// Stream processing:
an overview of tools and cloud providers

//consult.red
Connected Devices & Systems

// Contents

> In this paper, we explain the typical characteristics of stream processing solutions and outline the key capabilities that the technologies need to provide

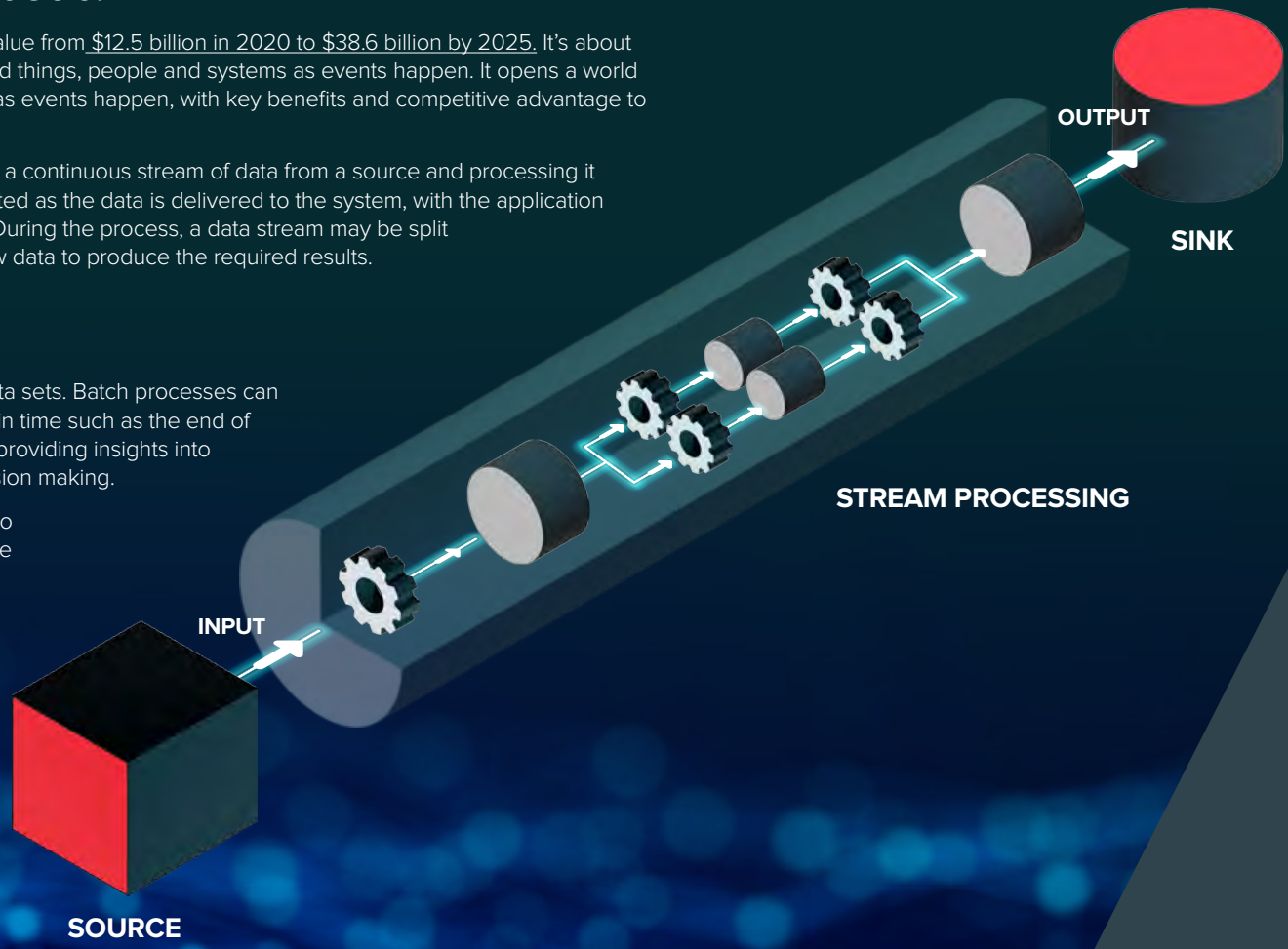
// Summary

Stream processing technologies are critical to organisations that need big data systems with near real-time analysis and decision-making capabilities. In this paper, we explain the typical characteristics of stream processing solutions and outline the key capabilities that the technologies need to provide. We describe key tools, frameworks and platforms currently available to support real-time processing requirements and identify some of the factors to address when considering a stream processing development.

// What is stream processing all about?

Stream processing is an exploding market – expected to increase in value from $12.5 billion in 2020 to $38.6 billion by 2025. It's about analysing data in motion, processing the data that flows from connected things, people and systems as events happen. It opens a world of possibilities for use cases that require decision-making and actions as events happen, with key benefits and competitive advantage to enterprises who adopt it.

The illustration below shows a stream processing application receiving a continuous stream of data from a source and processing it to provide real-time, actionable insights. The computations are conducted as the data is delivered to the system, with the application making highly efficient calculations and delivering outputs to the sink. During the process, a data stream may be split into separate streams, joined with other streams and enriched with new data to produce the required results.



// Stream processing vs batch processing

Unlike stream processing, batch processing acts on finite, bounded data sets. Batch processes can run when a complete data set has been received, or at defined points in time such as the end of a day or week. This type of processing is useful for complex analytics, providing insights into historical data, identifying trends, and generally supports reactive decision making.

Compared to batch processing stream processing is generally harder to implement. But has the advantage that a stream processing architecture can also be used for batch processing applications to gain additional value from the same data. The same isn't true the other way round – you can't take data collected for batch processing and later apply stream processing to it.

// What are the benefits of stream processing?

The power of stream processing is in enabling near real-time responses to events that are taking place. For instance, in fraud prevention it's no good informing a customer at the end of the month or week that you've detected a suspicious transaction. The transactions need to be processed in a stream and actionable insights delivered as quickly as possible. Stream processing opens up the potential for unlimited business use cases, from fraud prevention to computer gaming to real-time logistics routing. By developing a stream processing solution to meet one business case, opportunities unfold to find new ways of using the data – gaining even more value from the investment. Enterprises can use stream processing for competitive advantage with benefits that include:

✓ **Operational efficiencies** ✓ **Improved customer experiences** ✓ **Faster and better decision-making** ✓ **Support for integration and digital transformation**

This is illustrated in the use cases listed in the table below.

Table 1: Typical stream processing use cases

Industry/market	Data streams	Benefits
Banking, finance	User activity, banking transactions	Fraud detection by detecting anomalies and identifying suspicious patterns
Cybersecurity	Network events, user activity, suspicious volumes of traffic from a single IP address or user account	Identify security breaches, router attacks, compromised accounts, potential DDOS and ransomware attacks
Infrastructure & applications	Network logs, user activity	Monitor system and application performance. Detect outages and failures. Predictive analytics for infrastructure failures
eCommerce	User activity on websites and social media platforms, browsing history, clicks	Targeted advertising, real-time pricing strategies, upselling and cross-selling Enhanced customer experience online and in store
Healthcare	Sensor data from patient monitoring equipment and wearables, connected devices in homes	Fast response to changes in patient conditions Predictive analytics for early warning of patient deterioration More independence for elderly and vulnerable people who can stay in their homes
Environmental	Sensor data such as temperature, carbon monoxide, footfall, desk and meeting room usage, CCTV	Improved health and safety measures and productivity in workplaces Covid-secure practices, such as social distancing Intruder detection
Logistics and transportation	Sensor data from vehicle trackers and other devices	More efficient and responsive journey planning Improved cold chain management Predictive analytics for vehicle maintenance
Industry and manufacturing	Sensor data from equipment and machinery	Improved safety, security, and operational efficiency of processes and plants Preventative and predictive maintenance of equipment and machinery
Emergency services	Sensor data from vehicle trackers, traffic and weather conditions	Faster times for emergency vehicles to reach destinations Better responses on the ground, such as detecting changes in wind direction when firefighters are tackling a fire
Smart cities	Sensor data from connected devices	More efficient management of assets, resources and services, including utilities, water supplies, waste, community services Traffic monitoring, congestion detection, accident detection Predictive analytics for early warning of asset or service failures
Connected vehicles	Sensor data from connected devices	Enabling autonomous vehicles Improved safety by supporting drivers and operators Preventative and predictive maintenance of vehicle parts
Sports	Sensor data from trackers, cameras and other devices	Real-time updates to leader boards, statistics, commentary, safety information

The technologies for capturing, processing, and storing real-time data streams are advancing and maturing.
Understanding the options and choosing the best technologies for your business case is key to success with stream processing.

// Go Green – an example use case

The paper uses an example use case 'Go Green' to illustrate key points about stream processing.

A logistics company owns a fleet of vehicles for transportation. To meet sustainability requirements, they must provide information about the carbon footprint associated with the products they transport. With this information, the company can:

- Monitor emissions from the fleet to detect whether they're likely to breach their emissions limit
- Implement predictive analytics to investigate rises in emissions, which might indicate that vehicle components need maintenance
- Enable their customers to use sustainability information when making purchasing decisions

The company plans to implement a stream processing application – which they're calling Go Green – to process real-time data from the vehicles.
To calculate the quantity of greenhouse gas emissions produced when transporting a product, the application needs the following data for each vehicle:

- Waypoints on the journey
- Fuel level at each waypoint
- Weight of cargo

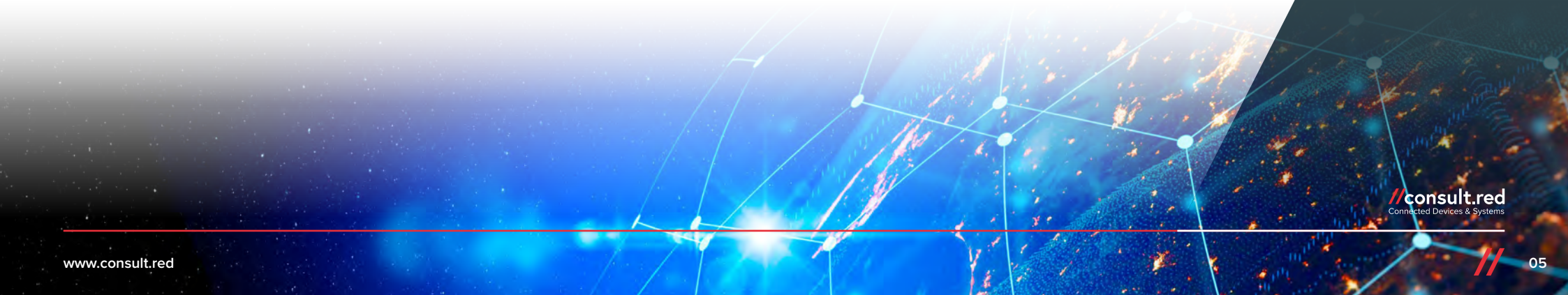
By calculating greenhouse gas emissions at multiple points of the journey, the application can monitor the total emissions for the fleet, perform predictive analytics on breaching the emissions limit and vehicle maintenance, and compute the carbon footprint for each product transported.

To make these calculations, the company will equip all vehicles with the following sensors:

- Fuel-level sensor to provide accurate fuel level measurements
- Load weight sensor on each axle to measure the weight of the load at different points of the vehicle
- GNSS sensor to provide current geographic coordinates of the vehicle

The company identifies additional benefits that they can realise from the stream processing implementation in the future:

- Trigger alerts if vehicles are overloaded, or the cargo is not distributed evenly in the vehicle
- Use vehicle locations to manage routing and journey times in response to events such as vehicle breakdowns, heavy traffic or road closures
- Detect theft of vehicles and provide vehicle location data to law enforcement agencies
- Monitor the fuel economy of different drivers by recording driver data with the journey information
- Detect and react to potential driving infringements in real-time



// Characteristics of stream processing

Unbounded data streams

A data stream is unbounded – there's no recognisable beginning or end to the data. Data records are often very small, but the volume of records can be extremely high. For example, a deployment of thousands of sensors might report readings every few seconds. Other use cases have varying data distributions, with quiet periods interspersed with peaks. For example, tracking devices in vehicles might report in large numbers during rush hours while few might send data in the middle of the night.

Sometimes more than one task must be performed on data streams to achieve the required outputs. The nature of the tasks determines whether they can be performed in sequence (serially) or simultaneously (in parallel).

// Go Green – data streams

When a vehicle is in operation, the sensors in the vehicle deliver a continuous flow of measurements of fuel level, load weight and vehicle position. These data streams are unbounded as there is no planned end for processing the data and no requirement to close a stream before processing. The measurements from each type of sensor form separate data streams to the application. The rate of flow of each stream will vary, with peaks of data during the day and quieter periods when few vehicles are in use.

Real-time processing

There are two ways in which stream processing applications process data:

- Real-time: processing individual events as they arrive at the system
- Micro-batching: collecting small batches of events before processing

Real-time processing individual events can be computationally inefficient as the overhead associated with transporting a single event in a packet or the scheduling of a task to process it can far outweigh the benefit. Micro-batching is a compromise whereby small batches of events are processed together to improve computational efficiency but the batch size is kept sufficiently small so latency goals are still met.

With batch windows of around 50 milliseconds, a micro-batching system can deliver near real-time results, but has a high resource overhead in the processing power required for managing the batches. If higher latency is acceptable, longer batch windows can be used with lower overheads. For use cases when it's critical that events or changes in the current state trigger immediate actions or alerts, real-time processing is required.

// Go Green – real-time processing

The Go Green stream processing application will use a real-time processing engine to generate accurate carbon footprint calculations for the fleet.

Stateful processing

Sometimes individual events in a data stream can be processed independently. In other cases, the application needs knowledge from previous events in order to process a new event. This is referred to as stateful processing. Stateful processing brings challenges:

- For processing speed, states need to be held in memory
- States must also be written to persistent storage for failovers to work
- With distributed systems, states need to be managed so that they don't have to be duplicated across nodes

// Go Green – stateful processing

The Go Green application needs to use stateful processing for two reasons:

- To calculate the fuel consumption, it needs to know the previous fuel level as well as the current level
- All the waypoints for each segment of a journey must be processed together and not distributed to different nodes

Scalability

As data volumes grow, the system processing resources need to increase to handle the load. Scalability is the ability of the stream processing system to adapt to changing data volumes. There are two ways of scaling stream processing systems:

- **Vertical scaling** (or scaling up) means increasing the size and power of the computer. This can include any compute resources, such as size and speed of disk, memory, CPUs, CPU cores etc. Eventually, the limit of compute resources will be reached or the costs will become unacceptable, and no further vertical scaling will be possible.
- **Horizontal scaling** (or scaling out) means adding more computer nodes to the system and distributing the workload among the nodes.

The choice of vertical or horizontal scaling depends upon the use case. Horizontal scaling provides the capability and flexibility to handle ever-increasing data flows. However, it requires careful management if stateful processing is used. States are held in memory for speed of processing but duplicating states across multiple nodes creates overheads and requires more resources. To avoid duplicating states, a mechanism such as partitioning can be used. Partitions are created and assigned to different compute nodes . Events are assigned to a partition based on a partition key logic to ensure that they are processed in the correct partition. Partitioning ensures that all the events requiring access to a particular state are processed by the same compute node. This minimises the need to duplicate state between compute nodes and enables the workload to be evenly distributed across the nodes.

// Go Green – scalability

Over time, the logistics company increases its vehicle fleet. If the application runs on a single computer, it will eventually become limited by the available compute power and amount of memory required to hold the state information for all the vehicles. A more flexible approach is to use horizontal scaling and distribute the processing across multiple compute nodes.

The Go Green application uses stateful processing, which means partitions are needed to avoid duplicating states across the nodes. Each vehicle is assigned to one partition, ensuring that all events for the vehicle are processed in one partition on one node.

> The Go Green stream processing application will use a real-time processing engine to generate accurate carbon footprint calculations for the fleet.

// Capabilities

Various stream processing tools, frameworks and libraries exist that can significantly expedite the development of stream processing systems, however, they offer a range of capabilities. Some of the key features are outlined below.

Data enrichment

A raw data stream may not be enough on its own to produce the required outputs. Instead, the data must be enriched with other sources of information. These might belong to the enterprise, such as details of customers, or be acquired from external sources, such as weather and traffic data.

One challenge of enriching data streams is the delay this can add to processing time. To minimise delays, optimization techniques often supported:

- Asynchronous I/O – the processor handles requests and responses concurrently, thereby reducing the overall processing time
- Caching – frequently used data can be stored in cache to reduce the read access time
- Distributed joins – data from an external source can be transformed into another stream before joining with the main data stream to reduce processing time

// Go Green – data enrichment

The logistics company wants to enrich the vehicle location data with map matching. Snapping the path taken by a vehicle to the road infrastructure held by a mapping tool provides more accurate information about vehicle location and journey details. To implement map matching, the Go Green application can use an external service such as the HERE Maps API or GoogleMaps. For optimised speed, the application uses asynchronous I/O to process events while waiting for a response from the external API.

Data enrichment can also be used to add more value to the application:

- With information about the maximum weight load for the vehicles, the application can raise alerts about overloading. As this type of data is relatively static, it can be held in cache.
- To analyse the fuel economy of the drivers, the application needs to maintain the current assignment between driver and vehicle. This can be done in two ways:
  - Enrich the fuel data stream with driver assignment data
  - Build a new data stream of driver assignments and join it with the fuel stream

Time windows

Windowing is an approach that breaks an unbounded data stream into segments for processing. Windows are a view on a stream, with the events in the window kept in memory and available for fast processing. Different types of windows may be supported:

- **Tumbling windows** have a fixed size and don't overlap or have gaps. An example of tumbling windows is for measuring audience figures in 5-minute windows.
- **Hopping windows** have a fixed size but can overlap. Events can be assigned to more than one window. Hopping windows can be used for measuring trends in real time, for example, a 30-minute average of a channel audience measured every 5 minutes.
- **Sliding windows** have a fixed size and a sliding interval that determines when a new window is created. Events can be assigned to more than one window. Sliding windows are like hopping windows but they slide in real time. For example, an event could be triggered if 5 unsuccessful logins are detected during a 15-minute window.
- **Session windows** group events by periods of activity. They don't have fixed start or end times and don't overlap with other session windows. A session window closes after a period of inactivity, during which it doesn't receive any events. An example of session windows is to represent user mouse activity, when there may be long periods of idle time interspersed with high numbers of mouse clicks. If the idle time exceeds the inactivity period, the current window is closed and the next mouse click opens a new window.

Event timestamping is critical in stream processing as it determines the order in which events are processed and determines whether an event is still valid, particularly if it arrives late. If time windows are used, timestamps determine the window into which events are assigned.

Key timestamp types are:

- **Event generation –** the time that the event occurred. This is assigned by the source system that created or logged the event. Not all devices can create a timestamp and some that do may not have a precise clock onboard. Analysis of the business case is required to determine how to handle event generation timestamps and problems such as device clocks that are out of sync. If the source can't provide reliable timestamps, the stream processing system needs to generate timestamps when it receives events.
- **Event received –** the time the event was delivered. If the source doesn't have a clock or provides inaccurate timestamps, the event received timestamp must be used for ordering events.
- **Event processing –** the time that the event was processed. This timestamp might be relevant for the computation or to determine that further computation is not required, for example, if the event has expired. It also shows how long it took before the event was processed, which may be a useful input to analytics and performance measurements.

// Go Green – time windows

As route matching algorithms gain in accuracy with the increasing numbers of waypoints, the application can use time windows to gather waypoints. If there's a requirement to use map matching on the whole route of vehicles, session windows could be used to combine a series of waypoints in windows separated by periods of time when the vehicle is not moving.

Checkpointing

Checkpointing is a fault tolerance mechanism that handles system failures and restarts. It requires snapshots of the data stream metadata to be saved to persistent storage to enable the system to recreate the correct state if it needs to recover from failure. Fault tolerance becomes harder to achieve with the stateful calculations. Fault tolerance becomes harder to achieve with the stateful calculations as the system must have a means to recover the state values that were being held in memory when the node failed.

Alternatives to checkpointing include:

- Recreating the state from a point within the data stream. In some cases, data streams can be compacted, with the latest state or output used if the application needs to restart.
- Continue without recreating the state – in some cases, it's acceptable to create the state using new events.

The mechanism to use depends on the use case.

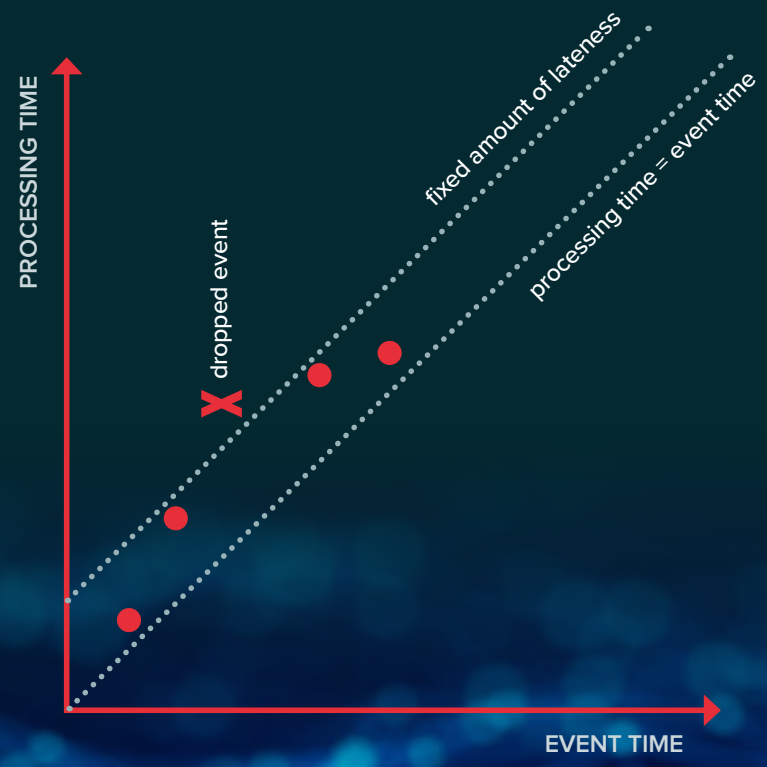
// Go Green – checkpointing

While a vehicle is travelling to its destination, the system has an open time window with several waypoints waiting to be sent to the map matching service. To avoid losing the states held in memory if a computing node fails, checkpointing is used. At regular, configurable intervals, the processing engine persists the state and a pointer to the specific place in the events stream.

This allows the system to recover to normal functioning of the pipeline after failure as the state is read from the persistent storage and processing is resumed from the appropriate position in the stream.

Watermarking

Watermarking is a process for handling late events. A watermark is a time threshold that defines how long the system waits for late events. If a late event arrives before the watermark, it is processed. If it arrives after the watermark, it is ignored.



Different tools offer different ways of setting watermarks. For example, Kafka Streams provides two approaches:

- Continuous refinement, which enables outputs to be updated when late events arrive
- Suppress operation with a configurable grace period

Continuous refinement has some disadvantages:

- Not all data sinks support updates to the outputs
- Some actions that were taken in response to outputs cannot be undone (for example, sending an email)
- It makes memory management unpredictable as there's no way of knowing whether late events will arrive

// Go Green – watermarking

If the sensors on the vehicles rely on cellular connectivity, it's possible that some messages may be delayed if signal reception fails in some areas. Onboard devices might buffer the messages and transmit them when they can reconnect. Analysis is required to determine how to handle late messages. A watermark can be configured to ensure that only late messages that are still useful are processed. Any others that fall outside this period will be discarded.

Watermarking is a process for handling late events. A watermark is a time threshold that defines how long the system waits for late events.

// Tools and environments

Stream processing with Apache Flink

Apache Flink is a complete distributed streaming engine written in Scala and Java. It incorporates a framework for creating data flow pipelines using different levels of abstraction:

- SQL – high-level language for analytics
- Table API – declarative domain-specific language (DSL)
- DataStream API – core APIs for stream and batch data processing (Java, Scala, Python)
- Stateful stream processing – low-level building block (Java, Scala)

Flink supports the following key requirements:

- Scalability – distributed system with horizontal scaling capability
- Fault tolerance – achieved with a checkpointing mechanism
- Speed – in-memory processing enables real-time performance
- Portability – runs in all common cluster environments

Stream processing with Hazelcast Jet

Hazelcast Jet is an in-memory, distributed stream processing engine written in Java and with a Java API to build pipelines in a dataflow programming model. It models computations as a network of tasks connected with one-way data pipes. The results of one task form the inputs to the next task in the form of a directed acyclic graph (DAG).

Hazelcast Jet supports the following key requirements:

- Scalability – distributed system with horizontal scaling capability
- Fault tolerance – achieved by using repeatable/acknowledging sources with a combination of distributed snapshots
- Speed – in-memory processing enables real-time performance

Stream processing with Apache Spark

Apache Spark is a high-speed, scalable, distributed processing system, capable of near real-time performance. It supports Java, Scala, Python, R and SQL languages and runs in most common cluster environments.

Spark uses micro-batching rather than event-based processing. The size of the batches determines the response speed – the smaller the batch window, the faster the speed.

Considerations for using Spark:

- The processing overhead for managing batches increases as batch window size decreases. A batch size of 50 milliseconds can deliver near real-time performance, but significant processing power will be required to manage batches of this size.
- The batch window size is fixed and autoscaling can't be used in response to data flow. If the system can experience wide variations in traffic flow, system resources need to be designed to cope with the processing power at the peaks, even if there will be quiet periods when these resources aren't required.
- A batch size of 2 seconds typically gives a reasonable balance between processing power required for business logic and that required for batch management. If this latency is acceptable, Spark may be a suitable tool to consider for a stream processing application.

Stream processing with Kafka Streams

Kafka Streams is a library for building streaming applications, designed for writing applications to transform and enrich data in Apache Kafka. It offers a straightforward approach to writing mission-critical, real-time applications and microservices while benefiting from Kafka's server-side cluster technology. It requires a platform to run Kafka Stream applications.

The library can be used in Java or Scala applications. The applications don't run inside the Kafka cluster, but they should be deployed as close as possible to the cluster to minimise the network latency. Kafka Streams supports the following key requirements:

- Fault tolerance – achieved by Kafka topic persistence, an offset committing mechanism and RocksDB with checkpointing
- Scalability – Kafka event log is provided; the scalability of workers/microservices must be considered when designing the architecture of client applications
- Exactly-once processing is supported
- Order of events can be guaranteed within a single partition

Stream processing with Apache Beam

Apache Beam is not a distributed stream processing engine but a framework that provides a unified programming model for Java, Python and Go. Applications built with Apache Beam can be executed in many runtime environments including:

- Apache Flink
- Apache Spark
- Apache Samza
- Google Dataflow

Stream processing with Spring Cloud Data Flow

Spring Cloud Data Flow is a set of frameworks and tools that facilitates building and deploying data pipelines that consist of Spring Boot applications:

- Long-lived applications can be implemented with the Spring Cloud Stream microservice framework
- Short-lived applications can be implemented using the Spring Cloud Task framework

Applications can be deployed to the Cloud Foundry platform or the cluster management service Kubernetes. Spring Cloud Data Flow represents a different architectural approach to the other platforms described in this section. Apache Flink and Google Dataflow applications run on a dedicated engine cluster, which gives them a richer environment for performing more complex calculations. This can be achieved with Spring Cloud Data Flow by using Kafka Streams applications with Apache Kafka.

// Support from cloud and third-party providers

This section describes the stream processing services provided by cloud and third-party providers, highlighting the key features that they provide. The stream processing frameworks described above can be hosted on almost any cloud infrastructure. However, using a ready-made stream processing service can be more expedient and offer greater flexibility by being easier to scale as demand grows. It may also be lower cost, especially at low demand. Stream processing with AWS Kinesis Data Analytics.

Stream processing with AWS Kinesis Data Analytics

AWS Kinesis Data Analytics is a service providing a runtime environment for Apache Flink Applications.

- No infrastructure/hardware provisioning – pure serverless solution
- Auto-scaling based on incoming traffic
- Costs are proportional to the processing power used
- Supports Java, Scala, Python and many well tested libraries for implementing processing tasks

Stream processing with Google Dataflow Prime

Google Dataflow Prime is a fully managed stream processing service that provides a runtime environment for Apache Beam Pipelines.

- No infrastructure/hardware provisioning – pure serverless solution
- Auto-scaling according to processing needs
- Costs are proportional to the processing power used
- Supports Java, Python, Go

Note: Google Dataflow Prime is not yet in General Availability

Stream processing with Azure Stream Analytics

Real-time analytics engine as service offered by Microsoft Azure.

- PaaS model – no hardware or infrastructure provisioning is required
- No autoscaling out-of-the box. Options are:
  - Manual scaling
  - Scaling based on schedule
  - Configurable triggering for scaling based on selected input data metrics
- Costs are proportional to allocated resources
- Provides SQL-like query language with some built-in functions and the possibility of using JavaScript or C# UDFs. No general-purpose language support is available, which limits use cases.

Stream processing with Confluent Cloud

Confluent Cloud is a fully managed Kafka service, accessible from AWS, Google Cloud and Microsoft Azure.

- No infrastructure to manage – Kafka as a Service
- Throughput based sizing for Kafka cluster, manual scaling ksqlDB with Confluent Streaming Units (CSU)
- Costs proportional to resources used for Kafka cluster, additional costs proportional to allocated resources when using ksqlDB
- ksqlDB or processors implemented with Kafka Streams API

Any complex stream processing which does not fit well into ksqlDB model requires external applications to be implemented. UDFs are not supported in Confluent Cloud. Customised processors are deployed outside of Kafka clusters and their nature, management costs, and scaling must be considered separately.

Table 2: Comparison of cloud provider and third-party stream processing products

	AWS Kinesis Data Analytics	Google DataFlow Prime	Azure Stream Analytics	Confluent Cloud
Environment management/provisioning	No effort – pure serverless	No effort – pure serverless	Dedicated cluster has to be provisioned and managed	Kafka as a Service, processing infrastructure must be considered separately
Autoscaling	No effort – full autoscaling	No effort – full autoscaling	No autoscaling. Possible to configure triggers (research on dependency between input data metrics and resources needed)	Kafka sized automatically, but processing infrastructure must be considered separately
Cost	Proportional to usage	Proportional to usage	Proportional to allocated resources	Proportional to Kafka resources used, but processing cost is not included
General purpose languages supported for processing logic	Java, Scala, Python	Java, Python, Go	No	Java, Scala
Cloud providers supported	AWS	GCP	Azure	AWS, GCP, Azure
Comment	Meets all key requirements for stream processing	Meets all key requirements for stream processing but not yet in General Availability	Not as advanced as AWS Kinesis and Google DataFlow Prime, but sufficient for analytical purposes if no complicated logic requiring a general-purpose language is required	Provides part of the solution but a platform to host the processing applications must be considered separately

// Choosing the right technology

With stream processing technology, there's no one size fits all approach that works for all use cases. Some of the key factors that need to be considered are described below.

Existing technical stack and experience

- Streaming platforms support various languages. It makes sense to choose a platform with a language that your team is experienced in using. For example, if your team uses Kafka, the choice of Kafka Streams might be the best fit.
- Cloud provider support is important.
- The use of Apache Beam to provide an abstraction layer is a sensible choice if you might move to a different runtime environment in the future.

Complexity of calculations

The first stream processing platforms used SQL tools to perform analytical tasks. This offered a better solution to real-time requirements than extract, transform, load (ETL) batch processing. Over time, pipelines were introduced to provide even better performance for stream processing. Implementing pipelines requires a modern, general-purpose language, with support from many proven libraries.

Currently Azure Stream Analytics only supports a Stream Analytics Query Language with a choice of built-in functions and the ability to extend them with User-Defined Functions (UDFs) written in C# or JavaScript.

Infrastructure

Both AWS Kinesis and Google Dataflow Prime offer maintenance-free solutions. They have a true serverless approach, with no infrastructure or hardware provisioning required. They both support horizontal auto-scaling to provide more pipeline processing power when it's needed with no management intervention.

Azure Stream Analytics offers two ways to deploy a pipeline:

- Assign jobs to a processing unit
- Use a cluster as a single-tenant deployment for complex and demanding use cases

Azure does not support autoscaling. Scaling can be implemented manually by setting up a schedule based on experience or by configuring custom triggers based on selected input data metrics.

Confluent Cloud provides part of the solution with Kafka. Although it manages the service and scales automatically, processing applications must be handled separately.

Flexibility

Apache Beam provides a future-proof approach. Pipelines can be implemented in Java, Python or Go and deployed in various runtime environments, including Samza, Flink, Spark and Dataflow. Beam applications are portable. For example, if the application was developed for deployment to AWS Kinesis Data Analytics with Apache Flink runtime, it can later be easily migrated to Google Cloud Dataflow Prime.

Developing a solution using Kafka, Kafka Streams and Kubernetes as a cluster for processing applications is more complex but can be deployed using cloud services from AWS, Microsoft or Google.

Velocity

The time it takes to achieve results from a stream processing application depends on the technology used. In complex cases where a query language doesn't meet the business logic requirements, AWS Kinesis Data Analytics or Google Dataflow Prime would be the choice. Azure Stream Analytics doesn't support this requirement and the environment setup required for application processing using Kafka Streams takes significant time.

//consult.red
Connected Devices & Systems

// Contacts

For more information on streaming processing or to discuss your requirements get in touch with one of our experienced consultants.

// Stuart Griffin

stuart.griffin@consult.red
+44 (0) 7869 422 971

// Rahul Mehra

rahul.mehra@consult.red
+44 (0) 7869 422 971
+44 (0) 1274 287 710

// Kamil Krupa

kamil.krupa@consult.red
+48 696 656 492