# Building event-driven architectures with AWS


## ABSTRACT

Event-driven architectures are an architecture style that uses events and asynchronous communication to loosely couple an application's components. Event-driven architectures can help you boost agility and build reliable, scalable applications. This guide introduces and summarizes key concepts around event-driven architectures. It covers patterns within event-driven architectures and identifies the Amazon Web Services (AWS) services that are commonly used to implement them. Finally, it provides best practices for building event-driven architectures, from designing event schemas to handling idempotency

SECTION 1

# Event-driven architecture overview and key concepts

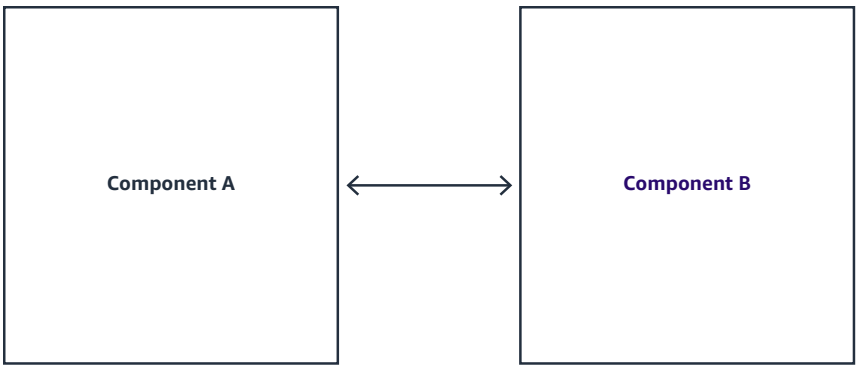
## Tight vs. loose coupling

**Coupling** is the measure of dependency each component of an application has on one another.

There are various forms of coupling that systems may share:

- Data format dependency (binary, XML, JSON)
- Temporal dependency (the order in which components need to be called)
- Technical dependency (Java, C++, Python)

Tightly coupled systems can be particularly effective if the application has few components or if a single team or developer owns the entire application. However, when components are more tightly coupled, it becomes increasingly likely that a change or operational issue in one component will propagate to others.

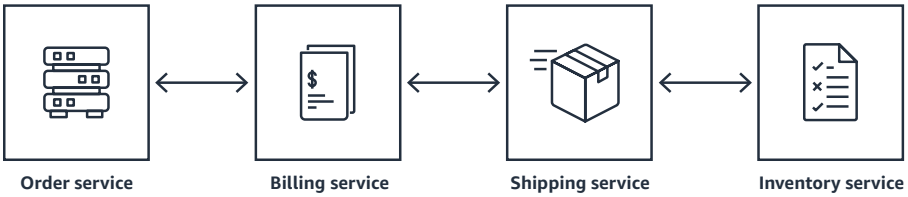


For complex systems with many teams involved, tight coupling can have drawbacks.

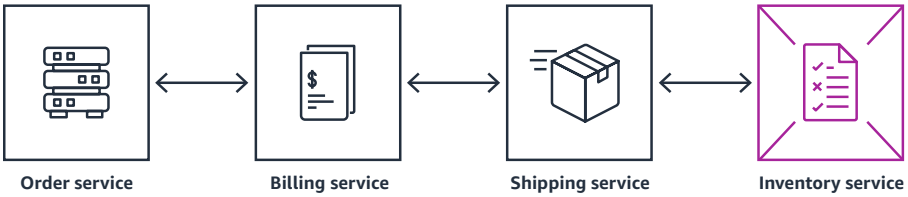
When components are tightly interdependent, it can be difficult and risky to make changes isolated to a single component without affecting others. This can slow down development processes and reduce feature velocity.

Tightly coupled components can also affect an application's scalability and availability. If two components depend on one another's synchronous responses, a failure in one will cause the other to fail. These failures can reduce the application's overall fault tolerance.

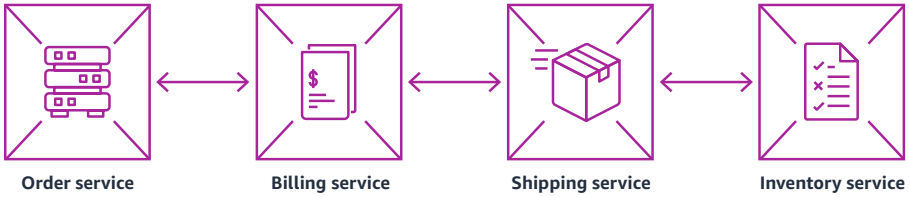
For example, an ecommerce application may have multiple services (orders, billing, shipping, inventory) and make a synchronous chain of calls to these services.




## A failure in one of these services...



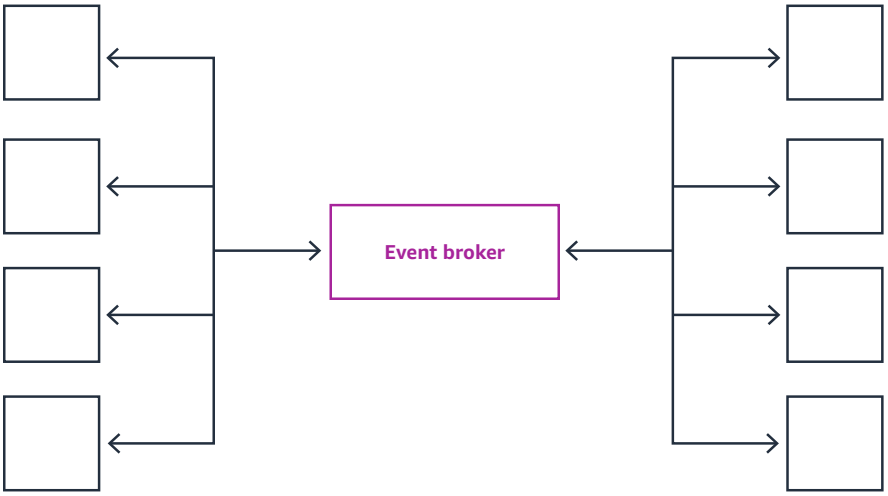

## ...will impact all the others.



**Reducing coupling** is the act of reducing interdependency between components and the awareness each component must have of one another.

Event-driven architectures achieve loose coupling through asynchronous communication via events. This happens when one component doesn't need another to respond. Instead, the first component may send an event and continue without impact should the second component delay or fail.

When communicating with events, components only need to be aware of the independent events. They don't require knowledge of the transmitting component or any other components' behavior (e.g., error handling, retry logic). So long as the event format remains the same, changes in any single component won't impact the others. This allows making changes to an application with less risk. When asynchronous events abstract components from one another, complex applications become more resilient and accessible.



## Core benefits of interdependence

A shift in mindset is required when building an event-driven architecture due to the unique characteristics and considerations of asynchronous systems. Yet, this architecture style offers important benefits for complex applications:

- **Build and deploy applications independently in loosely coupled applications.** Development teams working on individual services have fewer dependencies. Changing one service will have less risk of impacting others.
- **Build new features using events without changing existing applications.** Since components emit events, event-driven architectures are easily extensible. Events can also be analyzed for business reports and audits.
- **Scale and fail components independently in loosely coupled components.** Applications with loosely coupled components have fewer single points of failure, as well as increased resiliency.

## Key concepts of event-driven architectures

An event is the signal of a changed state, such as an item in a shopping cart or a credit card application. Events occur in the past (e.g., "OrderCreated" or "ApplicationSubmitted") and are immutable, meaning they can't be changed. This helps in distributed systems because there are no changes across components to keep in sync.

Events are observed, not directed. A component that emits an event has no particular destination nor awareness of downstream components that may consume the event. Event-driven architecture possesses these key components:

- **Event producers** publish events. Some examples include front-end websites, microservices, Internet of Things (IoT) devices, AWS services, and software-as-a-service (SaaS) applications.
- **Event consumers** are downstream components that activate in events. Multiple consumers may be found in the same event. Consuming events includes starting workflows, running analyses, or updating databases.
- **Event brokers** mediate between producers and consumers, publishing and consuming shared events while mitigating the two sides. They include event routers that push events to targets and event stores from which consumers pull events.

## Example use cases

- **Microservices communication:** You can build, deploy, and scale microservices independently with event-driven architectures in microservices-based applications, such as the ecommerce application previously mentioned.
- **IT automation:** Your infrastructure with AWS services already generates events, including Amazon Elastic Compute Cloud (Amazon EC2) instances, state-change events, Amazon CloudWatch log events, AWS CloudTrail security events, and many others. You can use these events to automate your infrastructure for validating configurations, reading tags in a log, auditing user behavior, or remediating security incidents.
- **Application integration:** Events allow you to integrate other applications. You can send events from on-premises applications to the cloud and use them to start building new applications. Integrating SaaS applications enables you to create custom workflows using those events. And you can use SaaS applications for services like customer relationship management, payment processing, customer support, or application monitoring and alerting.

SECTION 2

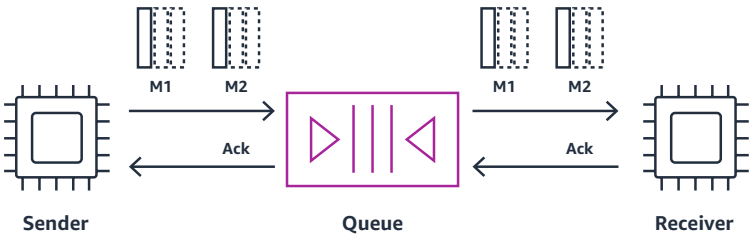
# Common patterns in event-driven architectures

This section details the building blocks and patterns commonly found in event-driven architectures. These patterns enable decoupled, asynchronous communication between producers and consumers—while allowing for the unique characteristics that are increasingly necessary to fulfill various application requirements.

## Point-to-point messaging

**Point-to-point messaging** is a pattern in which producers send messages typically intended for a single consumer. Point-to-point messaging often uses messaging queues as its event broker. Queues are messaging channels that allow asynchronous communication between a sender and receiver. Queues provide a buffer for messages in case the consumer is unavailable or needs to control the number of messages processed at a given time. Messages persist until the consumer processes them and deletes them from the queue.

In microservices applications, asynchronous, point-to-point messaging between microservices is referred to as "**dumb pipes**."



Services like **Amazon Simple Queue Service** (Amazon SQS) and **Amazon MQ** are commonly used as message queues in event-driven architectures. You can also use **asynchronous AWS Lambda invocations**.

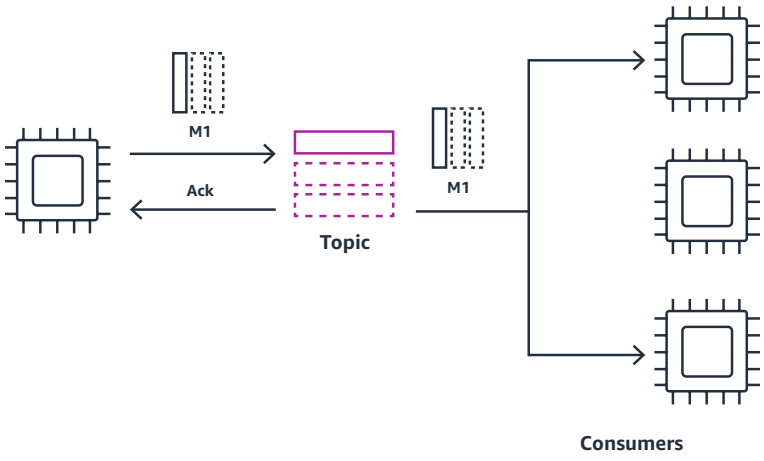
In synchronous invocations, the caller waits for the AWS Lambda function to complete its execution and the function to return a value. In asynchronous invocations, the caller places the event on an internal queue, which is then processed by the AWS Lambda function.

With this model, you no longer need to manage an intermediary queue or router. The caller can move on after sending the event to the AWS Lambda function. The function can send the result to a **destination**, with configurations varying on success or failure. The internal queue between the caller and the function ensures that messages are durably stored.

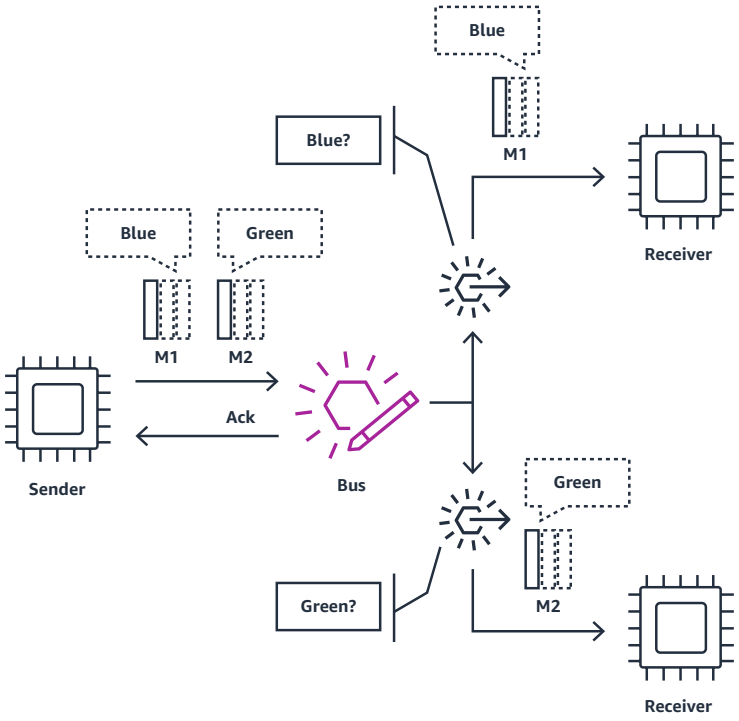
## Pub/sub messaging

**Pub/sub messaging** is a way in which producers send the same message to many consumers. Whereas point-to-point messaging usually sends messages to just one consumer, publish-subscribe messaging allows you to broadcast messages and send a copy to each consumer. The event broker in these models is frequently an event router. Unlike queues, event routers typically don't offer persistence of events.

One type of event router is a topic, a messaging destination that facilitates hub-and-spoke integrations. In this model, producers publish messages to a hub, and consumers subscribe to the topics of their choice.



Another type of event router is an event bus, which provides complex routing logic. While topics push all sent messages to subscribers, event buses can filter the incoming flow of messages and push them to different consumers based on event attributes.




You can use **Amazon Simple Notification Service** (Amazon SNS) to create topics and **Amazon EventBridge** to create event buses. Amazon EventBridge supports persisting events through its **archive** functionality. **Amazon MQ** also supports topics and routing.

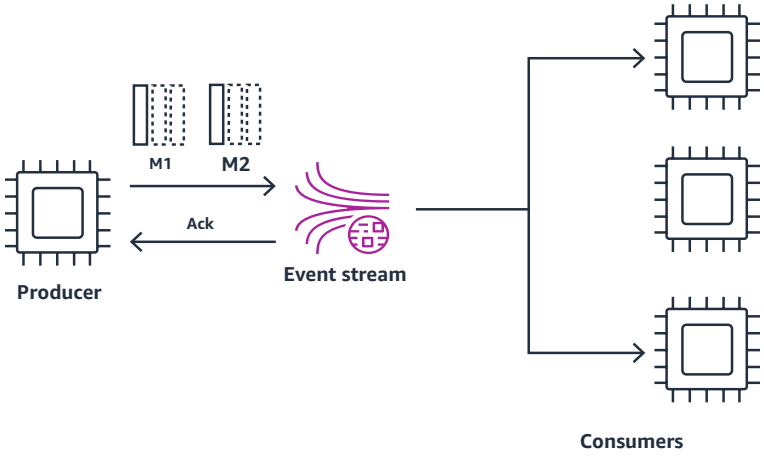
## Event streaming

The use of streams, or continuous flows of events or data, is another method of abstracting producers and consumers. In contrast to event routers, though comparable to queues, streams typically require consumers to poll for new events. Consumers maintain their unique filtering logic to determine which events they want to consume while tracking their position in the stream.

Event streams are continuous flows of events, which may be processed individually or together over a period of time. A rideshare application, which streams a customer's changing locations as events, is an example of event streaming. Each "LocationUpdated" event exists as a meaningful data point used to visually update the customer's location on a map. It could also analyze location events over time to provide insights, such as the driver's speed.

Data streams differ from event streams in that they always interpret data over time. In this model, individual data points, or records, aren't independently useful. Data streaming applications are often used to either persist the data after an optional enrichment or to process the data over a preset time period to derive real-time analytics. An example could be IoT device sensor data streaming. Individual sensor reading records may not be valuable without context, but records collected over a period of time can help tell a richer story.

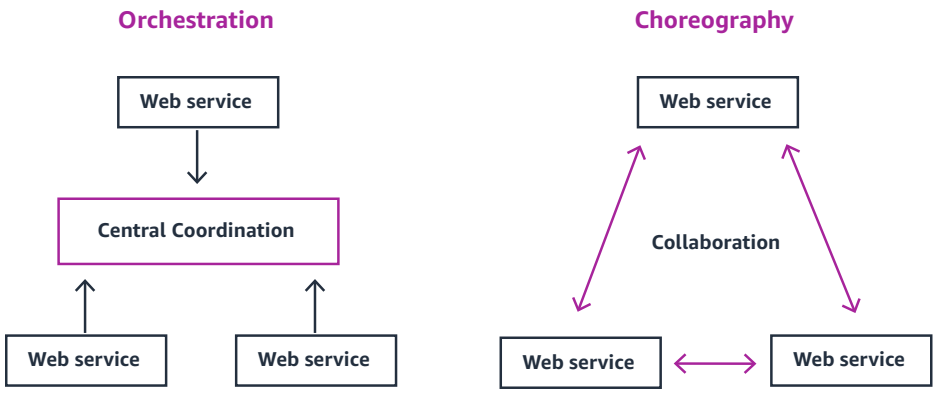
**Amazon Kinesis Data Streams** and **Amazon Managed Streaming for Apache Kafka** (Amazon MSK) can be used for event- and data-streaming use cases.



## Choreography and orchestration

Choreography and orchestration are two different models for how distributed services can communicate with one another. In orchestration, communication is more tightly controlled. A central service coordinates the interaction and order in which services are invoked.

Choreography achieves communication without tight control. Events flow between services without any centralized coordination. Many applications will use both choreography and orchestration for different use cases.




Communication between **bounded contexts** is often how choreography is used most effectively. With choreography, producers don't have expectations of how and when the event will be processed. Producers are only responsible for sending events to an event ingestion service and adhering to the schema. This reduces dependencies between the two bounded contexts.

Often within a bounded context, you need to control the sequence of service integration, maintain state, and handle errors and retries. These use cases are better suited for orchestration.

Event buses such as **Amazon EventBridge** can be used for choreography, and workflow orchestration services like **AWS Step Functions** or **Amazon Managed Workflows for Apache** Airflow (Amazon MWAA) can help build for orchestration. Examples of how you may use choreography and orchestration together could include sending an event to trigger an AWS Step Functions workflow, followed by **emitting events** at different steps.

## Connecting event sources

Many applications have external event sources. These can include SaaS applications, such as business applications responsible for running payroll, storing records, or ticketing. You can also ingest events from an existing application or database running on premises. Event-driven architectures can use events from all these sources.

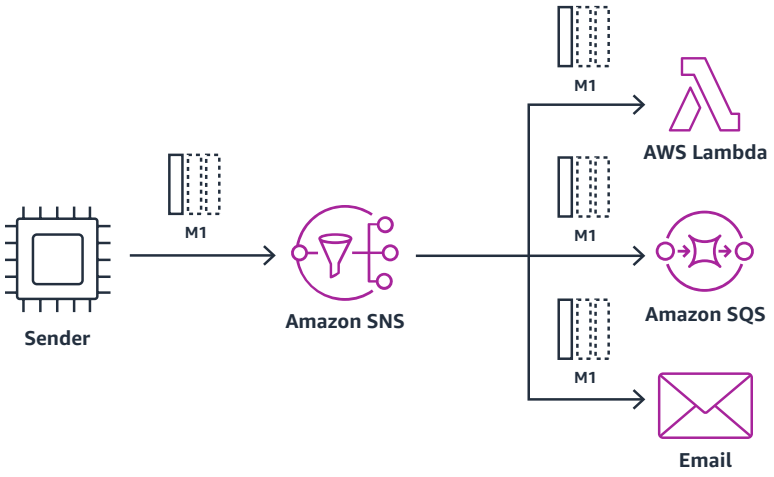
When applications emit business events, a common way to propagate the events is with a connector or message broker. These connectors bridge SaaS applications or on-premises sources, and they send events to a stream or a router, allowing consumers to process them. You can use **Amazon EventBridge partner event sources** to send events from integrated SaaS applications to your AWS applications.

This **post** demonstrates an example of building a mainframe connector with either **Amazon MQ** or **Amazon MSK**.

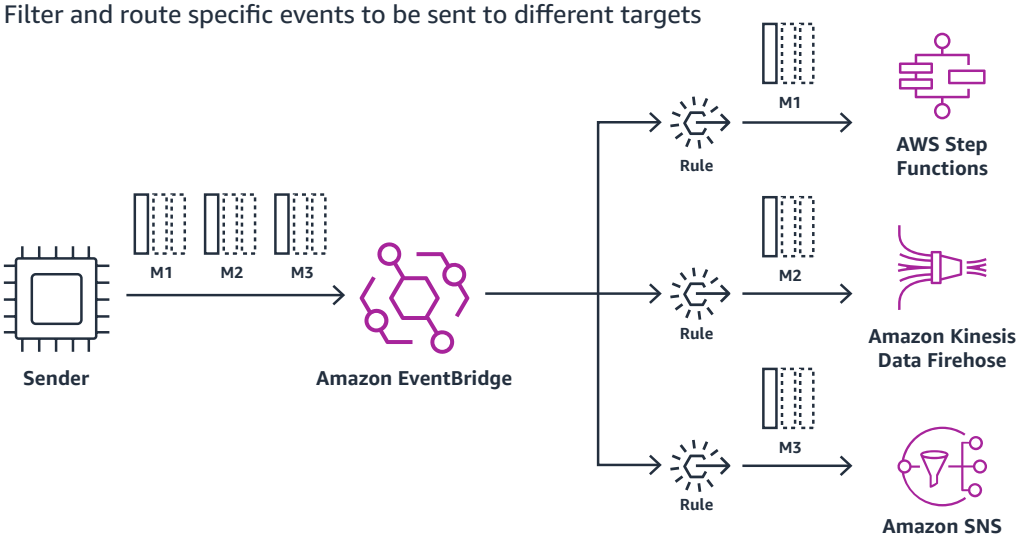
## Combining patterns

Though any one pattern may meet your requirements, event-driven architectures will often combine a series of patterns that:

- Fan out to send the same message to multiple subscribers of a single topic

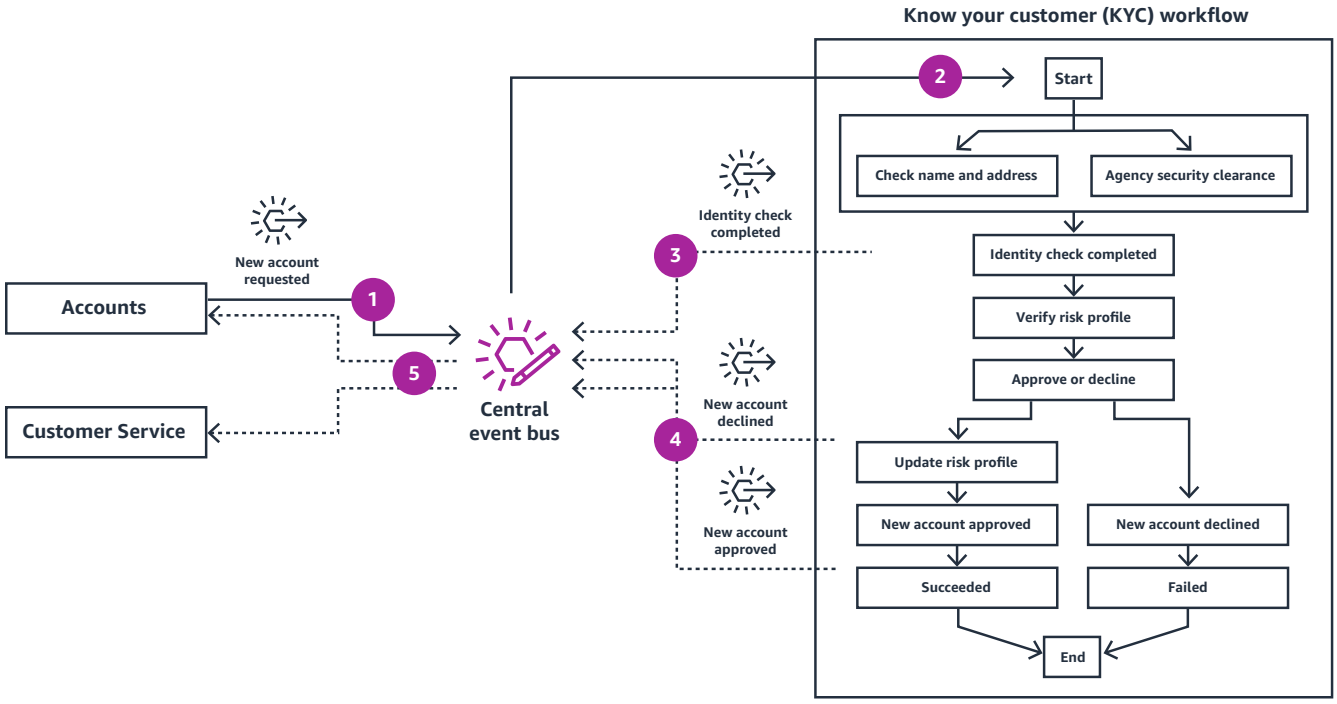


- Filter and route specific events to be sent to different targets




- Filter events and route them to a queue for persistence


- Orchestrate workflows and emit events at steps within the workflow



SECTION 3

# Considerations with event-driven architectures

While event-driven architectures are helpful when building and operating applications at scale, it can introduce new complications and challenges. This section will discuss considerations to keep in mind as you're designing your event-driven architecture.

## Eventual consistency

In an event-driven architecture, events are first-class citizens, and data management is decentralized. As a result, applications are often **eventually consistent**, meaning data may not be perfectly synchronized across the application but will ultimately become consistent.

Eventual consistency can complicate matters when processing transactions, handling duplicates, and determining the exact overall state of the system. Some components in many applications are better suited than others for handling eventually consistent data.

## Variable latency

Event-driven applications communicate across networks, unlike monolithic applications, which execute all processes using the same memory on a single device. This introduces variable latency. While it's possible to engineer event-driven applications to minimize latency, monolithic applications can always be optimized for lower latency at the expense of scalability and accessibility.

Workloads that require consistent low-latency performances aren't good candidates for event-driven architecture. Relevant examples include high-frequency trading applications in banks or sub-millisecond robotics automation in warehouses.

## Testing and debugging

It can be challenging to build end-to-end test cases for eventually consistent applications built with diverse technologies. Events produce side effects that can be difficult to catch in your integration testing.

A good practice is to refer to the **Test Pyramid**, which recommends many unit tests that can be run in isolation and a smaller number of integration or end-to-end test cases. **Contract testing** verifies that two systems (such as microservices) can communicate with one another, making it an ideal testing strategy for event-driven applications.

Debugging event-driven applications also introduces new challenges when compared to monolithic applications. With different systems and services passing events, it's often impossible to record and reproduce the exact state of multiple services when an error occurs. Each service has separate log files that together contain the full context around a given event. Its complexity may require your operations team to adopt new tools and skills.

## Organizational culture

Adopting event-driven architecture isn't solely a technological decision. Unlocking the full benefits of event-driven architecture requires a shift in both mindset and development culture. Developers will need a high degree of autonomy to make technology and architecture choices within the context of their own microservices.

You may also need to make changes such as organizing teams around business domains, building a **decentralized governance model** with a DevOps culture, and practicing evolutionary design principles. While these changes may mean upskilling your teams, event-driven architecture offers your application benefits in agility, scalability, and reliability.

SECTION 4

# Best practices

This section outlines best practices for making architectural choices and handling common challenges in event-driven architectures.

## Designing events

As previously defined, an event is a signal that a state has changed. An event describes something that happened in the past (e.g., “OrderCreated”) and is immutable or cannot be changed.

An event schema represents the structure of an event and tells you about its data. For example, an “OrderCreated” event might include fields like product ID, quantity, and shipping address and also that the product ID and quantity are integers and the shipping address is a text string.

A schema registry is a shared location where teams can store their event schemas. Using a schema registry allows you to build services that consume another service’s events without the need to coordinate with its developers.

When designing event schemas, you can choose to have sparse events or events with full state descriptions. A sparse event has very little data about the event, perhaps just its event ID. A full state description includes many details.

**Sparse event example:**

```
{
    "source": "com.orders",
    "detail-type": "OrderCreated",
    "detail": {
      "metadata": {
        "idempotency-key": "c1b95b88"
      },
      "data": {
        "order-id": "1073459984"
      }
    }
}
```

**Full state description example:**

```
{
    "source": "com.orders",
    "detail-type": "OrderCreated",
    "detail": {
      "metadata": {
        "idempotency-key": "c1b95b88"
      },
      "data": {
        "order-id": "1073459984",
        "status": "Open",
        "total": "237.51",
        "product-id": "09845221",
        "quantity": "1"
      }
    }
}
```

Both types of events have their trade-offs. When emitting a sparse event with many consumers that requires its details, all consumers will request more data at once. This can overwhelm the service or database that holds the event’s information.

Sending full state descriptions with your events will require you to consider backward compatibility. Event schemas are much like a contract between producers and consumers. Altering the information in the event can impact consumers and should be avoided.

You should also consider the cost of calculating the values in your event data. When you start building your application, the data may all be in the same location, such as in the same table in a database. As your application evolves, this may change. You may store different types of data in different locations over time, increasing the cost to calculate values.

While most applications' events will fall somewhere between sparse and full state descriptions, sparsity is a good place to start. Identify fields and values that are shared and used by the most consumers, then add those to your event schema first. As new requirements happen, you can add more data to events, but be sure to consider the backward compatibility of additional data.

## Idempotency

**Idempotency** is the property of an operation that can be applied multiple times without changing the result beyond the initial execution. You can safely run an "idempotent operation" multiple times without side effects, such as duplicates or inconsistency of data.

Idempotency is an important concept for event-driven architectures because they'll likely use retry mechanisms. For instance, asynchronously invoking an AWS Lamda function with an event wherein the function initially fails, AWS Lamda has built-in **retry logic** and will reinvoke the function. This increases your application's resiliency, but also means that a message can be processed multiple times by the function. This is an important consideration when managing orders, payments, or any kind of transaction that must be handled only once.

You can choose to build all your services as idempotent. This is helpful when handling duplicate events so any operation can be run multiple times without side effects. However, this approach can increase your application's complexity. Another option would be to include a unique identifier in each event as an **idempotency key**.

Once the event is processed, update the persistent data store with the results. If any new events arrive with the same idempotency key, return the result from the initial request.

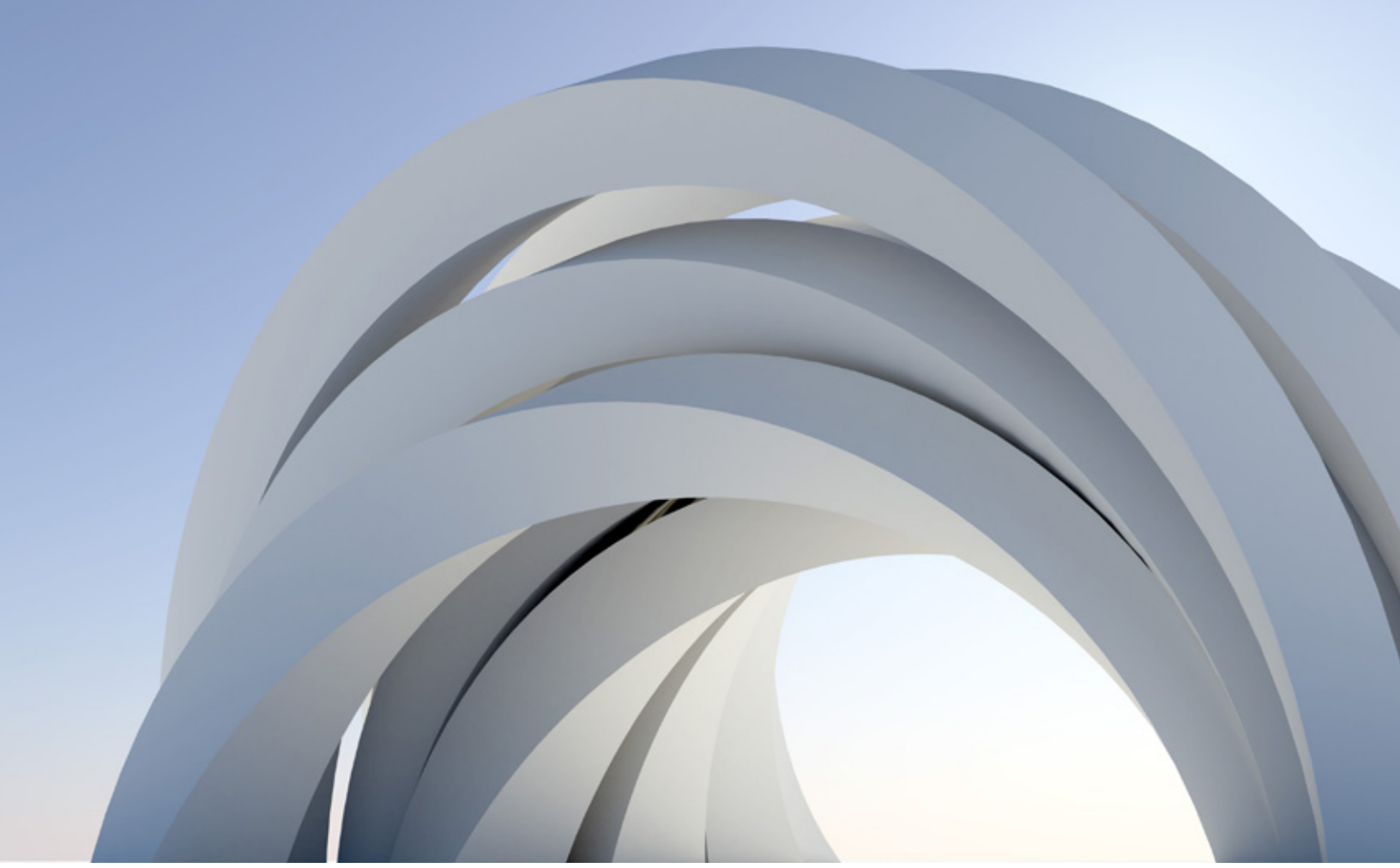
## Ordering

An important consideration with event-driven architecture is whether your application requires events delivered in a specific order (ordered events) or if the order doesn't matter (unordered events). Order is usually guaranteed within a specific scope.

You can choose to build with a service that guarantees order, such as Amazon Kinesis Data Streams or Amazon SQS FIFO (First-In-First-Out). Amazon Kinesis Data Streams preserve order within the messages in a shard. In Amazon SQS FIFO, order is preserved within a message group ID. However, it's important to understand that guaranteed order comes with drawbacks when compared to unordered events, such as increased cost and potential operations bottlenecks.

Other services, like Amazon EventBridge and Amazon SQS standard queues, offer best-effort ordering. This means that your application must assume that it will receive messages out of order. For many applications, this won't be an issue. However, if your application requires order, it's simple to mitigate.

Building to handle out-of-order events can increase your application's resiliency. This ensures your application doesn't fail in error should the event be sent out of order. For example, if your application has customer orders, you might assume you can't receive an "OrderUpdated" event before an "OrderCreated" event. You can handle this by creating a partial transaction record in a database when an "OrderUpdated" event is received while waiting for the "OrderCreated" event to be received. You can then generate an operational report with details of incomplete transactions to review and uncover issues. Rather than assuming events will be in order and otherwise fail, you can build to handle events out of order and increase your application's fault tolerance and scalability.



SECTION 5

# Conclusion

Event-driven architectures are an architecture style that can be used to increase agility and build scalable, reliable applications across your enterprise. While the approach can introduce new challenges and complexity, event-driven architectures are an effective method of building complex applications by empowering multiple teams to work independently.

AWS offers a large and growing portfolio of proven solutions and managed services that can help your organization build event-driven architectures. Having read this guide, you've gained invaluable knowledge about implementing event-driven architecture concepts, as well as the best practices and relevant services that can help your team build successful event-driven architectures with AWS.

SECTION 6

# Resources

## Whitepaper

Implementing Microservices with AWS
https://docs.aws.amazon.com/whitepapers/latest/microservices-on-aws/microservices-on-aws.html

## AWS Skill Builder course

https://explore.skillbuilder.aws/learn/course/70/architecting-serverless-solutions

## Tutorials and blogs with AWS Serverless

https://serverlessland.com/

## AWS Workshops

https://workshop.serverlesscoffee.com/
https://event-driven-architecture.workshop.aws/