine the appropriate values for SUs, decide whether you would like to optimize latency of the Stream Analytics job or the throughput of each SU. SUs may always be added to a job to increase the throughput of a well-partitioned Stream Analytics query. Additional SUs do increase the cost of running the job.

Determine the latency *tolerance* for your Stream Analytics job. Increasing the batch size will increase the latency of your Azure Machine Learning service requests and the latency of the Stream Analytics job.

Increasing the batch size allows the Stream Analytics job to process **more events** with the **same number** of Machine Learning web service requests. The increase of Machine Learning web service latency is usually sublinear to the increase of batch size.

It's important to consider the most cost-efficient batch size for a Machine Learning web service in any given situation. The default batch size for web service requests is 1000. You can change this default size using the [Stream Analytics REST API](#) or the [PowerShell client for Stream Analytics](#).

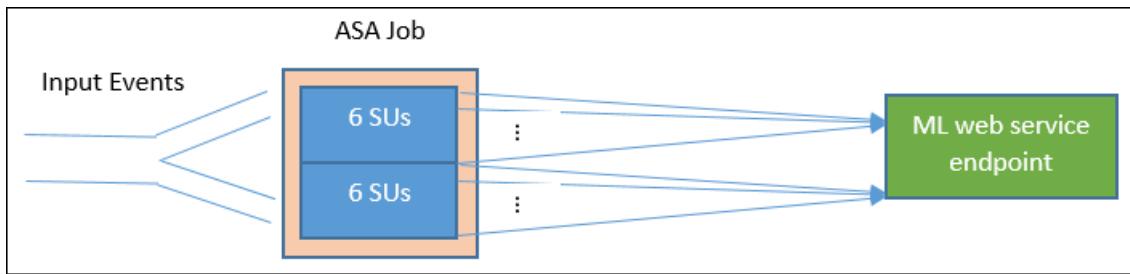
Once you've decided on a batch size, you can set the number of streaming units (SUs), based on the number of events that the function needs to process per second. For more information about streaming units, see [Stream Analytics scale jobs](#).

Every 6 SUs get 20 concurrent connections to the Machine Learning web service. However, 1 SU job and 3 SU jobs get 20 concurrent connections.

If your application generates 200,000 events per second, and the batch size is 1000, then the resulting web service latency is 200 ms. This rate means that every connection can make five requests to the Machine Learning web service each second. With 20 connections, the Stream Analytics job can process 20,000 events in 200 ms and 100,000 events in a second.

To process 200,000 events per second, the Stream Analytics job needs 40 concurrent connections, which come out

to 12 SUs. The following diagram illustrates the requests from the Stream Analytics job to the Machine Learning web service endpoint – Every 6 SUs has 20 concurrent connections to Machine Learning web service at max.



In general, B for batch size, L for the web service latency at batch size B in milliseconds, the throughput of a Stream Analytics job with N SUs is:

$$20 * \text{ceil}(N/6) * B * (1000/L)$$

You can also configure the 'max concurrent calls' on the Machine Learning web service. It's recommended to set this parameter to the maximum value (200 currently).

For more information on this setting, review the [Scaling article for Machine Learning Web Services](#).

Example – Sentiment Analysis

The following example includes a Stream Analytics job with the sentiment analysis Machine Learning function, as described in the [Stream Analytics Machine Learning integration tutorial](#).

The query is a simple fully partitioned query followed by the **sentiment** function, as shown in the following example:

```
WITH subquery AS (
    SELECT text, sentiment(text) as result from input
)
Select text, result.[Score]
Into output
From subquery
```

Let's examine the configuration necessary to create a Stream Analytics job, which does sentiment analysis of tweets at a rate of 10,000 tweets per second.

Using 1 SU, could this Stream Analytics job handle the traffic? The job can keep up with the input using the default batch size of 1000. The default latency of the sentiment analysis Machine Learning web service (with a default batch size of 1000) creates no more than a second of latency.

The Stream Analytics job's **overall** or end-to-end latency would typically be a few seconds. Take a more detailed look into this Stream Analytics job, *especially* the Machine Learning function calls. With a batch size of 1000, a throughput of 10,000 events takes about 10 requests to the web service. Even with one SU, there are enough concurrent connections to accommodate this input traffic.

If the input event rate increases by 100x, then the Stream Analytics job needs to process 1,000,000 tweets per second. There are two options to accomplish the increased scale:

1. Increase the batch size.
2. Partition the input stream to process the events in parallel.

With the first option, the job **latency** increases.

With the second option, you will have to provision more SUs to have more concurrent Machine Learning web service requests. This greater number of SUs, increases the job **cost**.

Let's look at the scaling using the following latency measurements for each batch size:

LATENCY	BATCH SIZE
200 ms	1000-event batches or below
250 ms	5,000-event batches
300 ms	10,000-event batches
500 ms	25,000-event batches

1. Using the first option (**not** provisioning more SUs). The batch size could be increased to **25,000**. Increasing the batch size in this way will allow the job to process 1,000,000 events with 20 concurrent connections to the Machine Learning web service (with a latency of 500 ms per call). So the additional latency of the Stream Analytics job due to the sentiment function requests against the Machine Learning web service requests would be increased from **200 ms to 500 ms**. However, batch size **can't** be increased infinitely as the Machine Learning web services requires the payload size of a request be 4 MB or smaller, and web service requests timeout after 100 seconds of operation.
2. Using the second option, the batch size is left at 1000, with 200-ms web service latency, every 20 concurrent connections to the web service would be able to process $1000 * 20 * 5$ events = 100,000 per second. So to process 1,000,000 events per second, the job would need 60 SUs. Compared to the first option, Stream Analytics job would make more web service batch requests, in turn generating an increased cost.

Below is a table for the throughput of the Stream Analytics job for different SUs and batch sizes (in number of events per second).

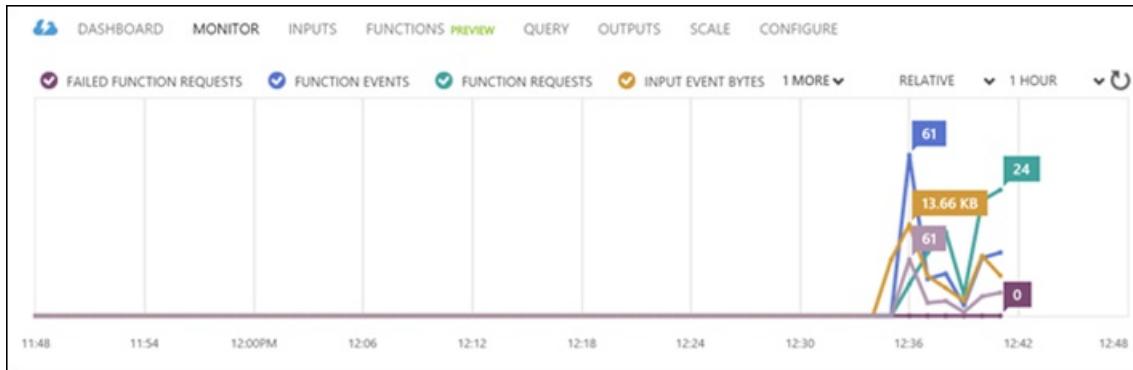
BATCH SIZE (ML LATENCY)	500 (200 MS)	1,000 (200 MS)	5,000 (250 MS)	10,000 (300 MS)	25,000 (500 MS)
1 SU	2,500	5,000	20,000	30,000	50,000
3 SUs	2,500	5,000	20,000	30,000	50,000
6 SUs	2,500	5,000	20,000	30,000	50,000
12 SUs	5,000	10,000	40,000	60,000	100,000
18 SUs	7,500	15,000	60,000	90,000	150,000
24 SUs	10,000	20,000	80,000	120,000	200,000
...
60 SUs	25,000	50,000	200,000	300,000	500,000

By now, you should already have a good understanding of how Machine Learning functions in Stream Analytics work. You likely also understand that Stream Analytics jobs "pull" data from data sources and each "pull" returns a batch of events for the Stream Analytics job to process. How does this pull model impact the Machine Learning web service requests?

Normally, the batch size we set for Machine Learning functions won't exactly be divisible by the number of events returned by each Stream Analytics job "pull". When this occurs, the Machine Learning web service is called with "partial" batches. Using partial batches avoids incurring additional job latency overhead in coalescing events from pull to pull.

New function-related monitoring metrics

In the Monitor area of a Stream Analytics job, three additional function-related metrics have been added. They are **FUNCTION REQUESTS**, **FUNCTION EVENTS** and **FAILED FUNCTION REQUESTS**, as shown in the graphic below.



The are defined as follows:

FUNCTION REQUESTS: The number of function requests.

FUNCTION EVENTS: The number events in the function requests.

FAILED FUNCTION REQUESTS: The number of failed function requests.

Key Takeaways

To scale a Stream Analytics job with Machine Learning functions, consider the following factors:

1. The input event rate.
2. The tolerated latency for the running Stream Analytics job (and thus the batch size of the Machine Learning web service requests).
3. The provisioned Stream Analytics SUs and the number of Machine Learning web service requests (the additional function-related costs).

A fully partitioned Stream Analytics query was used as an example. If a more complex query is needed, the [Azure Stream Analytics forum](#) is a great resource for getting additional help from the Stream Analytics team.

Next steps

To learn more about Stream Analytics, see:

- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Management .NET SDK: Set up and run analytics jobs using the Azure Stream Analytics API for .NET

5/29/2019 • 7 minutes to read • [Edit Online](#)

Learn how to set up and run analytics jobs using the Stream Analytics API for .NET using the Management .NET SDK. Set up a project, create input and output sources, transformations, and start and stop jobs. For your analytics jobs, you can stream data from Blob storage or from an event hub.

See the [management reference documentation for the Stream Analytics API for .NET](#).

Azure Stream Analytics is a fully managed service providing low-latency, highly available, scalable, complex event processing over streaming data in the cloud. Stream Analytics enables customers to set up streaming jobs to analyze data streams, and allows them to drive near real-time analytics.

NOTE

We have updated the sample code in this article with Azure Stream Analytics Management .NET SDK v2.x version. For sample code using the uses legacy (1.x) SDK version, please see [Use the Management .NET SDK v1.x for Stream Analytics](#).

Prerequisites

Before you begin this article, you must have the following requirements:

- Install Visual Studio 2019 or 2015.
- Download and install [Azure .NET SDK](#).
- Create an Azure Resource Group in your subscription. The following example is a sample Azure PowerShell script. For Azure PowerShell information, see [Install and configure Azure PowerShell](#);

```
# Log in to your Azure account
Add-AzureAccount

# Select the Azure subscription you want to use to create the resource group
Select-AzureSubscription -SubscriptionName <subscription name>

# If Stream Analytics has not been registered to the subscription, remove the remark      symbol (#) to
run the Register-AzProvider cmdlet to register the provider namespace
#Register-AzProvider -Force -ProviderNamespace 'Microsoft.StreamAnalytics'

# Create an Azure resource group
New-AzureResourceGroup -Name <YOUR RESOURCE GROUP NAME> -Location <LOCATION>
```

- Set up an input source and output target for the job to connect to.

Set up a project

To create an analytics job, use the Stream Analytics API for .NET, first set up your project.

1. Create a Visual Studio C# .NET console application.
2. In the Package Manager Console, run the following commands to install the NuGet packages. The first one is the Azure Stream Analytics Management .NET SDK. The second one is for Azure client authentication.

```
Install-Package Microsoft.Azure.Management.StreamAnalytics -Version 2.0.0
Install-Package Microsoft.Rest.ClientRuntime.Azure.Authentication -Version 2.3.1
```

3. Add the following **appSettings** section to the App.config file:

```
<appSettings>
    <add key="ClientId" value="1950a258-227b-4e31-a9cf-717495945fc2" />
    <add key="RedirectUri" value="urn:ietf:wg:oauth:2.0:oob" />
    <add key="SubscriptionId" value="YOUR SUBSCRIPTION ID" />
    <add key="ActiveDirectoryTenantId" value="YOUR TENANT ID" />
</appSettings>
```

Replace values for **SubscriptionId** and **ActiveDirectoryTenantId** with your Azure subscription and tenant IDs. You can get these values by running the following Azure PowerShell cmdlet:

```
Get-AzureAccount
```

4. Add the following reference in your .csproj file:

```
<Reference Include="System.Configuration" />
```

5. Add the following **using** statements to the source file (Program.cs) in the project:

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.Threading;
using System.Threading.Tasks;

using Microsoft.Azure.Management.StreamAnalytics;
using Microsoft.Azure.Management.StreamAnalytics.Models;
using Microsoft.Rest.Azure.Authentication;
using Microsoft.Rest;
```

6. Add an authentication helper method:

```
private static async Task<ServiceClientCredentials> GetCredentials()
{
    var activeDirectoryClientSettings =
        ActiveDirectoryClientSettings.UsePromptOnly(ConfigurationManager.AppSettings["ClientId"], new
        Uri("urn:ietf:wg:oauth:2.0:oob"));
    ServiceClientCredentials credentials = await
        UserTokenProvider.LoginWithPromptAsync(ConfigurationManager.AppSettings["ActiveDirectoryTenantId"],
        activeDirectoryClientSettings);

    return credentials;
}
```

Create a Stream Analytics management client

A **StreamAnalyticsManagementClient** object allows you to manage the job and the job components, such as input, output, and transformation.

Add the following code to the beginning of the **Main** method:

```

string resourceGroupName = "<YOUR AZURE RESOURCE GROUP NAME>";
string streamingJobName = "<YOUR STREAMING JOB NAME>";
string inputName = "<YOUR JOB INPUT NAME>";
string transformationName = "<YOUR JOB TRANSFORMATION NAME>";
string outputName = "<YOUR JOB OUTPUT NAME>";

SynchronizationContext.SetSynchronizationContext(new SynchronizationContext());

// Get credentials
ServiceClientCredentials credentials = GetCredentials().Result;

// Create Stream Analytics management client
StreamAnalyticsManagementClient streamAnalyticsManagementClient = new
StreamAnalyticsManagementClient(credentials)
{
    SubscriptionId = ConfigurationManager.AppSettings["SubscriptionId"]
};

```

The **resourceGroupName** variable's value should be the same as the name of the resource group you created or picked in the prerequisite steps.

To automate the credential presentation aspect of job creation, refer to [Authenticating a service principal with Azure Resource Manager](#).

The remaining sections of this article assume that this code is at the beginning of the **Main** method.

Create a Stream Analytics job

The following code creates a Stream Analytics job under the resource group that you have defined. You will add an input, output, and transformation to the job later.

```

// Create a streaming job
StreamingJob streamingJob = new StreamingJob()
{
    Tags = new Dictionary<string, string>()
    {
        { "Origin", ".NET SDK" },
        { "ReasonCreated", "Getting started tutorial" }
    },
    Location = "West US",
    EventsOutOfOrderPolicy = EventsOutOfOrderPolicy.Drop,
    EventsOutOfOrderMaxDelayInSeconds = 5,
    EventsLateArrivalMaxDelayInSeconds = 16,
    OutputErrorPolicy = OutputErrorPolicy.Drop,
    DataLocale = "en-US",
    CompatibilityLevel = CompatibilityLevel.OneFullStopZero,
    Sku = new Sku()
    {
        Name = SkuName.Standard
    }
};
StreamingJob createStreamingJobResult =
streamAnalyticsManagementClient.StreamingJobs.CreateOrReplace(streamingJob, resourceGroupName,
streamingJobName);

```

Create a Stream Analytics input source

The following code creates a Stream Analytics input source with the blob input source type and CSV serialization.

To create an event hub input source, use **EventHubStreamInputDataSource** instead of

BlobStreamInputDataSource. Similarly, you can customize the serialization type of the input source.

```

// Create an input
StorageAccount storageAccount = new StorageAccount()
{
    AccountName = "<YOUR STORAGE ACCOUNT NAME>",
    AccountKey = "<YOUR STORAGE ACCOUNT KEY>"
};

Input input = new Input()
{
    Properties = new StreamInputProperties()
    {
        Serialization = new CsvSerialization()
        {
            FieldDelimiter = ",",
            Encoding = Encoding.UTF8
        },
        Datasource = new BlobStreamInputDataSource()
        {
            StorageAccounts = new[] { storageAccount },
            Container = "<YOUR STORAGE BLOB CONTAINER>",
            PathPattern = "{date}/{time}",
            DateFormat = "yyyy/MM/dd",
            TimeFormat = "HH",
            SourcePartitionCount = 16
        }
    }
};

Input createInputResult = streamAnalyticsManagementClient.Inputs.CreateOrReplace(input, resourceName,
streamingJobName, inputName);

```

Input sources, whether from Blob storage or an event hub, are tied to a specific job. To use the same input source for different jobs, you must call the method again and specify a different job name.

Test a Stream Analytics input source

The **TestConnection** method tests whether the Stream Analytics job is able to connect to the input source as well as other aspects specific to the input source type. For example, in the blob input source you created in an earlier step, the method will check that the Storage account name and key pair can be used to connect to the Storage account as well as check that the specified container exists.

```

// Test the connection to the input
ResourceTestStatus testInputResult = streamAnalyticsManagementClient.Inputs.Test(resourceName,
streamingJobName, inputName);

```

Create a Stream Analytics output target

Creating an output target is similar to creating a Stream Analytics input source. Like input sources, output targets are tied to a specific job. To use the same output target for different jobs, you must call the method again and specify a different job name.

The following code creates an output target (Azure SQL database). You can customize the output target's data type and/or serialization type.

```

// Create an output
Output output = new Output()
{
    Datasource = new AzureSqlDatabaseOutputDataSource()
    {
        Server = "<YOUR DATABASE SERVER NAME>",
        Database = "<YOUR DATABASE NAME>",
        User = "<YOUR DATABASE LOGIN>",
        Password = "<YOUR DATABASE LOGIN PASSWORD>",
        Table = "<YOUR DATABASE TABLE NAME>"
    }
};

Output createOutputResult = streamAnalyticsManagementClient.Outputs.CreateOrReplace(output, resourceGroupName, streamingJobName, outputName);

```

Test a Stream Analytics output target

A Stream Analytics output target also has the **TestConnection** method for testing connections.

```

// Test the connection to the output
ResourceTestStatus testOutputResult = streamAnalyticsManagementClient.Outputs.Test(resourceGroupName, streamingJobName, outputName);

```

Create a Stream Analytics transformation

The following code creates a Stream Analytics transformation with the query "select * from Input" and specifies to allocate one streaming unit for the Stream Analytics job. For more information on adjusting streaming units, see [Scale Azure Stream Analytics jobs](#).

```

// Create a transformation
Transformation transformation = new Transformation()
{
    Query = "Select Id, Name from <your input name>", // '<your input name>' should be replaced with the value you put for the 'inputName' variable above or in a previous step
    StreamingUnits = 1
};
Transformation createTransformationResult =
streamAnalyticsManagementClient.Transformations.CreateOrReplace(transformation, resourceGroupName, streamingJobName, transformationName);

```

Like input and output, a transformation is also tied to the specific Stream Analytics job it was created under.

Start a Stream Analytics job

After creating a Stream Analytics job and its input(s), output(s), and transformation, you can start the job by calling the **Start** method.

The following sample code starts a Stream Analytics job with a custom output start time set to December 12, 2012, 12:12:12 UTC:

```
// Start a streaming job
StartStreamingJobParameters startStreamingJobParameters = new StartStreamingJobParameters()
{
    OutputStartMode = OutputStartMode.CustomTime,
    OutputStartTime = new DateTime(2012, 12, 12, 12, 12, 12, DateTimeKind.Utc)
};
streamAnalyticsManagementClient.StreamingJobs.Start(resourceGroupName, streamingJobName,
startStreamingJobParameters);
```

Stop a Stream Analytics job

You can stop a running Stream Analytics job by calling the **Stop** method.

```
// Stop a streaming job
streamAnalyticsManagementClient.StreamingJobs.Stop(resourceGroupName, streamingJobName);
```

Delete a Stream Analytics job

The **Delete** method will delete the job as well as the underlying sub-resources, including input(s), output(s), and transformation of the job.

```
// Delete a streaming job
streamAnalyticsManagementClient.StreamingJobs.Delete(resourceGroupName, streamingJobName);
```

Get support

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

You've learned the basics of using a .NET SDK to create and run analytics jobs. To learn more, see the following articles:

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Management .NET SDK](#).
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Implement CI/CD for Stream Analytics on IoT Edge using APIs

2/14/2019 • 3 minutes to read • [Edit Online](#)

You can enable continuous integration and deployment for Azure Stream Analytics jobs using REST APIs. This article provides examples on which APIs to use and how to use them. REST APIs aren't supported on Azure Cloud Shell.

Call APIs from different environments

REST APIs can be called from both Linux and Windows. The following commands demonstrate proper syntax for the API call. Specific API usage will be outlined in later sections of this article.

Linux

For Linux, you can use `Curl` or `Wget` commands:

```
curl -u { <username:password> } -H "Content-Type: application/json" -X { <method> } -d "{ <request body>}" { <url> }
```

```
wget -q -O- --{ <method> }-data=<request body>--header=Content-Type:application/json --auth-no-challenge --http-user=<Admin> --http-password=<password> <url>
```

Windows

For Windows, use Powershell:

```
$user = "<username>"  
$pass = "<password>"  
$encodedCreds = [Convert]::ToBase64String([Text.Encoding]::ASCII.GetBytes(("{}:{}" -f $user,$pass)))  
$basicAuthValue = "Basic $encodedCreds"  
$headers = New-Object "System.Collections.Generic.Dictionary[[String],[String]]"  
$headers.Add("Content-Type", 'application/json')  
$headers.Add("Authorization", $basicAuthValue)  
$content = "<request body>"  
$response = Invoke-RestMethod <url>-Method <method> -Body $content -Headers $Headers  
echo $response
```

Create an ASA job on Edge

To create Stream Analytics job, call the PUT method using the Stream Analytics API.

METHOD	REQUEST URL
PUT	<code>https://management.azure.com/subscriptions/{subscription-id}/resourcegroups/{resource-group-name}/providers/Microsoft.StreamAnalytics/streamingjobs/{job-name}?api-version=2017-04-01-preview</code>

Example of command using `curl`:

```
curl -u { <username:password> } -H "Content-Type: application/json" -X { <method> } -d "{ <request body>}"
https://management.azure.com/subscriptions/{subscription-id}/resourcegroups/{resource-group-
name}/providers/Microsoft.StreamAnalytics/streamingjobs/{jobname}?api-version=2017-04-01-preview
```

Example of request body in JSON:

```
{
  "location": "West US",
  "tags": { "key": "value", "ms-suppressjobstatusmetrics": "true" },
  "sku": {
    "name": "Standard"
  },
  "properties": {
    "sku": {
      "name": "standard"
    },
    "eventsLateArrivalMaxDelayInSeconds": 1,
    "jobType": "edge",
    "inputs": [
      {
        "name": "{inputname}",
        "properties": {
          "type": "stream",
          "serialization": {
            "type": "JSON",
            "properties": {
              "fieldDelimiter": ",",
              "encoding": "UTF8"
            }
          },
          "datasource": {
            "type": "GatewayMessageBus",
            "properties": {
            }
          }
        }
      }
    ],
    "transformation": {
      "name": "{queryName}",
      "properties": {
        "query": "{query}"
      },
      "package": {
        "storageAccount" : {
          "accountName": "{blobstorageaccountname}",
          "accountKey": "{blobstorageaccountkey}"
        },
        "container": "{blobcontainer}"
      },
      "outputs": [
        {
          "name": "{outputname}",
          "properties": {
            "serialization": {
              "type": "JSON",
              "properties": {
                "fieldDelimiter": ",",
                "encoding": "UTF8"
              }
            },
            "datasource": {
              "type": "GatewayMessageBus",
              "properties": {
              }
            }
          }
        }
      ]
    }
  }
}
```

For more information, see the [API documentation](#).

Publish Edge package

To publish a Stream Analytics job on IoT Edge, call the POST method using the Edge Package Publish API.

METHOD	REQUEST URL
POST	<code>https://management.azure.com/subscriptions/{subscriptionid} }/resourceGroups/{resourcegroupname}/providers/Microsoft .StreamAnalytics/streamingjobs/{jobname}/publishededgepackage ?api-version=2017-04-01-preview</code>

This asynchronous operation returns a status of 202 until the job has been successfully published. The location response header contains the URI used to get the status of the process. While the process is running, a call to the URI in the location header returns a status of 202. When the process finishes, the URI in the location header returns a status of 200.

Example of an Edge package publish call using **curl**:

```
curl -d -X POST  
https://management.azure.com/subscriptions/{subscriptionid}/resourceGroups/{resourcegroupname}/providers/Micro  
soft.StreamAnalytics/streamingjobs/{jobname}/publishededgepackage?api-version=2017-04-01-preview
```

After making the POST call, you should expect a response with an empty body. Look for the URL located in the HEAD of the response and record it for further use.

Example of the URL from the HEAD of response:

```
https://management.azure.com/subscriptions/{**subscriptionid**}/resourcegroups/{**resourcegroupname**}/provide  
rs/Microsoft.StreamAnalytics/StreamingJobs/{**resourcename**}/OperationResults/023a4d68-ffaf-4e16-8414-  
cb6f2e14fe23?api-version=2017-04-01-preview
```

A Wait for one to two minutes before running the following command to make an API call with the URL you found in the HEAD of the response. Retry the command if you do not get a 200 response.

Example of making API call with returned URL with **curl**:

```
curl -d -X GET  
https://management.azure.com/subscriptions/{subscriptionid}/resourceGroups/{resourcegroupname}/providers/Micro  
soft.StreamAnalytics/streamingjobs/{resourcename}/publishededgepackage?api-version=2017-04-01-preview
```

The response includes the information you need to add to the Edge deployment script. The examples below show what information you need to collect and where to add it in the deployment manifest.

Sample response body after publishing successfully:

```
{  
    edgePackageUrl : null  
    error : null  
    manifest : "{\"supportedPlatforms\": [{\"os\":\"linux\", \"arch\":\"amd64\", \"features\":[]}],  
    {\"os\":\"linux\", \"arch\":\"arm\", \"features\":[]}, {\"os\":\"windows\", \"arch\":\"amd64\", \"features\":  
    []}], \"schemaVersion\": \"2\", \"name\": \"{jobname}\", \"version\": \"1.0.0.0\", \"type\": \"docker\", \"settings\": {\"image\": \"  
    {imageurl}\", \"createOptions\": null}, \"endpoints\": {\"inputs\": [], \"outputs\": [{\"outputnames\"}]}, \"twin\":  
    {\"contentType\": \"assignments\", \"content\": {\"properties.desired\": {\"ASAJobInfo\": \"  
        {asajobsasurl}\", \"ASAJobResourceId\": \"{asajobresourceid}\", \"ASAJobEtag\": \"  
        {etag}\", \"PublishTimeStamp\": \"{publishtimestamp}\"}}}}}}\"  
    status : \"Succeeded\"  
}
```

Sample of Deployment Manifest:

```
{
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "runtime": {
          "type": "docker",
          "settings": {
            "minDockerVersion": "v1.25",
            "loggingOptions": "",
            "registryCredentials": {}
          }
        },
        "systemModules": {
          "edgeAgent": {
            "type": "docker",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-agent:1.0",
              "createOptions": "{}"
            }
          },
          "edgeHub": {
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-hub:1.0",
              "createOptions": "{}"
            }
          }
        }
      },
      "modules": {
        "<asajobname>": {
          "version": "1.0",
          "type": "docker",
          "status": "running",
          "restartPolicy": "always",
          "settings": {
            "image": "<settings.image>",
            "createOptions": "<settings.createOptions>"
          }
        },
        "version": "<version>",
        "env": {
          "PlanId": {
            "value": "stream-analytics-on-iot-edge"
          }
        }
      }
    },
    "$edgeHub": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "routes": {
          "route": "FROM /* INTO $upstream"
        },
        "storeAndForwardConfiguration": {
          "timeToLiveSecs": 7200
        }
      },
      "<asajobname>": {
        "properties.desired": {<twin.content.properties.desired>}
      }
    }
  }
}
```

After the configuration of the deployment manifest, refer to [Deploy Azure IoT Edge modules with Azure CLI](#) for deployment.

Next steps

- [Azure Stream Analytics on IoT Edge](#)
- [ASA on IoT Edge tutorial](#)
- [Develop Stream Analytics Edge jobs using Visual Studio tools](#)

Install Azure Stream Analytics tools for Visual Studio

2/18/2019 • 2 minutes to read • [Edit Online](#)

Azure Stream Analytics tools support Visual Studio 2017, 2015, and 2013. This article describes how to install and uninstall the tools.

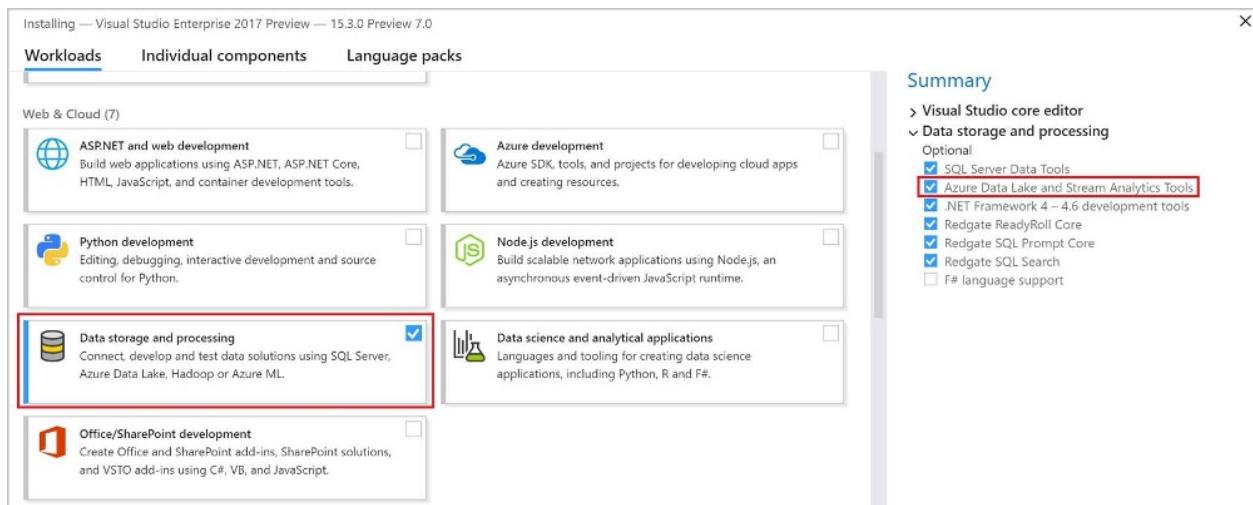
For more information on using the tools, see [Stream Analytics tools for Visual Studio](#).

Install

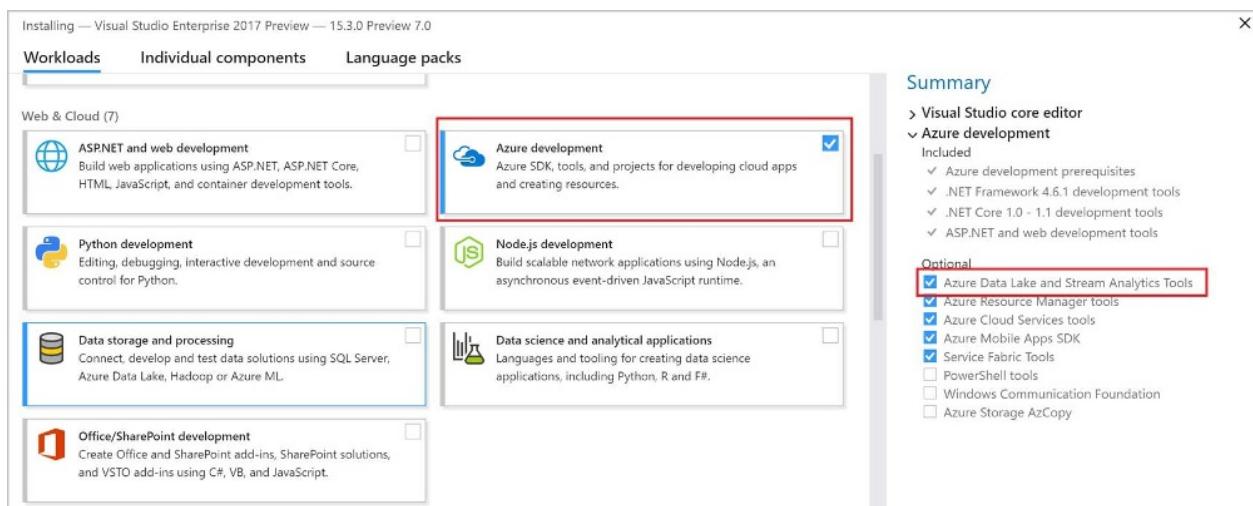
Recommended: Visual Studio 2019 and 2017

- Download [Visual Studio 2019 \(Preview 2 or above\)](#) and [Visual Studio 2017 \(15.3 or above\)](#). Enterprise (Ultimate/Premium), Professional, and Community editions are supported. Express edition is not supported. Visual Studio 2017 on Mac is not supported.
- Stream Analytics tools are part of the **Azure development** and **Data storage and processing** workloads in Visual Studio 2017. Enable either one of these two workloads as part of your Visual Studio installation.

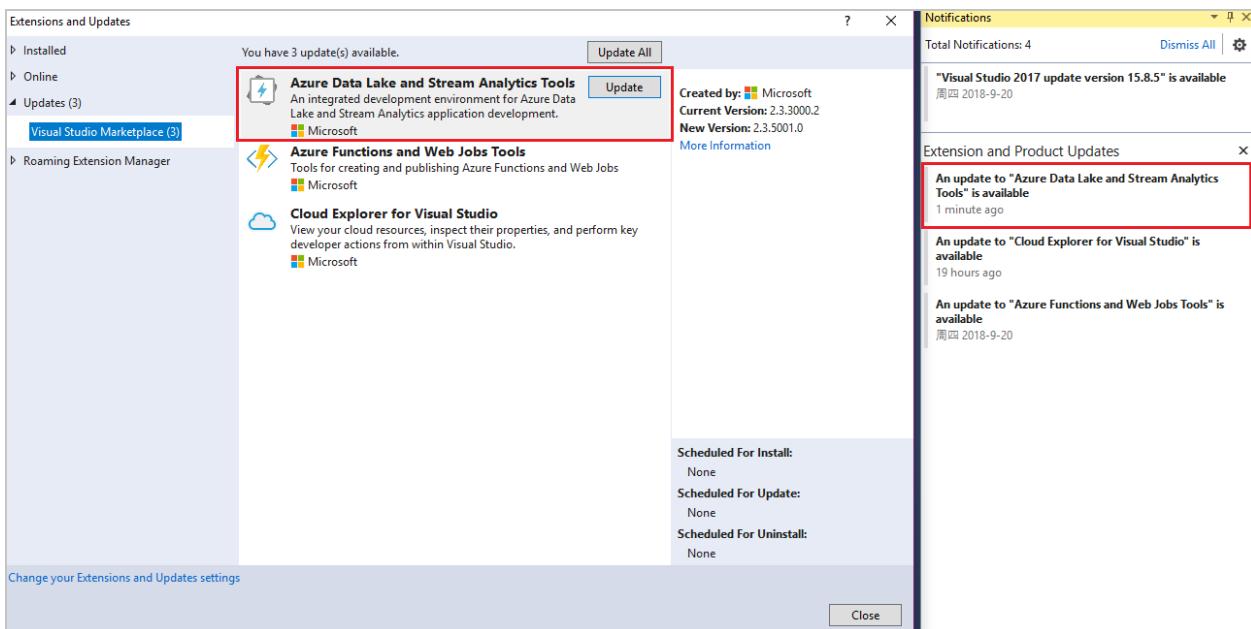
Enable the **Data storage and processing** workload as shown:



Enable the **Azure development** workload as shown:



- In the Tools menu, choose **Extensions and Updates**. Find Azure Data Lake and Stream Analytics tools in the installed extensions and click **Update** to install the latest extension.



Visual Studio 2015, 2013

- Install Visual Studio 2015 or Visual Studio 2013 Update 4. Enterprise (Ultimate/Premium), Professional, and Community editions are supported. Express edition is not supported.
- Install the Microsoft Azure SDK for .NET version 2.7.1 or above by using the [Web platform installer](#).
- Install [Azure Stream Analytics tools for Visual Studio](#).

Update

Visual Studio 2019 and 2017

The new version reminder shows up in the Visual Studio notification.



Visual Studio 2015 and 2013

The installed Stream Analytics tools for Visual Studio check for new versions automatically. Follow the instructions in the pop-up window to install the latest version.

Uninstall

Visual Studio 2019 and 2017

Double-click the Visual Studio installer, and select **Modify**. Clear the **Azure Data Lake and Stream Analytics Tools** check box from either the **Data storage and processing** workload or the **Azure development** workload.

Visual Studio 2015 and 2013

Go to Control Panel, and uninstall **Microsoft Azure Data Lake and Stream Analytics tools for Visual Studio**.

Test Stream Analytics queries locally with Visual Studio

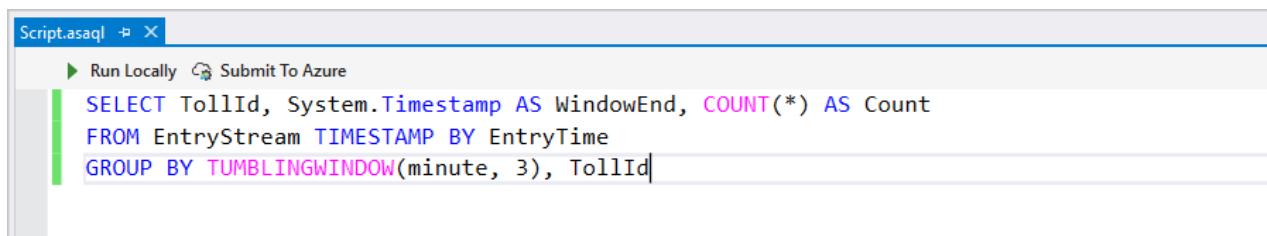
9/10/2018 • 2 minutes to read • [Edit Online](#)

You can use Azure Stream Analytics tools for Visual Studio to test your Stream Analytics jobs locally with sample data.

Use this [Quickstart](#) to learn how to create a Stream Analytics job using Visual Studio.

Test your query

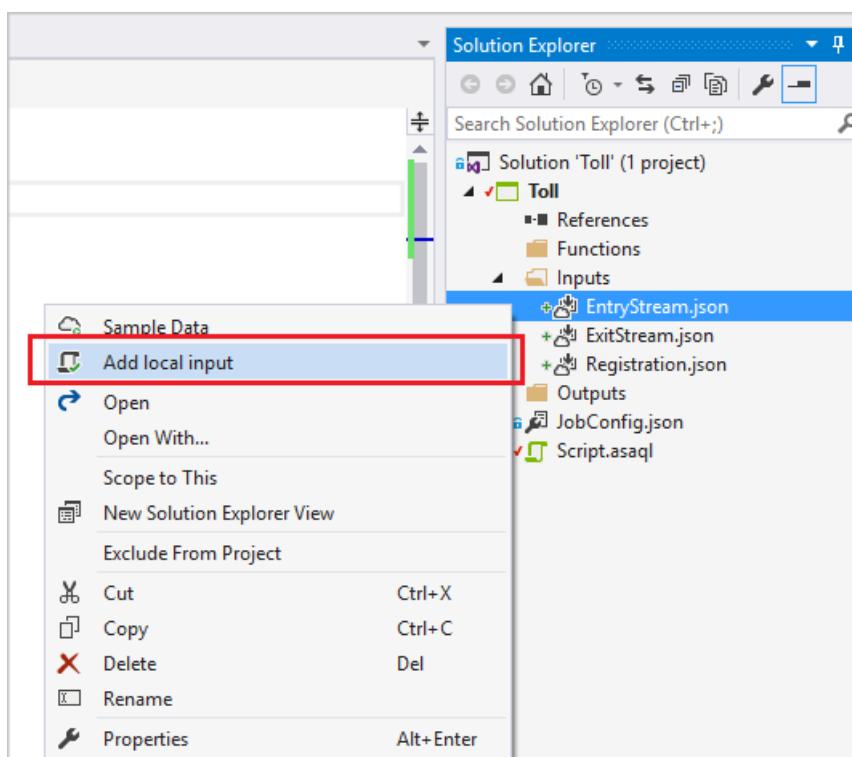
In your Azure Stream Analytics project, double-click **Script.asaql** to open the script in the editor. You can compile the query to see if there are any syntax errors. The query editor supports IntelliSense, syntax coloring, and an error marker.



```
Script.asaql + X
▶ Run Locally  Submit To Azure
SELECT TollId, System.Timestamp AS WindowEnd, COUNT(*) AS Count
FROM EntryStream TIMESTAMP BY EntryTime
GROUP BY TUMBLINGWINDOW(minute, 3), TollId
```

Add local input

To validate your query against local static data, right-click the input and select **Add local input**.



In the pop-up window, select sample data from your local path and **Save**.

Add local input X

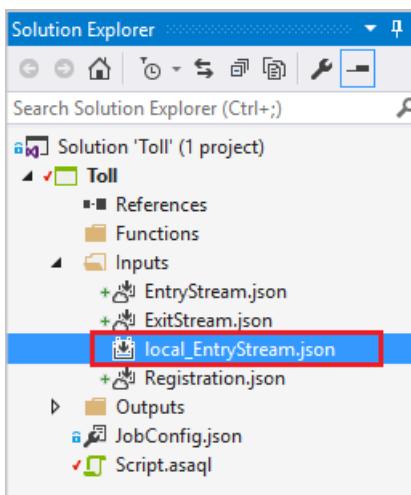
Stream Analytics Local Input Configuration

Please configure the Local Input properties for your Stream Analytics job

* Input Alias	<input type="text" value="EntryStream"/>
* Source Type	<input type="text" value="Data Stream"/>
* Event Serialization Format ?	<input type="text" value="Json"/>
* Local Input File Path	<input type="text" value="D:\Projects\TollApp\Data\Entry.json"/> Browse

Save Cancel

A file named **local_EntryStream.json** is added automatically to your inputs folder.



Select **Run Locally** in the query editor. Or you can press F5.

Script.asaql X

▶ Run Locally Submit To Azure

```
SELECT TollId, System.Timestamp AS WindowEnd, COUNT(*) AS Count
FROM EntryStream TIMESTAMP BY EntryTime
GROUP BY TUMBLINGWINDOW(minute, 3), TollId
```

The output can be viewed in a table format directly from Visual Studio.

The screenshot shows the Stream Analytics Local Run Results interface. At the top, there are two buttons: 'Run Locally' and 'Submit to Azure'. Below them is a code editor window containing the following T-SQL query:

```
select EntryStream.EntryTime, EntryStream.LicensePlate, EntryStream.TollId
FROM EntryStream
```

Below the code editor is a status bar showing '100 %' and '0 changes | 0 authors, 0 changes'. The main area displays the 'Stream Analytics Local Run Results' with a table titled 'output'. The table has three columns: 'entrytime', 'licenseplate', and 'tollid'. The data is as follows:

entrytime	licenseplate	tollid
2014-09-10T20:01:00.0000000	JNB 7001	1
2014-09-10T20:02:00.0000000	YXZ 1001	1
2014-09-10T20:02:00.0000000	ABC 1004	3
2014-09-10T20:03:00.0000000	XYZ 1003	2
2014-09-10T20:03:00.0000000	BNJ 1007	1
2014-09-10T20:05:00.0000000	CDE 1007	2
2014-09-10T20:06:00.0000000	BAC 1005	2

At the bottom right of the results table is a button labeled 'Open Results Folder'.

You can find the output path from the console output. Press any key to open the result folder.

The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window displays the log of a local run. The log starts with 'Start Local Run' and shows the execution of a Stream Analytics job. It includes logs for input detection, script compilation, query construction, and the creation of an output CSV file. The final message indicates a successful local run and provides the path to the output file.

```
=====
Start Local Run =====
1/17/2017 12:23:32 PM : Info : There is 1 input(s) for job
1/17/2017 12:23:32 PM : Info : Input Alias : 'EntryStream' , Input Type : 'Stream'
1/17/2017 12:23:32 PM : Info : Begin to compile script ....
1/17/2017 12:23:33 PM : Info : Compile job script successfully
1/17/2017 12:23:33 PM : Info : Begin to construct query inputs ...
1/17/2017 12:23:33 PM : Info : Detecting several necessary inputs for running this script : entrystream
1/17/2017 12:23:33 PM : Info : Construct query inputs successfully
1/17/2017 12:23:33 PM : Info : Begin to execute query ...
1/17/2017 12:23:34 PM : Info : Execute query successfully
1/17/2017 12:23:34 PM : Info : There is 1 output(s) for job
1/17/2017 12:23:34 PM : Info : Create file C:\Users\sujie\AppData\Local\Temp\ASALocalRun\2017-01-17-12-23-32\output.csv
for : output

===== Local run successfully : Save all output result to C:\Users\sujie\AppData\Local\Temp\ASALocalRun\2017-01-17-12-23-32 =====
Press any key to open output result folder.

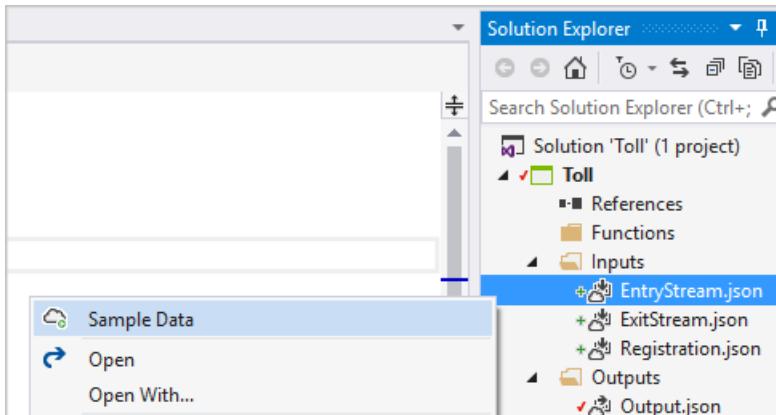
Press any key to continue . . .
```

Check the results in the local folder.

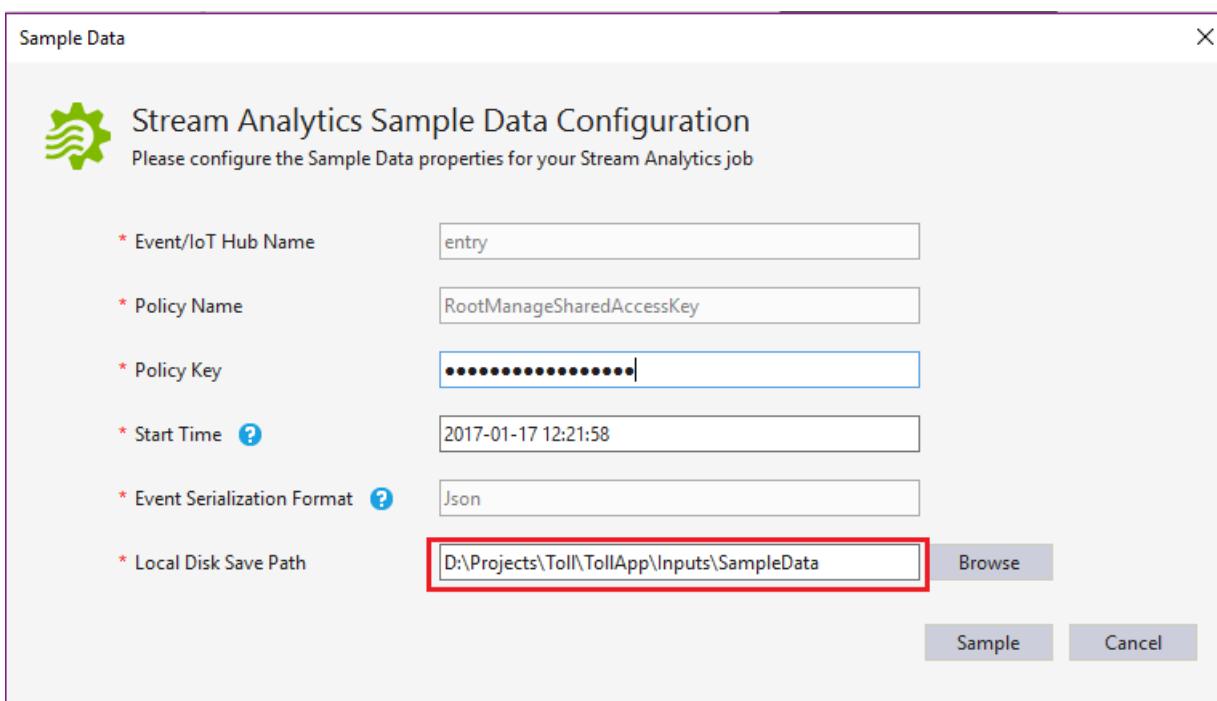
Name	Date modified	Type	Size
output.csv	8/8/2018 11:25 AM	Microsoft Excel Co...	1 KB
output.json	8/8/2018 11:25 AM	JSON File	2 KB

Sample input

You can also collect sample input data from your input sources to a local file. Right-click the input configuration file, and select **Sample Data**.



You can only sample data streaming from Event Hubs or IoT Hubs. Other input sources are not supported. In the pop-up dialog box, fill in the local path to save the sample data and select **Sample**.



You can see the progress in the **Output** window.

```
Output
Show output from: Stream Analytics Output
12/12/2016 12:25:53 : Create Directory d:\Projects\Toll\Toll\Inputs\SampleData
12/12/2016 12:25:53 : Sampling data ...
12/12/2016 12:25:53 : It may take several minutes, wait please
12/12/2016 12:26:11 : Sampling data to d:\Projects\Toll\Toll\Inputs\SampleData\EntryStream-86d34ef4-f4ac-468d-9f08-efcbc5f2a4.json successfully
```

Next steps

- [Use Visual Studio to view Azure Stream Analytics jobs](#)

- [Quickstart: Create a Stream Analytics job using Visual Studio](#)
- [Tutorial: Deploy an Azure Stream Analytics job with CI/CD using Azure DevOps](#)
- [Continuously integrate and develop with Stream Analytics tools](#)

Test live data locally using Azure Stream Analytics tools for Visual Studio (Preview)

12/7/2018 • 2 minutes to read • [Edit Online](#)

Azure Stream Analytics tools for Visual Studio allows you to test jobs locally from the IDE using live event streams from Azure Event Hub, IoT Hub, and Blob Storage. Live data local testing can't replace the [performance and scalability testing](#) you can perform in the cloud, but you can save time during functional testing by not having to submit to the cloud each time you want to test your Stream Analytics job. This feature is in preview and shouldn't be used for production workloads.

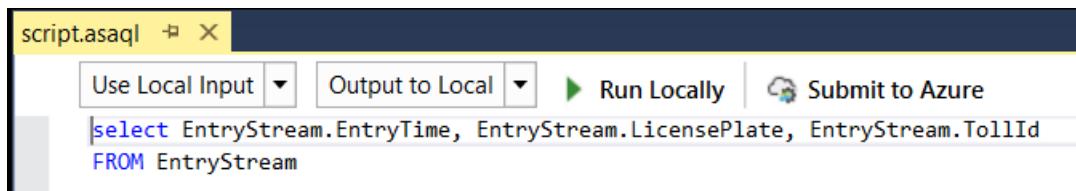
Testing options

The following local testing options are supported:

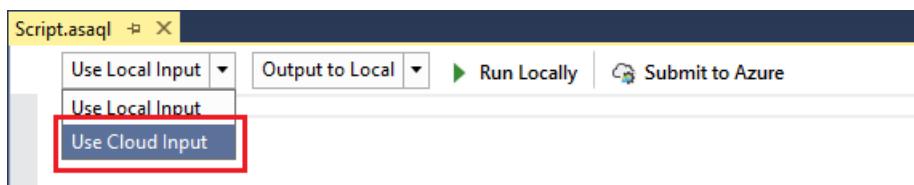
INPUT	OUTPUT	JOB TYPE
Local static data	Local static data	Cloud/Edge
Live input data	Local static data	Cloud
Live input data	Live output data	Cloud

Local testing with live data

- After you've created an [Azure Stream Analytics cloud project in Visual Studio](#), open **script.asaql**. The local testing uses local input and local output by default.



- To test live data, choose **Use Cloud Input** from the dropdown box.



- Set the **Start Time** to define when the job will start processing input data. The job might need to read input data ahead of time to ensure accurate results. The default time is set to 30 minutes ahead of the current time.



- Click **Run Locally**. A console window will appear with the running progress and job metrics. If you want to stop the process, you can do so manually.

```
C:\windows\system32\cmd.exe
DeserializationError : 0
=====
8/30/2018 11:13:27 PM
=====Metrics report=====
DroppedOrAdjustedEvents : 0
InputEventBytes : 211.06 Kib
Errors : 0
LateInputEvents : 0
InputEvents : 1.10 K
OutputEvents : 1.10 K
ConversionErrors : 0
EarlyInputEvents : 0
DeserializationError : 0
=====

8/30/2018 11:13:32 PM
=====Metrics report=====
DroppedOrAdjustedEvents : 0
InputEventBytes : 211.06 Kib
Errors : 0
LateInputEvents : 0
InputEvents : 1.10 K
OutputEvents : 1.10 K
ConversionErrors : 0
EarlyInputEvents : 0
DeserializationError : 0
=====
```

The output results are refreshed every three seconds with the first 500 output rows in the local run result window, and the output files are placed in your project path **ASALocalRun** folder. You can also open the output files by clicking **Open Results Folder** button in the local run result window.

Script.asaql

Use Cloud Input ▾ Output to Local ▾ Start Time 8/30/2017 10:39:02 PM Run Locally Submit to Azure

```
select * into Output from Input
```

Stream Analytics Local Run Results

output

TollId	EntryTime	LicensePlate	State	Make	Model	VehicleType	VehicleWeight	Toll	Tag	EventProcessedUtcTime	Partition
25	8/30/2018 5:22:17 AM	YQB 4701	PA	Toyota	Corolla	1	0	4	205429834	8/31/2018 6:18:33 AM	1
23	8/30/2018 5:22:18 AM	RFQ 2880	CT	Toyota	Camry	1	0	5	837531417	8/31/2018 6:18:34 AM	1
5	8/30/2018 5:22:18 AM	ZXC 3799	OR	Ford	Mustang	1	0	4	369391039	8/31/2018 6:18:34 AM	1
52	8/30/2018 5:22:20 AM	FYW 2785	CT	Ford	Focus	1	0	6	946930889	8/31/2018 6:18:34 AM	1
18	8/30/2018 5:22:20 AM	JFT 2176	CT	Honda	Civic	1	0	4	776717000	8/31/2018 6:18:34 AM	1
7	8/30/2018 5:22:21 AM	HYE 3407	CT	Peterbilt	389	2	2.675	21	343648940	8/31/2018 6:18:34 AM	1
5	8/30/2018 5:22:21 AM	LAU 3358	CA	Toyota	Rav4	1	0	5	172508389	8/31/2018 6:18:34 AM	1
64	8/30/2018 5:22:23 AM	VMD 9600	WA	Honda	CRV	1	0	6	358358187	8/31/2018 6:18:34 AM	1
0	8/30/2018 5:22:23 AM	ZRR 9041	WA	Volvo	S80	1	0	4	318778359	8/31/2018 6:18:34 AM	1
64	8/30/2018 5:22:23 AM	EUI 9201	PA	Honda	Accord	1	0	5	261534104	8/31/2018 6:18:34 AM	1
5	8/30/2018 5:22:24 AM	LZW 7265	CT	Chevy	Malibu	1	0	6	289557657	8/31/2018 6:18:34 AM	1
38	8/30/2018 5:22:26 AM	KBJ 5849	CA	Honda	Civic	1	0	5	660111395	8/31/2018 6:18:34 AM	1

Open Results Folder

- If you want to output the results to your cloud output sinks, choose **Output to Cloud** from the second dropdown box. Power BI and Azure Data Lake Storage are not supported output sinks.

Script.asaql

Use Cloud Input ▾ Output to Cloud ▾ Output Start Time 2018-9-17 22:08:29 Run Locally Submit to Azure

Limitations

- Power BI and Azure Data Lake Storage are not supported output sinks due to authentication model limitations.
- Only cloud input options have [time policies support](#), while local input options do not.

Next steps

- [Create a Stream Analytics job by using the Azure Stream Analytics tools for Visual Studio](#)
- [Install Azure Stream Analytics tools for Visual Studio](#)
- [Test Stream Analytics queries locally with Visual Studio](#)
- [Use Visual Studio to view Azure Stream Analytics jobs](#)

Use Visual Studio to view Azure Stream Analytics jobs

9/26/2018 • 2 minutes to read • [Edit Online](#)

Azure Stream Analytics tools for Visual Studio makes it easy for developers to manage their Stream Analytics jobs directly from the IDE. With Azure Stream Analytics tools, you can:

- [Create new jobs](#)
- Start, stop, and [monitor jobs](#)
- Check job results
- Export existing jobs to a project
- Test input and output connections
- [Run queries locally](#)

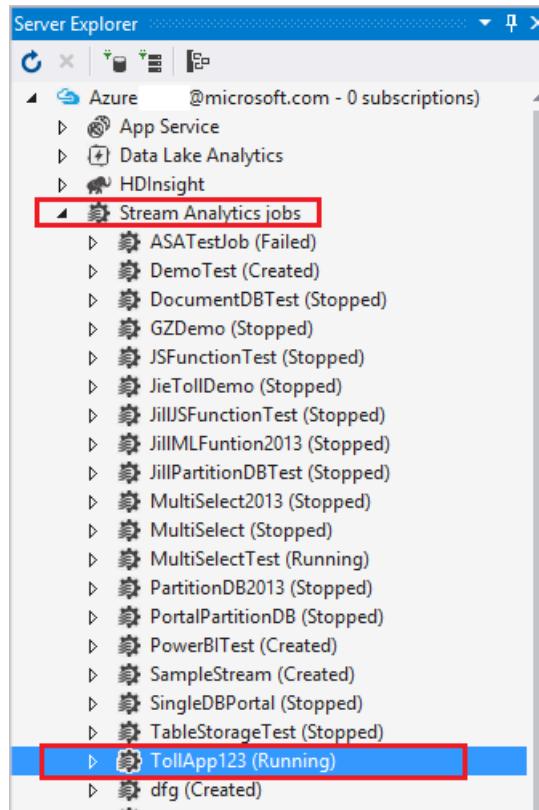
Learn how to [install Azure Stream Analytics tools for Visual Studio](#).

Explore the job view

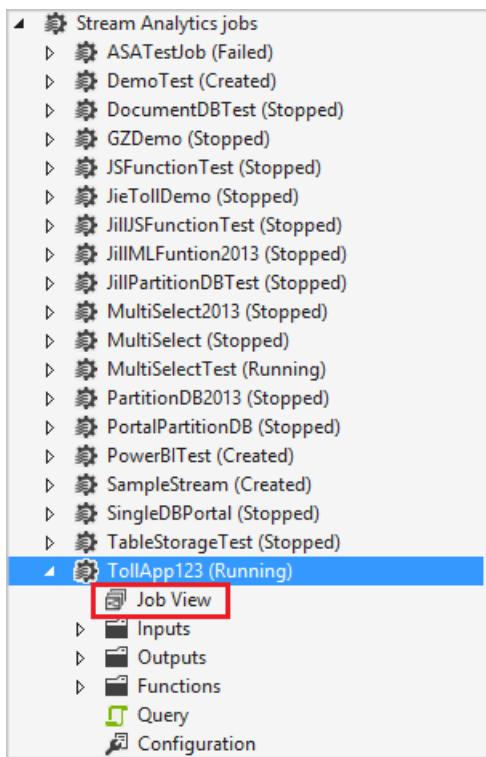
You can use the job view to interact with Azure Stream Analytics jobs from Visual Studio.

Open the job view

1. In **Server Explorer**, select **Stream Analytics jobs** and then select **Refresh**. Your job should appear under **Stream Analytics jobs**.



2. Expand your job node, and double-click on the **Job View** node to open a job view.



Start and stop jobs

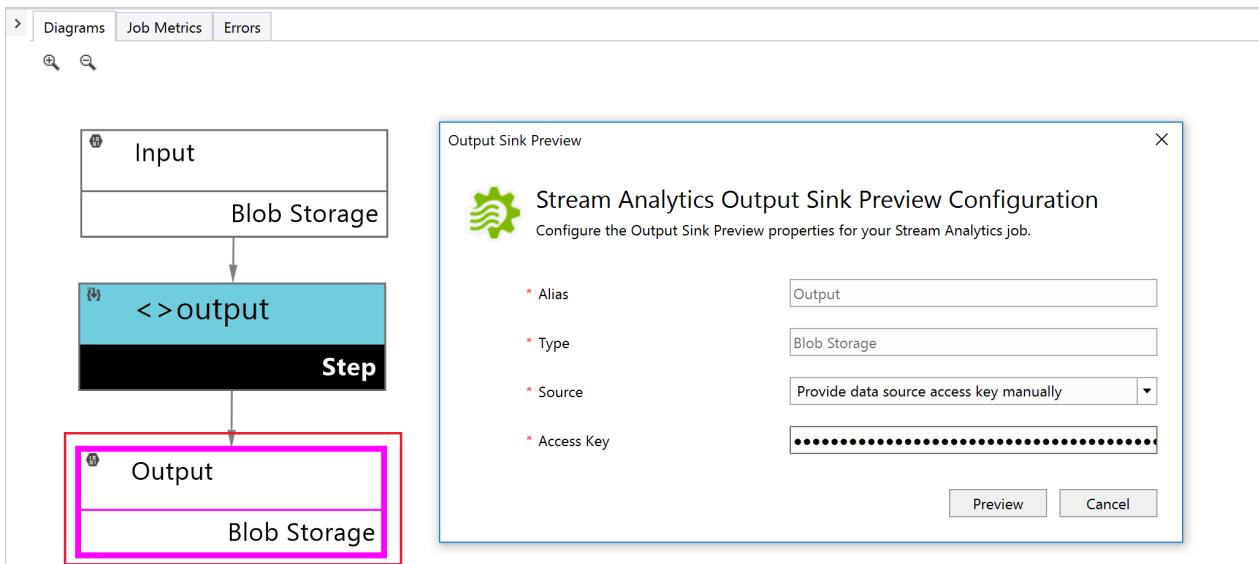
Azure Stream Analytics jobs can be fully managed from the job view in Visual Studio. Use the controls to start, stop, or delete a job.

Status	Created
Job Type	Cloud
Creation Time	7/19/2018 11:52:31 AM
Job Output Start Mode	N/A
Job Output Start Time	N/A
Last Output Time	N/A
Data Locale	en-US
Output Error Handling	Retry
Late Arrival Tolerance	5 Second(s)
Out of Order Tolerance	5 Second(s)
Out of Order Actions	Adjust
Compatibility Level	1.0
Stream Analytics Units	1

Job Metrics (Last 30 Mins)	
Total Input Events	0
Input Event Size	0 B
Total Output Events	0
Late Input Events	0
Out of Order Events	0
Events Handled by Functions	0
External Function Calls	0
Failed External Function Calls	0
Data Conversion Errors	0

Check job results

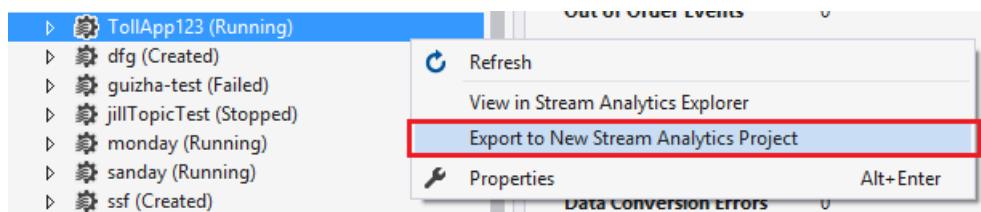
Stream Analytics tools for Visual Studio currently supports output preview for Azure Data Lake Storage and blob storage. To view result, simply double click the output node of the job diagram in **Job View** and enter the appropriate credentials.



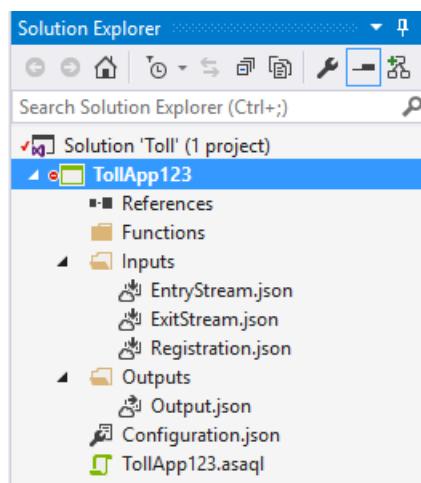
Export jobs to a project

There are two ways you can export an existing job to a project.

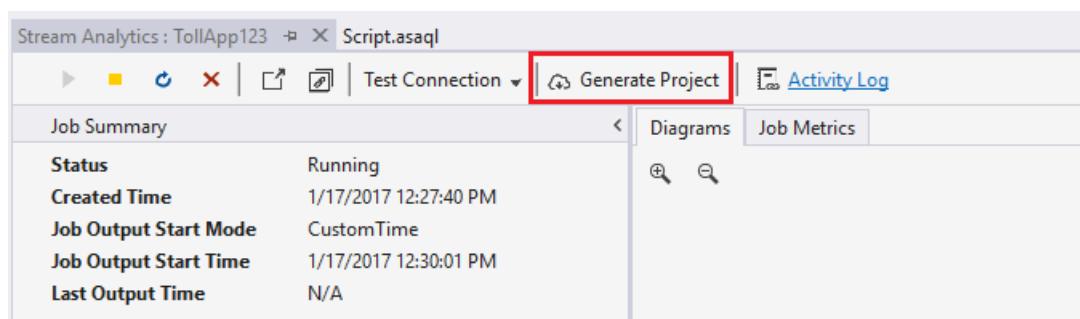
1. In **Server Explorer**, under the Stream Analytics Jobs node, right-click the job node. Select **Export to New Stream Analytics Project**.



The generated project appears in **Solution Explorer**.



2. In the job view, select **Generate Project**.



Test connections

Input and output connections can be tested from the **Job View** by selecting an option from the **Test Connection** dropdown.

The screenshot shows the Azure Stream Analytics Job View interface. At the top, there's a toolbar with icons for back, forward, refresh, and a dropdown labeled "Test Connection". Below the toolbar is a navigation bar with "Generate Project" and "Activity Log" buttons. On the right, there's a "Display" dropdown set to "Last 30 Mins". The main area contains a table with job details:

Creation Time	1/1/2018 3:51:25 PM
Job Output Start Mode	CustomTime
Job Output Start Time	1/1/2018 11:52:36 AM
Last Output Time	N/A
Data Locale	en-US
Outout Error Handling	Drop

To the right of the table is a diagram titled "Input" which connects to "Blob Storage". A red box highlights the "Test Connection" dropdown in the toolbar.

The **Test Connection** results are displayed in the **Output** window.

The screenshot shows the "Output" window with a yellow header. It displays the following text:

```
Fetching job info...
Testing input Input connection...
Succeed!
Test completed!
```

Next steps

- [Monitor and manage Azure Stream Analytics jobs using Visual Studio](#)
- [Quickstart: Create a Stream Analytics job using Visual Studio](#)
- [Tutorial: Deploy an Azure Stream Analytics job with CI/CD using Azure Pipelines](#)
- [Continuously integrate and develop with Stream Analytics tools](#)

Develop Stream Analytics Edge jobs using Visual Studio tools

6/21/2019 • 4 minutes to read • [Edit Online](#)

In this tutorial, you learn how to use Stream Analytics tools for Visual Studio. You learn how to author, debug, and create your Stream Analytics Edge jobs. After you create and test the job, you can go to the Azure portal to deploy it to your devices.

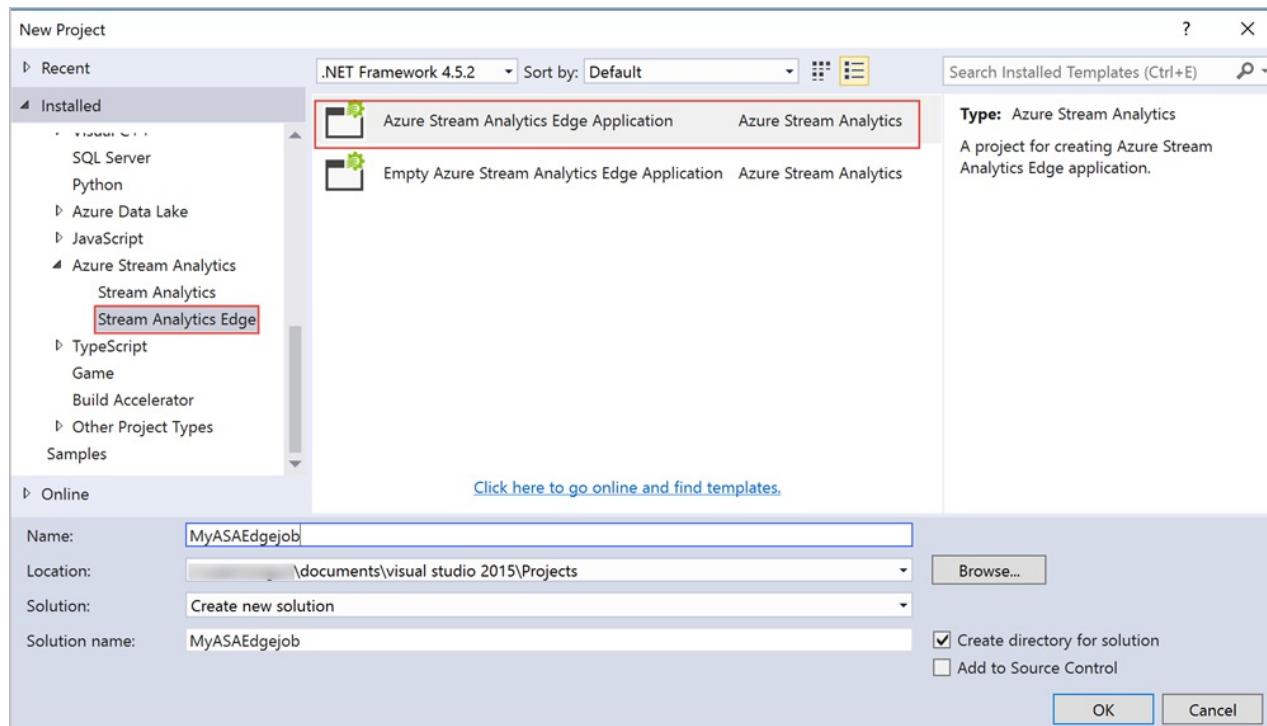
Prerequisites

You need the following prerequisites to complete this tutorial:

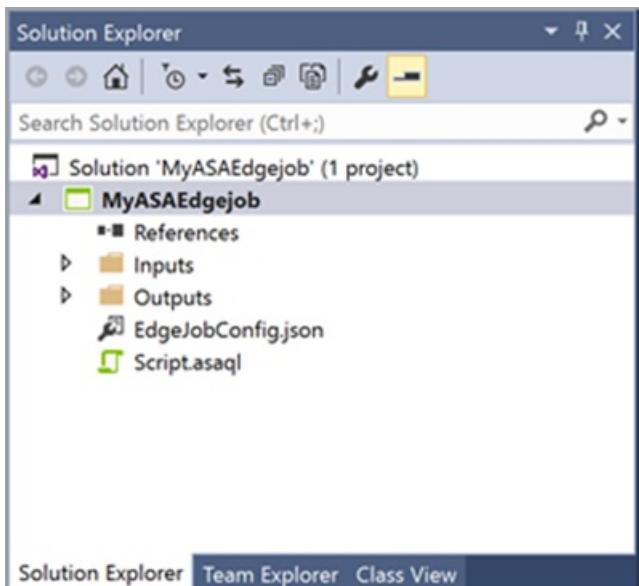
- Install [Visual Studio 2019](#), [Visual Studio 2015](#), or [Visual Studio 2013 Update 4](#). Enterprise (Ultimate/Premium), Professional, and Community editions are supported. Express edition isn't supported.
- Follow the [installation instructions](#) to install Stream Analytics tools for Visual Studio.

Create a Stream Analytics Edge project

From Visual Studio, select **File > New > Project**. Navigate to the **Templates** list on the left > expand **Azure Stream Analytics > Stream Analytics Edge > Azure Stream Analytics Edge Application**. Provide a Name, Location, and Solution name for your project and select **OK**.



After the project gets created, navigate to the **Solution Explorer** to view the folder hierarchy.

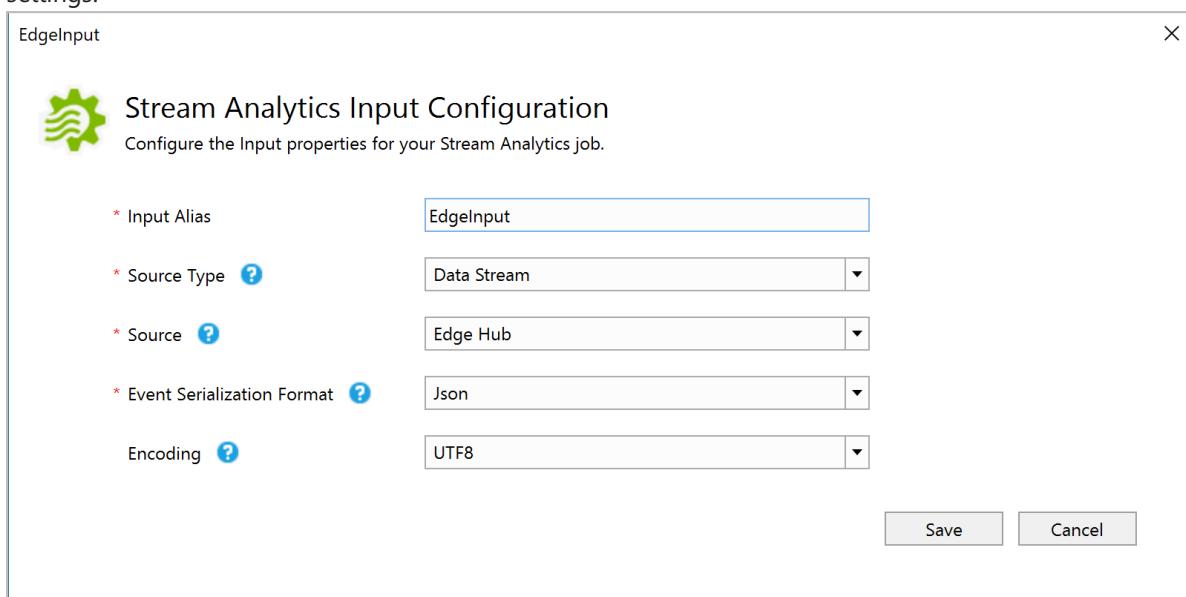


Choose the correct subscription

1. From your Visual Studio **View** menu, select **Server Explorer**.
2. Right click on **Azure** > Select **Connect to Microsoft Azure Subscription** > and then sign in with your Azure account.

Define inputs

1. From the **Solution Explorer**, expand the **Inputs** node you should see an input named **EdgeInput.json**. Double-click to view its settings.
2. Set Source Type to **Data Stream**. Then set Source to **Edge Hub**, Event Serialization Format to **Json**, and Encoding to **UTF8**. Optionally, you can rename the **Input Alias**, let's leave it as is for this example. In case you rename the input alias, use the name you specified when defining the query. Select **Save** to save the settings.



Define outputs

1. From the **Solution Explorer**, expand the **Outputs** node you should see an output named **EdgeOutput.json**. Double-click to view its settings.
2. Make sure to set Sink to select **Edge Hub**, set Event Serialization Format to **Json**, set Encoding to **UTF8**,

and set Format **Array**. Optionally, you can rename the **Output Alias**, let's leave it as is for this example. In case you rename the output alias, use the name you specified when defining the query. Select **Save** to save the settings.

The screenshot shows the 'Stream Analytics Output Configuration' dialog box. It includes fields for 'Output Alias' (set to 'EdgeOutput'), 'Sink' (set to 'Edge Hub'), 'Event Serialization Format' (set to 'Json'), 'Encoding' (set to 'UTF8'), and 'Format' (set to 'Array'). At the bottom are 'Save' and 'Cancel' buttons.

Define the transformation query

Stream Analytics jobs deployed in the Stream Analytics IoT Edge environments support most of [Stream Analytics Query Language reference](#). However, the following operations aren't yet supported for Stream Analytics Edge jobs:

CATEGORY	COMMAND
Other operators	<ul style="list-style-type: none">PARTITION BYTIMESTAMP BY OVERJavaScript UDFUser-defined aggregates (UDA)GetMetadataPropertyValueUsing more than 14 aggregates in a single step

When you create a Stream Analytics Edge job in the portal, the compiler will automatically warn you if you aren't using a supported operator.

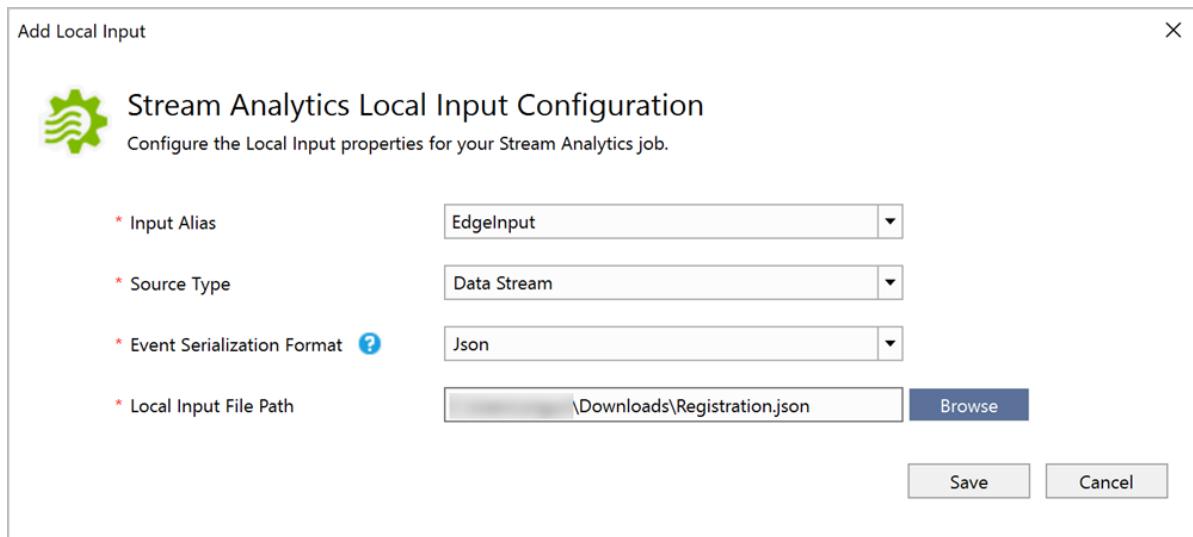
From your Visual Studio, define the following transformation query in the query editor (**script.asaql** file)

```
SELECT * INTO EdgeOutput  
FROM EdgeInput
```

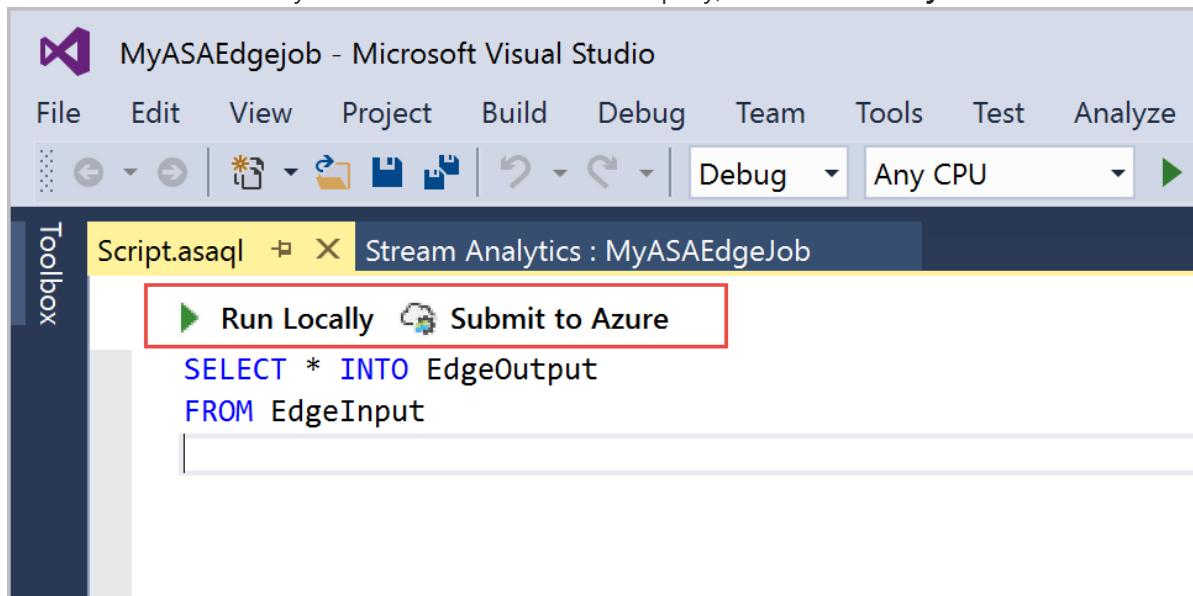
Test the job locally

To test the query locally, you should upload the sample data. You can get sample data by downloading Registration data from the [GitHub repository](#) and save it to your local computer.

1. To upload sample data, right click on **EdgeInput.json** file and choose **Add Local Input**
2. In the pop-up window > **Browse** the sample data from your local path > Select **Save**.

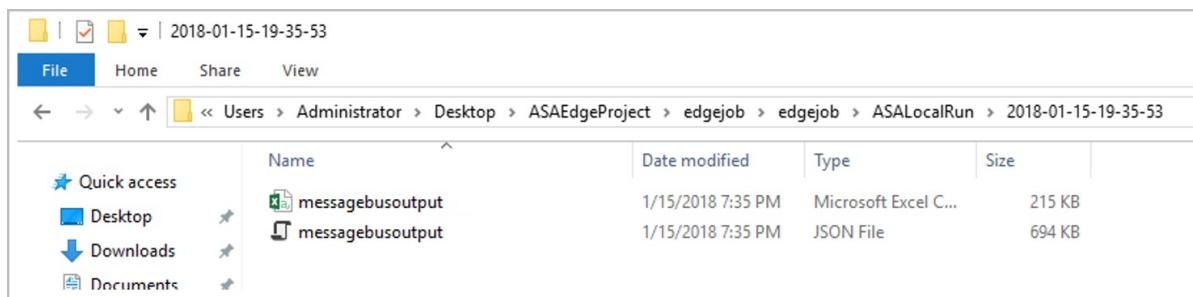


3. A file named **local_EdgeInput.json** is added automatically to your inputs folder.
4. You can either run it locally or submit to Azure. To test the query, select **Run Locally**.



5. The command prompt window shows the status of the job. When the job runs successfully, it creates a folder that looks like "2018-02-23-11-31-42" in your project folder path "Visual Studio 2015\Projects\MyASAEedgejob\MyASAEedgejob\ASALocalRun\2018-02-23-11-31-42". Navigate to the folder path to view the results in the local folder:

You can also sign in to the Azure portal and verify that the job is created.



Submit the job to Azure

1. Before you submit the job to Azure, you must connect to your Azure Subscription. Open **Server Explorer** > right click on **Azure** > **Connect to Microsoft Azure subscription** > sign in to your Azure subscription.
2. To submit the job to Azure, navigate to the query editor > select **Submit to Azure**.

3. A pop-up window opens. Choose to update an existing Stream Analytics Edge job or create a new one. When you update an existing job, it will replace all the job configuration, in this scenario, you'll publish a new job. Select **Create a New Azure Stream Analytics Job** > enter a name for your job something like **MyASAEEdgeJob** > choose the required **Subscription**, **Resource Group**, and **Location** > Select **Submit**.

The screenshot shows a 'Submit Job' dialog box. At the top, there are two radio button options: 'Update existing Azure Stream Analytics job' and 'Create a new Azure Stream Analytics job'. The second option is selected and highlighted with a red border. Below the radio buttons are four input fields with asterisks indicating they are required:

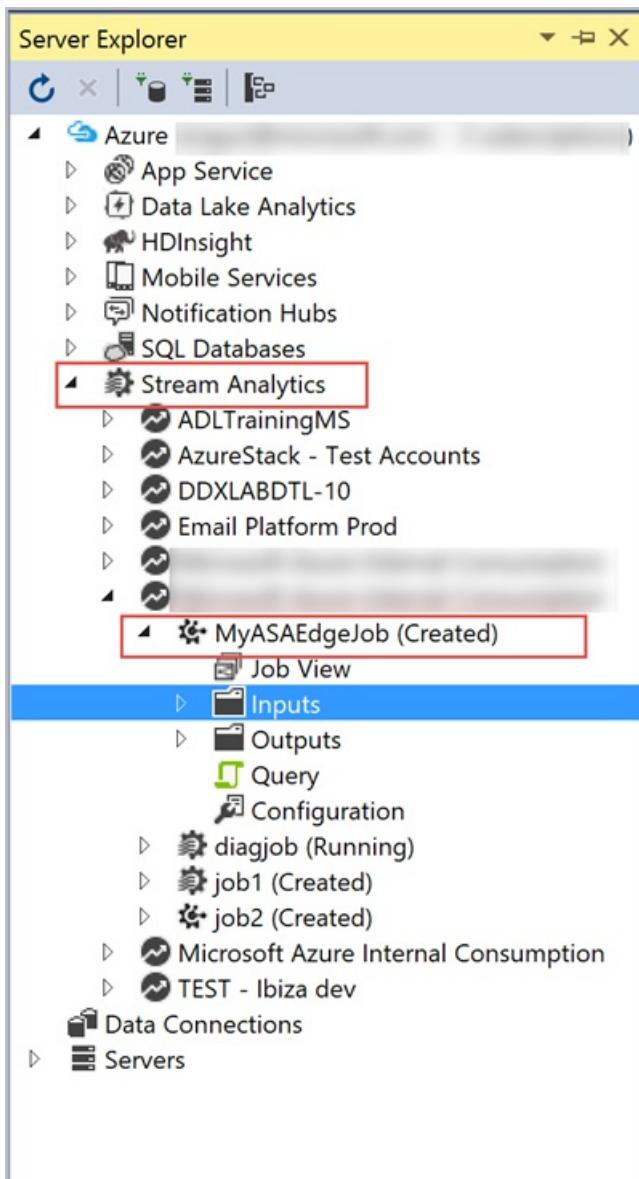
- * Job Name: MyASAEEdgeJob
- * Subscription: (dropdown menu)
- * Resource Group: ASAIoT
- * Location: West US 2

At the bottom right of the dialog box are two buttons: 'Submit' and 'Cancel'.

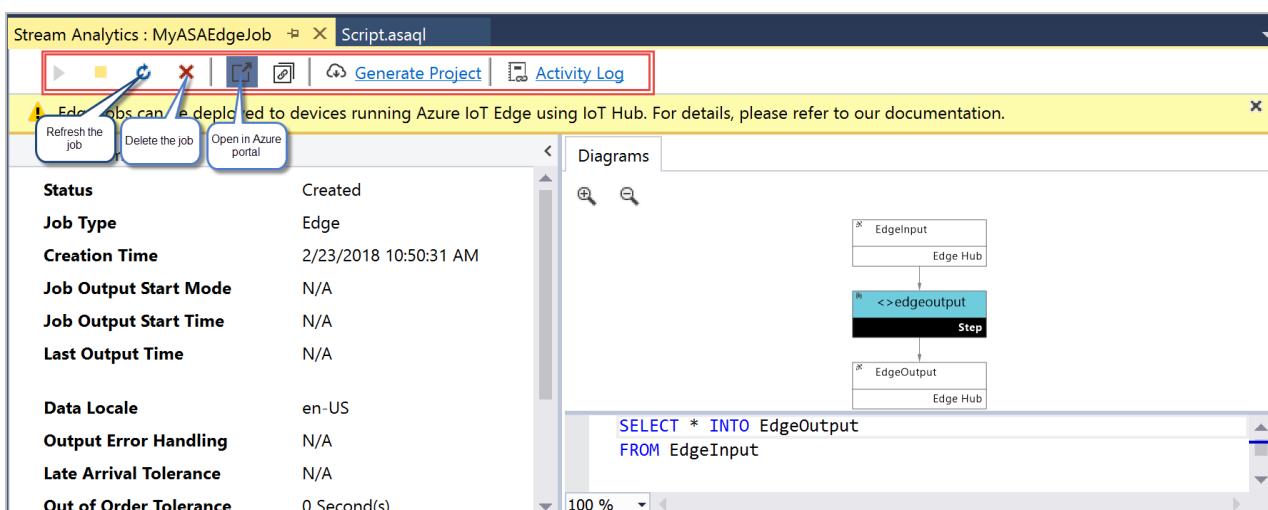
Now your Stream Analytics Edge job has been created. You can refer to the [Run jobs on IoT Edge tutorial](#) to learn how to deploy it to your devices.

Manage the job

You can view the status of job and the job diagram from the Server Explorer. From **Stream Analytics in Server Explorer**, expand the subscription and the resource group where you deployed the Stream Analytics Edge job. You can view the MyASAEEdgejob with status **Created**. Expand your job node and double-click on it to open the job view.



The job view window provides you with operations such as refreshing the job, deleting the job, and opening the job from Azure portal.



Next steps

- More information on Azure IoT Edge
- ASA on IoT Edge tutorial
- Send feedback to the team using this survey

Continuously integrate and develop with Azure Stream Analytics CI/CD NuGet package

5/16/2019 • 2 minutes to read • [Edit Online](#)

This article describes how to use the Azure Stream Analytics CI/CD NuGet package to set up a continuous integration and deployment process.

Use version 2.3.0000.0 or above of [Stream Analytics tools for Visual Studio](#) to get support for MSBuild.

A NuGet package is available: [Microsoft.Azure.Stream.Analytics.CICD](#). It provides the MSBuild, local run, and deployment tools that support the continuous integration and deployment process of [Stream Analytics Visual Studio projects](#).

NOTE

The NuGet package can be used only with the 2.3.0000.0 or above version of Stream Analytics Tools for Visual Studio. If you have projects created in previous versions of Visual Studio tools, just open them with the 2.3.0000.0 or above version and save. Then the new capabilities are enabled.

For more information, see [Stream Analytics tools for Visual Studio](#).

MSBuild

Like the standard Visual Studio MSBuild experience, to build a project you have two options. You can right-click the project, and then choose **Build**. You also can use **MSBuild** in the NuGet package from the command line.

```
./build/msbuild /t:build [Your Project Full Path]
/p:CompilerTaskAssemblyFile=Microsoft.WindowsAzure.StreamAnalytics.Common.CompileService.dll
/p:ASATargetsFilePath="[NuGet Package Local Path]\build\StreamAnalytics.targets"
```

When a Stream Analytics Visual Studio project builds successfully, it generates the following two Azure Resource Manager template files under the **bin/[Debug/Retail]/Deploy** folder:

- Resource Manager template file

```
[ProjectName].JobTemplate.json
```

- Resource Manager parameters file

```
[ProjectName].JobTemplate.parameters.json
```

The default parameters in the parameters.json file are from the settings in your Visual Studio project. If you want to deploy to another environment, replace the parameters accordingly.

NOTE

For all the credentials, the default values are set to null. You are **required** to set the values before you deploy to the cloud.

```
"Input_EntryStream_sharedAccessPolicyKey": {  
    "value": null  
},
```

Learn more about how to [deploy with a Resource Manager template file and Azure PowerShell](#). Learn more about how to [use an object as a parameter in a Resource Manager template](#).

To use Managed Identity for Azure Data Lake Store Gen1 as output sink, you need to provide Access to the service principal using PowerShell before deploying to Azure. Learn more about how to [deploy ADLS Gen1 with Managed Identity with Resource Manager template](#).

Command-line tool

Build the project

The NuGet package has a command-line tool called **SA.exe**. It supports project build and local testing on an arbitrary machine, which you can use in your continuous integration and continuous delivery process.

The deployment files are placed under the current directory by default. You can specify the output path by using the following `-OutputPath` parameter:

```
./tools/SA.exe build -Project [Your Project Full Path] [-OutputPath <outputPath>]
```

Test the script locally

If your project has specified local input files in Visual Studio, you can run an automated script test by using the `localrun` command. The output result is placed under the current directory.

```
localrun -Project [ProjectFullPath]
```

Generate a job definition file to use with the Stream Analytics PowerShell API

The `arm` command takes the job template and job template parameter files generated through build as input. Then it combines them into a job definition JSON file that can be used with the Stream Analytics PowerShell API.

```
arm -JobTemplate <templateFilePath> -JobParameterFile <jobParameterFilePath> [-OutputFile <asaArmFilePath>]
```

Example:

```
./tools/SA.exe arm -JobTemplate "ProjectA.JobTemplate.json" -JobParameterFile  
"ProjectA.JobTemplate.parameters.json" -OutputFile "JobDefinition.json"
```

Next steps

- [Quickstart: Create an Azure Stream Analytics cloud job in Visual Studio](#)
- [Test Stream Analytics queries locally with Visual Studio](#)
- [Explore Azure Stream Analytics jobs with Visual Studio](#)

Develop .NET Standard user-defined functions for Azure Stream Analytics Edge jobs (Preview)

4/25/2019 • 5 minutes to read • [Edit Online](#)

Azure Stream Analytics offers a SQL-like query language for performing transformations and computations over streams of event data. There are many built-in functions, but some complex scenarios require additional flexibility. With .NET Standard user-defined functions (UDF), you can invoke your own functions written in any .NET standard language (C#, F#, etc.) to extend the Stream Analytics query language. UDFs allow you to perform complex math computations, import custom ML models using ML.NET, and use custom imputation logic for missing data. The UDF feature for Stream Analytics Edge jobs is currently in preview and shouldn't be used in production workloads.

Overview

Visual Studio tools for Azure Stream Analytics make it easy for you to write UDFs, test your jobs locally (even offline), and publish your Stream Analytics job to Azure. Once published to Azure, you can deploy your job to IoT devices using IoT Hub.

There are three ways to implement UDFs:

- CodeBehind files in an ASA project
- UDF from a local project
- An existing package from an Azure storage account

Package path

The format of any UDF package has the path `/UserCustomCode/CLR/*`. Dynamic Link Libraries (DLLs) and resources are copied under the `/UserCustomCode/CLR/*` folder, which helps isolate user DLLs from system and Azure Stream Analytics DLLs. This package path is used for all functions regardless of the method used to employ them.

Supported types and mapping

UDF TYPE (C#)	AZURE STREAM ANALYTICS TYPE
long	bigint
double	double
string	nvarchar(max)
dateTime	dateTime
struct	IRecord
object	IRecord
Array<object>	IArray

UDF TYPE (C#)	AZURE STREAM ANALYTICS TYPE
dictionary<string, object>	IRecord

CodeBehind

You can write user-defined functions in the **Script.asql** CodeBehind. Visual Studio tools will automatically compile the CodeBehind file into an assembly file. The assemblies are packaged as a zip file and uploaded to your storage account when you submit your job to Azure. You can learn how to write a C# UDF using CodeBehind by following the [C# UDF for Stream Analytics Edge jobs](#) tutorial.

Local project

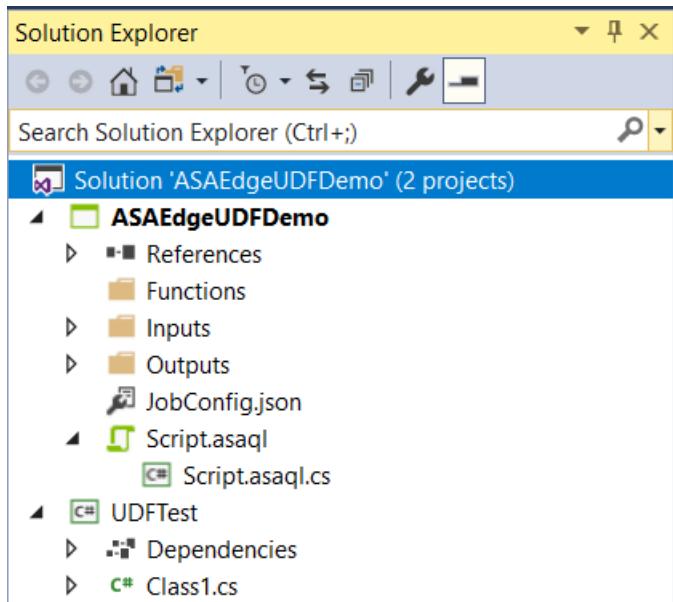
User-defined functions can be written in an assembly that is later referenced in an Azure Stream Analytics query. This is the recommended option for complex functions that require the full power of a .NET Standard language beyond its expression language, such as procedural logic or recursion. UDFs from a local project might also be used when you need to share the function logic across several Azure Stream Analytics queries. Adding UDFs to your local project gives you the ability to debug and test your functions locally from Visual Studio.

To reference a local project:

1. Create a new class library in your solution.
2. Write the code in your class. Remember that the classes must be defined as *public* and objects must be defined as *static public*.
3. Build your project. The tools will package all the artifacts in the bin folder to a zip file and upload the zip file to the storage account. For external references, use assembly reference instead of the NuGet package.
4. Reference the new class in your Azure Stream Analytics project.
5. Add a new function in your Azure Stream Analytics project.
6. Configure the assembly path in the job configuration file, `JobConfig.json`. Set the Assembly Path to **Local Project Reference or CodeBehind**.
7. Rebuild both the function project and the Azure Stream Analytics project.

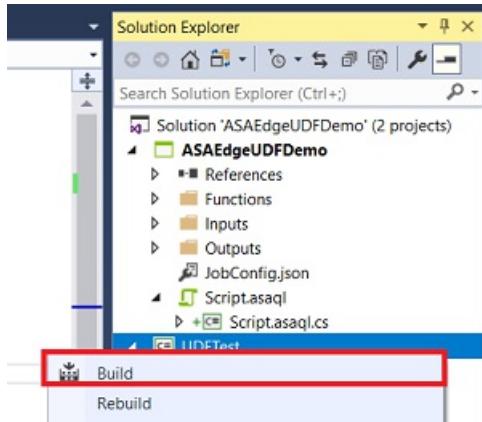
Example

In this example, **UDFTest** is a C# class library project and **ASAEdgeUDFDemo** is the Azure Stream Analytics Edge project, which will reference **UDFTest**.

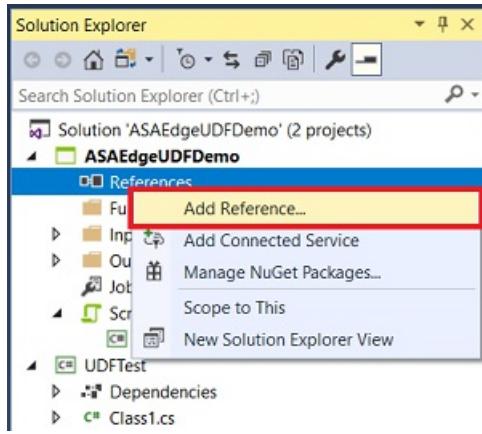


1. Build your C# project, which will enable you to add a reference to your C# UDF from the Azure Stream

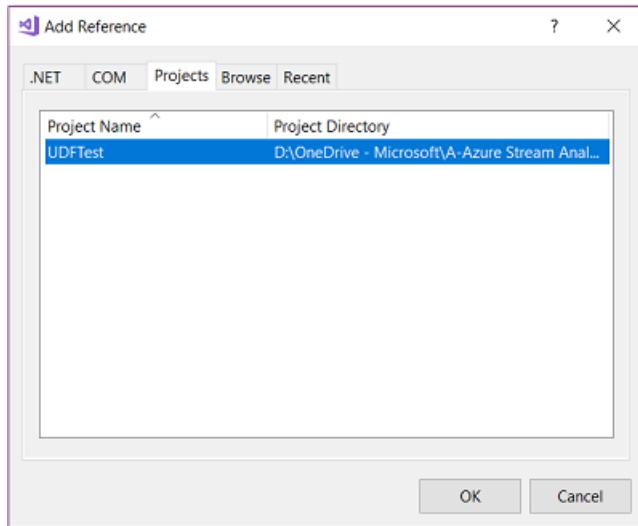
Analytics query.



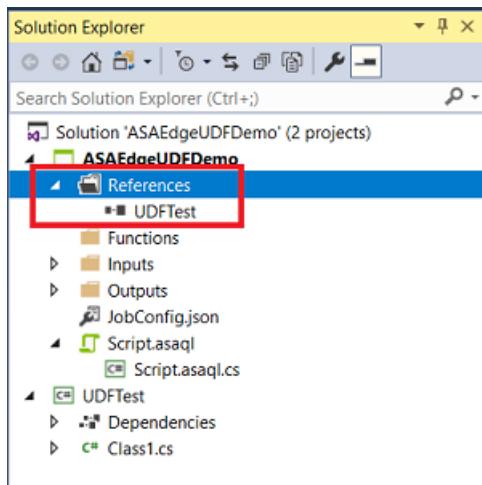
2. Add the reference to the C# project in the ASA Edge project. Right-click the References node and choose Add Reference.



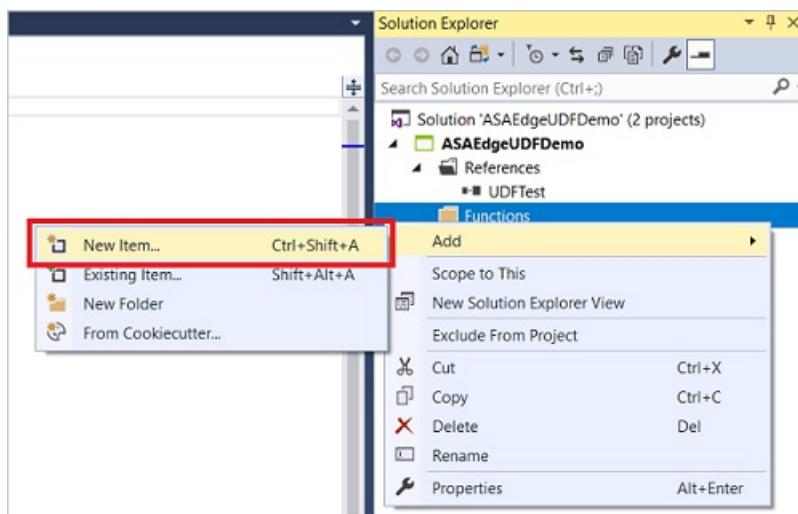
3. Choose the C# project name from the list.



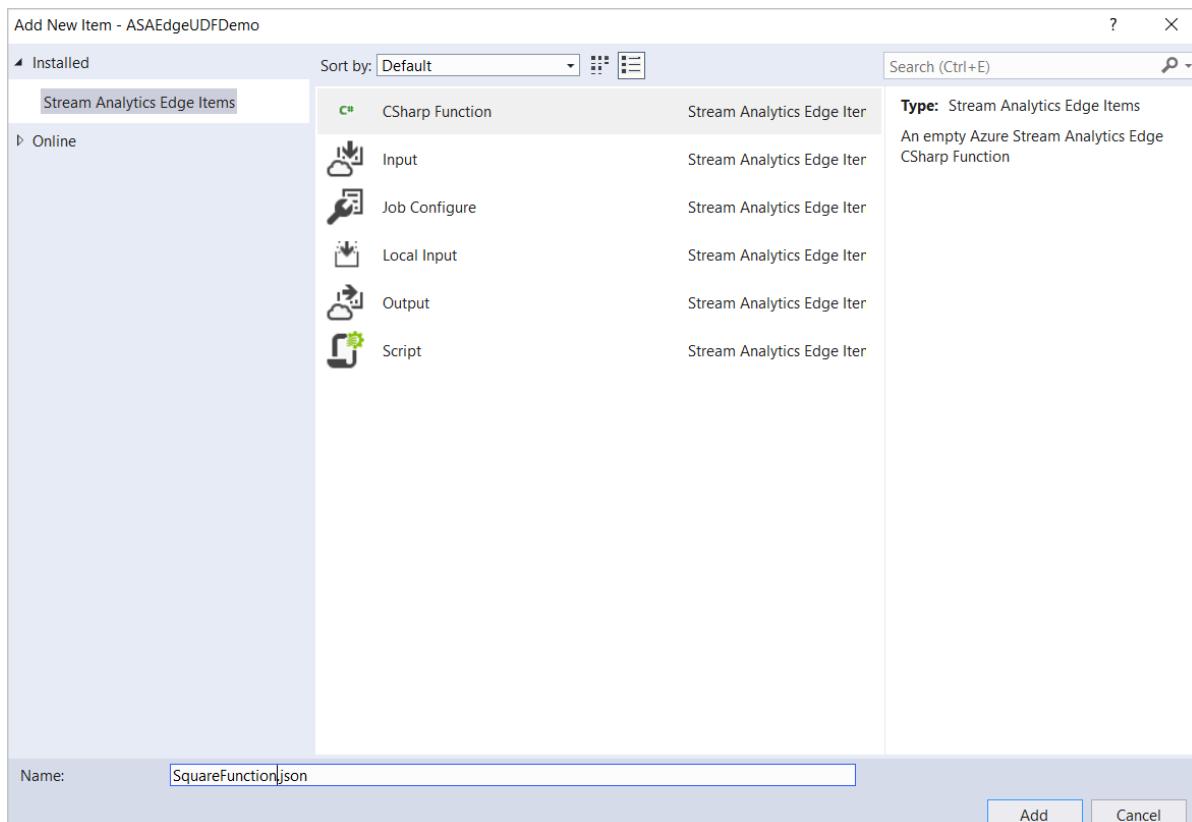
4. You should see the **UDFTest** listed under **References** in **Solution Explorer**.



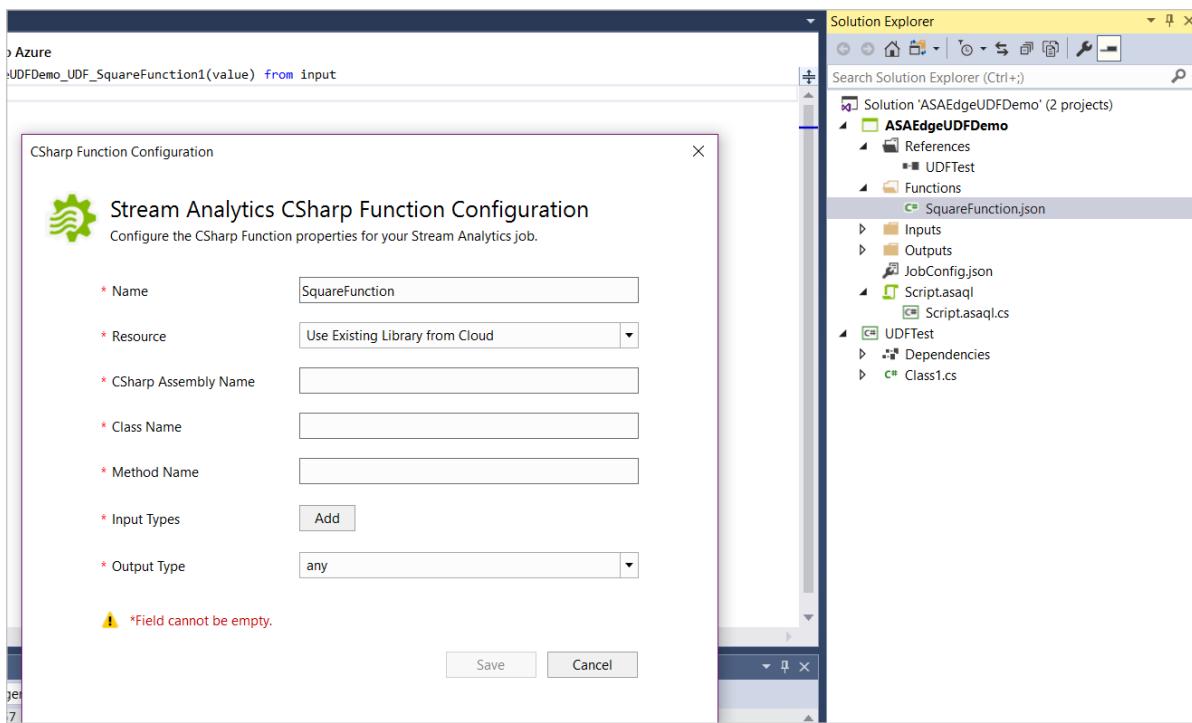
5. Right click on the **Functions** folder and choose **New Item**.



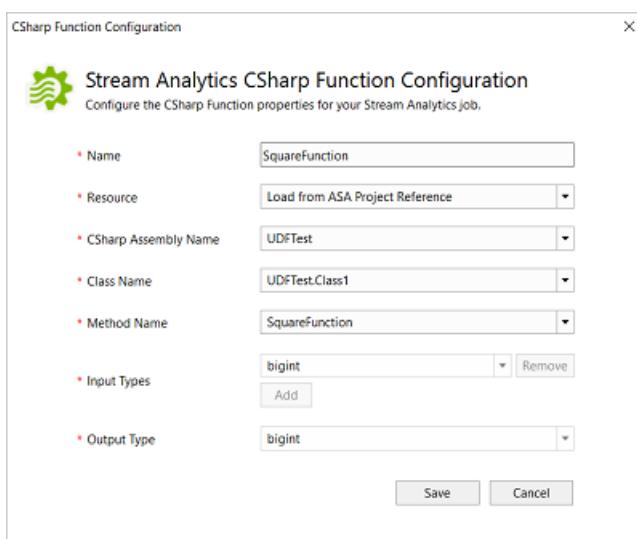
6. Add a C# function **SquareFunction.json** to your Azure Stream Analytics project.



7. Double-click the function in **Solution Explorer** to open the configuration dialog.



8. In the C# function configuration, choose **Load from ASA Project Reference** and the related assembly, class, and method names from the dropdown list. To refer to the methods, types, and functions in the Stream Analytics Edge query, the classes must be defined as *public* and the objects must be defined as *static public*.



Existing packages

You can author .NET Standard UDFs in any IDE of your choice and invoke them from your Azure Stream Analytics query. First compile your code and package all the DLLs. The format of the package has the path

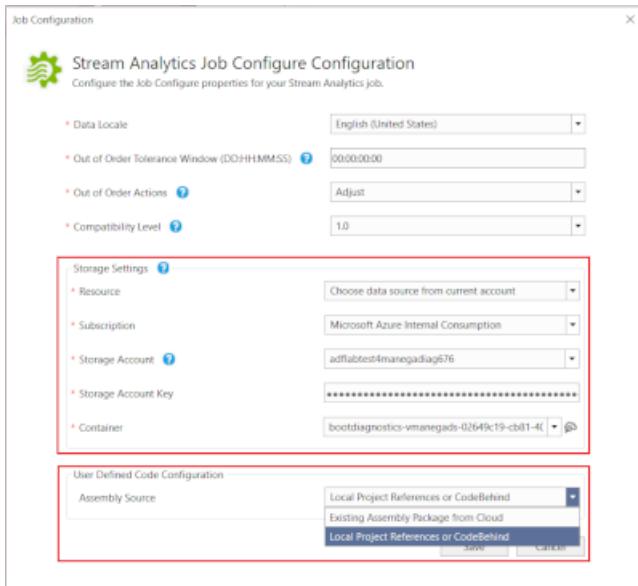
`/UserCustomCode/CLR/*`. Then, upload `UserCustomCode.zip` to the root of the container in your Azure storage account.

Once assembly zip packages have been uploaded to your Azure storage account, you can use the functions in Azure Stream Analytics queries. All you need to do is include the storage information in the Stream Analytics Edge job configuration. You can't test the function locally with this option because Visual Studio tools will not download your package. The package path is parsed directly to the service.

To configure the assembly path in the job configuration file, `JobConfig.json`:

Expand the **User-Defined Code Configuration** section, and fill out the configuration with the following suggested values:

SETTING	SUGGESTED VALUE
Assembly Source	Existing Assembly Packages from Cloud
Resource	Choose data from current account
Subscription	Choose your subscription.
Storage Account	Choose your storage account.
Container	Choose the container you created in your storage account.



Limitations

The UDF preview currently has the following limitations:

- .NET Standard languages can only be used for Azure Stream Analytics on IoT Edge. For cloud jobs, you can write JavaScript user-defined functions. To learn more, visit the [Azure Stream Analytics JavaScript UDF tutorial](#).
- .NET Standard UDFs can only be authored in Visual Studio and published to Azure. Read-only versions of .NET Standard UDFs can be viewed under **Functions** in the Azure portal. Authoring of .NET Standard functions is not supported in the Azure portal.
- The Azure portal query editor shows an error when using .NET Standard UDF in the portal.
- Because the custom code shares context with Azure Stream Analytics engine, custom code can't reference anything that has a conflicting namespace/dll_name with Azure Stream Analytics code. For example, you can't reference *Newtonsoft.Json*.

Next steps

- [Tutorial: Write a C# user-defined function for an Azure Stream Analytics Edge job \(Preview\)](#)
- [Tutorial: Azure Stream Analytics JavaScript user-defined functions](#)
- [Use Visual Studio to view Azure Stream Analytics jobs](#)

Test Stream Analytics queries locally with Visual Studio Code

5/16/2019 • 2 minutes to read • [Edit Online](#)

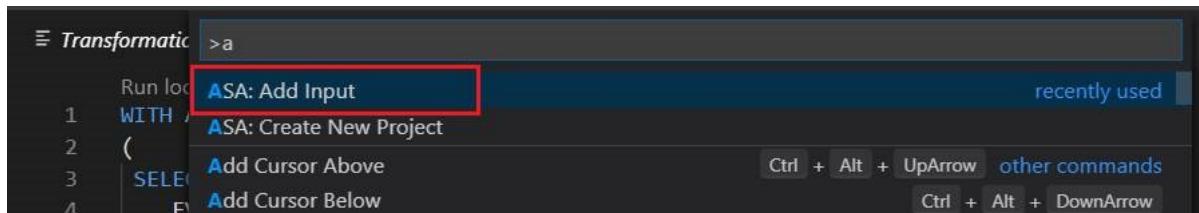
You can use Azure Stream Analytics tools for Visual Studio Code to test your Stream Analytics jobs locally with sample data.

Use this [quickstart](#) to learn how to create a Stream Analytics job using Visual Studio Code.

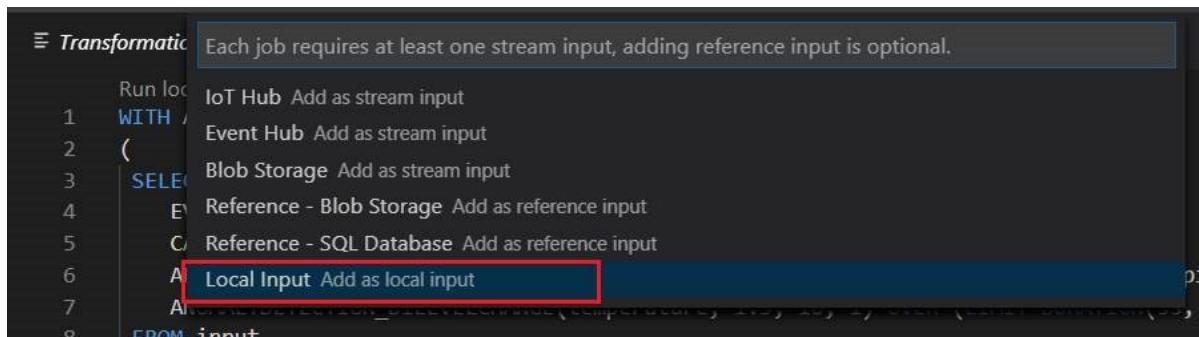
Run queries locally

You can use the Azure Stream Analytics extension for Visual Studio Code to test your Stream Analytics jobs locally with sample data.

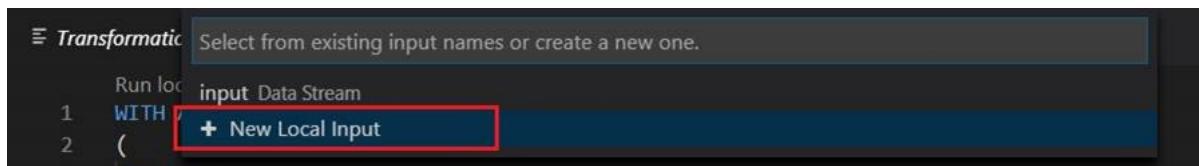
- Once you've created your Stream Analytics job, use **Ctrl+Shift+P** to open the command palette. Then type and select **ASA: Add Input**.



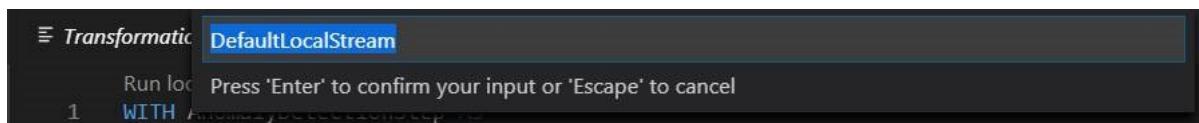
- Select **Local Input**.



- Select **+ New Local Input**.



- Enter the same input alias that you used in your query.



- In the **LocalInput_DefaultLocalStream.json** file, enter the file path where your local data file is located.

```
1 LocalInput_DefaultLocalStream.json ×
1  {
2   Edit
3   "InputAlias": "DefaultLocalStream",
4   Select
5   "Type": "Data Stream",
6   Select
7   "Format": "Json",
8   Select local file path
9   "FilePath": "[Required]",
10  "ScriptType": "InputMock"
11 }
```

6. Return to your query editor, and select **Run locally**.

```
Transformation.asaql ×
Run locally | Select from your subscriptions
1 WITH AnomalyDetectionStep AS
2 (
3   SELECT
4     EVENTPROCESSEDUTCTIME as time,
5     CAST(temperature as float) as temp,
6     ANOMALYDETECTION_SPIKE(temperature, 3, 3, 1)
7     ANOMALYDETECTION_BILEVELCHANGE(temperature,
8   FROM input
9 )
```

Next steps

- [Create an Azure Stream Analytics cloud job in Visual Studio Code \(Preview\)](#)
- [Explore Azure Stream Analytics jobs with Visual Studio Code \(Preview\)](#)

Continuously integrate and develop with Stream Analytics CI/CD npm package

5/22/2019 • 2 minutes to read • [Edit Online](#)

This article describes how to use the Azure Stream Analytics CI/CD npm package to set up a continuous integration and deployment process.

Build the VS Code project

You can enable continuous integration and deployment for Azure Stream Analytics jobs using the **asa-streamanalytics-cicd** npm package. The npm package provides the tools to generate Azure Resource Manager templates of [Stream Analytics Visual Studio Code projects](#). It can be used on Windows, macOS, and Linux without installing Visual Studio Code.

Once you have [downloaded the package](#), use the following command to output the Azure Resource Manager templates. The **scriptPath** argument is the absolute path to the **asaql** file in your project. Make sure the **asaproj.json** and **JobConfig.json** files are in the same folder with the script file. If the **outputPath** is not specified, the templates will be placed in the **Deploy** folder under the project's **bin** folder.

```
azure-streamanalytics-cicd build -scriptPath <scriptFullPath> -outputPath <outputPath>
```

Example (on macOS)

```
azure-streamanalytics-cicd build -scriptPath "/Users/roger/projects/samplejob/script.asaql"
```

When a Stream Analytics Visual Studio Code project builds successfully, it generates the following two Azure Resource Manager template files under the **bin/[Debug/Retail]/Deploy** folder:

- Resource Manager template file

```
[ProjectName].JobTemplate.json
```

- Resource Manager parameters file

```
[ProjectName].JobTemplate.parameters.json
```

The default parameters in the **parameters.json** file are from the settings in your Visual Studio Code project. If you want to deploy to another environment, replace the parameters accordingly.

NOTE

For all the credentials, the default values are set to null. You are **required** to set the values before you deploy to the cloud.

```
"Input_EntryStream_sharedAccessPolicyKey": {  
    "value": null  
},
```

Learn more about how to [deploy with a Resource Manager template file and Azure PowerShell](#). Learn more about how to [use an object as a parameter in a Resource Manager template](#).

To use Managed Identity for Azure Data Lake Store Gen1 as output sink, you need to provide Access to the service principal using PowerShell before deploying to Azure. Learn more about how to [deploy ADLS Gen1 with Managed Identity with Resource Manager template](#).

Next steps

- [Quickstart: Create an Azure Stream Analytics cloud job in Visual Studio Code \(Preview\)](#)
- [Test Stream Analytics queries locally with Visual Studio Code \(Preview\)](#)
- [Explore Azure Stream Analytics with Visual Studio Code \(Preview\)](#)

Explore Azure Stream Analytics with Visual Studio Code (Preview)

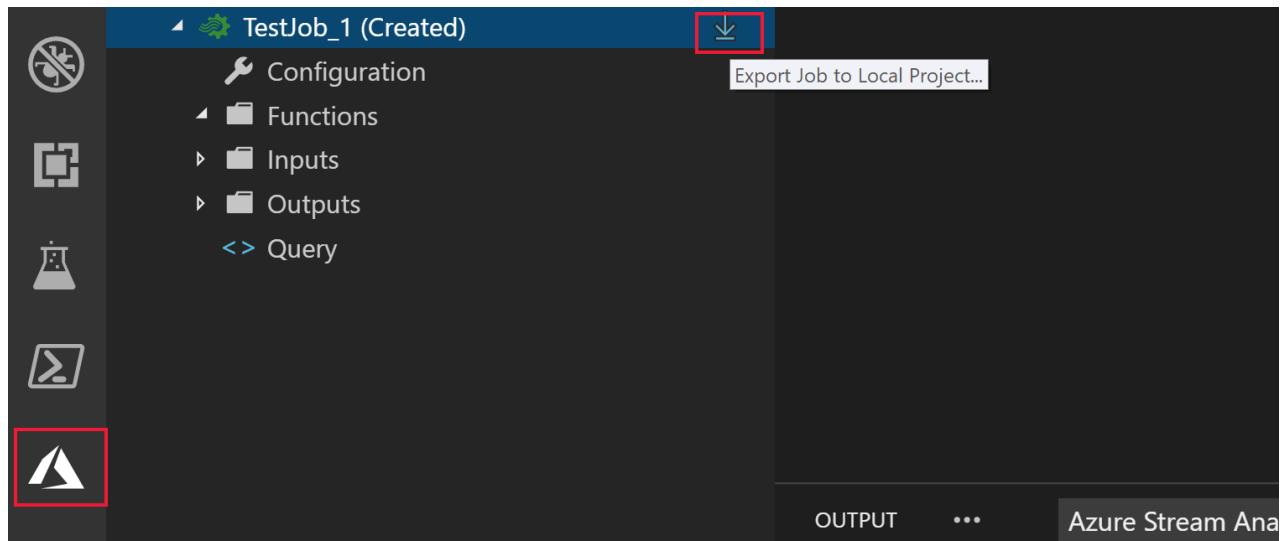
5/16/2019 • 2 minutes to read • [Edit Online](#)

The Azure Stream Analytics for Visual Studio Code extension gives developers a lightweight experience for managing their Stream Analytics jobs. It can be used on Windows, Mac and Linux. With the Azure Stream Analytics extension, you can:

- [Create](#), start, and stop jobs
- Export existing jobs to a local project
- List jobs and view job entities

Export a job to a local project

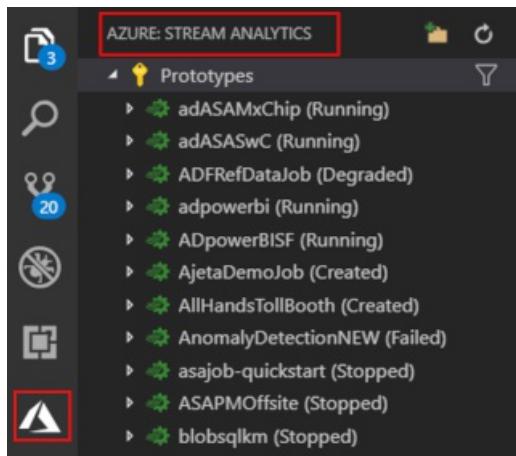
To export a job to a local project, locate the job you wish to export in the **Stream Analytics Explorer** in Visual Studio Code. Then select a folder for your project. The project is exported to the folder you select, and you can continue to manage the job from Visual Studio Code. For more information on using Visual Studio Code to manage Stream Analytics jobs, see the Visual Studio Code [quickstart](#).



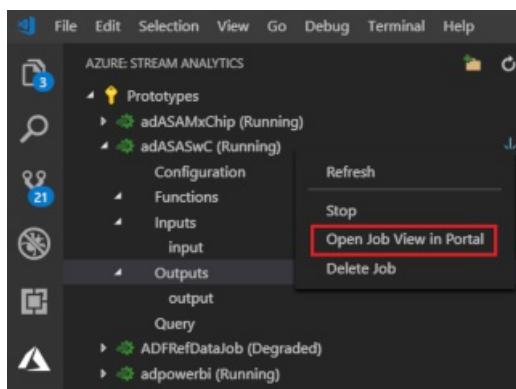
List job and view job entities

You can use the job view to interact with Azure Stream Analytics jobs from Visual Studio.

1. Click the **Azure** icon on Visual Studio Code Activity Bar and then expand **Stream Analytics node**. Your jobs should appear under your subscriptions.



2. Expand your job node, you can open and view the job query, configuration, inputs, outputs and functions.
3. Right click your job node, and choose the **Open Job View in Portal** node to open the job view in the Azure portal.



Next steps

- [Create an Azure Stream Analytics cloud job in Visual Studio Code \(Preview\)](#)

Troubleshoot input connections

4/11/2019 • 5 minutes to read • [Edit Online](#)

This page describes common issues with input connections and how to troubleshoot them.

Input events not received by job

1. Test your connectivity. Verify connectivity to inputs and outputs by using the **Test Connection** button for each input and output.
2. Examine your input data.

To verify that input data is flowing into Event Hub, use [Service Bus Explorer](#) to connect to Azure Event Hub (if Event Hub input is used).

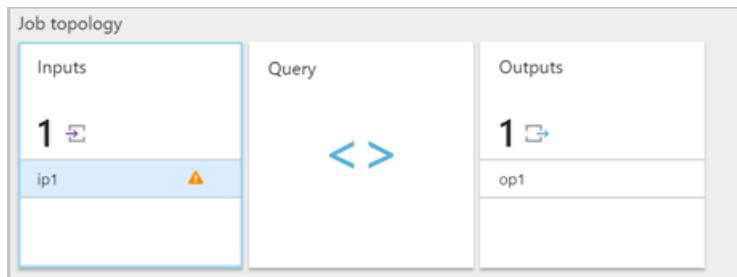
Use the **Sample Data** button for each input, and download the input sample data.

Inspect the sample data to understand the shape of the data: the schema and the [data types](#).

Malformed input events causes deserialization errors

Deserialization issues are caused when the input stream of your Stream Analytics job contains malformed messages. For example, a malformed message could be caused by a missing parenthesis or a brace in a JSON object, or an incorrect timestamp format in the time field.

When a Stream Analytics job receives a malformed message from an input, it drops the message and notifies you with a warning. A warning symbol is shown on the **Inputs** tile of your Stream Analytics job. This warning sign exists as long as the job is in running state:



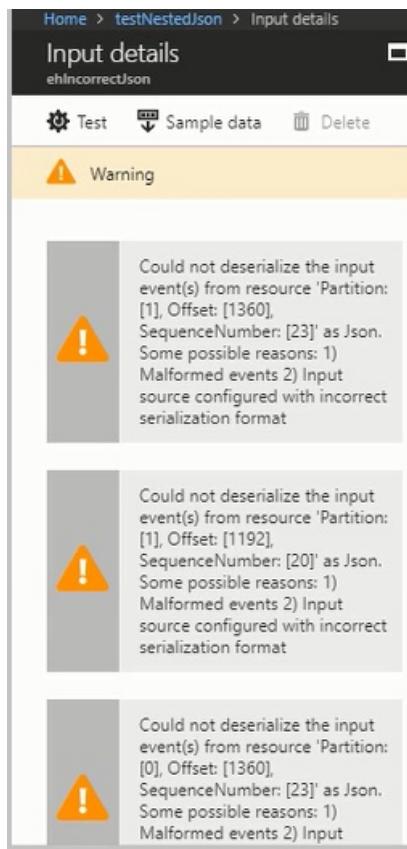
Enable the diagnostics logs to view the details of the warning. For malformed input events, the execution logs contain an entry with the message that looks like:

```
Could not deserialize the input event(s) from resource <blob URI> as json.
```

What caused the deserialization error

You can take the following steps to analyze the input events in detail to get a clear understanding of what caused the deserialization error. You can then fix the event source to generate events in the right format to prevent you from hitting this issue again.

1. Navigate to the input tile and click on the warning symbols to see the list of issues.
2. The input details tile displays a list of warnings with details about each issue. The example warning message below includes the partition, offset, and sequence numbers where there is malformed JSON data.



- To find the JSON data with the incorrect format, run the [CheckMalformedEvents.cs](#) code available in the [GitHub samples repository](#). This code reads the partition ID, offset, and prints the data that's located in that offset.
- Once you read the data, you can analyze and correct the serialization format.
- You can also [read events from an IoT Hub with the Service Bus Explorer](#).

Job exceeds maximum Event Hub Receivers

A best practice for using Event Hubs is to use multiple consumer groups to ensure job scalability. The number of readers in the Stream Analytics job for a specific input affects the number of readers in a single consumer group. The precise number of receivers is based on internal implementation details for the scale-out topology logic and is not exposed externally. The number of readers can change when a job is started or during job upgrades.

The error shown when the number of receivers exceeds the maximum is:

```
The streaming job failed: Stream Analytics job has validation errors: Job will exceed the maximum amount of Event Hub Receivers.
```

NOTE

When the number of readers changes during a job upgrade, transient warnings are written to audit logs. Stream Analytics jobs automatically recover from these transient issues.

Add a consumer group in Event Hubs

To add a new consumer group in your Event Hubs instance, follow these steps:

- Sign in to the Azure portal.
- Locate your Event Hubs.
- Select **Event Hubs** under the **Entities** heading.
- Select the Event Hub by name.

5. On the **Event Hubs Instance** page, under the **Entities** heading, select **Consumer groups**. A consumer group with name **\$Default** is listed.
6. Select **+ Consumer Group** to add a new consumer group.

NAME	LOCATION
\$Default	Central US
consumergroup1	Central US

7. When you created the input in the Stream Analytics job to point to the Event Hub, you specified the consumer group there. **\$Default** is used when none is specified. Once you create a new consumer group, edit the Event Hub input in the Stream Analytics job and specify the name of the new consumer group.

Readers per partition exceeds Event Hubs limit

If your streaming query syntax references the same input Event Hub resource multiple times, the job engine can use multiple readers per query from that same consumer group. When there are too many references to the same consumer group, the job can exceed the limit of five and thrown an error. In those circumstances, you can further divide by using multiple inputs across multiple consumer groups using the solution described in the following section.

Scenarios in which the number of readers per partition exceeds the Event Hubs limit of five include the following:

- Multiple SELECT statements: If you use multiple SELECT statements that refer to **same** event hub input, each SELECT statement causes a new receiver to be created.
- UNION: When you use a UNION, it's possible to have multiple inputs that refer to the **same** event hub and consumer group.
- SELF JOIN: When you use a SELF JOIN operation, it's possible to refer to the **same** event hub multiple times.

The following best practices can help mitigate scenarios in which the number of readers per partition exceeds the Event Hubs limit of five.

Split your query into multiple steps by using a WITH clause

The WITH clause specifies a temporary named result set that can be referenced by a FROM clause in the query. You define the WITH clause in the execution scope of a single SELECT statement.

For example, instead of this query:

```
SELECT foo  
INTO output1  
FROM inputEventHub  
  
SELECT bar  
INTO output2  
FROM inputEventHub  
...
```

Use this query:

```
WITH data AS (  
    SELECT * FROM inputEventHub  
)  
  
SELECT foo  
INTO output1  
FROM data  
  
SELECT bar  
INTO output2  
FROM data  
...
```

Ensure that inputs bind to different consumer groups

For queries in which three or more inputs are connected to the same Event Hubs consumer group, create separate consumer groups. This requires the creation of additional Stream Analytics inputs.

Get help

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Troubleshoot Azure Stream Analytics outputs

6/23/2019 • 6 minutes to read • [Edit Online](#)

This page describes common issues with output connections and how to troubleshoot and address them.

Output not produced by job

1. Verify connectivity to outputs by using the **Test Connection** button for each output.
2. Look at **Monitoring Metrics** on the **Monitor** tab. Because the values are aggregated, the metrics are delayed by a few minutes.
 - If Input Events > 0, the job is able to read input data. If Input Events is not > 0, then:
 - To see whether the data source has valid data, check it by using [Service Bus Explorer](#). This check applies if the job is using Event Hub as input.
 - Check to see whether the data serialization format and data encoding are as expected.
 - If the job is using an Event Hub, check to see whether the body of the message is *Null*.
 - If Data Conversion Errors > 0 and climbing, the following might be true:
 - The output event does not conform to the schema of the target sink.
 - The event schema might not match the defined or expected schema of the events in the query.
 - The datatypes of some of the fields in the event might not match expectations.
 - If Runtime Errors > 0, it means that the job can receive the data but is generating errors while processing the query.
 - To find the errors, go to the [Audit Logs](#) and filter on *Failed* status.
 - If InputEvents > 0 and OutputEvents = 0, it means that one of the following is true:
 - Query processing resulted in zero output events.
 - Events or its fields might be malformed, resulting in zero output after query processing.
 - The job was unable to push data to the output sink for connectivity or authentication reasons.
 - In all the previously mentioned error cases, operations log messages explain additional details (including what is happening), except in cases where the query logic filtered out all events. If the processing of multiple events generates errors, Stream Analytics logs the first three error messages of the same type within 10 minutes to Operations logs. It then suppresses additional identical errors with a message that reads "Errors are happening too rapidly, these are being suppressed."

Job output is delayed

First output is delayed

When a Stream Analytics job is started, the input events are read, but there can be a delay in the output being produced in certain circumstances.

Large time values in temporal query elements can contribute to the output delay. To produce correct output over the large time windows, the streaming job starts up by reading data from the latest time possible (up to seven days ago) to fill the time window. During that time, no output is produced until the catch-up read of the outstanding input events is complete. This problem can surface when the system upgrades the streaming jobs, thus restarting the job. Such upgrades generally occur once every couple of months.

Therefore, use discretion when designing your Stream Analytics query. If you use a large time window (more than

several hours, up to seven days) for temporal elements in the job's query syntax, it can lead to a delay on the first output when the job is started or restarted.

One mitigation for this kind of first output delay is to use query parallelization techniques (partitioning the data), or add more Streaming Units to improve the throughput until the job catches up. For more information, see [Considerations when creating Stream Analytics jobs](#)

These factors impact the timeliness of the first output that is generated:

1. Use of windowed aggregates (GROUP BY of Tumbling, Hopping, and Sliding windows)
 - For tumbling or hopping window aggregates, results are generated at the end of the window timeframe.
 - For a sliding window, the results are generated when an event enters or exits the sliding window.
 - If you are planning to use large window size (> 1 hour), it's best to choose hopping or sliding window so that you can see the output more frequently.
2. Use of temporal joins (JOIN with DATEDIFF)
 - Matches are generated as soon as when both sides of the matched events arrive.
 - Data that lacks a match (LEFT OUTER JOIN) is generated at the end of the DATEDIFF window with respect to each event on the left side.
3. Use of temporal analytic functions (ISFIRST, LAST, and LAG with LIMIT DURATION)
 - For analytic functions, the output is generated for every event, there is no delay.

Output falls behind

During normal operation of the job, if you find the job's output is falling behind (longer and longer latency), you can pinpoint the root causes by examining these factors:

- Whether the downstream sink is throttled
- Whether the upstream source is throttled
- Whether the processing logic in the query is compute-intensive

To see those details, in the Azure portal, select the streaming job, and select the **Job diagram**. For each input, there is a per partition backlog event metric. If the backlog event metric keeps increasing, it's an indicator that the system resources are constrained. Potentially that is due to of output sink throttling, or high CPU. For more information on using the job diagram, see [Data-driven debugging by using the job diagram](#).

Key violation warning with Azure SQL Database output

When you configure Azure SQL database as output to a Stream Analytics job, it bulk inserts records into the destination table. In general, Azure stream analytics guarantees [at least once delivery](#) to the output sink, one can still [achieve exactly-once delivery](#) to SQL output when SQL table has a unique constraint defined.

Once unique key constraints are set up on the SQL table, and there are duplicate records being inserted into SQL table, Azure Stream Analytics removes the duplicate record. It splits the data into batches and recursively inserting the batches until a single duplicate record is found. If the streaming job has a considerable number of duplicate rows, this split and insert process has to ignore the duplicates one by one, which is less efficient and time-consuming. If you see multiple key violation warning messages in your Activity log within the past hour, it's likely that your SQL output is slowing down the entire job.

To resolve this issue, you should [configure the index](#) that is causing the key violation by enabling the IGNORE_DUP_KEY option. Enabling this option allows duplicate values to be ignored by SQL during bulk inserts and SQL Azure simply produces a warning message instead of an error. Azure Stream Analytics does not produce primary key violation errors anymore.

Note the following observations when configuring IGNORE_DUP_KEY for several types of indexes:

- You cannot set IGNORE_DUP_KEY on a primary key or a unique constraint that uses ALTER INDEX, you need to drop and recreate the index.
- You can set the IGNORE_DUP_KEY option using ALTER INDEX for a unique index, which is different from PRIMARY KEY/UNIQUE constraint and created using CREATE INDEX or INDEX definition.
- IGNORE_DUP_KEY doesn't apply to column store indexes because you can't enforce uniqueness on such indexes.

Column names are lower-cased by Azure Stream Analytics

When using the original compatibility level (1.0), Azure Stream Analytics used to change column names to lower case. This behavior was fixed in later compatibility levels. In order to preserve the case, we advise customers to move to the compatibility level 1.1 and later. You can find more information on [Compatibility level for Azure Stream Analytics jobs](#).

Get help

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Troubleshoot Azure Stream Analytics queries

12/10/2018 • 3 minutes to read • [Edit Online](#)

This article describes common issues with developing Stream Analytics queries and how to troubleshoot them.

Query is not producing expected output

1. Examine errors by testing locally:

- On the **Query** tab, select **Test**. Use the downloaded sample data to [test the query](#). Examine any errors and attempt to correct them.
- You can also [test your query directly on live input](#) using Stream Analytics tools for Visual Studio.

2. If you use **Timestamp By**, verify that the events have timestamps greater than the [job start time](#).

3. Eliminate common pitfalls, such as:

- A **WHERE** clause in the query filtered out all events, preventing any output from being generated.
- A **CAST** function fails, causing the job to fail. To avoid type cast failures, use **TRY_CAST** instead.
- When you use window functions, wait for the entire window duration to see an output from the query.
- The timestamp for events precedes the job start time and, therefore, events are being dropped.

4. Ensure event ordering policies are configured as expected. Go to the **Settings** and select **Event Ordering**.

The policy is *not* applied when you use the **Test** button to test the query. This result is one difference between testing in-browser versus running the job in production.

5. Debug by using audit and diagnostic logs:

- Use [Audit Logs](#), and filter to identify and debug errors.
- Use [job diagnostic logs](#) to identify and debug errors.

Job is consuming too many Streaming Units

Ensure you take advantage of parallelization in Azure Stream Analytics. You can learn to [scale with query parallelization](#) of Stream Analytics jobs by configuring input partitions and tuning the analytics query definition.

Debug queries progressively

In real-time data processing, knowing what the data looks like in the middle of the query can be helpful. Because inputs or steps of an Azure Stream Analytics job can be read multiple times, you can write extra `SELECT INTO` statements. Doing so outputs intermediate data into storage and lets you inspect the correctness of the data, just as *watch variables* do when you debug a program.

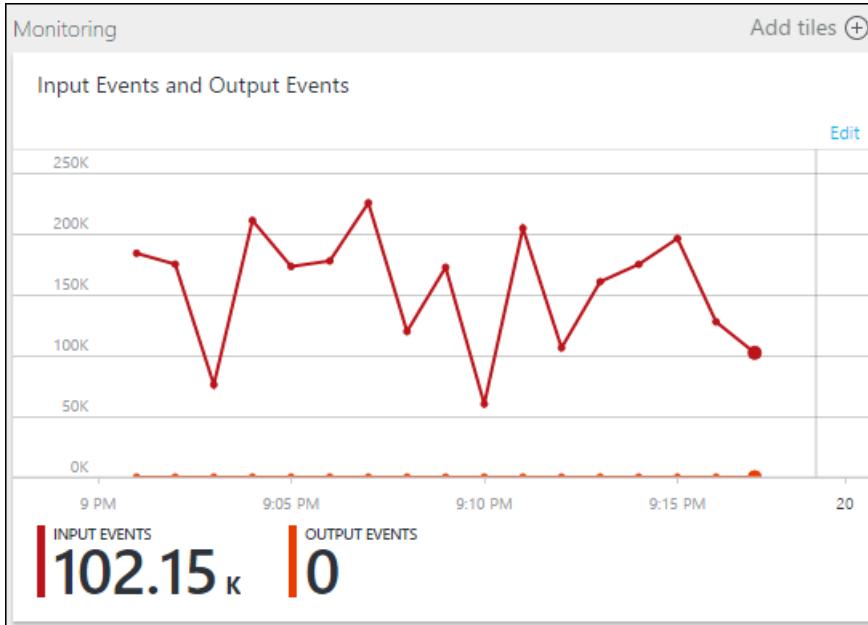
The following example query in an Azure Stream Analytics job has one stream input, two reference data inputs, and an output to Azure Table Storage. The query joins data from the event hub and two reference blobs to get the name and category information:

```

1 SELECT
2     ref1.name AS name,
3     ref2.category AS category,
4     input1.[from],
5     input1.time,
6     input1.temp,
7     input1.hmdt
8 INTO output1
9 FROM input1
10 JOIN ref1 ON input1.[from] = ref1.[from]
11 JOIN ref2 ON input1.[from] = ref2.[from]

```

Note that the job is running, but no events are being produced in the output. On the **Monitoring** tile, shown here, you can see that the input is producing data, but you don't know which step of the **JOIN** caused all the events to be dropped.



In this situation, you can add a few extra SELECT INTO statements to "log" the intermediate JOIN results and the data that's read from the input.

In this example, we've added two new "temporary outputs." They can be any sink you like. Here we use Azure Storage as an example:

The figure is a screenshot of the "Outputs" blade in a cloud service configuration interface. The title bar says "Outputs" and "DebugJobSample". There is a "Add" button. The main table lists three outputs:

NAME	SINK	...
output1	Table storage	...
temp1	Table storage	...
temp2	Table storage	...

You can then rewrite the query like this:

```

1 WITH Step1 AS
2 (
3     SELECT
4         ref1.name AS name,
5         input1.[from],
6         input1.time,
7         input1.temp,
8         input1.hmdt
9     FROM input1
10    INNER JOIN ref1 ON input1.[from] = ref1.[from]
11 ),
12 Step2 AS
13 (
14     SELECT
15         ref2.category AS category,
16         Step1.name,
17         Step1.[from],
18         Step1.time,
19         Step1.temp,
20         Step1.hmdt
21     FROM Step1
22    INNER JOIN ref2 ON Step1.[from] = ref2.[from]
23 )
24
25 -- Normal data flow into output1
26 SELECT *
27 INTO output1
28 FROM Step2
29
30 -- Log input data into temp1
31 SELECT *
32 INTO temp1
33 FROM input1
34
35 -- Log data after 1st JOIN into temp2
36 SELECT *
37 INTO temp2
38 FROM Step1

```

Now start the job again, and let it run for a few minutes. Then query temp1 and temp2 with Visual Studio Cloud Explorer to produce the following tables:

temp1 table

temp1 [Table] temp2 [Table]									
Enter a WCF Data Services filter to limit the entities returned									
The filter returned a large number of entities. Narrow the filter or click here to download the remaining entities.									
PartitionKey	RowKey	Timestamp	EventEnqueuedUtc	EventProcessedUtc	Partition	from	time	temp	
00d57097-860...	2016-05-10T01...	5/10/2016 22:52:29	5/10/2016 22:42...	5/10/2016 22:45...	0	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:42...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 22:46:13	5/10/2016 22:42...	5/10/2016 22:44...	1	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:43...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 22:48:03	5/10/2016 22:42...	5/10/2016 22:44...	1	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:43...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 22:48:03	5/10/2016 22:42...	5/10/2016 22:44...	1	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:43...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 22:48:03	5/10/2016 22:42...	5/10/2016 22:44...	1	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:43...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 22:48:03	5/10/2016 22:42...	5/10/2016 22:44...	1	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:43...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 22:50:01	5/10/2016 22:42...	5/10/2016 22:44...	3	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:43...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 22:50:09	5/10/2016 22:42...	5/10/2016 22:44...	1	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:43...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 23:02:00	5/10/2016 22:42...	5/10/2016 22:56...	1	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:44...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 23:09:31	5/10/2016 22:42...	5/10/2016 22:50...	3	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:44...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 23:09:31	5/10/2016 22:42...	5/10/2016 22:50...	3	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:44...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 23:02:00	5/10/2016 22:42...	5/10/2016 22:56...	1	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:44...	50	
00d57097-860...	2016-05-10T01...	5/10/2016 23:02:00	5/10/2016 22:42...	5/10/2016 22:56...	1	00d57097-8609-4c73-88fb-36de7f882782	5/10/2016 01:44...	50	

temp2 table

temp1 [Table] temp2 [Table]

Enter a WCF Data Services filter to limit the entities returned

The filter returned a large number of entities. Narrow the filter or [click here](#) to download the remaining entities.

PartitionKey	RowKey	Timestamp	from	hrndt	name	temp	time
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:07...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:41...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:08...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:41...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:10...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:41...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:04...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:42...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:06...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:42...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:06...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:42...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:06...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:42...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:06...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:42...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:06...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:43...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:06...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:43...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:08...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:43...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:09...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:43...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:04...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:43...
00d57097-8609-...	2016-05-10T01...	5/11/2016 00:04...	00d57097-8609-4c73-88fb-36de7f882782	30	Conf Room 34/...	50	5/10/2016 01:43...

As you can see, temp1 and temp2 both have data, and the name column is populated correctly in temp2. However, because there is still no data in output, something is wrong:

output1 [Table] temp1 [Table] temp2 [Table]

Enter a WCF Data Services filter to limit the entities returned

Table contains no entities. Start using the table by selecting the Add Entity button.

PartitionKey	RowKey	Timestamp

By sampling the data, you can be almost certain that the issue is with the second JOIN. You can download the reference data from the blob and take a look:

refdata2.json ref2 [Container] output1 [Table] temp1 [Table] temp2 [Table]

<No Schema Selected>

refdata2.json	ref2 [Container]	output1 [Table]	temp1 [Table]	temp2 [Table]
<No Schema Selected>				
<pre>{ "from": "{ad143ee2-44bc-401a-9eb9-2058368581bc}", "category": "12 People Conference Room", "from": "{0d13f1aa-1644-464c-a410-9dcfb59a8441}", "category": "10 People Conference Room", "from": "{00d57097-8609-4c73-88fb-36de7f882782}", "category": "8 People Conference Room", "from": "{e56bbde3-345d-4482-9db3-d9b1be389ee1}", "category": "8 People Conference Room", "from": "{0ff1aa5d-3bb4-4d1f-b06f-108732e659de}", "category": "12 People Conference Room", "from": "{ac7f271a-f062-401d-b56f-44d34d0fed68}", "category": "8 People Conference Room", "from": "{efba6953-6b65-418f-bff8-4f16ce5f2467}", "category": "6 People Conference Room", "from": "{4b7219c3-4cc4-400f-98a3-bf0bdae38ad9}", "category": "6 People Conference Room" }</pre>				

As you can see, the format of the GUID in this reference data is different from the format of the [from] column in temp2. That's why the data didn't arrive in output1 as expected.

You can fix the data format, upload it to reference blob, and try again:

```

refdata2.json ✎ X output1 [Table] temp1 [Table] temp2 [Table]
<No Schema Selected>
[{"from": "ad143ee2-44bc-401a-9eb9-2058368581bc", "category": "12 People Conference Room"}, {"from": "0d13f1aa-1644-464c-a410-9dcbf59a8441", "category": "10 People Conference Room"}, {"from": "00d57097-8609-4c73-88fb-36de7f882782", "category": "8 People Conference Room"}, {"from": "e56bbde3-345d-4482-9db3-d9b1be389ee1", "category": "8 People Conference Room"}, {"from": "0ff1aa5d-3bb4-4d1f-b06f-108732e659de", "category": "12 People Conference Room"}, {"from": "ac7f271a-f062-401d-b56f-44d34d0fed68", "category": "8 People Conference Room"}, {"from": "efba6953-6b65-418f-bff8-4f16ce5f2467", "category": "6 People Conference Room"}, {"from": "4b7219c3-4cc4-400f-98a3-bf0bdae38ad9", "category": "6 People Conference Room"}]

```

This time, the data in the output is formatted and populated as expected.

PartitionKey	RowKey	Timestamp	from	category	name	hmdt	temp	time
00d57097-8609...	2016-05-10T01...	5/11/2016 01:05...	00d57097-8609...	8 People Conference Room	Conf Room 34/1349...	30	50	5/10/2016 01
00d57097-8609...	2016-05-10T01...	5/11/2016 01:05...	00d57097-8609...	8 People Conference Room	Conf Room 34/1349...	30	50	5/10/2016 01
00d57097-8609...	2016-05-10T01...	5/11/2016 01:05...	00d57097-8609...	8 People Conference Room	Conf Room 34/1349...	30	50	5/10/2016 01
00d57097-8609...	2016-05-10T01...	5/11/2016 01:05...	00d57097-8609...	8 People Conference Room	Conf Room 34/1349...	30	50	5/10/2016 01
00d57097-8609...	2016-05-10T01...	5/11/2016 01:05...	00d57097-8609...	8 People Conference Room	Conf Room 34/1349...	30	50	5/10/2016 01
00d57097-8609...	2016-05-10T01...	5/11/2016 01:05...	00d57097-8609...	8 People Conference Room	Conf Room 34/1349...	30	50	5/10/2016 01
00d57097-8609...	2016-05-10T01...	5/11/2016 01:04...	00d57097-8609...	8 People Conference Room	Conf Room 34/1349...	30	50	5/10/2016 01
00d57097-8609...	2016-05-10T01...	5/11/2016 01:04...	00d57097-8609...	8 People Conference Room	Conf Room 34/1349...	30	50	5/10/2016 01
00d57097-8609...	2016-05-10T01...	5/11/2016 01:04...	00d57097-8609...	8 People Conference Room	Conf Room 34/1349...	30	50	5/10/2016 01

Get help

For further assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Troubleshoot Azure Stream Analytics by using diagnostics logs

6/21/2019 • 6 minutes to read • [Edit Online](#)

Occasionally, an Azure Stream Analytics job unexpectedly stops processing. It's important to be able to troubleshoot this kind of event. Failures can be caused by an unexpected query result, by connectivity to devices, or by an unexpected service outage. The diagnostics logs in Stream Analytics can help you identify the cause of issues when they occur and reduce recovery time.

It is highly recommended to enable diagnostic logs for all production jobs.

Log types

Stream Analytics offers two types of logs:

- [Activity logs](#) (always on), which give insights into operations performed on jobs.
- [Diagnostics logs](#) (configurable), which provide richer insights into everything that happens with a job.
Diagnostics logs start when the job is created and end when the job is deleted. They cover events when the job is updated and while it's running.

NOTE

You can use services like Azure Storage, Azure Event Hubs, and Azure Monitor logs to analyze nonconforming data. You are charged based on the pricing model for those services.

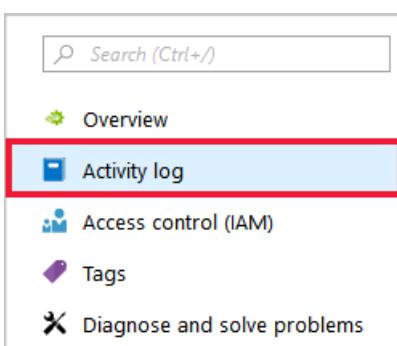
NOTE

This article was recently updated to use the term Azure Monitor logs instead of Log Analytics. Log data is still stored in a Log Analytics workspace and is still collected and analyzed by the same Log Analytics service. We are updating the terminology to better reflect the role of [logs in Azure Monitor](#). See [Azure Monitor terminology changes](#) for details.

Debugging using activity logs

Activity logs are on by default and give high-level insights into operations performed by your Stream Analytics job. Information present in activity logs may help find the root cause of the issues impacting your job. Do the following steps to use activity logs in Stream Analytics:

1. Sign in to the Azure portal and select **Activity log** under **Overview**.



2. You can see a list of operations that have been performed. Any operation that caused your job to fail has a red info bubble.
3. Click an operation to see its summary view. Information here is often limited. To learn more details about the operation, click **JSON**.

```

1  {
2      "channels": "Operation",
3      "correlationId": "40b413a0-f541-474e-8153-3beef3acb863",
4      "description": "",
5      "eventDataId": "dd3989d9-84d2-426e-a8b6-aca02234db3f",
6      "eventName": {
7          "value": "",
8          "localizedValue": ""
9      },

```

4. Scroll down to the **Properties** section of the JSON, which provides details of the error that caused the failed operation. In this example, the failure was due to a runtime error from out of bound latitude values. Discrepancy in the data that is processed by a Stream Analytics job causes a data error. You can learn about different [input and output data errors](#) and why they occur.

```

42     "properties": {
43         "Message Time": "2018-12-18 18:13:53Z",
44         "Error": "- Latitude must be within -90 and 90 degrees.\r\n",
45         "Message": "Runtime exception occurred while processing events, - Latitude must be within -90 and 90 degrees.\r\n, : OutputSourceAlias:pbicounty;",
46         "Type": "SqlRuntimeError",
47         "Correlation ID": "40b413a0-f541-474e-8153-3beef3acb863"
48     },
49     "relatedEvents": []
50 }

```

5. You can take corrective actions based on the error message in JSON. In this example, checks to ensure latitude value is between -90 degrees and 90 degrees need to be added to the query.
6. If the error message in the Activity logs isn't helpful in identifying root cause, enable diagnostic logs and use Azure Monitor logs.

Send diagnostics to Azure Monitor logs

Turning on diagnostic logs and sending them to Azure Monitor logs is highly recommended. Diagnostics logs are **off** by default. To turn on diagnostics logs, complete these steps:

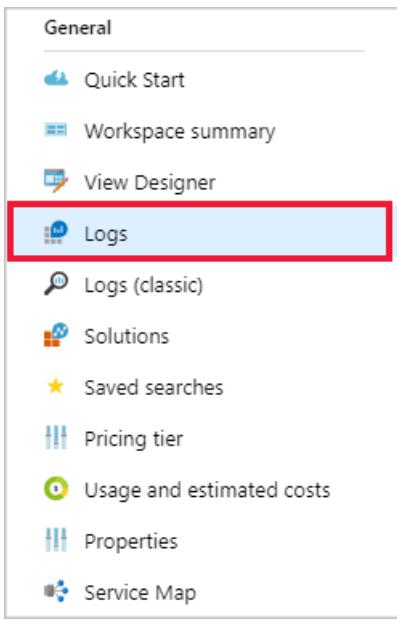
1. Sign in to the Azure portal, and navigate to your Stream Analytics job. Under **Monitoring**, select **Diagnostics logs**. Then select **Turn on diagnostics**.

The screenshot shows the 'Diagnostics logs' section of the Stream Analytics job configuration. On the left, there's a sidebar with various monitoring and support options. The 'Diagnostics logs' option is selected and highlighted with a red box. At the top, there are filters for Subscription, Resource group, Resource type, and Resource, along with a search bar and refresh button.

2. Create a **Name** in **Diagnostic settings** and check the box next to **Send to Log Analytics**. Then add an existing or create a new **Log analytics workspace**. Check the boxes for **Execution** and **Authoring** under **LOG**, and **AllMetrics** under **METRIC**. Click **Save**. It is recommended to use a Log Analytics workspace in the same Azure region as your Stream Analytics job to prevent additional costs.

The screenshot displays three windows side-by-side. The leftmost window is 'Diagnostics settings' with 'Send to Log Analytics' checked. The middle window is 'Log Analytics Workspaces' with 'Create New Workspace' highlighted. The rightmost window is a detailed 'Log Analytics workspace' configuration dialog where 'Create New' is selected, and a workspace named 'asaomgworkspace' is chosen. Other fields include 'Subscription' (asa_demo), 'Resource group' (asa_demo), 'Location' (West US 2), and 'Pricing tier' (Per GB). A blue 'OK' button is at the bottom right of the configuration dialog.

3. When your Stream Analytics job starts, diagnostic logs are routed to your Log Analytics workspace. Navigate to the Log Analytics workspace and choose **Logs** under the **General** section.



4. You can [write your own query](#) to search for terms, identify trends, analyze patterns, and provide insights based on your data. For example, you can write a query to filter only diagnostic logs that have the message "The streaming job failed." Diagnostic logs from Azure Stream Analytics are stored in the **AzureDiagnostics** table.

The screenshot shows the Azure Stream Analytics query editor. At the top, there is a 'Run' button and a 'Time range: Last 24 hours' dropdown. Below the editor area, it says 'Completed. Showing results from the last 24 hours.' and indicates '00:00:00.251' and '7 records'. The display time is set to '(UTC+00:00)'. The query itself is: `AzureDiagnostics
| where parse_json(properties_s).JobFailedMessage == "The streaming job failed."`. The results table below shows 7 records with columns: SourceSystem, MG, ManagementGroupName, TimeGenerated [UTC], Computer, Environment_s, Region_s, and OperationName. The data includes entries for Azure starting at 2018-12-18T18:13:55.710 and ending at 2018-12-18T18:09:31.100, with various operations like 'Start job', 'Process Events', and 'Receive Events'.

SourceSystem	MG	ManagementGroupName	TimeGenerated [UTC]	Computer	Environment_s	Region_s	OperationName
Azure			2018-12-18T18:13:55.710		Prod	West US 2	Start job
Azure			2018-12-18T18:13:53.933		Prod	West US 2	Process Events: Processing events Runti
Azure			2018-12-18T18:12:48.184		Prod	West US 2	Receive Events
Azure			2018-12-18T18:12:47.806		Prod	West US 2	Receive Events
Azure			2018-12-18T18:12:37.003		Prod	West US 2	Start job
Azure			2018-12-18T18:09:31.288		Prod	West US 2	Start streaming job
Azure			2018-12-18T18:09:31.100		Prod	West US 2	Start streaming job

5. When you have a query that is searching for the right logs, save it by selecting **Save** and provide a Name and Category. You can then create an alert by selecting **New alert rule**. Next, specify the alert condition. Select **Condition** and enter the threshold value and the frequency at which this custom log search is evaluated.

* Search query [?](#)

```
AzureDiagnostics
| where parse_json(properties_s).JobFailedMessage == "The streaming job failed."
```

Query to be executed : AzureDiagnostics | where parse_json(properties_s).JobFailedMessage == "The streaming job failed." [| count](#)
For time window : 12/18/2018, 1:34:41 PM - 12/18/2018, 1:39:41 PM

Alert logic

Based on ?	Condition ?	* Threshold ?
Number of results	Greater than	1

Condition preview

Whenever the custom log search is greater than 1 count

Evaluated based on

* Period (in minutes) ?	* Frequency (in minutes) ?
5	5

Done

- Choose the action group and specify alert details, like name and description, before you can create the alert rule. You can route the diagnostic logs of various jobs to the same Log Analytics workspace. This allows you to set up alerts once that work across all jobs.

Diagnostics log categories

Azure Stream Analytics captures two categories of diagnostics logs:

- Authoring:** Captures log events that are related to job authoring operations, such as job creation, adding and deleting inputs and outputs, adding and updating the query, and starting or stopping the job.
- Execution:** Captures events that occur during job execution.
 - Connectivity errors
 - Data processing errors, including:
 - Events that don't conform to the query definition (mismatched field types and values, missing fields, and so on)
 - Expression evaluation errors
 - Other events and errors

Diagnostics logs schema

All logs are stored in JSON format. Each entry has the following common string fields:

NAME	DESCRIPTION
time	Timestamp (in UTC) of the log.

NAME	DESCRIPTION
resourceId	ID of the resource that the operation took place on, in upper case. It includes the subscription ID, the resource group, and the job name. For example, /SUBSCRIPTIONS/6503D296-DAC1-4449-9B03-609A1F4A1C87/RESOURCEGROUPS/MY-RESOURCE-GROUP/PROVIDERS/MICROSOFT.STREAMANALYTICS/STREAMINGJOBS/MYSTREAMINGJOB .
category	Log category, either Execution or Authoring .
operationName	Name of the operation that is logged. For example, Send Events: SQL Output write failure to mysqloutput .
status	Status of the operation. For example, Failed or Succeeded .
level	Log level. For example, Error , Warning , or Informational .
properties	Log entry-specific detail, serialized as a JSON string. For more information, see the following sections in this article.

Execution log properties schema

Execution logs have information about events that happened during Stream Analytics job execution. The schema of properties varies depending on whether the event is a data error or a generic event.

Data errors

Any error that occurs while the job is processing data is in this category of logs. These logs most often are created during data read, serialization, and write operations. These logs do not include connectivity errors. Connectivity errors are treated as generic events. You can learn more about the cause of various different [input and output data errors](#).

NAME	DESCRIPTION
Source	Name of the job input or output where the error occurred.
Message	Message associated with the error.
Type	Type of error. For example, DataConversionError , CsvParserError , or ServiceBusPropertyColumnMissingError .
Data	Contains data that is useful to accurately locate the source of the error. Subject to truncation, depending on size.

Depending on the **operationName** value, data errors have the following schema:

- **Serialize events** occur during event read operations. They occur when the data at the input does not satisfy the query schema for one of these reasons:
 - *Type mismatch during event (de)serialize*: Identifies the field that's causing the error.
 - *Cannot read an event, invalid serialization*: Lists information about the location in the input data where the error occurred. Includes blob name for blob input, offset, and a sample of the data.
- **Send events** occur during write operations. They identify the streaming event that caused the error.

Generic events

Generic events cover everything else.

NAME	DESCRIPTION
Error	(optional) Error information. Usually, this is exception information if it's available.
Message	Log message.
Type	Type of message. Maps to internal categorization of errors. For example, JobValidationException or BlobOutputAdapterInitializationFailure .
Correlation ID	GUID that uniquely identifies the job execution. All execution log entries from the time the job starts until the job stops have the same Correlation ID value.

Next steps

- [Introduction to Stream Analytics](#)
- [Get started with Stream Analytics](#)
- [Scale Stream Analytics jobs](#)
- [Stream Analytics query language reference](#)
- [Stream Analytics data errors](#)

Azure Stream Analytics data errors

6/21/2019 • 6 minutes to read • [Edit Online](#)

Data errors are errors that occur while processing the data. These errors most often occur during data de-serialization, serialization, and write operations. When data errors occur, Stream Analytics writes detailed information and example events to the diagnostic logs. In some cases, summary of this information is also provided through portal notifications.

This article outlines the different error types, causes, and diagnostic log details for input and output data errors.

Diagnostic log schema

See [Troubleshoot Azure Stream Analytics by using diagnostics logs](#) to see the schema for diagnostic logs. The following JSON is an example value for the **Properties** field of a diagnostic log for a data error.

```
{  
    "Source": "InputTelemetryData",  
    "Type": "DataError",  
    "DataErrorCode": "InputDeserializerError.InvalidData",  
    "BriefMessage": "Json input stream should either be an array of objects or line separated objects. Found token type: Integer",  
    "Message": "Input Message Id: https://\\exampleblob.blob.core.windows.net\\inputfolder\\csv.txt Error:  
Json input stream should either be an array of objects or line separated objects. Found token type: Integer",  
    "ExampleEvents": "[\"1,2\\\\u00d\\\\u00a3,4\\\\u00d\\\\u00a5,6\"]",  
    "FromTimestamp": "2019-03-22T22:34:18.5664937Z",  
    "ToTimestamp": "2019-03-22T22:34:18.5965248Z",  
    "EventCount": 1  
}
```

Input data errors

InputDeserializerError.InvalidCompressionType

- Cause: The input compression type selected doesn't match the data.
- Portal notification provided: Yes
- Diagnostic log level: Warning
- Impact: Messages with any deserialization errors including invalid compression type are dropped from the input.
- Log details
 - Input message identifier. For Event Hub, the identifier is the PartitionId, Offset, and Sequence Number.

Error message

```
"BriefMessage": "Unable to decompress events from resource  
'https://\\exampleblob.blob.core.windows.net\\inputfolder\\csv.txt'. Please ensure compression setting fits  
the data being processed."
```

InputDeserializerError.InvalidHeader

- Cause: The header of input data is invalid. For example, a CSV has columns with duplicate names.
- Portal notification provided: Yes
- Diagnostic log level: Warning

- Impact: Messages with any deserialization errors including invalid header are dropped from the input.
- Log details
 - Input message identifier.
 - Actual payload up to few kilobytes.

Error message

```
"BriefMessage": "Invalid CSV Header for resource
'https://exampleblob.blob.core.windows.net/inputfolder/csv.txt'. Please make sure there are no
duplicate field names."
```

InputDeserializerError.MissingColumns

- Cause: The input columns defined with CREATE TABLE or through TIMESTAMP BY doesn't exist.
- Portal notification provided: Yes
- Diagnostic log level: Warning
- Impact: Events with missing columns are dropped from the input.
- Log details
 - Input message identifier.
 - Names of the columns that are missing.
 - Actual payload up to a few kilobytes.

Error messages

```
"BriefMessage": "Could not deserialize the input event(s) from resource
'https://exampleblob.blob.core.windows.net/inputfolder/csv.txt' as Csv. Some possible reasons: 1)
Malformed events 2) Input source configured with incorrect serialization format"
```

```
"Message": "Missing fields specified in query or in create table. Fields expected:ColumnA Fields found:ColumnB"
```

InputDeserializerError.TypeConversionError

- Cause: Unable to convert the input to the type specified in the CREATE TABLE statement.
- Portal notification provided: Yes
- Diagnostic log level: Warning
- Impact: Events with type conversion error are dropped from the input.
- Log details
 - Input message identifier.
 - Name of the column and expected type.

Error messages

```
"BriefMessage": "Could not deserialize the input event(s) from resource
'https://exampleblob.blob.core.windows.net/inputfolder/csv.txt' as Csv. Some possible reasons: 1)
Malformed events 2) Input source configured with incorrect serialization format"
```

```
"Message": "Unable to convert column: dateColumn to expected type."
```

InputDeserializerError.InvalidData

- Cause: Input data is not in the right format. For example, the input isn't valid JSON.
- Portal notification provided: Yes

- Diagnostic log level: Warning
- Impact: All events in the message after an invalid data error has been encountered are dropped from the input.
- Log details
 - Input message identifier.
 - Actual payload up to few kilobytes.

Error messages

```
"BriefMessage": "Json input stream should either be an array of objects or line separated objects. Found token type: String"
```

```
"Message": "Json input stream should either be an array of objects or line separated objects. Found token type: String"
```

InvalidInputTimeStamp

- Cause: The value of the TIMESTAMP BY expression can't be converted to datetime.
- Portal notification provided: Yes
- Diagnostic log level: Warning
- Impact: Events with invalid input timestamp are dropped from the input.
- Log details
 - Input message identifier.
 - Error message.
 - Actual payload up to few kilobytes.

Error message

```
"BriefMessage": "Unable to get timestamp for resource 'https://exampleBlob.blob.core.windows.net/inputfolder\\csv.txt' due to error 'Cannot convert string to datetime'"
```

InvalidInputTimeStampKey

- Cause: The value of TIMESTAMP BY OVER timestampColumn is NULL.
- Portal notification provided: Yes
- Diagnostic log level: Warning
- Impact: Events with invalid input timestamp key are dropped from the input.
- Log details
 - The actual payload up to few kilobytes.

Error message

```
"BriefMessage": "Unable to get value of TIMESTAMP BY OVER COLUMN"
```

LateInputEvent

- Cause: The difference between application time and arrival time is greater than late arrival tolerance window.
- Portal notification provided: No
- Diagnostic log level: Information
- Impact: Late input events are handled according to the "Handle other events" setting in the Event Ordering section of the job configuration. For more information see [Time Handling Policies](#).
- Log details

- Application time and arrival time.
- Actual payload up to few kilobytes.

Error message

```
"BriefMessage": "Input event with application timestamp '2019-01-01' and arrival time '2019-01-02' was sent later than configured tolerance."
```

EarlyInputEvent

- Cause: The difference between Application time and Arrival time is greater than 5 minutes.
- Portal notification provided: No
- Diagnostic log level: Information
- Impact: Early input events are handled according to the "Handle other events" setting in the Event Ordering section of the job configuration. For more information see [Time Handling Policies](#).
- Log details
 - Application time and arrival time.
 - Actual payload up to few kilobytes.

Error message

```
"BriefMessage": "Input event arrival time '2019-01-01' is earlier than input event application timestamp '2019-01-02' by more than 5 minutes."
```

OutOfOrderEvent

- Cause: Event is considered out of order according to the out of order tolerance window defined.
- Portal notification provided: No
- Diagnostic log level: Information
- Impact: Out of order events are handled according to the "Handle other events" setting in the Event Ordering section of the job configuration. For more information see [Time Handling Policies](#).
- Log details
 - Actual payload up to few kilobytes.

Error message

```
"Message": "Out of order event(s) received."
```

Output data errors

OutputDataConversionError.RequiredColumnMissing

- Cause: The column required for the output doesn't exist. For example, a column defined as Azure Table PartitionKey doesn't exist.
- Portal notification provided: Yes
- Diagnostic log level: Warning
- Impact: All output data conversion errors including missing required column are handled according to the [Output Data Policy](#) setting.
- Log details
 - Name of the column and either the record identifier or part of the record.

Error message

```
"Message": "The output record does not contain primary key property: [deviceId] Ensure the query output contains the column [deviceId] with a unique non-empty string less than '255' characters."
```

OutputDataConversionError.ColumnNameInvalid

- Cause: The column value doesn't conform with the output. For example, the column name isn't a valid Azure table column.
- Portal notification provided: Yes
- Diagnostic log level: Warning
- Impact: All output data conversion errors including invalid column name are handled according to the [Output Data Policy](#) setting.
- Log details
 - Name of the column and either record identifier or part of the record.

Error message

```
"Message": "Invalid property name #deviceIdValue. Please refer MSDN for Azure table property naming convention."
```

OutputDataConversionError.TypeConversionError

- Cause: A column can't be converted to a valid type in the output. For example, the value of column is incompatible with constraints or type defined in SQL table.
- Portal notification provided: Yes
- Diagnostic log level: Warning
- Impact: All output data conversion errors including type conversion error are handled according to the [Output Data Policy](#) setting.
- Log details
 - Name of the column.
 - Either record identifier or part of the record.

Error message

```
"Message": "The column [id] value null or its type is invalid. Ensure to provide a unique non-empty string less than '255' characters."
```

OutputDataConversionError.RecordExceededSizeLimit

- Cause: The value of the message is greater than the supported output size. For example, a record is larger than 1 MB for an Event Hub output.
- Portal notification provided: Yes
- Diagnostic log level: Warning
- Impact: All output data conversion errors including record exceeded size limit are handled according to the [Output Data Policy](#) setting.
- Log details
 - Either record identifier or part of the record.

Error message

```
"BriefMessage": "Single output event exceeds the maximum message size limit allowed (262144 bytes) by Event Hub."
```

OutputDataConversionError.DuplicateKey

- Cause: A record already contains a column with the same name as a System column. For example, CosmosDB output with a column named ID when ID column is to a different column.
- Portal notification provided: Yes
- Diagnostic log level: Warning
- Impact: All output data conversion errors including duplicate key are handled according to the [Output Data Policy](#) setting.
- Log details
 - Name of the column.
 - Either record identifier or part of the record.

```
"BriefMessage": "Column 'devicePartitionKey' is being mapped to multiple columns."
```

Next steps

- [Troubleshoot Azure Stream Analytics by using diagnostics logs](#)
- [Understand Stream Analytics job monitoring and how to monitor queries](#)

Performing sentiment analysis by using Azure Stream Analytics and Azure Machine Learning Studio (Preview)

6/17/2019 • 7 minutes to read • [Edit Online](#)

This article describes how to quickly set up a simple Azure Stream Analytics job that integrates Azure Machine Learning Studio. You use a Machine Learning sentiment analytics model from the Cortana Intelligence Gallery to analyze streaming text data and determine the sentiment score in real time. Using the Cortana Intelligence Suite lets you accomplish this task without worrying about the intricacies of building a sentiment analytics model.

You can apply what you learn from this article to scenarios such as these:

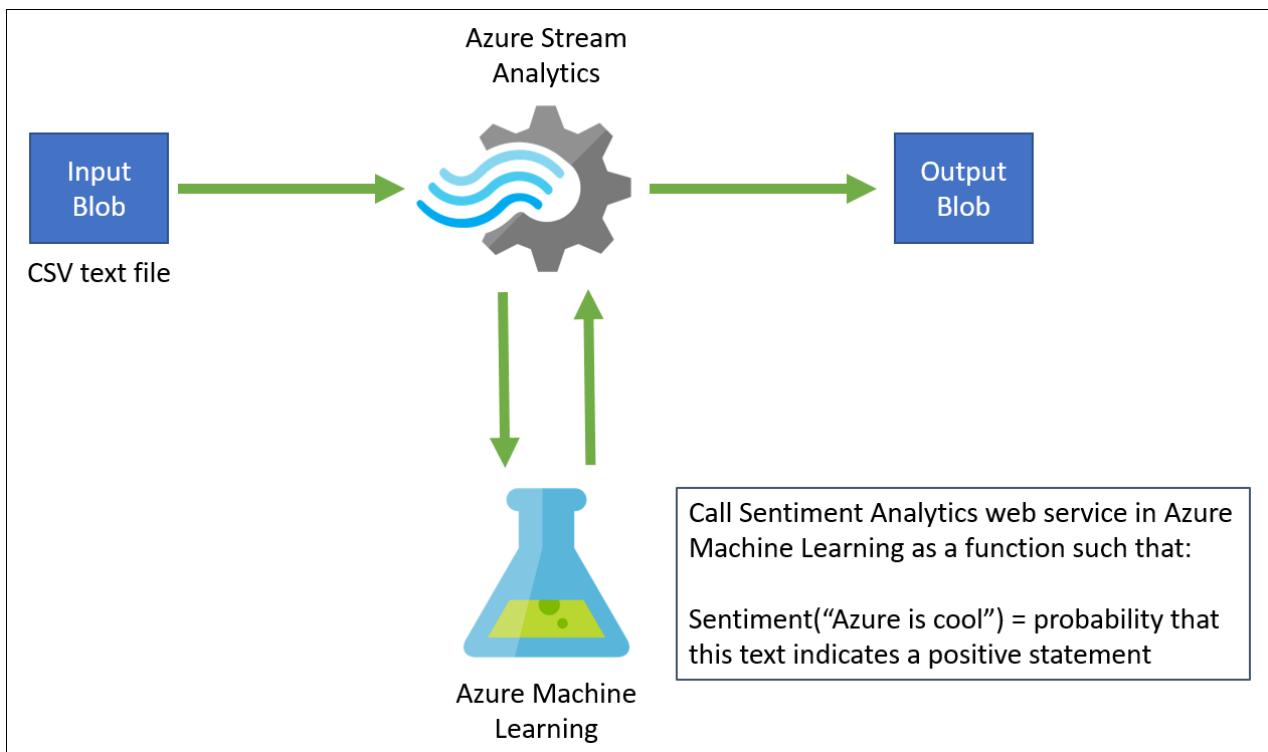
- Analyzing real-time sentiment on streaming Twitter data.
- Analyzing records of customer chats with support staff.
- Evaluating comments on forums, blogs, and videos.
- Many other real-time, predictive scoring scenarios.

In a real-world scenario, you would get the data directly from a Twitter data stream. To simplify the tutorial, it's written so that the Streaming Analytics job gets tweets from a CSV file in Azure Blob storage. You can create your own CSV file, or you can use a sample CSV file, as shown in the following image:

Text
Azure is cool
Cortana Analytics rocks
I hate going to the gym

The Streaming Analytics job that you create applies the sentiment analytics model as a user-defined function (UDF) on the sample text data from the blob store. The output (the result of the sentiment analysis) is written to the same blob store in a different CSV file.

The following figure demonstrates this configuration. As noted, for a more realistic scenario, you can replace blob storage with streaming Twitter data from an Azure Event Hubs input. Additionally, you could build a [Microsoft Power BI](#) real-time visualization of the aggregate sentiment.



Prerequisites

Before you start, make sure you have the following:

- An active Azure subscription.
- A CSV file with some data in it. You can download the file shown earlier from [GitHub](#), or you can create your own file. For this article, it is assumed that you're using the file from GitHub.

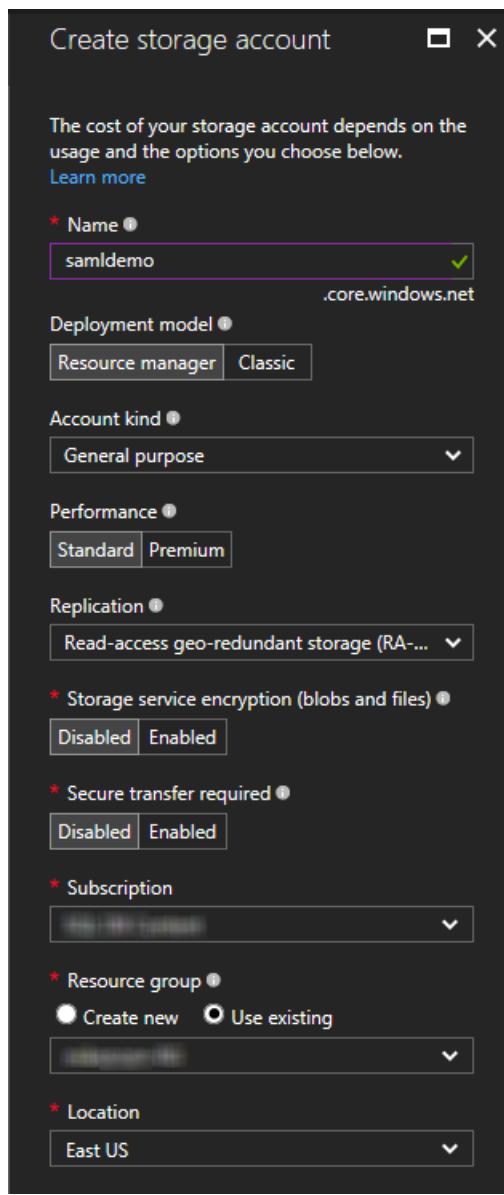
At a high level, to complete the tasks demonstrated in this article, you do the following:

1. Create an Azure storage account and a blob storage container, and upload a CSV-formatted input file to the container.
2. Add a sentiment analytics model from the Cortana Intelligence Gallery to your Azure Machine Learning Studio workspace and deploy this model as a web service in the Machine Learning workspace.
3. Create a Stream Analytics job that calls this web service as a function in order to determine sentiment for the text input.
4. Start the Stream Analytics job and check the output.

Create a storage container and upload the CSV input file

For this step, you can use any CSV file, such as the one available from GitHub.

1. In the Azure portal, click **Create a resource > Storage > Storage account**.
2. Provide a name (`sam1demo` in the example). The name can use only lowercase letters and numbers, and it must be unique across Azure.
3. Specify an existing resource group and specify a location. For location, we recommend that all the resources created in this tutorial use the same location.



4. In the Azure portal, select the storage account. In the storage account blade, click **Containers** and then click **+ Container** to create blob storage.

The screenshot shows the Azure Storage account 'samldemo' - Containers page. On the left, there's a sidebar with links like Overview, Activity log, Access control (IAM), Tags, and Diagnose and solve problems. Below that is a SETTINGS section with Access keys, Configuration, Shared access signature, Properties, Locks, and Automation script. Under BLOB SERVICE, the 'Containers' link is highlighted with a red box. At the top right, there's a '+ Container' button (also highlighted with a red box) and a Refresh button. The main area shows essentials for the storage account, including Status (Primary: Available, Secondary: Available), Location (East US, West US), Subscription (change), and Subscription ID. A search bar at the top says 'Search containers by prefix'. Below it, a table header says 'NAME' and a message says 'You don't have any containers yet. Click '+ Container' to start creating one.'

5. Provide a name for the container (`azuresamldemoblob` in the example) and verify that **Access type** is set to **Blob**. When you're done, click **OK**.

This screenshot shows the 'New container' dialog box. It has fields for 'Name' (containing 'azuresamldemoblob') and 'Access type' (set to 'Blob'). At the bottom are 'OK' and 'Cancel' buttons, with 'OK' highlighted with a red box.

6. In the **Containers** blade, select the new container, which opens the blade for that container.

7. Click **Upload**.

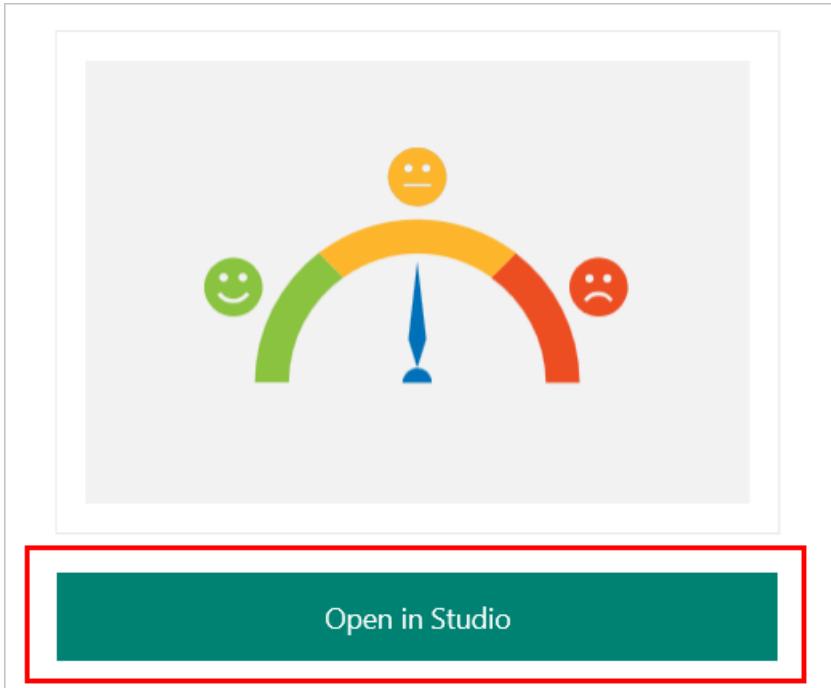
This screenshot shows the 'azuresamldemoblob' container blade. It displays the container name and has buttons for 'Upload', 'Refresh', and 'Delete container'. The 'Upload' button is highlighted with a red box.

8. In the **Upload blob** blade, upload the **sampleinput.csv** file that you downloaded earlier. For **Blob type**, select **Block blob** and set the block size to 4 MB, which is sufficient for this tutorial.
9. Click the **Upload** button at the bottom of the blade.

Add the sentiment analytics model from the Cortana Intelligence Gallery

Now that the sample data is in a blob, you can enable the sentiment analysis model in Cortana Intelligence Gallery.

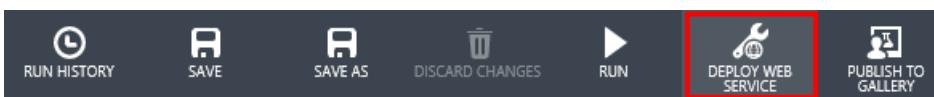
1. Go to the [predictive sentiment analytics model](#) page in the Cortana Intelligence Gallery.
2. Click **Open in Studio**.



3. Sign in to go to the workspace. Select a location.
4. Click **Run** at the bottom of the page. The process runs, which takes about a minute.



5. After the process has run successfully, select **Deploy Web Service** at the bottom of the page.



6. To validate that the sentiment analytics model is ready to use, click the **Test** button. Provide text input such as "I love Microsoft".

DASHBOARD CONFIGURATION

General New Web Services Experience [preview](#)

Published experiment

[View snapshot](#) [View latest](#)

Description

No description provided for this web service.

API key

AZfvegv6/r01fx3VBz2w/fOxortF3AhLjWr5EQwJ/NaWpBQqQLIUIpt4QYPK/fO+zTShyd5oLk/HEToxmgQLQ==

Default Endpoint

API HELP PAGE TEST

REQUEST/RESPONSE Test [Test preview](#)

BATCH EXECUTION [Test preview](#)

If the test works, you see a result similar to the following example:

✓ 'Predictive Experiment - Mini Twitter sentiment analysis' test returned ["positive", "0.715057671070099"]...

7. In the **Apps** column, click the **Excel 2010 or earlier workbook** link to download an Excel workbook. The workbook contains the API key and the URL that you need later to set up the Stream Analytics job.

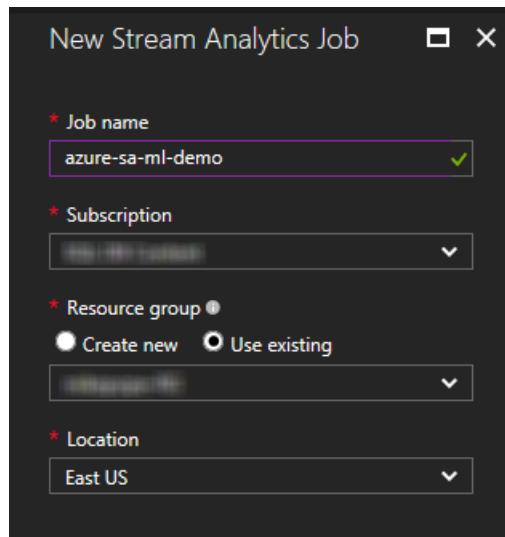
WEB SERVICE URL
ACCESS KEY

Create a Stream Analytics job that uses the Machine Learning model

You can now create a Stream Analytics job that reads the sample tweets from the CSV file in blob storage.

Create the job

1. Go to the [Azure portal](#).
2. Click **Create a resource > Internet of Things > Stream Analytics job**.
3. Name the job `azure-sa-ml-demo`, specify a subscription, specify an existing resource group or create a new one, and select the location for the job.



Configure the job input

The job gets its input from the CSV file that you uploaded earlier to blob storage.

1. After the job has been created, under **Job Topology** in the job blade, click the **Inputs** option.
2. In the **Inputs** blade, click **Add Stream Input > Blob storage**
3. Fill out the **Blob Storage** blade with these values:

FIELD	VALUE
Input alias	Use the name <code>datainput</code> and select Select blob storage from your subscription
Storage account	Select the storage account you created earlier.
Container	Select the container you created earlier (<code>azuresamldemoblob</code>)
Event serialization format	Select CSV

The screenshot shows the Azure Stream Analytics job blade. On the left, there's a sidebar with options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Locks), Job Topology (Inputs, Functions, Query, Outputs), and Configure (Scale, Locale). The 'Inputs' option under Job Topology is highlighted with a red box. The main area shows a table with one row for 'datainput1' (Stream source type). To the right, there's a detailed configuration pane for 'datainput1'. It includes fields for Input alias (set to 'datainput1'), Subscription (Microsoft Azure Internal Consumption), Storage account (set to 'sngnsa1'), Storage account key (redacted), Container (Create new selected, container name 'cont1'), Path pattern (empty), Date format (YYYY/MM/DD), Time format (HH), and Partitions (empty). A 'Save' button is at the bottom.

- Click **Save**.

Configure the job output

The job sends results to the same blob storage where it gets input.

- Under **Job Topology** in the job blade, click the **Outputs** option.
- In the **Outputs** blade, click **Add > Blob storage**, and then add an output with the alias `datamloutput`.
- Fill out the **Blob Storage** blade with these values:

FIELD	VALUE
Output alias	Use the name <code>datamloutput</code> and select Select blob storage from your subscription
Storage account	Select the storage account you created earlier.
Container	Select the container you created earlier (<code>azuresamldemoblob</code>)
Event serialization format	Select CSV

The screenshot shows the Azure Stream Analytics job 'azure-sa-ml-demo - Outputs'. On the left, there's a sidebar with 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'SETTINGS' (Locks), 'JOB TOPOLOGY' (Inputs, Functions, Query, Outputs), and 'CONFIGURE' (Scale, Locale). The 'Outputs' section is highlighted with a red box. The main area shows a 'datamoutput' output being configured. The 'NAME' field is 'datamoutput'. The 'SINK' dropdown is set to 'Blob'. The 'Output alias' is 'datamoutput1'. The 'Provide Blob storage settings manually' radio button is selected. The 'Subscription' dropdown shows 'Microsoft Azure Internal Consumption (6872d2e0-b57d-41...'. The 'Storage account' dropdown is set to 'sngunsa1'. The 'Storage account key' field contains a masked value. The 'Container' dropdown shows 'cont1'. The 'Path pattern' field is empty. The 'Date format' dropdown is set to 'YYYY/MM/DD'. The 'Time format' dropdown is set to 'HH'. The 'Event serialization format' dropdown is also highlighted with a red box. At the bottom right, a blue 'Save' button is visible.

4. Click **Save**.

Add the Machine Learning function

Earlier you published a Machine Learning model to a web service. In this scenario, when the Stream Analysis job runs, it sends each sample tweet from the input to the web service for sentiment analysis. The Machine Learning web service returns a sentiment (`positive`, `neutral`, or `negative`) and a probability of the tweet being positive.

In this section of the tutorial, you define a function in the Stream Analysis job. The function can be invoked to send a tweet to the web service and get the response back.

1. Make sure you have the web service URL and API key that you downloaded earlier in the Excel workbook.
2. Navigate to your job blade > **Functions** > **+ Add** > **AzureML**
3. Fill out the **Azure Machine Learning function** blade with these values:

FIELD	VALUE
Function alias	Use the name <code>sentiment</code> and select Provide Azure Machine Learning function settings manually which gives you an option to enter the URL and key.
URL	Paste the web service URL.
Key	Paste the API key.

4. Click **Save**.

Create a query to transform the data

Stream Analytics uses a declarative, SQL-based query to examine the input and process it. In this section, you create a query that reads each tweet from input and then invokes the Machine Learning function to perform sentiment analysis. The query then sends the result to the output that you defined (blob storage).

1. Return to the job overview blade.
2. Under **Job Topology**, click the **Query** box.
3. Enter the following query:

```
WITH sentiment AS (
    SELECT text, sentiment1(text) as result
    FROM datainput
)

SELECT text, result.[Score]
INTO datamloutput
FROM sentiment
```

The query invokes the function you created earlier (`sentiment`) in order to perform sentiment analysis on each tweet in the input.

4. Click **Save** to save the query.

Start the Stream Analytics job and check the output

You can now start the Stream Analytics job.

Start the job

1. Return to the job overview blade.
2. Click **Start** at the top of the blade.

3. In the **Start job**, select **Custom**, and then select one day prior to when you uploaded the CSV file to blob storage. When you're done, click **Start**.

Check the output

1. Let the job run for a few minutes until you see activity in the **Monitoring** box.
2. If you have a tool that you normally use to examine the contents of blob storage, use that tool to examine the `azuresamldemoblob` container. Alternatively, do the following steps in the Azure portal:
 - a. In the portal, find the `samldemo` storage account, and within the account, find the `azuresamldemoblob` container. You see two files in the container: the file that contains the sample tweets and a CSV file generated by the Stream Analytics job.
 - b. Right-click the generated file and then select **Download**.

The screenshot shows the Azure Storage Explorer interface. At the top, it displays the container name `azuresamldemoblob`. Below the container name are standard navigation buttons: Upload, Refresh, Delete container, Properties, and Access policy. Under the container name, the text "Location: azuresamldemoblob" is shown. A search bar below it contains the placeholder text "Search blobs by prefix (case-sensitive)". The main area is titled "NAME" and lists two files: "-1415627...108_1.csv" and "SA-ML-demo.csv". The file "-1415627...108_1.csv" has a checkmark next to it. A context menu is open over the "-1415627...108_1.csv" file, listing three options: "Edit metadata", "Download", and "Delete". The "Download" option is highlighted with a red box.

3. Open the generated CSV file. You see something like the following example:

text	scored probabilities
Azure is so cool	0.733864188
Cortana Analytics rocks	0.590195894
I hate going to Gym	0.403451264

View metrics

You also can view Azure Machine Learning function-related metrics. The following function-related metrics are displayed in the **Monitoring** box in the job blade:

- **Function Requests** indicates the number of requests sent to a Machine Learning web service.
- **Function Events** indicates the number of events in the request. By default, each request to a Machine Learning web service contains up to 1,000 events.

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Integrate REST API and Machine Learning](#)
- [Azure Stream Analytics Management REST API Reference](#)

Anomaly detection in Azure Stream Analytics

6/21/2019 • 3 minutes to read • [Edit Online](#)

Available in both the cloud and Azure IoT Edge, Azure Stream Analytics offers built-in machine learning based anomaly detection capabilities that can be used to monitor the two most commonly occurring anomalies: temporary and persistent. With the **AnomalyDetection_SpikeAndDip** and **AnomalyDetection_ChangePoint** functions, you can perform anomaly detection directly in your Stream Analytics job.

The machine learning models assume a uniformly sampled time series. If the time series is not uniform, you may insert an aggregation step with a tumbling window prior to calling anomaly detection.

The machine learning operations do not support seasonality trends or multi-variate correlations at this time.

Model accuracy and performance

Generally, the model's accuracy improves with more data in the sliding window. The data in the specified sliding window is treated as part of its normal range of values for that time frame. The model only considers event history over the sliding window to check if the current event is anomalous. As the sliding window moves, old values are evicted from the model's training.

The functions operate by establishing a certain normal based on what they have seen so far. Outliers are identified by comparing against the established normal, within the confidence level. The window size should be based on the minimum events required to train the model for normal behavior so that when an anomaly occurs, it would be able to recognize it.

The model's response time increases with history size because it needs to compare against a higher number of past events. It is recommended to only include the necessary number of events for better performance.

Gaps in the time series can be a result of the model not receiving events at certain points in time. This situation is handled by Stream Analytics using imputation logic. The history size, as well as a time duration, for the same sliding window is used to calculate the average rate at which events are expected to arrive.

Spike and dip

Temporary anomalies in a time series event stream are known as spikes and dips. Spikes and dips can be monitored using the Machine Learning based operator, **AnomalyDetection_SpikeAndDip**.



Examples of Spike and Dip anomalies

In the same sliding window, if a second spike is smaller than the first one, the computed score for the smaller spike is probably not significant enough compared to the score for the first spike within the confidence level specified.

You can try decreasing the model's confidence level to detect such anomalies. However, if you start to get too many alerts, you can use a higher confidence interval.

The following example query assumes a uniform input rate of one event per second in a 2-minute sliding window with a history of 120 events. The final SELECT statement extracts and outputs the score and anomaly status with a confidence level of 95%.

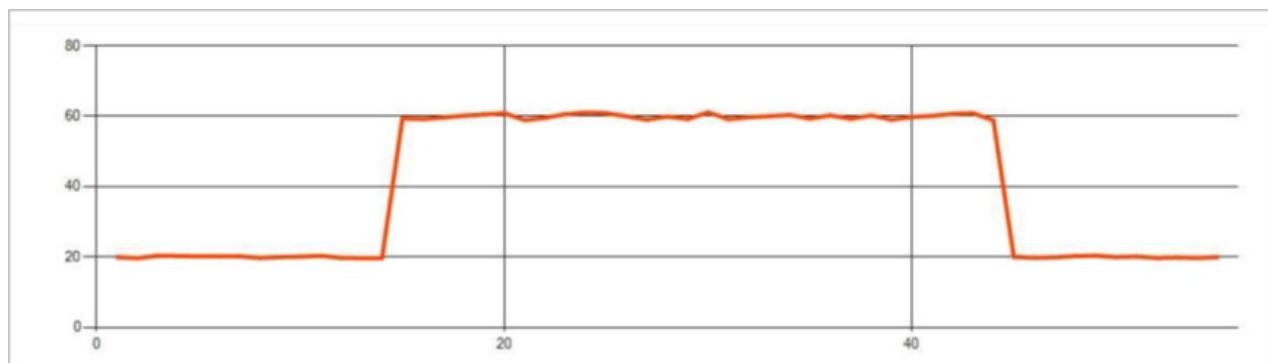
```
WITH AnomalyDetectionStep AS
(
    SELECT
        EVENTENQUEUEDUTCTIME AS time,
        CAST(temperature AS float) AS temp,
        AnomalyDetection_SpikeAndDip(temperature AS float, 95, 120, 'spikesanddips')
            OVER(LIMIT DURATION(second, 120)) AS SpikeAndDipScores
    FROM input
)
SELECT
    time,
    temp,
    CAST(GetRecordPropertyValue(SpikeAndDipScores, 'Score') AS float) AS
    SpikeAndDipScore,
    CAST(GetRecordPropertyValue(SpikeAndDipScores, 'IsAnomaly') AS bigint) AS
    IsSpikeAndDipAnomaly
INTO output
FROM AnomalyDetectionStep
```

Change point

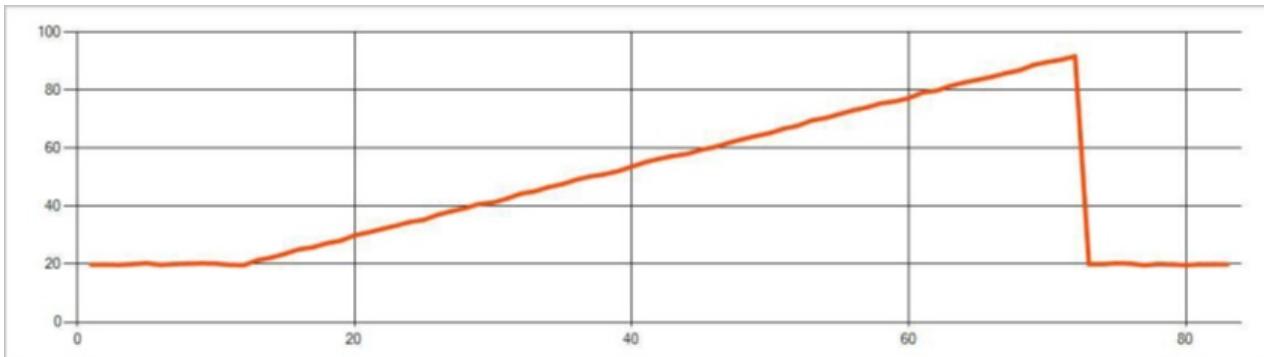
Persistent anomalies in a time series event stream are changes in the distribution of values in the event stream, like level changes and trends. In Stream Analytics, such anomalies are detected using the Machine Learning based [AnomalyDetection_ChangePoint](#) operator.

Persistent changes last much longer than spikes and dips and could indicate catastrophic event(s). Persistent changes are not usually visible to the naked eye, but can be detected with the **AnomalyDetection_ChangePoint** operator.

The following image is an example of a level change:



The following image is an example of a trend change:



The following example query assumes a uniform input rate of one event per second in a 20-minute sliding window with a history size of 1200 events. The final SELECT statement extracts and outputs the score and anomaly status with a confidence level of 80%.

```
WITH AnomalyDetectionStep AS
(
    SELECT
        EVENTENQUEUEDUTCTIME AS time,
        CAST(temperature AS float) AS temp,
        AnomalyDetection_ChangePoint(CAST(temperature AS float), 80, 1200)
        OVER(LIMIT DURATION(minute, 20)) AS ChangePointScores
    FROM input
)
SELECT
    time,
    temp,
    CAST(GetRecordPropertyValue(ChangePointScores, 'Score') AS float) AS
    ChangePointScore,
    CAST(GetRecordPropertyValue(ChangePointScores, 'IsAnomaly') AS bigint) AS
    IsChangePointAnomaly
INTO output
FROM AnomalyDetectionStep
```

Anomaly detection using machine learning in Azure Stream Analytics

The following video demonstrates how to detect an anomaly in real time using machine learning functions in Azure Stream Analytics.

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

Azure Machine Learning Studio integration in Stream Analytics (Preview)

6/17/2019 • 4 minutes to read • [Edit Online](#)

Stream Analytics supports user-defined functions that call out to Azure Machine Learning Studio endpoints. REST API support for this feature is detailed in the [Stream Analytics REST API library](#). This article provides supplemental information needed for successful implementation of this capability in Stream Analytics. A tutorial has also been posted and is available [here](#).

Overview: Azure Machine Learning Studio terminology

Microsoft Azure Machine Learning Studio provides a collaborative, drag-and-drop tool you can use to build, test, and deploy predictive analytics solutions on your data. This tool is called the *Azure Machine Learning Studio*. The studio is used to interact with the Machine Learning resources and easily build, test, and iterate on your design. These resources and their definitions are below.

- **Workspace:** The *workspace* is a container that holds all other Machine Learning resources together in a container for management and control.
- **Experiment:** *Experiments* are created by data scientists to utilize datasets and train a machine learning model.
- **Endpoint:** *Endpoints* are the Azure Machine Learning Studio object used to take features as input, apply a specified machine learning model and return scored output.
- **Scoring Webservice:** A *scoring webservice* is a collection of endpoints as mentioned above.

Each endpoint has apis for batch execution and synchronous execution. Stream Analytics uses synchronous execution. The specific service is named a [Request/Response Service](#) in Azure Machine Learning studio.

Machine Learning resources needed for Stream Analytics jobs

For the purposes of Stream Analytics job processing, a Request/Response endpoint, an [apikey](#), and a swagger definition are all necessary for successful execution. Stream Analytics has an additional endpoint that constructs the url for swagger endpoint, looks up the interface and returns a default UDF definition to the user.

Configure a Stream Analytics and Machine Learning UDF via REST API

By using REST APIs you may configure your job to call Azure Machine Language functions. The steps are as follows:

1. Create a Stream Analytics job
2. Define an input
3. Define an output
4. Create a user-defined function (UDF)
5. Write a Stream Analytics transformation that calls the UDF
6. Start the job

Creating a UDF with basic properties

As an example, the following sample code creates a scalar UDF named *newudf* that binds to an Azure Machine Learning Studio endpoint. Note that the *endpoint* (service URI) can be found on the API help page for the chosen service and the *apiKey* can be found on the Services main page.

```
PUT :  
/subscriptions/<subscriptionId>/resourceGroups/<resourceGroup>/providers/Microsoft.StreamAnalytics/streamingjo  
bs/<streamingjobName>/functions/<udfName>?api-version=<apiVersion>
```

Example request body:

```
{  
    "name": "newudf",  
    "properties": {  
        "type": "Scalar",  
        "properties": {  
            "binding": {  
                "type": "Microsoft.MachineLearning/WebService",  
                "properties": {  
                    "endpoint":  
                    "https://ussouthcentral.services.azureml.net/workspaces/f80d5d7a77fb4b46bf2a30c63c078dca/services/b7be5e40fd19  
4258796fb402c1958eaf/execute ",  
                    "apiKey": "replacekeyhere"  
                }  
            }  
        }  
    }  
}
```

Call RetrieveDefaultDefinition endpoint for default UDF

Once the skeleton UDF is created the complete definition of the UDF is needed. The RetrieveDefaultDefinition endpoint helps you get the default definition for a scalar function that is bound to an Azure Machine Learning Studio endpoint. The payload below requires you to get the default UDF definition for a scalar function that is bound to an Azure Machine Learning endpoint. It doesn't specify the actual endpoint as it has already been provided during PUT request. Stream Analytics calls the endpoint provided in the request if it is provided explicitly. Otherwise it uses the one originally referenced. Here the UDF takes a single string parameter (a sentence) and returns a single output of type string which indicates the "sentiment" label for that sentence.

```
POST :  
/subscriptions/<subscriptionId>/resourceGroups/<resourceGroup>/providers/Microsoft.StreamAnalytics/streamingjo  
bs/<streamingjobName>/functions/<udfName>/RetrieveDefaultDefinition?api-version=<apiVersion>
```

Example request body:

```
{  
    "bindingType": "Microsoft.MachineLearning/WebService",  
    "bindingRetrievalProperties": {  
        "executeEndpoint": null,  
        "udfType": "Scalar"  
    }  
}
```

A sample output of this would look something like below.

```
{
  "name": "newudf",
  "properties": {
    "type": "Scalar",
    "properties": {
      "inputs": [{
        "dataType": "nvarchar(max)",
        "isConfigurationParameter": null
      }],
      "output": {
        "dataType": "nvarchar(max)"
      },
      "binding": {
        "type": "Microsoft.MachineLearning/WebService",
        "properties": {
          "endpoint":
"https://ussouthcentral.services.azureml.net/workspaces/f80d5d7a77ga4a4bbf2a30c63c078dca/services/b7be5e40fd194258896fb602c1858eaf/execute",
          "apiKey": null,
          "inputs": {
            "name": "input1",
            "columnNames": [
              {
                "name": "tweet",
                "dataType": "string",
                "mapTo": 0
              }
            ],
            "outputs": [
              {
                "name": "Sentiment",
                "dataType": "string"
              }
            ],
            "batchSize": 10
          }
        }
      }
    }
  }
}
```

Patch UDF with the response

Now the UDF must be patched with the previous response, as shown below.

```
PATCH :
/subscriptions/<subscriptionId>/resourceGroups/<resourceGroup>/providers/Microsoft.StreamAnalytics/streamingjobs/<streamingjobName>/functions/<udfName>?api-version=<apiVersion>
```

Request Body (Output from RetrieveDefaultDefinition):

```
{
  "name": "newudf",
  "properties": {
    "type": "Scalar",
    "properties": {
      "inputs": [{
        "dataType": "nvarchar(max)",
        "isConfigurationParameter": null
      }],
      "output": {
        "dataType": "nvarchar(max)"
      },
      "binding": {
        "type": "Microsoft.MachineLearning/WebService",
        "properties": {
          "endpoint":
"https://ussouthcentral.services.azureml.net/workspaces/f80d5d7a77ga4a4bbf2a30c63c078dca/services/b7be5e40fd194258896fb602c1858eaf/execute",
          "apiKey": null,
          "inputs": {
            "name": "input1",
            "columnNames": [
              {
                "name": "tweet",
                "dataType": "string",
                "mapTo": 0
              }
            ],
            "outputs": [
              {
                "name": "Sentiment",
                "dataType": "string"
              }
            ],
            "batchSize": 10
          }
        }
      }
    }
  }
}
```

Implement Stream Analytics transformation to call the UDF

Now query the UDF (here named scoreTweet) for every input event and write a response for that event to an output.

```
{
  "name": "transformation",
  "properties": {
    "streamingUnits": null,
    "query": "select *,scoreTweet(Tweet) TweetSentiment into blobOutput from blobInput"
  }
}
```

Get help

For further assistance, try our [Azure Stream Analytics forum](#)

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)

- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)

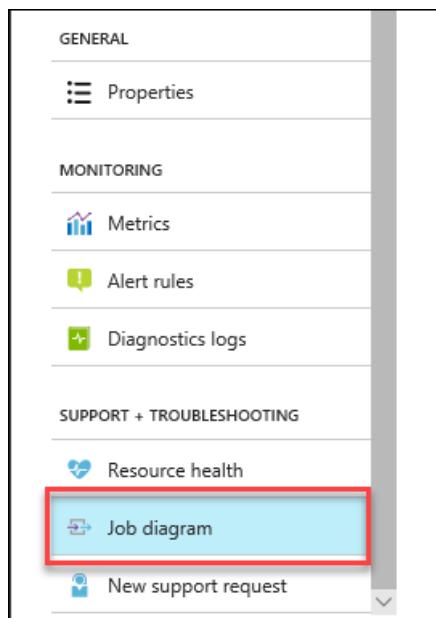
Data-driven debugging by using the job diagram

4/17/2018 • 3 minutes to read • [Edit Online](#)

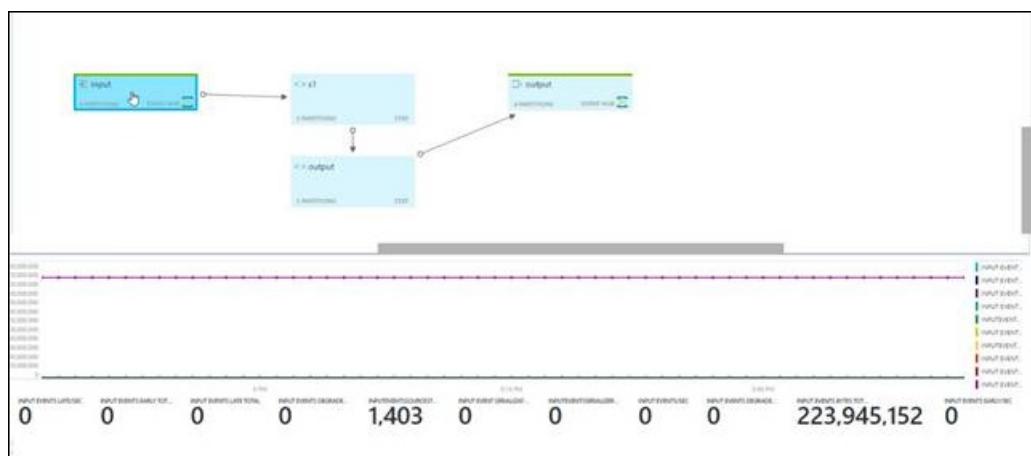
The job diagram on the **Monitoring** blade in the Azure portal can help you visualize your job pipeline. It shows inputs, outputs, and query steps. You can use the job diagram to examine the metrics for each step, to more quickly isolate the source of a problem when you troubleshoot issues.

Using the job diagram

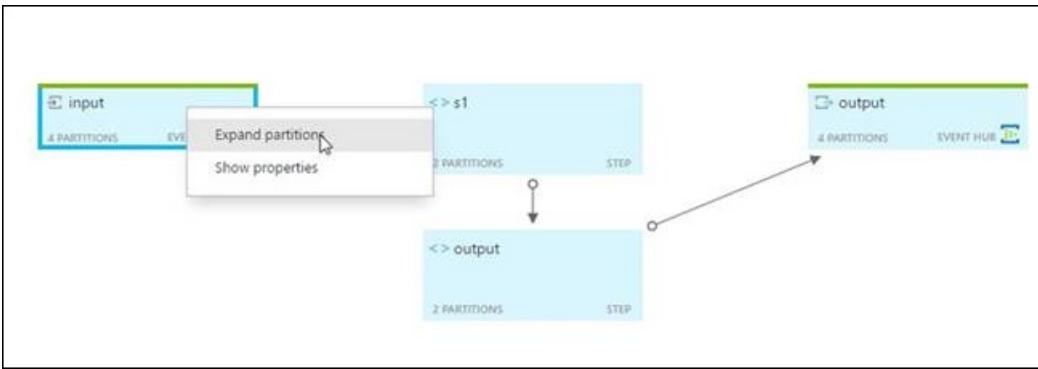
In the Azure portal, while in a Stream Analytics job, under **SUPPORT + TROUBLESHOOTING**, select **Job diagram**:



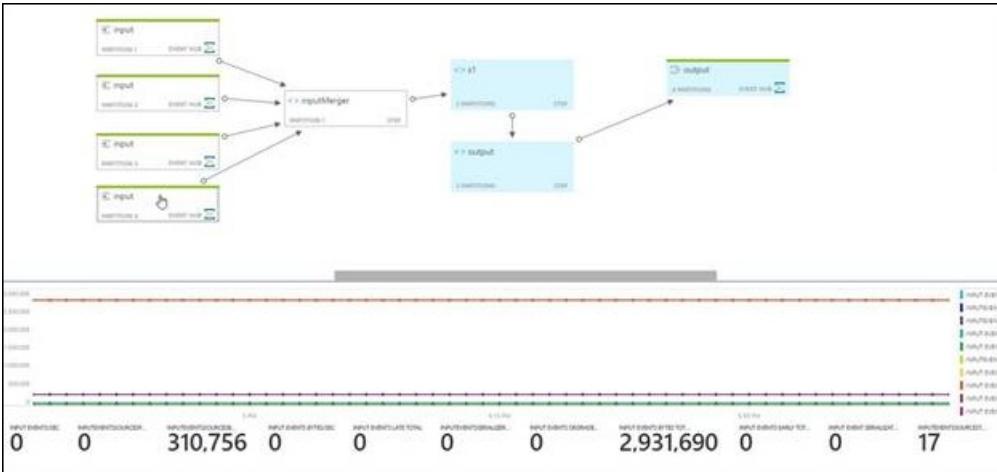
Select each query step to see the corresponding section in a query editing pane. A metric chart for the step is displayed in a lower pane on the page.



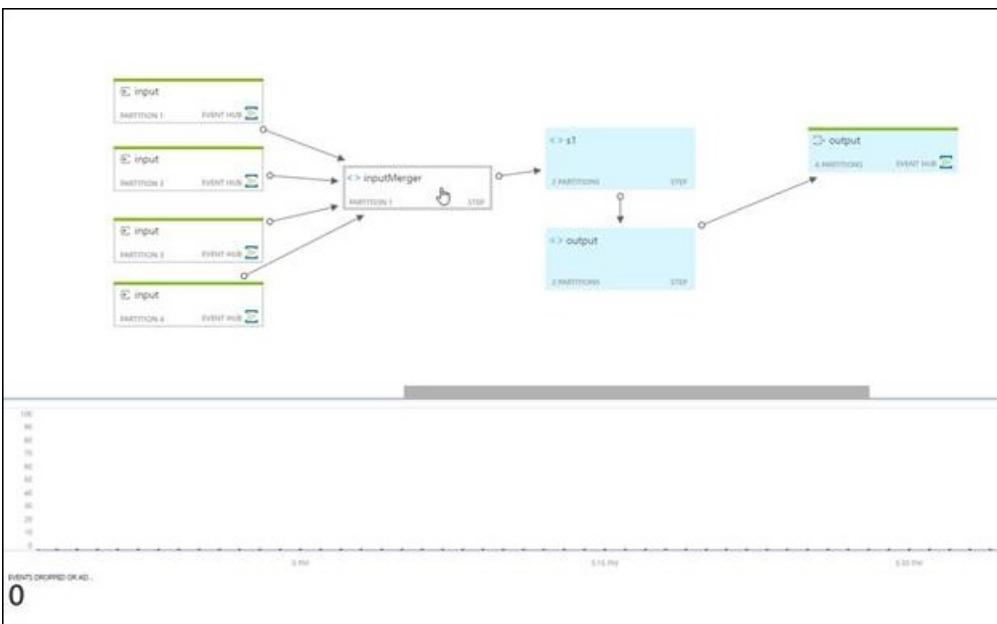
To see the partitions of the Azure Event Hubs input, select ... A context menu appears. You also can see the input merger.



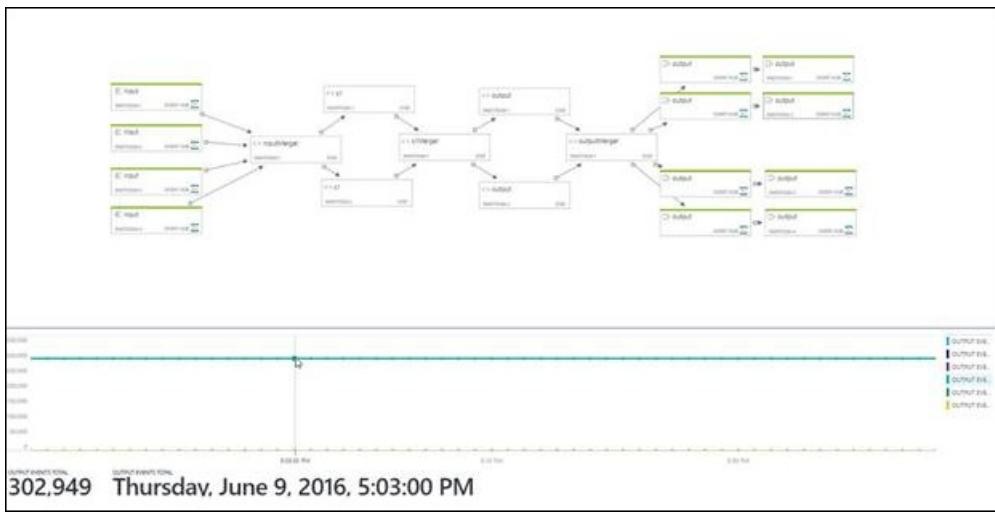
To see the metric chart for only a single partition, select the partition node. The metrics are shown at the bottom of the page.



To see the metrics chart for a merger, select the merger node. The following chart shows that no events were dropped or adjusted.



To see the details of the metric value and time, point to the chart.



Troubleshoot by using metrics

The **QueryLastProcessedTime** metric indicates when a specific step received data. By looking at the topology, you can work backward from the output processor to see which step is not receiving data. If a step is not getting data, go to the query step just before it. Check whether the preceding query step has a time window, and if enough time has passed for it to output data. (Note that time windows are snapped to the hour.)

If the preceding query step is an input processor, use the input metrics to help answer the following targeted questions. They can help you determine whether a job is getting data from its input sources. If the query is partitioned, examine each partition.

How much data is being read?

- **InputEventsSourcesTotal** is the number of data units read. For example, the number of blobs.
- **InputEventsTotal** is the number of events read. This metric is available per partition.
- **InputEventsInBytesTotal** is the number of bytes read.
- **InputEventsLastArrivalTime** is updated with every received event's enqueued time.

Is time moving forward? If actual events are read, punctuation might not be issued.

- **InputEventsLastPunctuationTime** indicates when a punctuation was issued to keep time moving forward. If punctuation is not issued, data flow can get blocked.

Are there any errors in the input?

- **InputEventsEventDataNullTotal** is a count of events that have null data.
- **InputEventsSerializerErrorsTotal** is a count of events that could not be deserialized correctly.
- **InputEventsDegradedTotal** is a count of events that had an issue other than with deserialization.

Are events being dropped or adjusted?

- **InputEventsEarlyTotal** is the number of events that have an application timestamp before the high watermark.
- **InputEventsLateTotal** is the number of events that have an application timestamp after the high watermark.
- **InputEventsDroppedBeforeApplicationStartTimeTotal** is the number events dropped before the job start time.

Are we falling behind in reading data?

- **Input Events Backlogged (Total)** tells you how many more messages need to be read for Event Hubs and Azure IoT Hub inputs. When this number is greater than 0, it means your job can't process the data as fast as it is coming in. In this case you may need to increase the number of Streaming Units and/or make sure your job can be parallelized. You can see more info on this on the [query parallelization page](#).

Get help

For additional assistance, try our [Azure Stream Analytics forum](#).

Next steps

- [Introduction to Stream Analytics](#)
- [Get started with Stream Analytics](#)
- [Scale Stream Analytics jobs](#)
- [Stream Analytics query language reference](#)
- [Stream Analytics management REST API reference](#)

Parse JSON and Avro data in Azure Stream Analytics

6/21/2019 • 2 minutes to read • [Edit Online](#)

Azure Stream Analytics support processing events in CSV, JSON, and Avro data formats. Both JSON and Avro data can be structured and contain some complex types such as nested objects (records) and arrays.

Record data types

Record data types are used to represent JSON and Avro arrays when corresponding formats are used in the input data streams. These examples demonstrate a sample sensor, which is reading input events in JSON format. Here is example of a single event:

```
{  
    "DeviceId" : "12345",  
    "Location" :  
    {  
        "Lat": 47,  
        "Long": 122  
    },  
    "SensorReadings" :  
    {  
        "Temperature" : 80,  
        "Humidity" : 70,  
        "CustomSensor01" : 5,  
        "CustomSensor02" : 99,  
        "SensorMetadata" :  
        {  
            "Manufacturer": "ABC",  
            "Version": "1.2.45"  
        }  
    }  
}
```

Access nested fields in known schema

Use dot notation (.) to easily access nested fields directly from your query. For example, this query selects the Latitude and Longitude coordinates under the Location property in the preceding JSON data. The dot notation can be used to navigate multiple levels as shown below.

```
SELECT  
    DeviceID,  
    Location.Lat,  
    Location.Long,  
    SensorReadings.SensorMetadata.Version  
FROM input
```

Select all properties

You can select all the properties of a nested record using '*' wildcard. Consider the following example:

```
SELECT input.Location.*  
FROM input
```

The result is:

```
{  
    "Lat" : 47,  
    "Long" : 122  
}
```

Access nested fields when property name is a variable

Use the [GetRecordPropertyValue](#) function if the property name is a variable.

For example, imagine a sample data stream needs to be joined with reference data containing thresholds for each device sensor. A snippet of such reference data is shown below.

```
{  
    "DeviceId" : "12345",  
    "SensorName" : "Temperature",  
    "Value" : 75  
}
```

```
SELECT  
    input.DeviceID,  
    thresholds.SensorName  
FROM input      -- stream input  
JOIN thresholds -- reference data input  
ON  
    input.DeviceId = thresholds.DeviceId  
WHERE  
    GetRecordPropertyValue(input.SensorReadings, thresholds.SensorName) > thresholds.Value  
    -- the where statement selects the property value coming from the reference data
```

Convert record fields into separate events

To convert record fields into separate events, use the [APPLY](#) operator together with the [GetRecordProperties](#) function. For example, if the previous example had several records for SensorReading, the following query could be used to extract them into different events:

```
SELECT  
    event.DeviceID,  
    sensorReading.PropertyName,  
    sensorReading.PropertyValue  
FROM input as event  
CROSS APPLY GetRecordProperties(event.SensorReadings) AS sensorReading
```

Array data types

Array data types are an ordered collection of values. Some typical operations on array values are detailed below. These examples assume the input events have a property named "arrayField" that is an array datatype.

These examples use the functions [GetArrayElement](#), [GetArrayElements](#), [GetArrayLength](#), and the [APPLY](#) operator.

Working with a specific array element

Select array element at a specified index (selecting the first array element):

```
SELECT  
    GetArrayElement(arrayField, 0) AS firstElement  
FROM input
```

Select array length

```
SELECT
    GetArrayLength(arrayField) AS arrayLength
FROM input
```

Convert array elements into separate events

Select all array element as individual events. The [APPLY](#) operator together with the [GetArrayElements](#) built-in function extracts all array elements as individual events:

```
SELECT
    arrayElement.ArrayIndex,
    arrayElement.ArrayValue
FROM input as event
CROSS APPLY GetArrayElements(event.arrayField) AS arrayElement
```

See Also

[Data Types in Azure Stream Analytics](#)

Azure Stream Analytics preview features

5/30/2019 • 2 minutes to read • [Edit Online](#)

This article summarizes all the features currently in preview for Azure Stream Analytics. Using preview features in a production environment isn't recommended.

Public previews

The following features are in public preview. You can take advantage of these features today, but don't use them in your production environment.

Visual Studio Code for Azure Stream Analytics (Released May 2019)

Azure Stream Analytics jobs can be authored in Visual Studio Code. See our [VS Code getting started tutorial](#).

Anomaly Detection

Azure Stream Analytics introduces new machine learning models with support for *spike* and *dips* detection in addition to bi-directional, slow positive, and slow negative trends detection. For more information, visit [Anomaly detection in Azure Stream Analytics](#).

Integration with Azure Machine Learning

You can scale Stream Analytics jobs with Machine Learning (ML) functions. To learn more about how you can use ML functions in your Stream Analytics job, visit [Scale your Stream Analytics job with Azure Machine Learning functions](#). Check out a real-world scenario with [Performing sentiment analysis by using Azure Stream Analytics and Azure Machine Learning](#).

JavaScript user-defined aggregate

Azure Stream Analytics supports user-defined aggregates (UDA) written in JavaScript, which enable you to implement complex stateful business logic. Learn how to use UDAs from the [Azure Stream Analytics JavaScript user-defined aggregates](#) documentation.

Live data testing in Visual Studio

Visual Studio tools for Azure Stream Analytics enhance the local testing feature that allows you to test your queries against live event streams from cloud sources such as Event Hub or IoT hub. Learn how to [Test live data locally using Azure Stream Analytics tools for Visual Studio](#).

.NET user-defined functions on IoT Edge

With .NET standard user-defined functions, you can run .NET Standard code as part of your streaming pipeline. You can create simple C# classes or import full project and libraries. Full authoring and debugging experience is supported in Visual Studio. For more information, visit [Develop .NET Standard user-defined functions for Azure Stream Analytics Edge jobs](#).

Other previews

The following features are also available in preview.

C# custom deserializer for Azure Stream Analytics on IoT Edge and Cloud (Announced May 2019)

Developers can implement custom deserializers in C# to deserialize events received by Azure Stream Analytics. Examples of formats that can be serialized include Parquet, Protobuf, XML, or any binary format. Sign up for this preview [here](#).

Parquet Output (Announced May 2019)

Parquet is a columnar format enabling efficient big data processing. By outputting data in Parquet format in a data lake, you can take advantage of Azure Stream Analytics to power large scale streaming ETL and run batch processing, train machine learning algorithms or run interactive queries on your historical data. Sign up for this preview [here](#).

One-click integration with Event Hubs (Announced May 2019)

With this integration, you will now be able to visualize incoming data and start to write a Stream Analytics query with one click from the Event Hub portal. Once your query is ready, you will be able to productize it in few clicks and start to get real-time insights. This will significantly reduce the time and cost to develop real-time analytics solutions. Sign up for this preview [here](#).

Support for Azure Stack (Announced May 2019)

This feature enabled on the Azure IoT Edge runtime, leverages custom Azure Stack features, such as native support for local inputs and outputs running on Azure Stack (for example Event Hubs, IoT Hub, Blob Storage). This new integration enables you to build hybrid architectures that can analyze your data close to where it is generated, lowering latency and maximizing insights. Sign up for this preview [here](#).

Reference architecture: Real-time event processing with Microsoft Azure Stream Analytics

12/17/2018 • 2 minutes to read • [Edit Online](#)

The reference architecture for real-time event processing with Azure Stream Analytics is intended to provide a generic blueprint for deploying a real-time platform as a service (PaaS) stream-processing solution with Microsoft Azure.

Summary

Traditionally, analytics solutions have been based on capabilities such as ETL (extract, transform, load) and data warehousing, where data is stored prior to analysis. Changing requirements, including more rapidly arriving data, are pushing this existing model to the limit. The ability to analyze data within moving streams prior to storage is one solution, and while it is not a new capability, the approach has not been widely adopted across all industry verticals.

Microsoft Azure provides an extensive catalog of analytics technologies that are capable of supporting an array of different solution scenarios and requirements. Selecting which Azure services to deploy for an end-to-end solution can be a challenge given the breadth of offerings. This paper is designed to describe the capabilities and interoperation of the various Azure services that support an event-streaming solution. It also explains some of the scenarios in which customers can benefit from this type of approach.

Contents

- Executive Summary
- Introduction to Real-Time Analytics
- Value Proposition of Real-Time Data in Azure
- Common Scenarios for Real-Time Analytics
- Architecture and Components
 - Data Sources
 - Data-Integration Layer
 - Real-time Analytics Layer
 - Data Storage Layer
 - Presentation / Consumption Layer
- Conclusion

Author: Charles Feddersen, Solution Architect, Data Insights Center of Excellence, Microsoft Corporation

Published: January 2015

Revision: 1.0

Download: [Real-Time Event Processing with Microsoft Azure Stream Analytics](#)

Get help

For further assistance, try the [Azure Stream Analytics forum](#)

Next steps

- [Introduction to Azure Stream Analytics](#)
- [Get started using Azure Stream Analytics](#)
- [Scale Azure Stream Analytics jobs](#)
- [Azure Stream Analytics Query Language Reference](#)
- [Azure Stream Analytics Management REST API Reference](#)