2023 IEEE/ACM 5th International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT) | 979-8-3503-0188-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/SERP4IoT59158.2023.00010

Event Driven Architecture: An Exploratory Study on The Gap between Academia and Industry

Nader Trabelsi
École de technologie supérieure
Montreal, Canada
nader.trabelsi.1@ens.etsmtl.ca

Cristiano Politowski
École de technologie supérieure
Montreal, Canada
cristiano.politowski@etsmtl.ca

Ghizlane El Boussaidi
École de technologie supérieure
Montreal, Canada
ghizlane.elboussaidi@etsmtl.ca

***Abstract*—Due to their applicability in different domains, Internet of Things software solutions have gained significant attention in many industries. Suitable architectures are needed in order to satisfy the needs of such applications. Event-Driven Architectures (EDA) ease the exchange of data between IoT devices while making each one of them completely decoupled from the others. EDA has been described in many studies and some industrial solutions are proposed to support such architecture. However, no study has evaluated the gap between the developed theory and the proposed industrial solutions. In this exploratory study, we selected and analyzed three industrial platforms according to what is described in the literature. Our results show that there is a lack of a unified definition of the elements of an EDA in academic studies. We also found that event-based communication is implemented differently in each platform, serving different use cases, which indicates the need to establish a commonly accepted definition and design-methodology for this type of architecture.**

***Index Terms*—Event-Driven Architecture, Event-based architecture, Software architecture, IoT**

I. Introduction

IoT applications provide insights to companies about their products and services to improve their decision-making processes. An IoT application is typically composed of devices (i.e., sensors and actuators) connected to the Cloud. Scalability and heterogeneity are key attributes of such applications. Event-Driven Architecture (EDA) can help achieve these quality attributes. EDA has been around for more than two decades [1], and has been used at different levels of the technological stack, from hardware, like electrical circuits and sensors [2, 3] and networking, like event-triggered protocols[4], to software, like manufacturing information systems [5].

EDA is an architectural pattern where components communicate by reacting to events published by other components. This enables high decoupling between software components. In an IoT application, EDA enables components (e.g., sensors, services, etc.) to produce events that can be consumed by other components listening to these events. Despite its growing popularity in IoT systems, there is no consensus in the literature on the set of components that make up the EDA pattern and their interactions. Moreover, existing industry platforms that support EDA provide different capabilities in terms of event specification, streaming and management. Designers and developers need to understand the theoretical concepts related to EDA and how they are supported by these platforms to make appropriate choices and decisions in their IoT projects.

The goal of this exploratory study is to discuss EDA elements as introduced in academia and the gaps related to industry implementations. By elements we mean the components and the interactions between them. Specifically, our study aims at answering the following research questions:

- RQ1. What are the elements of Event-Driven Architecture according to academic studies?
- RQ2. Can the elements of Event-Driven Architecture, defined in the literature, be found in industrial platforms?

To answer RQ1, we searched for studies about EDA. We then excluded studies that (i) are not software-related (*e.g.*, electrical circuits), (ii) are not related to software design or architectures such as algorithms or any low-level implementation concern, or (iii) propose an architecture that is event-driven without describing the pattern itself. We finally selected a set of 8 studies that are described in Section II. For RQ2, we analyzed three EDA industry platforms and investigated the gap between industry and academia.

The academic results show a lack of consensus on the definition of EDA. In addition, the industry platforms are loosely connected to the theory we found in academia. Also, the studied platforms presented features that were not discussed in the academic definitions.

This paper is organized as follows. Section II discusses related work. In Section III we describe the methodology we used to collect data and analyze the results. In Sections IV and V, we report the results of the academic studies and the industrial platforms, respectively. In Section VI, we compare the industrial solutions to what was described in the literature. Section VII concludes the paper.

II. Related Work

There are several studies that discuss elements of an EDA. Carzaniga et al. [6] studied variations of the EDA style that had been introduced by some middleware infrastructures and discussed the architectural elements that differentiated between them, namely the subscription mechanisms, the structure of notification and the scalability properties. Mühl et al. [7] discussed some basic concepts of event-based systems including the EDA terminology, the interaction model and data modelling, filtering and routing.

Hinze et al. [8] discussed the foundations of event-based systems and their functional and non-functional features.

979-8-3503-0188-5/23/$31.00 ©2023 IEEE
DOI 10.1109/SERP4IoT59158.2023.00010

Cristea et al. [9] presented EDA components and their roles in the generation, consumption, processing and transmission of events. They discussed the impact of different deployments (*i.e.*, assignment of logical components to physical machines) on the quality attributes.

Michelson [10] reviewed EDA and described its fundamental notions and benefits. Richards [11] and Goniwada [12] discussed two variations of the EDA according to its topology, namely the mediator-based and the broker-based. Strothmann and Bachmann [13] introduced the main components of an EDA and discussed different types of events. They also discussed the benefits and challenges of adopting this architectural pattern.

All these studies contributed to defining EDA and its elements. However, to the best of our knowledge, no study evaluated the gap between the academic definition of EDA and its implementation by industry platforms.

III. Methodology of the Study

Our methodology was organized into two phases. Each phase targets one of the research questions.

A. Synthesis of EDA elements from Academic Studies

In this phase, we studied academic literature that discusses EDAs to answer our first research question **"RQ1. What are the elements of Event-Driven Architecture according to academic studies?"**. The surveyed studies were shortly described in Section II.

In software architecture, we have two types of elements: *Components* and *Interactions* between them [14]. For EDA, we also need to discuss the *Event* concept. Thus, we developed our first research question into three sub-questions:
(1) **RQ1.1 What are the characteristics of an Event?** We looked in the academic studies for anything that relates to the concept of an event, *e.g.*, definition, content, etc.
(2) **RQ1.2 What are the architectural components of an EDA?** We looked into all the components of the EDA.
(3) **RQ1.3 What are the interactions between the components of EDA?** We looked at how the communication between the components is done.

We then proceeded to synthesize the results either by (i) concluding a common description for each element of EDA or by (ii) presenting the distinct views on an EDA element when a common description can not be found. We use a scenario of an IoT weather application to explain the EDA elements. The main features of this application are: (i) it allows users and systems to monitor the weather and hourly and daily forecasts in many cities; (ii) the data provided by the application are coming from various sensors (*e.g.*, temperature, wind speed); (iii) users and systems can choose to receive alerts about some weather conditions (*e.g.*, temperature, humidity) and/or weather types (*e.g.*, sunny, cloudy, rainy). The results of this phase are reported in Section IV.

B. Study of EDA Platforms and analysis of the gaps

In this phase, we studied three common EDA platforms to answer our second research question **"RQ2. Can the elements of Event-Driven Architecture, defined in the literature, be found in industrial platforms?"**.

The first platform we choose was **Apache Kafka**. Apache Kafka is open source and has 20.19% market share (out of 138,769 companies) as an Enterprise Application Integration[1]. We also selected the AWS (Amazon Web Services) **EventBridge Event Bus**[2], the one recommended by AWS to build an application where components react to events[3]. The third platform was Google's **Eventarc**[4]. which is Google's recommended solution to build EDA[5].

To understand how the selected platforms support EDA, we analyzed their official documentation. For each element (*i.e.*, components and interactions) considered in the literature, we tried to find a realization of it in the platforms. Finally, we synthesized the results identifying some gaps between the theory and the practice of EDAs.

IV. EDA in Academia

In this section, we present EDA elements that we synthesized from the literature.

A. RQ1.1 What are the characteristics of an Event?

An event is **a change in the state of a software component** which is of interest to another system or component [6, 7, 9, 13]. In the described IoT weather application, the state of a service linked to a sensor is represented by the value of the weather condition measured by that sensor.

1) Categories of Events: Events can be classified according to their *level of abstraction*, *complexity*, *concern*, or *notability*.

The level of abstraction of events can be *primitive* (*i.e.*, basic) or *abstract* (*i.e.*, derived) [7, 8]. A primitive event carries raw simple data created by a source (*e.g., sensor*). When received by another component, the handling of the primitive event can generate an abstract (derived) event. In the IoT weather application, a primitive event is a temperature sensor reading, this event can then be consumed by another component that, by comparing the reading to the previously received one, detects no change to which it reacts by creating and publishing a new 'stable temperature' event. The latter event is therefore derived from (or abstracts) the primitive one received from the sensor.

The complexity of an event can be *atomic (simple)* or *composite (complex)*. The complex events are aggregations of atomic events or other composite events. In the IoT weather example, a composite event can be a timestamped sequence of sensors' readings across different regions of a city. Complexity is a question of composition that is usually the result of some Complex Event Processing [10].

[1]https://enlyft.com/tech/enterprise-application-integration
[2]https://aws.amazon.com/eventbridge/
[3]https://aws.amazon.com/event-driven-architecture/
[4]https://cloud.google.com/eventarc/
[5]https://cloud.google.com/eventarc/docs/event-driven-architectures

In an EDA, events can be of two categories: *business* or *technical*. Technical events refer to infrastructure or environment events. All other types of events are considered business because they are directly associated with the domain. A brainstorming exercise (i.e., EventStorming [15]) can help find out events in a particular domain. In the IoT weather example, the loss of connection to a sensor is a technical event. While a snowfall warning is a business event.

Finally, events can be *ordinary* or *notable* [10]. Ordinary events may be evaluated for notability thus causing the generation of a notable events. In the IoT weather application, a temperature exceeding a defined threshold triggers the generation of a high-temperature event.

2) Event Content and Exchange Format: Some of the academic studies specify what an event should contain, and, in some cases, the exchange formats of the event.

a) Content: An event has two parts: (i) a header that contains a description of the instance, this description includes a reference to the event specification (identifier, name), an occurrence number, the event source and the occurrence time (a timestamp) [8, 10]; and (ii) a body containing the details of the event.

b) Exchange Format: The data contained within an event can be presented and exchanged using different formats [7]: (i) tuples that represent an ordered set of attributes (*e.g.*, `(Temperature,''Glasgow'', 45)`); (ii) structured documents organized as a set of attributes each having a unique name, they can be flat or hierarchical; (iii) semi-structured documents where an attribute can only be addressed by its full path; and (iv) objects: where event consumers must register to the publisher.

B. RQ1.2 What are the architectural components of an EDA?

In this section, we summarize a common definition of the main architectural components described in the literature [6, 7, 8, 9, 10, 11, 12, 13]. These components are shown in Figure 1.

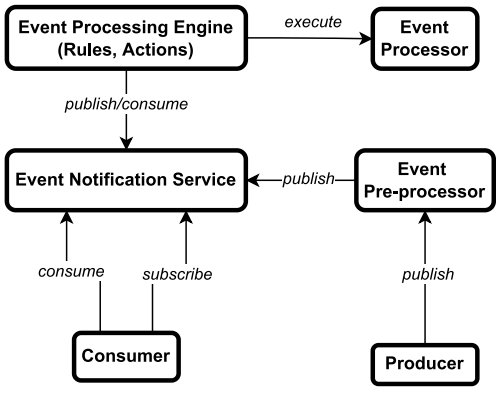

Fig. 1: Components of an Event-Driven Architecture.

1) Producer: or *Publisher*, is a component that acts as an event provider. It can play one or two of the following roles: (i) the *source* of events which is also called a generator, originator, or creator of events (in the IoT weather example, this is the sensor), and (ii) the event *sender* which is responsible for event publishing (in our example, this can be the software linked to the sensor) [9].

2) Consumer: represents the recipient of events. Consumers subscribe to the events they are interested in [7, 10, 12]. Michelson [10] suggests a definition that includes any downstream activity that may be triggered by an event, like an invocation of a service or the initiation of a process, and not only a subscriber. In the IoT weather application, a mobile app running on the users' devices can be consumer of the events.

3) Event Notification Service: Event Notification Service (or *ENS*) has two functions: (i) management of subscriptions initiated by the consumers who want to receive events that satisfy some criteria expressed in the subscription, and (ii) dissemination of event notifications from producers to consumers. In the IoT application, this component will receive the events about the weather conditions of the different cities, and route them to the interested users and systems.

4) Event Pre-processor: It is attached to a producer, it performs actions before the event is published to the ENS. Three types of pre-processing are described in the literature [10]. (i) *Transformation:* of the event into a standard format accepted and understandable by other components. In our IoT example, sensors may send readings formatted according to different norms, and an event pre-processor reformats each of them so they can be interpreted by a mobile app or other subscribing systems. (ii) *Filtering:* by selecting events based on some criteria. In example, we may decide to ignore a low level of precipitations in a region known to be arid. (iii) *Routing:* which has the role of evaluating the events generated by the producer against criteria of notability, and then, if necessary, generates and publishes notable events themselves.

5) Event Processing Engine: Applies rules defined according to the needs of the consumers. These rules, sometimes defined with the Event, Condition, Action (ECA) paradigm, associate conditions that the event must satisfy to a certain action that is triggered automatically when the condition(s) is met by the event [9]. When processing is based on multiple streams of events rather than individually-published events, continuous queries are used [8, 16].

6) Event Processor: executes actions decided by the event processing engine. These actions can be [10]: (i) notification of subscribers to events, (ii) an invocation of a service or (iii) capture of events (e.g., in a database). In our IoT example, an action can also be the direct control of an actuator (*e.g.*, a heating system).

Three additional EDA elements are discussed in some studies. (1) Event Channel: supports the communication of published events [10]. It adds some queuing function to the ENS and may be located between the ENS and consumers so as not to flood consumers by over-pushing events to them [11]. It can also be used inside the ENS to order events. (2) Event development tool: used to specify events and rules, and to ensure their co-evolution. (3) Event management: supports infrastructure administration and monitoring activities.

C. RQ1.3 What are the interactions between the components of EDA?

In this section, we describe four elements of interaction that we have found in the literature.

1) The event-based interaction model: Table I, adapted from [7], summarizes different interaction models. The *initiator* of interaction specifies whether the consumer or the provider of functionality/data is first to interact with the other. The *destination* represents the type of addressing (direct or indirect). In the event-based interaction model, as opposed to the request-reply model, the provider of functionality/data is the initiator of communication. The event notifications are not directed to a specific set of consumers. As opposed to the callback model, providers do not have to manage routing and subscriptions by themselves. In EDA, the Producer component plays the role of an initiator that is the provider of events. These events can trigger a reaction from the Consumer component which plays the role of a destination that is indirectly connected to the Producer. The event-based interaction model can be implemented using different communication techniques like publish/subscribe and point-to-point messaging.

TABLE I: Interaction models (adapted from [7])

Destination	**Initiator**	
	Consumer	*Provider*
Direct	Request/Reply	Callback
Indirect	Anonymous Request/Reply	Event-based

2) Transmission model: The ENS transmits notifications to consumers using one of the two following models [7, 8, 9, 10]. (i) Pull: the consumer actively looks for new events by requesting them from the ENS. In the IoT weather example, this can allow mobile app users to receive notifications whey they restore their internet connection. (ii) Push: the ENS pushes events to the consumer. For instance, an event (*e.g.*, ThunderstormDetected) can be considered an emergency and thus it has to be pushed to the consumer to be processed as soon as possible.

3) Subscription mechanisms: Consumers register their interest in a subset of events, based on one of the mechanisms described in Table II.

TABLE II: Subscriptions mechanisms in EDA

Mechanism	**Description**
Channel	Consumers are listening to events coming from a specific channel on which producers are publishing [6, 7, 8, 11].
Subject	Notifications contain events to which subjects, sometimes called topics, are attached. Consumers specify the subject(s) under which they want to receive notifications [6] [7] [8].
Content	Events are filtered based on the values given to the attributes of the event [6] [7] [8]. For instance, a user can choose to receive an alert only when the temperature of some city exceeds 25 Celsius.
Pattern	The filtering criteria relate to the structure (presence of attributes) in the event notification and not the values [6].
Type	Useful when we can have hierarchy of events. Consumers indicate what type of events they want to receive [7].

4) Topologies: Richards [11] and Goniwada [12] describe two topologies for routing events between producers and consumers in an EDA. (i) The *Broker* topology: where the only function of the ENS is to route event notifications between producers and consumers. In our IoT example, when a temperature sensor reading is published to the ENS, it will route it to the consumers interested in it. (ii) The *Mediator* topology where the ENS holds some of the application's business logic that enables it to react to initial events received from producers by translating them into a defined process consisting of a sequence of processing events sent to consumers. For instance, when a SnowstormComing event is received, the ENS (as a mediator) reacts by sending 2 processing events: (i) an event notification pushed to the users living in the city where the storm is detected, and (ii) an event to automated window-control systems of subscribed hotels.

V. EDA in Industry

In this section, we answer the RQ2, *Can the elements of EDA, defined in the literature, be found in industrial platforms?*. We investigated how each of the platforms implements each of the elements discussed in Section IV. For each platform, we give an overview and discuss details about the event, architectural elements and their interactions. We finish the section by analyzing the differences and similarities between the platforms.

A. Apache Kafka

Kafka is an open-source event streaming platform. Its goal is to allow applications to publish, consume, store, and process streams of events in real-time with high throughput. Kafka works with a cluster of servers called *brokers* where *events* are stored in different *partitions* within different *topics*.

1) Event: An event in Kafka is called a *record* or *message*. It's an object composed of three parts: a *Key*, a *Value* and a *Timestamp*. The Key is used by a particular partitioning strategy used by the producer, *i.e.*, messages are organized under the topic to enable parallelism when reading (from consumers) or writing (from producers) on different brokers. The Value is of primitive type or a custom data class instance. The Timestamp indicates the moment the producer sent the message. An event in Kafka also contains a headers section where to attach meta-data.

Defining an event in Kafka is done by defining a class representing the entity associated with the event for the Value part of the record. Moreover, to create an instance of it, it is necessary to instantiate the data class for the value and one class for the key. The latter plays the role of an identifier, chosen according to a partitioning strategy. In Kafka, there is no native way to categorize events as we discussed in Section IV-A1.

2) Architectural Components: A *producer* is defined using the Producer API, instantiating the class `KafkaProducer`. Likewise, a *consumer* is defined using the class `KafkaConsumer`. The ENS is supported with a cluster of brokers fills. It is done by storing the events published

by the producers and fulfilling the requests of consumers looking for new records. As the partitions under each topic are append-only logs of records, Kafka's cluster keeps track of the position of the last read record for each consumer. Consumers can also change this position to re-read the events.

Aside from the main elements, Kafka also has: (i) *Event Processing Engine*: Kafka Streams API is used to define processing pipelines and to detect complex events based on windowing techniques and joins between partitions. (ii) *Event Management*: Kafka Admin API provides ways to inspect and manage the configuration of a cluster of brokers. (iii) *Event Processor*: Kafka Processor API defines a unit of processing that can be plugged into a sequence of processors representing a pipeline in Kafka Streams. (iv) *Event Channel*: a topic represented by at least one partition, that is, a log file in which events are appended to the previously published ones.

3) Interactions between the Elements of EDA: Interaction is done by subscribing to the topic partitions from which a consumer wants to receive events. The consumer can read from them at its own pace.

Kafka supports three *subscription mechanisms*: (i) *Channel-based* by subscribing to a particular partition under a topic; (ii) *Subject-based* by subscribing to a topic; (iii) *Type-based* by using a wildcard string (*e.g.*, `"/weather/*"` for receiving all events published under topics starting with `weather`).

For the *transmission models*, only the pull model is supported by Kafka. Also, the broker *topology* is the only one supported in Kafka. The mediator topology is not supported because the Kafka broker does not process the events it receives nor does it initiate communication with the consumers.

B. EventBridge Event Bus

Offered by Amazon as part of their EventBridge suite of SaaS services aiming at helping companies in building event-driven applications, this service allows applications to receive, filter, transform and route events to other applications. It is designed to support integration with, not only the other AWS services or other SaaS providers, but also with applications outside of AWS.

1) Event: In EventBridge, an event is a JSON object that contains three fields: (i) *detail* where the actual event information is written; (ii) *detail-type* which contains a string indicating the name of an event; and (iii) *source*, the source of an event (*e.g.*, `aws.cloudfront` for Amazon CloudFront).

EventBridge uses its Schema Registry to define the structure of an event. This service allows developers to register a schema (a pre-defined structure) of JSON files. EventBridge requires the use of some specifications (OpenAPI or JSONSchema Draft4) when creating a JSON schema, in order to ensure consistency and validity between components of EDA.

To instantiate an event, EventBridge automatically generates code bindings that can be downloaded for a registered schema. The code binding contains the code needed to do the marshalling and unmarshalling of a JSON schema, *i.e.*, the transformation from a JSON schema to a JSON string format that can be handled in code, and vice-versa.

About the category of the events, while there is a pre-defined set of values that are possible for the *source* field, these values are very "AWS-centric", *e.g.*, `Object created` for the source `S3`. This categorization depends highly on the actions possible on every AWS service.

2) Architectural Components: To define an *producer*, we can use the provided API called *PutEvents* available as part of the official client library. As for the *consumer*, a consuming application, called target in EventBridge, must have an HTTP endpoint to which events are sent following the matching of an event by a defined rule in the event bus.

Regarding the ENS, the event bus is the one responsible for filtering, transforming and routing incoming events. It does save the events as long as the delivery to their targets is not successful.

The role of *Event Management* is partially filled thanks to the metrics of CloudWatch, the infrastructure monitoring service of AWS, where each metric has three properties: the event bus, the event source and the name of the rule that was triggered by the event.

3) Interactions between the Elements of EDA: The *event-based interaction model* is supported By creating a rule in the event bus, the target application will receive an HTTP request from EventBridge to the API endpoint it has defined as part of the rule. Whenever a new event, matching the criteria in the rule, is published, the target will be invoked as soon as it is available.

On the *subscription mechanisms*, there are two ways to subscribe to events: (i) *Content-based* by creating a rule using an event pattern that defines the structure, fields, and values to be matched; and (ii) *Pattern-based* which is possible through the definition of structure and fields in addition to the *Exists matching* technique by adding `{"exists": true}` as a value in the JSON of the event pattern.

About the *transmission models* in EventBridge, its Events are pushed to the targets. Finally, only the broker *topology* is supported with a limit of five targets per rule.

C. Eventarc

Google Cloud introduced Eventarc to help software designers build an EDA. Eventarc uses GCP's Pub/Sub as the transport layer. The source of events, called *provider*, sends events to Eventarc which, in turn, routes them to event *destinations* or *targets*.

1) Event: Eventarc accepts events in any format, and it sends them to destinations in a message formatted according to the CloudEvents specification which aims to describe and interpret the syntax of events in a common way [12, 13, 17].

In Eventarc: (i) there is no service associated with defining an event, (ii) it is the publisher's responsibility to manage the events it wants to publish, and (iii) events are categorized based on the source (a GCP resource) and the action that was applied to it. When creating a trigger for Google Cloud services, we must provide the name of the service and the operations, *i.e.*, the action that was applied on a specific resource (*e.g.*, a bucket name in the case of Cloud Storage).

2) Architectural Components: A *producer* is defined by (i) creating a ChannelConnection (a Google Cloud resource that bridges between the third-party provider and an event consumer) in a publisher project, and (ii) using the provided Eventarc Publishing API to communicate with the Channel-Connection. This is limited to one consumer. This means that for each consumer we have to define a ChannelConnection between the publisher and that consumer.

As for the *consumer*, there is no way to define an event destination that is not one of the following Google Cloud services: Cloud Functions, Cloud Run, GKE, and Workflow. For the ENS, the filtering and routing functions are both full-filled by Eventarc. Eventarc matches the incoming events to destinations based on criteria checked on the event. Eventarc relies on Google's Pub/Sub service as a delivery mechanism.

The auditability function of the *Event Management* element of EDA is insured by the Cloud Monitoring service, GCP's infrastructure monitoring service.

3) Interactions between the Elements of EDA: In Eventarc there are four types of event destinations: Cloud Functions, Cloud Run, GKE, and Workflow. All of them are invokable services. When a published event matches one of the triggers, it is directly transformed into a call to the service specified in the trigger.

Two *subscription mechanisms* are supported: (i) *Channel-based* which is the case when we specify the service name, the resource name and the operations as part of the trigger, Eventarc will create a Pub/Sub topic for the specified filter. (ii) *Subject-based*, this is the case when we specify the name of an existing Pub/Sub topic on which we want to receive new events. For the transmission model, Eventarc pushes events to the destinations. Finally, Eventarc receives and delivers events without any interpretation of their content which makes it a good fit for the broker *topology* but not for the mediator.

D. Comparison of the platforms

Table III summarizes the results of our analysis of the three platforms.

The event attributes proposed by Kafka and EventBridge for the content of event messages do not cover the same elements; the attributes of Kafka (key, value and timestamp) are mainly impacted by the partitioning algorithm used in Kafka. The attributes of an event in EventBridge include a reference to the event source and the event name (in the field detail type) to identify what the event is about. Eventarc does not provide any attributes that publishers must use but, on the other side, it sends events to targets using a structure declared using the CloudEvents specification.

In Kafka, events are objects, while in EventBridge they take the form of JSON. Eventarc accepts events in any format. For the event specification and instantiation, an event content in Kafka (the Value attribute) is, in most cases, a data class representing the entity or resource associated with the event; we instantiate an event by simply instantiating the class. EventBridge uses the Schema Registry service as a repository of the events' structures; to publish an event, developers simply look into the schema using the name or identifier of the event and then make a copy of it. Eventarc does not specify how an event can be declared and instantiated.

Kafka offers the Connect API that allows to directly integrate sources and sinks of data using *connectors* (*e.g.*, JDBC connector) that can be found on public repositories like Confluent Hub. EventBridge consumers must provide an HTTP endpoint to which event messages are posted. Eventarc only supports a limited set of four consumers, all part of the GCP ecosystem.

When it comes to the ENS, Kafka keeps records for a configurable amount of time (with no upper limit) which is useful in event-sourcing scenarios when events must be replayed or reprocessed by the receivers. The other two platforms use the exponential back-off strategy for the delivery of events to consumers with a period of retries going up to a maximum of 24 hours.

Kafka is the only platform out of the three that has event processing capabilities and an API to manage and monitor the internal state and organization of the ENS, while the other platforms support the monitoring function only.

In the interaction between consumers and the ENS, a different set of subscription mechanisms is supported by each platform. The delivery of events in Kafka is based on a pull model where consumers have to send requests for new events at their own rhythm as opposed to the push model, adopted by the others, where consumers are passive listeners that react upon invocation by an event pushed by notification service.

Although EventBridge and Eventarc categorize events based on the set of services offered by the corresponding cloud provider, these categories are specific to their own ecosystems.

All the platforms use middleware as part of the ENS to disseminate events from producers to consumers. It is used as a backbone component to accomplish the event-based interaction model that is well-supported in these platforms.

Every platform offers an API used by the producing applications or services to publish their events to the ENS.

Because none of the platforms offers functions to translate a published event to a sequence of calls to applications or services and/or transmission of events to consumers, they do not natively support the mediator topology.

VI. Discussion

In this section, we discuss the differences and similarities between the results of the analysis of the platforms and those of the analysis of academic descriptions of EDA.

A. EDA Events

Although EventBridge Schema Registry uses the OpenAPI and Draft4 specifications to declare the structure of an event in JSON, these specifications are not event-oriented and they are mainly used to ease the interpretation of JSON files. Eventarc uses CloudEvents to send events to targets but it does not require a special structure for incoming events. Among the three specifications found, CloudEvents is the only one that is event-oriented, however, it is not adopted by all the studied

TABLE III: Comparison of the EDA platforms.

Element \ Platform		Apache Kafka	AWS EventBridge Event Bus	GCP Eventarc
Event	Content	Key, Value, Timestamp	detail, detail-type, source	-
	Form	Object	JSON	Any
	Event specification	Class declaration	JSON using EventBridge Schema Registry	-
	Event instantiation	Class instantiation	Code bindings of a Schema	-
	Event categorization	-	AWS-specific	GCP-specific
Architectural Elements	Producer definition	Kafka Producer API Kafka Connect API	PutEvents API	Event Publishing API
	Consumer definition	Kafka Consumer API Kafka Connect API	Provide an HTTP endpoint	-
	Event notification service	Routing Persistence	Routing Temporary Persistence	Routing Temporary Persistence
	Support for other elements	Event Processing Engine (Kafka Streams API) Event Management (Kafka Admin API, Metrics) Event Channel (Topic-partition) Event Processor (Kafka Processor API)	Event Management (CloudWatch)	Event Management (Cloud Monitoring)
Architectural Interactions	Event-based interaction model	Initiator: Producer of records Destination: Indirectly addressed	Initiator: Event source Destination: Indirectly addressed	Initiator: Event provider Destination: Indirectly addressed
	Subscription mechanisms	Channel-based, Subject-based, Type-based	Content-based, Pattern-based	Channel-based, Subject-based
	Transmission models	Pull	Push	Push
	Topologies	Broker	Broker	Broker

platforms. This lack of a commonly accepted specification for defining the attributes of exchanged events might limit the interoperability of EDA solutions developed based on the platforms currently offered by the industry.

For the event exchange format, two of the models described in the literature are used in the platforms: events are objects in Kafka, while semi-structured documents are used in EventBridge in the form of JSON. For the event content as described in the literature, none of the platforms requires an occurrence number to be attached to or included in the event. Kafka records do not include an attribute for the event source; i.e., events are anonymous and it is not possible for the consumers to know who the producer was by reading the event. Eventarc does not specify any attribute that the event must contain.

When it comes to the information carried by the event, EventBridge imposes the use of two separate fields *detail-type* for the event name or identifier, and *detail* for full details so the level of details in the case of EventBridge is always a full details event and not a simple notification about the occurrence of it. None of the studied solutions supports the distinction between the state change and the state refresh event.

B. EDA Components

From the set of architectural components of EDA that are described in the literature, only a subset is implemented by the platforms. For the event development tool, despite the good coverage of Kafka's APIs on an important part of the roles that EDA components have to fill, they do not offer a ready-to-use Schema Registry to insure consistency and co-evolution of the event specification and the way consumers and producers handle it.

None of the platforms provides a method to create an event pre-processor. In Kafka, developers can create and attach interceptors to producers, these interceptors can mutate events before they are published to the cluster.

An important aspect of the implementation of the ENS by the three platforms is their ability, not just to route events to subscribed consumers, but to hold events up to a certain time limit when the consumer is not available in order to guarantee the temporal decoupling between producers and consumers.

The Event Processing Engine is partially satisfied by Kafka Streams API where multiple streams of events can join together. This provides data processing capabilities. This might suggest that what we call EDA platforms might serve different purposes depending on the use cases. We can observe that events are used in two different ways by these platforms. (i) Events as real-time data carriers, used when we want to detect and react to simple and complex business scenarios as soon as they happen. This is described by Michelson [10] as the real-time *flow of information*. (ii) Events as a communication form, where we focus on the asynchronous interaction between applications or services, this is described by Michelson [10] as the real-time *flow of work*.

C. Interactions of EDA Components

While all of the platforms help applications implement an event-based interaction, Kafka does not allow events to actively invoke services due to its lack of support for the push transmission model.

All subscription mechanisms that are defined in the literature are supported in the platforms but each platform supports a different subset of them. No other subscription mechanisms were found in the platforms.

Looking at the topologies described in the literature, none of the platforms allow developers to implement the mediator topology as it's not possible to define a central agent (*e.g.*, as part of the ENS) to transform incoming events to an ordered sequence of different events routed to different consumers. Eventarc is designed for microservices communication, however, in academic studies, EDA is considered a separate pattern from service-oriented and mircoservices architectures [10, 11].

D. Threats to Validity

Some limitations of this exploratory study are the low number of platforms we analyzed in addition to the limited set of academic studies from which we extracted the elements of event-driven architectures. We mitigated these threats by relying on statistics published by reporting companies and research groups when we selected the platforms and their providers, and by selecting studies that focus on providing a clear definition of the elements of event-driven architectures which is a criterion we consider to be the most important with regards to the objective of the study. Another threat is related to the coverage of the keywords used when searching for the academic studies. This should be mitigated in future work by a more systematic review of the EDA literature.

VII. Conclusion

In this paper, we presented an exploratory study on the gap between the theory around EDA and the solutions offered by the industry. We started by selecting the studies that provide a description of the elements of the event-driven architecture and some platforms that are designed to support it (Apache Kafka, AWS EventBridge, GCP Eventarc). We then analyzed the platforms by checking their realization of each element described in academia. Finally, we discussed the differences between what we observed in the industrial solutions and what we found in academia.

We found a lack of a common understanding of what event-driven architecture is and a clear method of how it should be designed. The differences in the industrial platforms indicate the absence of standardization in the specification of events. We also found that EDA has two main use cases, one is about real-time processing of data, while the other is about reactive communication between components of a software architecture.

Future work includes a more structured search procedure for EDA-relevant studies in academia and the exploration of more industrial solutions (*e.g.*, Azure EventGrid) to better understand the differences they have and provide guidelines on where to use each one. Moreover, we plan to design and implement an EDA application for an IoT case study using different platforms. This will allow to further evaluate the platforms and their adequacy for specific IoT applications considering their functional and non-functional requirements.

Acknowledgment

This research was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) RGPIN-2022-03766.

References

[1] D. S. Rosenblum *et al.*, "A Design Framework for Internet-Scale Event Observation and Notification," *SIGSOFT SEN*, vol. 22, no. 6, p. 344–360, nov 1997.

[2] S.-C. Liu *et al.*, *Event-based neuromorphic systems*. John Wiley & Sons, 2014.

[3] T. Delbrück *et al.*, "Activity-driven, Event-based Vision Sensors," in *ISCAS*. IEEE, 2010, pp. 2426–2429.

[4] L. Li *et al.*, "Event-based Network Consensus with Communication Delays," *Nonlinear Dynamics*, pp. 1847–1858, 2017.

[5] A. Theorin *et al.*, "An Event-Driven Manufacturing Information System Architecture for Industry 4.0," *IJPR*, vol. 55, no. 5, pp. 1297–1311, Mar. 2017.

[6] A. Carzaniga *et al.*, "Issues in Supporting Event-Based Architectural Styles," in *ISAW*. ACM Press, 1998, pp. 17–20.

[7] G. Mühl *et al.*, *Distributed Event-Based Systems*. Springer-Verlag, 2006.

[8] A. Hinze *et al.*, "Event-based applications and enabling technologies," in *DEBS*. ACM Press, 2009, p. 1.

[9] V. Cristea *et al.*, "Distributed Architectures for Event-Based Systems," in *Reasoning in Event-Based Distributed Systems*. Berlin, Heidelberg: Springer, 2011, vol. 347, pp. 11–45.

[10] B. Michelson, "Event-Driven Architecture Overview," , Boston, MA, Tech. Rep. 681, 2011.

[11] M. Richards, "Event-Driven Architecture," in *Software Architecture Patterns*, 2015.

[12] S. R. Goniwada, *Cloud Native Architecture and Design: A Handbook for Modern Day Architecture and Design with Enterprise-Grade Examples*. Apress, 2022.

[13] K. Strothmann and M. Bachmann, "Getting Started with Event-Driven Architectures," in *Event-Driven Architecture with SAP*. Rheinwerk Publishing Inc., 2021.

[14] L. Bass *et al.*, *Software Architecture in Practice, 4th Edition*, 1st ed. Addison-Wesley Professional, 2021.

[15] A. Brandolini, "Introducing EventStorming," 2013.

[16] O. Etzion and P. Niblett, *Event Processing in Action*. Manning Publications Co., 2010.

[17] Cloud-Native Computing Foundation. (2018) Cloudevents specification. [Online]. Available: cloudevents.io