



School of Information Technology and Communications
Department of Digital Systems

Master of Science
Information Systems & Services
Big Data and Analytics

Distributed stream and event processing pipeline in serverless architecture

Author: Orestis Fotiadis

Registration number: ME1825

Supervisor: Assistant Professor Dimosthenis Kyriazis

Piraeus, June 2021

Abstract

The increasing interconnection of our world over the last decade has led to an exponential growth of data that are emitted from personal devices, IoT sensors and other activities of our society. As a result, a very large amount of this data is produced in the form of continuous streams. Data stream processing in real-time has become a crucial operation for several business domains, however processing a large amount of data often from different sources still represents a challenge both technologically and operationally. These needs have led to the emergence of open source and commercial systems that aim to manage and analyze data streams. These systems are of continuing importance as the constant multiplication of data stream sources increases.

This dissertation presents the theoretical background on data processing and the two prevalent architectures for stream processing: The Lambda and Kappa architecture. We also present popular open-source and commercial products on the wider data streaming domain. In the second part of the dissertation, we present the design and implementation of a stream analytics pipeline built on top of Microsoft's Azure cloud platform. We use the Twitter API to stream and analyze data as well as to execute benchmarks to examine the solution's performance. Finally, we present the solution's artifacts as deployable templates.

Keywords: Big data; Cloud computing; Stream analytics; Real-time Processing.

Acknowledgements

My eternal gratitude goes to my wife, Koralia and our dog, Frodo. It would not have been possible for me to wade through this task without their patience and support. To my family, Triantafyllos, Kanella and Eleni, I owe everything. Finally, my sincerest thanks to my supervisor, professor Dimosthenis Kyriazis for his assistance.

Table of contents

Abstract	2
Acknowledgements	3
Table of contents	4
List of figures	8
List of tables	11
Listings	13
List of abbreviations	14
Chapter 1. Introduction	1
1.1 Methodology – approach.....	3
1.2 Dissertation outline.....	3
Chapter 2. Theoretical background	5
2.1 Data processing	5
2.1.1 Stream processing.....	12
2.1.2 Batch processing.....	14

2.1.3	A comparison between stream and batch processing	16
2.2	Stream processing challenges	17
2.3	Indicative business applications involving stream processing systems.....	18
2.4	Stream processing architecture patterns	19
2.4.1	Lambda architecture	20
2.4.2	Kappa architecture.....	24
2.4.3	A comparison between the Lambda and Kappa architectures.....	26
2.5	Architecture principles for a stream analytics solution	28
2.6	Open-source and commercial streaming solutions	31
2.6.1	Open – source streaming solutions.....	31
2.6.1.1	Apache Storm	31
2.6.1.2	Apache Flink	32
2.6.1.3	Apache Samza	33
2.6.2	Commercial streaming solutions	34
2.6.2.1	Amazon Kinesis.....	34
2.6.2.2	IBM InfoSphere Streams	35
2.6.2.3	Azure Stream Analytics.....	35
Chapter 3.	Solution architecture and design.....	37
3.1	Resource groups breakdown.....	41
3.2	Producer tier	43
3.2.1	Twitter streaming API.....	45

3.2.2	Twitter GET Trends/place API.....	46
3.2.3	Botometer API.....	46
3.2.4	NYC Taxi & Limousine Commission - green taxi trip records dataset.....	47
3.3	Ingestion tier.....	47
3.3.1	Twitter Stream Listener.....	50
3.3.1.1	Tweet object properties	54
3.3.1.2	Configurable properties	62
3.3.2	Twitter Trends Monitor	64
3.3.2.1	Configurable properties	68
3.3.3	NYC Green Taxi dataset publisher.....	69
3.3.3.1	Configurable properties	70
3.3.4	Botometer Checker.....	71
3.3.4.1	Configurable properties	73
3.3.5	Event Hub.....	76
3.4	Processing layer.....	81
3.4.1	Stream Analytics Job.....	81
3.5	Persistent storage tier.....	89
3.5.1	Blob storage.....	89
3.5.2	Cosmos DB database	90
3.6	Presentation tier	94
Chapter 4.	Performance evaluation	96

4.1	Dataset and producer application configuration.....	97
4.2	Event hub and Stream Analytics job configuration.....	99
4.3	Results	103
Chapter 5.	Conclusions and future work.....	108
	Availability of data and materials.....	111
	References	112
Annex I.	Supplementary materials	119
Annex II.	Development environment	120
Annex III.	Solution deployment guide.....	122
	Prerequisites	122
	Deploying the components.....	123
	Cleaning up the environment.....	127
Annex IV.	Sample ingestion data.....	128
	Twitter Streaming API, sample response for hashtag #tesla	128
	Twitter GET trends/place API endpoint sample response for location “New York”	134
	Botometer API sample response for Twitter account.....	149
	NYC Taxi & Limousine Commission - green taxi trip records dataset, sample data	152

List of figures

Figure 2-1: A comparison between the multi-store memory model and a typical stream analytics architecture.	7
Figure 2-2: A generic real-time system with data producers, a real-time processing layer and data consumers	9
Figure 2-3: An abstract reference architecture for a data stream management system [17].	11
Figure 2-4: Processing time lag and event time skew in a stream processing system.....	13
Figure 2-5: A high-level stream processing data flow.	14
Figure 2-6: High-level overview of a batch processing system [20].....	15
Figure 2-7: A high-level batch processing data flow.	16
Figure 2-8: A high-level data processing architecture. More details on the batch / real-time / stream processing architectures have been presented in section 2.1.....	20
Figure 2-9: Overview of the Lambda architecture [29].....	21
Figure 2-10: Overview of the Kappa architecture [31].	25
Figure 2-11: A five-layer architecture pattern for a stream analytics application [34].	30
Figure 2-12: Apache storm topology overview [5]	32
Figure 2-13: High-level overview of the Apache Flink stream processing solution [7].	33

Figure 2-14: High-level overview of the Samza stream processing framework [8].....	34
Figure 2-15: High-level overview of the Amazon Kinesis Stream product [38].....	35
Figure 2-16: High-level overview of the Azure Stream Analytics product.....	36
Figure 3-1: Architectural blueprint for a generic streaming data solution [16].....	39
Figure 3-2: Conceptual architecture diagram of the architecture for the solution presented in this thesis.....	40
Figure 3-3: Example of role-based access control for Azure resource groups.....	41
Figure 3-4: Overview of the solution's resource groups.....	42
Figure 3-5: The resource groups for the solution in the Azure UI. Note that the resource group “rg-vms-alpha” visible in the screenshot contains a Virtual Machine created for the solution development and is not to be considered part of the solution itself.....	43
Figure 3-6: Overview of the architecture's producer layer and its interaction with the ingestion layer.	45
Figure 3-7: Overview of the architecture's ingestion layer	48
Figure 3-8: Overview of the Azure Container Instance deployment process.....	49
Figure 3-9: Overview of a container instance's status in the Azure UI.	50
Figure 3-10: View of the tweets Cosmos DB container in the Azure UI. A single tweet is displayed as JSON document.	62
Figure 3-11: View of the application deployed in Azure as a container. The environmental properties set for the particular instance are displayed.....	64
Figure 3-12: The Twitter Trends Monitor application executing as an Azure Container Instance	67
Figure 3-13: Captured twitter trend events as they appear in Azure's stream analytics job.	67
Figure 3-14: A sample of NYC Green Taxi records sent as events to the Event Hub.	70
Figure 3-15: Architecture of an Azure Event Hub	78
Figure 3-16: Architecture overview of the Event Hub “evh-namespace-alpha” implemented for the solution architecture.	79

Figure 3-17: Overview in the Azure UI of an event hub deployed for the solution.....	80
Figure 3-18: Reference high-level overview of an Azure Stream Analytics process.....	83
Figure 3-19: High level overview of the Blob storage account implemented for the solution.....	90
Figure 3-20: The Power BI dashboard used as the presentation layer of the pipeline.	95
Figure 4-1: Data flow overview of the benchmark.....	97
Figure 4-2: Stream analytics job visualization used for the benchmark scenarios.....	100
Figure 4-3: Azure dashboard we used to monitor the Event Hub and Stream Analytics job performance during the scenario execution.....	103
Figure 4-4: Visualizations of throughput, requests, SU utilization and backlogged event metrics for both scenarios.....	105
Figure 4-5: Sample JSON of the stream analytics job outputs.....	106
Figure 4-6: Stream Analytics job performance for scenario 1	106
Figure 4-7: Stream Analytics job performance for scenario 2	106

List of tables

Table 1: Classification of real-time systems.....	8
Table 2: A high-level comparison between batch and real-time data processing	10
Table 3: A comparison between batch and stream data processing paradigms.....	17
Table 4: High-level comparison of the main characteristics of Lambda and Kappa architectures for stream processing applications.....	20
Table 5: A comparison of the primary attributes of Lambda and Kappa architectures.....	27
Table 6: A collection of pros and cons for the Lambda and Kappa architectures.....	27
Table 7: High-level view of the solution's architecture layers.	40
Table 10: Sample tweet processed and sent as event by the Twitter Stream Listener application.	53
Table 11: Sample tweet properties captured by the Twitter Stream Listener application.....	61
Table 12: Additional properties to the tweet JSON document appended by the Azure Stream Analytics Service.....	61
Table 13: System properties appended to the tweet JSON documents in the Azure Cosmos DB.	62
Table 14: List of environmental properties for the Twitter Stream Listener application.....	63
Table 15: Sample response JSON of the GET trends/place API endpoint for the location "Athens".	66

Table 16: List of environmental properties for the Twitter Trends Monitor application.	69
Table 17: List of environmental properties for the NYC Green Taxi dataset publisher application.	71
Table 18: Attributes returned by the Botometer API	73
Table 19: List of configurable environmental properties for the Botometer Checker Python application.	75
Table 8: Corresponding terms between Apache Kafka and Azure Event Hub concepts	76
Table 9: Figure 3 14: Comparison between Stream Analytics Job and Apache Spark Streaming	82
Table 20: Event Hub and Stream Analytics job configuration for benchmark scenarios.....	99
Table 21: Summary of the metrics observed for the evaluation scenarios.	104
Table 22: Visual Studio Code extensions used in the solution development.	121
Table 23: NYC Taxi & Limousine Commission sample dataset	152

Listings

Listing 1: The Stream Analytics Query used to process data captured in the Event Hub.	89
Listing 2: Cosmos DB SQL query example - querying a subset of user properties that have tweeted above a certain volume.	92
Listing 3: Cosmos DB SQL query example - querying a subset of user properties that have tweeted above a certain volume.	92
Listing 4: Cosmos DB SQL query example - counting tweets that contain a hashtag with a specified substring.	93
Listing 5: Cosmos DB SQL query example - querying usernames, hashtags and statuses count for tweets that contain a hashtag with a specified substring.	94
Listing 7: The stream analytics job query that is used for the benchmark scenarios.	103

List of abbreviations

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
APM	Application Performance Management (used in the context of Azure Application Insights service)
ARM	Azure Resource Manager (in the context of Azure resource templates)
ASA	Azure Stream Analytics
AZ	Azure (referring to Azure CLI)
CLI	Command Line Interface
COTS	Commercial off-the-shelf
DAG	Directed Acyclic Graph
DB	Database
GDPR	General Data Protection Regulation
JSON	JavaScript Object Notation
KCL	Kinesis Client Library
MVP	Minimum Viable Product
RBAC	Role-Based Access Control (in the context of Azure Access Management)

RU	Resource Unit (in the context of Azure Cosmos DB)
SKU	Stock Keeping Unit (referring to Azure virtual machine images)
SPL	Streams Programming Language
SU	Streaming Unit (referring to Azure Stream Analytics Job streaming unit configuration)
SU	Streaming Unit (in the context of Azure Stream Analytics)
TLC	Taxi and Limousine commission (exists in the dataset name used to run performance evaluation)
T-SQL	Transactional-SQL (referring to Microsoft's and Sybase's proprietary extension to the SQL used to interact with relational databases.
TU	Throughput Unit (in the context of Azure Event Hub)
UI	User Interface
VM	Virtual Machine
WOEID	Where On Earth IDentifier

Chapter 1. Introduction

Over the last decade the world has seen an unprecedented amount of data being generated in various fields, fueled initially by the rise of Web 2.0 and growing at ever-increasing rates with the adoption of Internet of Things (IoT) applications. This proliferation in data generation has been dubbed with the term "Big data" which refers to this increase in the volume of data that is difficult to store, process and analyze with traditional methodologies, technologies and software architectures. The proper collection, management and insights discovery from these data is being recognized as a significant potential competitive advantage for businesses, due to the breadth of their applications across the entire value chain [1].

The complexity, diversity and massive scale of large datasets require new methodologies and frameworks to be properly handled and has thus led to the need of having the ability to deploy and manage adequate workflows which integrate, manage and analyze data from widely distributed sources in a reasonable time frame. This need has also prompted the development of platforms that can adequately handle big data with consideration to performance, scalability and fault tolerance requirements.

The problem of how to efficiently handle data streams is not new: The high-level requirements that a real-time stream processing application should fulfill have been laid out as early as 2005 [ref].

The projects "Aurora" [3] and "Borealis" [4] ran by Brandeis University, Brown University, and MIT are early attempts at creating a stream processing framework.

Presently there is a large number of open-source stream processing solutions and frameworks such as Apache Foundation's Storm [5], Spark Streaming [6], Flink [7] and Samza [8] with each framework, tackling different aspects of big data streaming. On top of these frameworks, major cloud and technology vendors have developed commercial offerings (Amazon Kinesis [9], Azure Stream Analytics [10], Google Cloud Dataflow [11], IBM Infosphere [12]) targeted towards enterprise customers. Such solutions are typically offered with the SaaS model, taking away the complexity of maintaining a full infrastructure while offering increased efficiency. The challenges posed by emerging industry needs reinforce the feedback loop towards research and development efforts for improved frameworks and platforms.

The proliferation of stream processing frameworks combined with their increasing importance in business operations has prompted the proposal for the establishment of a standardized framework for managing streams, with the name "Flow": In [13] Urquhart proposes that flow is conceptualized as "networked software integration that is event-driven, loosely coupled, and highly adaptable and extensible. It is principally defined by standard interfaces and protocols that enable integration with a minimum of conflict and toil".

In this dissertation, we present a prototype solution with configurable components for a stream analytics pipeline which adheres to established architecture principles for applications in the stream analytics domain. The main features is that it has a "decoupled" ingestion component and that the stream processing component can work simultaneously with heterogenous incoming data and output the desired results. All components are deployed / maintained as-code so the entire solution can be very quickly be upgraded and extended to serve use case scenarios.

Thus, this dissertation has the following primary aims:

- To present an overview of the primary architecture models used in stream processing workflows and applications.
- To design and implement a fully functional and configurable stream processing solution built on a COTS cloud computing environment (Microsoft Azure) and execute real-world scenarios on it.

1.1 Methodology – approach

The methodology used in this dissertation is primarily based on empirical research work in utilizing Microsoft's Azure cloud platform offering to build an application in the space of stream analytics. We attempt to apply best practices in the solution design and architecture by performing a literature review of the relevant domain and by presenting an overview of the prevailing stream analytics architectures. In section Chapter 3, we present the architecture and a reference implementation of an application that ingests streaming data from several independent sources and pushes them as events to a message queuing subsystem. A stream analytics component which functions as a consuming subsystem reads the events from the message queue, performs a series of computations depending on the type of event being processed and publishes the results in a dashboard.

The solution that we implement uses a modular design, to facilitate scenarios where individual, loosely coupled components must be updated or to allow expansion by adding additional components not currently being covered in the solution scope. Finally, all the solution components and Azure platform deployment configurations that we have developed for this dissertation have been made publicly available in GitHub repositories using the MIT license.

1.2 Dissertation outline

The rest of this dissertation is organized as follows: Section Chapter 2 presents background information on cloud computing, data processing and stream analytics architectures and discusses the prevalent architecture patterns of stream processing solutions. Section Chapter 3 presents the

solution architecture that has been implemented for this thesis and discusses design and configuration choices made towards its implementation. Section 0 presents findings and benchmark results on the solution's performance under different use case scenarios. Section 0 summarizes the dissertation, presents its conclusions, and proposes further work for expanding the implemented solution. Finally, the Annexes provide hyperlinks to code repositories and other artifacts developed for this dissertation as well as a short deployment guide for the solution, aimed at the technically oriented reader.

Chapter 2. Theoretical background

In section Chapter 1 we introduced the general context of this dissertation, outlined an overview of the streaming data analytics solution under examination and presented its primary objectives. This section presents an overview on the theoretical background of streaming data and the primary architectural patterns for software applications that ingest, process, analyze and store them. We also provide an overview of the most frequent system-level challenges that data streaming solutions must respond to as well as common real-world business applications involving streaming data.

2.1 Data processing

Data processing can be defined as a series of actions or steps performed on data to verify, organize, transform, integrate, and extract data in an appropriate output form for subsequent use¹. In this context, data processing can be considered a specific case of the information processing domain, which is concerned with gathering, manipulating, storing, retrieving, and classifying recorded

¹ Note that the term of data processing in the context of the General Data Protection Regulation (GDPR) in Article 4(2) [16] is defined as "any operation or set of operations which is performed on personal data or on sets of personal data, whether or not by automated means, such as collection, recording, organisation, structuring, storage, adaptation or alteration, retrieval, consultation, use, disclosure by transmission, dissemination or otherwise making available, alignment or combination, restriction, erasure or destruction". In this context, data streaming solutions fall under the regulation if they are processing personal data as is usually the case, for example when processing application usage logs, transactions and other similar data types.

information. A popular information processing model from the cognitive psychology domain is the Atkinson-Shiffrin memory model or multi-store model [14] which describes the information describes flow between three permanent storage systems of memory: the sensory register (SR), short-term memory (STM) and long-term memory (LTM). It is worth noting the similarity between the multi store model and a typical modern stream processing pipeline: In Figure 2-1 we attempt to highlight this similarity by mapping the activities and subsystems involved between what effectively is a prevalent cognitive psychology model and an architecture paradigm for a data stream processing application.

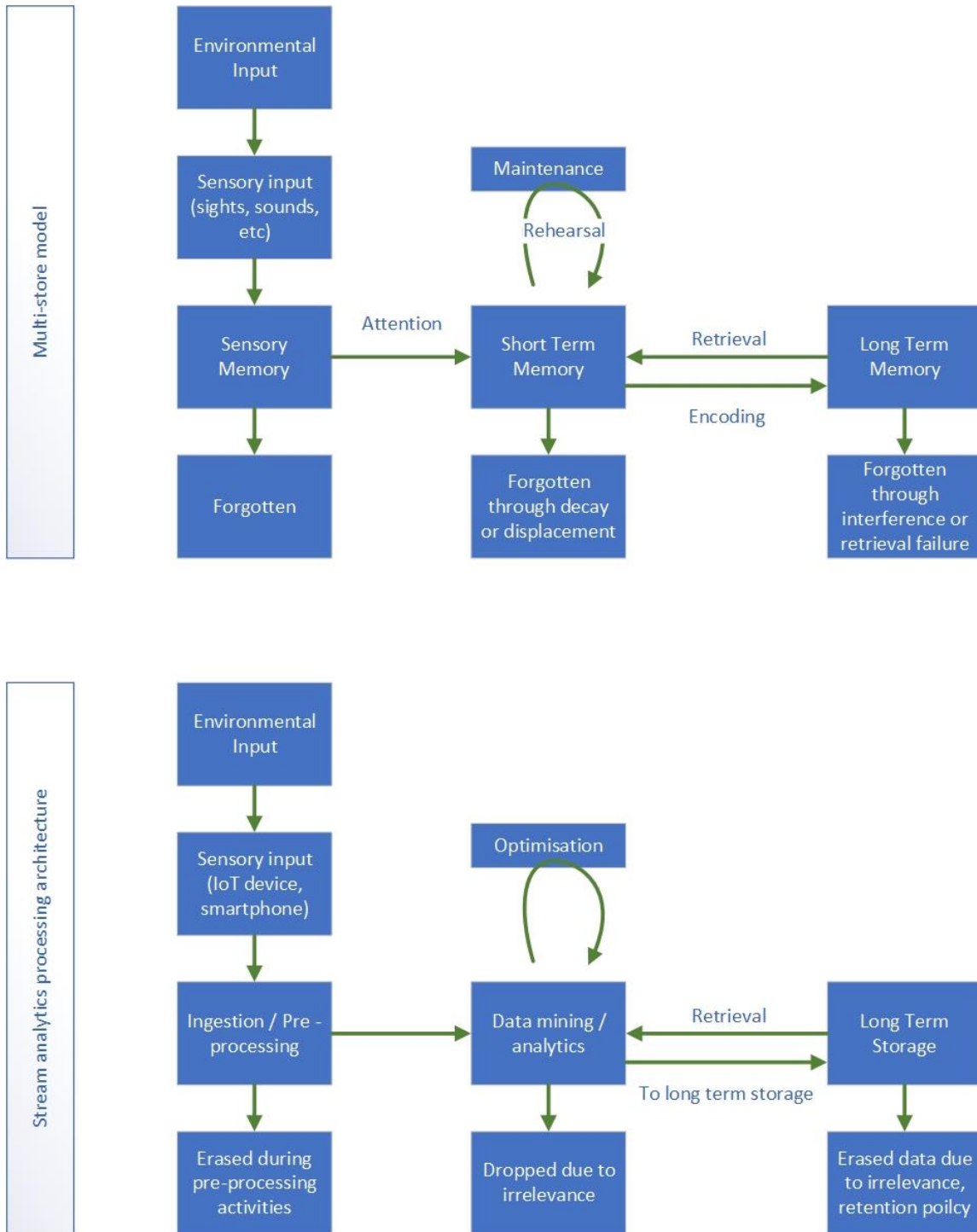


Figure 2-1: A comparison between the multi-store memory model and a typical stream analytics architecture.

At the highest level of abstraction, a real-time processing system consists of three elements:

- One or more data producers.
- The system which captures, processes and stores the incoming data.
- One or more data consumers.

In [16] a classification of different real-time systems is proposed primarily based on their latency and their tolerance for delay, that is the degree of system failure to meet its objectives if latency exceeds the expected threshold. Table 1 presents the proposed classification along with example applications for each class.

Classification	Example	Latency measurement	Delay tolerance
Hard real-time	Health monitoring applications in intensive care units.	Microseconds – milliseconds	None
Near real-time	Airline reservation system.	Milliseconds – seconds	Low
Soft real-time	Home automation.	Seconds - minutes	High

Table 1: Classification of real-time systems

Figure 2-2 displays a visualization of a real-time processing system with the three components discussed in the previous paragraph. Note that the consumption side (the data consumers in the diagram) are not necessary for the operation of a real-time processing system.

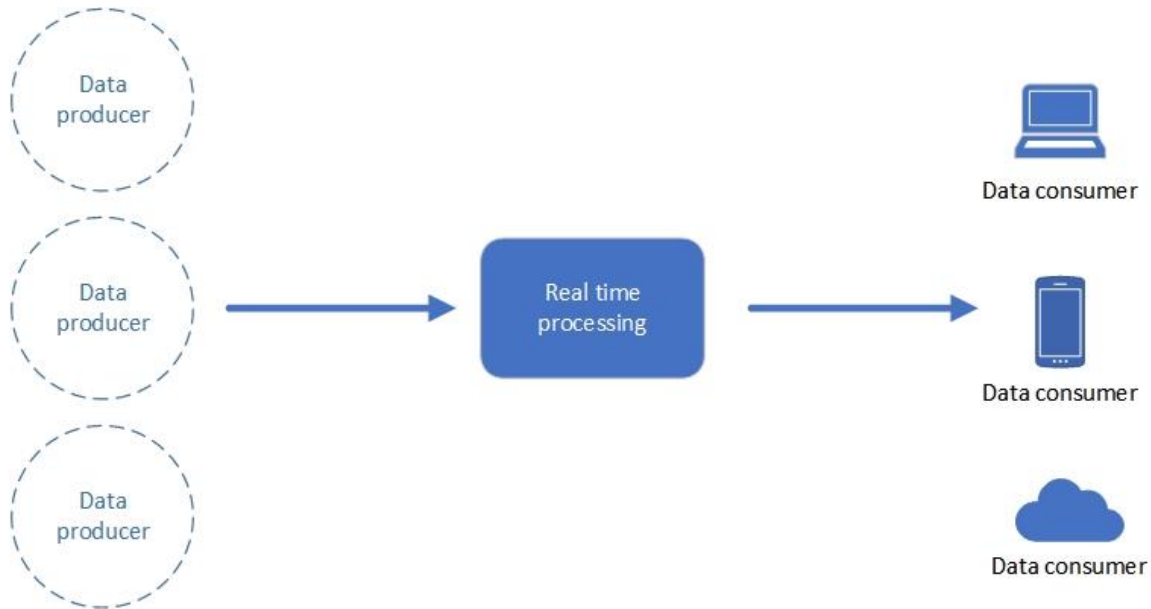


Figure 2-2: A generic real-time system with data producers, a real-time processing layer and data consumers

Stream processing systems which handle data volumes classified as “big data” have two major architectural paradigms which we will present in the following paragraphs:

- **Batch processing:** Referring to the processing of high-volume data (i.e. financial transactions) in batches and using a repetitive approach. Batch jobs can be executed with no end user interaction or can be scheduled to execute on a specific time frame.
- **Stream processing:** Referring to the processing of high-volume data that arrive (“data in motion”) and are processed in a real-time fashion. Stream processing also allows processed / analyzed incoming data to be fed into analytics tools or otherwise be made immediately available to data consumers.

Table 2 presents the primary differentiating attributes between the batch and real-time data processing paradigms as well as typical use cases that are usually supported by each one.

Batch processing	Real-time data processing
------------------	---------------------------

Data collection	Data are collected over a time frame before processing.	Data are collected in real-time, as they arrive in the system.
Dataset size	Limited by the size of the batch.	Unlimited – technically limited by the time the data are being streamed.
Performance – latency	Minutes to hours	Seconds to milliseconds.
Typical use cases	Transactions, Billing, Payroll applications, data transformations.	Traffic logs, Social media sentiment analysis, Log monitoring.

Table 2: A high-level comparison between batch and real-time data processing

In sections 2.1.1 and 2.1.2 we present the basic features of the batch and stream processing paradigms. The popularity and increased importance to business success of the ability to process data originating from streams in a continuous basis has given birth to multiple proprietary and open-source stream processing platforms. A selection of these platforms with each one's primary features is presented in section 2.6.

A data stream can be defined as an append-only sequence of timestamped items that arrive in a particular order. For applications involving publish / subscribe systems such as the one discussed in this dissertation, data are produced by several sources and consumed by consumers who subscribe independently to those data feeds, a data stream can be conceived as a sequence of events that are being reported in a continuous manner. A data stream may also be defined as a sequence of sets of elements, to cover scenarios where items arrive in batches, with each set containing elements that have arrived during the same predefined time interval [17].

A typical data streaming system with its major components is displayed in the diagram of Figure 2-3. An *input monitor* is responsible for handling the inputs from a variety of data sources. Streamed

data are stored in three partitions: temporary *working storage* (for example to execute a window query), *summary storage* for stream synopses (i.e. data aggregations), and *static storage* for metadata (e.g., physical location of each source). The query repository component is responsible for handling the query execution by allocating them into groups for shared processing. The query processor component communicates with the input monitor and may re-optimize the query plans in response to changing input rates. Finally, the result is streamed to an end user application, such as a dashboard which typically provides the functionality of further refining the output, for example to filter the query result based on an ad-hoc criterion [17].

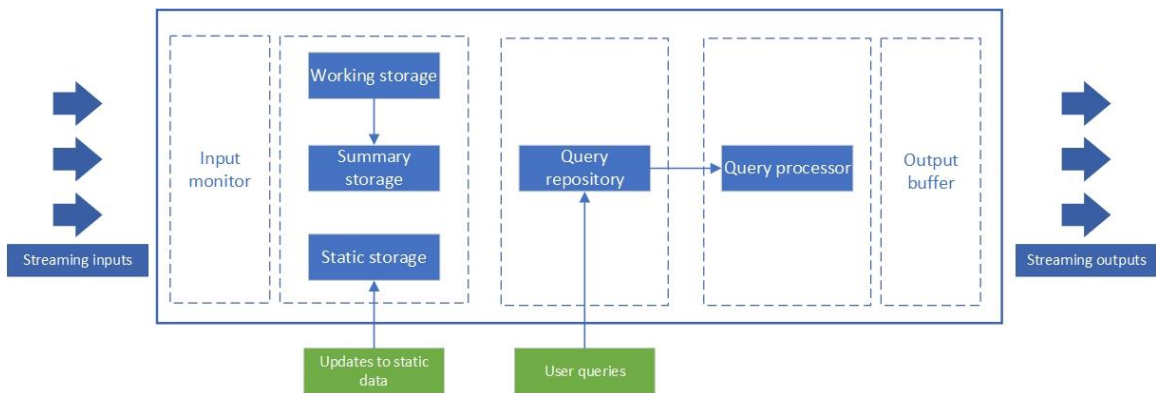


Figure 2-3: An abstract reference architecture for a data stream management system [17].

Using the definitions presented above, a data streaming model typically conforms to one of the following categories [18]:

1. **Unordered:** Individual items from various domains arrive in no particular order and without any pre-processing.
2. **Ordered:** Individual items from various domains are not preprocessed but arrive in some known order.
3. **Unordered aggregate:** Individual items from the same domain are preprocessed and only one item per domain arrives in no particular order.
4. **Ordered aggregate:** Individual items from the same domain are pre-processed and one item per domain arrives in some known order.

2.1.1 Stream processing

Stream processing is a big data processing paradigm that focuses on being able to instantaneously process and analyze data that are arriving from a device (producer) to another device (consumer). Referring to Table 1 where the different real-time systems are classified based on the time sensitivity aspect, a streaming data system can be defined as a “non-hard real-time system that makes its data available at the moment a client application needs it” [16]. This means that the client is not obliged to consume the processed data in real-time, but asynchronously, at the moment it’s needed. In [19] a streaming system is defined as “A type of data processing engine that is designed with infinite datasets in mind.”

A stream processing system typically is built to function with unbounded data, contrary to batch processing systems which are designed to handle bounded datasets. Unbounded datasets pose additional challenges to be tackled by the system [19]:

- They can arrive unordered with respect to their event time, which requires specific implementation to be done in order to analyze them in the context in which they occurred.
- The incoming data can be of varying event-time skew and as such the system needs to take into account that there will be a fraction of incoming data for a given event time within a specific time window.

It becomes obvious that the time factor is especially important for the successful implementation of a stream processing pipeline. The time factor can be classified as:

- The **Event time**, which refers to the time the event actually occurred, i.e. a timestamp on a log entry. Consideration of the event time is especially important for use cases such as billing applications or generally transaction related workflows.
- The **Processing time**, which refers to the time at which the event is observed in the system.

The skew between event time and processing time can be affected by variables such as shared resource limitations, the software implementation itself as well as features related to the specific dataset being ingested such as variance in throughput [19].

Figure 2-4 depicts the skew that can be observed between the event time and processing time in a stream processing system. In the figure, the X axis represents the event time. The Y axis represents the progress of processing time (the actual clock time observed by the system during the execution of the event processing activity).

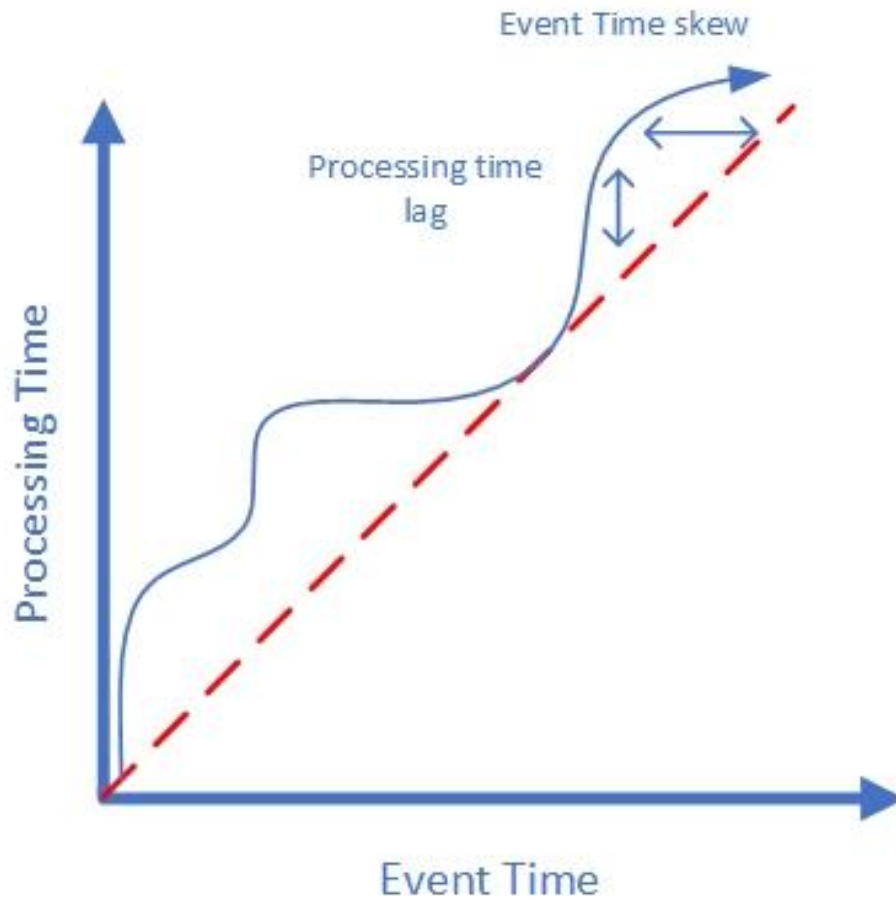


Figure 2-4: Processing time lag and event time skew in a stream processing system.

Typically, a streaming data pipeline's flow consumes a stream of messages, applies operations (i.e. transformations, aggregations, joins) to the incoming messages and publishes the output to another system or even another stream for further processing. Such a pipeline is composed of:

- One or more data source connectors, which handle the data that are ingested from the sources.
- One or more data sink connectors which extract the processed messages from the stream and publishes them to a "downstream" consumer, i.e. a data warehouse or a UI component.

Figure 2-7 presents a high-level overview of a stream processing data flow.

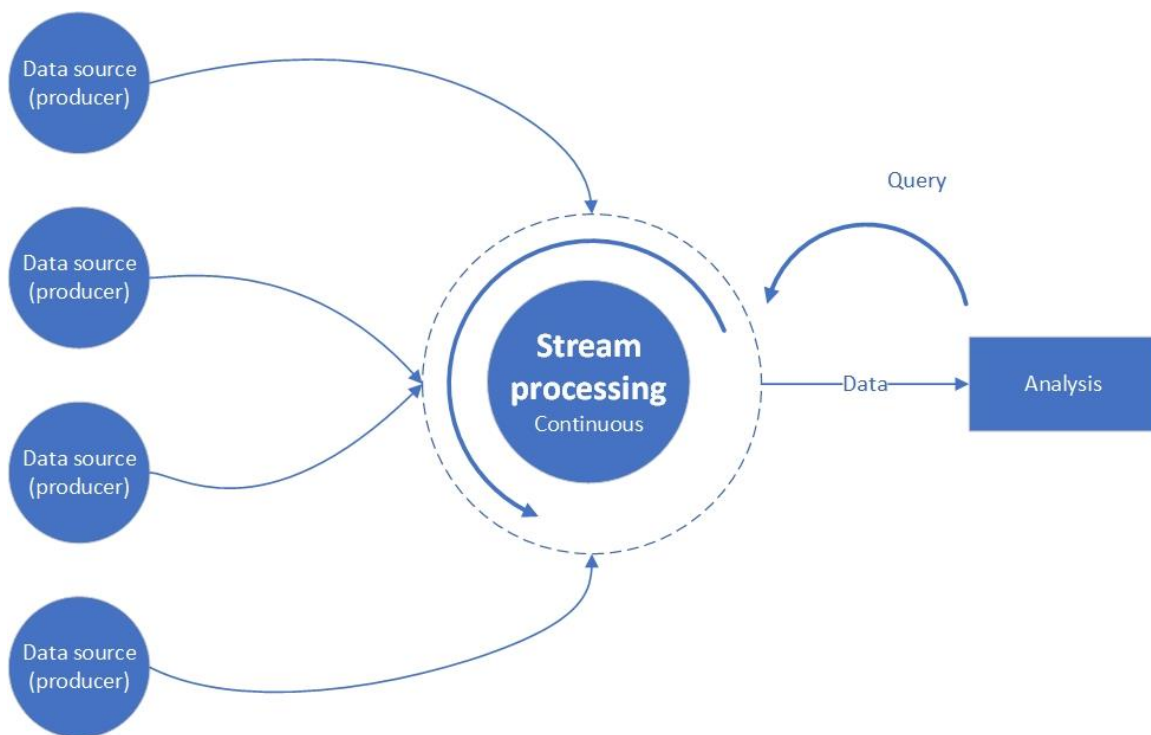


Figure 2-5: A high-level stream processing data flow.

2.1.2 Batch processing

In the batch processing paradigm, the data processing takes place in blocks (batches) of data that have been stored over a period of time. Each record in the batch is being processed with the same

algorithm and its results are published to the consuming systems before the next batch is processed in an iterative fashion. Depending on the implementation, a batch pipeline can be executed manually or recurringly based on a set of rules, for example when the batch reaches a certain size. This rule set sequence of steps can be handled by an orchestration component. The processing itself is usually handles by a parallelized job and may also include multiple iterative steps before the transformed results are published to a data store or other persistent storage for further usage such as reporting [20]. Figure 2-6 displays a high-level overview of a batch processing system described in this section. By its nature, batch processing implies a latency between the time the data appears in the pipeline's storage layer and the time it's made available in consuming systems. It is therefore important to note that batch processing may not be suitable for processing datasets that are time sensitive [21].

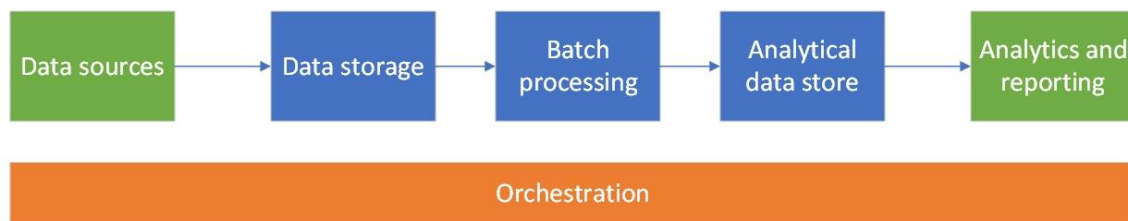


Figure 2-6: High-level overview of a batch processing system [20].

Frequent use cases that support the usage of a batch processing operation compared to streaming are the following [22]:

- When real-time data processing and results are not critical for the purpose they are needed (small time-sensitivity of the processed dataset).
- When very large volumes of data need to be processed with a resource intensive algorithm that requires access to the entire batch contents, for example a sorting algorithm.
- When a join operation is needed between relational database tables.
- Performing bulk operations in the dataset such as digital image resizing, conversion or other mass-editing activities.

Note that a batch processing system is considered to be significantly less complex compared to a stream processing system, as it does not require the implementation effort and resources required to always maintain connection to data sources and the data flow of a stream processing system.

Finally, Figure 2-7 presents a high-level overview of a typical batch processing data flow.

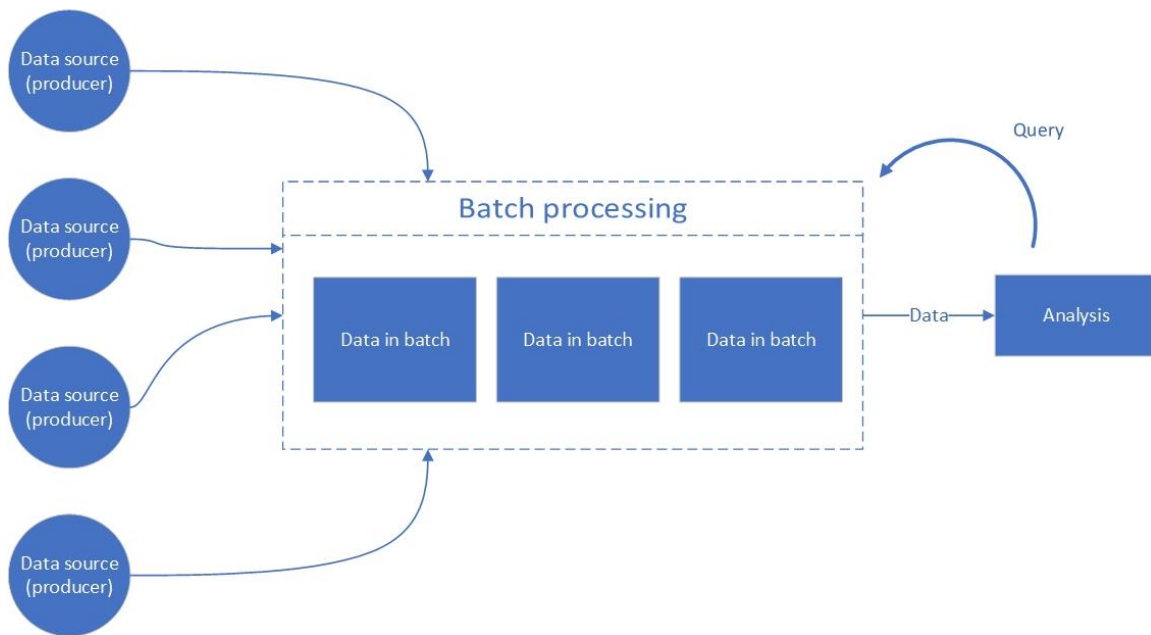


Figure 2-7: A high-level batch processing data flow.

2.1.3 A comparison between stream and batch processing

In this section we summarize the primary differences between the stream and batch processing paradigms. It is evident that there is no “best” approach as there are multiple factors to be taken into account, ranging from the availability of technical and computing resources to the nature of the business use case that the data processing will be required to fit into [23].

Factor	Batch processing	Stream processing
Hardware	More storage and computing resources to process large batches of data.	Less storage but more computing resources in order to maintain the system in an

		active state and guarantee real-time processing.
Latency / Performance	From minutes to days for more complex processing.	From milliseconds to seconds, as the time sensitivity is significantly lower for systems that depend on stream processing.
Dataset	Large batches of data (bounded set).	Continuous stream of incoming data (unbounded set).
Analysis	Complex computations and analysis algorithms	Simpler computations for reporting
Transactions	Each transaction is part of a group (the batch)	Each transaction is unique and stand-alone.

Table 3: A comparison between batch and stream data processing paradigms.

2.2 Stream processing challenges

An adequate stream processing system shall be able to handle system-level challenges related to the data that it manages that can be categorized as follows [24]:

Data ingestion: Handling massive streams of data can be a challenging task, requiring the system to have auto-scaling capabilities to handle the incoming data velocity. The type of incoming data (structured / unstructured) can affect in a major way the design of a stream processing system as typically unstructured data requires pre-processing activities (i.e. filtering, extraction).

Data management: Depending on the solution scope, the streaming system design must be able to handle persistent storage for the incoming data and potentially at different stages of the data lifecycle: For example, there may be a need to persist the originally streamed data for future

reference, analysis or auditing purposes and to also persist filtered or processed data for caching or future usage.

Data modeling: Low latency is a typical requirement for the processing capabilities of a stream processing system. Latency is affected by the data volume, variety, velocity and veracity and as such all these factors must be taken into account for the system design.

Data mining / analytics: The mining / analytics component of a data streaming solution must be able to handle heterogenous ingested data and produce the desired results (i.e. summaries, visualizations, dashboards) in a short amount of time, or even requiring near real-time publishing of the results.

2.3 Indicative business applications involving stream processing systems

Interest in processing and analyzing streaming data is reflected in the adoption of relevant techniques and technologies in a number of industries. In the following paragraphs we present sample business domains which are utilizing stream analytics in their operations. We expect that in the future, more businesses will transform their business models in an effort to reap benefits from analyzing real time data streams.

Social Media: Due to their popularity, social media platforms continuously generate large amounts of data that are being processed by numerous data services, tools and analytics platforms. Many of these platforms include stream analytics components in their systems to perform tasks such as text analytics, opinion mining and sentiment analysis [25].

Smart Cities: Smart City projects aim to improve the quality of life of different facets of communal living such as transportation, working environments, government services by utilizing digital technologies to collect data that are being continuously generated and using that data as inputs to improve decision making and to optimize the efficiency of city operations and services. In this context, real-time data and stream processing technologies can assist smart city projects to achieve

these objectives in areas such as traffic monitoring, real-time fault detection on critical equipment and public infrastructure monitoring [26].

Financial Services: The large volume of transactions that takes place daily in banks and other financial institutions is being processed using stream analytics solutions to extract insights and enhance their business intelligence capabilities. Popular uses of stream analytics techniques in the financial services can be found in the domains of money laundering/payment fraud detection, risk management and to survey the stock market for emerging trends.

2.4 Stream processing architecture patterns

A streaming data architecture can be conceived as a framework or a system designed and developed to ingest, process, and output large volumes of streaming data from a variety of sources. In this section we discuss streaming data architectures and elaborate on the Lambda and Kappa architecture, the two major architecture patterns for data / stream processing systems that are currently being used in the industry. In Information Technology, a System Architecture is the conceptual model that defines the structure, behavior, and more views of a system. An architecture typically consists of components and subsystems that cooperate to implement the overall system that is envisaged. The primary objectives of an architecture are to *explain* the structure of the underlying components / software, to *guide* the person who will do the actual implementation towards following a set of predefined patterns and to *enable* the requirements which directed the creation of the architecture [27].

When abstracted to its basic components, a simplified data processing architecture can be visualized as in the conceptual architecture diagram shown in Figure 2-8. The model presented here also follows the data flow patterns discussed in sections 2.1.1 and 2.1.2 for the batch and stream data processing paradigms.

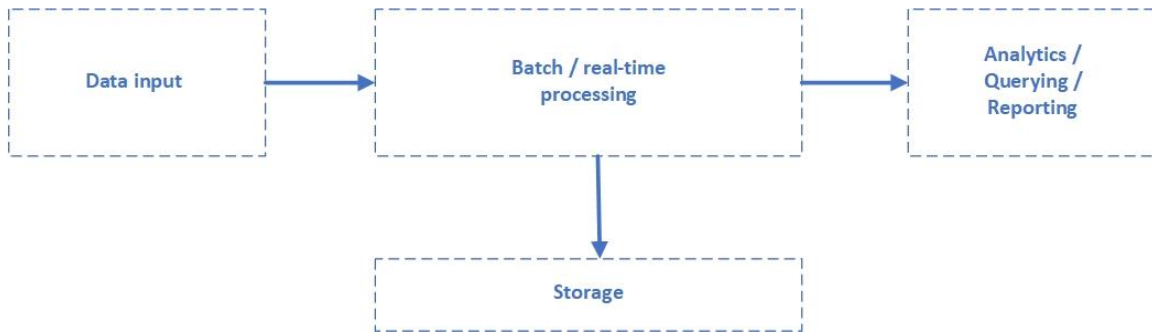


Figure 2-8: A high-level data processing architecture. More details on the batch / real-time / stream processing architectures have been presented in section 2.1.

For data streaming and especially related to Big Data two primary architecture patterns have emerged the last decade and attempt to tackle common problems and challenges that arise in the data processing domain.

Table 4 presents a high-level comparison of the main features of the primary emergent data processing architecture patterns [28], the Lambda and Kappa architectures. Each pattern is presented and elaborated in the next sections 2.4.1 2.4.2. Finally, section 2.4.3 presents a comparison of the main characteristics of each architecture.

Criteria	Lambda architecture	Kappa architecture
Layers	Batch, Real-time, Serving	Stream, Serving
Data processing	Batch and real-time	Real-time only
Processing guarantees	Yes, in batch but approximate in streaming	Exactly once processing
Re-processing paradigm	In every batch cycle	Only when code base changes
Real-time accuracy	Not guaranteed	Guaranteed

Table 4: High-level comparison of the main characteristics of Lambda and Kappa architectures for stream processing applications

2.4.1 Lambda architecture

The Lambda architecture is a way to process high-volumes of data and provides access to batch-processing and stream-processing methods by means of a hybrid approach which will be presented

in this section. It appeared in 2012 and its components, functions and purpose are thoroughly presented in [29], which is considered to be the “foundational” book on this architecture². The Lambda architecture’s primary objective is to have a data processing system that is scalable, robust, fault-tolerant that is linearly scalable and allows for write and read operations with low latency.

Figure 2-9 displays an overview of the three-layered Lambda architecture.

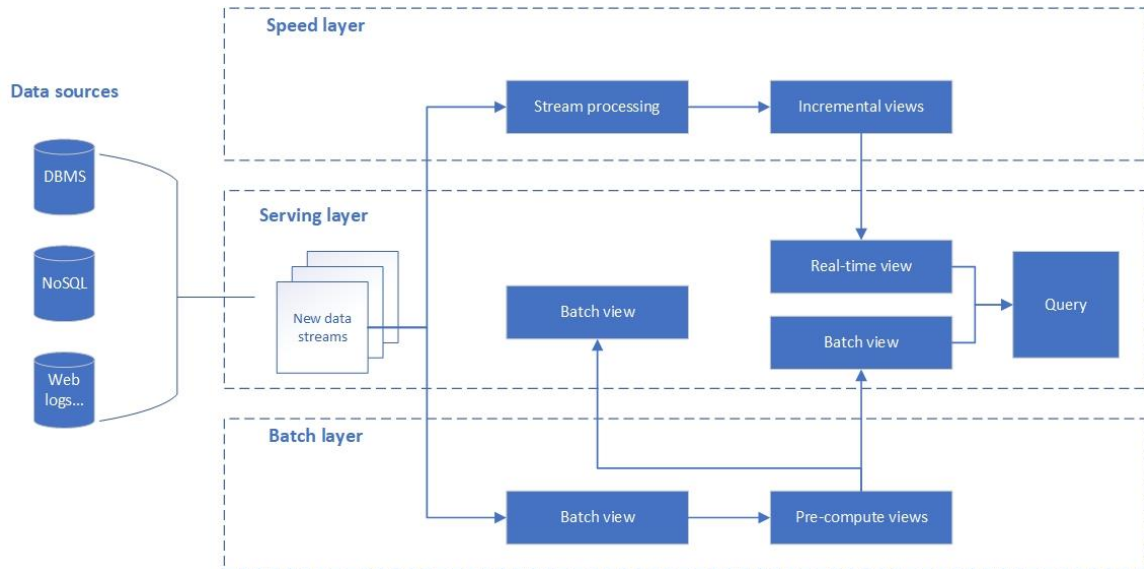


Figure 2-9: Overview of the Lambda architecture [29].

In [29] a list of attributes is presented that an architecture based on the Lambda model must fulfill to be deemed successful:

1. **Robustness and fault-tolerance:** The system must behave correctly despite infrastructure failures or data quality issues. The fault tolerance aspect revolves primarily around the preventing human errors, by building immutability and re-computation aspects and providing clear and simple recovery mechanisms.

² See also the precedent article [31] which established the foundations on the Lambda architecture: “How to beat the CAP theorem”.

2. **Low latency reads and updates:** the system shall be able to achieve low latency updates when dictated by the business use case and without compromising its robustness. Low latency is handled by the architecture's speed layer.
3. **Scalability:** The system shall be able to scale horizontally across all its layers, by adding more compute resources.
4. **Generalization:** The system shall be able to support a wide range of applications. Since the Lambda architecture is based on functions over data it is possible to generalize the processing to accommodate a large number of use cases.
5. **Extensibility:** The system shall be able to be extended with additional functionalities with minimal development effort. In addition, migration of old data in new formats should ideally be supported.
6. **Ability to execute ad hoc queries:** As large datasets can be examined and processed in different ways to extract insights. Therefore, the system shall be able to support the arbitrary execution of queries over the processed dataset without requiring development effort.
7. **Debuggability:** The system shall be able to provide necessary information for debugging activities to be performed, so those errors can be adequately traced. This aspect is accomplished in the Lambda Architecture through the batch layer by preferring to use re-computation algorithms when possible.

The Lambda architecture consists of a series of layers with each layer building upon the functionality implemented by the previous layer, with the three-layer system consisting of:

1. **The batch layer:** Stores the master copy of the dataset to be processed and precomputes batch views on it. In addition, the batch layer's responsibility is to store an immutable, master dataset which will be regularly updated in the future, and compute arbitrary functions on that dataset.

2. **The speed layer:** Updates whenever the batch layer finishes computing a view. Effectively the goal of the speed layer is to ensure that any new data that arrived while the pre-computation view was executed in the batch layer and therefore not represented there, have the same functions executed over them. Thus, the speed layer compensates for the “missing” data from the batch layer. In that sense the speed layer is similar to the batch layer as it produces views based on the received data – the difference being that the speed layer only looks at recent data compared to the batch layer which processes the entire dataset.
3. **The serving layer:** Loads the views emitted by the batch layer as a result of the functions it executed over the data. The serving layer is usually a distributed database that loads a batch view and enables random read queries. The layer's database also supports batch update and random read operations.

Finally, a **Query** component is responsible for submitting the end user's queries to both the serving layer and the speed layer and consolidating the results. This process provides the benefit to the end user to execute a complete query on all data, both in the Batch and the Speed layer, thus providing a *near real-time* result.

It should be noted that the Lambda architecture does have flaws that have been considered before implementing it for a big data processing activity [28]:

- The business logic has to be implemented twice: Once in the batch and once in the speed layer, leading to the need to maintain two separate codebases.
- The processing is asynchronous, even though it is compensated by the existence of the speed layer. Hence the computed results are expected to display even a small degree of inconsistency compared to the original data.

2.4.2 Kappa architecture

The Kappa architecture was originally proposed in [31] as an alternative over the Lambda architecture, aiming to set an architecture that would handle the entire computation over a single streaming layer, thus simplifying the overall architecture. The premise of the Kappa architecture is that the stream processing component of the data processing system (the “speed layer” of the Lambda architecture) can be adjusted to handle the entire computations over the data set and thus eliminating the need for including the “batch layer” in the architecture. It can therefore be conceived as an “improvement” on the Lambda architecture which comes with the added benefit of being able to handle both streaming and batch jobs with the same code base and infrastructure, while improving the latency requirement as the entire dataset is processed in real-time. In [32] it is proposed that a well-designed stream processing system should actually provide a superset of a batch processing system such as the Lambda architecture³.

Generally, the processing from solely a stream processing component is done as follows [31]:

1. The dataset for processing is loaded into a message queuing system such as Apache Kafka.
2. Processing is done by a stream processing job that starts processing from the beginning of the data stored in the Kafka topic and direct the output of the computation to a new output table (reprocessing is effectively done by “replaying” the Kafka topic whenever it’s needed).
3. When a reprocessing is needed, a second instance of the stream processing job is initialized and directs the output data to a new database output table (output table $n + 1$ in Figure 2-10)
4. The application (i.e. a dashboard) that uses the processed data is switched to read / query the new table.

³ This is also one of Apache Flink’s primary use cases: It exposes two execution modes (batch / streaming), where the batch mode is targeted at bounded datasets and the streaming mode is target at both bounded and unbounded datasets. See also the relevant documentation at: https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/datastream/execution_mode/.

5. Finally old job versions and old output tables are deleted, and the process repeats for new incoming data.

Note that unlike the Lambda architecture, the Kappa architecture is focused at processing data and it does not foresee their persistent storage. It is possible however to extend the architecture to include a persistent storage component that comes at an additional resource, development and maintenance cost.

Figure 2-10 provides an overview of the Kappa architecture. Notice the effective “merging” of the batch and speed layers presented in Lambda architecture (Figure 2-9) into a single stream processing layer.

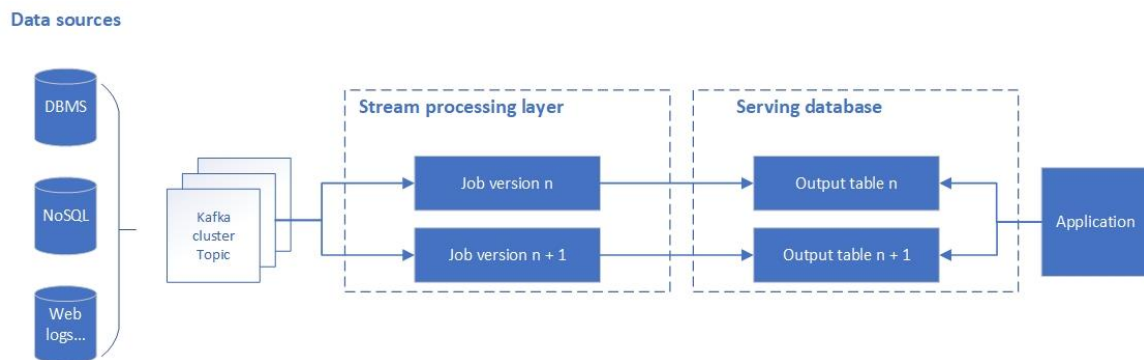


Figure 2-10: Overview of the Kappa architecture [31].

The primary appeal of the Kappa architecture is that it can be used to perform both real-time and batch processing using the same technology stack. This results in simplified development, testing and maintenance processes, accompanied by significant cost reductions. This is largely attributed to the possibility of dropping the batch layer can be dropped and instead only employ a stream processing component, with the primary objective of achieving real time or near-real time processing and at the same time simplifying the technology stack as the same stack is responsible for processing all incoming data.

2.4.3 A comparison between the Lambda and Kappa architectures

In the two previous sections of this chapter, we presented a summary of the two prevalent data processing architectures that are currently in use, dubbed “Lambda” and “Kappa” along with their basic functions and components. It is apparent that in a vacuum, the Kappa architecture can be seen as a strict improvement over its predecessor as it can perform the same operations that Lambda architecture does with arguably less resources and less development effort. However, when designing such a system, a major factor to be considered is the nature of the data that will be processed (i.e. whether it is a bounded or unbounded dataset) which will itself be largely dictated by the overall business use case that necessitates the implementation of such a system. Another factor to consider is usually the pre-existing infrastructure and other systems that will need to cooperate with the processing system, which may restrict the choice towards one or the other way. So, for example if computation – heavy operations which occur on a very specific time window such transaction reconciliation operations the Lambda architecture might be more suitable for the task. On the other hand an operation that relies heavily on data arriving at real time and need to be processed immediately may warrant the implementation of system based on the Kappa architecture.

	Lambda architecture	Kappa architecture
Fault tolerance	Yes	Yes
Scalability	Yes	Yes
Persistent storage	Yes	No
Layers	<ol style="list-style-type: none"> 1. Batch 2. Speed 3. Serving 	<ol style="list-style-type: none"> 1. Real-time 2. Serving
Data processing	Batch and streaming	Streaming

Processing guarantee	Yes for batch data, approximation for streaming data.	Exactly once processing.
Re-processing paradigm	In every batch cycle	Only when code base changes.
Real-time accuracy	No	Yes

Table 5: A comparison of the primary attributes of Lambda and Kappa architectures.

A list of pros and cons which were discussed in this section is presented in Table 6.

	Lambda architecture	Kappa architecture
Pros	<ul style="list-style-type: none"> • Offers a balance between speed and reliability. • Batch layer manages historical data with a fault tolerant distributed storage, thus making the system more resilient if a failure occurs. 	<ul style="list-style-type: none"> • Effectively only the “speed layer” is required to function. • Re-processing is required only when the code base is updated. • Can scale horizontally by adding more compute resources. • Single processing layer is less intensive on development and maintenance effort.
Cons	<ul style="list-style-type: none"> • Requires a more complicated code base due to the need to implement and maintain twice the same processing algorithm, once for each layer. 	<ul style="list-style-type: none"> • Requires errors handling to cover scenarios where data must be reprocessed or reconciled.

Table 6: A collection of pros and cons for the Lambda and Kappa architectures.

2.5 Architecture principles for a stream analytics solution

The Open Group [33] defines an architectural principle as "general rules and guidelines, intended to be enduring and seldom amended, that inform and support the way in which an organization sets about fulfilling its mission". At the solution level, we can similarly define a set of principles formatted as high - level requirements that must be met by a stream analytics solution to successfully fulfil its purpose. The sets of architecture principles proposed in [34] result in the creation of a succession of layers in order to allow the composition of a solution which can combine different data analytics technologies and allow each layer to independently communicate and cooperate with technologies in the adjacent layers. Based on the above we can define the following principles:

1. The solution shall be able to collect both structured and unstructured data generated by one or more sources. Data collection shall be done in the solution's **collection layer**.
2. The solution shall be able to store incoming data ingested in the **collection layer** or migrated data, both for short-term and for long-term duration in the storage layer.
3. The solution shall contain an **analytics layer** whose primary purpose will be to provide a cooperative and scalable distributed programming framework to process the data streams that are ingested in the collection layer.
4. The solution shall perform data mining, prediction and other ad-hoc tasks created by its users in the **analytics layer**.
5. The solution shall contain an **application layer** whose purpose is to realize the business needs from different users and present the output of the analysis layer (i.e., with a dashboard accessible to end users).
6. The solution shall be able to accept input data in streams and in batches.

7. The final output of the solution such as analytics results for value extraction, knowledge discovery and visualization shall be published to the **analytics layer** and made available to end users.
8. The solution shall be able to handle analytics in an abstract way such that it is possible to adapt it to different analytics needs.

Figure 2-11 proposed in [34] displays a five-layered architecture for a stream analytics system based on the above architecture principles.

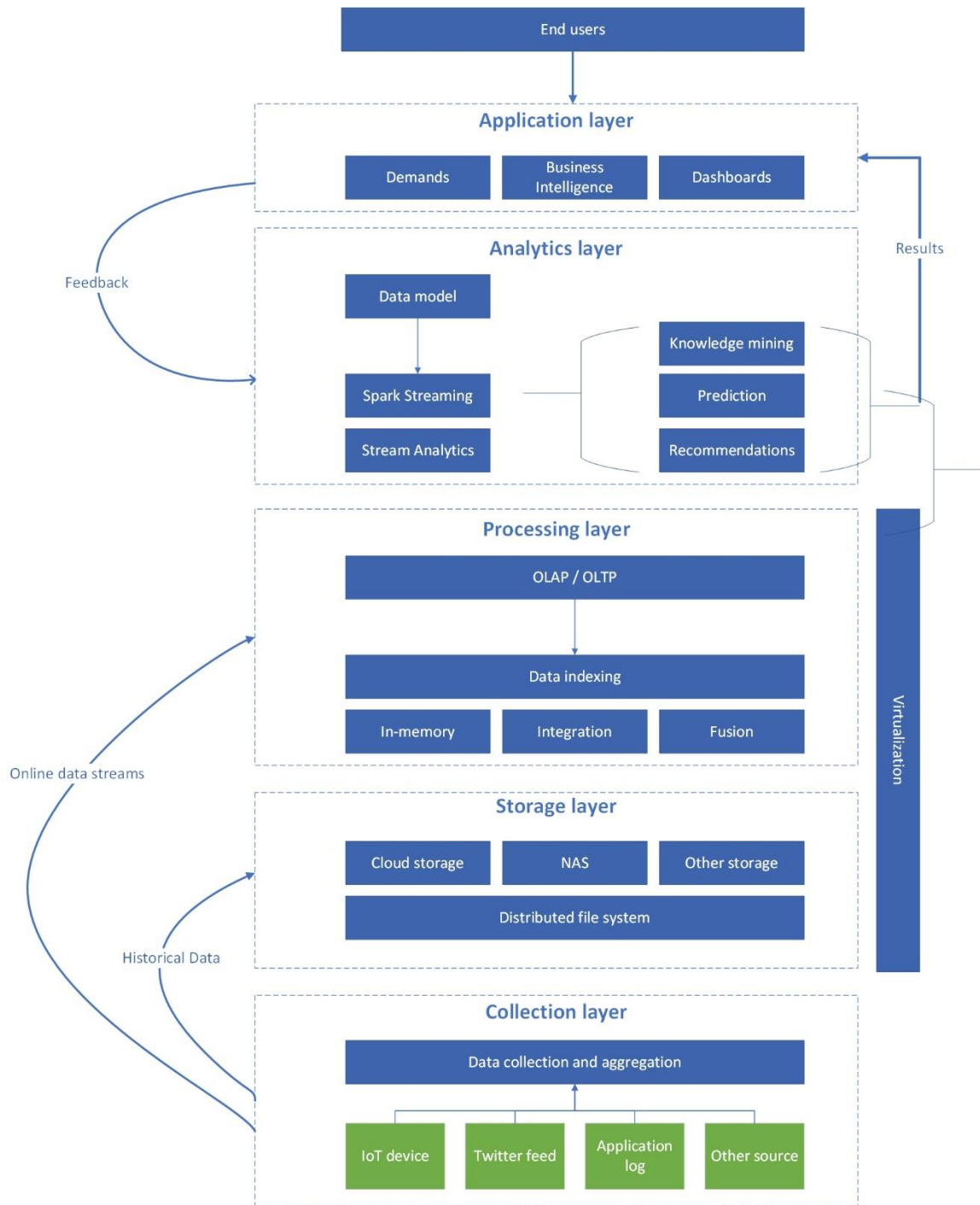


Figure 2-11: A five-layer architecture pattern for a stream analytics application [34].

2.6 Open-source and commercial streaming solutions

As discussed in section Chapter 1, the proliferation of big data has given birth to a multitude of open-source and commercial applications targeted at managing big data streams. This section presents a brief overview of the major solutions currently in the market.

2.6.1 Open – source streaming solutions

2.6.1.1 Apache Storm

Apache Storm is a distributed real-time computation system which can process unbounded streams of data [5]. Typical use cases for Storm are real-time analytics, ETL applications and social media feed processing. Architecturally, Apache Storm is composed of three components: Topology, Stream and Spout. A Storm application is designed as a "topology" in the shape of a directed acyclic graph (DAG) where the spouts and bolts function as the graph vertices. In this graph, edges are represented by streams which direct data flow between the nodes. A stream is an unbounded sequence of tuples that is processed and created in parallel in a distributed fashion. In a Storm topology, a spout is a source of streams. Typically, a spout reads tuples from an external source (for example an application log) and emit them into the topology. Depending on the configuration, a spout can be capable of replaying a tuple if it failed to be processed by Storm, or "forget" about the tuple as soon as it is emitted. Processing in a topology is done with bolts. Bolts can do operations such as filtering, functions, aggregations, and joins [5]. In its entirety, the storm topology acts as a data transformation pipeline with the advantage of being able to add multiple transformation steps over a stream source depending on the computation needs.

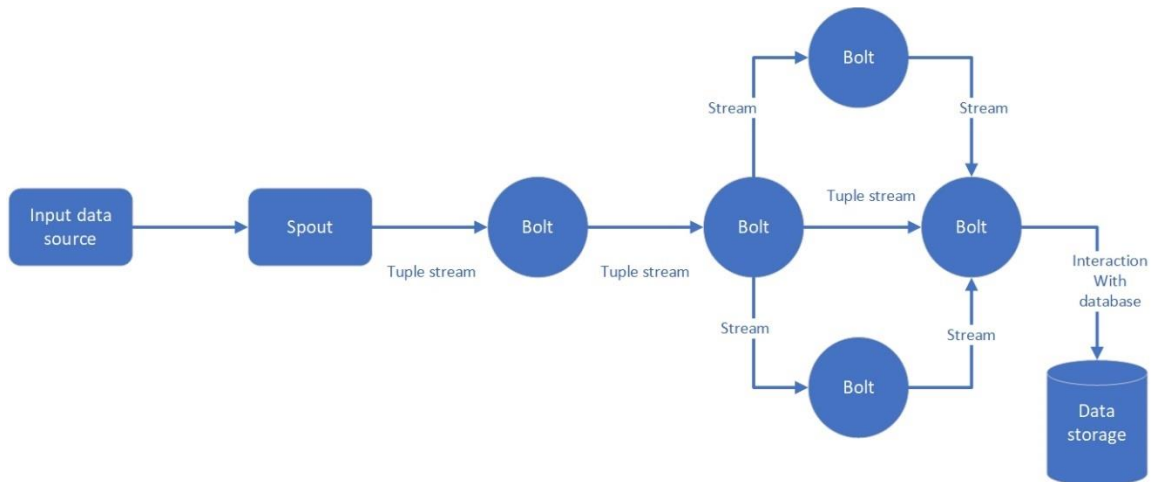


Figure 2-12: Apache storm topology overview [5]

2.6.1.2 Apache Flink

Apache Flink is a framework and distributed processing engine for stateful computations over unbounded and bounded data. Unbounded data streams are represented as having a defined start time but no end thus they are not expected to terminate and continuously generate data [7]. Bounded data streams have a defined start and end time and can thus be processed by Flink by ingesting all data before any computations are performed. For stream processing operations, Flink utilizes the `DataStream` API which provides functions such as windowing, aggregations, joins and also. In addition, Flink also features two relational APIs, the `Table API` and `SQL API` to handle relational data streams. Streams processed by Flink are fault tolerant as Flink has the capacity to use a combination of stream replay and checkpointing.

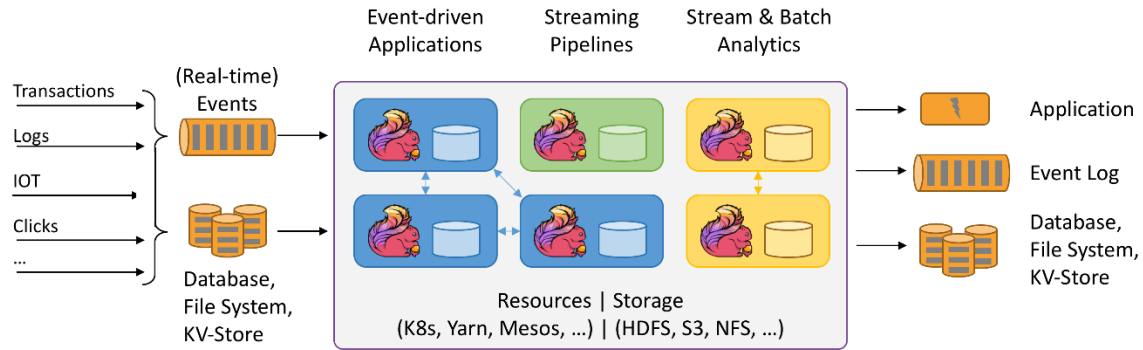


Figure 2-13: High-level overview of the Apache Flink stream processing solution [7].

2.6.1.3 Apache Samza

Apache Samza is a distributed stream processing framework that enables the creation of stateful applications that process real-time data from multiple sources [8]. The input unit that Samza processes is the “stream”, which is composed of immutable messages of a similar type or category. A stream can be read by any number of consumers either simultaneously or in separate time frames. The transformation on a set of input streams with the purpose of outputting a message to an output stream is called “job”. A Samza stream application can process messages from input streams, transform them and emits the results to an output stream or a database. Samza is able to handle a large volume of load by parallelizing streams into partitions and jobs into tasks [35]. Architecturally, Samza consists of three layers: a streaming layer based on Kafka, an execution layer based on Yarn and a processing layer which utilizes the Samza API. Both the execution and streaming layers can be switched for other applications, thus offering Samza increased flexibility in terms of fitting it into an existing data warehouse or similar infrastructure [36].

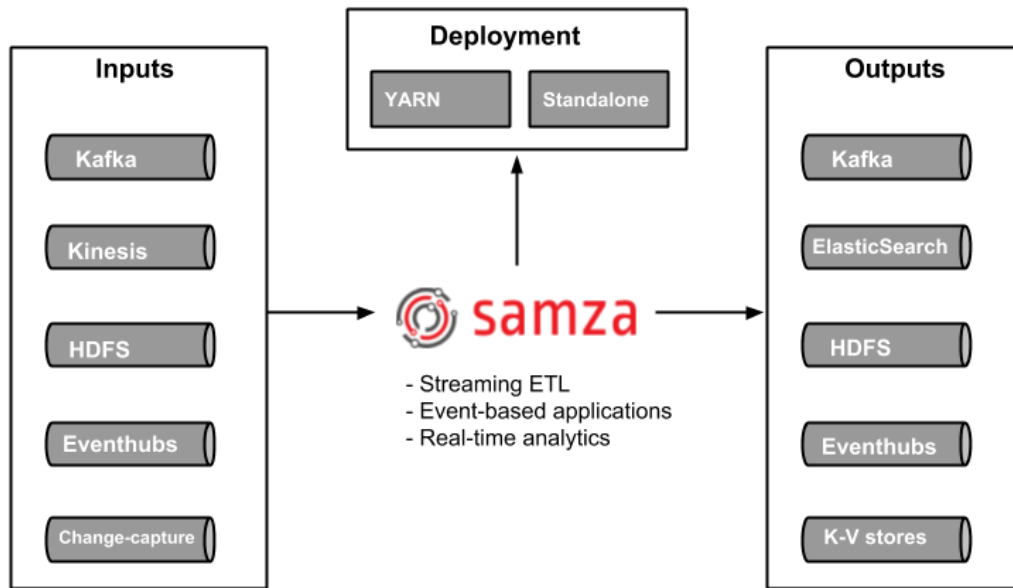


Figure 2-14: High-level overview of the Samza stream processing framework [8].

2.6.2 Commercial streaming solutions

2.6.2.1 Amazon Kinesis

Amazon Kinesis [9] is a SaaS product whose main function is to process massive data streams in real-time. Typical applications of Amazon Kinesis include real-time data ingestion from application logs, IoT telemetry and social media feeds into a variety of outputs such as databases and data lakes. A Kinesis stream consists of data records organized into tuples. Kinesis has a client library (KCL) which can be used to create consumer and worker components [37]. Therefore, KCL acts as an intermediary between the record processing logic and the incoming data stream. The KCL Worker functions as a client application, which receives the Kinesis stream performs the required processing and sends it to a specified output. The KCL Consumer application's purpose is to read and process records from data streams ingested from the KCL worker. Kinesis has the typical features offered by big data streaming products such as load-balancing by auto-scaling, fault tolerance by check-pointing and a configurable data retention period.

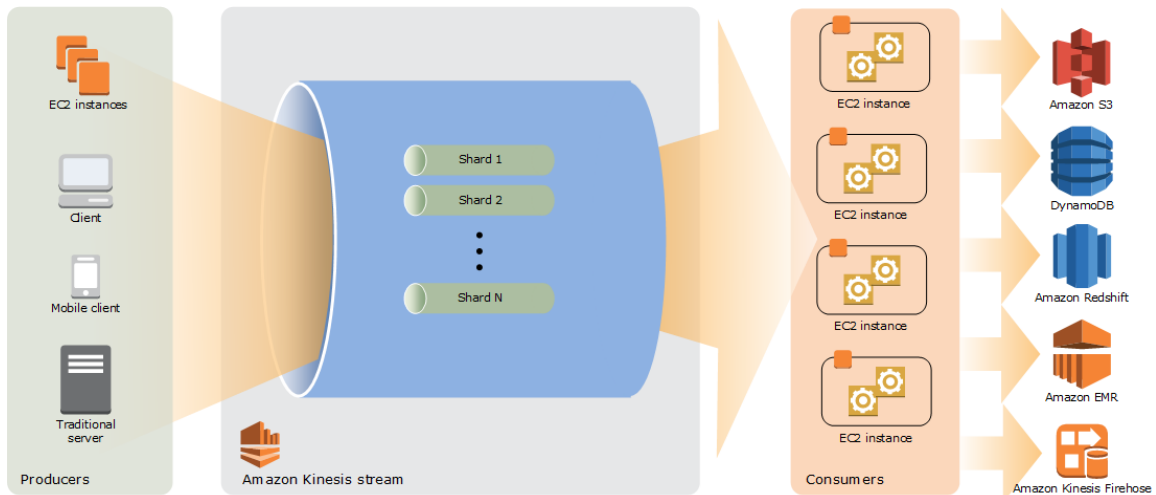


Figure 2-15: High-level overview of the Amazon Kinesis Stream product [38].

2.6.2.2 IBM InfoSphere Streams

IBM InfoSphere Streams [12] is a SaaS product which consists of a programming language, an API, an integrated development environment, and a runtime system that can run the applications on a single or distributed set of resources either hosted on IBM's cloud or on-premises. A stream processing application built with InfoSphere Streams consists of tuples, data streams, operators, processing elements (PEs), and jobs. A tuple represents an individual piece of structured or unstructured data in a stream with a data stream representing a running sequence of tuples. The data stream is processed by the operator component which produces an output stream. The operator - stream relationship is then broken down into a set of individual processing elements which are all packaged into a Job component. Stream applications are developed with the Streams Processing Language (SPL) which is proprietary to IBM or with Java and Python client libraries.

2.6.2.3 Azure Stream Analytics

Azure Stream Analytics (ASA) is a SaaS product of Microsoft's Azure cloud platform [10]. It is a real-time analytics engine that can handle processing of big data streams originating from input sources such as logs, IoT telemetry data and sensors and output results to multiple outputs such as data bases and data lakes.

An ASA's functional unit is the "job" which consists of an input, query, and an output. The job can be configured to ingest data from Azure Event Hubs (or Apache Kafka), Azure IoT Hub, or Azure Blob Storage. The processing query is based on the SQL query language, can be used to perform filter, sort, aggregation, join and other functions over the incoming stream. An ASA job can be configured with one or more outputs for the transformed data such as storage, power BI dashboards, a Service Bus topic or a function that uses the output as a trigger for a downstream application.

Azure Stream Analytics is the product that we have chosen to design a streaming solution architecture for this dissertation and is further elaborated in section 3.4.1.

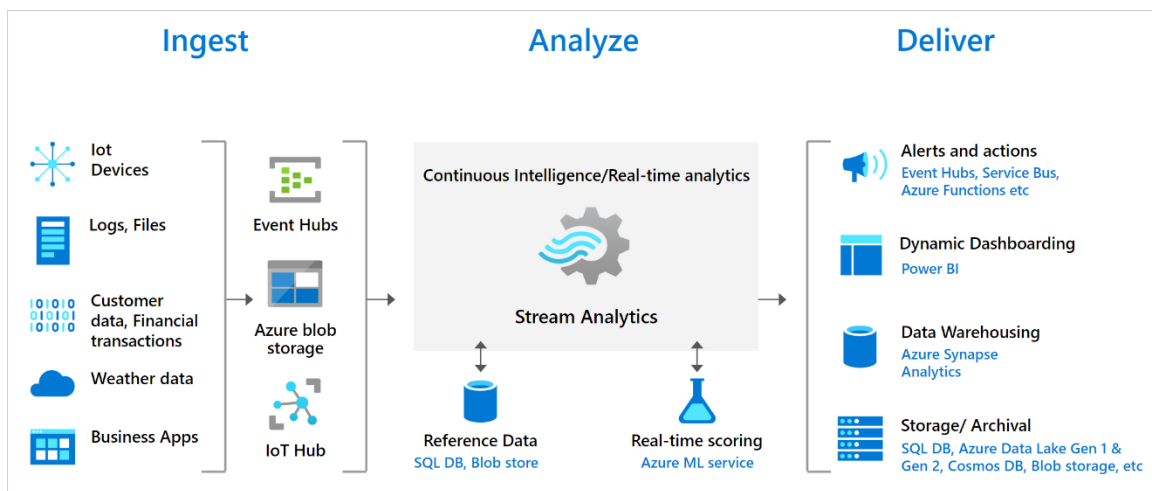


Figure 2-16: High-level overview of the Azure Stream Analytics product.

Chapter 3. Solution architecture and design

This chapter presents the architecture of our stream processing solution, outlines its components and discusses design and configuration choices. The solution will be built with components from the Azure cloud platform and the custom development that was needed was done in the Python programming language⁴. The purpose of the solution is to facilitate ingestion of data from multiple sources, store the incoming data as events in a message queue, perform analytics operations and publish the results in a dashboard as well as persist them in a data store for future reference or reprocessing.

The solution design is influenced by the 5-layer architecture and the relevant architecture principles proposed in [34] and presented in section 2.5. It is also based on the Kappa architecture presented in section 2.4.2 as we will be primarily working with streaming data that will be fetched from Twitter utilizing the Twitter streaming API. It should be noted however that with the proposed solution it is possible to facilitate scenarios where bounded datasets must be processed. We

⁴ Refer to the annexes for details of the technical implementation and the solution deployment guide:

- Development environment,
- Solution deployment guide

demonstrate that scenario – also for benchmarking purposes, in section **Error! Reference source not found.**

Figure 3-1 outlines an architectural blueprint for a general-purpose data stream processing solution.

Such a solution typically consists of the following components [16]:

- **Data producer:** This component represents the entities that generate the data that are captured and analyzed. For example, smartphone devices and applications, IOT devices, user activity on a system.
- **Collection tier:** This is the entry point for incoming data to the data streaming solution. Typically, the data collection is done using a commonly used interaction pattern, such as the request / response, publish / subscribe or the stream pattern.
- **Message queuing tier:** This tier is responsible for decoupling the collection and the analysis tier of the system by employing a three-tiered producer / broker / consumer layer to handle incoming data.
- **Analysis tier:** This tier is responsible for executing queries in the ingested data such as aggregations transformations and joins.
- **Long term storage:** This tier is responsible for persisting incoming data either in unprocessed or processed forms (or both) for future reference and re-processing.
- **In-memory data store:** An in-memory data store can be used to facilitate quicker read operations from the data access tier.
- **Data access tier:** Data consumers can access the processed data via this tier. Depending on the solution, this can be implemented with a variety of communication patterns, such as exposing API methods, Remote Procedure Calls (RPC).
- **Data consumer:** This component represents the entities that consume the processed data, for example smartphone applications, dashboards or other systems that further process or analyze the data.

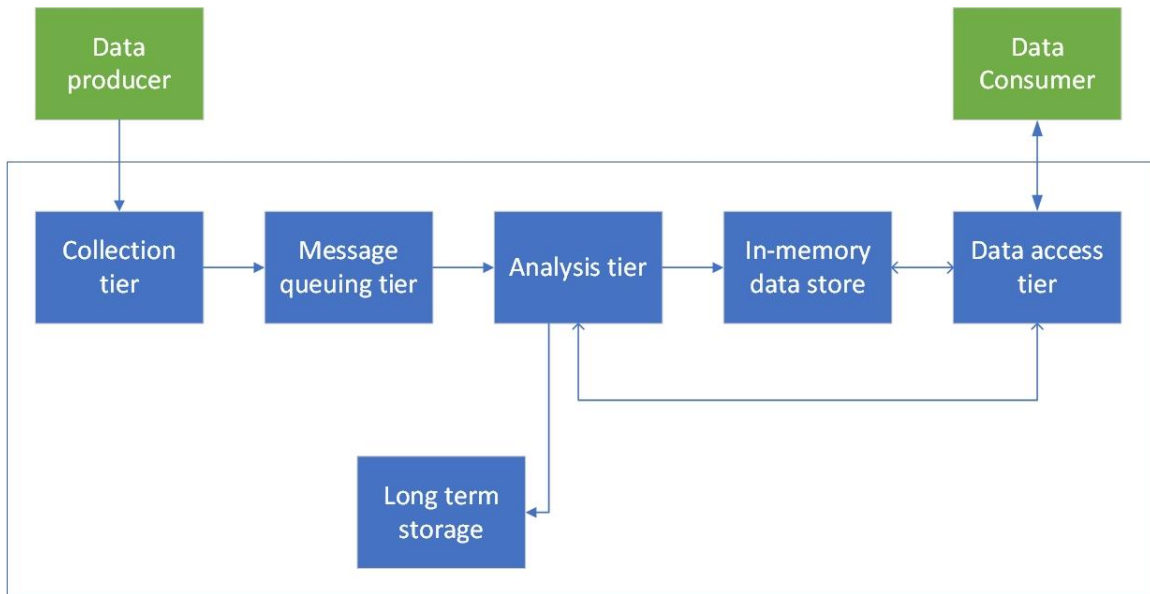


Figure 3-1: Architectural blueprint for a generic streaming data solution [16].

The solution architecture consists of six distinct conceptual layers which will be further analyzed in the following sections:

- Producer
- Ingestion
- Processing
- Storage
- Analytics / Reporting
- Infrastructure monitoring

Figure 3-2 presents a conceptual architecture diagram of the solution. As discussed in section Chapter 2, the solution architecture is modeled after the Kappa architecture pattern, which will enable the processing component to rely on just a single code base. In addition, the decoupled nature of the rest of the components enables us to modify them with minimal impact to the rest of the solution.

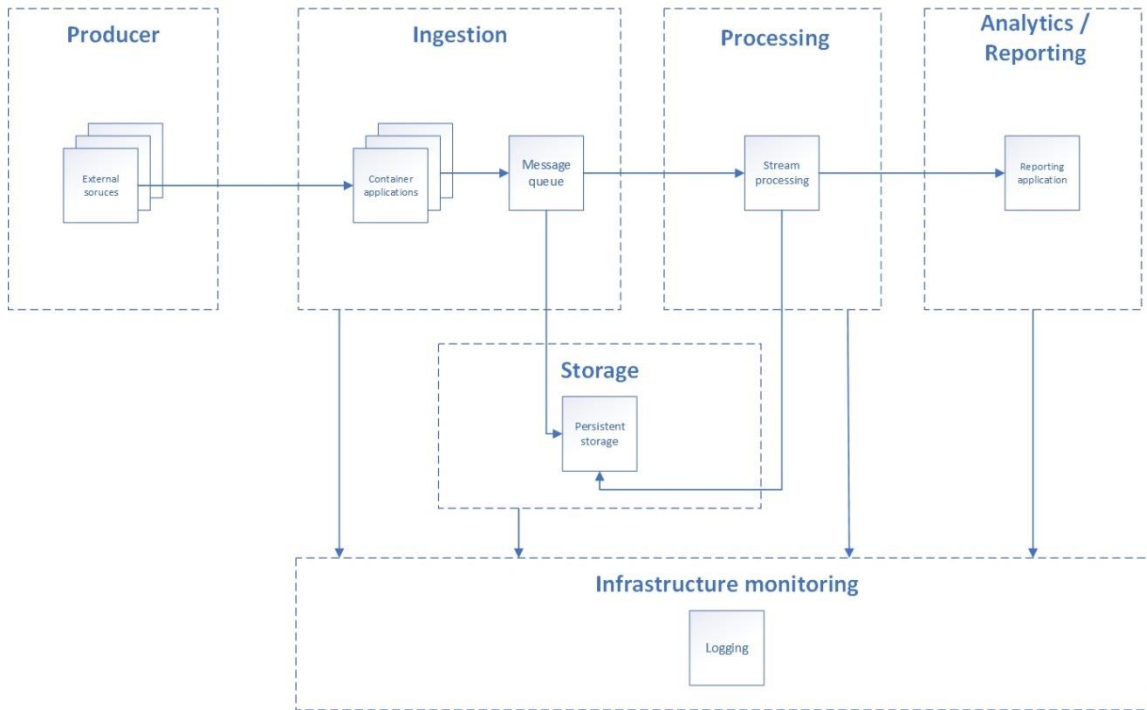


Figure 3-2: Conceptual architecture diagram of the architecture for the solution presented in this thesis.

Architecture layer	Contents
Producer	Collection of sources that produce data which will be ingested by the solution
Ingestion	Applications, Event Hub
Processing	Stream Analytics
Storage	Blob storage, Document DB
Presentation / Consumption	Analytics Dashboard

Table 7: High-level view of the solution's architecture layers.

The following sections present in detail each architecture tier and its components. We also discuss configuration options and provide specific implementation details for the components that were custom built.

3.1 Resource groups breakdown

A resource group in Azure is a container that holds related resources for a solution. Depending on the solution design, a resource group can include all the resources for the solution, or only those resources that are related in some way, for example they are elements of the same component. Further to the above, resource groups can be used to manage access to the underlying resources, as displayed in Figure 3-3. Role based access control is out of the scope of the solution presented in this paper and as such will not be further discussed or taken into account in the solution design.

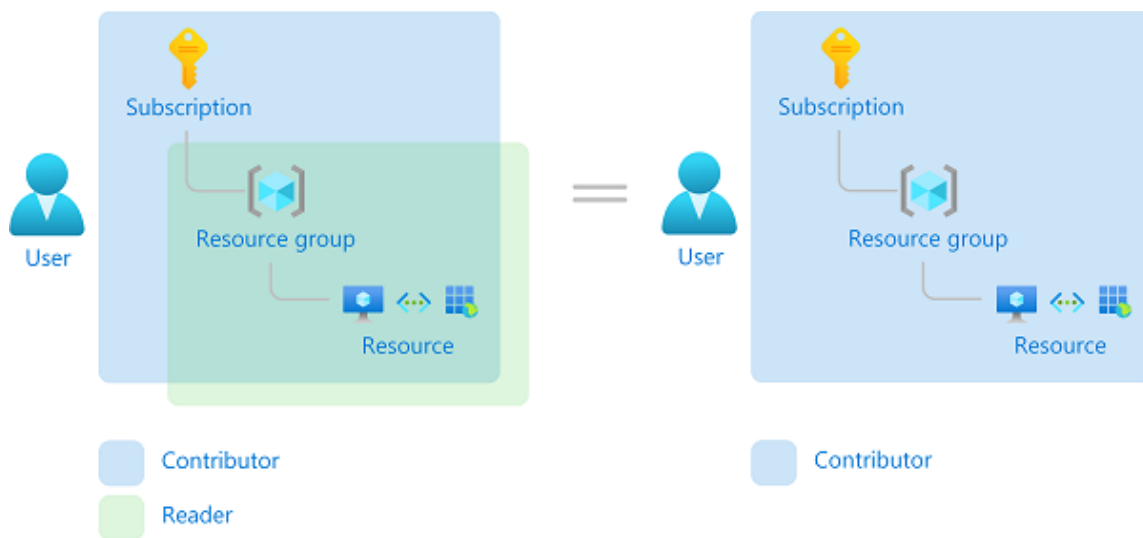


Figure 3-3: Example of role-based access control for Azure resource groups⁵.

The solution's components are separated into Resource Groups, which offer a way to logically group the different resources. As such, there is no "correct" way to organize the solution and it should be noted that resources can be transferred or redeployed to other resource groups if needed. Note that the breakdown of the solution's components into resource groups does not affect in any way its functionalities. However, the deployment guide in Annex III assumes that the resource groups defined in this section will be used.

⁵ Source: Azure RBAC documentation, available online at <https://docs.microsoft.com/en-us/azure/role-based-access-control/overview>.

The solution consists of the following three resource groups. The current break-down of the resources has been done primarily to hold components that are more cost-effective and store persistent information (blob storage, databases) into a dedicated resource group and to group resources that are deployed and re-deployed with different configuration depending on the business need (event hubs, stream analytics jobs) into a separate group.

- **rg-apps-alpha:** Contains the core application deployments that implement the streaming pipeline component of the solution.
- **rg-containers-alpha:** Contains the docker container deployments (Azure Container Instance resource type) that are used to host the Twitter stream listener and the Botometer checker applications.
- **rg-ops-alpha:** Contains deployments related to infrastructure health, security and performance monitoring.

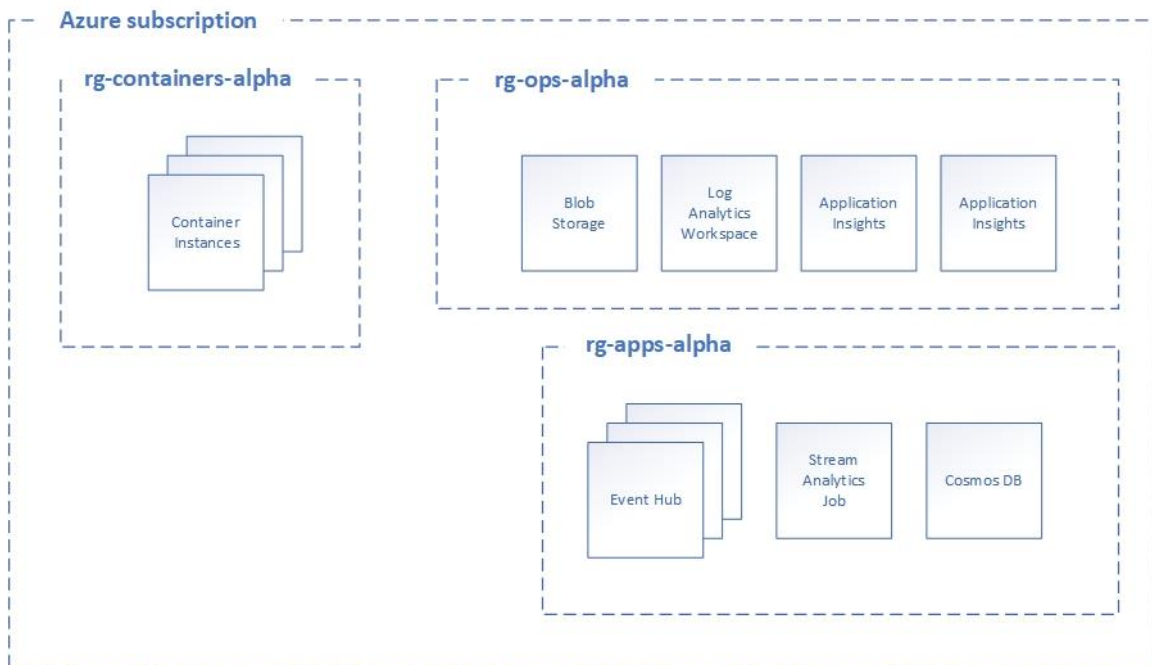


Figure 3-4: Overview of the solution's resource groups.

The resource groups have been annotated with `CATEGORY` and `ENVIRONMENT` tags⁶ to assist with management, maintenance, and cost monitoring activities. All resource groups have been provisioned in the West Europe location⁷.

Note that the component allocation over the resource groups outlined in this section attempts to follow a best practice approach. As described in the solution deployment guide in Annex III, all resources can be deployed in a single resource group with zero impact on the solution’s functionality or performance.

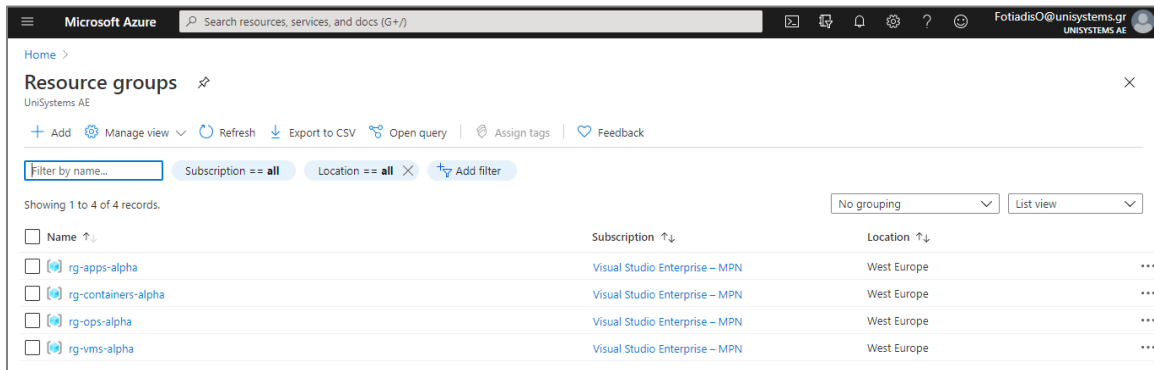


Figure 3-5: The resource groups for the solution in the Azure UI. Note that the resource group “rg-vms-alpha” visible in the screenshot contains a Virtual Machine created for the solution development and is not to be considered part of the solution itself.

3.2 Producer tier

The producer tier consists of external sources that act as data producers for the streaming pipeline solution presented in this dissertation. We will consider the following sources to serve as data producers for our solution:

1. Twitter streaming API [40].
2. Twitter’s GET trends/place API endpoint [41].
3. Botometer API exposed via the RapidAPI service [42].

⁶ See the following link for more information on Azure resource tagging: <https://docs.microsoft.com/en-us/azure/azure-resource-manager/management/tag-resources>.

⁷ The location here refers to the hosting location of the resource groups themselves and not of any contained resource groups, which have their own location provisioning property.

4. NYC Taxi & Limousine Commission - green taxi trip records dataset [43].

The following sections provide summary information on the data emitted from each of the above producers. Further to the above, to measure the run-time performance of the ingestion and processing components we will be imitating a high-volume producer emitting events using a subset of the dataset “NYC Taxi & Limousine Commission - green taxi trip records dataset”.

The above mentioned four sources were selected for the scenario presented in this work with the following criteria:

1. 24-hour access to a free, though rate-limited, streaming API which can provide data at a consistent frequency.
2. Availability of client libraries that can facilitate the development of the applications that will ingest the data in the Ingestion layer.
3. Opportunity to execute real-world scenarios with actual data that can be valuable to a business scenario.
4. Capacity to imitate a high-volume stream in a repeatable manner to be able to execute debugging and performance evaluation scenarios (this is the primary purpose for including the NYC taxi dataset).

From a conceptual point of view, the Producer tier can be extended to include any data producer - source that exposes data and that can be programmatically accessed by an application in the solution’s Ingestion tier. It is important to maintain documentation with each producer’s data model and how it is accessed so that the Ingestion tier’s applications can be appropriately updated in case the data model is updated. Figure 3-6 provides a conceptual overview of the solution’s Producer tier and its interaction with the Ingestion tier. The Ingestion tier displayed in the figure is composed of one container instance for each data producer and a single Event Hub namespace to receive the

ingested data as events. The rest of this section presents a brief overview of the APIs that will be used as producers in the different scenarios we will be working with.

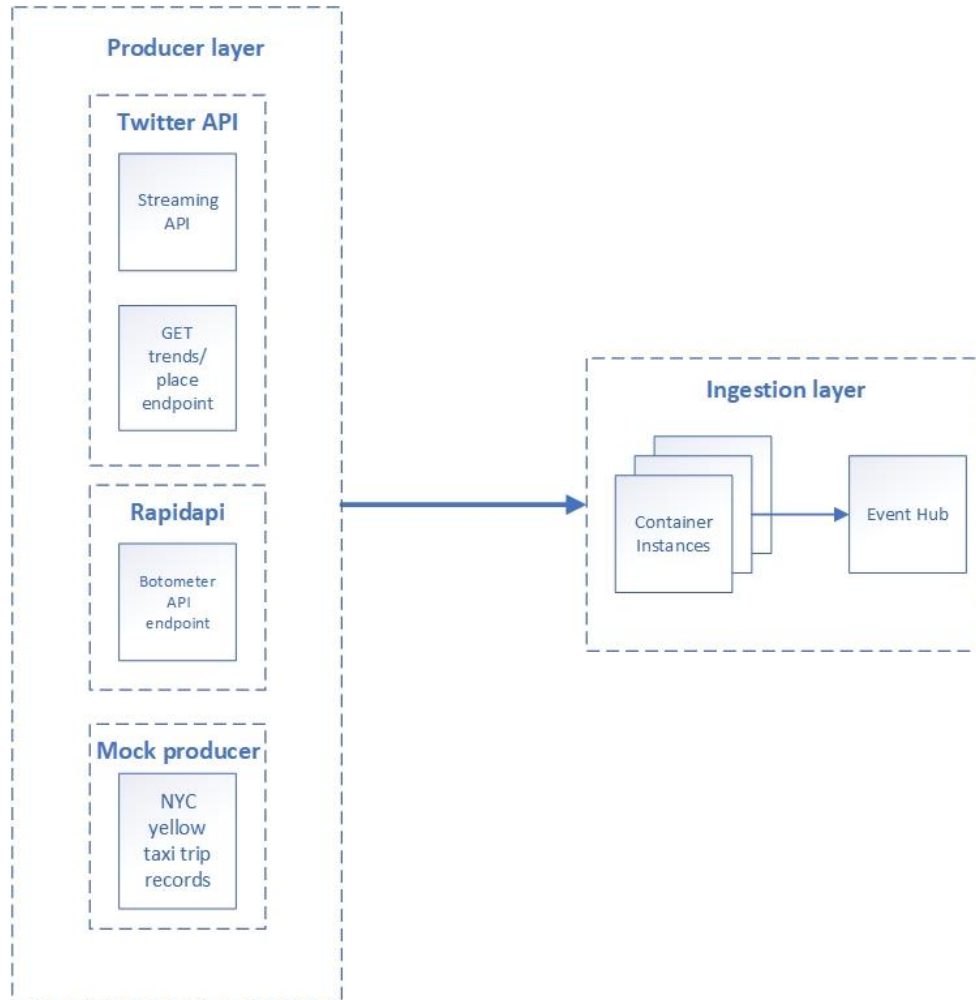


Figure 3-6: Overview of the architecture's producer layer and its interaction with the ingestion layer.

3.2.1 Twitter streaming API

Twitter is a microblogging and social networking service on which users post and interact with messages known as "tweets". As tweets are short and constantly generated, they are well suited for applications involving ingestion and processing of data streams. Twitter provides an API⁸ to stream tweets in real-time. Tweets are exposed in JSON format and thus are easy to consume and process

⁸ See also the official documentation on consuming streaming data: <https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data>.

with applications oriented towards data processing and analytics tasks. As described in section 3.3.1, a Python application has been developed which primarily utilizes the Tweepy Python library [44] to connect to the streaming API and retrieve tweets that contain a list of predefined hashtags. The API returns JSON objects representing tweets which will be ingested into the Event Hub as events for further processing. Due to ease of use and the platform's popularity, Twitter's streaming API has been widely utilized in academic research for numerous subjects such as sentiment analysis [45], real-time data analysis [46], real-time event recognition [47], health research [48] as well as a multitude of other fields.

3.2.2 Twitter GET Trends/place API

Twitter exposes trending topics for specific locations via the `GET TRENDS/PLACE` API endpoint [41]. The endpoint returns a JSON response containing the top 50 trending topics for a given place. A sample response for a location is provided in Sample ingestion data. In this dissertation we are going to utilize the `GET TRENDS/PLACE` API endpoint in a Python application to retrieve trends for a list of predefined locations and publish them as events into the Event Hub for persistent storage and processing, as described in section 3.3.2. The trends API has been utilized in research, for example to study the dynamics of emerging trends [49] and to perform real-time classification of tweets [50].

3.2.3 Botometer API

Botometer is an online service which evaluates the likelihood of a twitter account to belong to a bot or demonstrate a bot-like behavior [51]. This is done by utilizing Twitter's API to extract features of the account such as friends, social network structure, temporal activity patterns, language, and sentiment and comparing features of the tweet history of the account against a dataset of labeled examples. The outcome of this process is a "bot score" which indicates the likelihood that the account under examination belongs to a bot. A low score will indicate a likely human account while a high score will indicate a likely bot account.

Botometer is developed and published as a joint project by the Network Science Institute (IUNI), the Center for Complex Networks and Systems Research (CNetS) at the Luddy School of Informatics, Computing, and Engineering, and the Media School at Indiana University.

In the context of this dissertation, the Botometer service will be utilized to score twitter accounts that are ingested in the stream when monitoring twitter for hashtags of interest using the streaming API, so that tweets belonging to accounts with a high probability of being bots will be flagged. The combination of these can produce insights as to how much bot activity exists in popular hashtags, assessing whether influential users within a hashtag are bots and other similar scenarios.

3.2.4 NYC Taxi & Limousine Commission - green taxi trip records dataset

The New York City TLC dataset [52] contains records about yellow and green taxi trips, which have captured pick-up and drop-off dates/times, pick-up and drop-off locations, trip distances, itemized fares, rate types, payment types, and driver-reported passenger counts. The data are collected on behalf of the NYC Taxi and Limousine Commission (TLC) and are published in yearly datasets. In this dissertation the dataset is only used to benchmark and evaluate the performance of the various components of the solution. A sample of the dataset is presented in Sample ingestion data. The NYC taxi dataset has been thoroughly analyzed in numerous papers, such as in [53][54][55] to perform analytics on fare data, distances covered, as well as to test prediction algorithms regarding passenger and taxi driver behaviors.

3.3 Ingestion tier

The ingestion tier (also referred to as “collection tier” in reference architectures, i.e., in the reference architecture presented in section 2.5) serves as the foundation for the entire solution as it effectively is the point of entry of inputs. It consists of Python applications that execute in container instances, connect to the external data producers defined in section 3.1, perform minimal

preprocessing in the received inputs and publish them as events to an Event Hub instance. The ingestion layer of the architecture is presented in Figure 3-7.

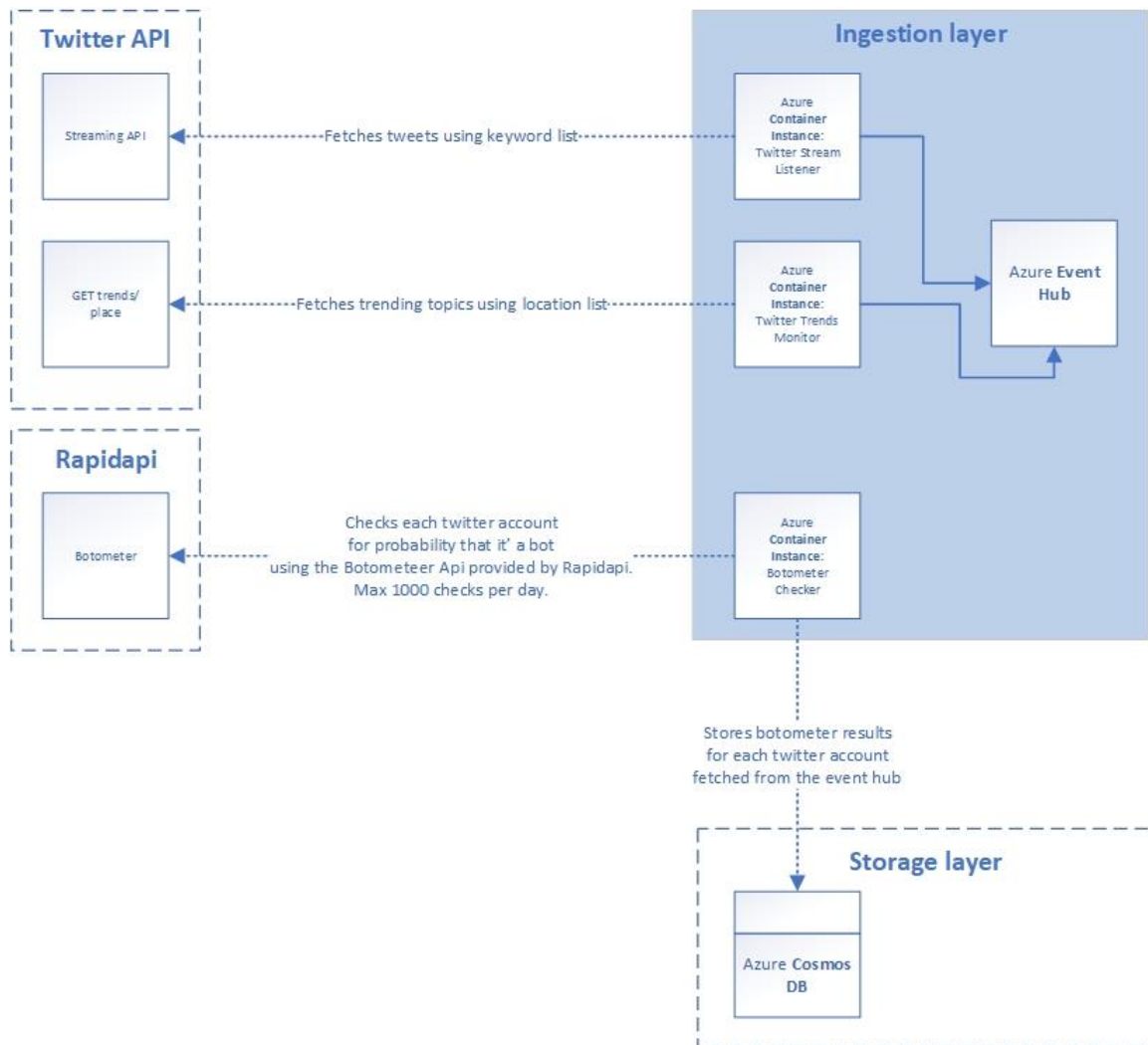


Figure 3-7: Overview of the architecture's ingestion layer

Each component is manually deployed as an Azure Container Instance using the deployment scripts provided in Development environment.

The applications in the solution's Ingestion layer that are presented in have been developed with the Python programming language, version 3.9.1. Visual Studio Code was used as IDE, with the extensions listed in Table 16. For individual libraries that were utilized (primarily Tweepy to access the Twitter streaming API, Botometer to access the Botometer API and various Azure API libraries

to interface with the different Azure components) and their versions, please refer to the respective requirements.txt file in the application repositories listed in Solution deployment guide.

All the Ingestion layer's applications were locally tested and debugged with Docker Desktop for Windows, version 3.2.0. The ARM templates and CLI deployment scripts have been developed in Visual Studio code utilizing the Azure CLI Tools and the Azure Resource Manager extensions. Once deployed, the container will execute until its allocated runtime finishes and is available for further repeated execution. The process of container deployment is shown in Figure 3-8 below.

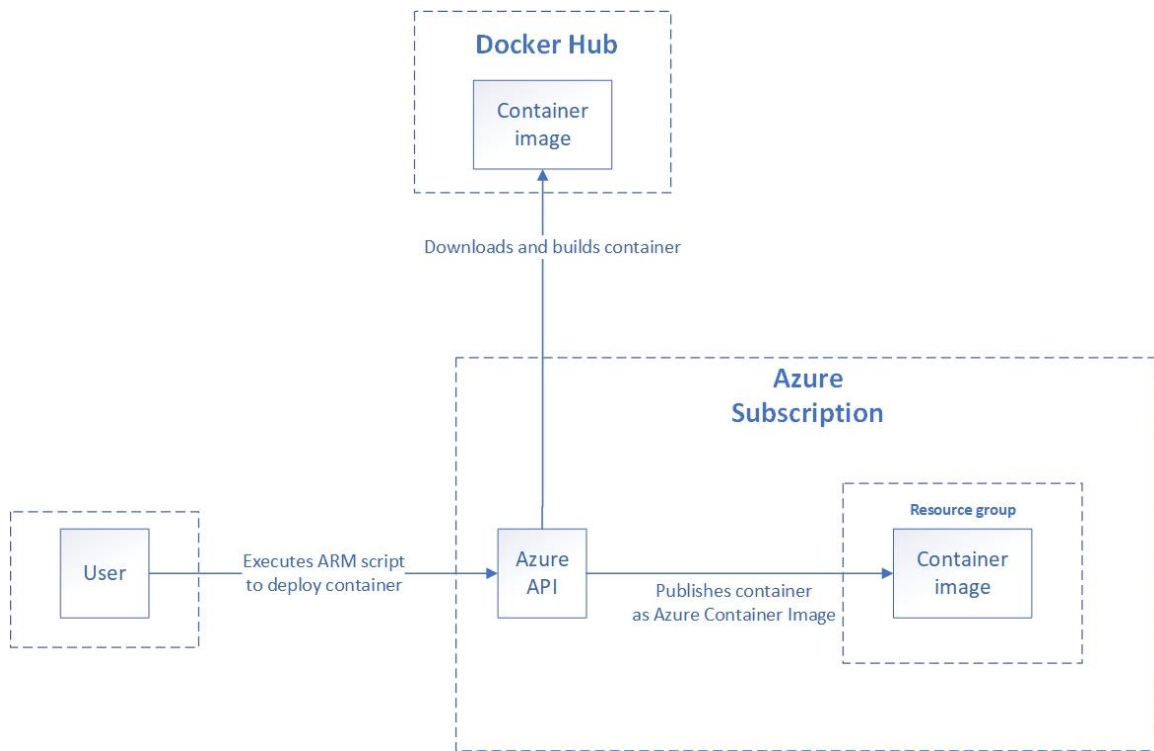


Figure 3-8: Overview of the Azure Container Instance deployment process

[1]. All components used as ingestion components in the solution are published in Docker Hub⁹ repositories (see also

⁹ Docker Hub is a service to share container images. See more information at <https://docs.docker.com/docker-hub/>.

Supplementary materials). Each application's functionality is thoroughly discussed in the next paragraphs of this section.

The screenshot shows the Azure portal interface for a container instance. At the top, it says 'ci-linux-alpha-trends | Containers'. Below that, there's a 'Refresh' button and a '1 container' indicator. A table lists the container details:

Name	Image	State	Previous state	Start time	Restart count
ci-twitter-trends-monitor	ofotiadis/twitter-trends-monitor:latest	Terminated	-	2021-02-07T21:58:24Z	0

Below the table, there are tabs for 'Events', 'Properties', 'Logs', and 'Connect'. The 'Events' tab is selected, and the 'Display time zone' is set to 'Local time'. A table of events is shown below:

Name	Type	First timestamp	Last timestamp	Message	Count
Started	Normal	2/7/2021, 11:58 PM GMT+2	2/7/2021, 11:58 PM GMT+2	Started container	1
Created	Normal	2/7/2021, 11:58 PM GMT+2	2/7/2021, 11:58 PM GMT+2	Created container	1
Pulled	Normal	2/7/2021, 11:58 PM GMT+2	2/7/2021, 11:58 PM GMT+2	Successfully pulled image 'ofotiadis/t...	1
Pulling	Normal	2/7/2021, 11:52 PM GMT+2	2/7/2021, 11:52 PM GMT+2	pulling image 'ofotiadis/twitter-trends...	1
Started	Normal	2/7/2021, 3:35 PM GMT+2	2/7/2021, 3:35 PM GMT+2	Started container	1
Created	Normal	2/7/2021, 3:34 PM GMT+2	2/7/2021, 3:34 PM GMT+2	Created container	1
Pulled	Normal	2/7/2021, 3:34 PM GMT+2	2/7/2021, 3:34 PM GMT+2	Successfully pulled image 'ofotiadis/t...	1
Pulling	Normal	2/7/2021, 3:33 PM GMT+2	2/7/2021, 3:33 PM GMT+2	pulling image 'ofotiadis/twitter-trends...	1

Figure 3-9: Overview of a container instance's status in the Azure UI.

3.3.1 Twitter Stream Listener

The Twitter Stream Listener application is a Python program that connects to the Twitter streaming API and retrieves tweets that contain one or more hashtags from a predefined list which is configured at the program's initialization. The collected tweets are converted to JSON objects and are sent as events to an Azure Event Hub instance (see more information in section 0 on the Event Hub configuration and purpose).

The application has been developed with the Python programming language, version 3.9.1. For individual libraries used (primarily Tweepy to access the Twitter streaming API and various Azure API libraries to interface with the different Azure components) and their versions, please refer to the respective requirements.txt file in the application repository (<https://github.com/orestisf/twitter-stream-listener/blob/main/requirements.txt>).

One of the application's objectives are to capture in a structured way as much information about an incoming tweet as possible so that it can be utilized in different processing scenarios while

minimizing – ideally eliminating the need to re-contact the Twitter API to get more information about a specific tweet.

The application will perform additional processing over each incoming tweet by appending additional properties by means of a boolean flag to indicate:

- If the tweet is a reply to another tweet.
- If the tweet contains a retweet.
- If the tweet contains a quote (a “quote” tweet is a wrapper around another tweet).
- If the tweet is associated with a place (for example a city or a point of interest).

This processing is done with the purpose of making easier queries and aggregations that will be done in the solution’s analytics layer.

For each of the above properties, additional related metadata are extracted from the tweet object, for example in case a tweet contains a retweet, the program will extract the retweeted tweet’s ID, user ID and username. This way, the application has the flexibility to accommodate multiple scenarios where tweets are used as data inputs to a streaming pipeline and furthermore to avoid re-contacting the Twitter API to request additional metadata about tweets after they are captured and stored in the solution’s database.

The architecture of the application provides the agility to instantiate multiple instances in different containers and track different sets of hashtags as well as route the captured tweets to separate event hub partitions.

Table 8 shows a sample tweet in JSON format processed by the Twitter Stream Listener application and ready to be sent to the Event Hub. A sample dataset of tweets collected with the application has been published in the link: <https://github.com/orestisf/twitter-stream-listener/blob/main/data-output-sample.json>.

```
[
  {
    "USER_ID_STR": "1300542919590522880",
    "USER_NAME": "LIL_RED",
    "USER_SCREENNAME": "LILRED30411902",
    "USER_LOCATION": "CHICAGO IL",
    "COORDINATES": NULL,
    "TWEET_LANG": "UND",
    "USER_CREATED": "31-AUG-2020",
    "CREATED": "08-FEB-2021",
    "FOLLOWERS_COUNT": 105,
    "FRIENDS": 111,
    "USER_LISTED_COUNT": 0,
    "USER_VERIFIED": FALSE,
    "STATUSES_COUNT": 16323,
    "FAVOURITES_COUNT": 16112,
    "TWEET_ID": "1358896365766643716",
    "TRUNCATED": FALSE,
    "HASHTAGS": [
      {
        "TEXT": "SAVEAMERICA",
        "INDICES": [
          0,
          12
        ]
      }
    ]
  }
]
```

```
    }  
  ],  
  "HASHTAGS_COUNT": 1,  
  "USER_MENTIONS": NULL,  
  "USER_MENTIONS_COUNT": 0,  
  "URLS": NULL,  
  "URLS_COUNT": 0,  
  "SYMBOLS": NULL,  
  "SYMBOLS_COUNT": 0,  
  "MEDIA_COUNT": 0,  
  "FULL_TEXT": "#SAVEAMERICA",  
  "CLEANTEXT": "",  
  "IS_REPLY": TRUE,  
  "TWEET_REPLY_ID_STR": NULL,  
  "TWEET_REPLY_USER_ID_STR": NULL,  
  "TWEET_REPLY_USER_SCREEN_NAME": NULL,  
  "HAS_RETWEET": FALSE,  
  "HAS_QUOTE": TRUE,  
  "QUOTED_TWEET_ID": "1358821818056863744",  
  "QUOTED_TWEET_USER_ID": "2314018987",  
  "QUOTED_TWEET_USER_SCREEN_NAME": "SHELBYKSTEWART",  
  "HAS_PLACE": FALSE  
}  
]  
]
```

Table 8: Sample tweet processed and sent as event by the Twitter Stream Listener application.

3.3.1.1 Tweet object properties

For each incoming tweet, the application will capture and send as an event to the Event Hub the properties presented in Table 9. For each tweet processed, the Azure Stream Analytics Service appends additional properties related to when the incoming event was enqueued for processing, and the Event Hub partition ID where the tweet was retrieved from. These properties are presented in Table 10. Finally, when the tweet is eventually stored as a JSON document in a Cosmos DB container where additional attributes are added (unique id, timestamp). These attributes are displayed in Table 11.

Attribute	Type	Example value
user_id_str	String	909848411352043520
user_name	String	Tom Germanotta Lipa
user_screenname	String	tom_germanotta
user_location	String	Denver, CO
coordinates	JSON	[[-74.026675, 40.683935], [-74.026675, 40.877483], [-73.910408, 40.877483

		<pre>], [-73.910408, 40.3935]] </pre>
tweet_lang	String	el
user_created	Date	18-Sep-2017
created	Date	27-Dec-2020
followers_count	Integer	231
friends	Integer	0
user_listed_count	Integer	0
user_verified	Integer	0
statuses_count	Integer	13113
favourites_count	Integer	11381
tweet_id	Integer – The unique ID of the tweet.	1343260640547364865
truncated	Integer (accepted values: 0,1)	0
hashtags		<pre> [{ "indices": [32, 38 </pre>

		<pre>], "text": "somehashtag" }] </pre>
user_mentions	String	someUserName
urls		<pre> [{ "indices": [32, 52], "url": "http://t.co/IOwBrTZR", "display_url": "youtube.com/watch?v=oHg5SJ...", "expanded_url": "http://www.youtube.com/watch?v=oHg5SJYRH A0" }] </pre>
symbols		<pre> [{ "indices": [12, 17], </pre>

		<pre> "text": "twtr" }] </pre>
media		<pre> [{ "display_url": "pic.twitter.com/5J1WJSRCy9", "expanded_url": "https://twitter.com/nolan_test/status/9300778475 35812610/photo/1", "id": 9.300778475358126e17, "id_str": "930077847535812610", "indices": [13, 36], "media_url": "http://pbs.twimg.com/media/DOhM30VVwAEpI Hq.jpg", "media_url_https": "https://pbs.twimg.com/media/DOhM30VVwAEp IHq.jpg" "sizes": { "thumb": { "h": 150, </pre>

		<pre>"resize": "crop", "w": 150 }, "large": { "h": 1366, "resize": "fit", "w": 2048 }, "medium": { "h": 800, "resize": "fit", "w": 1200 }, "small": { "h": 454, "resize": "fit", "w": 680 } }, "type": "photo", "url": "https://t.co/5J1WJSRCy9", }]</pre>
full_text		This is a tweet body.

is_reply	Integer (accepted values: 0,1)	1
tweet_reply_id_str		1343260640547364865
tweet_reply_user_id_str		someUserName
tweet_reply_user_screen_name	String	someUserName
has_retweet	Integer (accepted values: 0,1)	1
retweeted_tweet_id	Integer	1343260640547364865
retweeted_tweet_user_id	Integer	909848411352043520
retweeted_tweet_user_screen_name	String	SomeUserName
has_quote	Integer (accepted values: 0,1)	1
quoted_tweet_id	Integer	1343260640547364865
quoted_tweet_user_id	Integer	909848411352043520
quoted_tweet_user_screen_name		SomeUserName
has_place	Integer (accepted values: 0,1)	1
place_id		01a9a39529b27f36

place_url	String	https://api.twitter.com/1.1/geo/id/01a9a39529b27f36.json
place_type	String	city
place_name	String	Manhattan
place_full_name	String	Manhattan, NY
place_country_code	String	US
place_country	String	United States
place_bounding_box.type	String	Polygon
place_bounding_box.coords	JSON	[[[-74.026675, 40.683935], [[-74.026675, 40.877483], [[-73.910408, 40.877483], [[-73.910408,

		40.683935]]]
--	--	--------------------------

Table 9: Sample tweet properties captured by the Twitter Stream Listener application

ATTRIBUTE	TYPE	EXAMPLE VALUE
EVENTENQUEUEDUTCTIME	DateTime	2020-12- 27T18:21:13.9020000Z
EVENTPROCESSEDUTCTIME	DateTime	2020-12- 27T18:21:15.0320935Z
PARTITIONID	Integer	0

Table 10: Additional properties to the tweet JSON document appended by the Azure Stream Analytics Service.

ATTRIBUTE	TYPE	EXAMPLE VALUE
_RID	Resource ID ¹⁰ . A unique identifier for the document.	YLUF AJAion8GAAAAAAAAAAAA ==
_SELF	The unique addressable URI for the document.	db s/YLUF AA==/coll s/YLU FAJAion8=/docs/YLUF AJA ion8GAAAAAAAAAAAA==/
_ETAG	The document's etag used by Cosmos DB for concurrency control.	\ "00001000-0000-0d00- 0000-5fe8d0a00000\"

¹⁰ See the following link for system generated properties in Cosmos DB documents: <https://docs.microsoft.com/en-us/rest/api/cosmos-db/documents>.

_TS

Epoch¹¹ value in seconds (not milliseconds) since an item was last modified.

Table 11: System properties appended to the tweet JSON documents in the Azure Cosmos DB.

Finally, each tweet is stored as JSON document in the `tweets` Cosmos DB container. Figure 3-10 displays a view of a document stored in the Cosmos DB container.

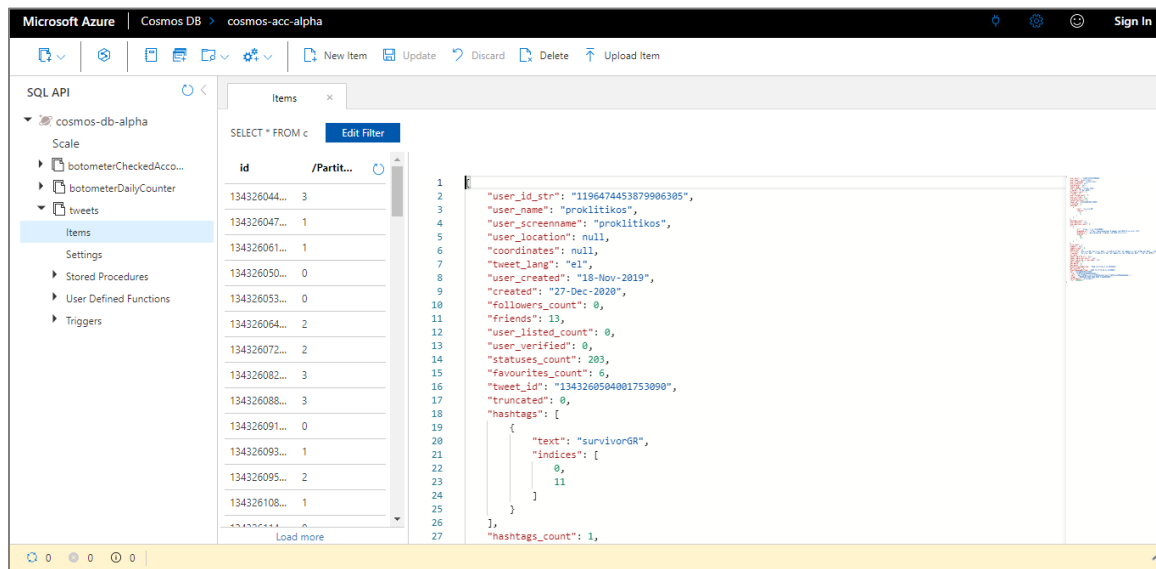


Figure 3-10: View of the tweets Cosmos DB container in the Azure UI. A single tweet is displayed as JSON document.

3.3.1.2 Configurable properties

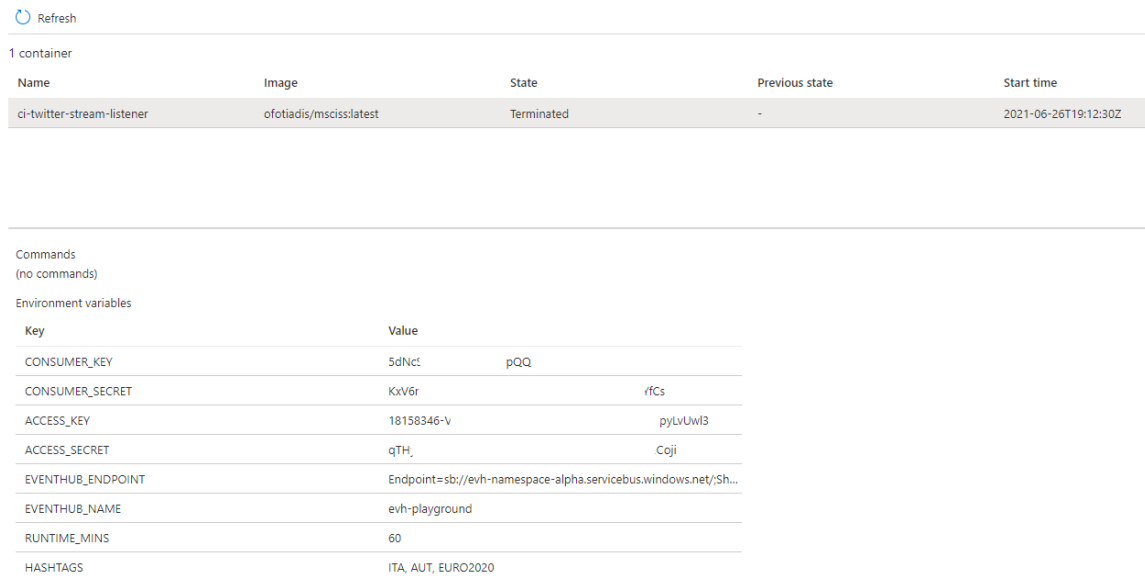
The Twitter Stream Listener application has several configuration items exposed as environmental properties to allow the user to run the container in any Azure subscription, to set the desired hashtags to be tracked and the total running time of the application without having to modify and re-publish the container itself. The configurable environmental properties of the Twitter Trends Monitor application are displayed in Table 12.

¹¹ See the following link for more information on the `_ts` property: <https://devblogs.microsoft.com/cosmosdb/new-date-and-time-system-functions/#converting-the-system-ts-property-to-a-datetime-string/>.

Environmental property	Explanation
CONSUMER_KEY	String - The consumer API key of the Twitter App.
CONSUMER_SECRET	String - The consumer API secret of the Twitter App.
ACCESS_KEY	String - The access token of the Twitter App.
ACCESS_SECRET	String - The access secret of the Twitter App.
EVENTHUB_ENDPOINT	String – The Connection string – primary key property ¹² of the Event Hub instance that will be used to push the processed tweets to.
EVENTHUB_NAME	String – The name of the Event Hub that the tweets will be pushed to as events.
RUNTIME_MINS	Integer – Define how many minutes the application will connect to the Twitter Streaming API before closing.
HASHTAGS	String - A comma separated list of hashtags that the application will monitor.

Table 12: List of environmental properties for the Twitter Stream Listener application.

¹² The property can be retrieved by this Azure CLI script: `az eventhubs eventhub authorization-rule keys list --resource-group rg-apps-alpha --namespace-name evh-namespace-alpha --eventhub-name evh-alpha --name common-sas-alpha --subscription xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxxxxxxxx`. See also the relevant documentation at: https://docs.microsoft.com/en-us/cli/azure/eventhubs/eventhub/authorization-rule/keys?view=azure-cli-latest#az_eventhubs_eventhub_authorization_rule_keys_list-examples.



Refresh

1 container

Name	Image	State	Previous state	Start time
ci-twitter-stream-listener	ofotiadis/mscisslatest	Terminated	-	2021-06-26T19:12:30Z

Commands
(no commands)

Environment variables

Key	Value
CONSUMER_KEY	5dNc1 pQQ
CONSUMER_SECRET	KxV6r rfCs
ACCESS_KEY	18158346-v pyLvUwl3
ACCESS_SECRET	qTHj Coji
EVENTHUB_ENDPOINT	Endpoint=sb://evh-namespace-alpha.servicebus.windows.net/Sh...
EVENTHUB_NAME	evh-playground
RUNTIME_MINS	60
HASHTAGS	ITA, AUT, EURO2020

Figure 3-11: View of the application deployed in Azure as a container. The environmental properties set for the particular instance are displayed.

3.3.2 Twitter Trends Monitor

The Twitter Trends Monitor is a Python program that polls at predefined intervals and for a list of predefined locations such as cities or countries, Twitter’s `GET TRENDS/PLACE` API endpoint for the Trending topics that trending at the specific time for that place. The response contains the top 50 trending topics and includes metadata on the name and WOEID¹³ of the location and timestamps on when the file was generated and requested.

The application has been developed with the Python programming language, version 3.9.1. For individual libraries used, refer to the respective requirements.txt file in the application repository (<https://github.com/orestisf/twitter-trends-monitor/blob/main/requirements.txt>).

Table 13 shows a sample response from the `GET TRENDS/PLACE` API for the location “Athens”. Due to its length, the response is presented truncated, with the full response sample being available in Sample ingestion data. A sample dataset of trends collected for several locations with this

¹³ WOEID (**W**here **O**n **E**arth **I**Dentifier) is a unique 32-bit reference identifier for locations which is used by the Twitter API to refer to locations for trending topics.

application is also published at the location: <https://github.com/orestisf/twitter-trends-monitor/blob/main/data-output-sample.json>.

```
[
  {
    "TRENDS": [
      {
        "NAME": "#ELAXAMOGELA",
        "URL": "HTTP://TWITTER.COM/SEARCH?Q=%23ELAXAMOGELA",
        "PROMOTED_CONTENT": "NONE",
        "QUERY": "%23ELAXAMOGELA",
        "TWEET_VOLUME": "NONE"
      },
      {
        "NAME": "#IKAPIA",
        "URL":
"HTTP://TWITTER.COM/SEARCH?Q=%23%CE%B9%CE%BA%CE%B1%CF%81%CE%B9%CE%B1",
        "PROMOTED_CONTENT": "NONE",
        "QUERY": "%23%CE%B9%CE%BA%CE%B1%CF%81%CE%B9%CE%B1",
        "TWEET_VOLUME": "NONE"
      },
      {
        "NAME": "ΜΑΝΤΣΕΣΤΕΡ ΓΙΟΥΝΑΙΤΕΝ",
        "URL":
"HTTP://TWITTER.COM/SEARCH?Q=%22%CE%9C%CE%B1%CE%BD%CF%84%CF%83%CE%B5%CF%8"
```

```

3%CF%84%CE%B5%CF%81+%CE%93%CE%B9%CE%BF%CF%85%CE%BD%CE%B1%CE%B9%CF%84%
CE%B5%CE%BD%CF%84%22",
    "PROMOTED_CONTENT": "NONE",
    "QUERY":
"%22%CE%9C%CE%B1%CE%BD%CF%84%CF%83%CE%B5%CF%83%CF%84%CE%B5%CF%81+%CE%
93%CE%B9%CE%BF%CF%85%CE%BD%CE%B1%CE%B9%CF%84%CE%B5%CE%BD%CF%84%22",
    "TWEET_VOLUME": "NONE"
  }
],
"AS_OF": "2021-02-06T10:49:38Z",
"CREATED_AT": "2021-01-07T16:56:34Z",
"LOCATIONS": [
  {
    "NAME": "GREECE",
    "WOEID": 23424833
  }
]
}
]

```

Table 13: Sample response JSON of the GET trends/place API endpoint for the location "Athens".

Even though the primary purpose of the application is to be used as a data ingestion component in the solution presented in this dissertation, it can also be used as a stand-alone program to collect Twitter trends for a list of locations at a specified time interval and persist them in a JSON file. Such a stand-alone version of the application is published as a Jupyter notebook for reference at the following location:

https://github.com/orestis/twitter-trends-monitor/blob/main/twitterTrendMonitor_v1_0.ipynb.

The application is also published as a docker container and is publicly available at the url:

<https://hub.docker.com/repository/docker/ofotiadis/twitter-trends-monitor>. This way it can be

executed in an Azure Container Instance as described in Solution deployment guide.

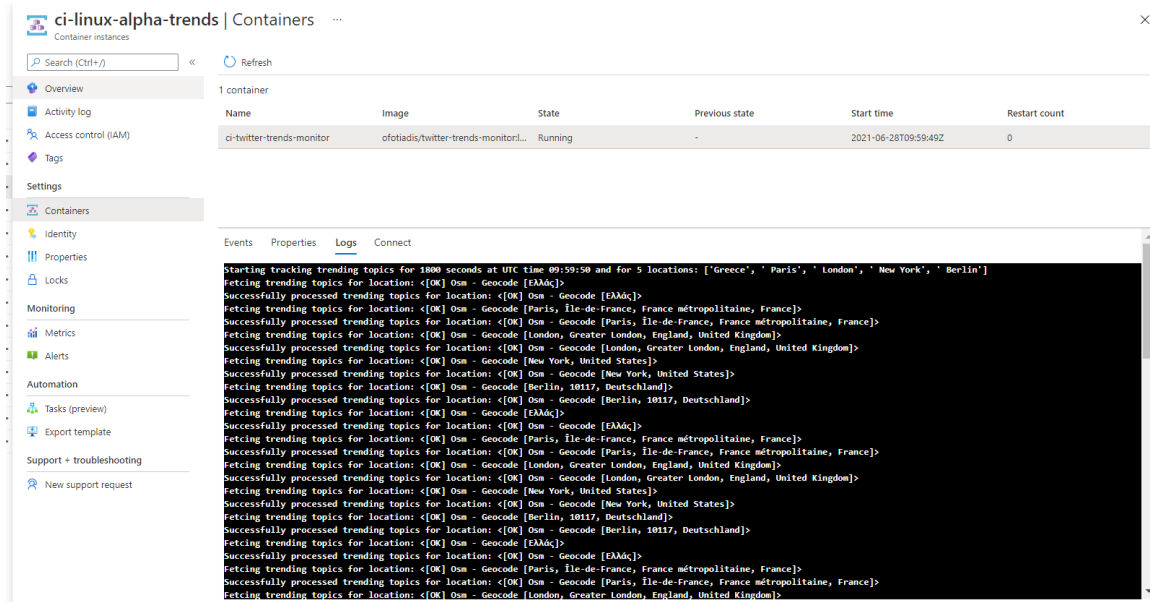


Figure 3-12: The Twitter Trends Monitor application executing as an Azure Container Instance

Scenarios where the application can be utilized include:

- Creating dashboards showing trending topics for a list of locations.
- Uncovering trends in a regional or global level.
- Monitoring the evolution of trending topics for a region over a time period.

Input preview [Test results](#)

Showing events from 'trendsinput'. This list of events might not be complete. Select a specific time range to show all events during that period.

View in JSON Table Raw Refresh Select time range Upload sample input Download sample data

trends	as_of	created_at	locations	EventProcessedUtcTime	Partitionid	EventEnqueuedUtcTime
[{"name":"#BELPOR","url":"..."}]	"2021-06-28T08:04:52.0000..."	"2021-05-31T07:39:50.0000..."	[{"name":"Berlin","woeid":638242}]	"2021-06-28T08:28:46.5641..."	0	"2021-06-28T08:04:53.0210..."
[{"name":"#28giugno","url":"..."}]	"2021-06-28T08:04:52.0000..."	"2021-06-10T06:22:48.0000..."	[{"name":"Rome","woeid":721943}]	"2021-06-28T08:28:46.5641..."	0	"2021-06-28T08:04:53.0210..."
[{"name":"#MondayMotivat..."}]	"2021-06-28T08:04:51.0000..."	"2021-06-01T20:13:53.0000..."	[{"name":"Paris","woeid":615702}]	"2021-06-28T08:28:46.5641..."	0	"2021-06-28T08:04:53.0210..."
[{"name":"#ΠολιωνΑγιοΙω..."}]	"2021-06-28T08:04:51.0000..."	"2021-06-02T21:10:56.0000..."	[{"name":"Greece","woeid":23424833}]	"2021-06-28T08:28:46.5641..."	0	"2021-06-28T08:04:53.0210..."

Figure 3-13: Captured twitter trend events as they appear in Azure's stream analytics job.

3.3.2.1 Configurable properties

The application has several configuration items exposed as environmental variables to allow the user to run the container in any Azure subscription and to set the location and polling frequency without having to modify and re-push the container itself. The configurable environmental properties of the Twitter Trends Monitor application are displayed in Table 14.

Environmental property	Explanation
CONSUMER_KEY	String - The consumer API key of the Twitter App.
CONSUMER_SECRET	String - The consumer API secret of the Twitter App.
ACCESS_KEY	String - The access token of the Twitter App.
ACCESS_SECRET	String - The access secret of the Twitter App.
APP_INSIGHTS_KEY	String – The Instrumentation key of the Application Insights instance which is used by the application to log exceptions.
EVENTHUB_ENDPOINT	String – The Connection string – primary key property of the Event Hub instance that will be used to push the processed tweets to.
EVENTHUB_NAME	String – The name of the Event Hub that the tweets will be pushed to as events.
RUNTIME_MINS	Integer – Define how many minutes the application will execute.
QUERY_INTERVAL_SEC	Integer – Define every how many seconds the application will connect to the Twitter <code>GET</code>

	TRENDS/PLACE API endpoint ¹⁴ and retrieve trends for each location.
LOCATIONS	String - A comma separated list of the locations that the application will submit to the Twitter API to retrieve the trends.

Table 14: List of environmental properties for the Twitter Trends Monitor application.

3.3.3 NYC Green Taxi dataset publisher

As discussed in section 3.1 the motive for this application is to use it to imitate event generation in a predictable manner, for benchmarking and testing purposes. The “NYC Green Taxi dataset publisher” is a Python application that:

- Downloads the dataset in Parquet format as it exists in the `azureml.opendatasets` Python library;
- Samples a number of records that can be configured by the user and exports these records as a Pandas dataframe;
- Filters the dataframe to keep only a subset of the available attributes (`'vendorID'`, `'passengerCount'`, `'lpepPickupDatetime'`, `'lpepDropoffDatetime'`, `'tripDistance'`, `'tipAmount'`, `'totalAmount'`);
- Converts each record to JSON;
- Sends each record as an event to an Azure Event Hub.

To examine different scenarios, the application can be configured to send events in two ways:

1. Serially, one-by-one.
2. In a batch, with a size that is configurable by the user (i.e., 500 events per batch);

¹⁴ Note that the free API allows for a maximum of 75 requests every 15 minutes. See also API reference at <https://developer.twitter.com/en/docs/twitter-api/v1/rate-limits>.

This has been done to be able to test the Event Hub’s performance in scenarios with varying volumes.

vendorID	passengerCount	lpepPickupDatetime	lpepDropoffDatetime	tripDistance	tipAmount	totalAmount	EventProcessedUtc...	Partitionid
2	2	1453080735000	1453081013000	1.37	1.46	8.76	*2021-06-30T05:44:17....	2
2	1	1452673985000	1452674375000	1.21	0	7.3	*2021-06-30T05:44:17....	1
1	1	1454086729000	1454090521000	2.3	0	20.3	*2021-06-30T05:44:17....	2
2	1	1452915195000	1452915447000	0.75	0	6.3	*2021-06-30T05:44:17....	1
2	1	1451657757000	1451658089000	1.38	0	7.3	*2021-06-30T05:44:17....	2
1	1	1453897478000	1453897768000	0.8	0	6.3	*2021-06-30T05:44:17....	1
2	1	1453806861000	1453809334000	5.8	0	28.3	*2021-06-30T05:44:17....	2
2	1	1454118298000	1454119005000	2.31	2.36	14.16	*2021-06-30T05:44:17....	1
2	1	1452674848000	1452676479000	4.48	0	21.8	*2021-06-30T05:44:17....	2
2	1	1453319932000	1453319953000	0.07	0	35	*2021-06-30T05:44:17....	1
2	1	1452942439000	1452942830000	1.01	0	6.8	*2021-06-30T05:44:17....	2

Figure 3-14: A sample of NYC Green Taxi records sent as events to the Event Hub.

3.3.3.1 Configurable properties

The NYC Green Taxi dataset publisher application has several configuration items exposed as environmental properties to allow the user to run the container in any Azure subscription. The configurable environmental properties of the application are displayed in Table 12.

Environmental property	Explanation
EVENTHUB_ENDPOINT	String – The Connection string – primary key property ¹⁵ of the Event Hub instance that will be used to push the processed tweets to.
EVENTHUB_NAME	String – The name of the Event Hub that the tweets will be pushed to as events.

¹⁵ The property can be retrieved by this Azure CLI script: `az eventhubs eventhub authorization-rule keys list --resource-group rg-apps-alpha --namespace-name evh-namespace-alpha --eventhub-name evh-alpha --name common-sas-alpha --subscription xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxxxxxxxx`. See also the relevant documentation at: https://docs.microsoft.com/en-us/cli/azure/eventhubs/eventhub/authorization-rule/keys?view=azure-cli-latest#az_eventhubs_eventhub_authorization_rule_keys_list-examples.

sample_size	Integer – The number of taxi trip records that will be sent to the Event Hub as events.
execution_type	String – Accepted values: “Single”, ‘Batch’. Whether the records will be published one-by-one or as a batch.
batch_size	Integer – The number of records that will be included in a batch.

Table 15: List of environmental properties for the NYC Green Taxi dataset publisher application.

3.3.4 Botometer Checker

The Botometer Checker application is a Python program that utilizes the Botometer service (see section 3.2.3 for details on the service) to assess each tweet captured with the Twitter Stream Listener application in whether it was posted by a bot account. It will then store each user’s score in a persistent store to make it available when analyzing tweet datasets.

The application has been designed to function as follows:

1. Connects to the solution’s Cosmos DB and reads tweets stored as documents.
2. For each tweet, it checks the Cosmos DB container to determine whether the account that posted this tweet has been checked against Botometer in the past.
3. For each unchecked account, calls the Botometer API and stores the response (see a sample at Botometer API sample response for Twitter account) with the account’s assessment in the “botometerCheckedAccounts” container of the solution’s Cosmos DB database.

The application can thus function in two ways:

1. Synchronously, executing in parallel with the when the Twitter Stream Listener application.
2. Asynchronously, executed at a different time frame using tweets already stored in the Event Hub.

The following table presents a sample response returned from Botometer along with each attribute's description.

Attribute	Example value	Description
user_id	115410344236278579 7	The unique ID of the user in Twitter.
user_screen_name	ierodidaskalos	The screen name of the user's Twitter account.
user_majority_language	el	The language used for the majority of the user's tweets.
cap		The user's Complete Automation Probability (CAP) score, a cumulative probability: it represents the probability that accounts with a score equal to or greater than this are automated.
english	0.874857504	CAP for accounts tweeting primarily in the english language.
universal	0.660850031	CAP for accounts tweeting primarily in languages other than english.
raw_scores_english / universal		The bot score in the [0,1] range is given, both using English (all features) and Universal (language-independent) features; in each case the overall score is provided as well as the sub-scores for each bot class as shown in the rows below.
display_scores_english / universal		Same as raw scores, but in the [0,5] range.

astroturf	2	Represents manually labeled political bots and accounts involved in follow trains that systematically delete content.
fake_follower	1.2	Represents bots purchased to increase follower counts.
financial	0.1	Represents bots that post using cashtags.
other	4.6	Represents miscellaneous other bots obtained from manual annotation, user feedback, etc.
self_declared	0	Represents bots from botwiki.org.
spammer	0	Represents accounts labeled as spambots from several datasets.

Table 16: Attributes returned by the Botometer API

3.3.4.1 Configurable properties

The Botometer checker application has several configuration elements exposed as environmental properties to allow the user to run the container in any Azure subscription. In addition, the purpose of the configurable elements is to set the user's personal Twitter and RapidAPI API keys. The configurable environmental properties of the application are displayed in Table 12.

Environmental property	Explanation
CONSUMER_KEY	String - The consumer API key of the Twitter App.
CONSUMER_SECRET	String - The consumer API secret of the Twitter App.
ACCESS_KEY	String - The access token of the Twitter App.
ACCESS_SECRET	String - The access secret of the Twitter App.

EVENTHUB_CONNECTION_STRING	String – The Connection string – primary key property ¹⁶ of the Event Hub instance that will be used to push the processed tweets to.
EVENTHUB_NAME	String – The name of the Event Hub that the tweets will be pushed to as events.
EVENTHUB_CONSUMER_GROUP	String – The consumer group name that will be used to read the event hub topic which contains the tweets.
COSMOSDB_URL	String – The endpoint for the Cosmos DB account.
COSMOSDB_KEY	String – The connection string for the Cosmos DB account.
COSMOSDB_DB_NAME	String – The Cosmos DB database name
COSMOSDB_BOTOMETER_INDEX_CONTAINER_NAME	String – The container name that will store the Botometer results.
COSMOSDB_BOTOMETER_COUNTER_CONTAINER_NAME	String – The container name that will store the daily usage of the Botometer API.
STORAGE_CONNECTION_STRING	String – The connection string for the Blob store that is used to save the event hub consumer’s checkpoints.
BLOB_CONTAINER_NAME	String – The blob container name for the Blob store that is used to save the event hub consumer’s checkpoints.
RAPIDAPI_KEY	String – The personal RapidAPI key that will be used to access the Botometer API.

¹⁶ The property can be retrieved by this Azure CLI script: `az eventhubs eventhub authorization-rule keys list --resource-group rg-apps-alpha --namespace-name evh-namespace-alpha --eventhub-name evh-alpha --name common-sas-alpha --subscription xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxxxxxxxx`. See also the relevant documentation at: https://docs.microsoft.com/en-us/cli/azure/eventhubs/eventhub/authorization-rule/keys?view=azure-cli-latest#az_eventhubs_eventhub_authorization_rule_keys_list-examples.

BOTOMETER_SESSION_LIMIT	Integer – How many Twitter accounts to send to Botometer API per session.
--------------------------------	---

Table 17: List of configurable environmental properties for the Botometer Checker Python application.

3.3.5 Event Hub

Azure Event Hubs is a scalable publish-subscribe data integrator which is capable of consuming large volumes of events per second from input sources. It abstracts the complexity of ingesting multiple different input streams directly into the solution's analytics layer. Its functionality is thus similar to Apache Kafka, which is also designed to handle large scale stream ingestion driven by real-time events. At a high-level, both Kafka and Event Hub are a distributed, partitioned and replicated commit log service and both use a partitioned consumer model thus enabling concurrent consumers to independently subscribe and read the events. Each system uses a different terminology for similar concepts. To familiarize the reader with the terminology used for the Event Hub architecture and configuration, Table 18 presents a comparison of equivalent terms between Apache Kafka and Azure Event Hub¹⁷.

Apache Kafka concept	Azure Event Hub concept
Cluster	Namespace
Topic	Event Hub
Partition	Partition
Consumer Group	Consumer Group
Offset	Offset

Table 18: Corresponding terms between Apache Kafka and Azure Event Hub concepts

Azure Event hub uses the Advanced Message Queuing Protocol (AMQP) to handle its messaging workflow. AMQP is an open standard application layer protocol for message-oriented middleware. AMQP enables a flow controlled, message-oriented communication with message-delivery guarantees such as (O'Hara, 2007):

¹⁷ Table originally provided in the Azure documentation, see: <https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-for-kafka-ecosystem-overview#kafka-and-event-hub-conceptual-mapping>.

1. at-most-once (each message is delivered once or never),
2. at-least-once (each message is certain to be delivered, but may do so multiple times)
3. exactly-once (each message will always certainly arrive and do so only once)

An Event Hub is used in the industry to accommodate scenarios where a high volume of events must be ingested into a streaming / analytics pipeline for further processing. Typical usage scenarios include:

- Anomaly detection
- Application logging
- Analytics pipelines
- Live dashboards
- Data archiving
- Transaction processing

A reference high-level architecture of an Azure event hub in is shown in Figure 3-15.

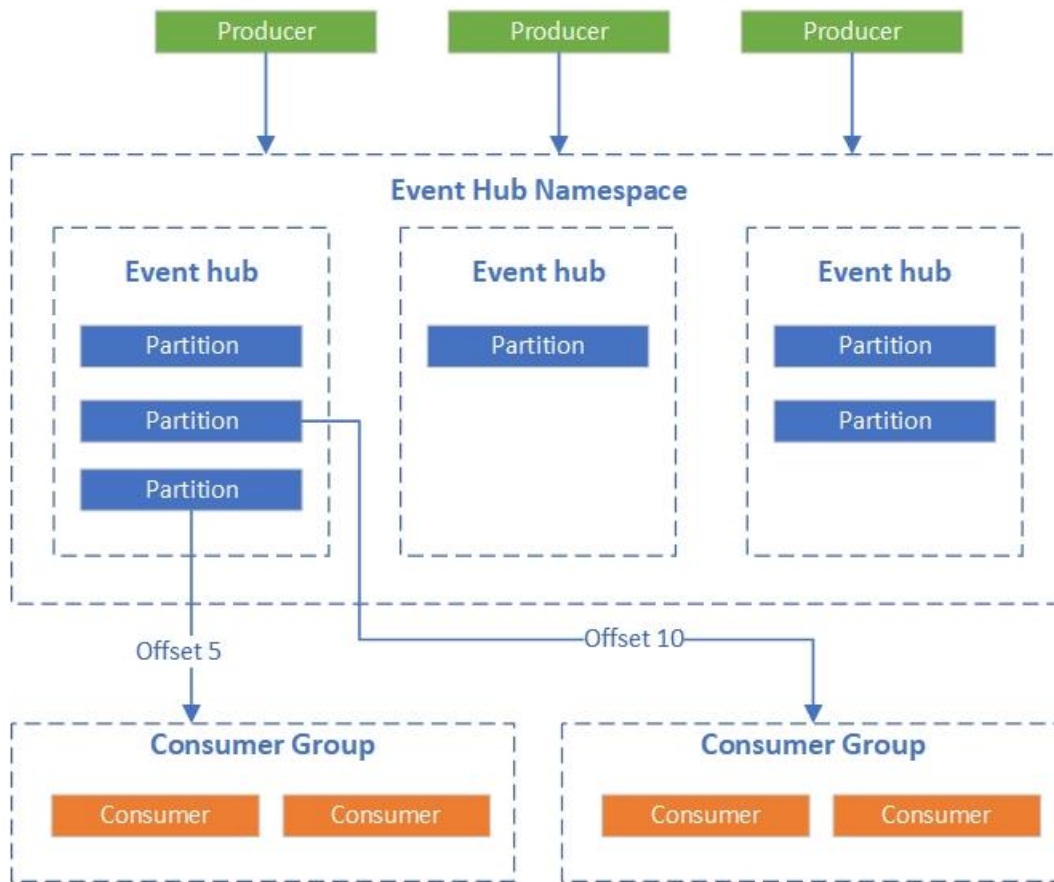


Figure 3-15: Architecture of an Azure Event Hub

Using the above diagram’s pattern, the Event Hub designed and deployed for the solution presented in this dissertation is displayed in Figure 3-16.

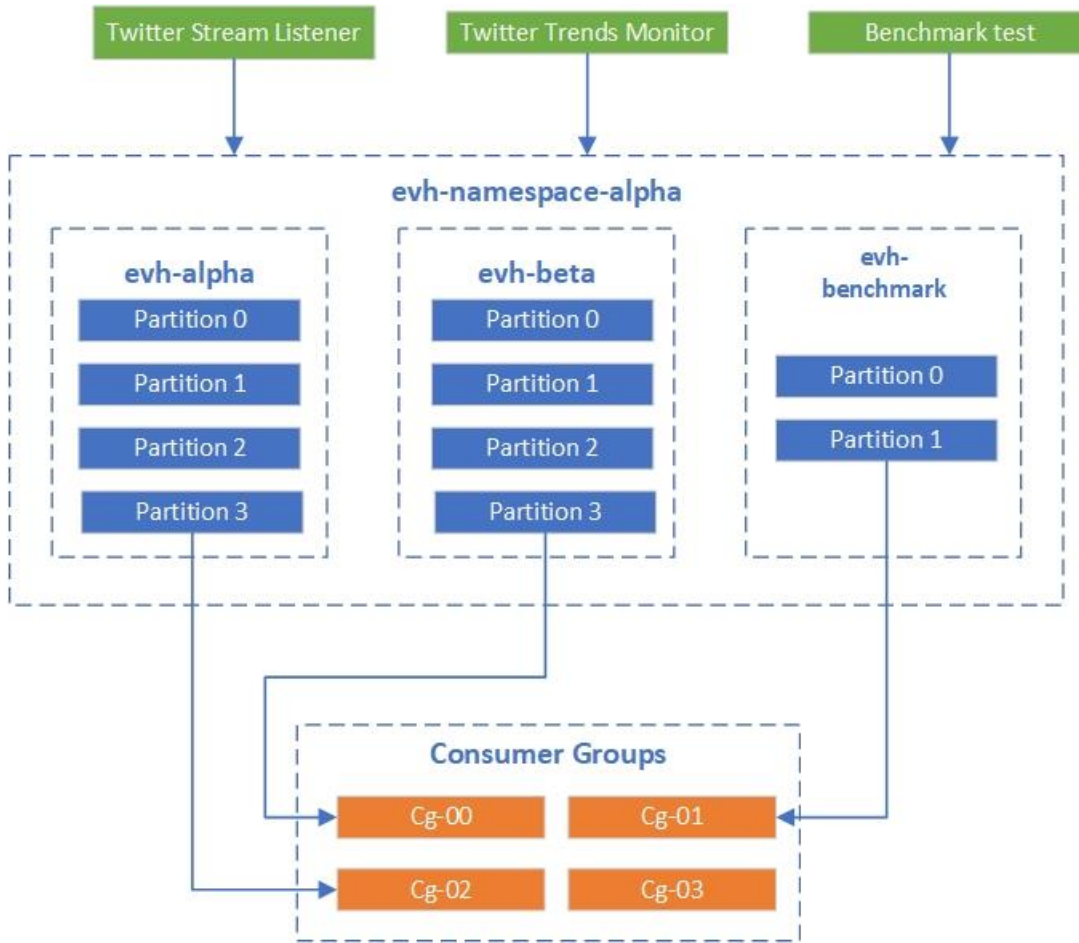


Figure 3-16: Architecture overview of the Event Hub “evh-namespace-alpha” implemented for the solution architecture.

Three event hubs have been created (evh-alpha, evh-beta and evh-benchmark), with each event hub configured as follows:

- 4 partitions
- 4 consumer groups

To be able to “replay” a stream, the event hubs are configured with a 7-day retention period. Stream replay can be utilized in scenarios where the ingested data need to be processed with a different algorithm or for troubleshooting purposes.

The Event Hub Namespace for the scenarios examined in this dissertation will be configured with 1 throughput unit, without the possibility to auto-inflate (effectively auto-scale) up to 2 throughput

units, as we will also be examining its performance under load in section 0. This means that *all* event hubs (throughput units are shared across all event hubs in the namespace we have deployed) in the namespace can consume¹⁸:

- Up to 2 MB per second of ingress events or 2000 ingress events per second (whichever comes first).
- Up to 4 MB per second of egress events, or 8192 egress events per second.
- Up to 168 GB of event storage for retention.

In case these numbers are violated, ingress will be throttled, and an exception will be received by the component that publishes the event¹⁹.

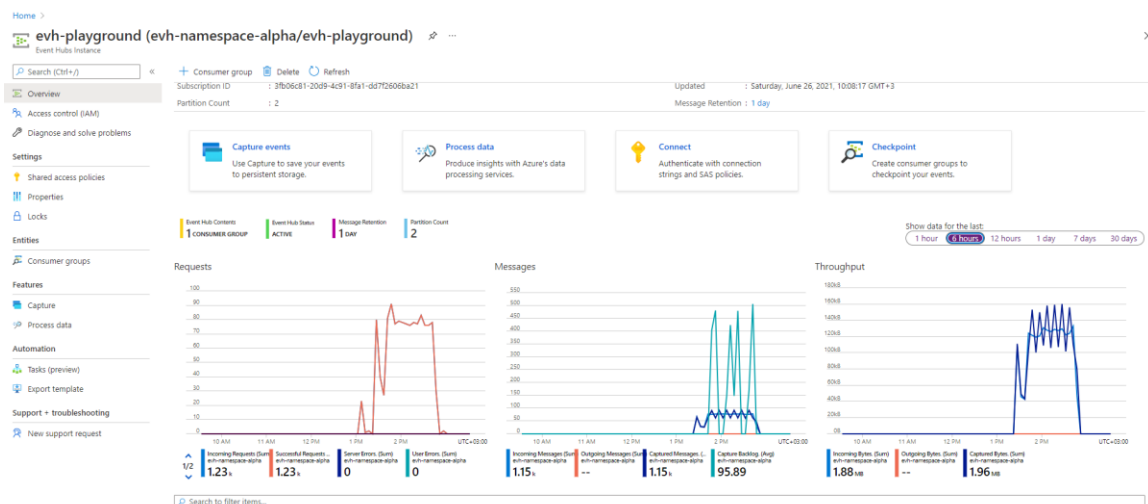


Figure 3-17: Overview in the Azure UI of an event hub deployed for the solution.

¹⁸ The numbers provided are calculated using the official documentation information, available at: <https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-scalability#throughput-units>.

¹⁹ Note that we currently do not perform exception handling due to Event Hub throttling in the producer tier's applications that we have developed for this solution.

3.4 Processing layer

The processing layer contains components that receive the events generated by the producer layer, process them and output them to the storage and analytics layers. In layered architectures, the processing activities that typically takes place in the processing layer can be:

- Cleansing
- Integration
- Indexing
- Fusion

In the solution, the processing layer consists of a Stream analytics Job which is responsible of:

- Receiving and parsing incoming events from the Event Hub.
- Processing the incoming events to aggregate and join data, for example to produce the top hashtags over a x-minute time window.
- Depending on the scenario, outputting the processed events in JSON format to:
 - o Blob files
 - o Cosmos DB container
 - o Power BI Dashboard

3.4.1 Stream Analytics Job

As the core functionality of the Azure Stream Analytics service has been presented in section 2.6.2.3, in this section we present more details on the service as well as the specific configuration that we have implemented for the solution under consideration.

Azure Stream Analytics is built on Trill, an in-memory, high-throughput streaming analytics engine [39]. Trill supports a wide range of use case scenarios for analytics such as:

1. **Real-time analytics:** Application monitoring, fraud detection.

2. **Real-time analytics with historical data:** Scenarios where it is needed to correlate live data stream with historical activity.
3. **Offline:** Back - testing an analytics algorithm over historical data, perform general data transformations on a bounded dataset.

Capability	Azure Stream Analytics Job	Apache Spark Streaming
Temporal/windowing support	Yes	Yes
Input data formats	Avro, JSON or CSV, UTF-8 encoded	Any format using custom code
Scalability	Yes, by query partitioning	Yes, bound by cluster size
Late arrival and out of order event handling support	Yes	Yes

Table 19: Figure 3 14: Comparison between Stream Analytics Job and Apache Spark Streaming²⁰

²⁰ The table contains information compiled from the Azure Data Architecture guide, available at <https://docs.microsoft.com/en-us/azure/architecture/data-guide/technology-choices/stream-processing>.

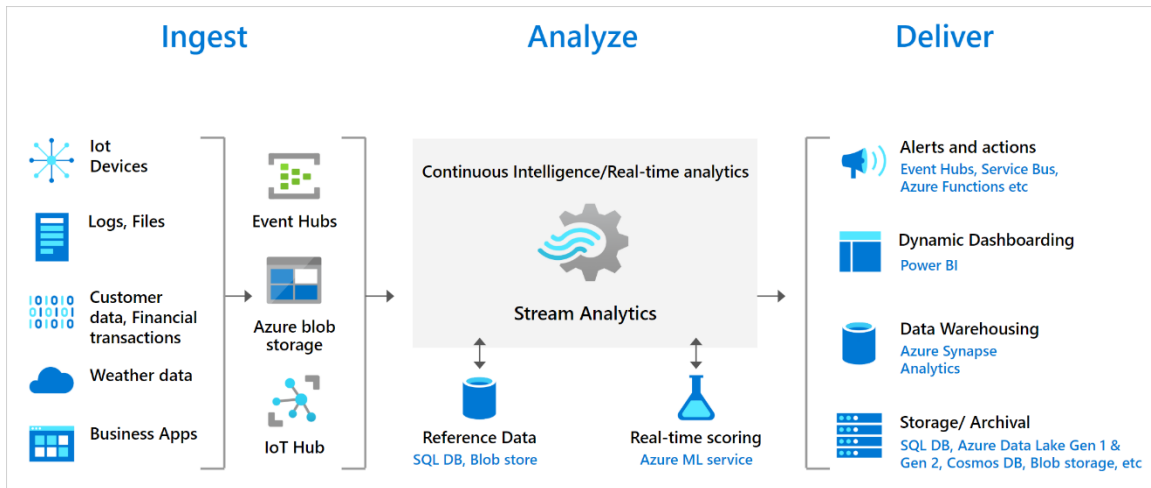


Figure 3-18: Reference high-level overview of an Azure Stream Analytics process²¹.

A Stream Analytics Job consists of the following three configuration items:

- Input
- Query
- Output

To perform computations, Stream Analytics offers the "Stream Analytics Query Language", which is a subset of standard T-SQL syntax. The listing below showcases an example stream analytics query using the Stream Analytics Query Language, with two steps and the usage of the column TollBoothId as partition key.

²¹ Source: Azure Stream Analytics documentation, available at <https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-introduction>.

```
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
```

Code block 3-1: Example ASA query

In our solution, the Stream Analytics job will work with the Twitter Stream Listener, the Botometer Checker and the Twitter Trends Monitor as input sources.

The job is configured with the following inputs:

1. **twitterStreamInput:** Reads the Event Hub topic which stores tweets captured with the Twitter Stream Listener application.
2. **twitterTrendsInput:** Reads the Event Hub topic which stores the trending topics for different countries captured with the Twitter Trends Monitor application.
3. **botometerInput:** Reads a blob path which contains twitter accounts checked with the Botometer Checker application along with their bot scores.

The job will perform the following outputs to persistent storage:

1. Send all received tweets from twitterStreamInput to a Cosmos DB container.
2. Send all received trends from twitterTrendsInput to a Cosmos DB container.

The stream analytics job also performs the following data aggregation outputs, both to a blob store and to a Power BI dashboard:

1. Creates a list for the top hashtags over the received tweets.
2. Creates a list of the top users over the received tweets.
3. Create a list of the most retweeted users.
4. Creates a list of the top locations over the received tweets.

All the above outputs are processed with a 2-minute tumbling window to produce near-real time results for the Power BI dashboard that is available to the end users of the pipeline.

The query that executes the processing described above is presented in Listing 1.

```
WITH rawTweets AS (  
    SELECT *  
    FROM [twitterStreamInput]  
),  
topUsers AS (  
    SELECT  
    System.Timestamp [WindowTime],  
    [user_screename],  
    COUNT(*) AS numberOfTweets  
    FROM  
    [twitterStreamInput]  
    GROUP BY  
    TumblingWindow(second, 60), [user_screename]  
),  
topRetweetedUsers AS (  
    SELECT  
    System.Timestamp [WindowTime],
```

```
[retweeted_tweet_user_screen_name],  
  
COUNT(*) AS numberOfRetweets  
  
FROM  
  
[twitterStreamInput]  
  
    WHERE [retweeted_tweet_user_screen_name] IS NOT NULL  
  
GROUP BY  
  
    TumblingWindow(second, 60), [retweeted_tweet_user_screen_name]  
  
)  
  
topLocations AS (  
  
SELECT  
  
    System.Timestamp [WindowTime],  
  
    [user_location],  
  
COUNT(*)  
  
FROM  
  
[twitterStreamInput]  
  
WHERE  
  
    [user_location] IS NOT NULL  
  
GROUP BY  
  
    TumblingWindow(second, 60), [user_location]  
  
HAVING COUNT(*) >= 1  
  
)  
  
topHashtags AS (  
  
SELECT  
  
    hashtags.Arrayvalue.text as tags  
  
FROM
```

```
[twitterStreamInput] as e
CROSS APPLY
    GetArrayElements(e.hashtags) as hashtags
),
trends AS (
    SELECT *
    FROM [twitterTrendsInput]
),
trendVolumes AS (
    SELECT
        i.locations,
        i.as_of as 'asOfDate',
        i.created_at as 'createdDate',
        hashtagLocation.ArrayValue.name as hashtagLocationName,
        hashtagLocation.ArrayValue.woeid as hashtagLocationWoeid,
        trendingHashtags.ArrayValue.name as hashtagName,
        trendingHashtags.ArrayValue.tweet_volume as hashtagVolume
    FROM twitterTrendsInput i
    CROSS APPLY GetArrayElements(trends) AS trendingHashtags
    CROSS APPLY GetArrayElements(i.locations) AS hashtagLocation
)

SELECT * INTO [twitterStreamOutput-Blob] FROM rawTweets

SELECT * INTO [twitterStreamOutput-Cosmos] FROM rawTweets
```

```
SELECT * INTO [twitterStreamOutput-locations-Blob] FROM topLocations
```

```
SELECT * INTO [twitterStreamOutput-locations-PBI] FROM topLocations
```

```
SELECT * INTO [twitterStreamOutput-topUsers-PBI] FROM topUsers
```

```
SELECT * INTO [topRetweetedUsers-PBI] FROM topRetweetedUsers
```

```
SELECT * INTO [twitterTrendsOutput-Cosmos] FROM trends
```

```
SELECT * INTO [trendingHashtags-Blob] FROM trendVolumes
```

```
SELECT
```

```
    COUNT(*)
```

```
INTO
```

```
    [twitterStreamOutput-volume-PBI]
```

```
FROM
```

```
[rawTweets]
```

```
GROUP BY
```

```
    TumblingWindow(second, 120)
```

```
SELECT
```

```
    tags, COUNT(*)
```

```
INTO
```



```
[twitterStreamOutput-hashtags-PBI]
FROM
    topHashtags
GROUP BY
    tags, TumblingWindow(second, 30)
SELECT *
INTO [greeceTrendsOutput-PBI]
FROM trendVolumes
WHERE hashtagLocationName = 'Greece'
```

Listing 1: The Stream Analytics Query used to process data captured in the Event Hub.

3.5 Persistent storage tier

The primary purpose of the storage tier is to persist events that are ingested and processed through the streaming pipeline for long term archiving, reference and other general-purpose retrieval. In addition, it stores the checkpoints from stateful Event Hub consumer applications (Stream Analytics Jobs), logs and other reference data.

The storage layer of the solution consists of the following resources:

- Blob Storage (to handle unstructured data)
- Azure Cosmos DB (to handle structured data)

The following paragraphs provide information on how each component has been configured.

3.5.1 Blob storage

Blob Storage is an Azure resource designed for storing unstructured data. Azure Blob storage supports several different types of blob entities, though in the solution we will be working with "block blobs", which store files used by applications such as the solution's Stream Analytics and

the Event Hub components. Blobs are organized into entities called containers, with a container being functionally similar to a directory in a file system.

Blobs exist under an "account" entity which provides a unique namespace for accessing the data stored within. The account name combined with the Azure Storage blob endpoint constitute the base address for the objects stored in the account. For the solution and based on the Blob Storage configuration, the endpoint will be: <http://st00000alpha.blob.core.windows.net>.

Figure 3-19 displays the blob containers and indicative files that will be stored in them.

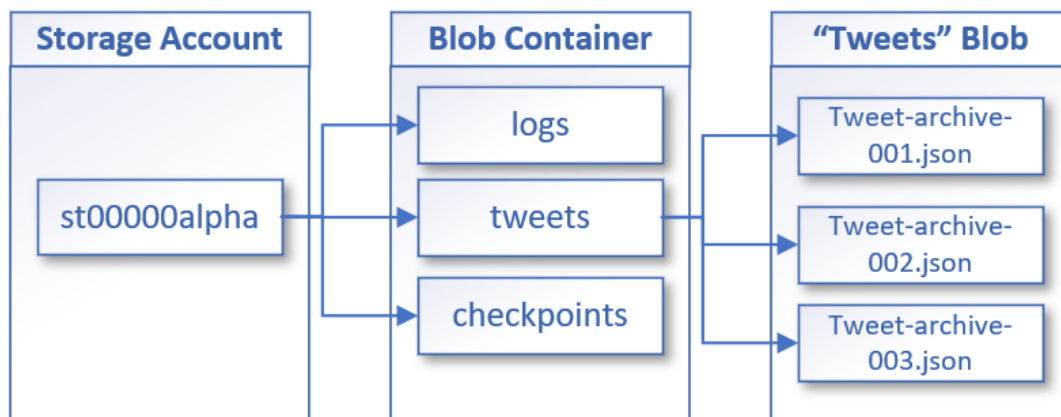


Figure 3-19: High level overview of the Blob storage account implemented for the solution.

3.5.2 Cosmos DB database

For the solution's needs a Cosmos DB resource has been provisioned with the following properties

For cost control purposes, the

The Cosmos DB resource in the solution has the following containers:

1. **tweets:** Stores tweets fetched from the Twitter Stream Listener application as JSON documents.

2. **botometerCheckedAccounts:** Stores information about Twitter accounts checked with the Botometer Checker application as JSON documents.
3. **botometerDailyCounter:** Stores details about how many times the Botometer Checker application is used on a daily basis.
4. **benchmarks:** Stores records processed from the NYC Taxi mock producer application which is used to perform performance tests on the solution as JSON documents.

Each container has been allocated 400 RUs and has been configured with a different partition key depending on the documents that it will store, as shown

CONTAINER	PARTITION KEY
Tweets	/PartitionId
Trends	/PartitionId
botometerCheckedAccounts	/user_majority_lang
botometerDailyCounter	/id

The database is set up to use the SQL API and thus gives us the possibility to execute SQL like statements over JSON documents. As an example, we provide below a few sample queries for the “tweets” container, which stores tweets captured with the Twitter Stream Listener application (see section 3.3.1).

```
SELECT {"Username":c.user_name, "StatusesCount":c.statuses_count} AS UserInfo
FROM c
WHERE c.statuses_count >= 1000
-- example results
[
  {
```

```
"UserInfo": {  
  "Username": "Maritel",  
  "StatusesCount": 2466  
}  
}  
]
```

Listing 2: Cosmos DB SQL query example - querying a subset of user properties that have tweeted above a certain volume.

```
SELECT {"Username":c.user_name, "StatusesCount":c.statuses_count} AS UserInfo  
FROM c  
WHERE c.statuses_count >= 1000  
  
-- example results  
[  
  {  
    "UserInfo": {  
      "Username": "Maritel",  
      "StatusesCount": 2466  
    }  
  }  
]
```

Listing 3: Cosmos DB SQL query example - querying a subset of user properties that have tweeted above a certain volume.

```
SELECT VALUE COUNT(1)  
FROM tweets c  
WHERE lower(c.hashtags[0].text) LIKE "%ακρο%"  
  
-- example results
```

```
[  
  6  
]
```

Listing 4: Cosmos DB SQL query example - counting tweets that contain a hashtag with a specified substring.

```
SELECT  
  
  c.user_name AS userName,  
  
  c.statuses_count AS statusCount,  
  
  c.user_created AS userAge,  
  
  c.hashtags as hashtags  
  
FROM tweets c  
  
WHERE lower(c.hashtags[0].text) LIKE "%ακρο%"  
  
-- example results
```

```
[
  {
    "userName": "Maritel",
    "statusCount": 2466,
    "userAge": "04-Oct-2020",
    "hashtags": [
      {
        "text": "Ακροπολη",
        "indices": [
          34,
          43
        ]
      }
    ]
  }
]
```

Listing 5: Cosmos DB SQL query example - querying usernames, hashtags and statuses count for tweets that contain a hashtag with a specified substring.

3.6 Presentation tier

For the stream analytics pipeline presented in this dissertation, the presentation layer consists of a Power BI dashboard which presents the outputs produced by the Stream Analytics job from section 3.4.1.

A version of this dashboard is displayed in Figure 3-20 below.

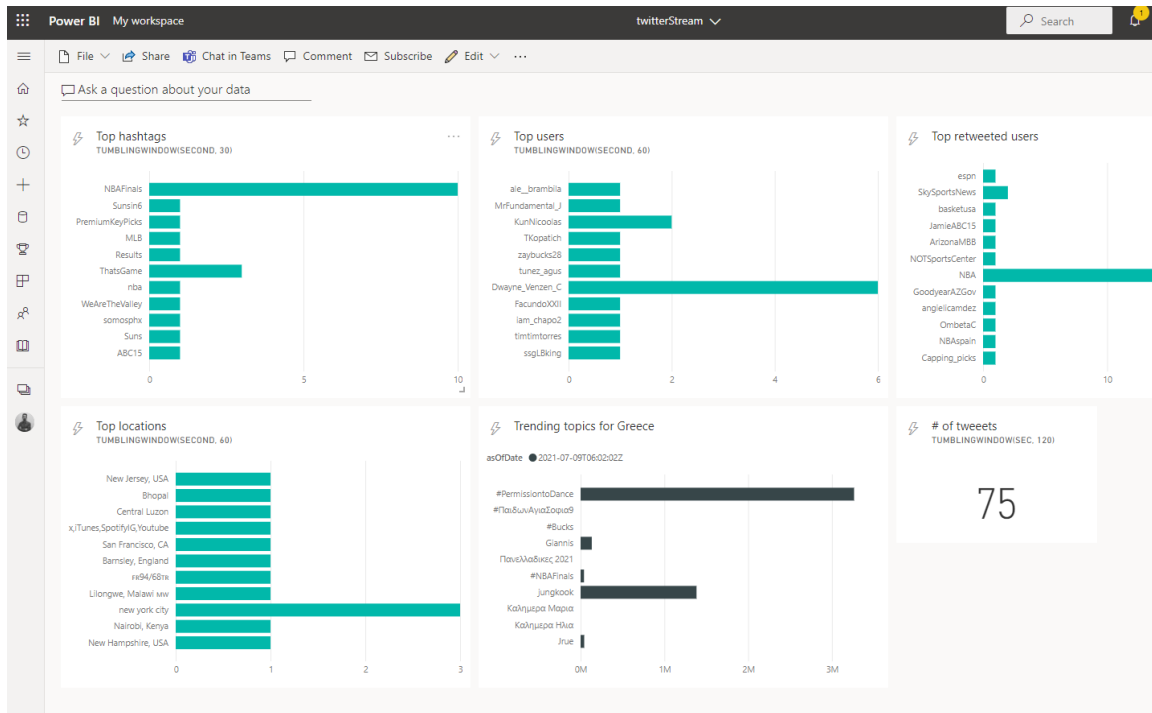


Figure 3-20: The Power BI dashboard used as the presentation layer of the pipeline.

Note that due to product limitations, it is currently not possible to automate the configuration of the dashboard and as such each tile displayed must be configured manually.

Chapter 4. Performance evaluation

In this section we present a series of evaluations carried out on the implemented solution to demonstrate its performance and scalability by executing streaming data ingestion and processing scenarios using a subset of the “NYC Taxi & Limousine Commission - green taxi trip records” dataset presented in section 3.2.4. We use the dataset as a source to imitate two data streams, one regarding the fare information (passengers, pick-up and drop-off coordinates, trip type) and another one regarding the payment information (fare, surcharges, taxes, tips). These events are published as events to two event hub topics. A Stream Analytics Job reads the event streams, joins the streams based on the partitions and performs aggregations on the incoming data (see Listing 6 below). Finally, the processed stream outputs are saved in a blob store. We are executing two scenarios, where we modify the throughput and processing capabilities of the Event Hub instances and of the Stream Analytics job. The data sample used is the same throughout both scenarios.

The data flow of this operation is shown in Figure 4-1 below. In the next paragraphs we go through the configuration of the event hub and the stream analytics job that we use for the scenario and present the results.

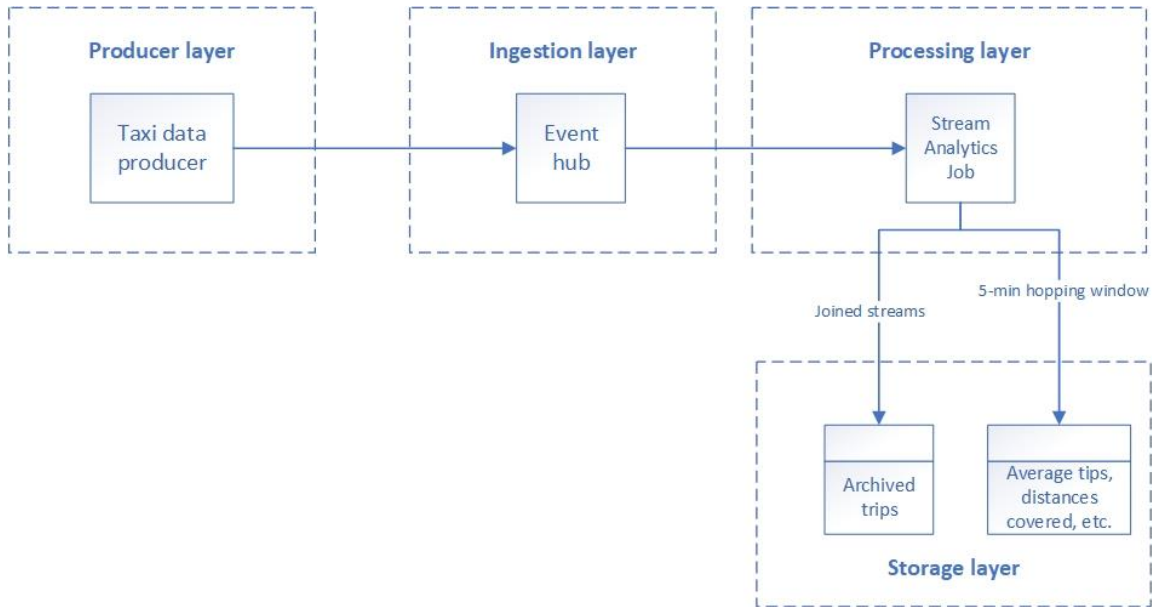


Figure 4-1: Data flow overview of the benchmark

4.1 Dataset and producer application configuration

In this section we present the application setup which we use to publish the trip records to the Event Hub. A sample of the “NYC Taxi & Limousine Commission – green taxi trip records” dataset’s records that will be processed in the scenarios is available at Sample ingestion data. For the performance evaluation scenario, we have developed a Jupyter notebook ²² which performs the following:

1. Loads the “NYC Taxi & Limousine Commission – green taxi trip records” dataset using the `azureml.opendatasets` Python library.
2. Samples 500,000 records from the dataset and loads them into a Pandas Dataframe object (the dataset originally exists as a parquet file in the Python library).
3. Splits the dataset into two separate datasets:
 - a. Fare data: Contains information about the fare. We use the following columns from the dataset:

²² The notebook is published in GitHub and is available in the following link: <https://github.com/orestisf/Notebooks/blob/master/NYC-Green-Taxi-Event-Hub-Publisher.ipynb>.

- i. vendorID
 - ii. tripType
 - iii. lpepPickupDatetime lpepDropoffDatetime
 - iv. passengerCount
 - v. tripDistance
 - vi. puLocationId
 - vii. doLocationId
 - viii. pickupLongitude
 - ix. pickupLatitude
 - x. dropoffLongitude
 - xi. dropoffLatitude
 - b. Payment data: Contains information about the trip's payment. We use the following columns from the dataset:
 - i. vendorID
 - ii. rateCodeID
 - iii. paymentType
 - iv. fareAmount
 - v. extra
 - vi. mtaTax
 - vii. improvementSurcharge
 - viii. tipAmount
 - ix. tollsAmount
 - x. totalAmount
4. For each record we use the data frame's index as the record's unique ID so that it can be used by the stream analytics job to correlate the events in the two streams.
 5. Converts each dataframe row to JSON.

6. Instantiates two event hub producer clients (one for the “payment” data and one for the “fare” data) and sends each JSON as an event to the corresponding Event Hub.
7. Sends each record as event to the designated event hub, in batches of 1000 records:
 - a. Records from the “fare” dataset are sent to the evh-taxi-fare-data event hub.
 - b. Records from the “payments” dataset are sent to the evh-taxi-payment-data event hub.

4.2 Event hub and Stream Analytics job configuration

For the performance evaluation scenario, we will use two event hub instances (evh-taxi-fare-data, evh-taxi-payment-data), with two partitions each. We will use the least resource-intensive configuration for both components, in order to establish a baseline of the performance that can be achieved. For the second scenario, we enable auto-inflate on the Event Hub up to 4x the baseline TPUs and increase the processing power of the Stream Analytics Job to 3x. Based on this, the configuration is summarized in the following table.

Benchmark scenario	Event Hub configuration	Stream analytics job configuration
Scenario 1	<ol style="list-style-type: none"> 1. 1 throughput unit (TPU). 2. Auto-inflate disabled. 3. 2 partitions. 	1 streaming unit (SU).
Scenario 2	<ol style="list-style-type: none"> 1. 1 throughput unit 2. Auto-inflate enabled with 4 throughput units maximum. 3. 2 partitions. 	3 streaming units.

Table 20: Event Hub and Stream Analytics job configuration for benchmark scenarios

The query that is executed by the stream analytics job is visualized in Figure 4-2. Listing 6 contains the query that we use to process the events.

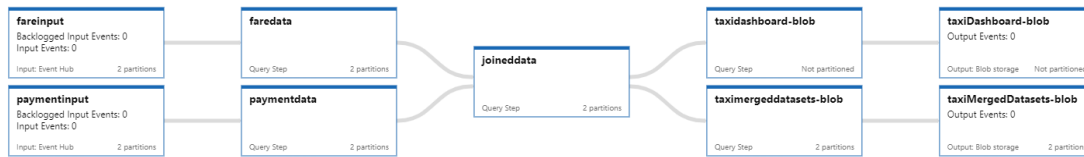


Figure 4-2: Stream analytics job visualization used for the benchmark scenarios.

The job functions as follows:

1. Reads the events from the event hub topics using partitioning based on the partition ID (WITH statements fareData and paymentData).
2. Joins the events based on the partition ID and the record ID (joinedData step).
3. Outputs to a blob the average tip per mile amount (calculated as the sum of tips divided by the sum of trip distances), the average trip distance and the average number of passengers over a 5-minute hopping window with 1-minute hops.
4. Outputs to another blob the joined record for archiving purposes and to validate the results.

```
WITH fareData AS (
    SELECT recordId AS fareId,
           vendorID,
           DATEADD(millisecond, lpepPickupDatetime, '1970-01-01T00:00:00Z')
    as 'PickupTime',
           DATEADD(millisecond, lpepDropoffDatetime, '1970-01-01T00:00:00Z')
    as 'DropoffTime',
           tripType,
           tripDistance,
           passengerCount,
           PartitionId
    FROM [fareinput]
```

```
        PARTITION BY PartitionId
    ),
paymentData AS (
    SELECT recordId as paymentId,
           vendorID,
           rateCodeID,
           paymentType,
           fareAmount,
           extra,
           mtaTax,
           improvementSurcharge,
           tipAmount,
           tollsAmount,
           totalAmount,
           PartitionId
    FROM [paymentinput]
    PARTITION BY PartitionId
),
joinedData AS (
    SELECT
        tf.fareId,
        tf.tripDistance,
        tf.tripType,
        tf.passengerCount,
```

```
tf.vendorID,  
tp.paymentId,  
tp.paymentType,  
tp.fareAmount,  
tp.mtaTax,  
tp.extra,  
tp.tipAmount,  
tp.totalAmount  
FROM [fareData] tf  
PARTITION BY PartitionId  
JOIN [paymentData] tp PARTITION BY PartitionId  
ON tf.PartitionId = tp.PartitionId  
AND tf.vendorID = tp.vendorID  
AND tf.fareId = tp.paymentId  
--AND tr.PickupTime = tf.PickupTime  
AND DATEDIFF(minute, tf, tp) BETWEEN 0 AND 5  
)  
SELECT System.Timestamp AS WindowTime, tr.vendorID,  
       ROUND(SUM(tr.TipAmount) / SUM(tr.tripDistance),3) AS  
AverageTipPerMile,  
       ROUND(AVG(tr.tripDistance),3) AS AverageTripDistange,  
       ROUND(AVG(tr.passengerCount),3) AS AveragePassengerCount  
INTO [taxiDashboard-blob]  
FROM [joinedData] tr
```

```

GROUP BY HoppingWindow(Duration(minute, 5), Hop(minute, 1)),
tr.VendorID

select *

INTO [taxiMergedDatasets-blob]

FROM [joinedData]

```

Listing 6: The stream analytics job query that is used for the benchmark scenarios.

4.3 Results

To monitor the Event Hub and Stream Analytics job performance we use Azure's built-in resource metrics that are automatically available when deploying a component. To monitor the performance during the test execution we have designed a dashboard which monitors in real-time the component metrics, as shown in Figure 4-3.

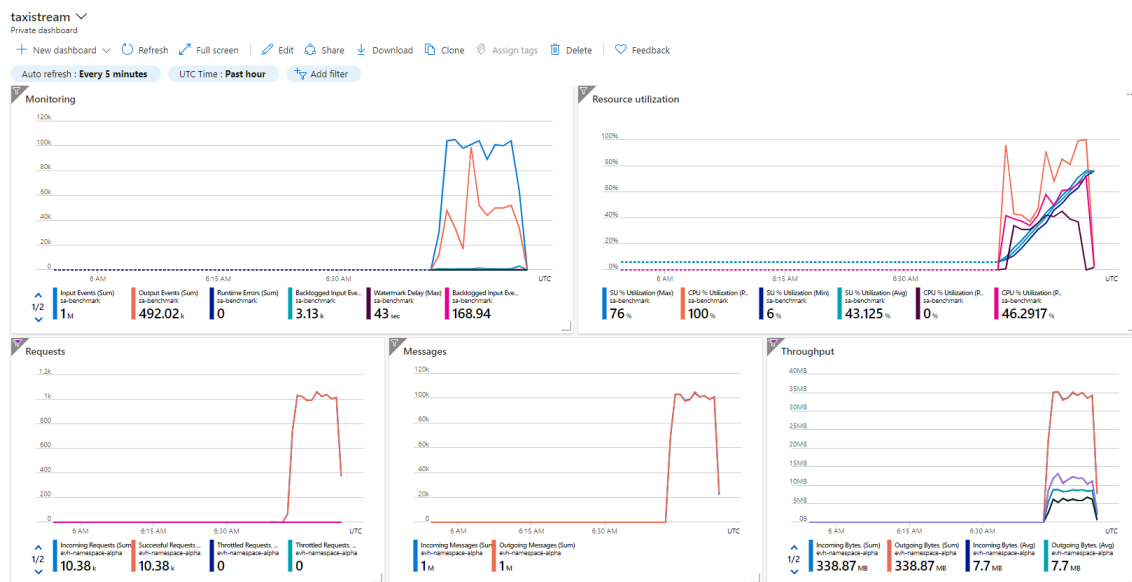


Figure 4-3: Azure dashboard we used to monitor the Event Hub and Stream Analytics job performance during the scenario execution.

Table 21 presents the metrics monitored during the scenario execution. For each metric we have measured the sum (where applicable) as well as the max, min, and average values. We also calculate the performance change for each metric between the two scenarios.

Metric	Scenario 1 (1 TPU / 1 SU)				Scenario 2 (1 TPU auto- inflate to 4 TPU / 3 SU)				Performance difference
	Sum	Max	Min	Average	Sum	Max	Min	Average	
SU % Utilization	-	91 %	14 %	60.25 %	-	76 %	6 %	48.85 %	18.92 %
Backlogged Input Events	-	9.700	0	298.30	-	3.130		129.40	56.62 %
Watermark Delay	-	1.33 min	0 sec	3.32 sec		43 sec	0 sec	1.80 sec	45.78 %
Incoming Bytes	376.7 MB	10.6 MB	267.4 Kb	3 MB	338.9 MB	13.1 MB	539.4 Kb	7.7 MB	156.67 %
Outgoing Bytes	338.9 MB	10.6 MB	794.5 Kb	4.8 MB	338.9 MB	13.1 MB	0 B	7.1 MB	47.92 %
Throttled Requests	113	4	1	2.09	0	0	0	0	100%

Table 21: Summary of the metrics observed for the evaluation scenarios.

From the scenario execution we observe the following:

- **Throttled requests** were eliminated after enabling auto-inflate.
- **Throughput** (Incoming bytes / Outgoing bytes) rate was almost doubled for both incoming and outgoing side.
- **Streaming Unit utilization** dropped by ~20% – note here that, even the observed average of 60.25% of the 1st scenario is not considered high enough to cause bottlenecks, however it did cause throttled requests and delayed events.

- **Watermark delay** was visibly improved from 3.32 sec to 1.80 sec on average which is also consistent with the decrease of backlogged input events and the overall better performance of the pipeline during scenario 2.

Figure 4-4 displays visualizations of a subset of the metrics that were monitored during the scenario execution.



Figure 4-4: Visualizations of throughput, requests, SU utilization and backlogged event metrics for both scenarios.

The figure below shows a sample of the output JSON files captured in the blob store as outputs the stream analytics job.

```
0_9800b12f34f5455a885e82a7a07c0eb_1.json 1 X
C:\Users> ofoti > Downloads > 0_9800b12f34f5455a885e82a7a07c0eb_1.json > ...
1 {"WindowTime": "2021-07-07T06:11:00.000000Z", "VendorID": 2, "AverageTipPerMile": 0.383, "AverageTripDistance": 2.72, "AveragePassengerCount": 1.424}
2 {"WindowTime": "2021-07-07T06:11:00.000000Z", "VendorID": 1, "AverageTipPerMile": 0.491, "AverageTripDistance": 2.669, "AveragePassengerCount": 1.262}
3 {"WindowTime": "2021-07-07T06:12:00.000000Z", "VendorID": 2, "AverageTipPerMile": 0.383, "AverageTripDistance": 2.72, "AveragePassengerCount": 1.424}
4 {"WindowTime": "2021-07-07T06:12:00.000000Z", "VendorID": 1, "AverageTipPerMile": 0.491, "AverageTripDistance": 2.669, "AveragePassengerCount": 1.262}
5 {"WindowTime": "2021-07-07T06:13:00.000000Z", "VendorID": 2, "AverageTipPerMile": 0.383, "AverageTripDistance": 2.72, "AveragePassengerCount": 1.424}
6 {"WindowTime": "2021-07-07T06:13:00.000000Z", "VendorID": 1, "AverageTipPerMile": 0.491, "AverageTripDistance": 2.669, "AveragePassengerCount": 1.262}
7 {"WindowTime": "2021-07-07T06:14:00.000000Z", "VendorID": 2, "AverageTipPerMile": 0.383, "AverageTripDistance": 2.72, "AveragePassengerCount": 1.424}
8 {"WindowTime": "2021-07-07T06:14:00.000000Z", "VendorID": 1, "AverageTipPerMile": 0.491, "AverageTripDistance": 2.669, "AveragePassengerCount": 1.262}
9 {"WindowTime": "2021-07-07T06:15:00.000000Z", "VendorID": 2, "AverageTipPerMile": 0.448, "AverageTripDistance": 2.84, "AveragePassengerCount": 1.386}

0_309c4d1c5904be79ef14184a3ade2c9_1 (1).json 1 X
C:\Users> ofoti > Downloads > 0_309c4d1c5904be79ef14184a3ade2c9_1 (1).json > ...
1 [{"fareId": 2816, "tripDistance": 11.3, "tripType": 1.0, "passengerCount": 1, "VendorID": 2, "paymentId": 2816, "paymentType": 1, "fareAmount": 17.0, "metaTax": 0.5, "extra": 0.5, "tipAmount": 1.66, "totalAmount": 19.96}]]
2 [{"fareId": 2818, "tripDistance": 0.91, "tripType": 1.0, "passengerCount": 1, "VendorID": 2, "paymentId": 2818, "paymentType": 2, "fareAmount": 5.0, "metaTax": 0.5, "extra": 0.5, "tipAmount": 0.0, "totalAmount": 6.3}]]
3 [{"fareId": 2819, "tripDistance": 1.83, "tripType": 1.0, "passengerCount": 1, "VendorID": 2, "paymentId": 2819, "paymentType": 2, "fareAmount": 8.5, "metaTax": 0.5, "extra": 0.5, "tipAmount": 0.0, "totalAmount": 9.8}]]
4 [{"fareId": 2821, "tripDistance": 1.82, "tripType": 1.0, "passengerCount": 1, "VendorID": 2, "paymentId": 2821, "paymentType": 2, "fareAmount": 10.0, "metaTax": 0.5, "extra": 0.5, "tipAmount": 0.0, "totalAmount": 11.3}]]
5 [{"fareId": 2822, "tripDistance": 0.80, "tripType": 1.0, "passengerCount": 1, "VendorID": 2, "paymentId": 2822, "paymentType": 2, "fareAmount": 3.0, "metaTax": 0.5, "extra": 0.5, "tipAmount": 0.0, "totalAmount": 4.3}]]
6 [{"fareId": 2829, "tripDistance": 1.41, "tripType": 1.0, "passengerCount": 1, "VendorID": 2, "paymentId": 2829, "paymentType": 2, "fareAmount": 9.0, "metaTax": 0.5, "extra": 0.5, "tipAmount": 0.0, "totalAmount": 10.3}]]
```

Figure 4-5: Sample JSON of the stream analytics job outputs.

The two figures below display the built-in job monitor diagram that was produced by the Stream Analytics job. The increased performance due to the scaled-up components is visible in the second figure.

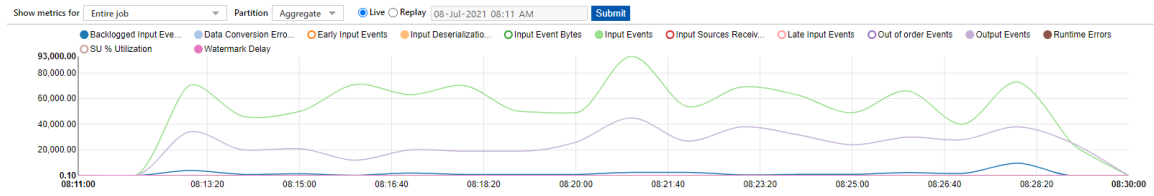


Figure 4-6: Stream Analytics job performance for scenario 1

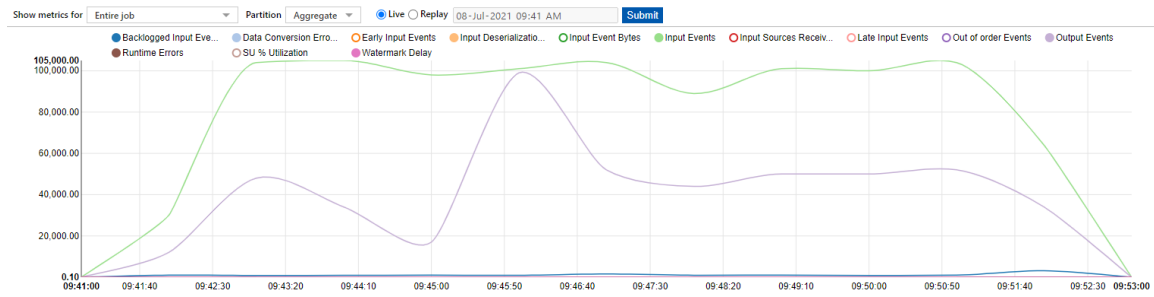


Figure 4-7: Stream Analytics job performance for scenario 2

From the results posted for the two scenarios we examined, it is evident that the proposed pipeline is more than adequate at capturing and processing inputs from the Twitter Stream Listener, Trends Monitor and Botometer Checker applications.

In addition, when there is a need to process – or re-process a fairly large dataset, for example to perform an ad-hoc analysis of archived data, both components of the solution can be adequately scaled to the task’s needs. Note that the maximum TPUs supported by the Event Hub is 20 and the maximum SUs supported by the Stream Analytics Job is 192 and as such even the “high-performing” scenario two is positioned exceptionally low on the full spectrum of the components’ capabilities.

Chapter 5. Conclusions and future work

The last decade, the rise of big data has given birth to a multitude of technologies and frameworks to be used in receiving processing and consuming big data streams. These technologies, initially available in commodity form (i.e. as open-source projects) have evolved to be offered as commercial services via all the major cloud providers, thus enabling their easier integration in corporate infrastructures.

In this dissertation we provided an overview of the primary big data processing paradigms that are in use the last years, the batch and stream processing. We presented the two major architectural frameworks, the “Lambda” and “Kappa” architecture, discussed the advantages and disadvantages of each architecture and the problems that they attempt to solve. We designed and implemented an end-to-end stream processing solution based on the Kappa architecture and hosted it in the Azure cloud. We also provided a method to deploy all the architecture’s components, so that it can be easily deployed with new configuration to cover a wide range of use cases. Finally, all the solution’s artifacts and deployment templates were published to GitHub and Docker Hub.

The real-time stream processing solution proposed and discussed in section Chapter 3 can be improved in several areas, due to its modular design and depending on the business use cases that it will have to handle. We are discussing several such improvements in the following paragraphs.

- The **ingestion layer** can be expanded by developing additional “connector” applications to enable more sources to be ingested. As an example, for potential new implementation, we consider that the recent popularity of the social media application “TikTok” has already resulted in a rising academic interest²³ in collecting and analyzing content generated by TikTok users. Similarly to the Twitter Stream Listener component we developed for this dissertation (3.2.1), a Python application²⁴ can be developed to capture messages published to TikTok, extract their metadata and publish them to an Azure Event Hub.
- The **component deployment process** can be further automated by implementing a proper workflow that will deploy the entire solution infrastructure from scratch, as currently this process needs to be done manually following the steps described in Solution deployment guide. In addition, the resource templates can be re-implemented with the Bicep²⁵ Domain Specific Language that has been released by Microsoft with the primary purpose of being the best language to describe, validate, and deploy infrastructure to Azure.
- The **processing layer** can be expanded by implementing an alert subsystem that will trigger **email alerts** to end users when a particular predefined event takes place, for example when the tweet volume of a hashtag of interest surpasses a specific value. A prototype implementation can leverage the SendGrid²⁶ API to deliver these alerts.
- A **machine learning** subsystem can be added to the solution to enhance its insight extraction capabilities²⁷. As an example, a sentiment analysis can be performed in the data

²³ A collection of scholarly bibliography on TikTok is available in the following link: <https://tiktokcultures.com/bibliography/>.

²⁴ An unofficial Python API wrapper for TikTok is already available and can be utilized towards that purpose: <https://github.com/davidteather/TikTok-API>.

²⁵ Bicep is Domain Specific Language (DSL) for deploying Azure resources declaratively and is a transparent abstraction over ARM and ARM templates: <https://github.com/Azure/bicep>.

²⁶ See also the SendGrid API documentation at <https://docs.sendgrid.com/for-developers/sending-email/api-getting-started>.

²⁷ Azure provides a machine learning service that can be utilized for that purpose: <https://azure.microsoft.com/en-us/services/machine-learning/>.

collected from Twitter hashtags and have its results (i.e. the prevailing sentiment in the tweets for a hashtag over a hourly window) published in a dashboard.

Further utilization of the developed solution can involve the **automated collection and publication of a Twitter dataset** on topics that can be of use to the academic or business community: This would be possible by continuously monitoring over time of one or more Twitter hashtags related to a particular topic, extracting a subset of their attributes, storing the processed tweets in a persistent storage and publishing the resulting dataset in a publicly accessible location²⁸ for further usage by the academic community.

The domain of data stream processing and analytics is still relatively young: There are still technical and business challenges to be solved. As a closing remark we have to point out that the ever-increasing need for real-time data processing and the evolving needs of businesses and organizations that want to take advantage of these data is set to pose new challenges as there are increasing needs in scalability, resilience, security and other aspects. As such, the underlying frameworks and architectures will have to evolve as well to tackle those challenges.

²⁸ A similar work has been done by in collecting and publishing tweets in the Arabic language related to COVID-19. The authors are publishing the collected tweets at: <https://github.com/SarahAlqurashi/COVID-19-Arabic-Tweets-Dataset>.

Availability of data and materials

All code developed for this thesis has been made available in public repositories listed in

Supplementary materials. The same section also contains links to sample datasets collected for the purpose of demonstrating the functionality of the ingestion components.

References

- [2]. N. Tantalaki, S. Souravlas and M. Roumeliotis, "A review on big data real-time stream processing and its scheduling techniques", International Journal of Parallel, Emergent and Distributed Systems, vol. 35, no. 5, pp. 571-601, 2019. Available: 10.1080/17445760.2019.1585848.
- [3]. M. Stonebraker, U. Çetintemel and S. Zdonik, "The 8 requirements of real-time stream processing", ACM SIGMOD Record, vol. 34, no. 4, pp. 42-47, 2005. Available: 10.1145/1107499.1107504.
- [4]. B. Berg, "Aurora Project Page", cs.brown.edu, 2004. [Online]. Available: <http://cs.brown.edu/research/aurora/>.
- [5]. "The Borealis Project: Distributed Stream Processing Engine", cs.brown.edu, 2007. [Online]. Available: <http://cs.brown.edu/research/borealis/public/>.
- [6]. "Apache Storm: free and open source distributed real-time computation system", storm.apache.org, 2020. [Online]. Available: <https://storm.apache.org>.
- [7]. "Apache Spark: unified analytics engine for large-scale data processing project", spark.apache.org, 2021 [Online]. Available: <https://spark.apache.org/>.

- [8]. “Apache Flink: Stateful Computations over Data Streams”, flink.apache.org, 2021 [Online]. Available: <https://flink.apache.org>.
- [9]. “Apache Samza: A distributed stream processing framework”, samza.apache.org, 2020 [Online]. Available: <https://samza.apache.org>.
- [10]. “Amazon Kinesis: Easily collect, process, and analyze video and data streams in real time”, aws.amazon.com, 2021 [Online]. Available: <https://aws.amazon.com/kinesis>.
- [11]. Azure Stream Analytics: Serverless real-time analytics, from the cloud to the edge, 2021 <https://azure.microsoft.com/en-us/services/stream-analytics>.
- [12]. Google Cloud Dataflow: Unified stream and batch data processing that's serverless, fast, and cost-effective, 2021 <https://cloud.google.com/dataflow>.
- [13]. “InfoSphere Information Server: Flexibly meet your unique requirements — from data integration to data quality and data governance”, [Ibm.com](http://ibm.com), 2021. [Online]. Available: <https://www.ibm.com/analytics/information-server>.
- [14]. J. Urquhart, Flow architectures: The Future of Streaming and Event-Driven Integration, 1st ed. O’Reilly Media, Inc., 2021.
- [15]. R. C. Atkinson, R. M. Shiffrin, “Human memory: A proposed system and its control processes.” K. W. Spence & J. T. Spence, The psychology of learning and motivation: II. Academic Press, 1968. [https://doi.org/10.1016/S0079-7421\(08\)60422-3](https://doi.org/10.1016/S0079-7421(08)60422-3)
- [16]. “Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation)”, 2016. Available: <https://eur-lex.europa.eu/legal-content/EN/ALL/?uri=celex%3A32016R0679>.
- [17]. A. Psaltis, Streaming data, 1st ed. Shelter Island, NY: Manning Publications, 2017.
- [18]. M. Özsu, P. Valduriez, Principles of Distributed Database Systems, Third Edition. New York, Springer New York, 2011.

- [19]. Gilbert, A. C., Kotidis, Y., Muthukrishnan, S., and Strauss, M. J. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. Proc. 27th Int. Conf. on Very Large Data Bases, 2001. Available at: <http://www.vldb.org/conf/2001/P079.pdf>.
- [20]. Akidau, T., Chernyak, S. and Lax, R., 2018. Streaming systems. 1st ed. Sebastopol: O'Reilly Media.
- [21]. E. Levy, "Batch, Stream, and Micro-batch Processing: A Cheat Sheet", Upsolver, 2021. [Online]. Available: <https://www.upsolver.com/blog/batch-stream-a-cheat-sheet>.
- [22]. M. Sarda, "Batch Processing vs Stream Processing in Microsoft Azure", k21academy, 2020. [Online]. Available: <https://k21academy.com/microsoft-azure/data-engineer/batch-processing-vs-stream-processing/>.
- [23]. L. Shiff, "Batch Processing: An Introduction", BMC Blogs, 2020. [Online]. Available: <https://www.bmc.com/blogs/what-is-batch-processing-batch-processing-explained/>.
- [24]. "Batch Processing vs Real Time Data Streams", Confluent.io, 2021. [Online]. Available: <https://www.confluent.io/learn/batch-vs-real-time-data-processing/>.
- [25]. M. P. Singh, M. A. Hoque and S. Tarkoma, "A survey of systems for massive stream analytics", 2017. [Online]. Available: arXiv:05.09021v2.
- [26]. B. Batrinca and P. Treleaven, "Social media analytics: a survey of techniques, tools and platforms", AI & SOCIETY, vol. 30, no. 1, pp. 89-116, 2014. Available: <https://link.springer.com/article/10.1007/s00146-014-0549-4>.
- [27]. N. Panagiotou et al., "Intelligent Urban Data Monitoring for Smart Cities", Machine Learning and Knowledge Discovery in Databases, pp. 177-192, 2016. Available: <http://www.katakis.eu/wp-content/uploads/2013/07/ECML2016i.pdf>.
- [28]. H. Jaakkola and B. Thalheim, "Architecture-Driven Modelling Methodologies", Information Modelling and Knowledge Bases, vol. 12, pp. 97-116, 2011. [Online]. Available: https://www.researchgate.net/publication/221014046_Architecture-Driven_Modelling_Methodologies.

- [29]. S. Ounacer, M. Amine, S. Ardchir, A. Daif and M. Azouazi, "A New Architecture for Real Time Data Stream Processing", International Journal of Advanced Computer Science and Applications, vol. 8, no. 11, 2017. Available: https://www.researchgate.net/publication/321494828_A_New_Architecture_for_Real_Time_Data_Stream_Processing/link/5c21662b92851c22a3443831/download.
- [30]. N. Marz and J. Warren, Big Data: Principles and best practices of scalable real-time data systems, 1st ed. Shelter Island: Manning Publications Co, 2016.
- [31]. N. Marz, "How to beat the CAP theorem", Nathanmarz.com, 2021. [Online]. Available: <http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html>.
- [32]. J. Kreps, "Questioning the Lambda Architecture", O'Reilly Media, 2014. [Online]. Available: <https://www.oreilly.com/radar/questioning-the-lambda-architecture/>.
- [33]. T. Akidau, "Streaming 101: The world beyond batch", O'Reilly Media, 2015. [Online]. Available: <https://www.oreilly.com/radar/the-world-beyond-batch-streaming-101/>.
- [34]. "Architectural Principles", Opengroup.org, 1998. [Online]. Available: <http://www.opengroup.org/public/arch/p4/princ/princ.htm>.
- [35]. J. Zhu, B. Tang and V. Li, "A five-layer architecture for big data processing and analytics", International Journal of Big Data Intelligence, vol. 6, no. 1, p. 38, 2019. Available: <http://www.inderscience.com/storage/f811410691235271.pdf>.
- [36]. "Apache Samza - Core concepts", Samza.apache.org, 2021. [Online]. Available: <http://samza.apache.org/learn/documentation/1.6.0/core-concepts/core-concepts.html>.
- [37]. "Apache Samza - Architecture", Samza.apache.org, 2021. [Online]. Available: <https://samza.apache.org/learn/documentation/0.11/introduction/architecture.html>.
- [38]. "Using the Kinesis Client Library - Amazon Kinesis Data Streams", Docs.aws.amazon.com, 2021. [Online]. Available: <https://docs.aws.amazon.com/streams/latest/dev/shared-throughput-kcl-consumers.html>.

- [39]. "Amazon Kinesis Data Streams Terminology and Concepts - Amazon Kinesis Data Streams", Docs.aws.amazon.com, 2021. [Online]. Available: <https://docs.aws.amazon.com/streams/latest/dev/key-concepts.html>.
- [40]. B. Chandramouli et al., "Trill: A High-Performance Incremental Query Processor for Diverse Analytics", Proceedings of the VLDB Endowment, vol. 8, no. 4, pp. 401-412, 2014. Available: 10.14778/2735496.2735503.
- [41]. "Consuming streaming data", Developer.twitter.com. [Online]. Available: <https://developer.twitter.com/en/docs/tutorials/consuming-streaming-data>.
- [42]. "GET trends/place", Developer.twitter.com, 2021. [Online]. Available: <https://developer.twitter.com/en/docs/twitter-api/v1/trends/trends-for-location/api-reference/get-trends-place>.
- [43]. "Botometer by OSoMe", Botometer.osome.iu.edu, 2021. [Online]. Available: <https://botometer.osome.iu.edu/api>.
- [44]. "NYC Taxi & Limousine Commission - green taxi trip records - Azure Open Datasets Catalog", Azure.microsoft.com, 2021. [Online]. Available: <https://azure.microsoft.com/en-us/services/open-datasets/catalog/nyc-taxi-limousine-commission-green-taxi-trip-records/>.
- [45]. J. Roesslein, "Tweepy: Twitter for Python!", An easy-to-use Python library for accessing the Twitter API., 2021. [Online]. Available: <https://Github.Com/Tweepy/Tweepy>.
- [46]. M. Trupthi, S. Pabboju and G. Narasimha, "Sentiment Analysis on Twitter Using Streaming API", 2017 IEEE 7th International Advance Computing Conference (IACC), 2017. Available: 10.1109/iacc.2017.0186.
- [47]. A. Bifet, G. Holmes and B. Pfahringer, "MOA-TweetReader: Real-Time Analysis in Twitter Streaming Data", Discovery Science, pp. 46-60, 2011. Available: 10.1007/978-3-642-24477-3_7.

- [48]. M. Hasan, M. Orgun and R. Schwitter, "Real-time event detection from the Twitter data stream using the TwitterNews+ Framework", *Information Processing & Management*, vol. 56, no. 3, pp. 1146-1165, 2019. Available: [10.1016/j.ipm.2018.03.001](https://doi.org/10.1016/j.ipm.2018.03.001).
- [49]. L. Sinnenberg, A. Buttenheim, K. Padrez, C. Mancheno, L. Ungar and R. Merchant, "Twitter as a Tool for Health Research: A Systematic Review", *American Journal of Public Health*, vol. 107, no. 1, pp. e1-e8, 2017. Available: [10.2105/ajph.2016.303512](https://doi.org/10.2105/ajph.2016.303512).
- [50]. I. Annamoradnejad and J. Habibi, "A Comprehensive Analysis of Twitter Trending Topics", 2019 5th International Conference on Web Research (ICWR), 2019. Available: [10.1109/icwr.2019.8765252](https://doi.org/10.1109/icwr.2019.8765252).
- [51]. A. Zubiaga, D. Spina, R. Martínez and V. Fresno, "Real-time classification of Twitter trends", *Journal of the Association for Information Science and Technology*, vol. 66, no. 3, pp. 462-473, 2014. Available: [10.1002/asi.23186](https://doi.org/10.1002/asi.23186).
- [52]. M. Sayyadiharikandeh, O. Varol, K. Yang, A. Flammini and F. Menczer, "Detection of Novel Social Bots by Ensembles of Specialized Classifiers", *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020. Available: [10.1145/3340531.3412698](https://doi.org/10.1145/3340531.3412698).
- [53]. New York (N.Y.). Taxi And Limousine Commission, "New York City Taxi Trip Data, 2009-2018 (Version v1) [Data set]". Inter-University Consortium for Political and Social Research, 2019. Available: <https://doi.org/10.3886/ICPSR37254.V1>.
- [54]. U. Patel, "NYC Taxi Trip and Fare Data Analytics using BigData", Working paper, 2015. Available: https://www.researchgate.net/publication/287205718_NYC_Taxi_Trip_and_Fare_Data_Analytics_using_BigData.
- [55]. Y. Tang, "Big Data Analytics of Taxi Operations in New York City", *American Journal of Operations Research*, vol. 09, no. 04, pp. 192-199, 2019. Available: [10.4236/ajor.2019.94012](https://doi.org/10.4236/ajor.2019.94012).

- [56]. B. Alghuraybi, K. Marvaniya, G. Xia and J. Woo, "Analyze NYC Taxi Data Using Hive and Machine Learning", *International Journal of Database Theory and Application*, vol. 9, no. 6, pp. 191-198, 2016. Available: [10.14257/ijdta.2016.9.6.19](https://doi.org/10.14257/ijdta.2016.9.6.19).

Annex I. Supplementary materials

1. Twitter Stream Listener repository: <https://github.com/orestisf/botometer-checker-azure>
2. Twitter Stream Listener Docker image:
<https://hub.docker.com/repository/docker/ofotiadis/msciss>
3. Botometer Checker repository: <https://github.com/orestisf/botometer-checker-azure>
4. Botometer Checker Docker image:
<https://hub.docker.com/repository/docker/ofotiadis/botometer-checker-azure>
5. Twitter Trends Monitor repository: <https://github.com/orestisf/twitter-trends-monitor>
6. Twitter Trends Monitor Docker image: <https://hub.docker.com/r/ofotiadis/twitter-trends-monitor>
7. Azure components deployment templates repository: <https://github.com/orestisf/azure-arm-templates>
8. Sample dataset of tweets collected with the Twitter Stream Listener application:
<https://github.com/orestisf/twitter-stream-listener/blob/main/data-output-sample.json>
9. Sample dataset of trending topics collected with the Twitter Trends Monitor application:
<https://github.com/orestisf/twitter-trends-monitor/blob/main/data-output-sample.json>

Annex II. Development environment

The applications in the solution's Ingestion layer that are presented in have been developed with the Python programming language, version 3.9.1. Visual Studio Code was used as IDE, with the extensions listed in Table 22. For individual libraries that were utilized (primarily Tweepy to access the Twitter streaming API, Botometer to access the Botometer API and various Azure API libraries to interface with the different Azure components) and their versions, please refer to the respective `REQUIREMENTS.TXT` file in the application repositories listed in section 3.3.

All the Ingestion layer's applications were locally tested and debugged with Docker Desktop for Windows, version 3.2.0.

The ARM templates and CLI deployment scripts have been developed in Visual Studio code utilizing the Azure CLI Tools and the Azure Resource Manager extensions.

Extension Name	Description	Link
Azure Resource Manager Tools	Language server, editing tools and snippets for Azure Resource Manager (ARM) template files.	https://marketplace.visualstudio.com/items?itemName=msazurermttools.azurearm-vscode-tools

Azure Tools	Extension pack to interact with Azure resources.	https://marketplace.visualstudio.com/items?itemName=ms-vscode.vscode-node-azure-pack
Azure CLI Tools	Tools for developing and running commands of the Azure CLI.	https://marketplace.visualstudio.com/items?itemName=ms-vscode.azurecli
Pylance	Language server for Python in VS Code	https://marketplace.visualstudio.com/items?itemName=ms-python.vscode-pylance
Python	Linting, Debugging (multi-threaded, remote), Intellisense, Jupyter Notebooks, code formatting, refactoring, unit tests.	https://marketplace.visualstudio.com/items?itemName=ms-python.python
Docker	Build, manage, and deploy containerized applications from Visual Studio Code	https://marketplace.visualstudio.com/items?itemName=ms-azuretools.vscode-docker

Table 22: Visual Studio Code extensions used in the solution development.

Annex III. Solution deployment guide

This annex contains instructions to deploy the solution presented in this thesis into an Azure subscription. This instruction set assumes that the deployment takes place from scratch, in an empty subscription. However, the reader may use preexisting components (i.e. a Log analytics workspace) by appropriately modifying the template files.

Important note: Please refer to the latest readme file the repository where the scripts and templates are uploaded for the latest version of these instructions:
<https://github.com/orestisf/azure-arm-templates>.

Prerequisites

To replicate and use the solution presented in section Chapter 2 of this thesis, the following are required:

- An Azure subscription with adequate credits. The subscription's ID will have to be used in this section's scripts at the `--SUBSCRIPTION` parameter which is denoted by "XXXXXXXX-XXXX-XXXX-XXXXXXXXXXXXXXXX".
- The Azure principal that will execute the deployments must have at least `OWNER` role to the subscription's tenant.

- A Docker Hub account where the ingestion components will be uploaded. Alternatively
- Access to Azure CLI.
- A local copy (i.e., via `GIT CLONE`) of the ARM templates and the Azure CLI deployment scripts hosted in the GitHub repository: <https://github.com/orestisf/azure-arm-templates>.
- A copy of the Docker images for the ingestion components (Twitter stream listener and Botometer checker applications) from the following Docker Hub repositories, pushed to the user's Docker Hub account²⁹. These will be referenced when deploying the components as Azure Container Instances:
 - o <https://hub.docker.com/r/ofotiadis/msciss>
 - o <https://hub.docker.com/r/ofotiadis/botometer-checker-azure>.

In summary, the `AZ DEPLOYMENT SUB CREATE`³⁰ and `AZ DEPLOYMENT GROUP CREATE` CLI commands are used with the provided ARM templates to deploy each distinct component in the Azure subscription.

Deploying the components

The following paragraphs in this section provide instructions on deploying each individual component of the solution.

Creating the resource groups

Three resource groups will be deployed (rg-apps-alpha, rg-ops-alpha, rg-containers-alpha), each containing a separate set of the solution's resources as described in section 3.4. Note that this step is optional, and all resources may be deployed under the same resource group.

²⁹ Alternatively the docker images can be uploaded to an Azure Container Registry. To reduce costs, in this document we are not using one. The relevant documentation

³⁰ See also documentation: https://docs.microsoft.com/en-us/cli/azure/deployment/sub?view=azure-cli-latest#az_deployment_sub_create.

```
az deployment sub create --name resourceGroupOpsDeployment --
subscription XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX --location
"westeurope" `

--template-file "C:\path\to\file\resourceGroupOps.json" `

--parameters "C:\path\to\file\resourceGroupOps.parameters.json"
```

```
az deployment sub create --name resourceGroupAppsDeployment --
subscription XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX --location
"westeurope" `

--template-file "C:\path\to\file\resourceGroupApps.json" `

--parameters "C:\path\to\file\resourceGroupApps.parameters.json"
```

```
az deployment sub create --name resourceGroupContainersDeployment --
subscription XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX --location
"westeurope" `

--template-file "C:\path\to\file\resourceGroupContainers.json" `

--parameters "C:\path\to\file\resourceGroupContainers.parameters.json"
```

Deploy the Log Analytics workspace

A log analytics workspace will be provisioned, and its resulting workspace id will be referenced in subsequent deployments as discussed in **Error! Reference source not found.** so that they will be able to post metric and usage information to the Log analytics tables.

```
az deployment group create --name logAnalyticsWorkspaceDeployment --
subscription 3fb06c81-20d9-4c91-8fa1-dd7f2606ba21 --resource-group rg-
ops-alpha `

--template-file "C:\Users\ofoti\Dropbox\personal\University\UNUPI\MDA-
998 Dissertation\99. Working\logAnalyticsWorkspace.json" `

--parameters "C:\Users\ofoti\Dropbox\personal\University\UNUPI\MDA-998
Dissertation\99. Working\logAnalyticsWorkspace.parameters.json"
```

Deploy Application Insights resource

A single Application Insights resource will be provisioned, and its instrumentation key used to send telemetry data to Azure Monitor (see section **Error! Reference source not found.** for details on the Application Insights configuration) for the ingestion components of the solution.

```
az deployment group create --name applicationInsights --subscription
3fb06c81-20d9-4c91-8fa1-dd7f2606ba21 --resource-group rg-ops-alpha `
--template-file "C:\Users\ofoti\Dropbox\personal\University\UNIPPI\MDA-
998 Dissertation\99. Working\applicationInsights.json" `
--parameters "C:\Users\ofoti\Dropbox\personal\University\UNIPPI\MDA-998
Dissertation\99. Working\applicationInsights.parameters.json"
```

Deploy Event Hub namespace and event hubs

A single Event Hubs Namespace resource with three event hub instances will be provisioned. Each event hub will have 4 partitions and 3 consumer groups to allow for an equivalent number of consumer components (i.e. a stream analytics service or another processing service that reads events from the event hub).

```
az deployment group create --name eventHubDeployment --subscription
3fb06c81-20d9-4c91-8fa1-dd7f2606ba21 --resource-group rg-apps-alpha `
--template-file "C:\Users\ofoti\Dropbox\personal\University\UNIPPI\MDA-
998 Dissertation\99. Working\eventHub.json" `
--parameters "C:\Users\ofoti\Dropbox\personal\University\UNIPPI\MDA-998
Dissertation\99. Working\eventHub.parameters.json"
```

Deploy storage account and blob container

A single storage account with a blob container will be provisioned. The blob container will be used to store logs, as general purpose storage and for storing Event Hub consumer checkpoints.

```
az deployment group create --name storageAccountAndBlobContainer --
subscription 3fb06c81-20d9-4c91-8fa1-dd7f2606ba21 --resource-group rg-
apps-alpha `

--template-file "C:\Users\ofoti\Dropbox\personal\University\UNUPI\MDA-
998 Dissertation\99. Working\blobContainer.json" `

--parameters "C:\Users\ofoti\Dropbox\personal\University\UNUPI\MDA-998
Dissertation\99. Working\blobContainer.parameters.json"
```

Deploy Azure Container Instance for Twitter stream listener

The Twitter stream listener Python application is deployed as an Azure Container Instance, while the image itself that will be used is hosted in the user's Docker Hub repository. Before executing the deployment, environmental variables such as `EVENTHUB_ENDPOINT` MUST be configured in the `CONTAINERINSTANCE-TWITTERSTREAMLISTENER.PARAMETERS.JSON` file. Refer to section 3.3.1.2 for more information on the application's environmental variables.

```
az deployment group create --name ContainerInstanceTwitterStreamListener
--subscription 3fb06c81-20d9-4c91-8fa1-dd7f2606ba21 --resource-group
rg-apps-alpha `

--template-file "C:\Users\ofoti\Dropbox\personal\University\UNUPI\MDA-
998 Dissertation\99. Working\containerInstance-
TwitterStreamListener.json" `

--parameters "C:\Users\ofoti\Dropbox\personal\University\UNUPI\MDA-998
Dissertation\99. Working\containerInstance-
TwitterStreamListener.parameters.json"
```

Deploy Azure Container Instance for Twitter Trends Monitor

The Twitter Trends Monitor Python application is deployed as an Azure Container Instance, while the image itself that will be used is hosted in the user's Docker Hub repository. Before executing the deployment, environmental variables such as `EVENTHUB_ENDPOINT` MUST be configured in the `CONTAINERINSTANCE-TWITTERTRENDSMONITOR.PARAMETERS.JSON` file. Refer to section 3.3.2.1 for more information on the application's environmental variables.

```
az deployment group create --name ContainerInstanceTwitterStreamListener
--subscription 3fb06c81-20d9-4c91-8fa1-dd7f2606ba21 --resource-group
rg-containers-alpha `

--template-file "C:\Users\ofoti\Dropbox\personal\University\UNUPI\MDA-
998 Dissertation\99. Working\containerInstance-
TwitterStreamListener.json" `

--parameters "C:\Users\ofoti\Dropbox\personal\University\UNUPI\MDA-998
Dissertation\99. Working\containerInstance-
TwitterStreamListener.parameters.json"
```

Cleaning up the environment

To tear down the environment, it is enough to delete³¹ the three provisioned resource groups which contain all the solution's deployments, by running the following Azure CLI commands. Note that this operation will also delete any data ingested and persisted the Cosmos DB containers. You may export locally in JSON format any container using the Cosmos DB migration tool³². Alternatively, any data that have been persisted in blobs (i.e. in the storage account presented in 3.5.1) can be downloaded locally using the Azure Storage explorer tool³³.

```
az group delete --name rg-ops-alpha --yes
az group delete --name rg-apps-alpha --yes
az group delete --name rg-containers-alpha --yes
```

³¹ See also the documentation on the `az group delete` CLI command: https://docs.microsoft.com/en-us/cli/azure/group?view=azure-cli-latest#az_group_delete.

³² Executable available here: <https://docs.microsoft.com/en-us/azure/cosmos-db/import-data#Install>.

³³ See the following link for the Azure Storage Explorer: <https://azure.microsoft.com/en-us/features/storage-explorer/>.

Annex IV. Sample ingestion data

This annex presents sample responses from the Twitter API and a sample of the dataset used to evaluate the solution's performance.

Twitter Streaming API, sample response for hashtag #tesla

```
{
  "created_at": "Wed Feb 10 11:04:59 +0000 2021",
  "id": 1359458334215196673,
  "id_str": "1359458334215196673",
  "text": "\ud83d\udcb8 #tesla COMPRA #bitcoin ! #Ethereum est\u00e1 en graves PROBLEMAS \ud83d\ude30| CriptoNoticias #22 https://t.co/Ek2HUVVio5 via @YouTube",
  "source": "<a href=\"http://twitter.com/download/iphone\" rel=\"nofollow\">Twitter for iPhone</a>",
  "truncated": false,
  "in_reply_to_status_id": null,
  "in_reply_to_status_id_str": null,
  "in_reply_to_user_id": null,
  "in_reply_to_user_id_str": null,
  "in_reply_to_screen_name": null,
}
```

```
"user": {
  "id": 797800235078709249,
  "id_str": "797800235078709249",
  "name": "Guillem Ferrer",
  "screen_name": "GuillemFerrer3",
  "location": "Barcelona",
  "url": "https://youtube.com/c/GuillemFerrer",
  "description": "\ud83c\udd95 Creador de contenido sobre #blockchain y #criptomonedas \u26d3Apasionado del #bitcoin \ud83d\udcf0Not\u00e9dicias relevantes sobre el mundo cripto",
  "translator_type": "none",
  "protected": false,
  "verified": false,
  "followers_count": 140,
  "friends_count": 88,
  "listed_count": 2,
  "favourites_count": 324,
  "statuses_count": 252,
  "created_at": "Sun Nov 13 13:56:13 +0000 2016",
  "utc_offset": null,
  "time_zone": null,
  "geo_enabled": true,
  "lang": null,
  "contributors_enabled": false,
  "is_translator": false,
  "profile_background_color": "F5F8FA",
  "profile_background_image_url": "",
  "profile_background_image_url_https": "",
```

```
"profile_background_tile": false,

"profile_link_color": "1DA1F2",

"profile_sidebar_border_color": "C0DEED",

"profile_sidebar_fill_color": "DDEEF6",

"profile_text_color": "333333",

"profile_use_background_image": true,

"profile_image_url":
"http://pbs.twimg.com/profile_images/1327352799294840836/MV6DA0oq_normal.jpg",

"profile_image_url_https":
"https://pbs.twimg.com/profile_images/1327352799294840836/MV6DA0oq_normal.jpg",

"profile_banner_url":
"https://pbs.twimg.com/profile_banners/797800235078709249/1595247391",

"default_profile": true,

"default_profile_image": false,

"following": null,

"follow_request_sent": null,

"notifications": null

},

"geo": null,

"coordinates": null,

"place": null,

"contributors": null,

"is_quote_status": false,

"quote_count": 0,

"reply_count": 0,

"retweet_count": 0,

"favorite_count": 0,

"entities": {
```

```
"hashtags": [  
  {  
    "text": "tesla",  
    "indices": [  
      2,  
      8  
    ]  
  },  
  {  
    "text": "bitcoin",  
    "indices": [  
      16,  
      24  
    ]  
  },  
  {  
    "text": "Ethereum",  
    "indices": [  
      27,  
      36  
    ]  
  }  
],  
"urls": [  
  {  
    "url": "https://t.co/Ek2HUVVio5",
```

```
    "expanded_url": "https://youtu.be/c7vXGLvx0J0",
    "display_url": "youtu.be/c7vXGLvx0J0",
    "indices": [
      84,
      107
    ]
  }
],
"user_mentions": [
  {
    "screen_name": "YouTube",
    "name": "YouTube",
    "id": 10228272,
    "id_str": "10228272",
    "indices": [
      112,
      120
    ]
  }
],
"symbols": [],
},
"favorited": false,
"retweeted": false,
"possibly_sensitive": false,
"filter_level": "low",
```

```
"lang": "es",  
  
"timestamp_ms": "1612955099048"  
  
}
```

Twitter GET trends/place API endpoint sample response for location “New York”

```
[
  {
    "trends": [
      {
        "name": "Rose",
        "url": "http://twitter.com/search?q=Rose",
        "promoted_content": "None",
        "query": "Rose",
        "tweet_volume": 403824
      },
      {
        "name": "Britney",
        "url": "http://twitter.com/search?q=Britney",
        "promoted_content": "None",
        "query": "Britney",
        "tweet_volume": 169848
      },
      {
        "name": "Randle",
        "url": "http://twitter.com/search?q=Randle",
        "promoted_content": "None",
        "query": "Randle",
        "tweet_volume": "None"
      }
    ]
  }
]
```

```
    },
    {
      "name": "#SwitchtoDUBCHAENG",
      "url": "http://twitter.com/search?q=%23SwitchtoDUBCHAENG",
      "promoted_content": "None",
      "query": "%23SwitchtoDUBCHAENG",
      "tweet_volume": 142932
    },
    {
      "name": "Lauren London",
      "url": "http://twitter.com/search?q=%22Lauren+London%22",
      "promoted_content": "None",
      "query": "%22Lauren+London%22",
      "tweet_volume": "None"
    },
    {
      "name": "#뚝챙으로_바꾸자",
      "url": "http://twitter.com/search?q=%23EB%91%A1%EC%B1%99%EC%9C%BC%EB%A1%9C_%EB%B0%94%EA%BE%B8%EC%9E%90",
      "promoted_content": "None",
      "query": "%23EB%91%A1%EC%B1%99%EC%9C%BC%EB%A1%9C_%EB%B0%94%EA%BE%B8%EC%9E%90",
      "tweet_volume": 122213
    },
    {
      "name": "DUBCHAENG MELODY PROJECT",
      "url": "http://twitter.com/search?q=%22DUBCHAENG+MELODY+PROJECT%22",
```



```
"promoted_content": "None",  
"query": "%22DUBCHAENG+MELODY+PROJECT%22",  
"tweet_volume": 99107  
},  
{  
  "name": "alex quackity",  
  "url": "http://twitter.com/search?q=%22alex+quackity%22",  
  "promoted_content": "None",  
  "query": "%22alex+quackity%22",  
  "tweet_volume": "None"  
},  
{  
  "name": "Donovan Mitchell",  
  "url": "http://twitter.com/search?q=%22Donovan+Mitchell%22",  
  "promoted_content": "None",  
  "query": "%22Donovan+Mitchell%22",  
  "tweet_volume": "None"  
},  
{  
  "name": "Karl",  
  "url": "http://twitter.com/search?q=Karl",  
  "promoted_content": "None",  
  "query": "Karl",  
  "tweet_volume": 73500  
},  
{
```

```
"name": "#DedicatedToDUBCHAENG",
"url": "http://twitter.com/search?q=%23DedicatedToDUBCHAENG",
"promoted_content": "None",
"query": "%23DedicatedToDUBCHAENG",
"tweet_volume": 25238
},
{
  "name": "dahyun",
  "url": "http://twitter.com/search?q=dahyun",
  "promoted_content": "None",
  "query": "dahyun",
  "tweet_volume": 83944
},
{
  "name": "Kemba",
  "url": "http://twitter.com/search?q=Kemba",
  "promoted_content": "None",
  "query": "Kemba",
  "tweet_volume": "None"
},
{
  "name": "Mark Cuban",
  "url": "http://twitter.com/search?q=%22Mark+Cuban%22",
  "promoted_content": "None",
  "query": "%22Mark+Cuban%22",
  "tweet_volume": 27260
}
```

```
  },
  {
    "name": "Jazz",
    "url": "http://twitter.com/search?q=Jazz",
    "promoted_content": "None",
    "query": "Jazz",
    "tweet_volume": 43382
  },
  {
    "name": "#njnbg",
    "url": "http://twitter.com/search?q=%23njnbg",
    "promoted_content": "None",
    "query": "%23njnbg",
    "tweet_volume": "None"
  },
  {
    "name": "#SAPNAP",
    "url": "http://twitter.com/search?q=%23SAPNAP",
    "promoted_content": "None",
    "query": "%23SAPNAP",
    "tweet_volume": "None"
  },
  {
    "name": "Aunt Jemima",
    "url": "http://twitter.com/search?q=%22Aunt+Jemima%22",
    "promoted_content": "None",
```

```
"query": "%22Aunt+Jemima%22",
"tweet_volume": 24748
},
{
"name": "Wind Waker",
"url": "http://twitter.com/search?q=%22Wind+Waker%22",
"promoted_content": "None",
"query": "%22Wind+Waker%22",
"tweet_volume": "None"
},
{
"name": "Infinity Train",
"url": "http://twitter.com/search?q=%22Infinity+Train%22",
"promoted_content": "None",
"query": "%22Infinity+Train%22",
"tweet_volume": 11549
},
{
"name": "Tiafoe",
"url": "http://twitter.com/search?q=Tiafoe",
"promoted_content": "None",
"query": "Tiafoe",
"tweet_volume": "None"
},
{
"name": "Krispy Kreme",
```

```
"url":"http://twitter.com/search?q=%22Krispy+Kreme%22",
  "promoted_content":"None",
  "query":"%22Krispy+Kreme%22",
  "tweet_volume":"None"
},
{
  "name":"chris beard",
  "url":"http://twitter.com/search?q=%22chris+beard%22",
  "promoted_content":"None",
  "query":"%22chris+beard%22",
  "tweet_volume":"None"
},
{
  "name":"BOTW",
  "url":"http://twitter.com/search?q=BOTW",
  "promoted_content":"None",
  "query":"BOTW",
  "tweet_volume":"None"
},
{
  "name":"mark lee",
  "url":"http://twitter.com/search?q=%22mark+lee%22",
  "promoted_content":"None",
  "query":"%22mark+lee%22",
  "tweet_volume":39094
},
```

```
{
  "name": "Zelda",
  "url": "http://twitter.com/search?q=Zelda",
  "promoted_content": "None",
  "query": "Zelda",
  "tweet_volume": 22744
},
{
  "name": "Gobert",
  "url": "http://twitter.com/search?q=Gobert",
  "promoted_content": "None",
  "query": "Gobert",
  "tweet_volume": "None"
},
{
  "name": "Kiely",
  "url": "http://twitter.com/search?q=Kiely",
  "promoted_content": "None",
  "query": "Kiely",
  "tweet_volume": "None"
},
{
  "name": "Link to the Past",
  "url": "http://twitter.com/search?q=%22Link+to+the+Past%22",
  "promoted_content": "None",
  "query": "%22Link+to+the+Past%22",
```

```
"tweet_volume": "None"
},
{
  "name": "Venus Williams",
  "url": "http://twitter.com/search?q=%22Venus+Williams%22",
  "promoted_content": "None",
  "query": "%22Venus+Williams%22",
  "tweet_volume": "None"
},
{
  "name": "Jaylen Brown",
  "url": "http://twitter.com/search?q=%22Jaylen+Brown%22",
  "promoted_content": "None",
  "query": "%22Jaylen+Brown%22",
  "tweet_volume": "None"
},
{
  "name": "Pearl Milling Company",
  "url": "http://twitter.com/search?q=%22Pearl+Milling+Company%22",
  "promoted_content": "None",
  "query": "%22Pearl+Milling+Company%22",
  "tweet_volume": 11710
},
{
  "name": "Ocarina of Time",
  "url": "http://twitter.com/search?q=%22Ocarina+of+Time%22",
```

```
    "promoted_content": "None",
    "query": "%22Ocarina+of+Time%22",
    "tweet_volume": "None"
  },
  {
    "name": "Breath of the Wild",
    "url": "http://twitter.com/search?q=%22Breath+of+the+Wild%22",
    "promoted_content": "None",
    "query": "%22Breath+of+the+Wild%22",
    "tweet_volume": "None"
  },
  {
    "name": "Theis",
    "url": "http://twitter.com/search?q=Theis",
    "promoted_content": "None",
    "query": "Theis",
    "tweet_volume": "None"
  },
  {
    "name": "Djokovic",
    "url": "http://twitter.com/search?q=Djokovic",
    "promoted_content": "None",
    "query": "Djokovic",
    "tweet_volume": "None"
  },
  {
```



```
"name": "Ponce",
"url": "http://twitter.com/search?q=Ponce",
"promoted_content": "None",
"query": "Ponce",
"tweet_volume": "None"
},
{
  "name": "LET ME IN PLEASE",
"url": "http://twitter.com/search?q=%22LET+ME+IN+PLEASE%22",
"promoted_content": "None",
"query": "%22LET+ME+IN+PLEASE%22",
"tweet_volume": 11071
},
{
  "name": "YOU TOOK SO LONG",
"url": "http://twitter.com/search?q=%22YOU+TOOK+SO+LONG%22",
"promoted_content": "None",
"query": "%22YOU+TOOK+SO+LONG%22",
"tweet_volume": "None"
},
{
  "name": "Joe Ingles",
"url": "http://twitter.com/search?q=%22Joe+Ingles%22",
"promoted_content": "None",
"query": "%22Joe+Ingles%22",
"tweet_volume": "None"
```

```
  },  
  {  
    "name": "Caren",  
    "url": "http://twitter.com/search?q=Caren",  
    "promoted_content": "None",  
    "query": "Caren",  
    "tweet_volume": "None"  
  },  
  {  
    "name": "Bagley",  
    "url": "http://twitter.com/search?q=Bagley",  
    "promoted_content": "None",  
    "query": "Bagley",  
    "tweet_volume": "None"  
  },  
  {  
    "name": "Lori Harvey",  
    "url": "http://twitter.com/search?q=%22Lori+Harvey%22",  
    "promoted_content": "None",  
    "query": "%22Lori+Harvey%22",  
    "tweet_volume": "None"  
  },  
  {  
    "name": "My Valentine's Day",  
    "url": "http://twitter.com/search?q=%22My+Valentine%27s+Day%22",  
    "promoted_content": "None",
```

```
"query": "%22My+Valentine%27s+Day%22",
"tweet_volume": 36362
},
{
  "name": "sharks",
  "url": "http://twitter.com/search?q=sharks",
  "promoted_content": "None",
  "query": "sharks",
  "tweet_volume": "None"
},
{
  "name": "Spida",
  "url": "http://twitter.com/search?q=Spida",
  "promoted_content": "None",
  "query": "Spida",
  "tweet_volume": "None"
},
{
  "name": "Pistons",
  "url": "http://twitter.com/search?q=Pistons",
  "promoted_content": "None",
  "query": "Pistons",
  "tweet_volume": 14511
},
{
  "name": "Grant Williams",
```

```
    "url": "http://twitter.com/search?q=%22Grant+Williams%22",
    "promoted_content": "None",
    "query": "%22Grant+Williams%22",
    "tweet_volume": "None"
  },
  {
    "name": "Lowell",
    "url": "http://twitter.com/search?q=Lowell",
    "promoted_content": "None",
    "query": "Lowell",
    "tweet_volume": "None"
  },
  {
    "name": "The Celtics",
    "url": "http://twitter.com/search?q=%22The+Celtics%22",
    "promoted_content": "None",
    "query": "%22The+Celtics%22",
    "tweet_volume": 10960
  }
],
"as_of": "2021-02-10T09:58:50Z",
"created_at": "2021-02-08T21:22:21Z",
"locations": [
  {
    "name": "New York",
    "woeid": 2459115
  }
]
```

```
    }  
  ]  
}  
]
```

Botometer API sample response for Twitter account

```
{
  "id": "1154103442362785797",
  "user_id": "1154103442362785797",
  "user_screen_name": "ierodidaskalos",
  "user_majority_lang": "el",
  "cap": {
    "english": 0.8748575042966269,
    "universal": 0.6608500314332488
  },
  "display_scores_english": {
    "astroturf": 2,
    "fake_follower": 1.2,
    "financial": 0.1,
    "other": 4.6,
    "overall": 4.6,
    "self_declared": 0,
    "spammer": 0
  },
  "display_scores_universal": {
    "astroturf": 1.8,
    "fake_follower": 0.4,
    "financial": 0,
    "other": 2.4,
    "overall": 0.8,
    "self_declared": 0,
  }
}
```

```
    "spammer": 0.1
  },
  "raw_scores_english": {
    "astroturf": 0.4,
    "fake_follower": 0.24,
    "financial": 0.02,
    "other": 0.91,
    "overall": 0.91,
    "self_declared": 0.01,
    "spammer": 0
  },
  "raw_scores_universal": {
    "astroturf": 0.35,
    "fake_follower": 0.08,
    "financial": 0,
    "other": 0.48,
    "overall": 0.17,
    "self_declared": 0.01,
    "spammer": 0.02
  },
  "last_checked": "30/06/2021 13:10:03",
  "_rid": "dFFiA0ExCicWAAAAAAAAAA==",
  "_self": "dbs/dFFiAA==/colls/dFFiA0ExCic=/docs/dFFiA0ExCicWAAAAAAAAAA==/",
  "_etag": "\"04008c03-0000-0d00-0000-60dc6d2b0000\"",
  "_attachments": "attachments/",
  "_ts": 1625058603
```

}

NYC Taxi & Limousine Commission - green taxi trip records dataset, sample data

vendorID	lpepPickupDate	lpepDropoffDate	passengerCount	tripDistance	pickupLocationID	dropoffLocationID	rateCodeID	storeAndFwdFlag	paymentType	fareAmount	extra	mtaTax	improvementSurcharge	tipAmount	tollsAmount	totalAmount	tripType	puYear	puMonth
2	2018-06-05 18:00:51	2018-06-05 18:12:26	1	1.89	75	41	1	N	1	9.5	1	0.5	0.3	0	0	11.3	1	2018	6
2	2018-06-05 13:55:47	2018-06-05 14:05:28	1	0.94	41	74	1	N	2	7.5	0	0.5	0.3	0	0	8.3	1	2018	6
1	2018-06-05 17:46:27	2018-06-05 17:51:08	2	0.6	41	152	1	N	1	5	1	0.5	0.3	4	0	10.8	1	2018	6
1	2018-06-05 16:21:20	2018-06-05 16:44:10	2	1.9	255	37	1	N	2	14.5	1	0.5	0.3	0	0	16.3	1	2018	6
2	2018-06-05 09:30:38	2018-06-05 09:48:49	1	2.65	166	263	1	N	2	14	0	0.5	0.3	0	0	14.8	1	2018	6
2	2018-06-05 22:29:31	2018-06-05 22:35:41	1	1.11	260	7	1	N	1	6.5	0.5	0.5	0.3	2.34	0	10.14	1	2018	6
1	2018-06-05 08:24:56	2018-06-05 08:48:34	1	5.2	7	36	1	N	1	19	0	0.5	0.3	0	0	19.8	1	2018	6
2	2018-06-05 11:03:20	2018-06-05 11:21:29	1	2.14	17	17	1	N	1	12.5	0	0.5	0.3	0	0	13.3	1	2018	6
2	2018-06-05 14:23:23	2018-06-05 14:28:16	1	1.22	74	263	1	N	2	6	0	0.5	0.3	0	0	6.8	1	2018	6
2	2018-06-05 19:12:03	2018-06-05 19:15:01	1	0.68	75	74	1	N	2	4.5	1	0.5	0.3	0	0	6.3	1	2018	6

Table 23: NYC Taxi & Limousine Commission sample dataset