Red Hat

Detail

Event-driven architecture for a hybrid cloud blueprint

Table of contents

Introduction ... 2

API calls or event-driven systems? ... 2

Benefits of event-driven architectures ... 4

Moving toward event-driven systems ... 5

- Discovering your needs ... 5
- Communication types ... 6
- Event-driven implementation patterns ... 7
- Common event integration patterns (EIPs) ... 8

EDA use cases ... 9

Maturity level of EDA ... 10

Summary ... 11

facebook.com/redhatinc
@RedHat
linkedin.com/company/red-hat

redhat.com

Red Hat

Introduction

Organizations are increasingly adopting cloud computing to gain technical and business advantages. As part of this shift, application architectures are evolving to a model of distributed, modular, and portable components that can be easily deployed and run across cloud infrastructure, both on-premise and on public or private clouds. This hybrid cloud approach reduces infrastructure costs and increases the operational efficiency of applications. In this environment, strategic use of compute resources and effective coordination between application components are critical architecture design goals.

Event-driven architectures (EDAs) are ideal for hybrid cloud applications, enable multicloud, and span across different geographies. They result in architecture components that are better suited for real-world use cases, where decoupling of runtime and protocols are required to achieve fine-grained scaling. Cloud and container distributed systems are best suited for this architecture approach.

This detail is for software architects and developers. It covers key considerations for designing EDA strategy, common uses cases, and technologies that can help along the way. Red Hat® Middleware Portfolio provides a set of capabilities to better craft and build efficient and resilient EDAs.

API calls or event-driven systems?

Synchronous application programming interfaces (APIs) dominate the cloud-native world. Communication through Hypertext Transfer Protocol Secure (HTTP/S) is simple to implement and makes it easy to manage and trace a particular request. But there are limitations in areas like multicasting requests, circuit breaking to isolate failures, and decoupling services. EDAs address these limitations because they provide asynchronous communication and reactive programming approaches for effective fault tolerance, metrics in a highly distributed microservices architectures.

Figure 1 is a high-level logical architecture of a typical system that uses synchronous invocations as its main mode of communication. It has a well-established structural standard to set up contracts between systems. The pipe flow between calls is mostly in sequential order, so calls are easier to trace. If a transaction that spans multiple services needs to be handled, a Saga pattern can be implemented to provide data consistency in this highly distributed environment.

Red Hat

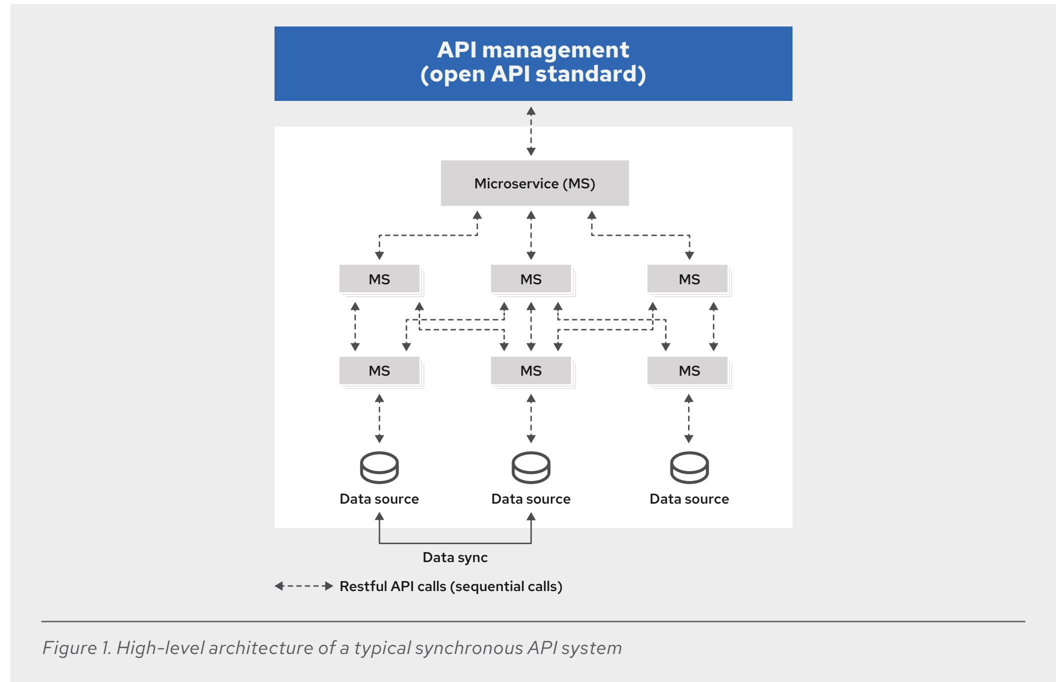

Figure 1. High-level architecture of a typical synchronous API system

Figure 2 shows a high-level logical architecture of a typical asynchronous system. Events are broadcast to applications, and each application responds to the events. Applications tend to be more modular because they depend less on any specific systems than events. This architecture is designed to handle larger volumes of streaming events and uses event sourcing as a way to handle transactions.

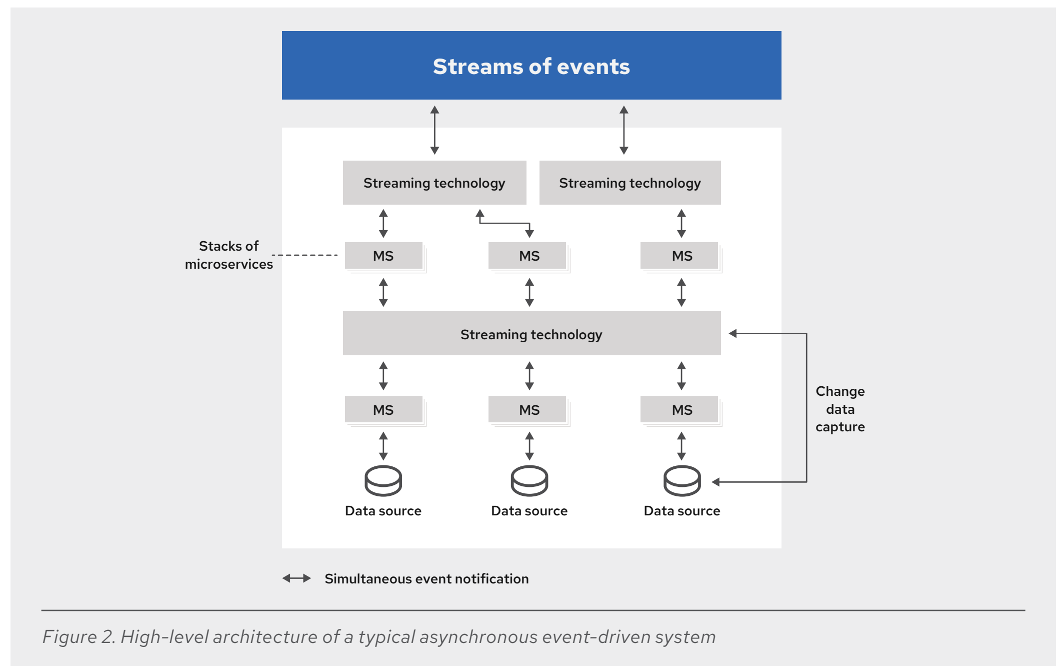

Figure 2. High-level architecture of a typical asynchronous event-driven system

Table 1 compares the two architectures.

Table 1. Synchronous API systems vs. asynchronous event-driven systems

	Synchronous API	Asynchronous events
Contract	Open API standard (Swagger)	Event data (Registry)
Security	API management, service mesh	Sender/In receiver
Transactions	Saga pattern	Event sourcing
Ordering	Guarantee sequential/Simultaneous	Simultaneous/Partially sequential
Data replication from datasource	Service call/Extract, transform, load (ETL)	Change data capture (CDC)
Tracing	Istio	N/A

There is no right or wrong answer between event-driven systems or systems based on API calls. It depends on the nature of the application. Both implementations co-exist, and they work best together in various use cases.

Whether an event-driven solution is using synchronous or asynchronous communication, events need to be consumed and processed by custom logic to take appropriate business actions. A group of related events could consequently be used to identify and trigger higher-value business events, which can then be fed into human-centric processes and serve as input to artificial intelligence and machine learning (AI/ML) systems. Organizations can use Red Hat Integration and Red Hat Runtimes to implement the business logic to handle events and generate higher value business events. Red Hat Integration is a cloud-native integration platform, and Red Hat Runtimes is a collection of well-established and modern frameworks and runtimes. For events that trigger business decisions or human interaction, organizations can use Red Hat Process Automation, a set of offerings for decision management and business process automation.

Benefits of event-driven architectures

EDAs are based on asynchronous, nonblocking communication and can release resource usage rather than wait for a response to get back. This is specially relevant for cloud- and container-native development, which demand high agility and flexibility from scalable, distributed cloud microservices environments. In container-native application development, the imperative programming pattern has shifted when developing container-native applications. Stateless microservices allow the system to become reactive. Domain-driven design (DDD) is also introduced, which enables systems to become loosely coupled.

EDA greatly enhances this decoupling from the communication standpoint by using sender/publisher and receiver/subscriber objects. Because multiple subscribers can receive events simultaneously, the system can have lower latency and higher throughput with the right events medium.
By subscribing to particular events, the system can react in real time and become more aware of the surrounding incidents.

Red Hat

Organizations can optimize cloud resource consumption, drawing on the following benefits from EDA:

- **Cloud-native nature:** EDAs are designed to work perfectly in the distributed world because cloud-native services are running separately on nodes or servers in the cloud.
- **Parallel processing (asynchronous):** On-linear calls allow faster data processing and can trigger a wider range of service and application processes waiting for events. The ability to scale processes out instead of up helps optimize resource usage and allows greater capacity.
- **Fault tolerance:** EDA emphasizes failure isolation and natural circuit breaking without the requirement of a separate implementation. The component decoupling stops failures from propagating to the rest of the system or application.
- **Optimization for real-time responsiveness:** The ability to process multiple events allows the system to respond before critical and problematic events happen rather than analyzing data too late.
- **High observability and extensibility:** Describing and implementing systems in terms of the events they produce (event-first design) makes it easier to observe the behavior of the system and extend it to meet future requirements without re-architecture.
- **Deployment across the hybrid cloud:** Components of EDAs can consume and generate events across cloud boundaries, taking advantage of individual cloud capabilities while remaining compliant with policy. Examples include requirements to keep data in-house while keeping computing resources close to the customer.

Moving toward event-driven systems

Discovering your needs

Before trying to make your system event driven, you should consider the following questions. They can help you understand the type of technology and solution that best fit your situation. We will elaborate on how the answers relate to the different types of EDAs later in this section.

- **Protocols:** What protocol is asynchronous data coming from: MQ Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP), Hypertext Transfer Protocol (HTTP), or others?
- **Security:** Who is authorized to publish events? Who has access rights to the events?
- **Ordering of events:** Is the sequence and order of the events important?
- **Throughput of events:** What is the expected throughput of the events within one second?
- **Persist events:** Should all the events be logged into persistence storage?
- **Broadcast route:** Where in the system should the events be broadcast? What is the format to broadcast? Does some information need to be shielded or obfuscated?
- **Issuer of events:** Is it important to know the issuer of the events?
- **Cross datacenter:** Will the event be broadcast to another datacenter?
- **Transaction:** Will this trigger a transaction? How is the rollback being handled? Does it require compensation?
- **Validation:** Is an extra step needed to validate the format or content of the events?

- **Additional enrichment:** Will there be a need to obtain data from other services or sources? Does the event need to be enriched with data from other sources?
- **Size of return data:** When reacting or responding to an asynchronous call, what is the size of the data?

It is important to note that the nature of cloud-native application development can also change the behavior of how events and commands are broadcast. One example is that decentralized microservices deployed across clouds can make simple transactions complicated. Another example is that the rise of Internet of Things (IoT) technology has significantly increased the volume of data transfer across networks. And if we look further down the road, on-demand resource optimization – which plays a huge part in serverless environments – is also based on EDA. This event-ready approach will set you on the right path for future extension.

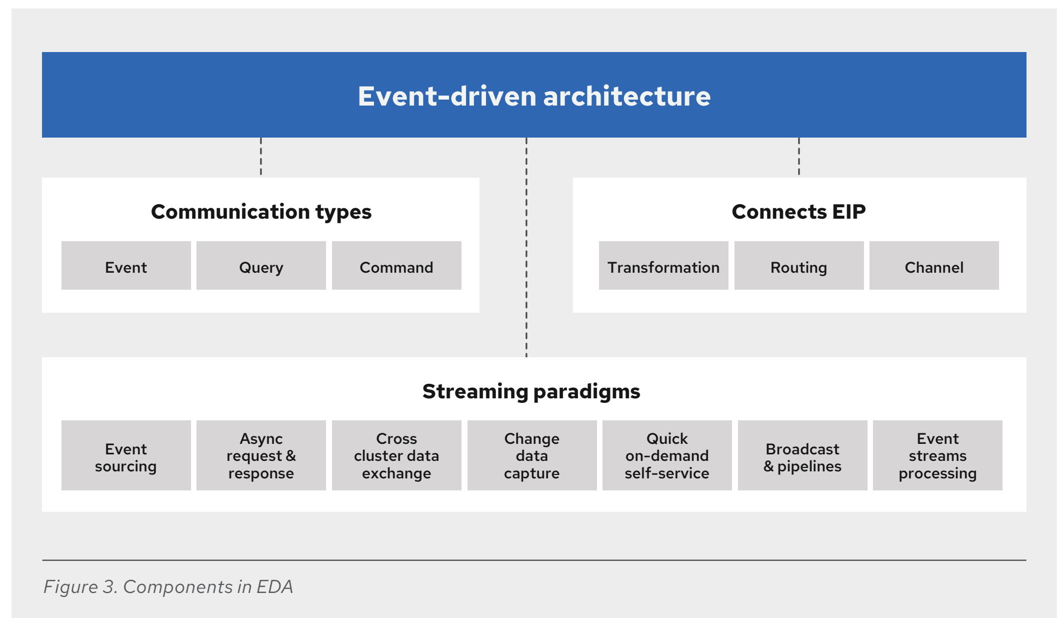

Figure 3. Components in EDA

Communication types

In an EDA, various events broadcast across the system. But in looking at all the asynchronous data, valuable communication types become evident. Below are some common communication types that organizations should consider when designing a proper cloud-native EDA. Organizations should handle each communication type differently to achieve the best performance and flexibility. Technology and solutions used for implementation also should depend on the type of communication used.

Red Hat

Event
An event is an immutable state and value of a particular entity that occured during operation among services. An event contains the time it happened and the issuer, sometimes with a key identifier. In addition, an event can be used to notify or replicate data across multiple surrounding systems, possibly across clouds, in one go. Normally, events are broadcast to make a system more aware of what is happening. There is no need for consumers to subscribe or listen to events.

Command
A command is an asynchronous form of a remote procedure call (RPC) with instructions to tell the recipient what to do, and it may cause a change of state in the system. A command will normally expect a return of its status. Sometimes complex transactions may occur as part of a command, so the ability to roll back and time out is crucial when dealing with this type of communication. With respect to the required parameters in a command object, at least one consumer needs to be present to process the command.

Query
Similar to commands, queries expect a response of returned results. However, queries do not cause any change of state in the system. They have read-only access and allow multiple duplicate calls. Queries typically require fast response time, and cache stores are used in many cases to accommodate this need. At least one consumer needs to be present to return the query results.

Event-driven implementation patterns

The amount of generated data has increased dramatically from various sources, including Software-as-a-Service (SaaS), internal systems, databases, IoT technology, and microservices. All this data growth demands higher throughput and faster performance for information processing.

Red Hat Integration, which contains AMQ streams (based on Apache Kafka), provides capabilities to fulfill the following event-driven patterns.

Event stream processing
These are systems that detect and react to critical conditions by querying a continuous data stream within a small time window. AMQ streams can be used to manage continuous data streams. Its ability to handle large throughput allows enterprises to become more aware, as more data from various sources is fed into the system. The ability to replay any stored events in sequential order will make the system more robust, and it can connect to other AI/ML systems for further analysis and behavior observation.

Broadcast and pipelines
The flow of data has multiple stages and consumers, forming a pipeline, or broadcast, of information. Data is sent and received via AMQ streams topics with some guarantee of order, high throughput, and recoverability by default. The receiver services can continue to enrich or transform the data, passing it onto the next step for other consumers.

Event sourcing
Maintaining data consistency across distributed systems is challenging. Event sourcing is a pattern that allows a system to log data changes in timed order. Any service can then replay the log to determine the current data state. The change logs can be used for dealing with transactions between the distributed microservices. The rapid change of state and the amount of data can be logged into AMQ streams topics and store all change state events in the order of occurrence, which is crucial for processing and maintaining the logs for future reference.

Change data capture
Relational databases are commonly used for data storage. The ability to capture events when data changes in the store – and auto populate these events for other services that need the latest state of the data – is crucial to keep data consistent. For this use case, open source distributed platform Debezium can capture changes in a relational database, and AMQ streams topics can then distribute data changes across the system.

AMQ streams provides a platform for all the above streaming event types, both on-premise and on the cloud, including those for applications with multicloud and hybrid cloud requirements.

Asynchronous request and response
Most human interaction with a system or external API calls needs a response. These bidirectional exchanges of data need multiple channels for asynchronous communication because an event can trigger sequence or parallel processing steps, each performing a specific function. In addition, guarantee of delivery, which is handing the result to the right request, becomes important. In such cases, AMQ broker provides a solution for this asynchronous request and response.

Quick on-demand self-service
Messaging-as-a-Service (MaaS) or quick, ad hoc integration services provide the ability to set up, remove, and maintain a small integration quickly and easily for developers without requiring expensive human resources or repetitive maintenance work. Red Hat Integration allows developers to create AMQ streams and AMQ broker topics on demand, and it provides a framework and platform to develop integration applications with ease.

Cross-cluster data exchange
Moving data within the same cluster of containers is easy. However, for hybrid cloud deployments, applications must be able to reliably move messages across container clusters without requiring too much detail about the moving data, or needing extra configuration. Lightweight message routers, such as AMQ interconnect, can fulfill the requirements of cross-cluster data exchanges.

Common event integration patterns (EIPs)

Applications often need to integrate with other applications and systems within an organization. So far, we have noted the types and patterns of EDA. But the content of the data is equally critical to consider. Even in a single domain, it is not likely that only one form of data needs to be transferred. For instance, in a logistics system, it is not just "deliver order" data that is required, but also schedules, warehouses, trucks, and workers. So how do we deal with all this integration complexity in EDA? Enterprise Integration established and systemized solutions for common integration problems.

Here are three patterns that are commonly used in a EDA microservices environment:

- **Data transformation**
    - **Message translator, channel adapter, and canonical data model:** Translates data formats from one form to another. It is primarily found in controller and dispatcher microservices where data must be translated from various types of clients into the entity of domain-driven design models. Or, it can be the other way around, and canonical unified entity models may need to transform into client-specific formats.
    - **Content enricher and content filter:** Enriches an existing message by appending it with data that has been retrieved from an external data source. It could also remove or simplify unwanted data in a message.

- **Data routing**
  - **Content-based routing and message routing:** The most common pattern consists of routing a message to the right recipient(s) based on its content or header.
  - **Splitter and aggregator:** This pattern breaks the data into small chunks for microservices, or combines different pieces of data for analytics and machine learning systems.
  - **Pipes:** Messages flow through pipes, which perform a specific function. As a filter finishes working on a message, it pushes the message into another pipe or service for further processing. Flow can be synchronous or asynchronous.
- **Channels**
  - **Pub-sub:** A publisher publishes events, and subscribers subscribe to them. A subscriber can react in real time upon the receipt of an event. Each subscriber gets its own copy of the message.
  - **Data type:** Consists of using a separate channel for each type of data. The sender selects the correct channel, and the receiver will know the data type based on the channel it was received on. This is a useful mechanism when proper contract definition for the event data type is not available. Applying this pattern in combination with a checking filter results in decreased data errors.

From different communication types and event behaviors described above to the implementation pattern, Red Hat Integration provides a broad set of capabilities to help organizations better craft their event management strategy and build efficient and resilient EDAs.

EDA use cases

Here are some real-world examples of how EDA is applied.

- **Reactive notification:** EDA allows events to quickly reach the targeted service and systems to react upon changes in state. These changes can be pushed to any AI system or can be bubbled up to other applications. Organizations can quickly respond to incidents, pre-empt potential issues, identify trends, and provide better services to customers.
- **Command query responsibility segregation (CQRS):** CQRS separates the input/update and query services with different access interfaces. EDA is an ideal approach to implement CQRS, as you can easily separate the query and result models into different channels. Caching can be introduced to speed up the read/output effort.
- **Behavior capture:** Using the large amounts of data that they capture, organizations can extract and track business behavior to predict potential future behavior and inform strategy and planning.
- **Auditing:** All the changes of state in a system can be stored at every step in log files. Change logs provide a valid capability for auditors to trace system changes at any point in time.
- **Complex event processing:** Multiple and/or complex events combined in a certain business criteria can trigger a business process or decision.
- **Cache store:** The logging of events in an in-memory cache can become a temporary datastore for services to retrieve the latest state without needing to access the actual back-end datastore.
- **Streaming between datacenters:** Large enterprises often have multiple clusters of clouds across various geographies for high availability, redundancy, backup, or disaster recovery. Events can be broadcast across multiple geographies to allow instant inquiries and make real-time decisions.

Maturity level of EDA

Figure 4 summarizes the five levels of EDA maturity:

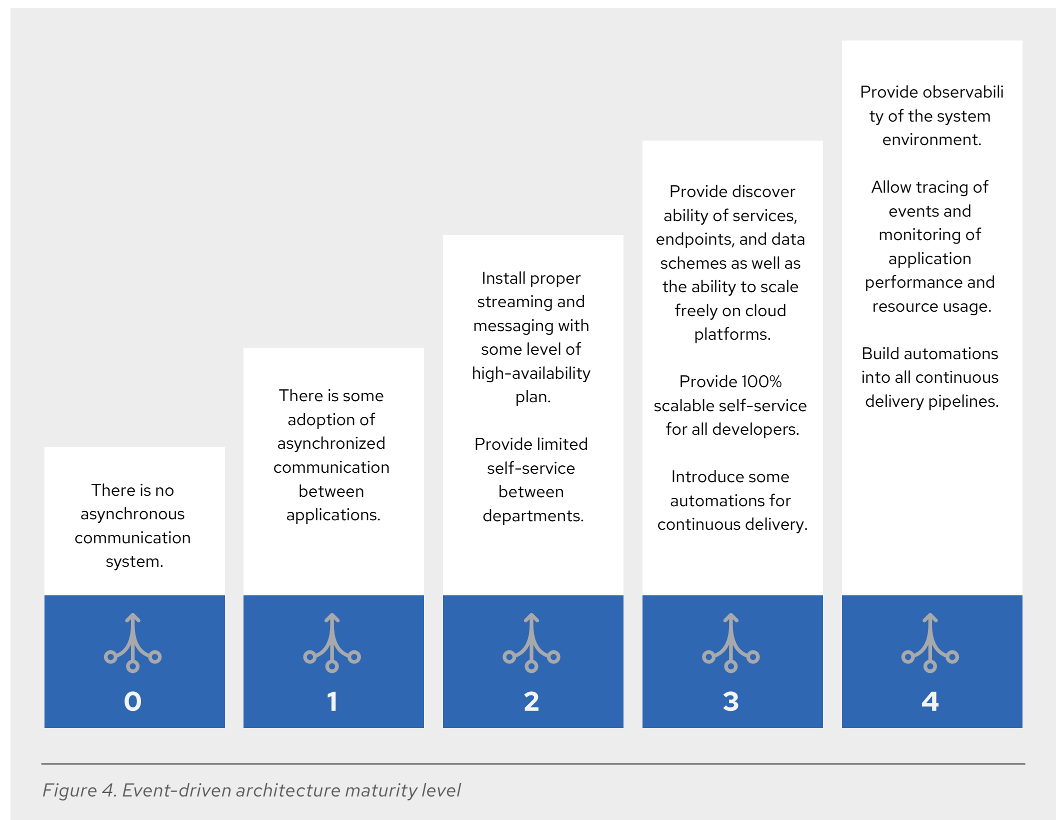

Figure 4. Event-driven architecture maturity level

Organizations at Level 0 have no applications and systems that use any type of asynchronous communication.

At Level 1, an organization starts to modularize services and adopt asynchronous calls between each application. Data operation teams provide infrastructure and common application services installation for application teams.

At Level 2, properly created streaming and messaging solutions are implemented between services and applications, and high availability is planned as part of the configurations. There is limited self-service for the development team to speed up delivery time.

At Level 3, EDAs provide scalable self-service both on-premise and in the cloud for resilient and quick delivery. Discoverable endpoints and services with data scheme description eliminate silos and deliver better communications. Automation is introduced for continuous delivery.

At Level 4, beyond having well-defined asynchronous communication between systems, an EDA also provides observability and events and message monitoring throughout the environment. Events can be traced to indicate if there are any problems. Automation is built in as part of the continuous delivery pipeline — from setup, to moving to different environments, to scaling up and down, through event phaseout.

Red Hat Integration offers capabilities that can help you mature at your own pace in your adoption of EDAs.

Summary

Microservices and hybrid cloud applications have become more complex primarily due to synchronous communication between components. Even though great work has been done in tooling and frameworks, not all use cases are a good fit for synchronous communication. Because of these inherent limitations, events and asynchronous communication are the ideal alternative.

EDAs result in systems and applications that are better suited for real-world use cases, where decoupling of runtime and protocols is required to achieve fine-grained scaling. Cloud and container distributed systems are best suited for this.

Red Hat Integration provides a set of capabilities for different types of communication patterns. When organizations use it in conjunction with Red Hat Runtimes and, eventually, Red Hat Process Automation, they can better craft and control event management to build efficient and resilient EDAs.

About Red Hat

Red Hat is the world's leading provider of enterprise open source software solutions, using a community-powered approach to deliver reliable and high-performing Linux, hybrid cloud, container, and Kubernetes technologies. Red Hat helps customers integrate new and existing IT applications, develop cloud-native applications, standardize on our industry-leading operating system, and automate, secure, and manage complex environments. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500. As a strategic partner to cloud providers, system integrators, application vendors, customers, and open source communities, Red Hat can help organizations prepare for the digital future.

facebook.com/redhatinc
@redhat
linkedin.com/company/red-hat

North America
1 888 REDHAT1
www.redhat.com

Europe, Middle East, and Africa
00800 7334 2835
europe@redhat.com

Asia Pacific
+65 6490 4200
apac@redhat.com

Latin America
+54 11 4329 7300
info-latam@redhat.com

Copyright © 2020 Red Hat, Inc. Red Hat and the Red Hat logo are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.