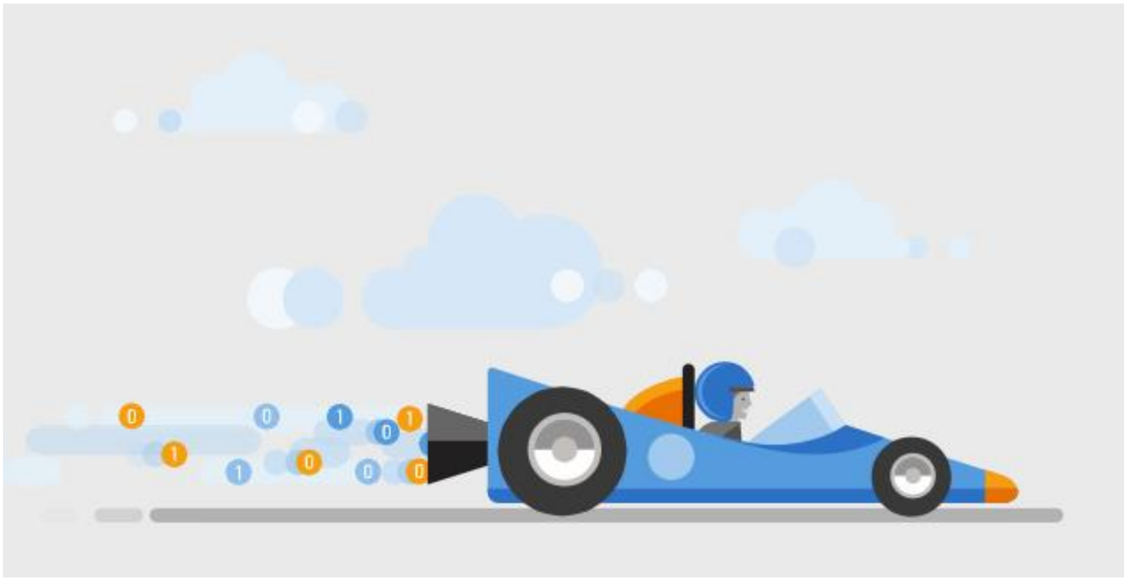
Introducing Data Accelerator

An easy to configure streaming pipeline for Apache Spark



Noteworthy Capabilities

- Frictionless configuration
- Code free rules and alerting
- Fast development feedback loop
- Establish a proof of concept in an Afternoon

Primary Use

ETL and Alerting for streaming pipelines on Apache Spark

Deployment Options

- Docker Image for Linux, Mac, and Windows
- ARM Template for Azure
- GitHub build and deploy

Where to get started

- Github page (Linked)
- Landing page (Docs, Linked)
- Stack Exchange (Linked)

Summary

Data Accelerator is an easy way to set up and run a streaming pipeline on Apache Spark. Within the Developer Tools product group at Microsoft we have used an instance of Data Accelerator to process events at a Microsoft scale since the fall of 2017. Data Accelerator isn’t just a pipe between an EventHub and a database, however, Data Accelerator allows us to reshape incoming events while continuing to stream, then route different parts of the same event into different data stores, and provide health monitoring and alerting over the status of the whole pipeline. And we’ve built a friction free configuration and code-free rules authoring experience that translates your work down into Spark-SQL and then into Scala, so you don't have to. Because we're the Developer Tools product group at Microsoft, the editing experience is pulled directly from the Visual Studio Code, complete with IntelliSense suggestions when editing our Spark-SQL syntax. On those days where you want to get into the nitty gritty refining rules and alerts, we've built a fast debug feedback loop to validate your queries and rules in real-time rather than the more typical "7 minutes and 20 clicks" loop of sending your query to a cluster in the cloud and waiting to discover you've put a semicolon in the wrong place. We've found Data Accelerator to make it easy to onboard new data sources, develop new rules and alerts, and quickly work through proof of concepts deployments, all while smoothly handling Microsoft scale Challenges.

Because of all that, we think some of you will find Data Accelerator interesting and we hope some find it useful enough to work with and even contribute back to the project. We can't wait to see what comes next.

The Problem we set out to solve:

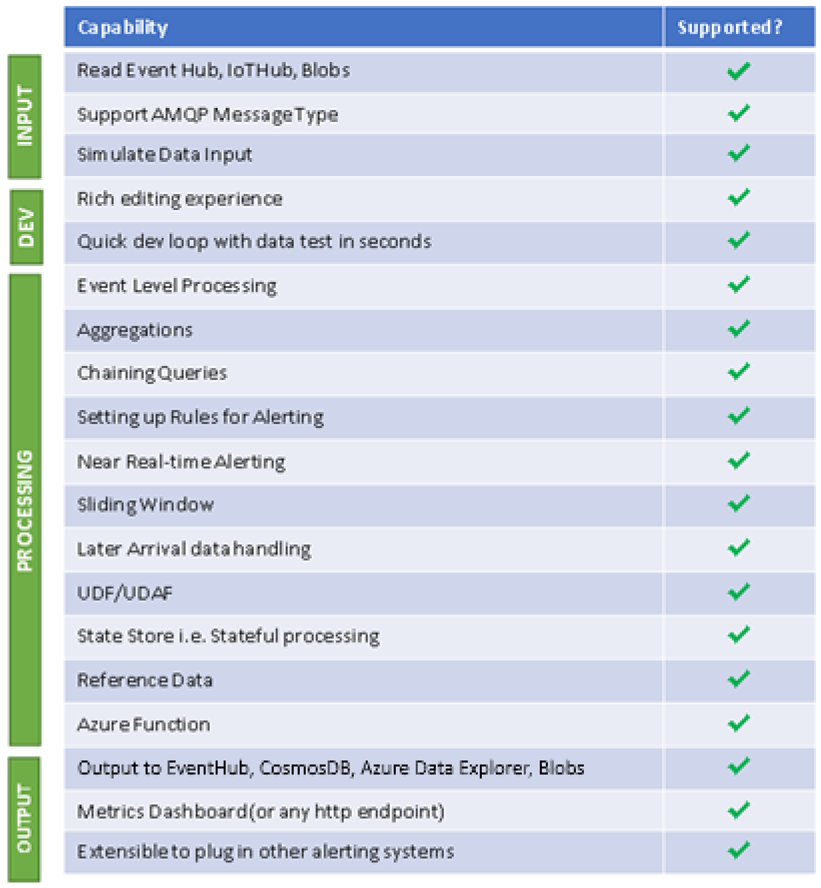
We built Data Accelerator to address a specific problem:

> How do you deal with a heterogeneous event stream from a variety of different input sources in a way that promotes quick discovery of data insights.

When to use Data Accelerator:

- You are looking to set up a streaming data pipeline on Spark and want to see the end to end running before the afternoon is done.
- You are in a heavy development / exploration / prototype cycle and need the fast develop-debug experience.
- You are looking to get into Spark streaming but don't want to configure and glue together the interop to support all the individual components or pick up another language before committing.

Data Accelerator’s Capabilities:



	Capability	Supported?
INPUT	Read Event Hub, IoTHub, Blobs	✓
	Support AMQP MessageType	✓
	Simulate Data Input	✓
DEV	Rich editing experience	✓
	Quick dev loop with data test in seconds	✓
PROCESSING	Event Level Processing	✓
	Aggregations	✓
	Chaining Queries	✓
	Setting up Rules for Alerting	✓
	Near Real-time Alerting	✓
	Sliding Window	✓
	Later Arrival data handling	✓
	UDF/UDAF	✓
	State Store i.e. Stateful processing	✓
	Reference Data	✓
	Azure Function	✓
OUTPUT	Output to EventHub, CosmosDB, Azure Data Explorer, Blobs	✓
	Metrics Dashboard (or any http endpoint)	✓
	Extensible to plug in other alerting systems	✓

Reshaping incoming events: Data Accelerator can support many different data sources, which leads to dealing with many different types of events. The flexibility of the data input within Data Accelerator supports processing across event variations between sources as well as within each source. It is often preferable to normalize incoming data prior to subsequent storage and analysis. Through combination of event and schema, Data Accelerator can recognize and modify events or event parts as they continue streaming through the pipeline. Reshaped events can be split, merged, with values modified or dropped.

Implicit schema detection: Hand editing JSON transformation definitions for various events can be tedious and error prone, certainly in a system with potentially thousands of minor variations in incoming data shape. There is a “Get Schema” feature within Data Accelerator that will sample the configured input data and generate candidate schema for the different detected events. This feature gets things running quickly and is a big time-saver when incorporating new data sources into an existing pipeline.

Rules engine: The telemetry pipeline Data Accelerator fits into connects many inputs, different Azure Event Hubs, Blobs, and IoT Hubs, to many outputs such as: hot, warm, and cold storage, email, Blobs, other event hubs and finally to an in-deployment web-based status dashboard. Because it operates over a heterogeneous and many input to many output data stream there was a need for a robust rules engine to ensure each event received the appropriate treatment while transiting the pipeline. The rules engine in Data Accelerator also supports Azure Functions and User Defined Functions (UDFs) to flavor or modify events as they are ingested. For instance, you could reference a custom image recognition library to convert events from a camera into an on/off status change event, or accumulator, or leverage a time window to only indicate a status change event if some pre-condition is already met. These modifications are managed within the pipeline rather than waiting on some round trip processing, potentially providing greater responsiveness and/or reduced storage requirements as portions of the event stream may be identified as irrelevant and dropped before they are written to more persistent storage outputs. Data Accelerator includes the capability to operate within an event; shaping different parts of a single event in different ways before sending the resulting forms to different output destinations. For others working with Streaming on Apache Spark with data sources from different release eras and/or brownfield development this capability should be welcome.

SQL-Spark syntax with Scala code-generation: A word on programming with Data Accelerator. The conventional wisdom is that working with Scala is hard. Even if Scala were easy, however, other languages and syntax are far more popular, meaning you are more likely to find developers fluent in SQL than Scala today. The base syntax of rules or queries within Data Accelerator is a lightly extended SQL-Spark syntax. A half-dozen keywords have been added, essentially as macros, for otherwise complex operations that are frequently performed, supporting operations such as:

• Time Windows	– manage aggregators and look-forward/back windows
• Create and Upsert Table	– manage a temporary table
• Output	– direct query output to various pre-configured storage options

Data Accelerator also supports hooks for incorporating arbitrary reference data, UDFs, and Azure Functions. This permits refining SQL-Spark queries with existing assets and code capabilities you may already have developed in your organization.

Fast debug loop: Data Accelerator was started by the Developer Tools Division at Microsoft and as developers the typical debug loop of modify some code, submit/deploy to a cloud-based execution environment, and wait on an outcome really drove us nuts. Data Accelerator can operate with a much faster feedback loop: Live Query samples the data, keeps a sample, then plays the query over that sample in the spark kernel returning a result in seconds. This develop-debug loop is essentially real-time and a huge productivity boost. For cases where live data is not available or reasonable, Data Accelerator works with an event simulator to provide an experience that is good enough to improve the viability of candidate code changes when you do submit them to your production environment.

Code-free configuration: While configuration files and the command line maybe the preferred configuration and deployment operations once a system is in production, Data Accelerator provides a code-free designer experience for setup and configuration which we've found useful for prototyping new pipelines. While this won't be for everyone, it certainly provides something more to hold onto for those who are not accustomed to configuring and running headless data pipelines. Data Accelerator's capability to deploy at immense scale encouraged development of this experience, as simple commands can have dramatic consequences when streaming the volumes of data the pipeline can move.

Health monitoring and alerting: Data Accelerator deploys with a status dashboard to help visualize the data flow through the system – it is great for making sure seeing that that system has been configured properly and data are flowing. The monitoring and alerting system can send its own events and make API calls to other endpoints and services, based on the operation of the rules engine. The possibilities here range from the silly to sublime, with a wide variety of inputs to operate over and outputs to interact with we can be limited by our imaginations.

Data Accelerator Implementation

Data Accelerator is driving by configuration files written in JSON. One set of configuration defines the parameters needed to on-board a data flow into Data Accelerator including connection strings to the input data store(s), input schema, processing rules, and connection strings to the output data store(s). Another set of configuration files captures any common processing rules that need to be applied to all data flows within a deployment. The Data Accelerator portal provides a designer interface for generating and maintaining these JSON artifacts.

The configuration files are stored in blob storage, with any secrets stored in an Azure Key Vault, and the configuration service manages the deployment of these files. Alternatively, a deployment script from your CICD system can manage the configuration and job execution.

Data Accelerator has four main components:

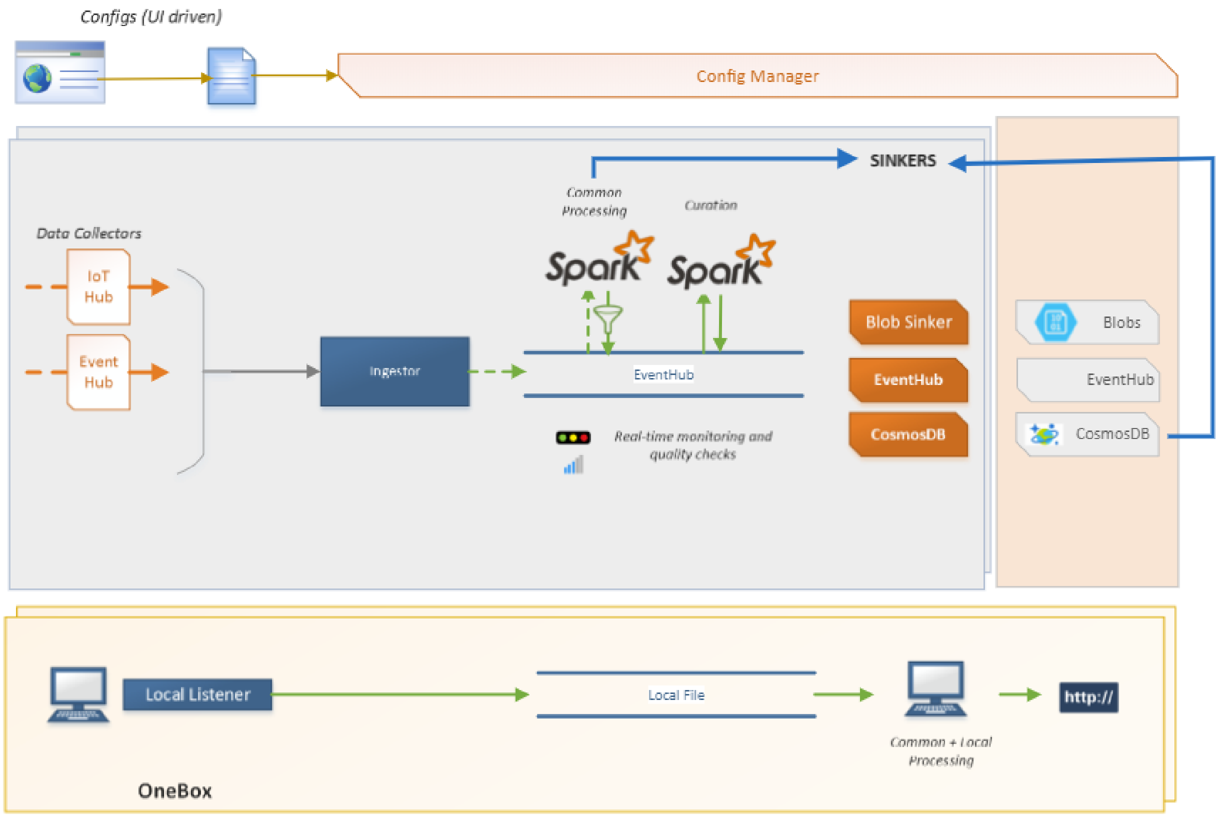
1. **Ingestor**: collects data from a channel like an EventHub and enables processing of data
2. **Central Processor**: runs the queries against the data
3. **Router**: slices and routes the data into different sinkers as well as routing data for further processing. Custom processors can be plugged in to listen and process these events.
4. **Sinker**: services sinking the data to different stores.

The Ingestor connects the data sources to the central processor. These are lightweight and extensible connections to facilitate adding new data stores.

The Central Processing engine reads the configuration files from blob storage and the queries contained are processed into SQL-like language and passed to Spark for execution. This includes any SQL code and the user-defined functions as well. The implementation of the central processing engine is written in Java and Scala, built on Spark version 2.3 and the Azure HDInsight stack.

The Router dispatches work, sending data to the appropriate sinker or custom processors.

The sinker enables storing the resulting processed data into a given data store. This is extensible like the Ingestor to facilitate adding new output data stores.



Metrics data are sent to the Azure Redis cache to enable fast reading from the web portal metrics dashboard. This is done via the metrics service.

Telemetry for the system is sent via Application Insights. The data is viewable on the Azure Portal by going to the Data Accelerator resource group. These data will include exceptions and overall status telemetry for the system allowing the operator to monitor and diagnose any issues that occur.

The system was built using Maven into packages.

Web Portal

The Data Accelerator portal was written in Node with React components and enables managing all the configurations, jobs, and query editing.

Services

Data Accelerator relies on several services to configure the system as well as move data to the Redis cache. The configuration service and the metric service are written in C# using .Net core.