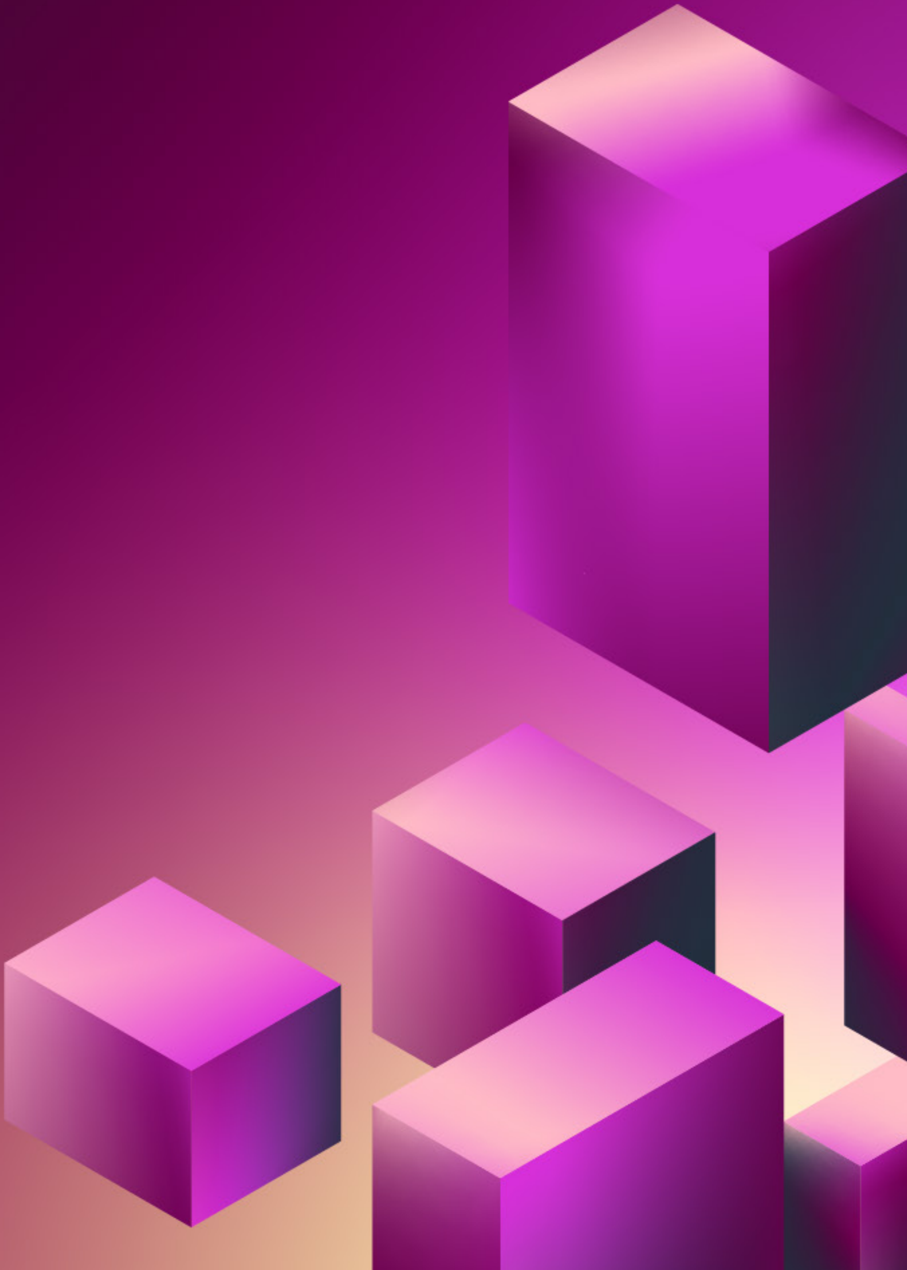
Building event-driven architectures on AWS



ABSTRACT

Events are everywhere—an event can be anything from a new account created, an item placed in a shopping cart, a financial document submitted, or a healthcare dataset uploaded. Events are the center of an application in an event-driven architecture, powering communication between integrated systems and development teams.

This guide introduces key concepts and patterns for event-driven architectures and identifies the Amazon Web Services (AWS) solutions and services commonly used to implement them. Additionally, it provides best practices for building event-driven architectures, from designing event schemas to handling idempotency.

SECTION 1

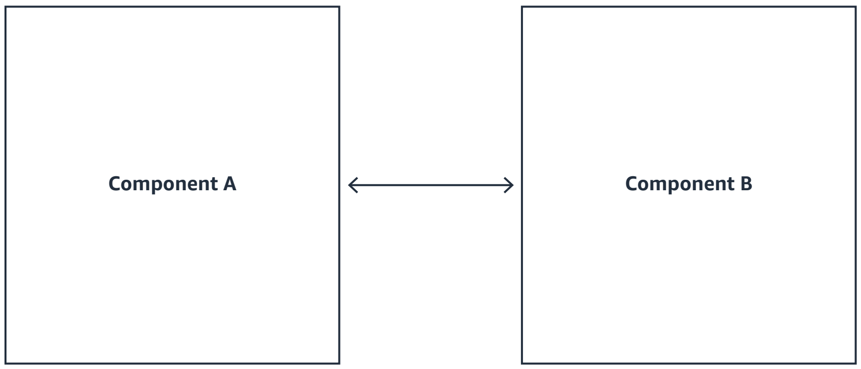
Event-driven architecture overview and key concepts

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 an operational issue in one component will propagate to others.

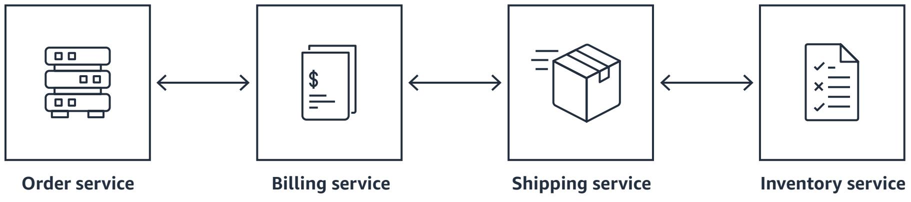


For complex systems with many teams involved, tight coupling can have drawbacks.

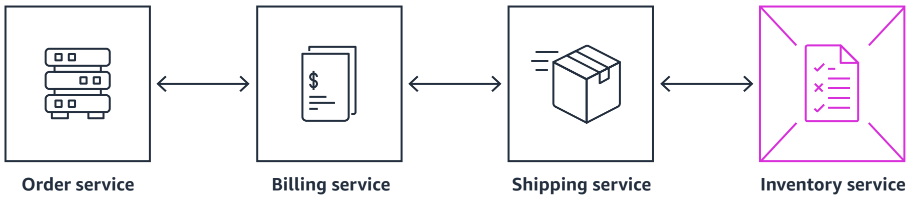
When components are tightly interdependent, it can be difficult and risky to make changes isolated to a single component without affecting others. This can slow down development processes and reduce feature velocity.

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 component will cause the other to fail. These failures can reduce the application's overall fault tolerance.

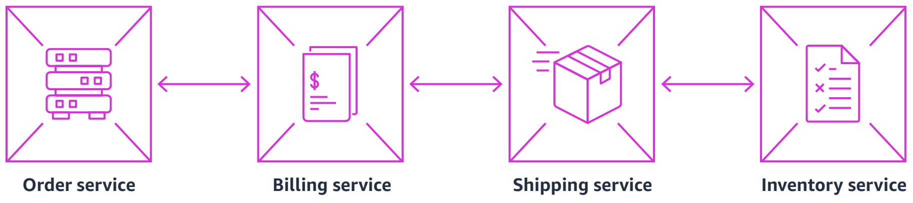
For example, an ecommerce application may have multiple services (orders, billing, shipping, inventory) and make a synchronous chain of calls to these services.



A failure in one of these services...



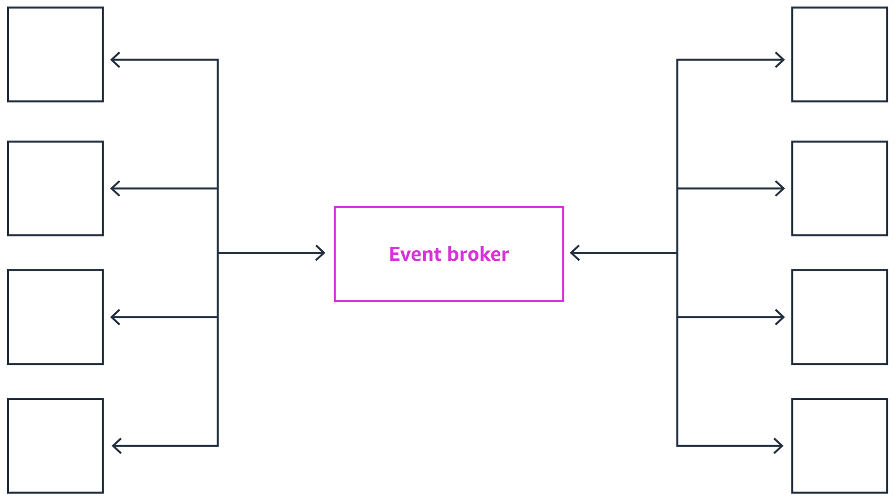
...will impact all the others.



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 does not require another to respond. Instead, the first component may send an event and continue without impact if the second component delays or fails.

When communicating using events, components only need to be aware of independent events. They do not require knowledge of the transmitting component or any other component's behavior (such as error handling or retry logic). As long as the event format remains the same, changes in any single component will not impact the others. This allows for less risk when making changes to an application. When asynchronous events abstract components from one another, complex applications become more resilient and accessible.



Core benefits of event-driven architectures

Building event-driven architecture requires a shift in mindset due to the unique characteristics and considerations of asynchronous systems. Yet, this style of architecture offers important benefits for complex applications. An event-driven architecture allows you to:

- **Build and deploy new features independently**
  Development teams working on individual services have fewer dependencies. They can publish and consume events from a central event broker, which allows developers to build and release features more independently without having to rely on another team to make code changes to facilitate integration. Changing one service will have less risk of impacting others.
- **Build new features using events without changing existing applications**
  Because components emit events, event-driven architectures are easily extensible. New services can subscribe to events already being published without impacting any existing applications or development teams. This enables businesses to build new features and products at a faster pace with less risk of disruption.
- **Enhance fault tolerance and scalability**
  With asynchronous events, upstream systems can buffer the volume of events they are sending to downstream systems. This allows applications to scale to handle peaks without overwhelming any part of the application. In an event-driven architecture, producers are unaware of any activity of downstream consumers, which makes them unaware of any failures.

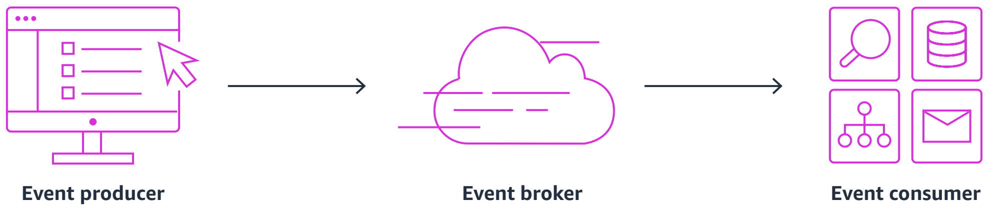
Key concepts of event-driven architectures

An event is a signal that a state has changed. For example, an item placed in a shopping cart or a credit card application submission. Events occur in the past (such as “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 or awareness of downstream components that may consume the event.

Event-driven architectures possess these key components:

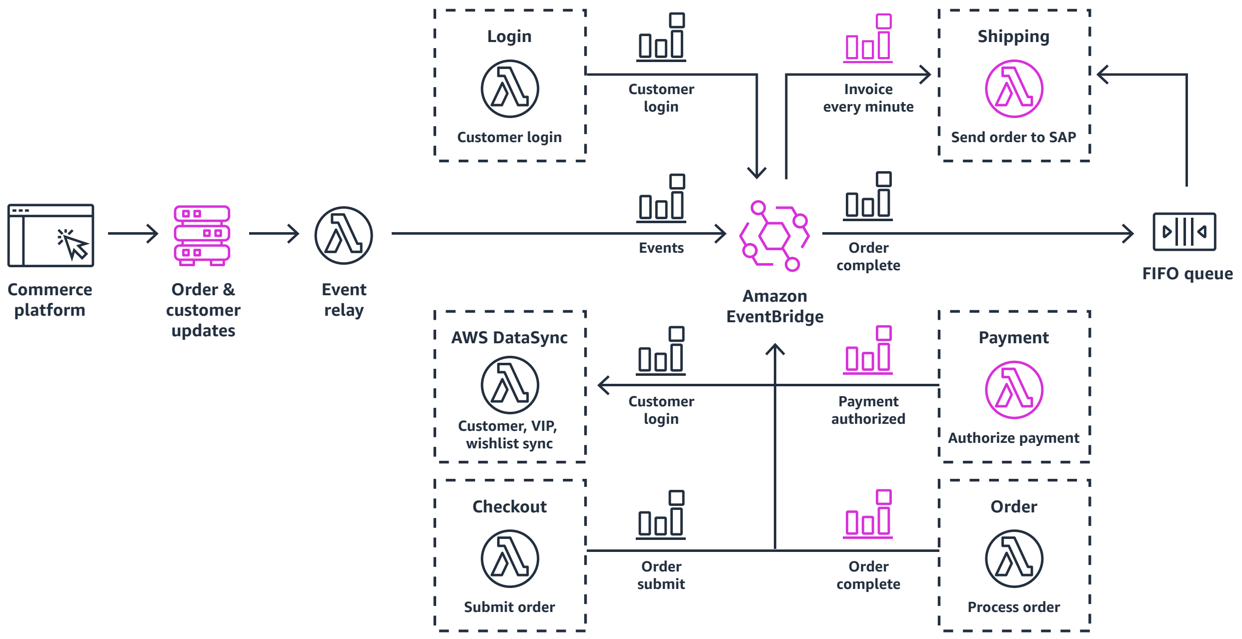
- **Event producers** publish events. Some examples of producers include frontend websites, microservices, Internet of Things (IoT) devices, AWS services, and software-as-a-service (SaaS) applications.
- **Event consumers** are downstream components that activate in events. Multiple consumers may be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



Example use cases

Microservices communication

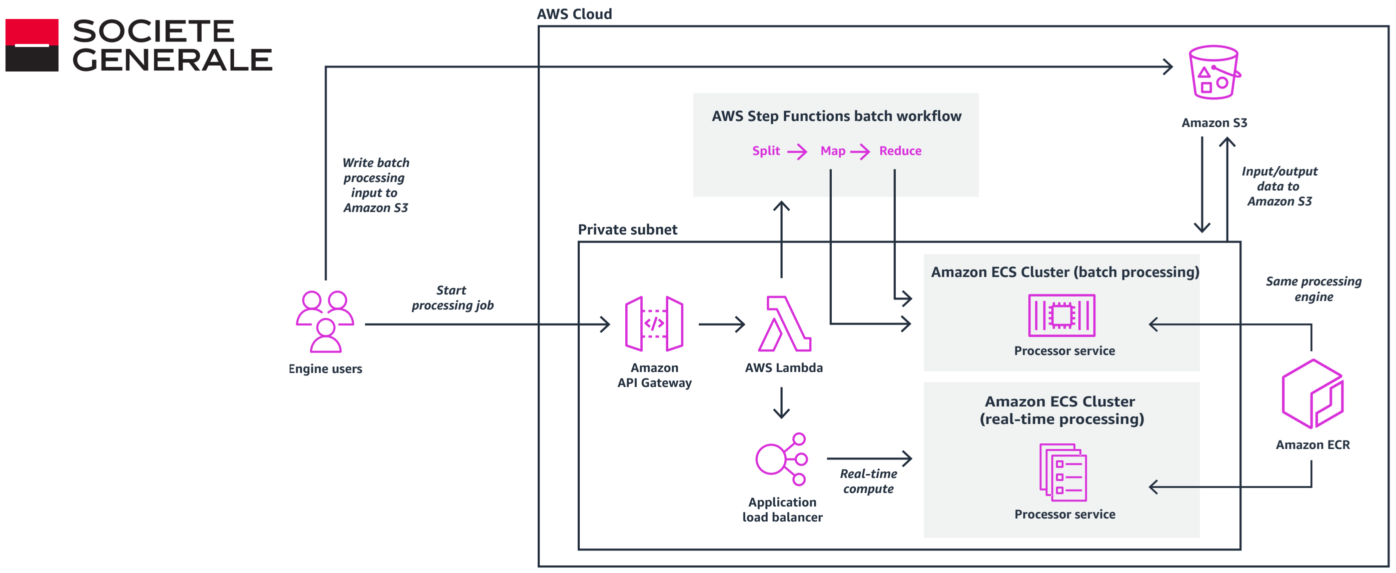
This use case is often seen in retail or media and entertainment websites that need to scale up to handle unpredictable traffic. A customer visits an ecommerce website and places an order. The order event is sent to an event router, and all the downstream microservices can pick up the order event for processing—for example, submitting the order, authorizing payment, and sending the order details to a shipping provider. Because each microservice can scale and fail independently, there are no single points of failure. This pattern has helped **LEGO** scale its ecommerce website to meet peak Black Friday traffic.



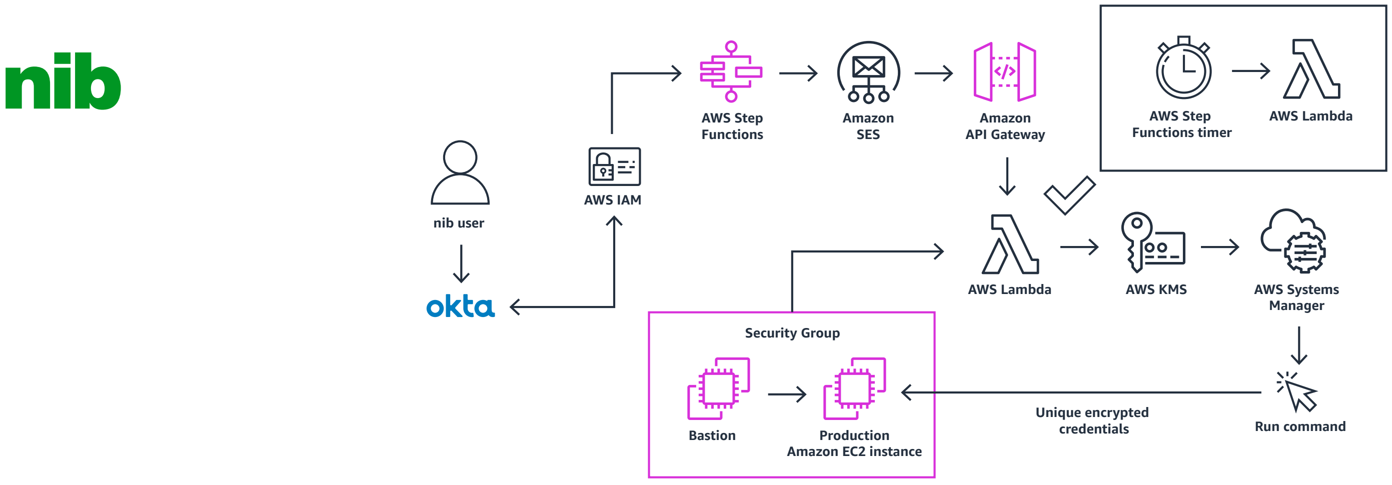
IT automation

Your infrastructure with AWS services already generates events, including **Amazon Elastic Compute Cloud** (Amazon EC2) instance 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.

Customers running compute-intensive workloads, such as financial analyses, genomic research, or media transcoding, can trigger compute resources to scale up for highly parallel processing and scale down those resources once the job is complete. **Société Générale** automatically scales resources up and down for credit risk analyses.


SOCIETE GENERALE
AWS Cloud
Amazon S3
AWS Step Functions batch workflow
Split
Map
Reduce
Write batch processing input to Amazon S3
Input/output data to Amazon S3
Private subnet
Amazon ECS Cluster (batch processing)
Same processing engine
Start processing job
Engine users
Amazon API Gateway
AWS Lambda
Processor service
Amazon ECS Cluster (real-time processing)
Amazon ECR
Real-time compute
Application load balancer
Processor service

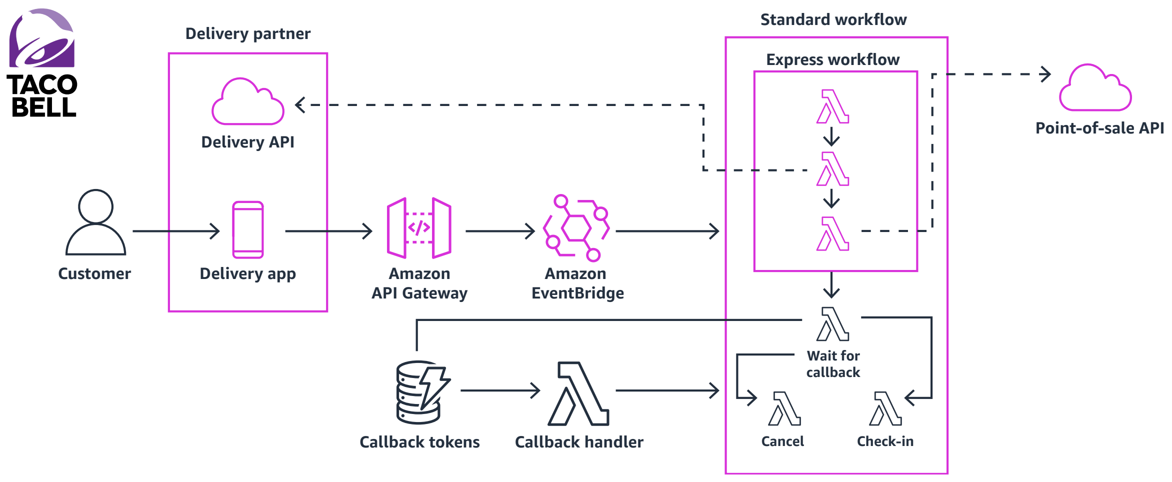
Customers in highly regulated industries, such as healthcare and finance, can use event-driven architectures to spin up security posture in response to an incident or take remediation action when a security policy sends an alert. **nib Group (nib)** spins up a time-bound, auditable security posture for secure access to resources.



Application integration

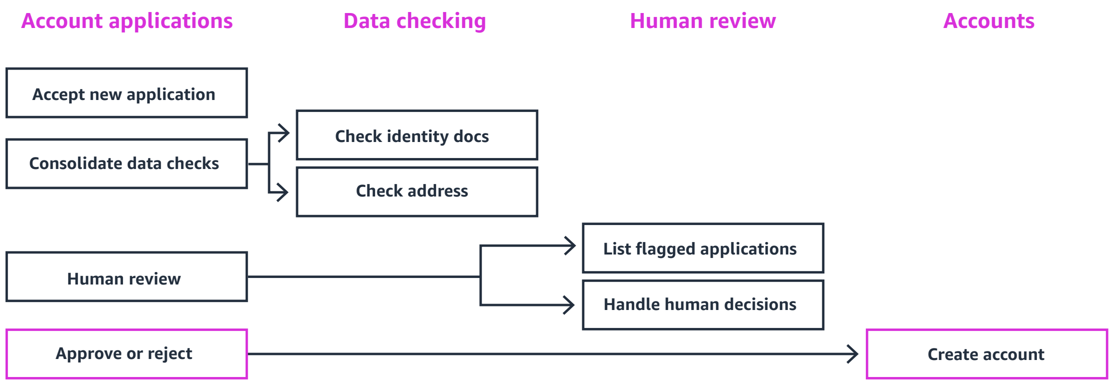
Events allow you to integrate other applications. You can send events from on-premises applications to the cloud and use them to start building new applications. Integrating SaaS applications enables you to create custom workflows using those events. You can use SaaS applications for services like customer relationship management, payment processing, customer support, or application monitoring and alerting.

According to **BetterCloud**, organizations used an average of 110 SaaS applications in 2021. Over half of respondents said the top challenge of their SaaS environment was a lack of visibility into user activity and data. To unlock siloed data, customers build event-driven architectures that ingest SaaS applications events or send events to their SaaS applications. **Taco Bell** built an order middleware solution to ingest incoming delivery partner app orders and send them directly to their in-store point-of-sale applications.



Automating business workflows

This use case is commonly seen in financial services transactions or in business process automation. Many business workflows require repeating the same steps, and those steps can be automated and executed with an event-driven model. For example, when a customer creates a new account application with a bank, the bank needs to run a few data checks (identity, documentation, address, and so on), and some accounts will require a human approval stage. All these steps can be orchestrated through a workflow service, with the workflow automatically triggered when a new account application is submitted.



SECTION 2

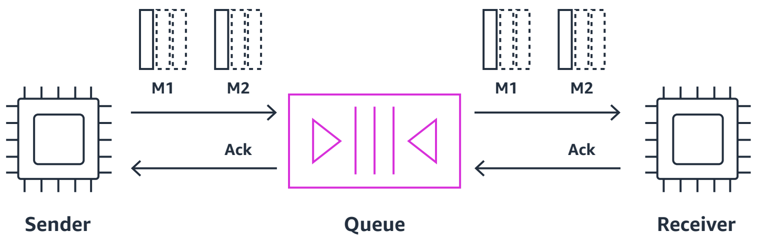
Common patterns in event-driven architectures

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 a message 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 sender and receiver and provide a buffer for messages in case the consumer is unavailable or needs to control the number of messages it can process at a given time. Messages will 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**."



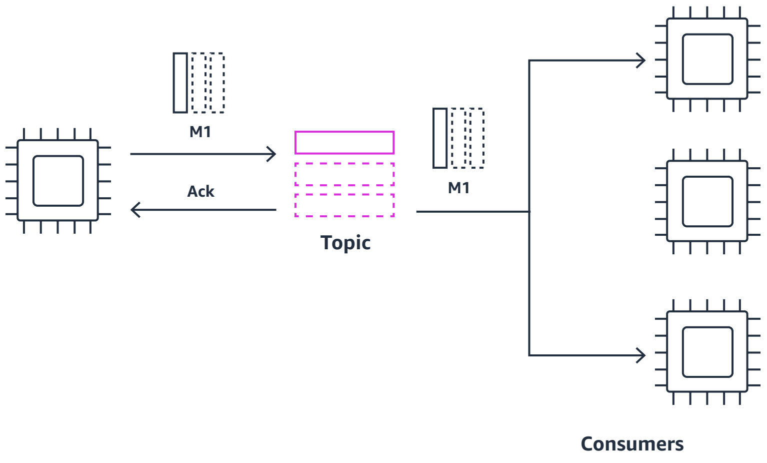
Services like **Amazon Simple Queue Service** (Amazon SQS) and **Amazon MQ** are commonly used as message queues in event-driven architectures. You can also use **asynchronous AWS Lambda invocations**.

In synchronous invocations, the caller waits for the 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 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 Lambda function. The function can send the result to a **destination**, with configurations varying on the success or failure. The internal queue between the caller and the function ensures that messages are durably stored.

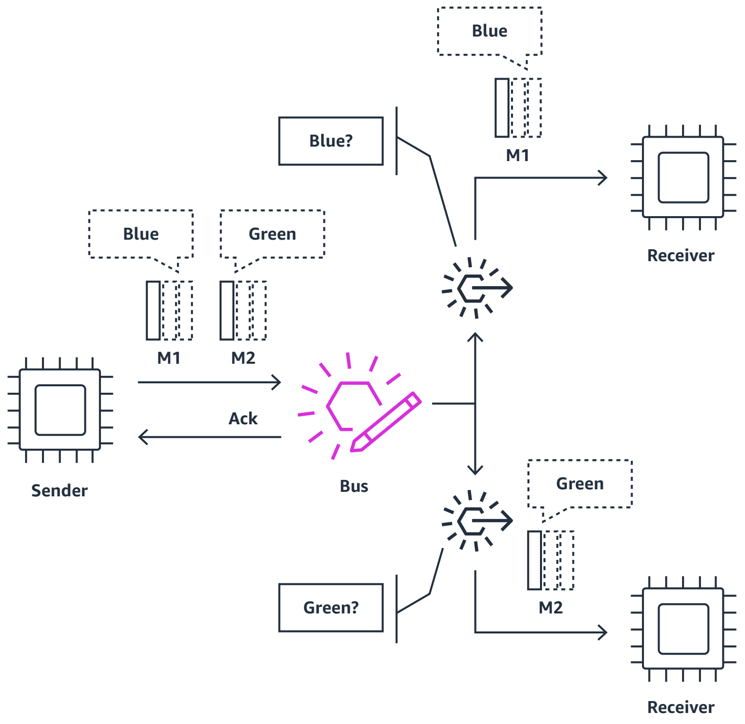
Publish-subscribe messaging

Pub/Sub messaging is a way that producers can send the same message to one or 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. EventBridge supports persisting events through its **archive** functionality. **Amazon MQ** also supports topics and routing.

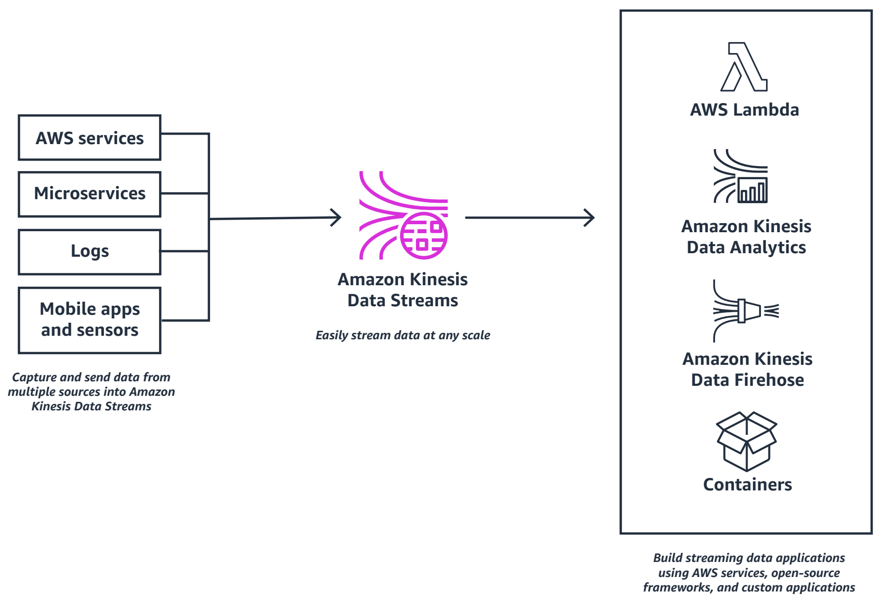
Event streaming

The use of streams, or continuous flows of events or data, is another method of abstracting producers and consumers. In contrast to event routers and 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. An example of event streaming is a rideshare application, which streams a customer's changing locations as events. Each "LocationUpdated" event is a meaningful data point used to update the customer's location on a map. It could also analyze location events over time to provide insights, such as driving speed.

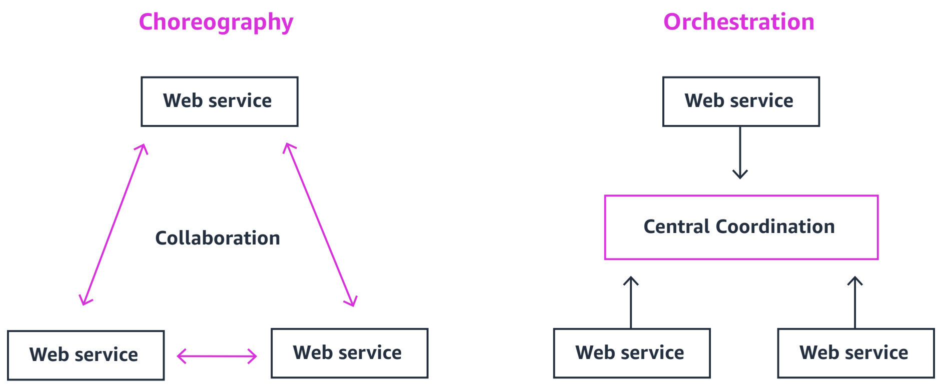
Data streams differ from event streams in that they always interpret data over time. In this model, individual data points, or records, are not independently useful. Data streaming applications are often used to either persist the data after an optional enrichment or to process the data over time to 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 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.


AWS services
Microservices
Logs
Mobile apps and sensors
Capture and send data from multiple sources into Amazon Kinesis Data Streams
Amazon Kinesis Data Streams
Easily stream data at any scale
AWS Lambda
Amazon Kinesis Data Analytics
Amazon Kinesis Data Firehose
Containers
Build streaming data applications using AWS services, open-source frameworks, and custom applications

Choreography and orchestration

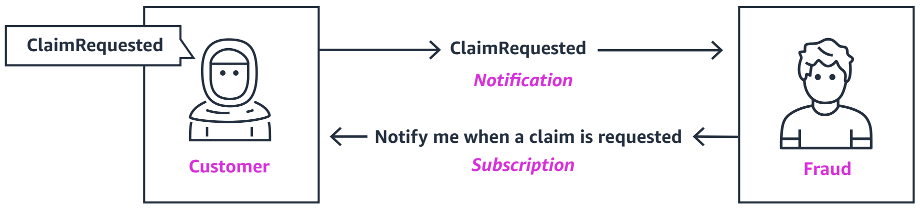
Choreography and orchestration are two different models for how distributed services can communicate with one another. Choreography achieves communication without tight control. In orchestration, communication is more tightly controlled. A central service coordinates the interaction and order in which services are invoked. Events flow between services without any centralized coordination. Many applications will use both choreography and orchestration for different use cases.



Choreography

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

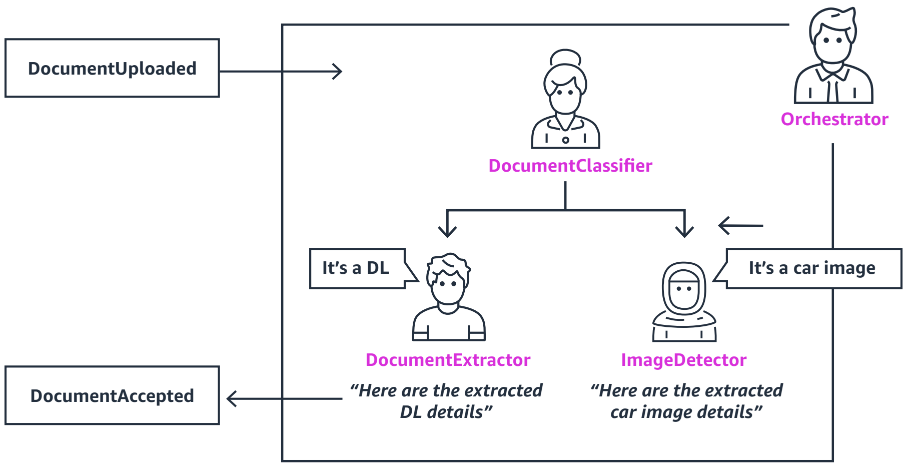
Below are two different domains or bounded contexts of an insurance claims processing application: Customer domain and Fraud domain. Customer domain emits a “ClaimRequested” event whenever an insurance claim is submitted. Fraud domain must subscribe to the “ClaimRequested” event emitted by the Customer domain in order to apply its domain logic when an insurance claim is submitted. When a “ClaimRequested” event is emitted, the Fraud domain gets notified. In the entire process of the choreography of events, neither the Customer (producer) nor the Fraud (consumer) domain is required to know about the internal business logic of the other. This choreographic approach facilitates loose coupling.



Orchestration

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 well suited for orchestration.

The figure below shows a Document Processing bounded context or domain in the insurance claims processing application. A “DocumentUploaded” event is received by the domain. The document processing domain includes an orchestrator that looks for the type of uploaded document. The orchestrator decides on the workflow path based on whether the uploaded image is a driver’s license or a car. The document classifier determines that the image is a driver’s license. It, therefore, directs the document extractor to extract all the information from the license. The extracted data is updated in a database before the document processing domain emits a “DocumentAccepted” event with all the details related to the image.

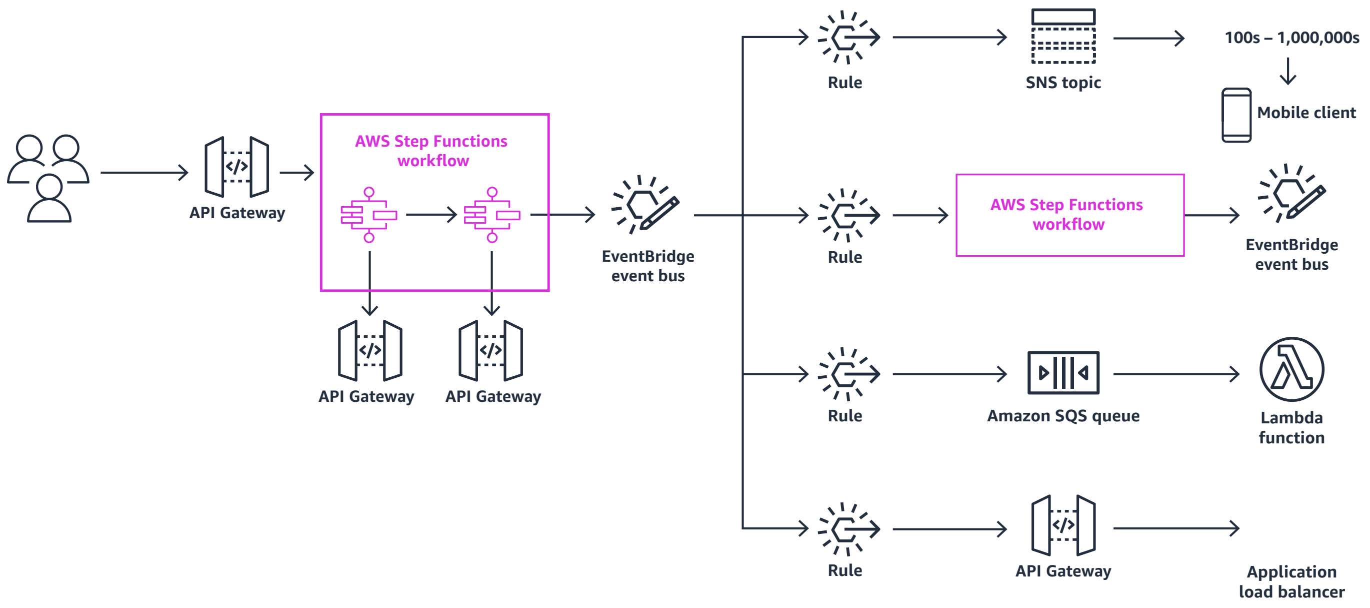


Event buses, such as **EventBridge**, can be used for choreography. Workflow orchestration services like **AWS Step Functions** or **Amazon Managed Workflows for Apache Airflow** (Amazon MWAA) can help build for orchestration. Examples of how you can use choreography and orchestration together include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

Choreography and orchestration together

The above examples of choreography and orchestration within the same application show that the two are not mutually exclusive. Many applications will use both choreography and orchestration for different use cases.

In the example below, the producer on the left emits events via an EventBridge event bus, and multiple consumers on the right are consuming those events. In addition to choreographing events between producer and consumer, the producer is also orchestrating two API calls to **Amazon API Gateway** by using Step Functions in its bounded context. You can also see that one of the consumers on the right is also orchestrating using Step Functions in its bounded context.



Together, choreography and orchestration give you the flexibility to address different needs in your domain-driven designs.

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 of these sources.

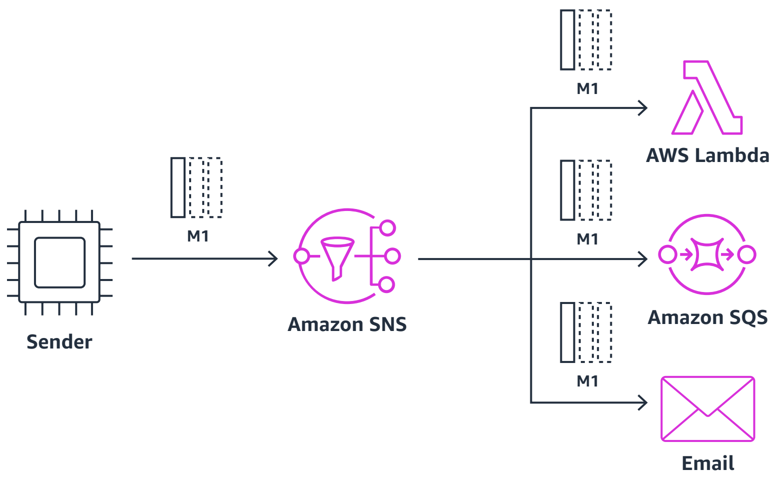
When applications emit business events, a common way to propagate the events is with a connector or message broker. These connectors bridge SaaS applications or on-premises sources and send events to a stream or a router, allowing consumers to process them. You can use **EventBridge partner event sources**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

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

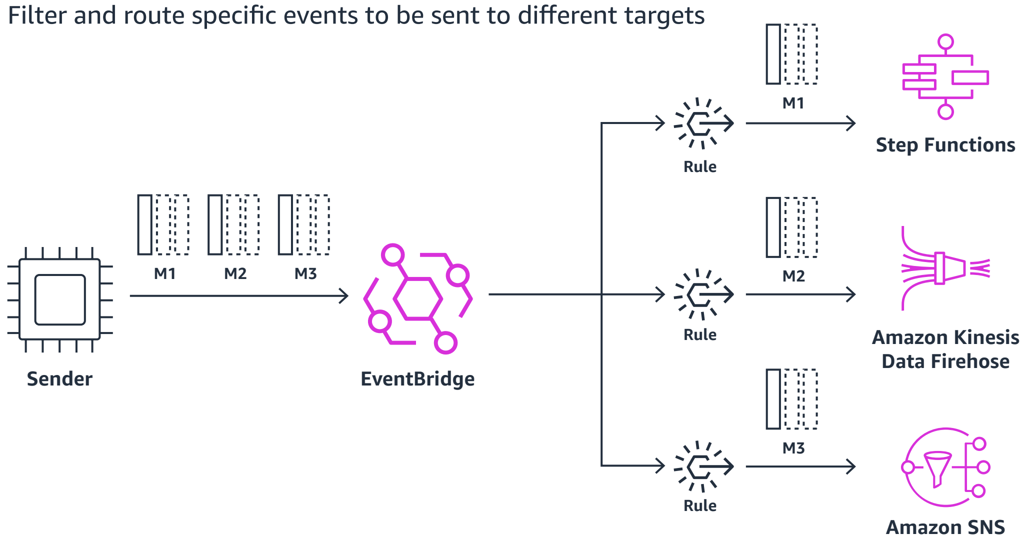
Combining patterns

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

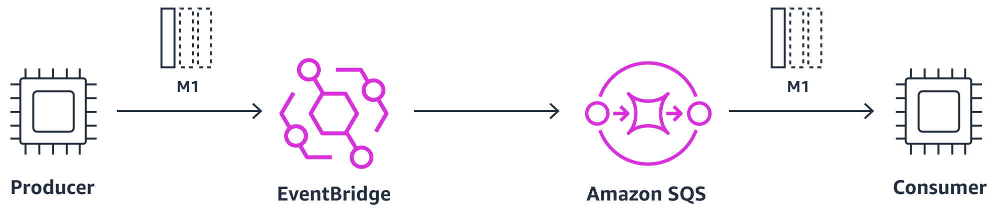
- Fan out to send the same message to multiple subscribers of a single topic



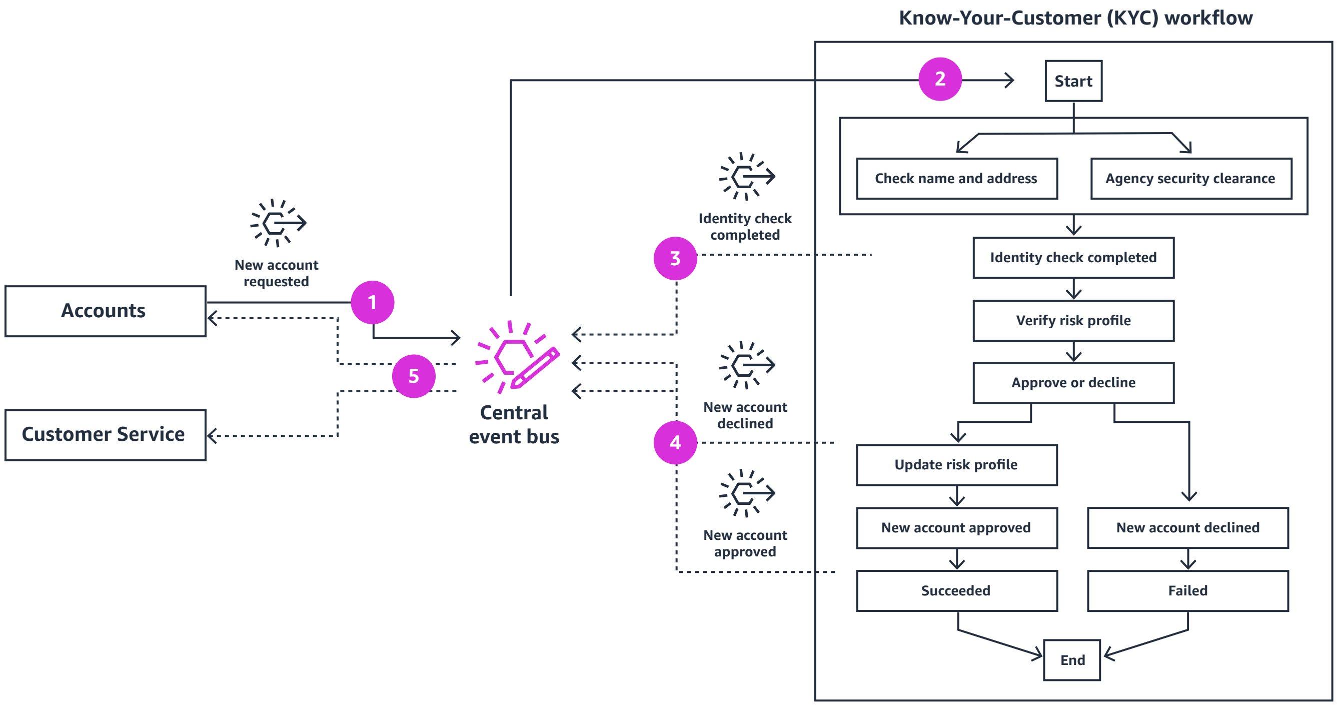
- Filter and route specific events to be sent to different targets



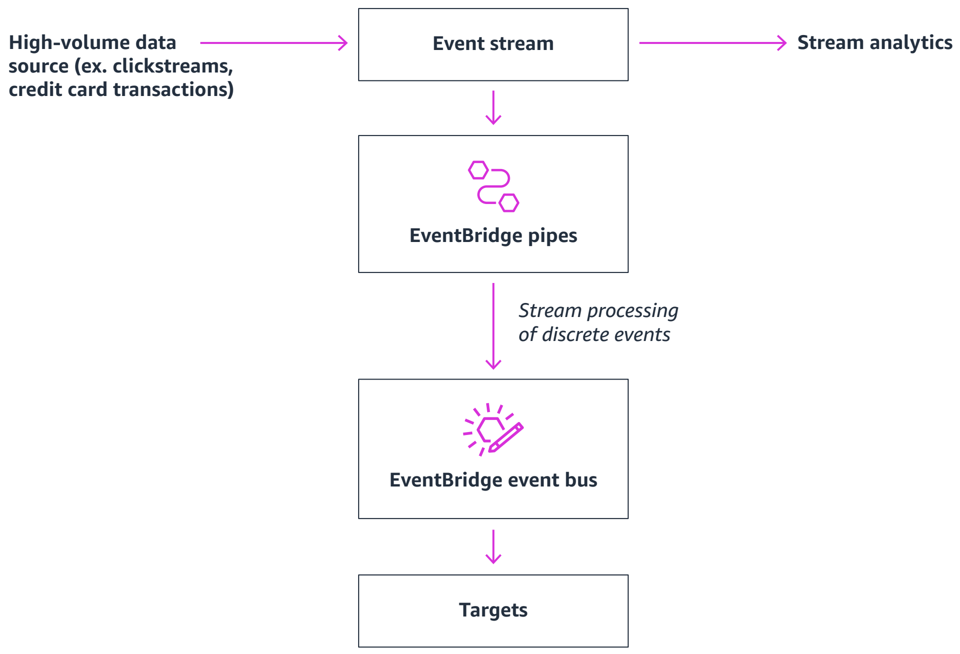
- Buffer event or message volume to downstream consumers with a queue



- Orchestrate workflows and emit events at steps within the workflow



- Combine event streaming platforms with EventBridge to subscribe to important business events without having to write integration code


High-volume data source (ex. clickstreams, credit card transactions)
Event stream
Stream analytics
EventBridge pipes
Stream processing of discrete events
EventBridge event bus
Targets

SECTION 3

Considerations with event-driven architectures

While event-driven architectures are helpful when building and operating applications at scale, they can introduce new complications and challenges. This section will present considerations to keep in mind as you design 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 to handle eventually consistent data.

Variable latency

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

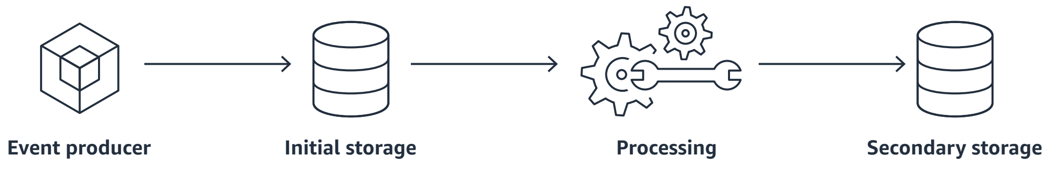
Workloads that require consistent low-latency performance are not good candidates for event-driven architectures. Relevant examples include high-frequency trading applications in banks or sub-millisecond robotics automation in warehouses.

Testing and debugging

Automated tests are crucial components of event-driven architectures. They help ensure that your systems are developed efficiently, accurately, and with high quality. This section provides some guidance for designing automated tests for event-driven architectures and asynchronous systems.

A generic asynchronous pattern

Asynchronous systems are typically composed of event publishers that send messages to event consumers that immediately store them for future processing. Later, downstream systems may perform operations on the stored data. Processed data may then be sent as output to additional services or placed into another storage location. Below is a diagram illustrating a generic asynchronous pattern.



Establish logical boundaries

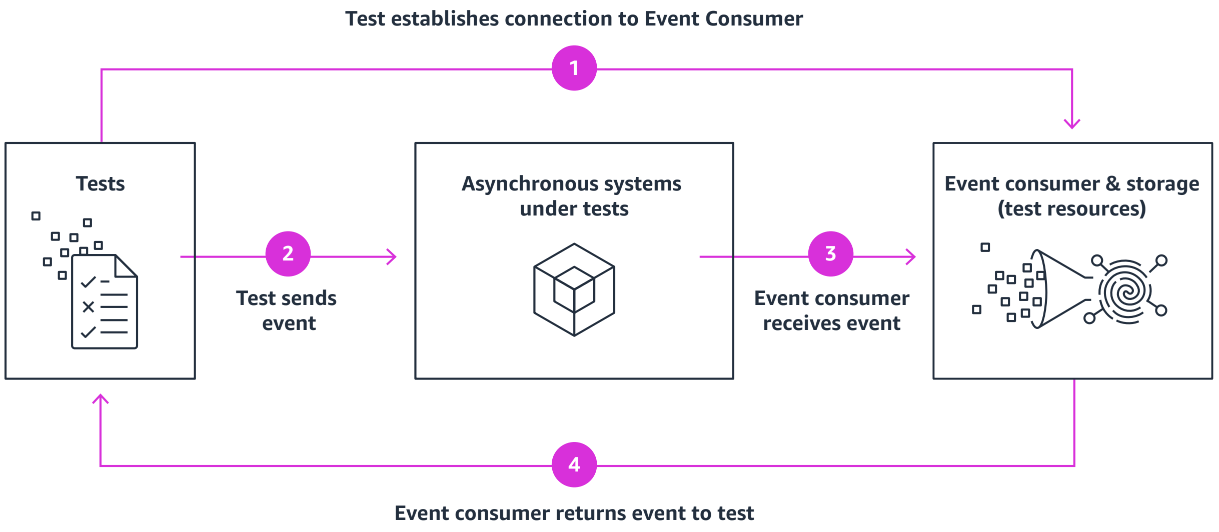
Asynchronous patterns like the one depicted in the diagram above rarely exist in isolation. Typically, a production system will be made up of many interconnected subsystems. To create a reasonable testing strategy, it is useful to break complex systems into a set of logical subsystems. A subsystem may be a group of services that work together to accomplish a single task and should have well-understood inputs and outputs. Breaking your complex architecture into smaller subsystems makes it easier to create isolated and targeted tests.

Create test harnesses

When testing asynchronous systems, it can be useful to create test harnesses. These harnesses contain resources that generate inputs for your subsystem and then receive the system's outputs. Your tests will use the harnesses to exercise the system and determine whether it is performing as expected. These harnesses are resources used for testing purposes only. They are not used by production features. Test harnesses are typically deployed to preproduction environments only. However, in some cases, you may want to validate requirements in production. If you can design your system to be tolerant of processing test data, you can deploy tests with the harnesses to production.

Configure test event producers and test event consumers

Test harnesses are usually composed of test event producers and test event consumers. The producers provide input to the system under test (SUT), and the consumers are configured to receive output. The automated tests send events to the producer and then communicate with the consumer to examine the output. If the output meets expectations, the test passes.



Test establishes connection to Event Consumer
1
Tests
2
Test sends event
Asynchronous systems under tests
3
Event consumer receives event
Event consumer & storage (test resources)
4
Event consumer returns event to test

Define service-level agreements

Although your architecture may be asynchronous, it is still useful to establish reasonable expectations about the maximum duration your system may take to process events before it is deemed to be in a failure state. These expectations can be explicitly defined as service-level agreements (SLAs). When you design tests, you may set timeouts that match your SLAs. If the system does not return results within the timeout period, you can consider it to be in violation of the SLA. Your tests should be designed to fail in that case.

Create schema and contract tests

Event-driven architectures decouple producers and consumers at the infrastructure layer, but these resources may still be coupled at the application layer by the event contract. Consumers may write business logic that expects events to conform to specific schemas. If schemas change over time, the consumer code may fail. Creating automated tests that validate event schemas can help increase the quality of your systems and prevent breaking changes.

Test at all stages

Testing is critical in event-driven architectures. With event-driven architectures, it can be hard to mimic the exact chain of reactions that sending an event will cause. Testing at all stages can help identify different use-case scenarios and find bugs. For organizations that have systems in highly regulated industries, like healthcare or financial systems, it is important to invest in comprehensive testing preproduction environments.

Create tooling for testing

Tooling investments can minimize the impact of failures. Use canary deployments to allow you to introduce code changes into your environment more slowly in defined increments rather than all at once. Feature flags allow you to introduce code and back it out quickly. Investments in observability tooling allow you to measure the rate of errors within your environments. Well-established rollback procedures in your continuous integration and continuous delivery (CI/CD) pipeline enable you to remove code that creates failures. Deploying code changes in frequent but small increments reduces deployment risks and increases agility.

View examples of testing

Code samples

This GitHub repository contains many examples of automated tests. **The project** creates a simple asynchronous system written in Python with tests and event listeners. To build the project and run the tests, follow the instructions in the **README** file.

Example integration test

Serverless test samples (GitHub)

After you run the tests, examine the code in the **/tests/** directory to see how the tests are written.

Example schema and contract tests

Navigate to the root of the ***serverless-test-samples*** project. Run the following command to find an **example project for schema and contract testing** written in TypeScript. Follow the instructions in the **README** file to run the tests.

Serverless test samples (GitHub)

After you run the tests, examine the code in the **/tests/** directory to see how the tests are written.

Organizational culture

Adopting an event-driven architecture is not just a technology decision. Unlocking its full benefits requires a shift in mindset and development culture. You may also need to make changes like 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 architectures deliver benefits in agility, scalability, and reliability to your application.

Developers will need a high degree of autonomy to make technology and architecture choices within the context of their own microservices. Leading with guardrails and observability can help you successfully implement the organizational change required for event-driven architectures. With guardrails and observability in place, you can promote a culture of autonomy while reinforcing standards.

Guardrails
Guardrails are predefined policies, controls, and standards that help enforce best practices across your organization. Using guardrails, organizations can help prevent accidental misconfigurations, enforce security policies, and reduce the risk of production incidents. By providing clear guidelines for secure development and operations, guardrails can help establish a culture of high quality.

Observability
Observability refers to the ability to monitor, analyze, and measure the behavior of systems and applications in production. As a result of implementing observability practices, organizations can gain a better understanding of their systems with regard to performance, reliability, security, and compliance. With minimal oversight, you can identify issues before they affect customers and continuously improve systems without affecting them. By promoting transparency, accountability, and continuous improvement, observability can serve to create a culture of reliability.

By providing clear guidelines for secure development and operations and by promoting transparency and continuous improvement, you can establish a culture of high autonomy, security, and reliability. Teams will feel empowered as owners of their systems and applications when these practices are embedded into the software development and IT operations lifecycles.

SECTION 4

Best practices

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

Designing events

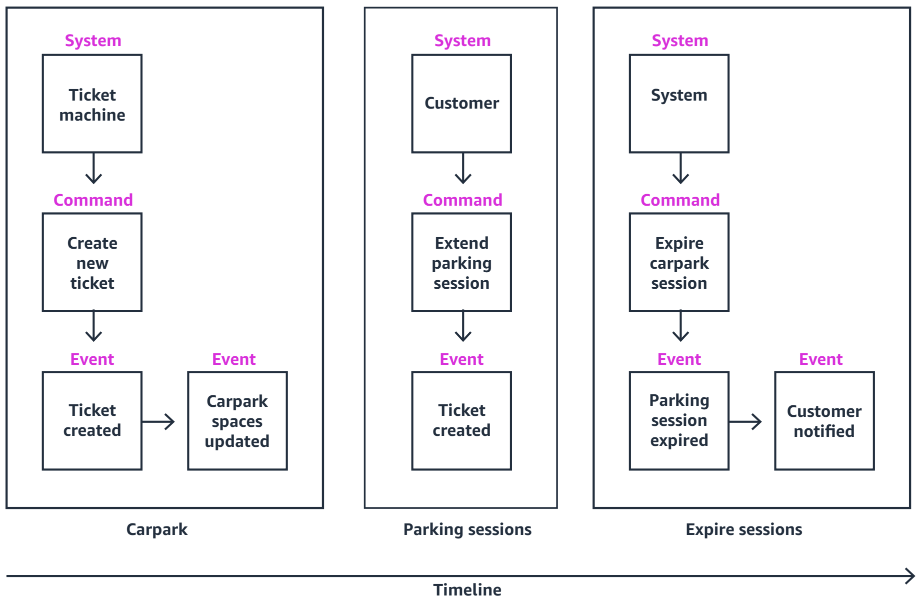
As we mentioned earlier, an event is a signal that a state has changed. An event describes something that happened in the past (for example, “OrderCreated”) and cannot be changed.

Constructing event-driven applications involves identifying events and devoting time to event design. This is crucial in developing and implementing an event-driven architecture that can scale across your organization over time with many producers and consumers.

Before designing events, it is important to identify events within your system—ones that are not only of importance to your architecture but also to your business. Raise events so existing or future downstream consumers can react and process business logic.

Identifying events with event storming

Event storming is a collaborative technique that helps to visually map out a system's behavior and identify events in event-driven architectures. The process involves gathering stakeholders from various domains of the system and facilitating a workshop where the system's events and actions can be jointly visualized and discussed. The primary goal is to generate a shared understanding of the system and identify critical business events.



During the event-storming process, you can highlight actors, commands, aggregates, and events, which can help you and your team understand the behavior of your system and also identify events before implementing them. Event storming is applicable to both new and existing projects. Developing a shared understanding among stakeholders can be beneficial in constructing event-driven architectures.

As events are identified, it is important to implement and define naming conventions for them.

Event naming conventions

Because events are immutable facts that have occurred in the past, the naming convention of events must be precise and clear—it must remove any ambiguity or vagueness when consumers discover and use them, as well as communicate the event's meaning and context.

For instance, comparing "UserLoggedIn" and "LoggedIn" highlights this point. The "UserLoggedIn" event provides clear meaning and context, conveying that a user has logged in. In contrast, "LoggedIn" lacks clarity about the event's meaning and context. Clear event naming conventions can aid downstream consumers in understanding the intent of the event and subscribing to it without requiring them to delve deeper into determining its relevance.

Once event naming conventions are established, it is important to understand the different event patterns you may want to consider. Many event-driven architectures involve **notification events and event-carried state transfer events**.

Notification events

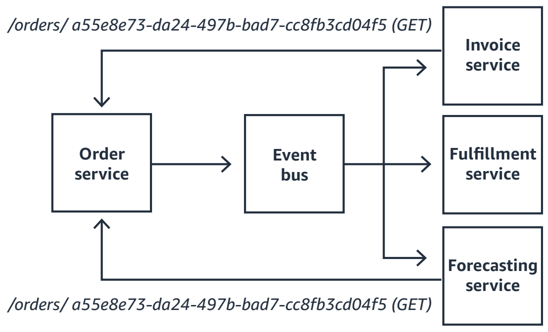
Notification events serve the purpose of informing downstream systems of an occurrence. For instance, if a user places an order, we can create an "OrderCreated" event that signals downstream consumers of the newly created order. Notification events contain a small payload, which conveys only the necessary information to downstream consumers. This simplicity ensures that producer contracts with consumers remain manageable, simple, and maintainable.

```
{
  “version”: “1”,
  “id”: “07fffe3b-4b25-48a7-b4a8-
432bcc3bfb2c”,
  “detail-type”: “OrderCreated”,
  “source”: “myapp.orders”,
  “account”: “123456789”,
  “time”: “2022-06-01T00:00:00Z”,
  “region”: “us-west-1”,
  “detail”: {
    “data”: {
        “orderId”: “3c947443-fd5f-4bfa-
        8a12-2aa348e793ae”,
        “userId”: “09586e5c-9983-4111-
        8395-2ad5cfd3733b”
    }
  }
}
```

Example of a notification event

Example of an EventBridge notification event with a simple payload

Downstream consumers may require additional information when dealing with notification events. For example, if an "OrderCreated" event is sent downstream with only "ordered," consumers in the downstream domain may need to retrieve further details about the order to process the event. This could lead to unwanted back pressure on the producer or another API to fetch the necessary information—a trade-off you should be aware of.



/orders/ a55e8e73-da24-497b-bad7-cc8fb3cd04f5 (GET)
Invoice service
Order service
Event bus
Fulfillment service
Forecasting service
/orders/ a55e8e73-da24-497b-bad7-cc8fb3cd04f5 (GET)

AWS services allow message or event filtering prior to their delivery to targets. For instance, **EventBridge** lets you establish rules to filter events before they arrive at downstream consumers. When dealing with notification events, the amount of filtering you can perform may be restricted due to the limited information contained within the event. If this is a concern, consider exploring event-carried state transfer (ECST) events.

ECST events

Notification events have a sparse and straightforward contract, while ECST events are the opposite. ECST events are published with more information for downstream consumers, reducing the likelihood that they will need to retrieve more information. If required, downstream consumers can also maintain a local cache copy of this information. With more data in the events, you can utilize filtering capabilities in AWS services to filter information before it reaches downstream consumers—for example, creating rules with filters on EventBridge for downstream consumers.

```
{
  "version": "1",
  ...
  "detail": {
    "metadata": {
      "domain": "ORDERS"
    },
    "data": {
      "order": {
        "id": "3c947443-fd5f-4bfa-8a12-
        2aa348e793ae",
        "amount": 50,
        "deliveryAddress": {
          "postCode": "PE1111"
        }
      },
      "user": {
        "id": "09586e5c-9983-4111-8395-
        2ad5cfd3733b",
        "firstName": "Dave",
        "lastname": "Boyne",
        "email": "dboyne@dboyne.com"
      }
    }
  }
```

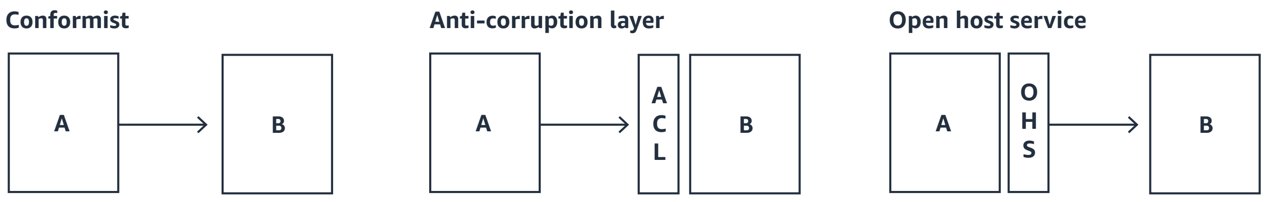
Example of an ECST event

When publishing events with larger payloads, several factors need to be considered. The first consideration is to prevent leaking implementation details from your domain/service into your events. You must carefully evaluate what information to include in your events and what to expose to downstream consumers. Be aware that boundaries exist between services, and through the event schema, you create a contract between producers and consumers. This contract necessitates maintenance and management, and the more information you include in your events, the more careful you must be in managing changes and versions of those events.

The event design pattern you choose will depend on your use case. Many applications incorporate both patterns. When consuming notification events or ECST events, you should consider consumer consumption patterns (bounded context mappings). Doing so can help you manage and mitigate risks when breaking changes occur between contracts with producers and consumers.

Bounded context mappings: Patterns to help when consuming events
Bounded context mappings are a technique used in domain-driven design (DDD) to define and document the relationships between different bounded contexts within a system. Bounded contexts are self-contained domains within a larger system, with clear boundaries and their own language, models, and business logic. When building event-driven architectures, it is common to use messages and events to communicate between these boundaries of systems.

When consuming a published event, consumers can choose from several options on how they want to consume that information. They can either conform to the event itself (conformist), write a wrapper to transform the event (anti-corruption layer [ACL]), or agree on a shared public language and push transformations back on the producer (open host service [OHS]).



Examples of bounded context mappings with event-driven architecture

Conformist pattern
The conformist pattern is a pattern used in event-driven architecture to consume events as they are published without any transformation or modification. This pattern ensures that downstream consumers are aligned with the producer's schema, providing a clear contract between the producer and consumer.

When using the conformist pattern, be aware of how much of the published domain information is leaking into your own domain/boundary. If you want to control the impact of contract changes between the producer and consumer, you may want to consider implementing an anti-corruption layer.

ACL
In an event-driven architecture, an ACL is a pattern used to ensure that data from one domain is not corrupted or misused in another domain. The ACL acts as a mediator between two bounded contexts that have different languages, models, or business logic, providing a translation layer between them. When consuming events from producers, implementing an ACL can assist in controlling the impact that contract changes can have and mapping between different domain models.

OHS
In an event-driven architecture, the OHS pattern is a design pattern used to enable multiple bounded contexts to communicate with each other by defining a shared public language. The OHS pattern involves establishing a shared public language between domains, which can be used as an intermediary for communication. The shared language is defined through a set of agreed-upon interfaces, contracts, or schemas, which can be used to push transformations back on the producer.

Implementing a bounded context mapping pattern can help you control how events are consumed by the producer and may give options to handle or isolate domains and even break changes with contracts. The mapping options you consider will depend on your use case.

Event-first thinking
The process of event identification and event design is ongoing. You may need to repeat the process regularly to ensure event relevance to changing business requirements. It is crucial to treat event design as a critical element when implementing event-driven architectures. Typically, events are designed and implemented as needed and often are an afterthought. But shifting your mindset to an **event-first thinking approach**—with clear intent and correct event patterns and consumption patterns—can help engineers build and implement event-driven architectures.

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 inconsistent data.

Idempotency is an important concept for event-driven architectures because they use retry mechanisms. For instance, asynchronously invoking a Lambda function with an event wherein the function initially fails, Lambda will use built-in **retry logic** to reinvoke 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 that 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.

```
{
   “source”: “com.orders”,
   “detail-type”: “OrderCreated”,
   “detail”: {
      “metadata”: {
         “idempotency-key”: “c9894c60-0558-
         4533-a9b0-8bb579303428”
      },
      “data”: {
         “orderId”: “e92570b8-3fb3-4a4b-
         b24b-d697919fe56c”
      }
   }
}
```

Example of an idempotency event

Ordering

An important consideration with event-driven architectures is whether your application requires events to be delivered in a specific order (ordered events) or without regard to order (unordered events). Order is usually guaranteed with specific scope.

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

Other services, like **EventBridge** and **Amazon SQS** standard queues, offer best-effort ordering. This means that your application must assume it will receive messages out of order. This will not be an issue for many applications. However, if your application requires order, AWS makes it simple to mitigate.

Unordered events
Building to handle out-of-order events can increase your application’s resiliency. This ensures your application does not fail if the event is 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 can be used to increase agility and build scalable, reliable applications across your enterprise. While the approach can introduce new challenges, event-driven architectures are an effective method of building complex applications by empowering multiple teams to work independently.

AWS offers a comprehensive set of serverless services for building event-driven architectures. With AWS, you can integrate events across over 200 AWS services, over 45 SaaS applications, as well as custom applications, all of which make it easier and faster to build scalable, event-driven applications.

We hope that, in reading this guide, you have gained a solid understanding of event-driven architecture concepts, best practices, and relevant services that can help your team build successful event-driven architectures with AWS.

SECTION 6

Resources

AWS Skill Builder: Architecting Serverless Solutions course ›

Event-driven architectures tutorials and blogs ›

AWS Workshop: Build a serverless and event-driven Serverlesspresso coffee shop ›

AWS Workshop: Building event-driven architectures on AWS ›