CHAPTER 6

Event-Driven Architecture

Event-driven architecture is not new; it has existed since the Unix operating system came on the scene.

Event-driven technology enables high-speed, asynchronous, machine-to-machine, or program-to-program communication with guaranteed, reliable delivery of events. Machines or programs exchange data each other. The queues or channels are the pathways that connect both the sender and the receiver. A sender or producer is a machine or program that sends events by writing data to the queues, and the receiver or consumer consumes the messages and sends an acknowledgment to confirm the message is received. The data exchange between the sender and receiver can be an object, JSON, XML, and byte. There are two ways to exchange information; the first is a point to point, and the second is publish and subscribe.

This chapter provides insight for anyone considering implementing an event-driven architecture; you should have a basic idea of software architecture, design, and development. There are plenty of books and whitepapers available on event-driven technology; I am not going to duplicate that information here. Instead, I will cover event-driven technology in the context of cloud native and real implementations as well as the problems you may face during implementation.

In this chapter, I will cover the following topics:

- What is event-driven architecture?
- What are events?
- Characteristics of event-driven architecture
- When to consider event-driven architecture
- What is complex event processing?
- Role of event-driven in microservices
- Case studies

S. R. Goniwada, *Cloud Native Architecture and Design*, https://doi.org/10.1007/978-1-4842-7226-8_6

Evolution of Event-Driven Architecture

Most applications in enterprises are required to interact with each other by transferring data. In 1971, File Transfer Protocol (FTP) was introduced to transfer data across applications and machines on Network Control Program (NCP). In the early 1980s, the TCP/IP protocol was introduced to draw communication across systems. Later, applications can use a shared database, located in a single physical box; therefore, no data has to be transferred from one application to another. After the introduction of TCP/IP, applications were developed to start exposing some of their functionality so that they could be accessed remotely by other applications via a remote procedure. The communication occurs in real time and is synchronous with high coupling and low cohesion.

The FTP, TCP/IP, and Remote Procedure Call (RPC) protocols are slow and unreliable, and the interaction between applications needs to support the evolution of applications and keep pace with changes in the applications being connected. To overcome the slowness and reliability, the messaging infrastructure was introduced. Messaging is more immediate than FTP, better encapsulated than shared databases, and more reliable than RPC.

Tightly Coupled World to Loosely Coupled World

The messaging in applications and across applications in an enterprise became popular with the maturity of message brokers and message-oriented middleware (MOM). Messaging is a technology that enables high-speed, asynchronous communication with reliable delivery. Applications communicate by sending data called *messages* to each other over a pipe known as a *queue*.

Messaging capabilities are typically provided by a separate software system called *message brokers*. A *messaging system* manages the way the database handles the data persistence. Just like a database administrator (DBA) manages the database, the messaging administrator manages the messaging system. The messaging system coordinates and manages the sending and receiving of messages across systems. The main task of the message system is to manage reliability.

The primary features of message queues are storage, asynchronous messaging, and routing. The message queues store messages or some type of buffer until they have been either read by a consumer or expire or explicitly removed from the queues. The main advantage of a messaging system is loose coupling. The receiving application may not be available for a few seconds to receive messages, or the network is not available, but the receiving application can receive messages once it is available. The message broker keeps retrying to send messages to the receiving applications. This allows for asynchronous nonblocking communication that provides a higher level of tolerance against failure. Enterprise messaging technologies such as IBM MQ, Active MQ, Rabbit MQ, Zero MQ, etc., can be used to decouple your applications for the reliable and guaranteed delivery of messages.

Message queues allow subscribers to subscribe to a message from the message provider. Queues usually manage some level of the transaction to make sure the desired action is executed before the message is removed from the queues. The messages are delivered at least once. Even if the consumer is not available, the queues try to deliver by using a retry configuration. The queues send messages to dead-letter queues after a failure to deliver messages to consumers or the messages will expire. You can use a point-to-point or publish-subscribe model for communication across applications or machines or programs.

We looked at message queue systems, and we saw that message queue systems are used extensively for interapplication communication.

Message Broker World to Event World

Over the years, there has been an evolution of microservices and real-time integration with lightweight data interaction. We are moving from static to dynamic by accumulating data in data lakes to enable data in transit and keep track of it while it is moving from place to place. The shift to event-driven architecture means moving from a data-centric model to an event-centric model. In an event-driven model, the event is a more important component, whereas with service-oriented architecture (SOA) or message queue platforms, the highest priorities were to not lose any data while transferring, to deliver at least once to the consumer, and to have rest of the process leave it to the consuming application to take care of the data. With event-driven architecture, you can address the challenges of SOA and MQ, and the priority is to respond to events as they occur. The older the events, the less valuable they are.

Along with the processing of events, there is a need to persist a record and allow the application to process historical data and real-time data without the threat of deletion by a broker. All these characteristics are not possible in message brokers; there needs to be a streaming platform. The streaming application addresses one-event-at-a-time processing with nanosecond latency with stateful processing and joins and the aggregation of messages. Event streaming platforms can be used for both simple and complex event processing, allowing event consumers to process and perform actions based on the result.

Today, event brokers offer efficient and scalable publish/subscribe event distribution based on routing-labeled events and not just messages to a queue. They support the following:

- Dealing with a consumer that is too slow or offline by managing the state of events on the fly
- Decoupling which data to send to which consuming applications, getting it there reliably, and managing changes to this set of consumers over time
- Providing services such as priority delivery, load balancing to consumers, and more

Event brokers make cloud native services simpler and allow a more real-time, responsive, scalable, efficient, and fault-tolerant system.

There are various event streaming platforms in the industry such as Apache Kafka and Confluent, AWS Kinesis, Spark, Google Data Flow, IBM Cloud Park, Lenses, Hazelcast Jet, IBM Event Streams, SAS Event Streaming Process, Solace, and Azure Event Hub.

In the subsequent section, we will provide a step-by-step approach for designing and implementing event-driven architecture.

Event

Anything that occurs in enterprises or systems is an *event*, such as a customer request, batch update, data change, an employee swiping a credit card, a customer buying a product in a retail ecommerce application, someone checking in for a flight, etc. These events exist everywhere and are constantly occurring, and it does not matter what kind of application it is or what industry it is in. Events are pervasive across any business. There is value in knowing about an event and being able to react to it quickly. The more quickly you can get information about events, the more effectively your business can react to them. An event is separate from a message because the event is an occurrence, and the message is the carrier of the information that relays information about the occurrence. In an event-driven architecture, an event likely commands one or more actions or processes in response to its occurrence.

An event is not the same as an event notification, which is a message or notification sent by the system to notify another part of the system that an event has taken place. The source of an event can be internal or external inputs.

There are two types of events:

- Business events
- Technical events

Business Events

The business events are typically what we care about from a functional perspective. We can derive these from the event storming exercise of domain-driven design. These events are not always initiated externally but created by other business events. For example, an order-placed business event creates an order-shipped business event. Ideally, we should keep these business events around in perpetuity.

The following are examples of business events:

- The customer swipes their credit or debit card at the retail outlet.
- Employees enter the office premises by swiping an ID card.
- A bill is paid.
- An order is placed.
- The order management system sends details to update the inventory system.
- The source data changes for replication to the target operational data store (ODS).

Technical Events

The technical events are derived from business events; typically many technical events can be generated from a single business event. These events are used to communicate between services or systems. These events are technical in nature and are the only trigger to perform a specific action.

The following are examples of technical events:

- Database updated
- File uploaded successfully
- Email notification sent

Each of these events is like a command of one or more actions such as the authorization of payments, authorization of the employee entry, an update, a reduction of inventory, etc. The response is to log events for monitoring and analytics purposes.

Processing an Event

Events are recorded to an event log, as shown in Figure 6-1, and then processed by one or more services. Events do not "fall off" of the log; instead, they are persisted.

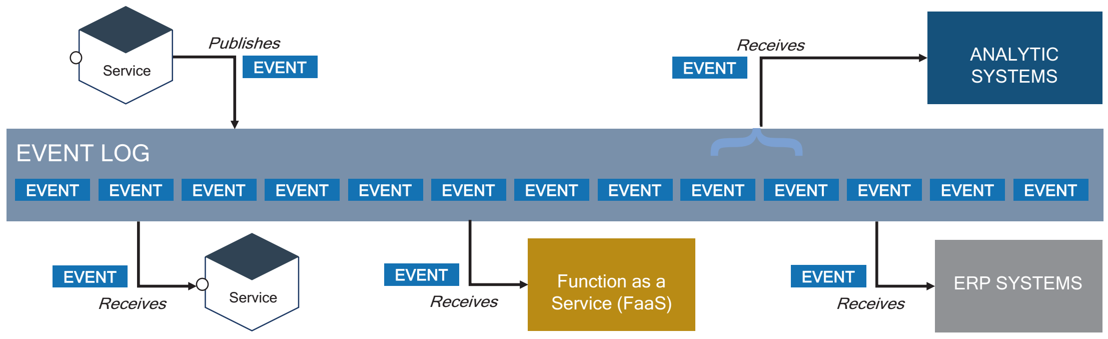


Figure 6-1. *Event processing in an enterprise landscape*

In event processing, the events are persisted on an event log, in a uniform schema, and events are typically organized by topics. Events of different types should not exist on the same topic. For example, customer payment and customer cart data are different topics, even though they relate to customer behavior like adding wish lists, etc. If you are interested in the order of events by customers across systems, consider creating a third topic of "customer actions" that relates only to the actions performed on a customer, discarding the rest.

Consider not defining the listeners in the first place. An event can have multiple listeners, and they may not all exist at the time of the event creation.

In Figure 6-1, the service publishes an event to the event log; the service, FaaS, ERP system, and analytics systems subscribe to an event from the log.

Event Handling in Domain Context

Events can be used for interdomain or intradomain communication. For example, as shown in Figure 6-2, in an ecommerce application, the ecommerce web and mobile applications make up one domain, and the back-end applications are another domain. If you want to send events between these two domains, you use business events.

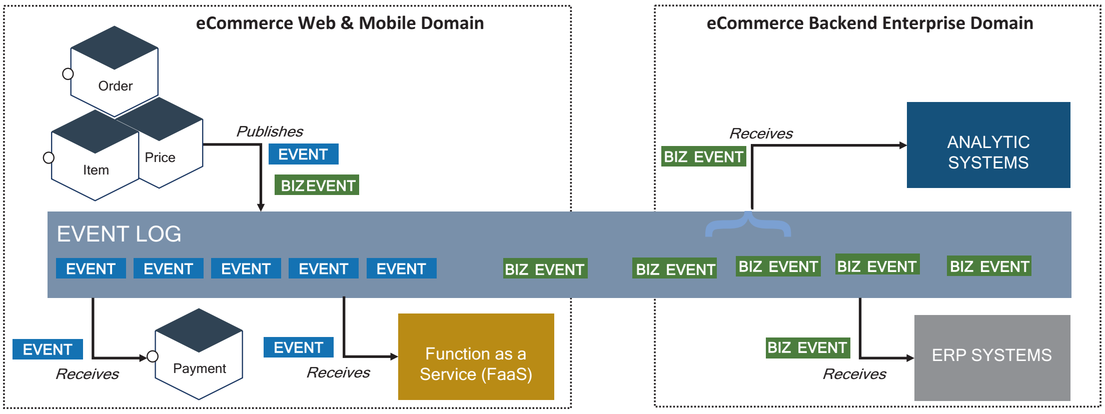


Figure 6-2. *Events across domains in an enterprise*

Within the same domain, the events can be technical or business events. Across domains, only business events are relevant. If you are using technical events, such as notifications, database updates, or requests received, across domains, then that could be a sign of ill-defined domains or a distributed monolithic system; in that case, it is not a cloud native architecture.

Event Governance

The following are the best practices for using events in an enterprise:

- Organizations should strive to make events discoverable to subscribers.
- Events should have a standard envelope that encloses them such as publisher ID, tracer IDs, etc., so that the events can stand alone as the systems evolve. I suggest using cloud event specifications.

- Events should be as small as possible, encompassing the data needed for that event. Topics should contain only one type of event. Smaller events and more topics are better suited to a distributed system.
- Within a domain, the team should design how to introduce new events, but across domains, you need to standardize namespacing and require a governance team to manage the events or they become uncontrollable.

What Is Event-Driven Architecture?

An event-driven architecture (EDA) is a distributed, asynchronous architecture that integrates applications and components through events. It is a combination of an event producer and an event consumer; an event producer detects an event and represents the event as a message object. After an event detection, it is transmitted from the event producer to the consumer through a channel. The event processing platform processes the event asynchronously and informs the event consumer about the occurrence. The event processing platforms will execute the correct response to an event and send it to the right consumers. For asynchronous communication, the consumer and the subscriber do not need to know or be aware of each other. EDA can be relatively complex given its inherent characteristics of asynchronous, distributed processing, issues that may occur due to a lack of responsiveness, failure of mediators, and brokers.

How Does Event-Driven Architecture Work?

As shown in Figure 6-3, event-driven architecture consists of four parts.

- Event producer/publisher
- Event consumer/subscriber
- Event broker or routers
- Event persistence (part of platform)

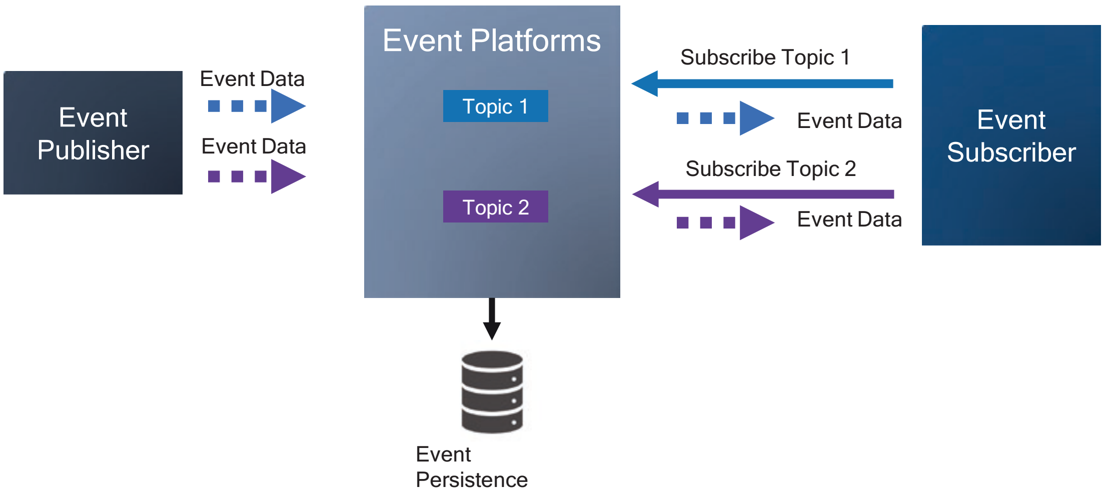


***Figure 6-3.** Event-driven architecture components*

An event producer publishes an event to the router, which filters and pushes the events to the consumers.

Let's consider the example shown in Figure 6-4, showing payment processing in banks or a centralized payment platform in a country. You will receive a lot of credit and debit transactions, and you need to process millions of transactions. In these transactions, there might be a few transactions that are related to terror funding or anti-money laundering. How will you find these dubious transactions? If you scan these transactions offline, it leads to a delay in identifying transactions. The only option is to identify in real time just before completing the transaction. In this case, the event-driven architecture helps you to identify the dubious transactions in real time.

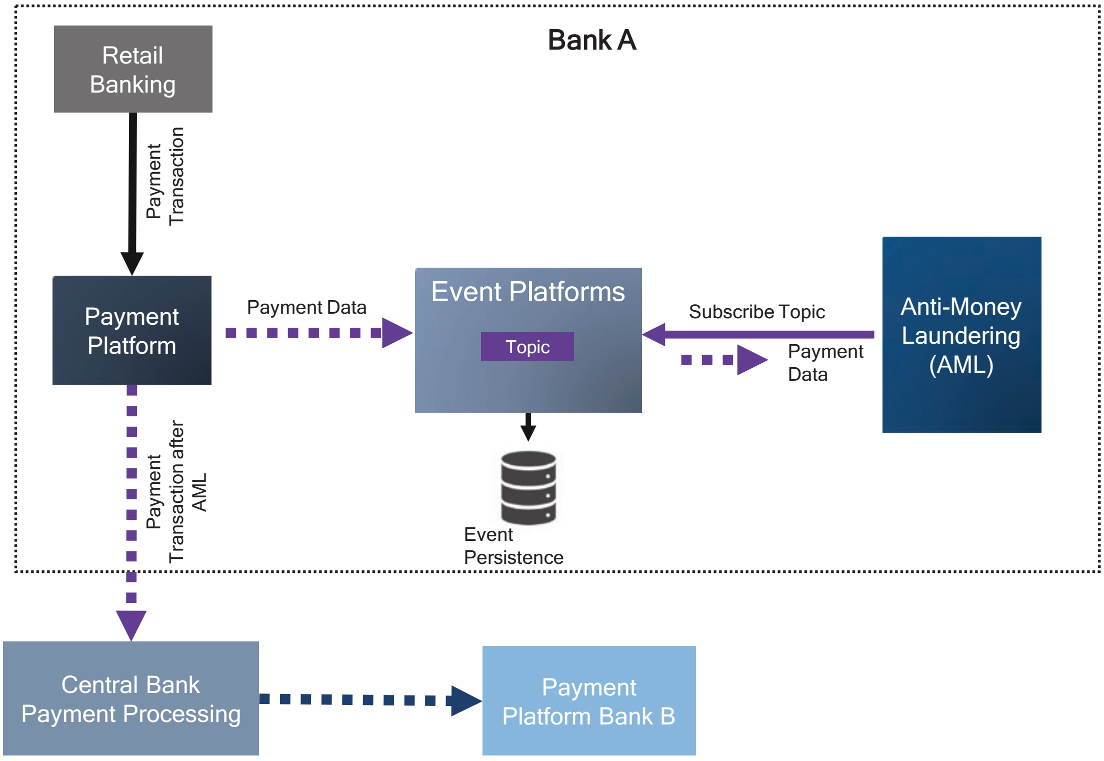


Figure 6-4. *Event-driven architecture example, payment platform*

Event-Driven Topologies

When you design an event-driven architecture, you may confuse which topology needs to be considered for your architecture and why.

In an event-driven architecture, there are two topologies. You need to choose the right topology for your use case.

- Mediator topology
- Broker topology

Mediator Topology

The mediator topology is like orchestration in an SOA enterprise service bus (ESB) or orchestrator components like Netflix Conductor or Uber Cadence. You use the mediator topology when you need to orchestrate multiple steps within an event through a central mediator. This topology is better suited for more complex situations where multiple steps are required to complete the process, thus requiring event processing coordination or orchestration.

The mediator topology consists of four components.

- Event queues
- Event mediator
- Channels
- Event processors

The event flow starts from the event originator by sending an event to event queues; these queues send events to the event mediator. The event mediator is the central component that controls the orchestration of services and is leveraged in a situation where a particular service needs to perform multiple steps sequentially to execute a certain business process. The event mediator sends asynchronous events to event channels to execute each step of the process. The event processor listens to each channel, receives an event, and executes the required business logic. The event mediator is not a business logic executer but is configured with orchestration to process an event.

As shown in the payment use case in Figure 6-5, the consumer makes a payment, and the "make payment" use case requires multiple steps to complete the payment process. The payment request is sent to the event mediator by the retail banking app or web application. The mediator orchestrates multiple steps like conducting AML, checking the payment, sending payment instructions to the central bank, etc. These steps are event processors to process the business logic.

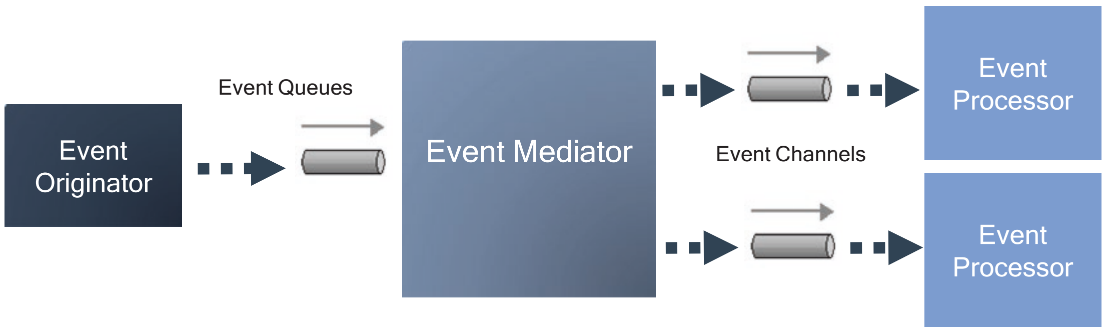


Figure 6-5. *Mediator topology architecture*

The software components are Camel, Fuse, etc., for the mediator topology along with Rabbit MQ, Active MQ, IBM MQ, or Kafka.

Broker Topology

In the broker topology, the message flow is distributed across the event processor components in a rope fashion through lightweight message brokers. It does not have a central component that controls the orchestration across processes as provided by the mediator topology. The broker topology mainly consists of a dumb broker and intelligent processor with dumb and pipe patterns.

There are two main components in the broker topology.

- Broker component
- Event processor component

The broker component can be centralized or federated and collaborates with all the events that are used within an event flow. The events contained within the broker can be message queues, topics, or a combination of both.

As you saw in Figure 6-6, there is no central mediator component controlling orchestration. In this topology, each event processor component is responsible for processing an event and publishing a new event indicating the action it just performed. The event processor acts as a broker for the rope of events. Once the event is processed by the processor, the other event is published so that another processor can proceed.

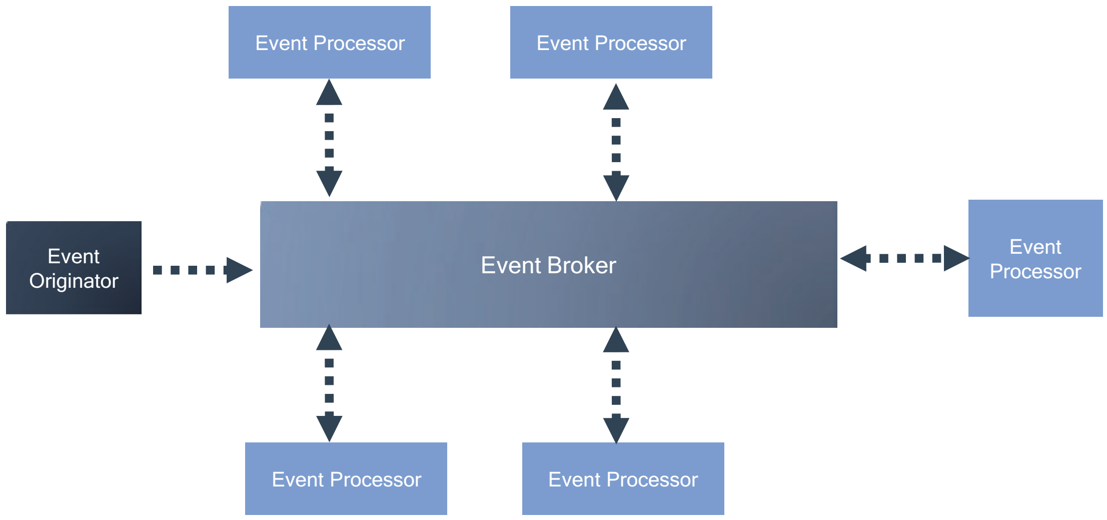


Figure 6-6. *Broker topology architecture*

In the same example of the payment processor that I mentioned, I have used a broker topology to integrate the payment process and to update the details in a banking application. Once the payment is processed, then we need to update the books to complete the transaction, and the payment services write the transaction to the Kafka broker. An event processor picks it up and inserts the record in MongoDB. The view transaction will retrieve the records from MongoDB and expose the API to users to view the transaction.

Choice of Topology

The rule of thumb is to choose the best topology for your use cases. The broker topology can be considered for a single event or chain of events requiring one task as their result. The mediator topology can be considered when using multiple tasks in response and thus requiring orchestration of each task.

Characteristics of Event-Driven Architecture

Almost all industry domains use event-driven architecture such as social media, financial markets, hospitality, Internet of Things (IoT), etc. Let's consider an example of IoT. Say your apartment building has installed sensors in each apartment to identify fire or smoke in the apartment building. The sensors send the details of events with a measurement of the average temperature of a room with a timestamp. The event-driven system will send events along with room temperature data to identify processes, and storing these events requires various EDA characteristics to complete the process. The following are the main characteristics of any event-driven architecture that you must follow:

- *Multicast communication*: The events are generated from the publishing systems, and event-driven systems can send these events to multiple event processors.
- *Real-time transmission*: The events are processed in real time to the event processors. The mode of processing or transmission is real time rather than batch processing.
- *Asynchronous communication*: The event does not need to wait for the event processor to be available before publishing an event.

- *Fine-grained communication*: Events are small units and are published as and when they occur.
- *Event ontology*: EDA systems always classify events in terms of some form of a group/hierarchy. This allows event processors to subscribe to a specific event or specific category of events.

Event-Driven Messaging Models

There are two basic models for transmitting the events in an event-driven architecture; you can use the right models for your use cases.

Event Messaging

In event messaging, the event consumers subscribe to the messaging published by the event originators. When an event originator publishes an event, the message is sent directly to all subscribers who want to consume it. The event broker handles the transmission of event messages between the originators and subscribers. The events will be deleted after all the consumers subscribe to them. An example of event messaging is the published/subscribe model. The event broker translates and routes messages to the subscriber.

Event Streaming

In event streaming, event originators publish streams of events to a broker. Event processors subscribe to the streams, but instead of receiving and consuming every event as it is published, event processors can consume events at any point and consume only the required events. The events are persisted and never deleted after the event processors consume them. The event streaming platforms are configured to persist events from a second to infinite time. This enables event streams to process real-time and historical data. The event streams can be used for both simple and complex event processing styles.

Event Processing Styles

Event processing is the process that takes events or streams of events, analyzes them, and takes automatic action. Each event processor must be independent and loosely coupled with other event processors. It tracks and processes streams of events so that opportunities and risks are proactively identified and optimized. There are three types of styles for event processing.

- Simple event processing (SEP)
- Complex event processing (CEP)
- Event stream processing (ESP)

Simple Event Processing

This event processing occurs when an event immediately triggers an action in the event processor. It is used to measure events that are related to specific measurable changes in conditions. SEP is used for real-time flow without any other constraint or consideration. Many events in architecture are simple, such as IoT sensors in your house that trigger when something happens in a house like a temperature change or smoke, etc. This type of event occurs when some notable, significant, and meaningful change of state or condition occurs. Typically this is used to take latency and cost out of the business process; simple event processing initiates action further down the application stream whenever a significant and meaningful change of state occurs in any hardware or software component of the system.

Event Stream Processing

In event stream processing, ordinary events that occur are filtered for notability and sent to event processors. This ensures that real-time information flows in and around the enterprise. This helps in real-time decision-making. In event stream processing, all the events are written to a log. Event processors don’t subscribe to anything; they simply read from any part of the stream at any time. The following are the components of event stream processing:

- Event collect
- Event enhance

- Event analyze
- Event dispatch

This flow creates a process in which events are detected using components. These components detect relationships between multiple events, perform event correlation, and establish event hierarchies.

The event stream processing uses a data streaming platform like Kafka to ingest events and processes or transform the event stream. This can be used to detect a pattern in event streams. The event stream processing can be used in various use cases, for example, in order processing. If we consider the sequence of events in a jewelry shop, the RFID sensor generates an event for each item that moves out of the display. In this scenario, the retailer is to be informed when the item is sold and moved out of a store.

Complex Event Processing

This is a set of processes for capturing and analyzing streams of data as they arrive in real time. The objective of this processing is to identify meaningful events in real-time situations and respond to them as quickly as possible. It is used when multiple events must take place before final events are generated. Each event need not be like the others, nor do events occur at the same time. CEP waits until all criteria are fulfilled before generating an event message to communicate action instructions. To generate a finale event, the CEP requires the following components:

- Event interpreters
- Event pattern definition
- Event pattern matching
- Event correlation techniques

The CEP has a strong impact on future information systems and the way we subscribe to and consume information. It plays an important role in many domains like logistics, energy management, finance, etc. The usage of this style is expected to grow further with the increasing number of decentralized microservices, digital twins, etc.

CEP does not only mediate information in the form of events between providers and consumers but supports the detection of dependencies among events by using event patterns. The events are generated by the composition and aggregation of multiple events and can generate a final event.

CEP is used for a scenario in which there is a large volume of events occurring and latency requirements are very low, in milliseconds. Some of the use cases are stock trading, predictive maintenance (digital twin), real-time marketing, etc.

Event-Driven Architecture Maturity Model

IT in enterprise organizations needs to support business disruption by improving the speed and responsiveness of their internal and customer-facing processes and systems. Irrespective of what industries you are in, there is an increase in eventing capability across enterprise ecosystems. An EDA not only publishes and subscribes to an event but involves planning and maturity of EDA across portfolios in an enterprise.

The EDA concept becomes more broadly adopted, and enterprises progress through increasing levels of maturity. As shown in Figure 6-7, every organization has to undergo five levels of steps to reach maturity because the eventing is complex, requires special skills, and most important is part of the organizational culture. You can use assessment techniques to assess an enterprise's maturity level.

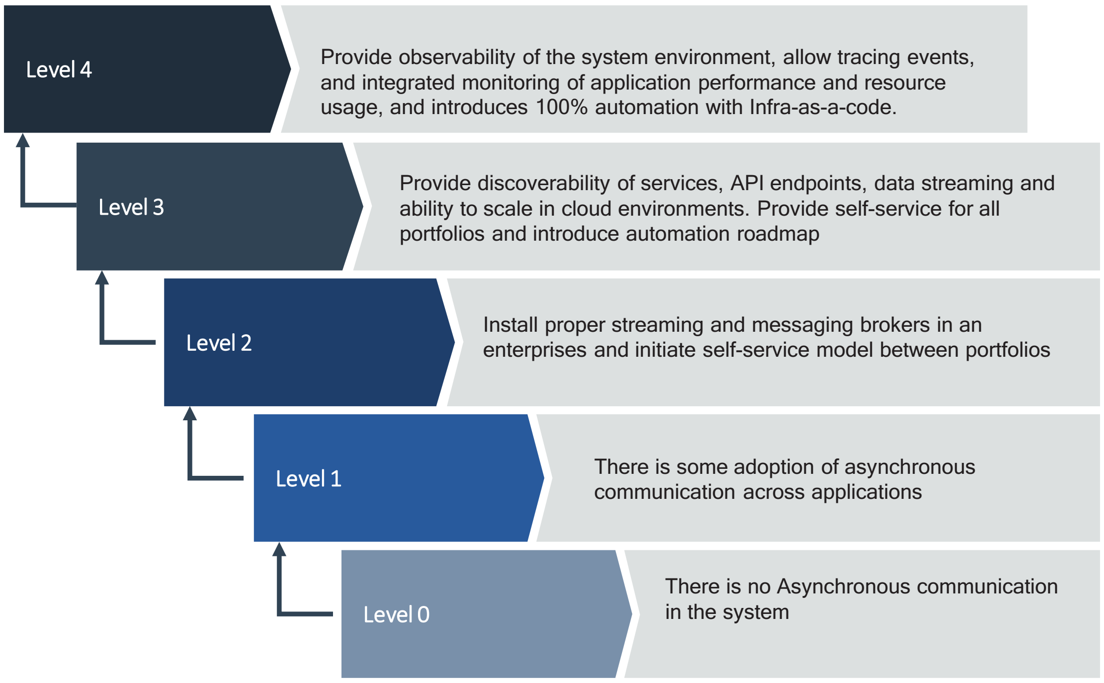

Level 4
Provide observability of the system environment, allow tracing events, and integrated monitoring of application performance and resource usage, and introduces 100% automation with Infra-as-a-code.
Level 3
Provide discoverability of services, API endpoints, data streaming and ability to scale in cloud environments. Provide self-service for all portfolios and introduce automation roadmap
Level 2
Install proper streaming and messaging brokers in an enterprises and initiate self-service model between portfolios
Level 1
There is some adoption of asynchronous communication across applications
Level 0
There is no Asynchronous communication in the system

Figure 6-7. *Maturity model*

Level 0: There is no asynchronous communication in the enterprise. All integration is synchronous through APIs or TCP/IP or FTPs. Few applications in an enterprise are using some kind of ESB integration mechanism across the heterogeneous system.

Level 1: There is some adoption of an asynchronous exchange of information across applications. An example is a point-to-point interaction between related systems using messaging platforms like MQs, ESB, etc.

Level 2: Business and data event streaming and messaging are used with some level of high availability and initiating of self-services. Multiple application interactions are carried out by using messaging infrastructure with messaging characteristics.

Level 3: Discoverability, scalability, and failover are managed. The application-producing events are less aware that all the clients can subscribe to the messages and the same events can be used for various other observability. The message network can handle the variable load by using cloud native architecture, and the event publisher and subscriber are unaware of the physical network topology. Some automation is introduced to handle the software engineering lifecycle.

Level 4: The observability principle is enabled across applications and portfolios and the software engineering lifecycle, and the infrastructure is fully automated. The events are pervasive in enterprises with multiple publishers and subscribers. The focus is more on scale and robust messaging infrastructure. Enterprise-wide observability is configured so that administrators can use integrated and intelligent monitoring in real time and trace messages across multiple nodes in an event mesh. Full automation is enabled with a single click of deployment. With this eventing maturity, the organization embraces cloud native tech stacks to support a variety of business disruptions, and that leads to change in the organization culture to embrace a new set of technologies.

Decoupling Use Case by Using Event-Driven Architecture

The decoupling helps enterprises with legacy systems to engage customers in the following ways:

- Keeping legacy systems
- Making these systems accessible from the cloud native systems
- Shipping data to modern technology
- Enabling enterprises to access cloud native technologies

The era of the big transformation project is over; enterprises are not willing to invest in multimillion, multiyear efforts on transformation; they need to realize business value quickly. Instead of big fat projects, you need to imagine a world in which value is delivered quickly and accessible to customers after a short duration minimum viable product (MVP) and then continuously thereafter, with the freedom to pivot. On the journey to cloud native, you can't ignore legacy systems. There are tons of business transactions occurring in those systems; therefore, you need to keep evolving your architecture by using decoupling principles.

Decoupling is the process of using cloud native technologies, development methodologies, and migration methods to build systems that execute strategy on top of legacy systems. When you apply a decoupling strategy to the entire enterprise, it leads to exponential changes in IT and a scalable, flexible, and resilient architecture that gives companies the agility to continuously innovate.

Organizations are under constant pressure to deliver customer expectations. The following are the key drivers for organizations to embrace cloud native architecture:

- Changing customer expectations
- Technology innovation
- Cost pressure
- Extended enterprises
- New unicorn entrants

Figure 6-8 illustrates a step-by-step approach of how to conduct a decoupling of an existing system into the cloud native technologies. When your systems need to undergo decoupling transformation, you must adhere to the following principles:

- *Layering*: Apply layering to isolate from the old system, and layer within the new system.
- *Suitable fragmentation*: Fragment capabilities remove conflicts of interest and increase agility, enabling cloud native replacement.
- *4Events*: Make sure all data is accessible in real time.
- *Available, Real-time data*: Build out data meshes and data lakes with real-time eventing capabilities to support the objectives.
- *Automation*: Implement single-click automation from developer box to the production box with the use of DevSecOps and infrastructure as code.
- *Cloud*: Leverage cloud capabilities to isolate infrastructure and platform.
- *Intelligence built-in*: Add artificial intelligence and machine learning into your services and operations.
- *No SPOF*: Avoid a single point of failure (SPOF).

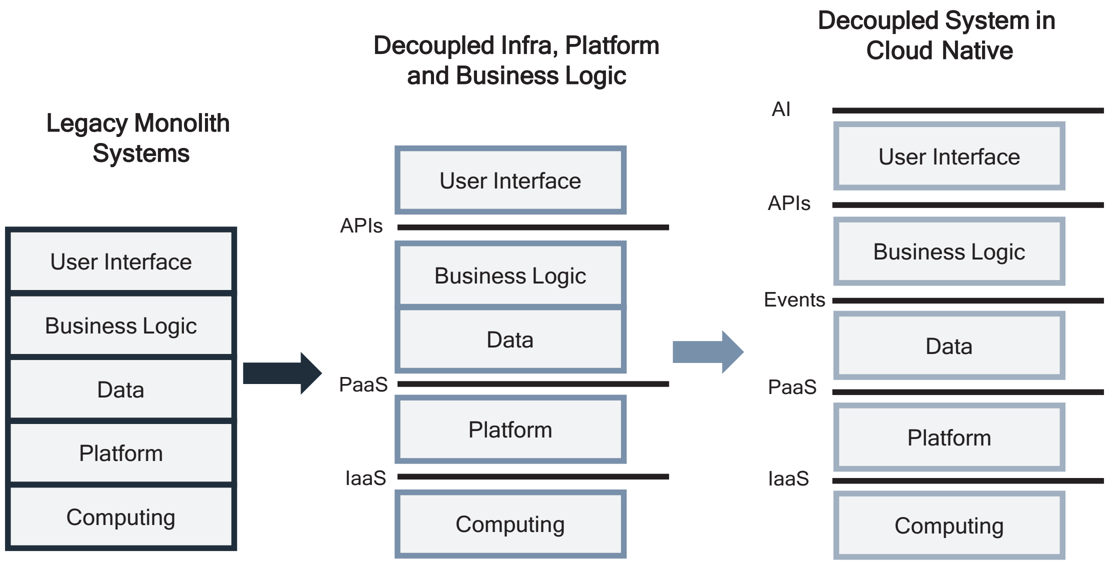


Figure 6-8. *Decoupling architecture*

Some of the myths of decoupling that you need to dispel for stakeholders are as follows:

- *A product alone solves a business problem*: Don't rely on someone else to solve our problems.
- *Perfect architecture and governance*: Avoid ivory tower thinking and focusing on overdesigned standards that don't drive business value.
- *End state is reachable*: Businesses don't stand still; what is valid today may not be tomorrow.
- *Old = bad*: Oftentimes, new technology is seen as the only way to solve problems.
- *All in one jar*: Oftentimes, a business uses one technology to solve all the problems.

When you want your application to be cloud native, then you need to apply the following modern-day approaches for decoupling your systems:

- Make data accessible (change data capture [CDC])
- Microservices
- Event-driven architecture
- Serverless
- Cloud
- Reactive interaction gateway

Make Data Accessible

If you look back at how application databases and data movement are designed traditionally, all are based on the pull-based model, with no information about changes. The data in the databases is static and reacts only when there is a request to modify the data by using SQL queries; it never reacts on its own. In the case of data replication, you should apply batch jobs to trigger a delta change in the source database and use extract, transform, and load (ETL) tools to load data in an (ODS) operational data store or data warehouse. Or use files to upload data to the ODS or data warehouse.

In event-driven architecture, events enable new real-time functionality to move data from the application and data store. You can apply replication either from business change events in an application or from technical change events in a database, as shown in Figure 6-9.

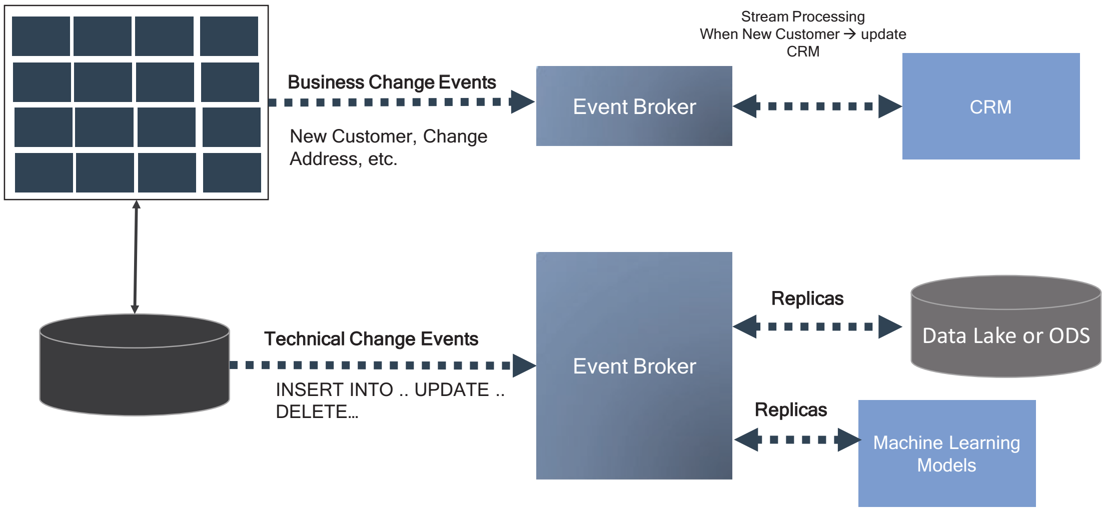


Figure 6-9. *Data accessible architecture*

How to Get Events and Make Data Accessible?

Many systems have native support for events. For example, databases like MongoDB, Couchbase, Cockroach DB, etc., and cloud services like AWS S3, Google Storage, and Blob storage in Azure, all these services provide an event when a file is uploaded. GitHub provides a webhook on all kinds of operations, and Salesforce provides change events.

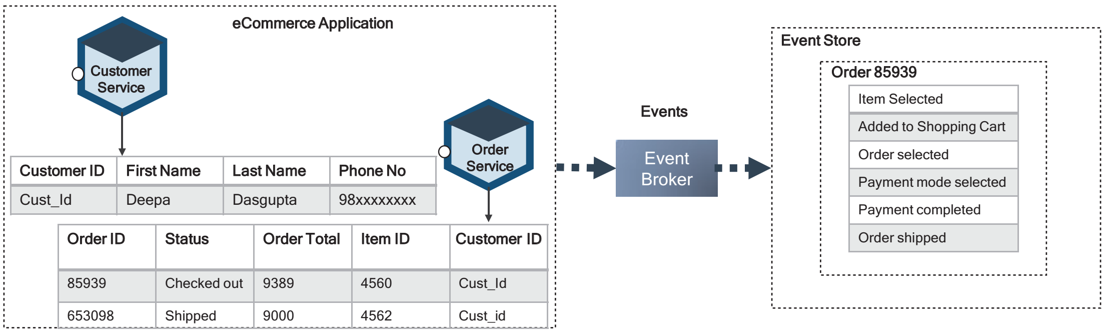
If services do not support a native eventing capability, then you need to build that functionality yourself by using the following methods:

- CDC tools such as Equalum, Hevo, Infosphere, Qlik, Oracle, etc., allow you to integrate various legacy systems, such as Oracle and DB2.
- You can build your custom component that has central transaction logic points, allowing you to add code to publish events.
- If you are developing a new cloud native application, then you need to consider building an eventing capability right from the start.

Where to Store Events?

An event store is a database designed for tracking events as they occur and maintaining a record of those events. Relational databases such as MySQL, Postgres, SQL Server, and Oracle can be used to track events but have certain limitations for this use. A relational database stores data in a tabular structure and isn't a good match for event data; in addition, facts stored in a relational database can be changed. You can present up-to-date data from a relational database, but the limitation is to track every action.

Event stores record each event as it occurs, and no event may be overwritten or deleted. For example, as shown in the Figure 6-10, ecommerce may allow each customer to browse catalogs and items as a set of events. Adding each item to a wish list or shopping cart, adding payment details, entering a shipping address, and checking out are all events that should be recorded as they happen, and records of these events never change. Events recorded in an event store are immutable. Facts derived from those events change over time; in the same example, the customer enters different payment methods for each purchase.



Customer ID	First Name	Last Name	Phone No
Cust_Id	Deepa	Dasgupta	98xxxxxxxx

Order ID	Status	Order Total	Item ID	Customer ID
85939	Checked out	9389	4560	Cust_Id
653098	Shipped	9000	4562	Cust_id

Order 85939
Item Selected
Added to Shopping Cart
Order selected
Payment mode selected
Payment completed
Order shipped

Figure 6-10. *Event store*

Event stores are ideal for applications where an audit trail, a machine learning model, a record of actions, etc., is desired. This is common for all event-driven transactional applications.

How to Get Data?

Getting data is often the hardest part; there are several ways to replicate data such as the following:

- Using CDC
- Replicating data to a log and using the log as a source of truth
- Using connectors, either industry tools or custom made

CDC

CDC is a process for identifying and capturing changes made to a data store; those changes can then applied to another data repository such as a data mesh or data lake or a data warehouse or event log by using event-driven architecture or other types of integration tools like ETL. CDC is the basis for another system with the same incremental changes or to store an audit trail of changes. The audit trail may subsequently be used for other uses such as updating to a data lake or data warehouse or running machine learning models across the changes.

CDC replicates data that has changed with database functions such as INSERT, UPDATE, and DELETE and makes a record of the change available to the CDC tool and event hub so it's available for other sources. CDC tools rely on database logs, which keep track of record changes internally for system recovery.

There are different approaches that a system can use to capture changes in the transaction databases, such as the following:

- Database transaction logs
- Use of timestamp column in a table
- Event streaming

The CDC tools scan databases for timestamp updates; if there are any updates, the transaction implements database triggers, and CDC tools capture the changes. This method degrades the performance of a transaction database.

Every database logs its transaction. The log scanners can identify any changes in these transactional logs, and the log scanner interprets and captures the changes in these transaction logs.

Event streaming is commonly used and relevant in cloud native architecture. It uses the publish/subscribe model of CDC, where a database triggers a log or publishes change events to a table and shares those changes with the CDC tool. The series of updates is sent to CDC in streams to be used to capture the changes in the CDC.

The event streams start the process of taking action on a series of data that originates from a data-driven application in an enterprise that continuously creates data. The term *event* refers to each data point in an application, and *stream* refers to the delivery of those events. During the streaming, there are many actions or logic that can be applied such as aggregation, analytics, enrichment, transformation, and ingestion. Event streaming is the real-time processing of data as soon as changes occur.

As shown in Figure 6-11, event stream processing works by handling a data set as one data point at a time rather than as a whole data set. Event streams are about continuously created data. In an event stream processing setup, there are two parts.

- The event storage
- Technology to take actions on changes in the database

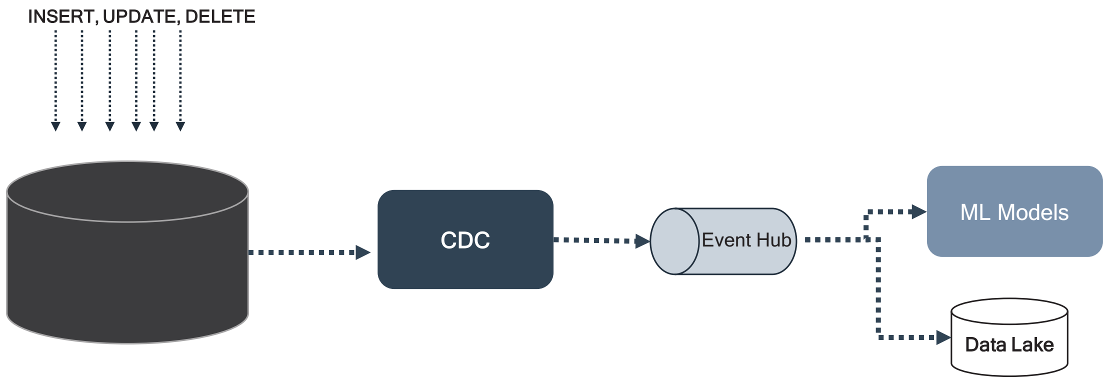


***Figure 6-11.** Data streaming by using CDC and events*

The event storage stores data based on the timestamp. You might capture every action of users in an ecommerce application, and each action of the user is an event. This is handled by streaming technologies like Kafka, Kinesis, etc. The stream processors act on the incoming data. The enriched stream events are published to the steaming technologies for stream persistence.

Real-Time Interactivity

Real-time interactivity is the backbone of the modern-day customer experience. It establishes a scalable and agile event processing capability and generates new representation.

To provide real-time connectivity, as shown in Figure 6-12, you need to use event processing and data streaming to integrate services and systems in your enterprises and merge, transform, and enrich relevant data across an organization.

A batch process is always too late to respond to customers and introduces a bumpy load pattern. Distributed log systems offer very high throughput, strict ordering per log file, and independent reads from multiple systems, but they do not support the event streaming.

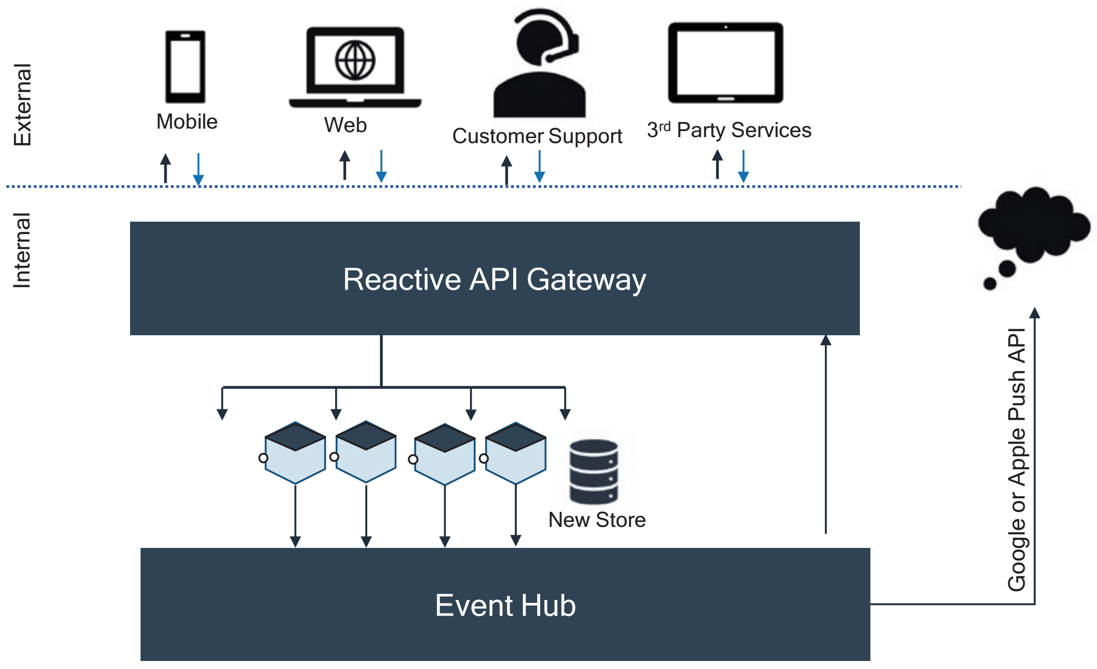


Figure 6-12. *Reactive architecture*

In an event hub or event streaming, a log is used instead of a service bus, and the service listens to the log and publishes messages to the log, typically on topics.

How to Use Existing Message Queues with Event Streams?

You can leverage your existing architecture, skills, and investments, and you can use event-driven techniques to offer more responsive and seamless integration with existing and new event streams. Event streams like Kafka, IBM Event Streams, etc., support connectivity to the existing MQs like Rabbit MQ, Active MQ, or IBM MQ. By combining the capabilities of event streams and message queues, you can combine your transactions in a combined application.

Let's consider a use case, as shown in Figure 6-13, in the travel and hospitality industry. The use cases are airline booking, car rental, hotel booking, flight status, and a weather report to provide a more personalized experience to customers. In these use cases, your client already developed part of the airline booking by using MQ, but you need to provide seamless and more personalized information to your customer.

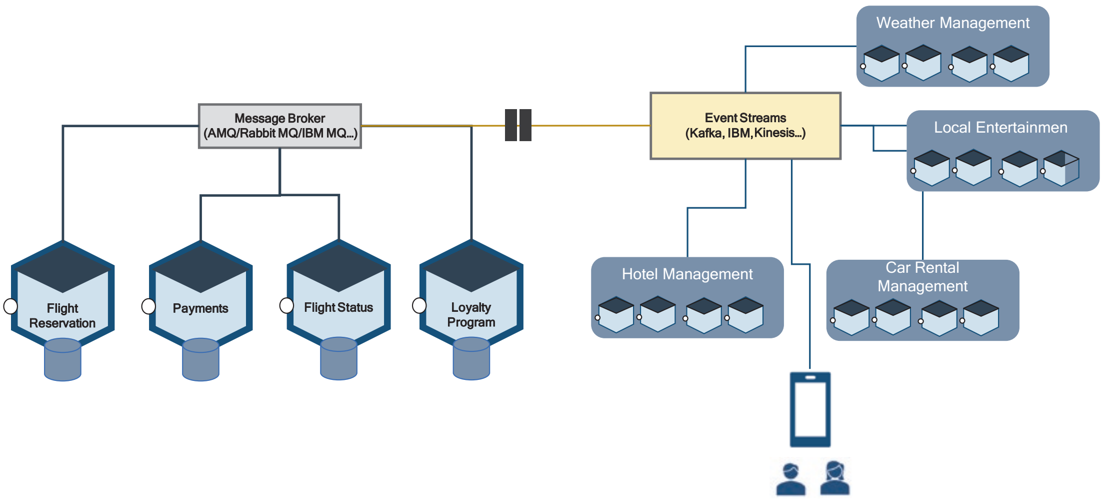
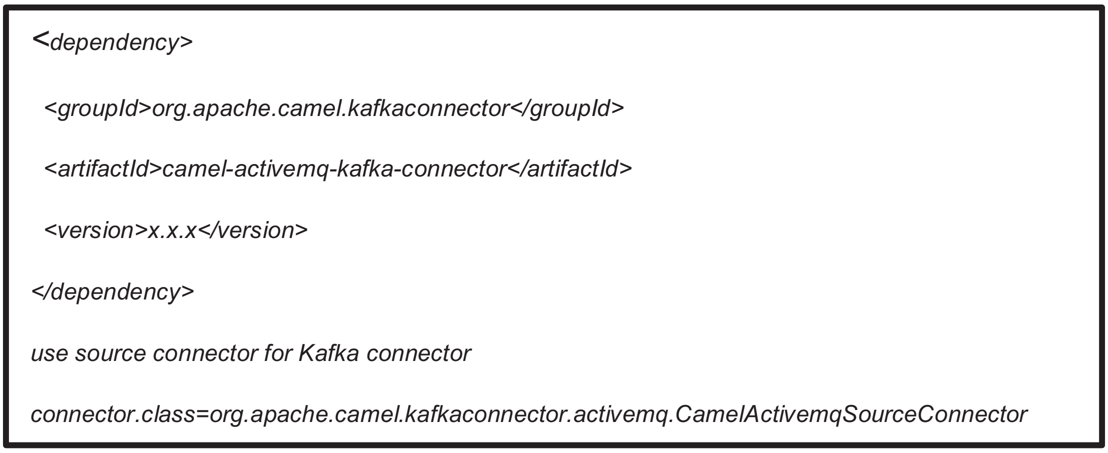


Figure 6-13. *Collaboration of eventing system with message queue systems*

Let's consider an example of a flight reservation; the management of flight reservation applications is already available in an enterprise with the decoupled architecture principles via MQ technologies. The airline management wants to enhance its business by providing a personalized experience for its customers. As mentioned earlier, the MQ is not meant for eventing and streaming. Therefore, I used event streaming technologies such as Kafka, IBM, or Kinesis to stream across various systems to provide seamless information to the customers.

To achieve interaction between MQs and event streams, you need to configure MQ to send and receive messages and events by using connectors. Event streams connect with various applications to manage a hotel reservation and location map, local entertainment details, and map and car rentals from the car booking management application.

For example, as shown in Figure 6-14, for the AMQ connection to Kafka, you need to configure a connector in the dependency file.



```
<dependency>
  <groupId>org.apache.camel.kafkaconnector</groupId>
  <artifactId>camel-activemq-kafka-connector</artifactId>
  <version>x.x.x</version>
</dependency>
use source connector for Kafka connector
connector.class=org.apache.camel.kafkaconnector.activemq.CamelActivemqSourceConnector
```

Figure 6-14. *AMQ and Kafka configuration file*

Transaction Management in Event-Driven Microservices

A legacy application usually has a single monolithic database. The ACID transactions can be easily maintained in a single monolithic database. ACID means the following:

- *A – Atomicity*: A transaction is an atomic unit. All the instructions within a transaction will successfully execute, or none of them will execute.
- *C – Consistency*: A transaction can bring the database from only one valid state to another, and data is in a consistent state when a transaction starts and when it ends.
- *I- Isolation*: One state of a transaction is invisible to another transaction. This ensures that concurrency is maintained across transaction and leaves the database in the same state.
- *D – Durability*: Changes that have been committed to the database should remain even in the case of failures.

As a result of ACID, your monolithic application and database can easily manage the database transactions.

When you decouple an application to a cloud native service or develop a new cloud native service, data access management becomes complex because of polyglot principles. Adopting a polyglot principle ensures that the microservices are loosely coupled and deploy and are managed independently of one another. If multiple services access the same data, then you need to handle coordination across cloud native services. One more obstacle is transaction management in polyglot microservices. The polyglot principle illustrates that each microservices can use different databases because a modern application stores diverse kinds of data, and one type of database is not always beneficial.

For some cloud native service, a NoSQL database might have a more convenient data model and offer much performance and scalability. It's similar for search microservices; you may be considering Elasticsearch for the graph-related store, and you might use graph databases like Neo4J, etc. In a nutshell, in one system, you might use multiple types of databases. Using polyglot persistence provides many benefits such as scalability, manageability, and high availability but introduces distributed data management challenges.

The following are the real challenges of using polyglot persistence in a cloud native service:

- Implementing a business transaction across services
- Retrieving data from multiple services

Let's analyze how these challenges impact your cloud native services.

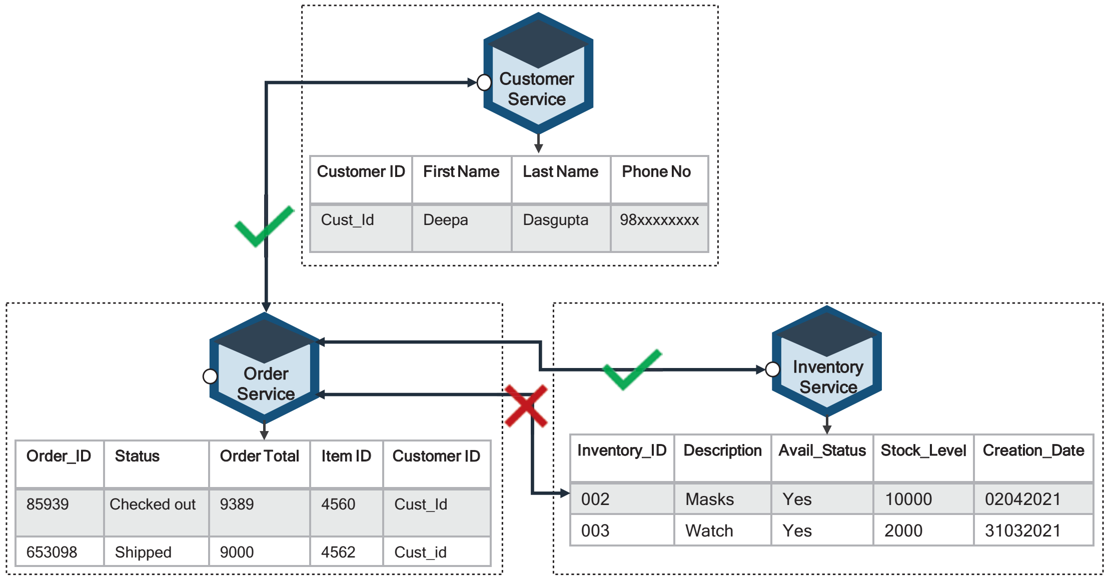
The first challenge is implementing a business transaction that maintains consistency across services. Let's consider the example of an ecommerce application. The ecommerce application consists of hundreds of cloud native services to manage various business cases such as Order, Customer, Inventory, Catalog, etc.

In Figure 6-15, I am considering three cloud native services (Customer Service, Order Service, and Inventory Service) to illustrate transaction management.

- *Customer Service*: The responsibility of this microservice is to maintain customer information.
- *Order Service*: The responsibility of this microservice is the management of orders.
- *Inventory Service*: The responsibility of this microservice is to manage the inventory, and a new order doesn't give confirmation if the inventory is less than the number of product requested.

In the traditional monolithic application of ecommerce, the Order service can simply use an ACID transaction to check the availability in the inventory and confirm the order.

In the cloud native service architecture, the Customer, Order, and Inventory tables are aligned to their services, as shown in Figure 6-15.



Customer ID	First Name	Last Name	Phone No
Cust_Id	Deepa	Dasgupta	98xxxxxxxx

Order_ID	Status	Order Total	Item ID	Customer ID
85939	Checked out	9389	4560	Cust_Id
653098	Shipped	9000	4562	Cust_id

Inventory_ID	Description	Avail_Status	Stock_Level	Creation_Date
002	Masks	Yes	10000	02042021
003	Watch	Yes	2000	31032021

Figure 6-15. *Cloud native service polyglot persistence*

The Order service cannot access the Inventory service table directly and can be used only through the Inventory service's APIs or channels. When cloud native services such as Customer, Order, and Inventory services decomposes a monolithic system into self-encapsulated services, it can break transactions.

This means a local transaction of a monolithic application becomes distributed into multiple services. Figure 6-16 shows how the transaction could be handled in a monolithic ecommerce application; it shows a customer order example with a monolithic ecommerce system using a local transaction.

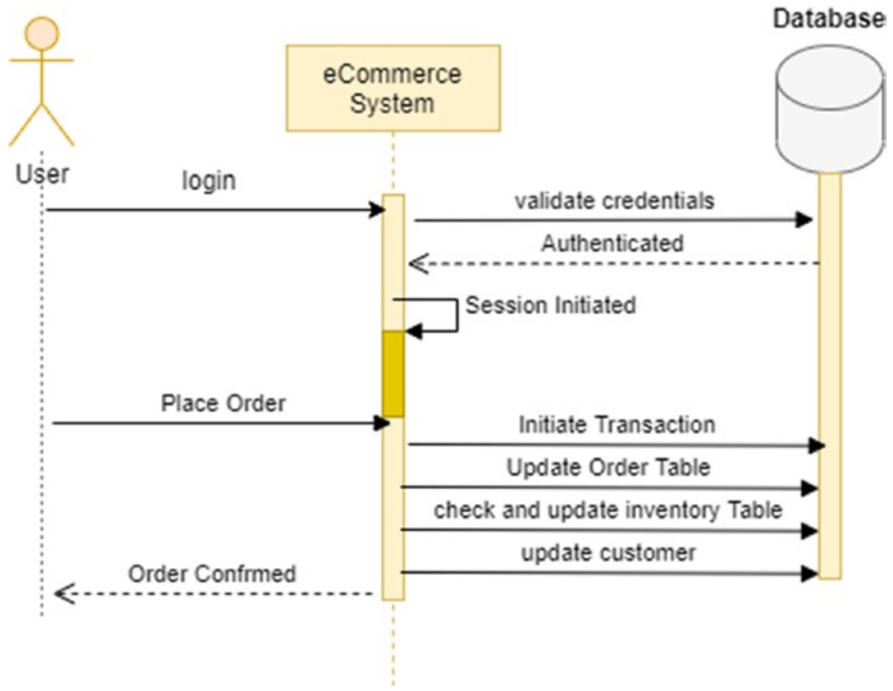


Figure 6-16. *Monolithic ecommerce system using local transaction*

As shown in Figure 6-16, the user logs in to the ecommerce system after authentication, and the system creates a session. The user places an order in the system, and the system creates a local transaction that manages multiple database tables by using an ACID transaction. If one step fails, the transaction can roll back.

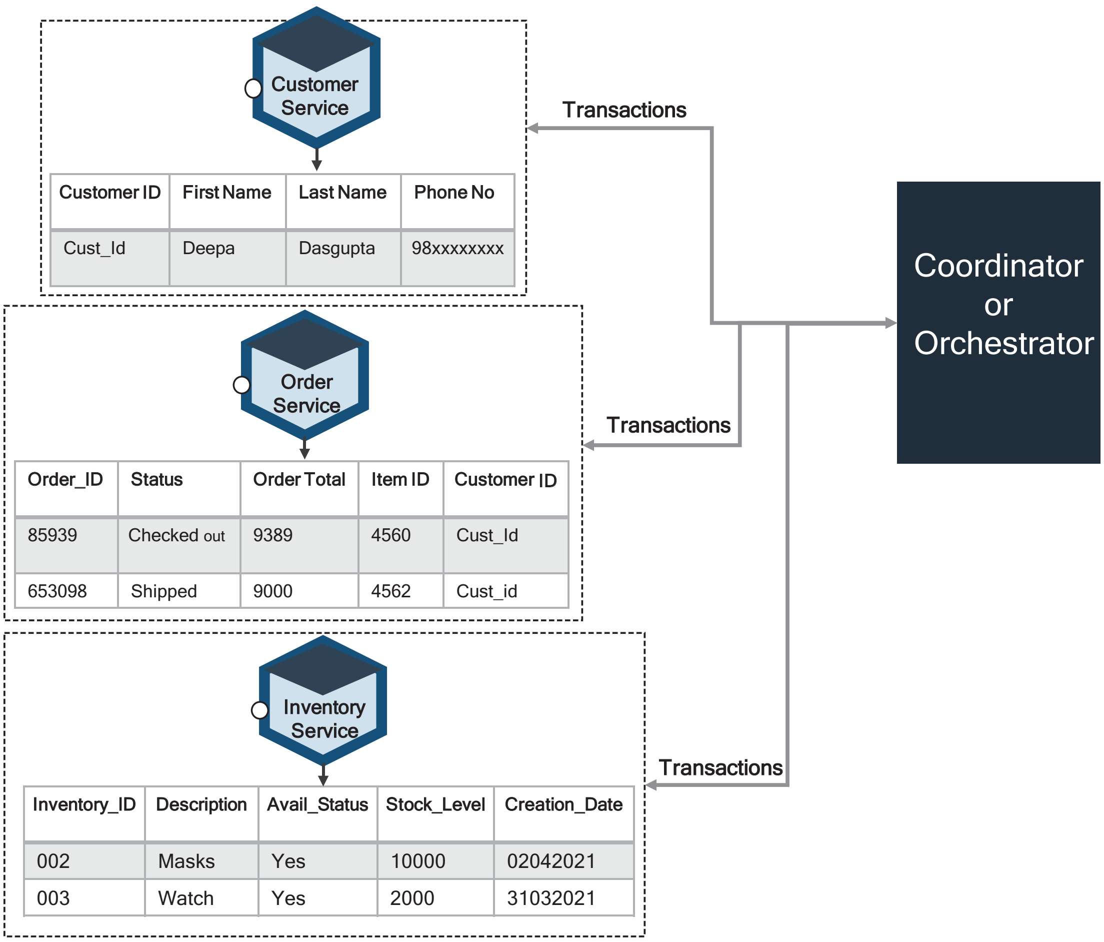
Two-Phase Commit in Cloud Native Services

In the cloud native services architecture, the Order service could potentially use the Inventory service through a distributed transaction's two-phase commit (2PC). The 2PC protocol ensures a database commit is implemented in the places where a commit operation is divided into two separate phases.

- Prepare phase
- Commit phase

Let me explain how you can use 2PC for a cloud native services architecture for the Customer, Order, and Inventory services.

In the preparation phase, the Customer, Order, and Inventory services of the transaction prepare to commit and notify the coordinator that they ready to complete the transaction. In the commit phase, the transaction is either a commit or rollback command issued by the transaction coordinator to all the services. Figure 6-17 shows the 2PC implementation for customer orders.



Customer ID	First Name	Last Name	Phone No
Cust_Id	Deepa	Dasgupta	98xxxxxxxx

Order_ID	Status	Order Total	Item ID	Customer ID
85939	Checked out	9389	4560	Cust_Id
653098	Shipped	9000	4562	Cust_id

Inventory_ID	Description	Avail_Status	Stock_Level	Creation_Date
002	Masks	Yes	10000	02042021
003	Watch	Yes	2000	31032021

Figure 6-17. *2PC commit*

In Figure 6-18, when a customer creates an order, the coordinator or orchestrator creates a global transaction with all the context information. It will interact with the Order service to create an order, and the order replies to the coordinator after completion of order creation. Then the coordinator sends a request to the Inventory service the check the inventory availability by product ID. The Inventory service sends

OK, and the stock is available. The coordinator sends a message to the Order service to confirm the order, and at the same time the coordinator sends a message to the Inventory service to update it. At any point in time, if the service fails to process, then the coordinator will abort the transaction and begin the rollback process.

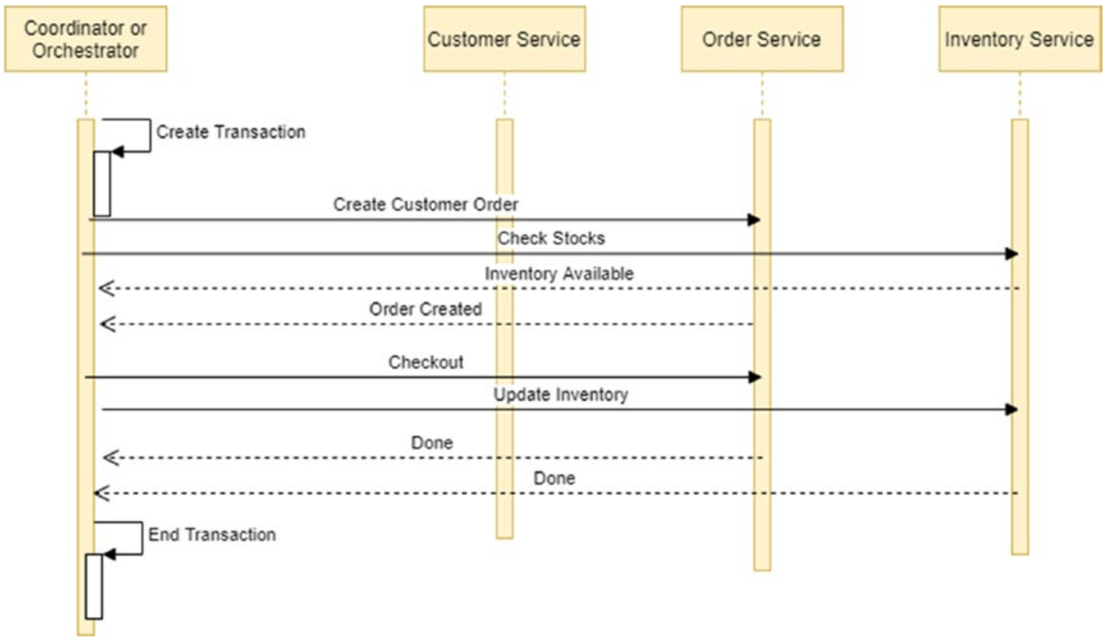


Figure 6-18. *Sequence steps of 2PC*

The benefit of 2PC is strong consistency.

- Prepare and commit 2PC phases to guarantee that transactions are atomic, either complete or none.
- 2PC allows read-write isolation; the changes are not reflected until the coordinator is not performing the commit.

The disadvantage of 2PC is that you can solve the transaction by using 2PC, but it is not at all a recommended approach for any cloud native architecture systems because of the following reasons:

- 2PC is synchronous (blocking); it will lock all the cloud native services until it completes the entire transaction. This could end up as a bottleneck in the whole system.

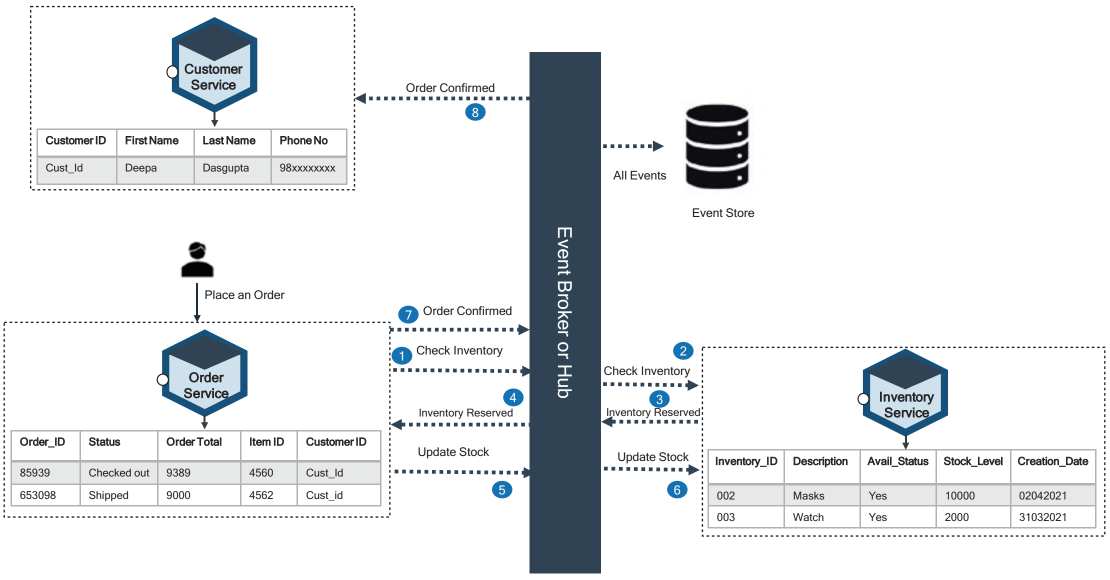
- This approach is very slow, due to the blocking of threads of all the participants' microservices.
- A coordinator or orchestrator is a single point of failure, and the whole system's transactions are based on the availability of a coordinator.
- The consistency, availability, and partition (CAP) theorem requires you to choose between availability and ACID properties. Based on my experience, the availability is better for cloud native.
- Modern databases such as NoSQL do not support 2PC.

Transactions with Events

In an event-driven architecture, a microservice publishes an event based on when a command is issued, and related cloud native services subscribe to events.

You can use events to implement transactions that span multiple participating services. You need to implement multiple steps to complete one business transaction, and each step in business transaction is processed with event publishes from previous step and triggers transaction to the next steps.

Figure 6-19 shows how to implement transactions by using event-driven architecture and event sourcing with the same use case as mentioned for 2PC.



Customer ID	First Name	Last Name	Phone No
Cust_Id	Deepa	Dasgupta	98xxxxxxxx

Order_ID	Status	Order Total	Item ID	Customer ID
85939	Checked out	9389	4560	Cust_Id
653098	Shipped	9000	4562	Cust_id

Inventory_ID	Description	Avail_Status	Stock_Level	Creation_Date
002	Masks	Yes	10000	02042021
003	Watch	Yes	2000	31032021

***Figure 6-19.** Transactions with event*

As shown in Figure 6-20, the microservices publish and subscribe to an event via an event broker and event store. Each service publishes an event to the event broker, and other services subscribe to an event as and when it is published.

Here are the steps a place an order transaction:

1. The customer places an order, and the Order service initiates an order confirmation transaction, called Begin Transaction.
2. The Order service publishes an event to check the inventory by passing the product ID.
3. The Inventory service subscribes to an event from the event broker and checks the stock level against the product.
4. If the stock available, then the Inventory service publishes an event after reserving a stock.
5. The Order service subscribes to an event and confirms the order.
6. The Order service publishes an event to update a stock.
7. The Order service publishes an event for confirming an order.
8. The Inventory service subscribes to an event and updates the stock level.
9. The Customer service subscribes to the event for a confirmed order and updates the details.
10. The transaction ends.

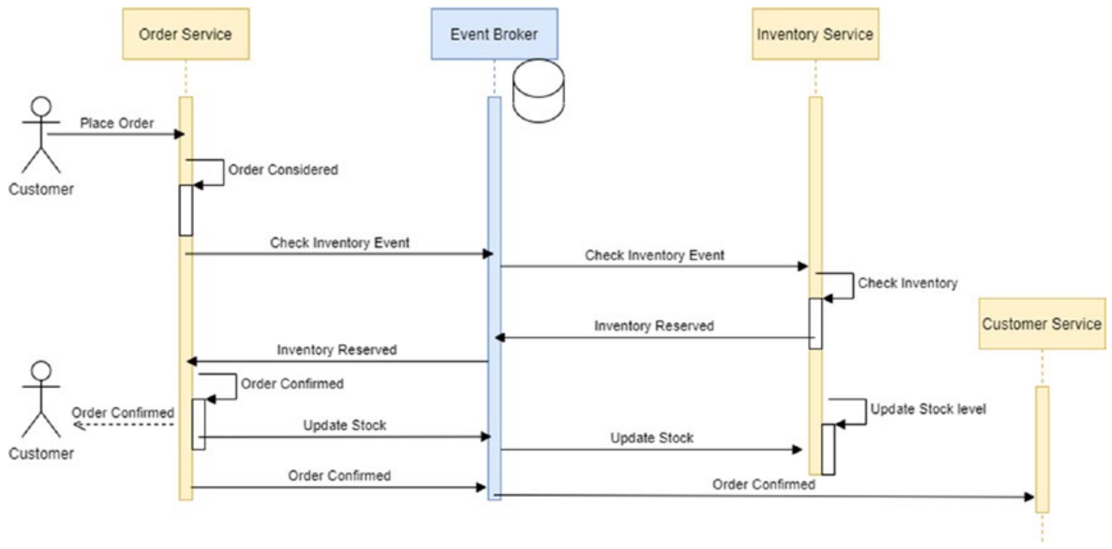
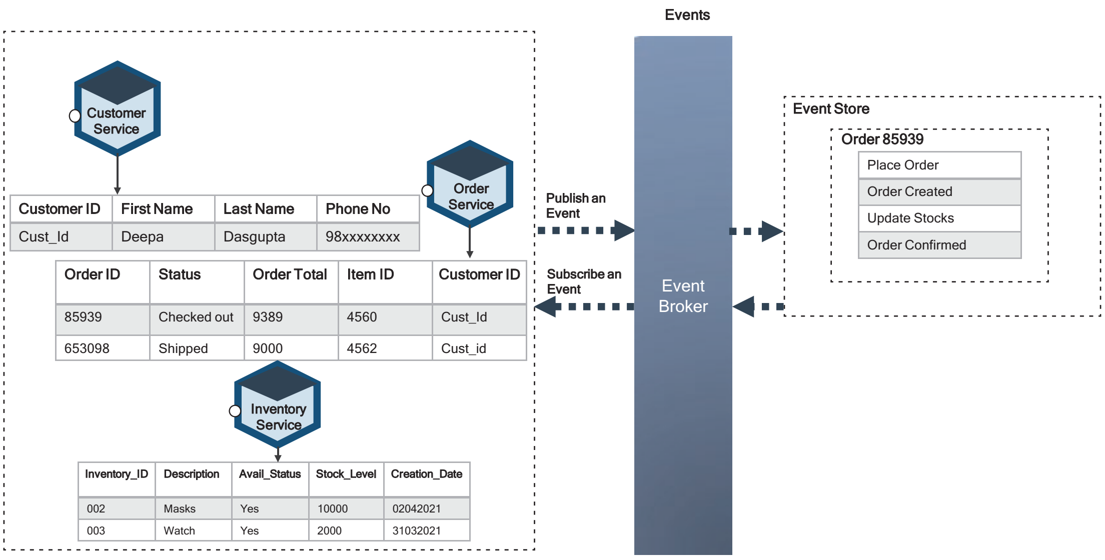


***Figure 6-20.** Event transaction sequence diagram*

Here each service updates its database and publishes an event, and the event broker saves each event in the event store. All these transactions do not adhere to ACID properties, but all follow eventual consistency properties. Throughout this entire transaction, atomicity is important. To manage the atomicity of your transaction, your event store plays an important role. For example, for order creation, you need to store an order in the order service database and publish an event to the event broker; these two things should happen atomically. If the service fails after one task, then it becomes an inconsistency in a transaction. To achieve this inconsistency, you need to manage the event store table to store all kinds of events that occur in the whole transaction.

The event sourcing and event table persist all kinds of events in a transaction; if any transaction fails in between, the service can construct a state by using the event store, as shown in Figure 6-21, and each service publishes and subscribes to an event by using the event broker.



Customer ID	First Name	Last Name	Phone No
Cust_Id	Deepa	Dasgupta	98xxxxxxxx

Order ID	Status	Order Total	Item ID	Customer ID
85939	Checked out	9389	4560	Cust_Id
653098	Shipped	9000	4562	Cust_id

Inventory_ID	Description	Avail_Status	Stock_Level	Creation_Date
002	Masks	Yes	10000	02042021
003	Watch	Yes	2000	31032021

***Figure 6-21.** Event store in a transaction*

One way of achieving an event-driven transaction is to use the saga pattern and CQRS, which are explained in Chapter 4.

Event-Driven Microservices Interaction

At a high level, there are two approaches to getting microservices to work together toward a common goal.

- Orchestration with synchronous
- Choreography with asynchronous

Orchestration entails actively controlling microservices like a conductor directing the musicians of an orchestra. Choreography entails establishing a pattern that microservices follow as the music plays, without requiring supervision or instructions.

The synchronous communication and orchestration across microservices are managed by the orchestrator. The orchestrator is not a new concept; it has existed since the SOA and ESB implementations were introduced. The ESB acts as an orchestrator and orchestrates across heterogeneous systems in an enterprise ecosystem. Let's look at an example of a utility payment from the banking web application. You want to pay an electricity bill through your web app, so you initiate the transaction by clicking the utility payment link. The web application sends

SOA requests to the ESB, and the ESB must orchestrate between the core banking application and utility payment gateway. It follows these steps:

1. The ESB calls the core banking API to credit an amount.
2. The ESB calls the utility gateway to issue a request for the payment.
3. The utility gateway responds with success.
4. The ESB calls the core banking API for confirmation and credits an amount in your savings bank account.
5. If any failure occurs in the utility gateway, then ESB needs to call the core banking API to reverse a transaction.

You may face several challenges in the microservice implementation related to how microservices interact with each other to complete a business use case. Choosing between orchestration and choreography will make a difference in how seamlessly the services function.

In an orchestra, each musician is awaiting a command from the conductor. They are each expert in playing their instrument, and yet they'd be collectively lost without the conductor. In orchestration, one service or any tools like Netflix Conductor or Uber Credence handle all communication between microservices and direct each service to perform the intended functions.

The downside of orchestration is the orchestrator is a single point of failure, and the controller needs to directly communicate with each service and wait for each service's response. These interactions are occurring across the network. Invocations and I/O blocking take longer, block threads, and impact service availability. In orchestration, each service is tightly dependent on other services, and they are synchronous, and each service must explicitly receive and respond to requests to make the whole service work; failure at any point could stop the process. The orchestration could rely on RESTful APIs. For some use cases, the orchestration is best suited; an example is Netflix using Conductor.

A choreography-based approach is like the dancers listening to the music and making the necessary moves because all dancers follow the same choreography. In this approach, you will avoid dependencies. So, each service works loosely coupled and independently.

In choreography, as shown in Figure 6-22, the event broker exchanges the information between microservices. It is like a fire-and-forget-it, decentralized way of broadcasting data known as *events*, and everything happens asynchronously, without waiting for a response. Each service observes its environment and subscribes to the message events to that channel and will know what to do from there.

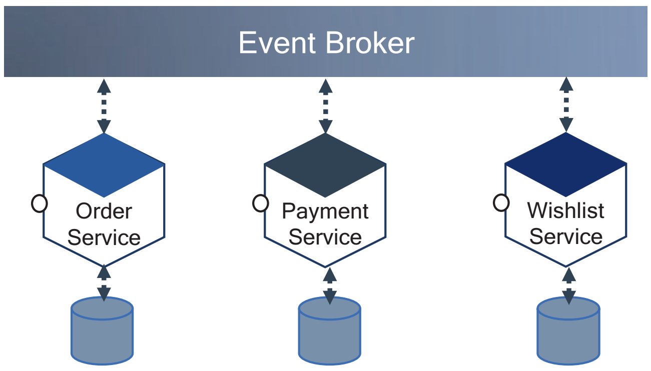


Figure 6-22. *Choreography in services*

The choreography isolates the microservices; this means if one service fails or fails to respond, it does not impact as a whole use case. You can use a various pattern like a circuit breaker to handle the failure.

The following are the advantages of choreography:

- Processing is faster as there is no requirement of the central conductor.
- There is no single point of failure.
- Each service is loosely coupled and is not aware of each service; therefore, it's easy to add and remove services.
- This resonated well with cloud native architecture.
- You can use several patterns like circuit breaker, CQRS, or event sourcing for effective management of interactions across microservices.

The disadvantages are that it is complex and requires special skills to configure and manage eventing capabilities.

Interaction Between Microservices

Managing across a few microservices is easy and does not require any extra management layer, but when microservices grow in your enterprises, you may face many challenges of managing services in a cloud native environment.

Many loosely coupled services that are independently and frequently changing do promote agility but introduce a lot of management challenges. Some of these challenges are traffic management, communication security, communication failures, etc. Many of these challenges can be resolved directly by writing code in a service, but embedding these configurations in code poses a lot of challenges because these configurations are complex and extremely error prone.

As shown in Figure 6-23, there are two ways you can handle these challenges without embedding code in a service for API interaction and event interaction across microservices.

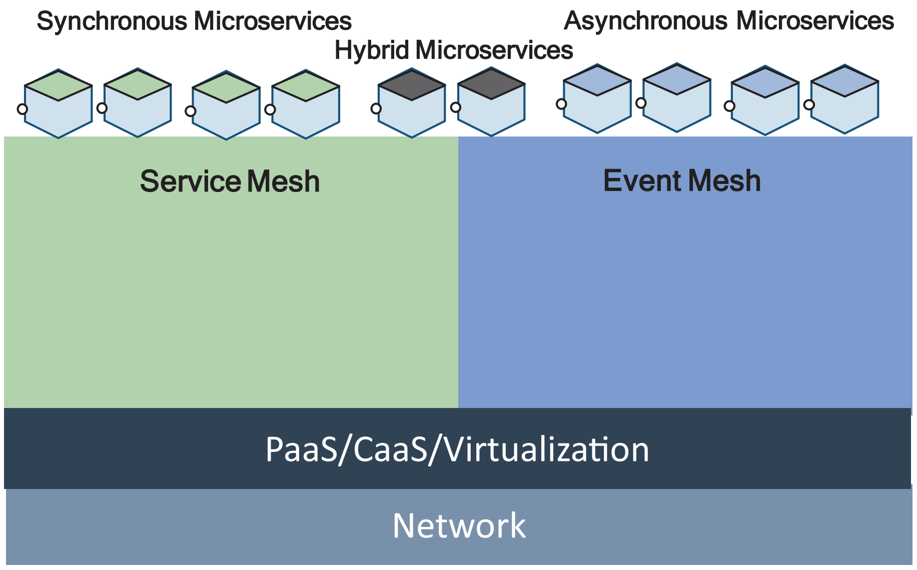


Figure 6-23. *Mesh architecture*

- *Service mesh*: Provides connection-level routing and traffic management for synchronous communication (HTTP(s)) interaction through sidecar injection into Kubernetes pods. The service mesh is focused on routing connections between endpoints by hijacking the connection requests and overriding the connection requests from the microservices.

- *Event mesh*: Handles asynchronous event-driven routing of information between microservices. It intelligently routes events between the event brokers, allowing the cluster or brokers to appear as a single virtual event broker.

The mesh frameworks allow you to observe, secure, and connect microservices. They don't establish connectivity between microservices but instead have policies and control applications on top of the existing network to govern how microservices interact. These frameworks shift the implementation logic out of the microservices code and move it to the network.

The service mesh and event mesh are not mutually exclusive, and you may consider implementing both depending on the use cases.

Service Mesh

A gateway centralizes the configuration and routes requests to the relevant microservices, and it can handle orchestration but with limitations.

The advent of cloud native architecture and the use of container and microservices platforms create a need for an orchestrator. The containers and orchestrators, coupled with the microservices for their speed to market and silos of the development pods, lead to service sprawl. The ability to run several distributed services requires a service mesh.

A service mesh solves the problems where microservices communicate using APIs. A service mesh uses the sidecar pattern to establish communication between microservices and ensures that the communication among containerized and often ephemeral application infrastructure services is fast, reliable, and secure.

Service Mesh Implementation

As shown in Figure 6-24, a service mesh will have a control plane to program the mesh and sidecar and serve as the control point for securing, observing, and routing decisions between services. The control plane transfers configurations to the proxies, and each proxy intercepts all inbound and outbound traffic. By intercepting traffic, the proxies will inject behavior into the communication flows between microservices.

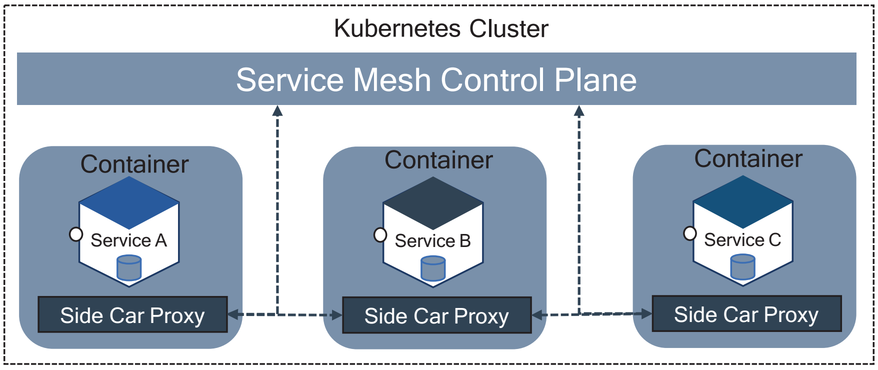


***Figure 6-24.** Service mesh architecture*

The following behavior will be handled by the service mesh:

- Traffic shaping with dynamic routing controls
- Resiliency support for service communication such as circuit breakers, timeouts, and retries
- Observability of traffic between services
- Tracing of communication flows
- Secure communication between services

In Figure 6-24, all services A, B, and C are executed through sidecar proxies. By having communication routed between the proxies, the proxies serve as a key control point for performing a task such as initiating Transport Layer Security (TLS) handshakes for encrypted communication with the previous behaviors.

Service meshes route data based on the connection URL and the ability to redirect a connection based on the content routing rules against the URL and HTTP header information.

Services meshes are one layer of your infrastructure and don't provide all that you need. They do give you the ability to bridge the divide between your infrastructure and your application.

Advantages and Disadvantages of Service Meshes

The advantages of service meshe is that they offer distributed debugging, provide topology and dependency management, participate in application lifecycle management, and participate in service and product management, offer deeper observability, provide multitenancy, have multicluster meshes, allow advanced circuit breakers with fallback paths, etc.

The service mesh provides a simpler network configuration for the microservices but with some caveats.

- The service mesh has no support for asynchronous events or stream processing.
- Most traffic and network services apply only to synchronous communication and the HTTP and GRPC protocols.
- A service mesh limits the connection-oriented routing and targeting of the transport connection, not the routing of actual data.

Microservices are using diverse message interaction patterns including publish/subscribe, point-to-point, push-with-reply, queuing, etc. In today's world, the microservices require a higher throughput and lower latency than you can meet by using Kubernetes clusters. It takes choreography rather than orchestration processing. In cloud native architecture, microservices require event-driven architecture because they require eventing capability, performance, and real-time processing that goes beyond a Kubernetes cluster. Here, you require an event mesh.

Event Mesh

A cloud native modern enterprise embraces event architecture, and every event-driven application requires a robust central system to move events quickly, reliably, and securely from publisher to subscriber.

An event mesh is an architectural layer that dynamically routes events from one microservice to another irrespective of deployment location. The event mesh is a key enabler for event-driven architecture. An event mesh is a dynamic infrastructure that propagates events across disparate cloud platforms and performs protocol translation.

A single event broker can handle only a certain volume of requests and microservices. There are different ways to scale, and one way is the event mesh.

In an event mesh, there is no underlying technology such as Kubernetes, and event brokers are designed to operate with or without a cloud. Event meshes route data based on topics and are transported with the event payload. It is a dynamic infrastructure that propagates events across multicloud platforms and performs protocol translations.

Figure 6-25 illustrates elements of an event mesh, and events can flow bi-directionally across the microservices irrespective of where they are deployed, whether it is in the same cloud, multicloud, or hybrid cloud.

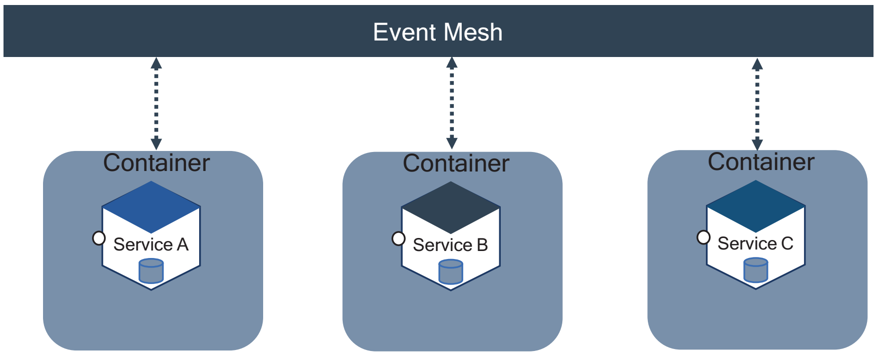


Figure 6-25. *Event mesh architecture*

An event mesh is configured along with an event broker. The event mesh translates any application into different languages and is deployed in different clouds. It publishes an event and lets the subscriber of another application deployed in a different cloud subscribe to that event . It also can be a different API altogether. This helps to separate the configuration from the business logic in microservices.

Characteristics of Event Mesh

The following are the characteristics of an event mesh:

- Made up of interconnected event brokers
- Environment agnostics; can deploy in any public cloud, PaaS, or non-cloud environment
- Dynamic and intelligent routing
- Security and WAN optimization

Event Mesh Capabilities

The following are the event mesh capabilities, and they are required for modern-day architecture:

- Supports publish and subscribe for events in various protocols such as Kafka, Knative, HTTP, AMQP, etc.
- Support for multiprotocol bridges between disparate events, microservices, and messaging platforms
- Supports on-premises and multicloud deployment to provide a uniform infrastructure
- Secure transmission of event messages

How Do Event Meshes Work?

Subscribers of events are connected to the event broker and register with the topic and configure the event type. When event messages arrive in the event broker, Event mesh routes them to the subscriber based on their subscriptions. In Figure 6-26, `/Inventory` would go to the Inventory service, the event with `/Payment` would be routed to the Payment service, and the `/dispatch` event would be routed to the Dispatch service.

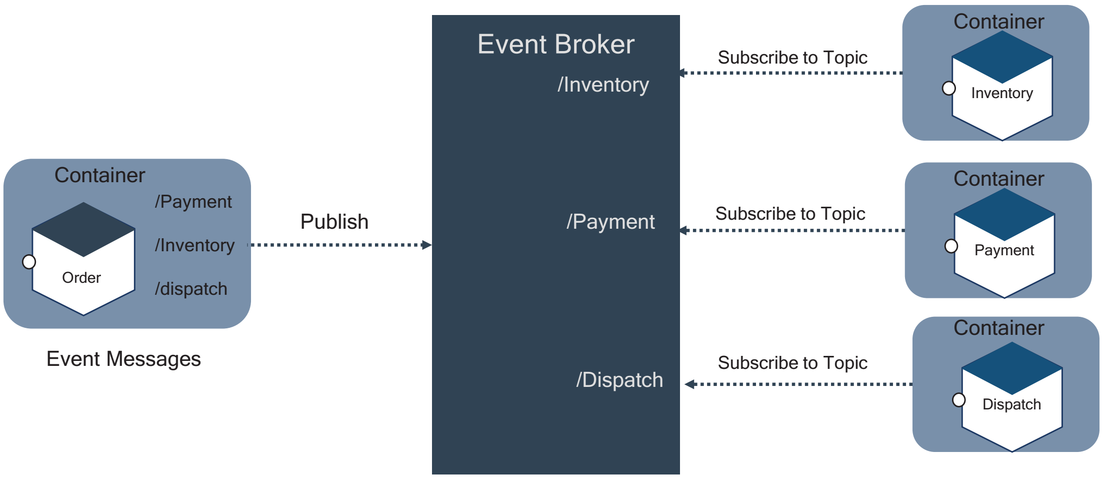


Figure 6-26. *Event mesh implementation*

The consuming microservices such as Inventory, Payment, and Dispatch are processing asynchronously and potentially in parallel. Each service uses processing overhead only when the event broker forwards an event based on the subscriptions. The event broker abstracts the routing of events between the publisher and the subscriber. All event brokers persist messages and don't need to be available when the event is initially published. They have the option of receiving events that were published while they were offline, but this impacts the customer experience.

Event Mesh in a Cluster of Brokers

I explained how the event broker manages the routing rules in a single broker. The complexity arises when you have a cluster of brokers, and each message is subscribed in a separate event broker. How will you manage this? The one option is to embed code in your microservices or configure them in the event broker to manage in a cluster. This is where an event mesh is useful to coordinate and collaborate across multiple event brokers to streamline the routing and publishing and subscription.

In Figure 6-27, the Order microservice sends a message to Event Broker 1 and asks for an order from the location to check inventory, local distribution, and local dispatch. All four microservices are deployed in the separate cloud with respective event brokers. For example, the Inventory microservice asks for any order microservices to check a local inventory where the order is originated. All brokers are connected with the event mesh, so that subscription is forwarded to the other event brokers in another cloud. When the matching event is published to Event Broker 1, the event mesh will forward it to Event Broker 2, because no microservices are connected to another event broker for this request, but other microservices are required to subscribe to other order events once the inventory is confirmed. The Inventory microservices checked the inventory and published an event to Event Broker 2 for availability and an event mesh is routed to Event Broker 1.

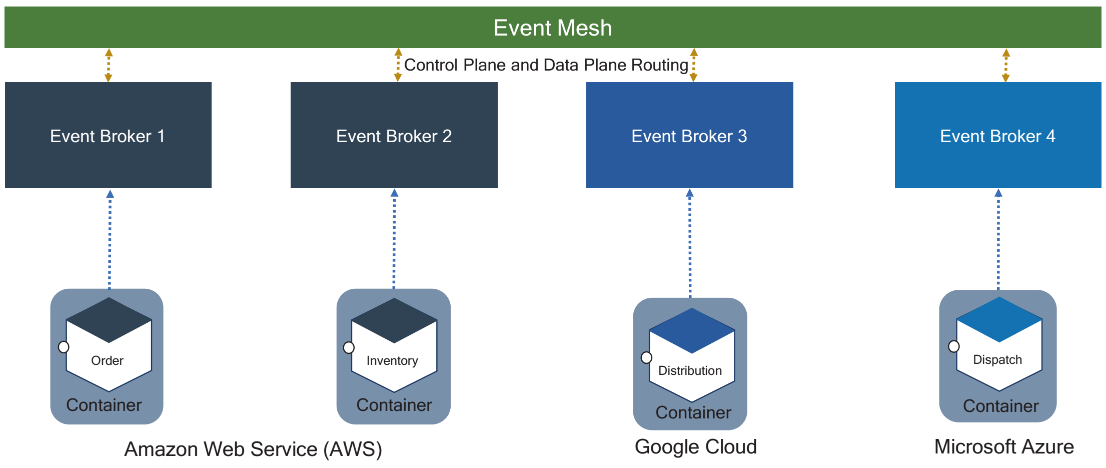


Figure 6-27. *Event mesh across cloud providers*

The Order microservice confirms the order and publishes an event called `confirm`. The event mesh routes to Event Broker 2, and the Inventory microservice subscribes to `confirm` events, updates stocks, and publishes an event called "confirm with the item" to Event Broker 2. The event mesh checks Event Broker 3 and Event Broker 4 for event subscription. The Distribution microservice subscribes to "confirm with the item," and the event mesh routes to Event Broker 3. The distribution microservice consumes and is ready for dispatch by publishing events to "dispatch" to Event Broker 3. The event mesh routes to Event Broker 4 for dispatch. All these microservices and event brokers are deployed in multicloud environments, and the event mesh can route within the cloud or multicloud environment.

While each event broker provides its local routing table based on topics, the control plane of the event mesh dynamically and transparently extends that routing information among all interconnected event mesh broker nodes like the Internet does for IP routes.

This is the way the event mesh makes many event brokers look like a single virtual event broker; it uses broker routing protocols to intelligently, dynamically, and efficiently route events.

Event Mesh's Control Plane

Not all event brokers enable an event mesh. The clustering of event brokers to provide high availability or local horizontal scaling is not an event mesh. If the local cluster does not provide intelligent routing between other clusters, then the event broker doesn't constitute an event mesh. Every event broker that does enable an event mesh provides a control plane.

The event broker must provide the tooling and capabilities like Kubernetes for service meshes. The control plane must provide high availability (HA) for event nodes and disaster recovery (DR) for broker nodes. The characteristics of the control plane are as follows:

- Configuration and monitoring of event broker nods in an HA cluster
- Dynamic real-time updates for routing tables on all event brokers nodes and clusters

The service mesh and event mesh work in different environments and for different use cases, but both can collaborate in an application. For example, a few microservices are required to work with synchronous HTTPS or GRPC, and a few microservices require an event-driven capability. In this case, the service mesh can be used for synchronous microservices, and the event mesh can be used for event-driven microservices

Box- and Port-Style Event-Driven Architecture

The *box- and port-style pattern* supports the observability of microservices or components. It provides a needed level of agility, timelines, information availability, and simplicity in a cost-effective way and provides the surrounding observability component to business services. This observability component can be deployed in the containers, cloud, and on-premises. Figure 6-28 illustrates how business components or microservices are wrapped with a componentless pattern and with observability.

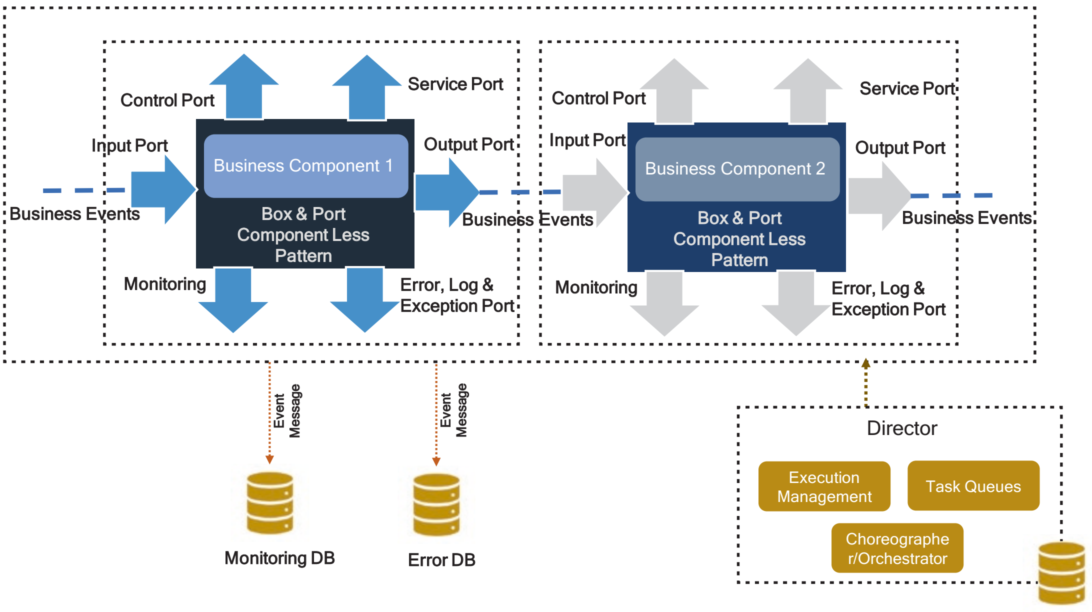


Figure 6-28. *Box- and port-style component less with observability*

On a distributed event-based platform, events are passed from one component to another component by using the message channels or events. This pattern externalizes all observability, interaction with other components, etc., from the core business logic. The ports are enabled for interaction between services or components.

As shown in Figure 6-28, the box and port are technical infrastructure components and support any kind of services irrespective of language and platform. The responsibility of this pattern is to convert message formats and capture errors and exceptions, log messages and traces, configure to topics, etc. This component supports observability to track each business component and supports multiple protocols to interact with business services. The only responsibility of the business service/component is business functionality, and the remaining technical details are maintained by the box and port.

The wrapper component provides an essential level of instrumentation, which means that it is possible to observe the processing of any component from outside. For example, the monitoring port is used to publish performance and other statistics to the dashboard. The operational monitoring includes but not limited to the following:

- Heartbeat monitoring
- The actual latency of every input-to-output event flow as well as average latency over a time window

- The actual wait time for the input event
- Error rates, etc.

The director is responsible for configuring an application into the technical component at startup time and lasts until the component or service shuts down. It does this by using the event delivery platform through the control port. The wiring instructions can be controlled by the director through the control port.

Components or services work with input and output ports. This is quite different from a normal object-oriented design, where one object can invoke a method on another object. Components do not invoke methods, nor do they call services or other components in an event-driven architecture. They send events to some other component, and then they continue processing input events. A response to a request may arrive asynchronously at an input port, at which point the component correlates that response to an earlier request and acts accordingly.

Characteristics of Box- and Port-Style Architecture

The following are the characteristics of box and port style:

- *Real-time operational behavior*: It can change the behavior of the system to dynamically react to incoming events.
- *Observations*: It observes all kinds of behavior and generates alerts when such behavior occurs and predicts failures based on the historical data.
- *Information dissemination*: It sends the right information to the right recipient with personalization.
- *Active and predictive diagnostics*: It can diagnose a problem that occurred based on historical data, predict the details, send alerts to the recipients, and send details to the dashboard.
- *Autoscaling*: Dynamic load distribution patterns such as queue with multiple subscribers are used to ensure that the workload is evenly distributed across all the components. The dashboard spins up the component by sending an instruction to the component with the manual intervention of the configuration file for Docker images.

This architecture style provides the most benefit to the existing legacy component-based systems where the observability details are hard-coded as part of the business logic in a service.

DevOps for Events

Event-driven cloud native architecture has gained a lot of attention; therefore, you need to have a DevOps pipeline for an event-driven architecture. Event-based systems could be comprised of many different enabling technologies such as Kafka, NATS, Solace, Confluent, microservices, serverless, CDC, etc.

The generic guidelines for DevOps are as follows:

- Treat events as API contracts; other systems may be reliant on event producer.
- Use schemas to encode events, with shared schema registries for access.
- Treat event configuration as code. The topics should be created by using scripts, and the event schema must be checked into your Artefactory tool.
- Use infrastructure as code to automate the configuration, installation, etc.

Event Security

In a distributed event-driven architecture, you must balance data democratization with the protection of sensitive data. The events must be encrypted between the publisher and the subscriber.

These are the types of encrypting events:

- Events in transit should always be encrypted using industry-standard encryption methodologies such as SSL/TLS.
- Disks/storage holding past events should always be encrypted in the file system or event store database.
- File-level encryption is the most secure way to encrypt the data, but it is more expensive; therefore, consider this only for sensitive data.

Field-Level Encryption Consideration

All data should be encrypted in transit and at rest. The level of field encryption depends on risk tolerance. If the topic contains no sensitive data, then do not use field-level encryption; therefore, you need a balance between security and performance. Figure 6-29 gives a clear strategy to choose what level of encryption is required in your system.

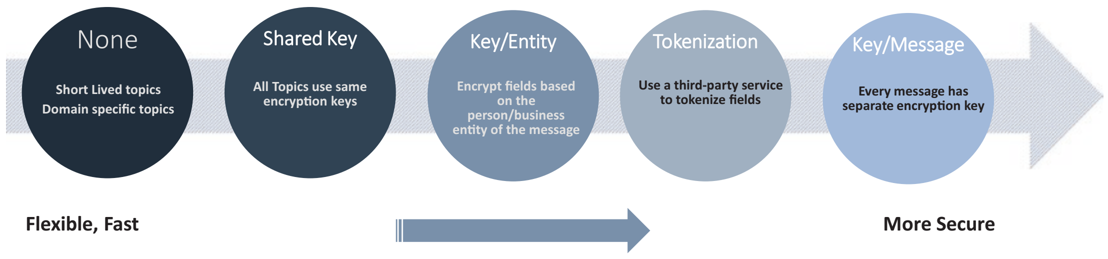

None
Short Lived topics
Domain specific topics
Shared Key
All Topics use same encryption keys
Key/Entity
Encrypt fields based on the person/business entity of the message
Tokenization
Use a third-party service to tokenize fields
Key/Message
Every message has separate encryption key
Flexible, Fast
More Secure

Figure 6-29. *Encryption level*

Cloud Events

In cloud native architecture, you can find events everywhere, and each event is published by a publisher with different event specifications. There is no common, standard way of publishing events in enterprises. This leads to constant learning across teams, leads to more error, and your Confluence documents might be full of event specifications across agile pods to refer to. This limits the potential libraries, tooling, and infrastructure to aid the delivery of events across systems in an enterprise.

As explained earlier in this chapter, an event includes context and data about an occurrence, and each occurrence is uniquely identified by the data of the event. The event represents facts and no destination, whereas the message conveys intent and transporting data from source to destination.

Events can be delivered through various industry-standard protocols, for example, AMQP, HTTP, MQTT, SMTP, and open-source protocols such as NATS, Kafka, or cloud vendor protocols, AWS Kinesis, Azure Event Grid, Google Pub/sub, etc.

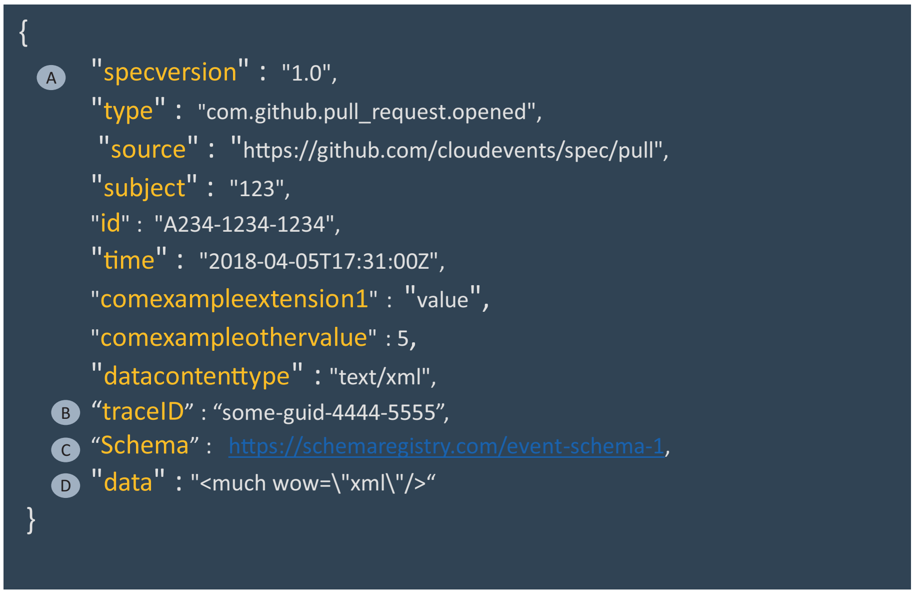
The objective of the Cloud Events specification is to define the interoperability of event systems that allow services to produce or consume events, whereas both the teams can be developed and deployed independently.

The Cloud Events specification contains a set of metadata, known as *attributes*, about the event being transferred between systems and how those pieces of metadata should appear in messages. This metadata contains a minimal set of information for routing to the respective services and helps to process the events. Along with this metadata, there is also a specification to serialize the events in different formats like JSON, and protocols like HTTP, AMQP, etc.

The Cloud Events specification defines four kinds of protocol elements.

- *Base specification*: Defines abstract information of attributes and associated rules.
- *The extensions*: Includes use-case-specific and overlapping sets of extension attributes and associated rules.
- *Event format encoding*: Defines how the information model of the base specification with an extension is encoded for mapping the header and payload of a protocol
- *Protocol binding*: Defines the application protocol transport frame, in the case of HTTP to the HTTP messages.

As shown in Figure 6-30, the Cloud Events specification ensures a consistent approach to traceability, schema version, origin, etc. It is just a standard and extended to meet the needs of your enterprise systems. For more details, refer to `https://cloudevents.io/`.



```
{
A   "specversion" : "1.0",
    "type" : "com.github.pull_request.opened",
     "source" : "https://github.com/cloudevents/spec/pull",
    "subject" : "123",
    "id" : "A234-1234-1234",
    "time" : "2018-04-05T17:31:00Z",
    "comexampleextension1" : "value",
    "comexampleothervalue" : 5,
    "datacontenttype" : "text/xml",
B   “traceID” : “some-guid-4444-5555”,
C   “Schema” : https://schemaregistry.com/event-schema-1,
D   "data" : "<much wow=\"xml\"/>“
}
```

Figure 6-30. *Cloud event metadata*

- *A*: The spec version is the version of the specification that the message is encoded to. This should match the Cloud Events specification. Between this marker and B are some of the Cloud Events spec fields you might find.
- *B*: This field is an "extension" of the Cloud Events specification. Here, the trace ID is used to track the event from place to place, usually tied to the origin. For example, a web request might be the originator of this trace ID, and all subsequent messages that are created throughout the system have this same trace ID. To define this tracer, consider the OpenTracing initiative.
- *C*: This schema is another Cloud Events extension, which declares how the data field is laid out. This allows the message to be decoded by services against a schema registry.
- *D*: The data field contains the actual important content, or business information, about an event. This data can be any format you like but should conform to a schema. The way data is structured within the data element is completely independent of the Cloud Events specification.

Summary

Constantly changing, real-time business needs demand cloud native transformation. The world is not slowing down, so your best bet is to identify ways you can cost effectively and efficiently upgrade your enterprise architecture to keep up with the times.

Events can float around on an event mesh to be consumed and acted upon by your microservices. Architects and engineers need real-time implementation details that help you to work together to achieve the real-time, event-driven goals. In this chapter, you learned all the details of an event-driven architecture and its implementation.