International Journal of Science and Research (IJSR)
ISSN: 2319-7064
SJIF (2019): 7.583

Autonomous Scheduling for Recurring Tasks to Manage Ingestion and Stream Processing

Mahidhar Mullapudi[1], Satish Kathiriya[2], Siva Karthik Devineni[3]

Abstract: ***The need for effective large-scale heterogeneous distributed data ingestion pipelines, crucial for transforming and processing data essential to advanced analytics and machine learning models, has seen a significant surge in importance. Modern services increasingly depend on near-real-time signals to precisely identify or predict customer behavior, sentiments, and anomalies, thereby facilitating informed, data-driven decision-making*[1]*.In the rapidly evolving landscape of large-scale enterprise data applications, the demand for efficient data ingestion and stream processing solutions has never been more critical*[2]*. This technical paper introduces a groundbreaking autonomous self-schedulable library designed for recurring jobs, addressing the challenges faced by enterprises in orchestrating complex data workflows seamlessly. Leveraging authoritative expertise in building robust enterprise applications, this library provides a paradigm shift in how organizations manage and execute recurring tasks within data pipelines*** **[3].**

Keywords: Modern Ingestion Platform, Autonomous Scheduling Library, Parallel Stream Processing.

1. Introduction

Real-time data ingestion and processing systems play a pivotal role in providing data to analytics and machine learning platforms, enabling the extraction of invaluable insights. At this scale the data is emitted from different data sources which would need processing power to run several jobs to ingest the data at different times. In this paper, we propose a robust library for self-scheduling recurring jobs to curate, process, and publish data to different downstream services depending on the use case. We propose a unique design that merges two different processing types – streaming and micro-batching depending on the amount of data at a certain time. Below are some of the key considerations while designing data intensive applications like these:

- **Ease-of-configurability:** Provide easy configuration options to efficiently manage data at scale, accommodating fluctuations in load and diverse data schemas.
- **Performance**: Evaluating acceptable latency levels, whether in seconds or minutes and determining the required throughput both per machine and in aggregate for each service.
- **Scalability**: Monitoring and addressing the scalability requirements of the system, considering data bursting at times and being able to handle the load [4].
- **Reliability**: Ensuring the system consistently performs its intended functions, verifying its capability to handle varying loads and data volumes without compromising functionality.
- **Fault-tolerance**: Identifying the types of failures the system can tolerate, establishing guaranteed semantics for the processing or output frequency of data, and detailing the storage and recovery mechanisms[4].

Within this paper, we meticulously detail the overarching architecture, present a comprehensive overview of the involved components, elucidate the system's design intricacies, and diligently adhere to industry’s best practices while integrating the considerations into our design decisions.

The structure of this paper is as follows. In Section 2, we provide a thorough examination of the system's architecture and delineate the intricate relationships among its integral components and takes deep dive into some of the key components, delves into the specifics within the system, shedding light on their roles, functionalities, and contributions to the overall framework.Section3 serves as a deep dive into the design and implementation phases of autonomous schedulable recurring library for data ingestion. Finally, in Section 4, we conclude by summarizing the impact of this well-designed library.

2. Systems Overview

Designing and building a library that can handle autonomous scheduling needs of data processing and ingestion pipelines require thorough understanding of intricacies that these large-scale data streaming services offer. In this section, we delve into a comprehensive overview of the architecture and diverse components integral to this library. Large-scale distributed data applications often face challenges related to unpredictable spikes in data traffic. To address this, a proactive approach is required to predict these spikes and choose the most suitable data processing strategy. Our proposed solution integrates intelligent

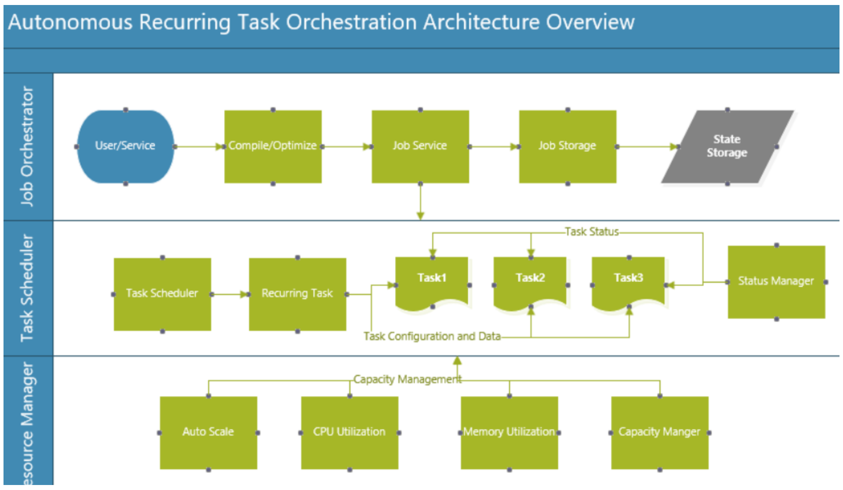

Figure 1: Autonomous Recurring Job Orchestrator Architecture Overview

Algorithms and adaptable processing techniques to enhance the overall efficiency of data applications. This paper mainly talks about the components below:

- Data sources
- Export/Publish Endpoints
- Autonomous Recurring Task scheduler
- Job Orchestrator[5]
- Resource Manager[6]

Illustrated in Figure 1 is the holistic architecture, depicting individual components and services. This depiction showcases the process of capturing data from diverse sources, ingesting and storing streaming data for real-time analytics, responding to events in real-time, and subsequently transporting the data across various processors based on specific use cases. Finally, the processed data is seamlessly fed into dependent systems for advanced data analytics and machine learning applications.

2.1 Data Sources

The exponential growth of data in contemporary ecosystems from various sources contributes to this expanding data landscape, encompassing structured and unstructured formats. Structured data, often originating from relational databases like Oracle database, SQL server, Azure SQL etc., transactional systems, and organized datasets, poses unique challenges in terms of schema variability and evolving data structures. In contrast, unstructured data, derived from sources like social media, sensor logs, and textual documents, presents challenges related to lack of predefined data models. Additionally, the influx of streaming data from real-time sources like mobile/IOT devices further complicates the data ingestion process [7][8].

These diverse data sources demand sophisticated recurring scheduling service platforms capable of seamlessly assimilating data in its varied forms, at different times with variable loads. So, the focus, therefore, lies in a versatile ingestion scheduling library capable of handling the intricacies posed by the heterogeneity and dynamic nature of data from diverse sources.

2.2 Export/Publish Endpoint

As we deal with different data sources, having a generic class to perform some common operations like validations, security, endpoints configuration, data ingestions, state storage, testing etc., becomes crucial. We propose a programming paradigm called Publish Endpoint that stores metadata about the owner of the data, authority of this data, necessary permissions, endpoint information of different down streams, publish history, list of entities/artifacts to publish and version data [9].

Maintaining owner data on the endpoint, allows retrieving schema data and implementing logic for data validations, and security checks implicitly. Below is a basic interface for Publish Endpoint.

```
interface IPublishEndpoint : ISecurityParticipant
{
    string Name { get; }

    Task PublishEntitiesAsync(
        [NotNull] IEnumerable<IEntity> entities,
        [NotNull] IUserMessageCollection messages,
        [NotNull] IResourceContainer attachments,
        CancellationToken cancellationToken);

    Task ResetStateAsync(CancellationToken ct);

    void ValidateEntitiesPublicationAsync(
        [NotNull] IEnumerable<IEntity> entities,
        [NotNull] IResourceContainer attachments,
        CancellationToken cancellationToken);
}
```

2.3 Job Orchestrator

A streaming application undergoes compilation and optimization, resulting in the generation of a set of orchestrated jobs, as detailed earlier. Once provisioned, the Job Orchestration layer takes charge of ensuring that all jobs remain aligned with configuration changes initiated either by the user or internal services. This orchestration system ensures that changes are executed in an ACIDF-compliant, guaranteeing consistency [10][11]. To achieve this, the orchestration system has three key components:

- Job Repository (Job Store): This repository houses the current and desired configuration parameters for each job.
- Orchestration Service (Job Service): Responsible to ensure that job changes are committed to the Job Repository atomically.
- State Synchronization Service (State Syncer): Executes job update actions to transition jobs from their current state to the desired state.

Configuration parameters encompass all the necessary information for initiating the tasks of a given job. Examples include details such as binary names and versions, the required number of tasks for the job, and the allocated resources for these tasks. ACIDF job updates play a pivotal role in maintaining the manageability and high decoupling of this intricate system. In environments with tens of thousands of jobs, issues like job update failures and conflicts are frequent due to updates originating from various services. Consequently, these issues must be automatically resolved, and the outcomes should be easily comprehensible. Moreover, the job orchestration system must exhibit flexibility and extensibility to accommodate new services as needed. Notable examples of such services include the auto scale, integrated into the system after its initial deployment, and an auto root-causer.

2.4 Task Scheduling Library Overview

In the realm of building a robust autonomous schedulable ingestion – orchestrating recurring jobs by proactively identifying traffic patterns and scaling the resources plays a pivotal role in seamlessly capturing huge amounts of data from diverse sources and channeling it towards processing systems for real-time data ensuring reliability, scalability, and fault tolerance.
Basic components under this task scheduling library are shown in Figure 2 below.

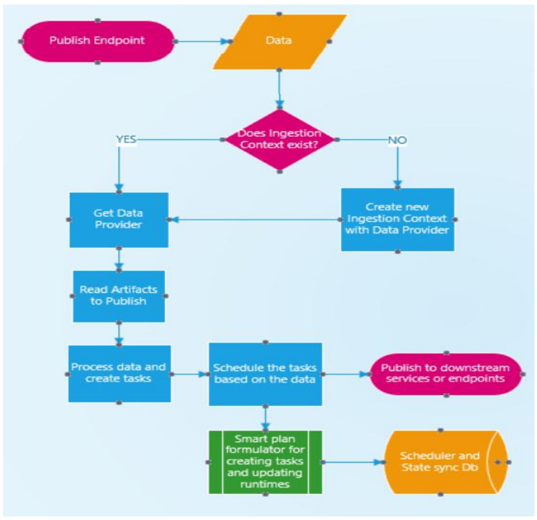
Publish Endpoint
Data
Does Ingestion Context exist?
YES
NO
Get Data Provider
Create new Ingestion Context with Data Provider
Read Artifacts to Publish
Process data and create tasks
Schedule the tasks based on the data
Publish to downstream services or endpoints
Smart plan formulator for creating tasks and updating runtimes
Scheduler and State sync Db

Figure 2: Task Scheduler Components Overview

- Ingestion Context
- Data Provider Factory
- Data Provider
- Recurring Task
- Scheduler

Once the data passes through basic evaluations for permissions and is ready for publish, we check if *IngestionContext* exists, and if it contains *DataProvider Factory.* Based on the configured data provider factory, we retrieve the data provider. Depending on the checks, we await the task to complete or get the results of the ingestion process and store state of the task in the endpoint container. These tasks are executed in different ways:

- When there is an event with data change, the listener activates the publish task and calls the corresponding endpoint.
- Recurring tasks wake up at a given time interval and create a micro-batch with the data changes from the last successful sync and start publishing. We identify the manifest last modified date to get delta of the changes.

As these tasks are configured at the endpoint level, there are several instances of these tasks for different endpoint types. So, scheduling these tasks, making sure they are executed in parallel, sync the state of the current & next tasks and perform scaling depending on the traffic needs a reliable library that can handle all the different scenarios. Below are some of the key considerations for the task scheduler:

- Ensure the scheduling of tasks without duplication, emphasizing that the system should never have concurrent instances of the same task. This holds true even in the event of failures in other components of the system. Additionally, it is crucial to prevent any loss of tasks. In scenarios where system components fail, newly created tasks may experience a delay in immediate scheduling.
- Implement fail-over mechanisms to redirect tasks to healthy hosts in the event of host failures. This contributes to system resilience and ensures continuous task execution.
- Automatically restart tasks that have encountered crashes. This capability minimizes downtime and maintains the integrity of task execution even during unexpected failures.
- Finally, implement a robust load balancing strategy to distribute tasks evenly across the cluster. This includes balancing CPU, memory, and IO usage to optimize resource utilization. Achieving a balanced distribution enhances overall performance and efficiency.

These considerations collectively contribute to a resilient and efficient task scheduling mechanism within the system, promoting reliability, fault tolerance, and optimal resource utilization [5][12].

2.5 Resource Manager

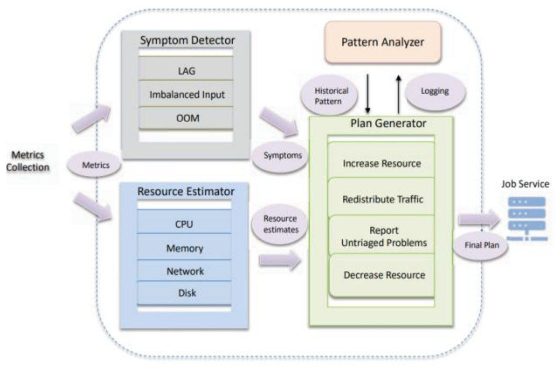

Figure 3: Architecture of Resource Manager.

We introduce Resource Evaluator and Plan Developer, pivotal components within the architecture of Resource Manager. The Resource Evaluator's primary function is to gauge the utilization of specific resources - such as CPU, memory, network bandwidth, and disk I/O—within a given job. The subsequent step involves the Plan Developer utilizing these estimations to formulate a resource adjustment plan. The accuracy of job resource estimation is contingent upon the inherent characteristics of the job.

Stateless jobs, encompassing tasks like filtering, projection, and transformation, do not necessitate state retention except for checkpoints used in case of task restarts. Conversely, stateful jobs, including aggregation and join operations, maintain application-specific state, involving memory and persistent storage, necessitating the restoration of relevant state components upon restarts. Stateless jobs typically exhibit high CPU intensity as they engage in input deserialization, data processing, and output serialization. Regardless of the dominant operation, CPU consumption correlates with input and output data sizes. For such jobs, metrics like input/output rates are harnessed to estimate the maximum stable processing rate per single-threaded task [5]. The CPU resource unit required for an input rate 'X' is estimated as:

$$X/(P * k * n)$$

where P denotes the maximum stable processing rate per single thread. The CPU resource estimation for recovering backlogged data 'B' within time 't' is expressed as:

$$(X + B/t) / (P * k * n)$$

In contrast, stateful jobs require estimation of CPU, memory, and disk usage. For example, in an aggregation job, memory size is proportional to the key cardinality of data held in memory. Conversely, for a join operator, memory/disk size correlates with join window size, degree of input matching, and degree of input disorder. The Resource Evaluator is configurable to estimate various dimensions of resource consumption, reporting them to the Plan Formulator. The Plan Formulator synthesizes decisions based on symptoms and resource estimates. It ensures the final plan provides adequate resources to execute a job by preventing downscaling decisions from jeopardizing the health of a job, averting untriaged problems from triggering unnecessary scaling decisions, and executing correlated adjustments for multiple resources. For instance, if a stateful job encounters a CPU bottleneck and the number of tasks is increased, the memory allocated to each task can be concurrently reduced [5].

3. Autonomous Scheduler-Deep Dive

This section dives deep into the autonomous scheduler and takes an opinionated view of how this can be implemented with code samples. We start by defining some interfaces that are needed to implement custom logic to create a autonomous scheduler:

Ingestion Publish Scope: this defines the scope of the ingestion – list of entities or artifacts that are available for processing and publishing.

```
interface IIngestionPublishScope
{
    IEnumerable<IEntity> GetEntitiesToPublish(
        ICatalog catalog,
        IUserMessageCollection messages,
        CancellationToken token);
}
```

Here is a concrete implementation for the entity manifest, which takes in a manifest-snapshot of data changes and transform that into artifacts.

```
[Serializable]
class EntityManifestPublishScope
      : IIngestionPublishScope
{
    private IReadOnlyEntityManifest? manifest;

    public EntityManifestPublishScope(
        IReadOnlyEntityManifest? manifest)
    {
        this.manifest = manifest;
    }

    IEnumerable<IEntity> GetEntitiesToPublish(
        ICatalog catalog,
        IUserMessageCollection messages,
        CancellationToken token)
    {
         return this.manifest?.GetEntityToDataTypeMapping()
        ?? catalog.Owner.TenantedEntities.OfType<Artifact>();
    }
}
```

Below are interfaces to capture information for scheduling, and retrieving data for ingestion:

```
interface ISchedulingInfo
{
    TimeSpan? AvailabilityDelay { get; }
    DistributedTaskPriority? Priority { get; }
    TimeSpan RecurringScheduleInterval { get; }
    ICollection<IDistributedTaskTag>? TagNames { get; }
}

public interface IReportingDataProvider
{
    IKeyedByTypeCollection<object> GetEntries();
    public Task<TimeSpan?> PrepareAsync(
        IUserMessageCollection messages,
        CancellationToken ct = default);
}

[Serializable]
class DataProviderFactoryActivator<TImpl> :
        DataTransformActivator<IDataProviderFactory, TImpl>
         where TImpl : class, IDataTransform, new()
{ }
```

Calculate Availability Delay: below is the method to calculate the availability delay which is the time when the task is scheduled to start. This is calculated based on when the recurring schedule is and when the previous task started to execute and completed. This configuration can be updated to match the requirements and needs of the SLAs.

```
TimeSpan? CalculateAvailabilityDelay(
       TimeSpan recurringScheduleInterval,
       DateTime previousTaskStartTime,
       DateTime currentTime)
{
    TimeSpan intervalTimeSpan
       = currentTime - previousTaskStartTime;
    bool isNextScheduleReached
      = intervalTimeSpan > recurringScheduleInterval;

    if (isNextScheduleReached)
    {
       return null; // no delay time is required.
    }

    DateTime nextTaskExpectedStartTime
       = previousTaskStartTime.Add(recurringScheduleInterval);
    return nextTaskExpectedStartTime - currentTime;
}
```

Retrieve Data Provider for the job: The algorithm below describes how to check and retrieve data for ingestion and maintain consistency [3][9]:

- Check if the *IngestionContext* exists.
- If it doesn't exist, create a context for the given catalog and publishing task.
- If the context exists and has a reporting provider: prepare the reporting provider.
- Set *DataProvider* to the existing provider.
- If the *IngestionContext* does not exist or has no *DataProvider*: create or use an existing data provider.
- Prepare the data provider, which sets the required data to be executed.
- If there is an availability delay, throw away a transient exception and return the reporting data provider.

```
FUNCTION GetDataProvider(provider, entities):
  context = CreateContext
    (publishTask, dataProvider)
  doesContextExist
    = context.ExistsAsync()

  IF doesContextExist
    AND context.DataProvider != NULL:
      availabilityDelay =
        context.DataProvider
        .PrepareAsync()
      dataProvider = context.DataProvider

  ELSE
      dataProvider =
        CreateNewDataProvider(provider, entities)
      availabilityDelay = provider.PrepareAsync()
      context = CreateIngestionContext
        (publishTask, dataProvider)

  IF availabilityDelay != NULL
    AND availabilityDelay != TimeSpan.Zero
     THROW GetTransientException(publishTask,
       availabilityDelay)

  RETURN dataProvider
```

Calculate Effective Auto Schedule Interval:

- Calculate total processing time by subtracting the planned start time from the current time.
- Check if the auto-schedule interval is greater than the total processing time.
- If yes, calculate the effective auto-schedule interval by subtracting the total processing time from the original auto-schedule interval.

Reschedule or Set Next Task:

- If the effective auto-schedule interval is calculated, call the *ScheduleNextTask* with the effective interval to create next recurring task.
- Set the previous task for the new task asynchronously. Save the new task with the messages provided and cancellation token.
- If the saving is successful, return the newly created task. If the effective auto-schedule interval is not calculated or the next task cannot be scheduled: no rescheduling occurs.

Handling Next Task:

- Get a new task from the *ScheduleNextTask*.
- Set the obtained task as the next task.
- Asynchronously, set the next task, save changes, and handle any exceptions.

```
private async Task HandleRescheduling(
   CancellationToken ct = default)
{
   TimeSpan? effectiveAutoScheduleInterval = null;
   TimeSpan totalProcessingTime
     = SystemTime.UtcNow.Subtract(this.PlannedStartTime);

   if (this.AutoScheduleInterval > totalProcessingTime)
   {
     effectiveAutoScheduleInterval
      = this.AutoScheduleInterval.Subtract(totalProcessingTime);
   }
   else
   {
     var nextTask = await this.ScheduleNextTask
           (effectiveAutoScheduleInterval, ct).ConfigureAwait(false);
     if (nextTask != null)
     {
       // Set next task
       await this.SetNextTaskAsync(nextTask, ct).ConfigureAwait(false);
       await this.SaveAsync(this.Messages, ct).ConfigureAwait(false);
     }
}

private async Task<RecurringTask> ScheduleNextTask(
   TimeSpan? effectiveAutoScheduleInterval,
   CancellationToken ct = default)
{
   RecurringTask nextTask = await CreateNextTaskAsync(
     this.AutoScheduleInterval,
     this.SetNextTaskName(),
     effectiveAutoScheduleInterval,
     this.schedulingTags, ct).ConfigureAwait(false);

   // Set previous task
   await nextTask.SetPreviousTaskAsync(this, ct);
   bool isSavedSuccessfully = await nextTask.Save(this.Messages, ct);

   if (isSavedSuccessfully)
     return nextTask;

   return null;
}
```

4. Conclusion

We presented an expansive autonomous task scheduling library tailored for large-scale stream processing applications. This library capitalizes on a robust cluster management system, augmenting it with loosely coupled microservices dedicated to determining job execution (Job Orchestrator), orchestrating task scheduling (Task Scheduler), and optimizing resource management strategies (Resource Manager). This integration results in a highly scalable and resilient management infrastructure, adept at supporting numerous pipelines processing extensive data volumes with minimal human intervention [6][9][13][14].

Moving forward, our trajectory involves integrating machine learning techniques for automating root cause analysis and incident mitigation, thereby alleviating the need for manual intervention in incidents. Additionally, we are exploring avenues to enhance resource utilization by minimizing reserved capacity headroom and refining task placement through the implementation of a continuous resource estimation algorithm. These advancements aim to further streamline and optimize the operational efficiency of our stream processing framework.

References

[1] "ADVANTAGES OF DATA-DRIVEN DECISION-MAKING," [Online]. Available: https://online.hbs.edu/blog/post/data-driven-decision-making.
[2] "Beyond The Buzzword: What Does Data-Driven Decision-Making Really Mean?," [Online]. Available: https://graduate.northeastern.edu/resources/data-driven-decision-making./.
[3] T. H. a. J. Puniš, "An Overview of Current Trends in Data Ingestion and Integration," no. 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO), Opatija, Croatia, 2021, pp. 1265-1270, doi: 10.23919/MIPRO52101.2021.9597149..
[4] Kleppmann, Martin, Designing Data-Intensive Applications, O'Reilly Media, 2017.
[5] C. L. T. V. L. M. Y. J.-S. G. N. S. A. B. Mei Yuan, "Turbine: Facebook's service management platform for stream processing," *IEEE*, pp. 1591-1602, 2020.
[6] "Hadoop Capacity Scheduler.," [Online]. Available: https://hadoop.apache.org/docs/r1.2.1/.
[7] "Apache Spark," [Online]. Available: https://spark.apache.org/.
[8] A. Flink. [Online]. Available: https://flink.apache.org/.
[9] S. E. G. F. S. H. S. R. a. K. T. P. Carbone, "State management in Apache Flink: Consistent stateful distributed stream processing," *VLDB*, 2017.
[10] "Apache Aurora.," [Online]. Available: http://aurora.apache.org/.
[11] -"No shard left behind: dynamic work rebalancing in Google," [Online]. Available: https://cloud.google.com/blog/products/gcp/.
[12] J. L. W. S. I. A. J. R. L. Guoqiang Jerry Chen, "Realtime Data Processing at Facebook," *SIGMOD*, p. 1, 2016.
[13] "Integrate Azure Stream Analytics with Azure Machine Learning," [Online]. Available: https://learn.microsoft.com/en-us/azure/stream-analytics/machine-learning-udf?source=recommendations.
[14] A. A. G. G.-G. G. H. F. C. a. K. M. P. Ta-Shma, "An Ingestion and Analytics Architecture for IoT Applied to Smart City Use Cases," *IEEE Internet of Things Journal*, vol. 5, pp. 765-774, 2018.
[15] "Azure Event Hubs," [Online]. Available: https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-about.
[16] "Apache Kafka," [Online]. Available: https://kafka.apache.org/.
[17] "Spark Streaming vs Flink vs Storm vs Kafka Streams vs Samza : Choose Your Stream Processing Framework," [Online]. Available: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b.
[18] "Why the Data You Use Is More Important Than the Model Itself," [Online]. Available: https://medium.com/swlh/why-the-data-you-use-is-more-important-than-the-model-itself-4a49736ea70c.
[19] "Is Data More Important Than Algorithms In AI?," [Online]. Available: https://www.forbes.com/sites/quora/2017/01/26/is-data-more-important-than-algorithms-in-ai/?sh=111664a542c1.
[20] "Process data from your event hub using Azure Stream Analytics," [Online]. Available: https://learn.microsoft.com/en-us/azure/event-hubs/process-data-azure-stream-analytics.
[21] "Use Azure Schema Registry in Event Hubs from Apache Kafka and other apps," [Online]. Available: https://learn.microsoft.com/en-us/azure/event-hubs/schema-registry-overview.
[22] "Azure Event Hubs," [Online]. Available: https://learn.microsoft.com/en-us/azure/event-hubs/event-hubs-about#how-it-works.
[23] "Deploy and score a machine learning model by using an online endpoint," [Online]. Available: https://learn.microsoft.com/en-us/azure/machine-learning/how-to-deploy-online-endpoints.
[24] J. L. M. H. D. D. M. F. T. R. V. Kalavri, "Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows.," *OSDI*, 2018.