**Chapter 5. Event-Driven Architecture Patterns**

*Event-driven architecture* is a software architecture paradigm that promotes generation, detection, consumption, and reaction based on events. An event-driven architecture allows us to build distributed and scalable cloud native applications. In contrast to the service composition patterns, which are mostly synchronous in nature, event-driven architectures are asynchronous. They provide a clean and decoupled way of designing cloud native applications, allowing simpler scaling, and are one of the fundamental architectures for building large-scale distributed cloud native applications.

*Events* are used for sharing information. In most cases, the application generating the event notification does not expect any response, and it lets the consuming application decide what to do with that information. Even if the applications generating the event notification expect a response, they expect it only indirectly.

**NOTE**

Events can be categorized as any significant occurrence or change in a system state. Let’s take an example of depositing $50 into Bob’s account. Now, Bob’s account balance has increased by $50; this incident is considered an event. The occurrence of this event can be sent to other systems, such as to Bob’s cell phone, as a notification. The event notification is typically an asynchronous message produced and transmitted with the event occurrence information. Though events just occur and do not travel, the term *event* is also used interchangeably to denote the message that notifies the event. This is mainly because event-driven architectures are built on top of an asynchronous messaging infrastructure that uses messages to identify and communicate event occurrences.

Some events can be used to issue a command or enforce an action on other systems, such as sending an event to update Bob’s current address. In this case, the application sending the event *does* expect the consuming system to perform an action, and so requires the intermediate systems to guarantee that the event is delivered. Though not required, message brokers and event buses can be used to reliably deliver such events. We’ll discuss event delivery guarantees in detail in the following section.

Event-driven cloud native applications can be implemented by using microservices, as well as by using serverless computing platforms such as Amazon Lambda and Azure Functions. This is because these platforms are natively event triggered. The use of serverless is especially useful when the frequency of event occurrence is low and when we can significantly save on infrastructure cost.

This chapter focuses on patterns for building cloud native event-driven applications that process discrete events. Some use cases require processing a series of events, in order to understand behavioral and temporal characteristics. Such a series of events ordered by time is called a *stream*, and because processing a stream is quite different from processing discrete events, we have dedicated [Chapter 6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#stream_processing_patterns-id00204) to discussing stream-processing patterns.

In this chapter, we cover fundamentals of event-driven architecture, and patterns that fall into the categories of event delivery, event-based state management, and event orchestration. We will also look at technologies related to these patterns, and how to test, enforce security, achieve continuous delivery using DevOps, and operate the applications with monitoring and observability.

**Event-Driven Architecture**

Unlike in the synchronous communication patterns presented in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns), in event-driven architecture, we cannot always directly send messages to consumers and get an acknowledgment that they have consumed them successfully. In most cases, we need intermediate systems such as message brokers to consume, store, and deliver events to their consumers while ensuring that no events are lost across system failures. Message brokers provide different event delivery guarantees, as follows:

*At-most-once delivery*

The event is delivered to the consumer only once or not at all. If the consumer is not online during a delivery attempt or if network failures occur, the consumer will not get the event. Most important, the message broker will not try to send the same event again.

*At-least-once delivery*

The event is guaranteed to be delivered to the consumer. However, the consumer may consume the same event multiple times because if the message broker does not get an acknowledgment from the consumer for the event delivery, it will assume that the consumer did not receive the event and will resend it. In this case, the consumer should be intelligent enough to handle duplicate events.

Unfortunately, we cannot achieve an *exactly once* delivery guarantee, which ensures that the event is delivered to the consumer once and only once, because of the uncertain nature of the network and systems.

**Exactly Once Processing**

Though the events are delivered with an at-least-once delivery guarantee, for correct execution of any business process, we need to achieve *exactly once processing*. This ensures that the event is processed once and only once. For example, we should not deposit $50 to Bob’s account twice when there is an actual event occurrence of one. We can achieve this by injecting sequence numbers to the events so consumers can identify and drop duplicate events before processing them.

We can also achieve exactly once processing when events are idempotent: the outcome of receiving the same event multiple times is no different from receiving the event only once. For example, let’s say the event contains an update to a user’s telephone number and we have an application to update the database based on that event. In this case, processing the same event multiple times by the consumer application will not affect the final state of the user data in the database. In the end, the database will have the same updated telephone number.

For business-critical information, such as user-transaction information, we should use exactly once processing with at-least-once delivery. If the event contains information that the business can lose, such as simple notification information or a periodic update, using at-most-once delivery might be sufficient. Implementing a higher level of guarantee comes with performance considerations and increased complexity, so it is important to select the lowest required guarantee.

**Message Broker Categories**

Message brokers can be divided into two main categories:

*Standard (store-backed) message brokers*

These are the standard message brokers that store events in a data store to enable serving to intended consumers. Most important, they purge events from their store upon delivery to consumers. Apache ActiveMQ and RabbitMQ are examples of these brokers.

*Log-based message brokers*

These brokers store events in commit logs. The events persist even beyond their being consumed. Therefore, these brokers allow consumers to replay events from a previous point in time. Apache Kafka and NATS are examples of this type.

Regardless of the category, different message brokers, and at times even the same message broker, can support various delivery guarantees. We discuss message brokers in detail, categorizing them, and outlining their supported event delivery patterns in [“Technologies for Event-Driven Architecture”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#technologies_for_event_driven_architect).

**CloudEvents**

Like APIs and data, events should have defined schemas. This helps event producers and consumers interoperate seamlessly. When defining events for cloud native applications, we recommend using the CloudEvents specification to define the structure of the event payload.

*CloudEvents* is a CNCF project that provides a common standard for describing events that enable interoperability across cloud native applications. The CloudEvents specification provides a common structure for the event with required attributes such as event ID, source, spec version, and type, and optional attributes such as data content type, schema, subject, and time. It also provides software development kits in languages such as Go, JavaScript, Java, C#, Ruby, and Python so you can rapidly adopt and leverage CloudEvents-based schemas.

**Event Schema**

The most common event data formats are JSON and XML, but binary formats such as Avro and Protobuf are also gaining popularity in high-performance scenarios. Whatever format you choose, you should always version the schema so that changes to the format are efficiently communicated and managed. As we discussed in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns), you should also use a schema registry, where possible, to store all event schemas so that consumers can fetch schemas on demand to interpret events. Apache Kafka, for example, is a message broker that supports both Avro-based events and a schema registry.

Let’s now discuss various event-delivery patterns such as Producer-Consumer, Publisher-Subscriber, Store and Forward, and Event Sourcing.

**Event-Delivery Patterns**

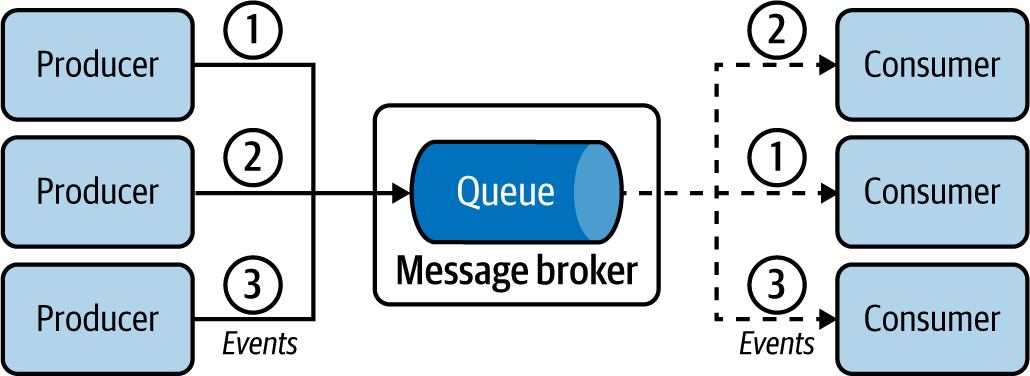
Let’s start by discussing some *event-delivery patterns* that are used across event-driven architectures. These patterns build on the knowledge gained about asynchronous communication patterns in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns). Here, we discuss various delivery guarantees as well as event schema management.

**Producer-Consumer Pattern**

The *Producer-Consumer pattern* enables producer applications and consumer applications to communicate asynchronously by using event queues. The queue manages which consumer processes which event, and which procedures need to be followed when the consumers fail during the processing of the event.

**How it works**

This pattern requires an intermediate message queue that is managed by a message broker ([Figure 5-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_delivery_from_producers_to_consum)). One or more producers can send events to the queue. The message broker typically persists the queued events in a durable store, thereby guaranteeing that the events will eventually be delivered to consumers. The message broker then delivers one event at a time, mostly following FIFO order, to the consumers on request. This helps consumers process events as they have capacity and to not become overloaded.



**Figure 5-1. Event delivery from producers to consumers**

When consumers are done processing the event, they can also send an acknowledgment to the message broker, so that the message broker can purge the event from its store.

**How it’s used in practice**

This pattern can be used for various scenarios, such as asynchronously delivering events from one application to another, making sure only one application processes a piece of data, ensuring event delivery, sharing workload among applications, handling sudden bursts of events, and decoupling applications. Let’s see how we can achieve these with some examples.

**Provide asynchronous event delivery**

In asynchronous event delivery, the most common use of this pattern, we pass events to another application without blocking. For example, let’s say we receive a loan application request. If the approval process takes a long time, we can simply add the request to an event queue so it can be processed by the loan processing application. At the same time, we can notify the customer that we have successfully received the request and will follow up about the outcome via email.

**Process each event by a single application**

We can use the Producer-Consumer pattern when we need events to be consumed and processed by only one of the available consumers. Using event queues ensures that events are not delivered to multiple consumers. For example, in the loan processing use case, we need only one application to process the loan request so there won’t be more than one credit check against the same customer.

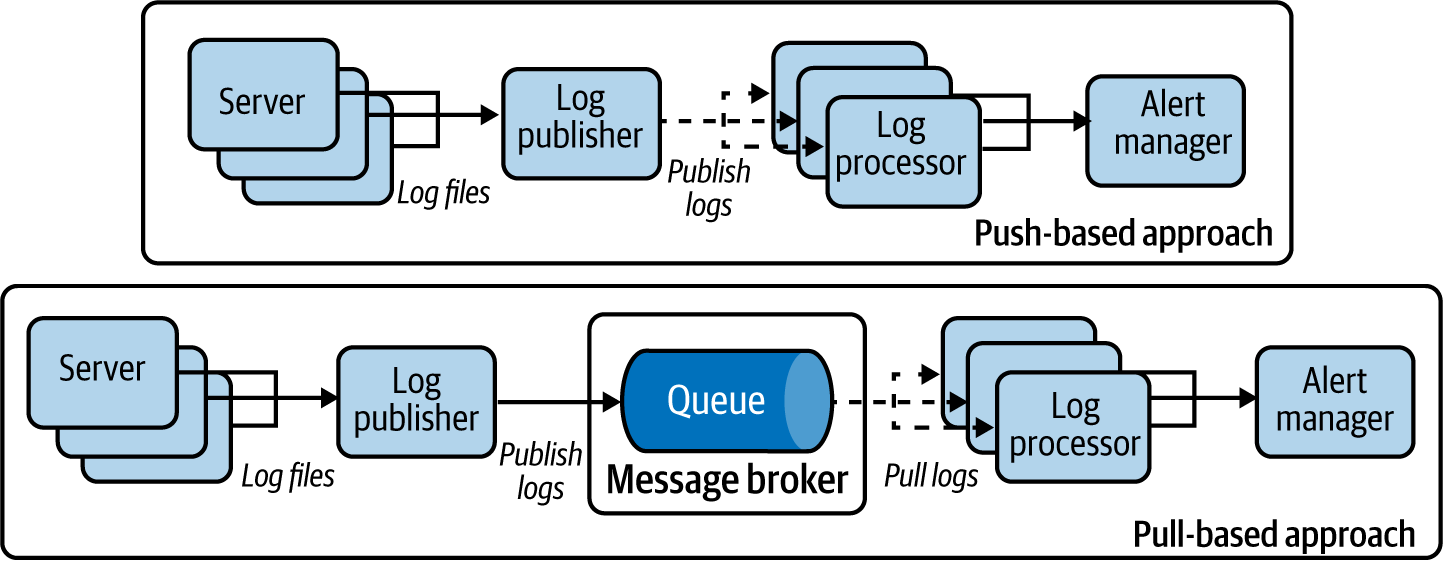
**Ensure event delivery**

Connecting producers and consumers that are not online at the same time is another key use case. For example, if we need to publish events to battery-powered wireless devices, we cannot guarantee that consumers will be online to consume the updates in real time. Devices may undergo network connectivity issues or simply run out of battery power. In this case, we can decouple the applications through queues so consumers will be guaranteed to fetch the events eventually.

Even if the producer, consumer, and message broker are online, events can get lost because of network failures. This can be overcome by using an at-least-once delivery guarantee provided by the message broker. The message broker uses acknowledgments to guarantee that the events are delivered. When the broker receives an event, it persists that durably and then sends an acknowledgment to the producer, stating that it has successfully consumed the event. The producer is updated to wait for the acknowledgment from the message broker when it publishes an event. Similarly, when the message broker delivers the message to the consumer, it expects an acknowledgment, and therefore the consumer is updated to send an acknowledgment when it receives an event. If the message broker does not receive the acknowledgment, it will try to redeliver the event, and so the consumer can get duplicate events. Messaging protocols such as AMQP support this functionality.

**Handle sudden bursts of events**

We can also use this pattern to buffer event bursts over a short period. We can queue and process events without extensively scaling consumers. In the example in [Figure 5-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#handling_bursts_of_logs_with_push__and), a data processing organization periodically pulls logs from servers for processing, resulting in a burst of logs every time we fetch a new log file. If we publish the burst of logs directly to the log processors, they will become overloaded and fail. But by buffering with a queue, we store and process the logs at the capacity of the log processors. Further, as they are using a pull-based approach, the consumers will not be overloaded as excess events are buffered at the event queue.



**Figure 5-2. Handling bursts of logs with push- and pull-based approaches**

**Share workload with fairness**

This pattern can be used to share the workload among multiple workers. Taking the previous example of processing log events, we have multiple log processors consuming events from the message broker in parallel and at their own speed. As the log processors are competing to pull events, we can ensure that events are processed as soon as possible. This also ensures that events are processed as long as at least one log processor is available, and no log processor will idle as long as events remain in the queue. This pattern also allows us to prioritize the oldest events by using FIFO semantics. This mimics queues in the real world: the first person in the queue will be served first. The Producer-Consumer pattern allows us to model the same event-processing semantics on cloud native applications.

**Considerations**

Most message brokers provide support for an at-least-once delivery guarantee. If the consumer fails to acknowledge the event delivery or times out, the broker will send that event to another consumer, making sure that the event is processed by at least one consumer.

A single message broker can host multiple event queues. We recommend that each use case and operation has its own queue; for example, events updating customer information might have their own queue, while events updating payments will have their own queue. Mixing different events in a single queue requires us to improve consumers to differentiate between events and handle them appropriately; this complicates the design, and slows performance as the number of events they receive increases.

When using queues, we have to be careful to handle bursts. While queues buffer events, there should be enough capacity to process all produced events over a certain time period. If we are constantly getting bursts of events (as in the log processing example in [Figure 5-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#handling_bursts_of_logs_with_push__and)), and if the input rate to the queue is consistently higher than the consumption rate, this pattern fails. To mitigate this risk, we should scale the number of consumers to increase consumption capacity.

This pattern helps us build decoupled systems and allows us to independently add and remove producers and consumers, scaling overall event processing.

**Related patterns**

The Producer-Consumer pattern is related to the following patterns (all covered in this chapter):

*Publisher-Subscriber pattern*

Can send the same event to multiple consumers for processing.

*Fire and Forget pattern*

Used when events need to be delivered to a single consumer with an at-most-once delivery guarantee without the help of a message broker.

*Store and Forward pattern*

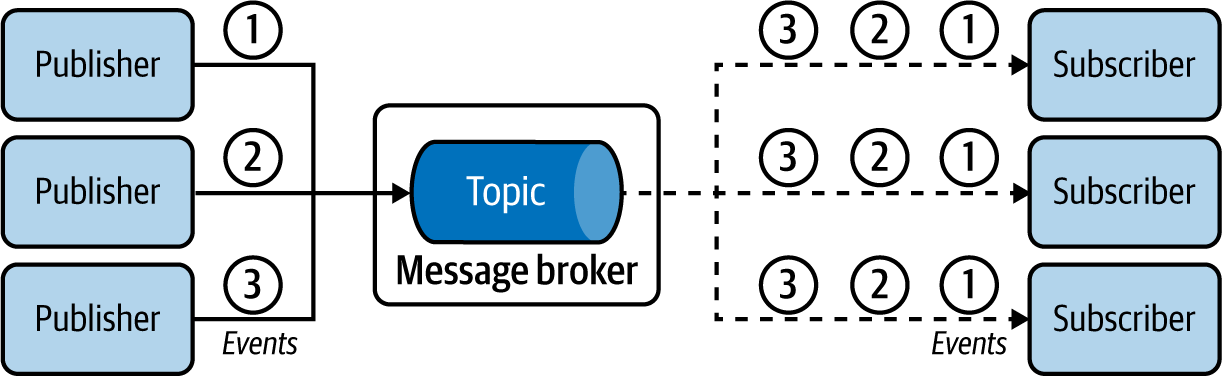
Allows asynchronous at-least-once event delivery without a message broker.

**Publisher-Subscriber Pattern**

The *Publisher-Subscriber pattern* enables applications to communicate asynchronously by using topics. The topic delivers every event to every subscriber.

**How it works**

This pattern uses *topics* to propagate the events from publishers to subscribers. The topic is a message broker concept. Multiple publishers can submit events to a topic hosted in the message broker ([Figure 5-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_delivery_from_publishers_to_multi)). The topic then publishes all those events to all its subscribers, and makes sure that every subscriber receives all incoming events.



**Figure 5-3. Event delivery from publishers to multiple subscribers**

The standard behavior of the pattern is best effort: the events are delivered at most once. When a subscriber misses an event due to unavailability or network failure, they will never receive the event. But we overcome this problem with a *durable subscription*, which guarantees that all messages are delivered to all the consumers at least once, accounting for subscribers who are temporarily unavailable.

**How it’s used in practice**

This pattern is used for scenarios such as broadcasting notifications in parallel to multiple recipients with both the best effort and at-least-once delivery guarantees. Let’s see how we can achieve these.

**Broadcast events**

This pattern is ideal for broadcasting information. For example, let’s assume we are building a system like Twitter, and we can use this pattern to send a notification of all tweets published about sports with #sports (the topic) to all users who have subscribed to that hashtag. The main advantage of this pattern is that it enables us to notify all interested subscribers, instead of only a single user or application.

**Deliver events with best effort**

Typically, events produced by publishers have the at-most-once delivery guarantee. For instance, subscribers do not receive events published during an outage. Though you risk losing events, this approach is still useful for scenarios such as status updates—like publishing current weather periodically to the weather topic so subscribed people can plan what to wear when they go out. Even when some updates are missed, the subscribers can appropriately take corrective decisions based on future events.

**Make sure all events are delivered to all subscribers**

Event delivery with best effort is not suitable when missed events impact the function of the system. For example, someone might be tracking a topic to find out the closing date to apply for a state exam. A subscriber who misses that event will miss the deadline and miss the chance to sit for the exam. You can use durable subscriptions to ensure that subscribers receive missed events when they are online again.

**Selectively deliver events to subscribers**

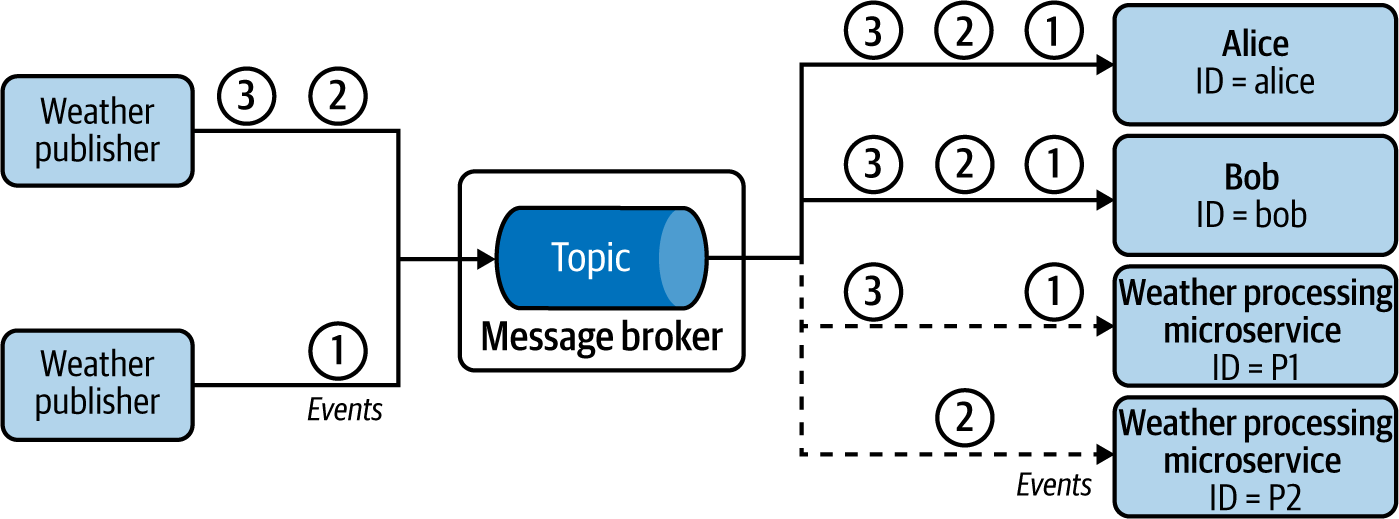
This pattern is useful when only specific events require delivery, in which case the most common approach is to leverage hierarchical topics. The topic name has a hierarchy such as news, news/sports, and news/politics. If the subscriber subscribes to the news topic, they will consume all events from news and all its subcategories, including news/sports and news/politics. But if they are interested in only sports news, they can subscribe to only news/sports.

An alternative approach for achieving selective delivery is to use filtering logic. A filtering condition such as news==sports is passed to the message broker upon subscribing to the topic, so that the message broker publishes only the sport news events to the subscriber.

**Share workloads**

This pattern can also be used to replicate and distribute events across multiple workers. In the example in [Figure 5-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#replicating_and_distributing_weather_ev), we process periodically published weather data in real time. We need to publish every event to human subscribers (Alice and Bob) so they can keep track of the weather, and, in parallel, distribute events across distributed Weather Processing microservices to share and process the messages.

To achieve this, we use the client ID of the subscribers. Each subscriber has a unique client ID such as alice, bob, or P1. Since the topic keeps track of event delivery based on client IDs, it ensures that all events are delivered to both human subscribers with the client IDs alice and bob. We set the same client ID for both instances of the Weather Processing microservices to P1. By doing this, the topic publishes each event to only one of the microservice instances. This enables us to share the workload across those microservices.



**Figure 5-4. Replicating and distributing weather events across multiple microservices**

**Considerations**

When subscribers cannot miss events published during downtime, we need a durable subscription. The message broker takes the responsibility of storing and delivering the events to subscribers as they come back online. Each durable subscription can be viewed as a dedicated event queue for each subscriber.

Alternatively, we can also use a commit log–based message broker such as Kafka or NATS to receive missed events published during subscriber downtime. These message brokers store all events to commit logs. Because they do not remove the events even after a successful or failed delivery attempt, upon request they can resend/replay previous events to subscribers. To retrieve missed events, subscribers must persist the last-processed event sequence ID, and during restart, they request the message broker to replay all the events since the last-processed sequence ID. This allows the system to achieve a higher delivery guarantee.

This pattern enables us to build decoupled systems and independently add or remove publishers and subscribers. Use the Producer-Consumer pattern when you need to share events among multiple consumers; but, as shown in [Figure 5-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#replicating_and_distributing_weather_ev), you can use topics with subscriptions based on client ID to broadcast events to some subscribers while distributing the events across a subset of subscribers sharing the same client ID.

**Related pattern**

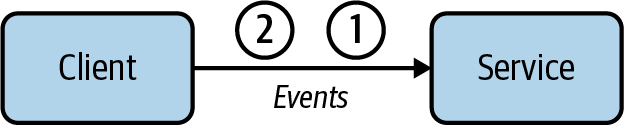
The Publisher-Subscriber pattern is related to the preceding Producer-Consumer pattern, which provides the capability to send an event to only one consumer for processing.

**Fire and Forget Pattern**

The *Fire and Forget pattern* enables clients (producers) to send events to respective consumers (services) with an at-most-once delivery guarantee without the use of a message broker. This pattern sends events by using standard APIs, discussed in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern).

**How it works**

Let’s imagine that a weather sensor periodically sends current temperature and humidity readings to a weather-prediction service hosted in the cloud. Because of technical limitations, instead of using a message broker, as depicted in [Figure 5-5](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#fire_and_forget_event_delivery_from_cli), it is designed to invoke the API of the service by using protocols such as HTTP. The client is interested only in whether the server received the events, and is not interested in the final outcome. When a client publishes an event to the service, it expects only an acknowledgment of the event by a relevant HTTP status code such as 202 Accepted.



**Figure 5-5. Fire and Forget event delivery from client to service**

**How it’s used in practice**

This pattern is useful when we need best-effort delivery of noncritical data, or when the receiving service does not possess the capability to subscribe or pull events from a client.

**Deliver events with best effort**

As in the preceding example, the weather service tries to deliver the event only once, and upon failure discards the event. This is acceptable for this use case, as the service can continue the real-time predictions based on future events delivered by the client.

**Deliver events to systems that do not support subscription**

This pattern can also be used when the client is sending events to a third-party service that does not possess the capability to subscribe and pull events from a message broker. Services owned by partner organizations usually are deployed behind an API. They have the capability to consume events only via protocols such as HTTP. We also use this pattern when the client is hosted in an internal network and the service cannot initiate a connection to the client.

**Considerations**

This pattern is useful when processing events that have depreciating value over time, such as the current weather. These events become outdated when their processing is delayed. Instead, processing more-recent events is significantly more valuable. The Fire and Forget pattern enables us to quickly discard events and pick up the next one for processing. We should use the Producer-Consumer pattern if we do not want to discard events when they cannot be delivered.

If needed, we can design the clients to retry event delivery upon failures to improve the success rate of delivery. In the preceding example, the weather client can retry several times until it receives a 202 Accepted response from the server. But this does not provide the at-least-once event delivery guarantee, because if a service outage occurs for a sustained period of time, the client is not designed to continuously accumulate all the incoming events until the service is back.

Say we need the client to load-balance events across multiple services—for example, a single weather client sending events in a round-robin manner to five weather services. Here, we either have the weather client use an intermediary such as a network load balancer to route the events to available services, or improve the client to keep track of all the weather service endpoints and perform the load-balancing logic within itself. We do not recommend the latter approach, as it introduces additional complexity to the weather client.

**Related patterns**

The Fire and Forget pattern is related to the following two patterns (both covered in this chapter):

*Producer-Consumer pattern*

Allows events to be delivered to a single consumer with higher delivery guarantees, as the consumer subscribes to an event queue.

*Store and Forward pattern*

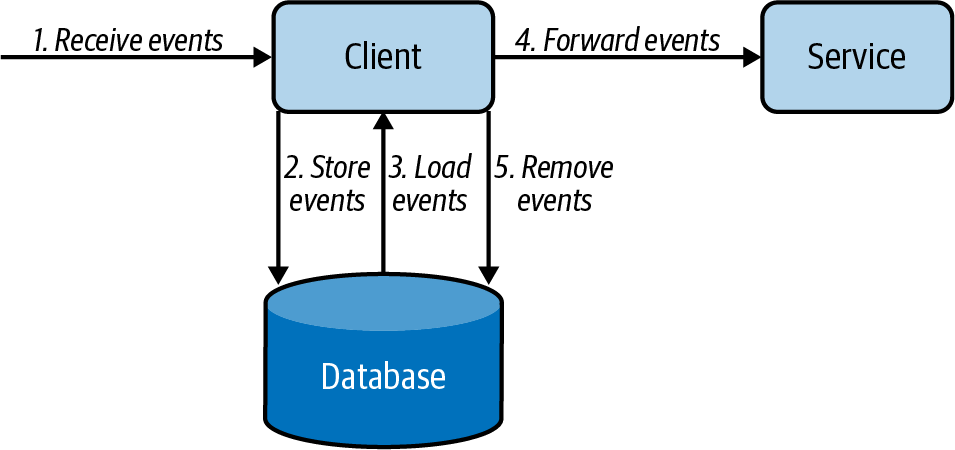
Delivers events to service endpoints with an at-least-once delivery guarantee.

**Store and Forward Pattern**

The *Store and Forward pattern* enables clients to send events to services with an at-least-once delivery guarantee. As with Fire and Forget, this pattern does not use message brokers but uses APIs to directly send events.

**How it works**

This pattern requires a complex client design to achieve the at-least-once event delivery guarantee. The client in this pattern first persists the events to a durable store, such as a database or queue in a message broker, before attempting to send them to the service ([Figure 5-6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#store_and_forward_event_delivery_from_c)). Upon successful event delivery, the client purges the events from the store. If delivery is unsuccessful, it retries to send the event. During this, as the client receives more events to send, it persists them to the store. Once the connection to the service is established, it will deliver all pending events, receive acknowledgment of the event consumption, and purge the events from its store.



**Figure 5-6. Store and Forward event delivery from client to service**

**How it’s used in practice**

This pattern is useful when delivering critical data with a message broker, or when the receiving service cannot subscribe or pull events from the client.

**Deliver events to services that do not support subscription**

Let’s assume we are publishing purchase order events to a partner service to fulfill the delivery. The partner services are usually hosted behind an API, and we assume they do not have the capability to subscribe and pull events from a message broker. We can use this pattern to call the service APIs via HTTP with an at-least-once delivery guarantee.

**Ensure event delivery during service unavailability**

In the preceding example, our organization does not have control over the availability of the service hosted by partners and third parties. In this situation, we use this pattern to store all incoming events and then deliver those events to the services as they become available.

**Considerations**

We recommend this pattern when message brokers cannot be used and when we need the events to be delivered with an at-least-once delivery guarantee. This is because using message brokers and adopting the Producer-Consumer pattern greatly simplifies the architecture and reduces the operational cost.

When using this pattern, use a separate durable store (database or queue) for each client application, when possible, to greatly simplify the design. For scalability reasons, let’s say we use five clients to send the order events to third-party services. But instead of using a dedicated store for each client, we recommend using a common event queue for all five clients to store the events. This allows us to distribute the events among the clients when they try to send events to the services. This also helps overcome client failures, as now other clients can fetch and publish events that are supposed to be sent by the failed client.

If you decide to use databases as the durable store, and still want to use multiple clients for event delivery, you must solve the problem of deciding which client publishes which event. We need to prevent multiple clients loading the same order event from the database and delivering it to the service. This causes duplicate events and risks overloading the service. To overcome this, we elect a single client to deliver a particular subset of events (for example, based on the hash of the event order number). Here, client selection is determined via leader election by using services such as ZooKeeper. Additionally, the same client can also deliver multiple subsets of events (for example, event order number hashes 2, 5, and 7).

**Related patterns**

The following patterns, covered in this chapter, are related to Store and Forward:

*Fire and Forget pattern*

Publishes events to service endpoints with an at-most-once delivery guarantee.

*Producer-Consumer pattern*

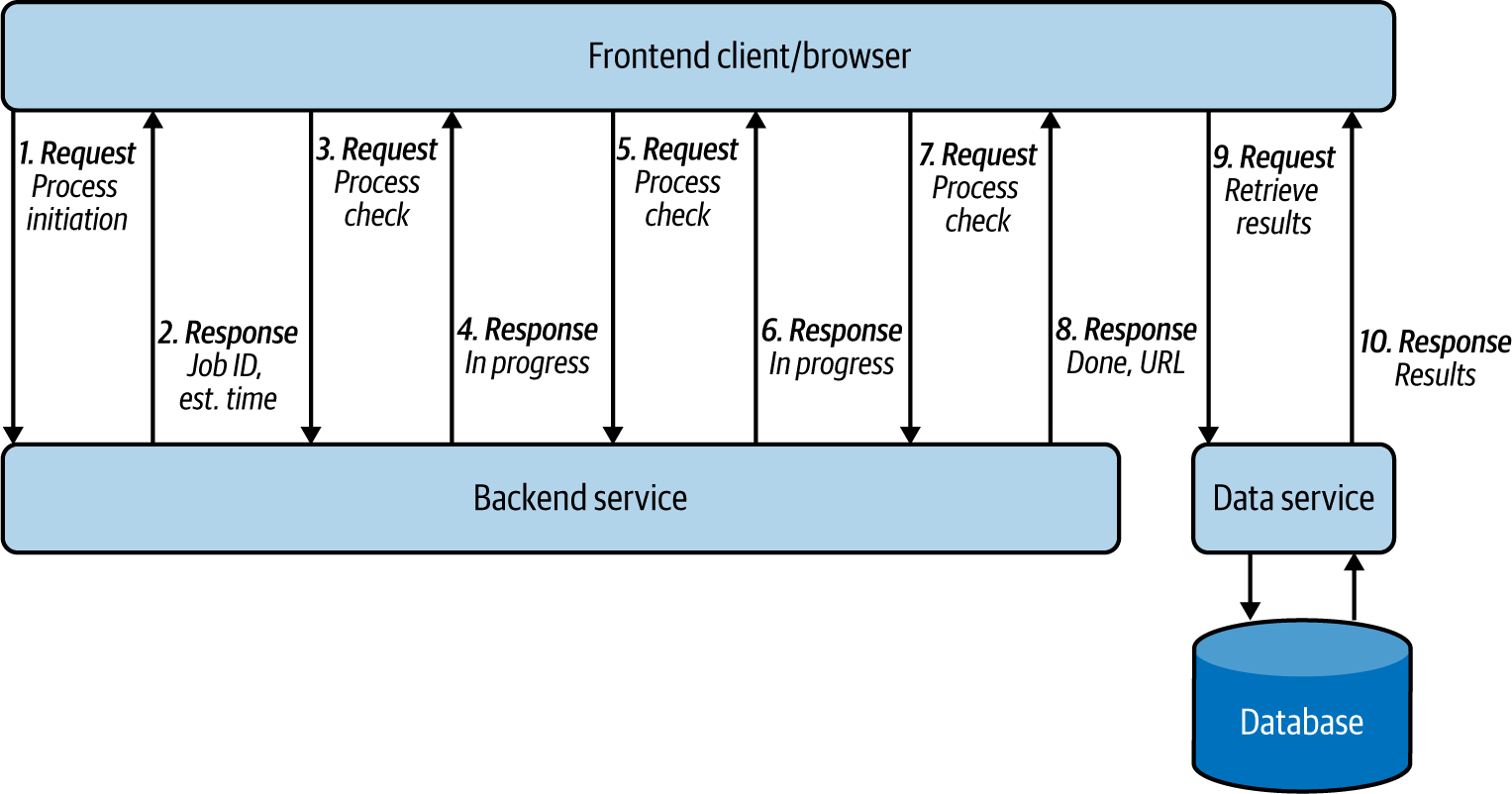
Allows events to be delivered to a single consumer with higher delivery guarantees when the consumer can subscribe to an event queue.

**Polling Pattern**

The *Polling pattern* enables clients such as web browsers to initiate a long-running job, periodically checking completion.

**How it works**

The frontend client or browser sends a request to initiate the process, such as insurance claim processing ([Figure 5-7](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#frontend_clientsolidusbrowser_repeatedl)). Because the processing takes time, the backend service immediately sends an acknowledgment stating that it has accepted the request and initiated the asynchronous job processing. Along with the acknowledgment, it sends a job ID and, potentially, an estimated time of job completion. Based on this information, the client periodically queries the backend to check if the claim processing has completed. Upon completion, the backend returns the results as part of the response to the query, or provides a redirection to an endpoint containing the results.



**Figure 5-7. Frontend client/browser repeatedly calling backend service to retrieve asynchronous job results**

**How it’s used in practice**

This pattern is useful when we need to retrieve the result of an asynchronous job without using a subscription or callback.

**Retrieve results from a long-running process that cannot notify of job completion**

When we integrate frontend and backend applications, usually only the frontend can initiate the connection to the backend, and backends cannot easily call frontends. When the backends are processing an asynchronous task, they can often take more time than the connection time-out, which means clients cannot stay connected to receive the results. In this case, we use this pattern to repeatedly call the backend for results.

This mimics repeatedly calling the insurance company to check if a claim has been processed. Even though it is not ideal, we need to design the client in this way when the backend systems have no way to automatically inform the client of an update.

**Deliver events to a client that cannot initiate subscription or callbacks**

This pattern is used when clients cannot subscribe to a message broker or expose an endpoint to receive updates from the backend. For example, in the preceding example, the browser cannot directly subscribe to the message broker, so it periodically polls for new updates.

**Considerations**

When implementing this pattern, we should ensure that the life cycle of the asynchronous job is maintained in the backend, because the frontend clients may fail and get restarted. For example, when the browser window is refreshed, the user should continue to get the correct status update of the insurance claim. Therefore, when the browser initiates the connection after refresh, the backend application should be able to correlate the new request to the previously initiated job, using the available information on the request, and return the appropriate response to the client.

You should keep in mind that polling backends in a continuous manner is a waste of resources for both the client and service, and adds delay to the response, as the backend service cannot inform the client until the next poll. The amount of continuous polling can be reduced by using the long polling technique: the service does not immediately send a response but holds the connection until the response is available, or until the connection times out. This reduces the number of polls and allows services to immediately respond when they have necessary data. We recommend using the long poll if the connection time-out between the client and service is reasonably high, the network is usually stable, and the service has the capacity to hold the request until connection time-out; otherwise, fall back to periodic polling.

We do not recommend using this pattern when the application supports callbacks such as webhooks or WebSockets for communication, because those options are efficient and much less resource intensive. Callback-based event delivery is discussed in detail next.

**Related patterns**

The following patterns (covered in this chapter) are related to the Polling pattern:

*Producer-Consumer pattern*

This is an alternative used when the participating applications publish and subscribe to a queue.

*Request Callback pattern*

This is also an alternative that’s used when the clients and services are capable of using WebSockets or webhooks.

**Request Callback Pattern**

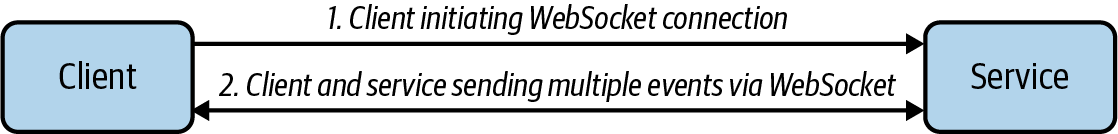
The *Request Callback pattern* enables applications to communicate asynchronously. The application provides the callback information with the request so responses can be delivered to the given callback.

**How it works**

In this pattern, one application should initiate the request with the callback information so that the responding application can deliver the responses asynchronously by using the callback. This pattern builds on top of the Asynchronous Request-Reply pattern from [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns), by providing two variations: using WebSockets or webhooks. Let’s see how they work.

**WebSockets**

To use WebSockets, both client and service should have the capability to communicate via the WebSocket protocol ([Figure 5-8](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#client_and_service_communicating_via_th)). The client initiates the connection to the service and establishes a long-running connection. Both the client and service persist the connection and communicate by sending events. This approach is used for clients requesting information via an event and waiting on the service to respond, or for exchanging multiple events. WebSocket is an HTTP-based technology, but HTTP2 and gRPC also provide similar callback-based communication.

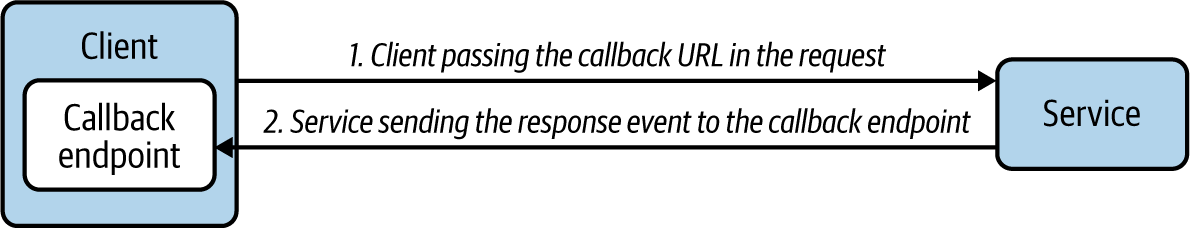


**Figure 5-8. Client and service communicating via the WebSocket protocol**

GraphQL uses WebSocket with its subscription feature, allowing clients to connect to a service and listen to real-time events according to the GraphQL query submitted when subscribing. Clients will then continuously get updates when the data in the service changes. A client can unsubscribe by sending a message to the server, or the server can unsubscribe due to errors or time-out. Refer to [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns) for more details on GraphQL.

**Webhooks**

In this approach, the client application issues a request and has the response delivered to a callback endpoint ([Figure 5-9](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#client_and_server_communicating_via_web)). The client sends the request with a callback URL. If the callback URL is consistent, we configure that on the service side, so we do not have to redundantly send the URL with the request. The response, when generated, is delivered to the callback URL.



**Figure 5-9. Client and server communicating via webhook**

Webhooks can be implemented in multiple ways. One way is to use WebSub. This open protocol, initially designed to extend Atom and RSS protocols for data feeds, has been adopted by the World Wide Web Consortium (W3C) as a Candidate Recommendation since April 2017. WebSub uses Publisher, Hub, and Subscriber microservices and uses HTTP for communication. The publishers publish content to the hub, via HTTP calls and by using HTTP headers to denote the topic information. The subscribers discover the hubs for their interested topic and make an HTTP POST request to the hub with their topic and callback URL. They will then get served with the relevant content through the HTTP POST calls by the hub, as the hub gets new data.

**How it’s used in practice**

We use this pattern to deliver responses asynchronously or when we need to receive continuous updates. This pattern mimics registering our telephone number as a callback so that the insurance agent can call to inform us of the status of our insurance claim. Following are some use cases in practice.

**Deliver response asynchronously**

This pattern is ideal when the service cannot respond within the connection time-out, such as in our insurance claim processing use case. The backend service acknowledges the request immediately, and delivers the results when the claim processing is completed. This also allows the backend to send the results instantly when the necessary data becomes available.

**Deliver updates continuously**

Though we can use this pattern to update the insurance claim status in the browser, this pattern is much more useful when we need real-time updates, such as monitoring stock prices. The browser establishes a WebSocket connection with the backend service to receive the latest updates and dynamically render them in the web page. This pattern can be used only when the client can receive the response from the server leveraging the WebSocket protocol, or by exposing an endpoint to be used as a callback.

**Considerations**

When implementing this pattern, the callback does not need to be an HTTP endpoint; it can be an email address, an event queue, or an event topic. We can also model this pattern so that when the service processing is done, instead of calling the client, the service calls another service to process the results based on the callback information provided by the client. When the result is big, the service can also store results in a durable store such as in Amazon S3, and then pass that URL to the callback so that the client can load the processed data.

The webhooks typically provide only an at-most-once delivery guarantee, as the service has to drop the response events when the callback is not available or if a network failure occurs. We can improve this to an at-least-once delivery guarantee by incorporating the Store and Forward pattern when delivering events to callbacks or by using a message broker when the participating applications have the capability to communicate via a message broker.

We recommend choosing WebSocket over webhooks when the client and server need to asynchronously communicate by sending more than one message. This is because WebSocket keeps the connection live throughout the communication, and reduces the cost of sending each new message. Subscribing to a stock symbol and receiving continuous stock price updates is a good example for this. At the same time, we recommend using webhooks over WebSocket when the client is expecting only a single response, and when the response time cannot be determined or if the response can take more than a few minutes (for example, when expecting the outcome of an insurance claim).

**Related patterns**

The following are related to the Request Callback pattern:

*Store and Forward pattern*

Complements this pattern by providing guaranteed callback event delivery (covered previously in this chapter).

*Polling pattern*

Provides an alternative when applications cannot establish callbacks (covered previously in this chapter).

*Asynchronous Request-Reply pattern*

An alternative approach to communicate asynchronously by using a message broker. This pattern is covered in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns).

**Summary of Event-Delivery Patterns**

This section has outlined common event-delivery patterns used by cloud native applications that are built in an event-driven architecture. [Table 5-1](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_delivery_patterns) summarizes when we should and should not use these patterns, and their benefits.

| **Pattern** | **When to use** | **When not to use** | **Benefits** |
| --- | --- | --- | --- |
| Producer-Consumer | There is a particular event to be consumed and processed by only one of the available consumers. We cannot confirm the availability of the consumers and producers. We see burst event production over a short period. We need to ensure fairness in the processing of events. | There is continuous high traffic, and the incoming event rate is much higher than the consumed event rate. Message brokers cannot be used in the solution, and producers or consumers cannot connect to a message broker. | Delivers an event to one consumer without duplicating the events. Can tolerate availability problems in consumers and producers. Handles spikes/bursts in traffic. |
| Publisher-Subscriber | In a notification delivery system. An at-most-once delivery guarantee is tolerated by subscribers. (It’s possible to miss the events if the subscriber is not available at the time of event notification.) We need selective delivery of events to the subscribers. | You cannot tolerate any missed events by the subscriber. Message brokers cannot be used in the solution, and producers or consumers cannot connect to a message broker. | Helps build an independent and decoupled system that can publish and subscribe. Scales well with multiple subscribers interested in the same topic. |
| Fire and Forget | At-most-once delivery guarantee is tolerated. Dropping events is acceptable, such as when delivering non-business-critical events. The consumer cannot pull any updates from the message broker. Only a set of consumers is available to be notified. | Any issues in delivering or processing business-critical events cannot be ignored. Message brokers can be included in the system, and producers and consumers can connect to a message broker. | No need to have a message broker to transfer messages from producer to consumer. Simple to implement and no need to have additional deployment and maintenance complexities. |
| Store and Forward | At-least-once delivery is required when both publisher and consumer are online and reachable at any time. Message brokers cannot be used in the solution. The consumer cannot pull any updates from the message broker. | Message brokers can be included in the system, and producers and consumers can connect to a message broker. | No need to have a message broker to transfer messages from producer to consumer. |
| Polling | Clients do not have the capability to subscribe to a message broker or to expose an endpoint to receive updates from the backend system. The service does not have the capability to call other endpoints upon completion of a job. You have long-running jobs. | You have short jobs where success/failure can be reported immediately. The applications can support callbacks such as webhooks, or WebSocket for communication. | Executes a long-running job and gets the response without having an additional infrastructure. |
| Request Callback | Handling the request can take more time than the typical connection time-out of a standard request. Clients are expecting updates from the services on one or more jobs. The applications have the capability to communicate by using WebSocket, or clients have the capability to expose a callback URL and services can call that URL to send updates. | Applications do not have the capability to communicate via WebSocket, or the clients cannot provide a callback. | Executes a long-running job and gets the response without increasing the traffic to service to continuously check for updates. More scalable approach, as updates are sent when the job is completed. |
| Table 5-1. Event-delivery patterns | | | |

**State Management Patterns**

In this section, we discuss how to build and maintain cloud native application state without coupling to a database, how to re-create application state at various times, and how to build applications with different domain models out of the same data. Here, we cover the Event Sourcing pattern, which is considered the foundation for building various other patterns such as the CQRS pattern introduced in [Chapter 4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch04.html#data_management_patterns).

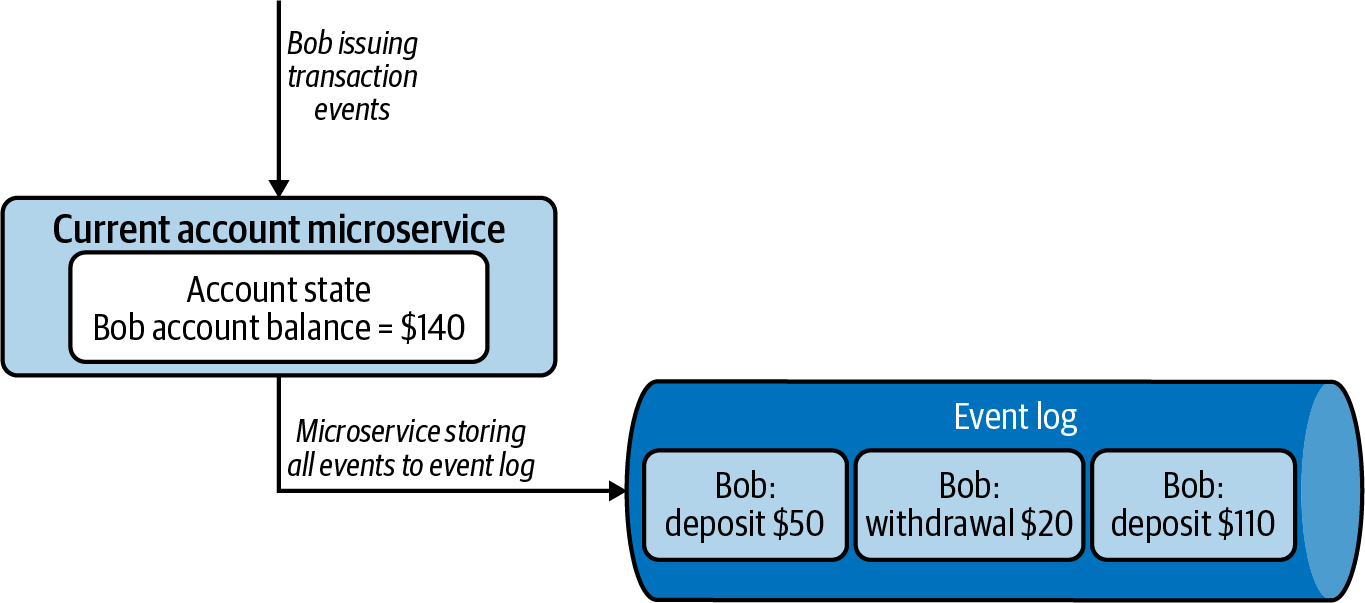
**Event Sourcing Pattern**

The *Event Sourcing pattern* enables us to store all changes to the application state as a sequence of events. This pattern not only is used to re-create application state at various points in time, and with different domain models, but also serves as an audit history to illustrate how we ended up in the current application state.

**How it works**

Every time an event updates the application state, the event is also recorded in a persistence store in the order of operation. [Figure 5-10](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#current_account_microservice_generating) shows a banking use case, in which transactions are performed on an account. Events occur, such as Bob depositing $110, withdrawing $20, and then again depositing $50.

If we consider only the current state of the application, we know only that Bob now has a balance of $140 in his account, but we do not know the events that led up to this. Now, as each event is stored in a persistence store (or event log), in the sequence of its occurrence, we can gain an understanding of how his account balance changed over time.



**Figure 5-10. Current account microservice generating an event log while updating application state**

Events are usually stored on a commit log, such as Apache Kafka, which allows us to read the events back from any point in time in a sequential manner by passing event sequence IDs. These events are read again by the application or other systems so that they can process the events that have occurred or re-create an application state with either a subset or all events.

**How it’s used in practice**

The Event Sourcing pattern unlocks the ability to system time-travel, build different domain models based on the same sequence of events, re-create a failed application state, run temporal queries, and replay events.

**Re-create application state**

Let’s assume the application state got corrupted during system failure. Since we have stored all the events corresponding to the state, we can simply replay all the events in order to re-create a specific application state. We also have to disable the system, prohibiting any notifications to external systems during state rebuilding. For example, while we are re-creating a state for Bob’s bank account, we should not send deposit and withdrawal notifications.

**Build different domain models**

Let’s assume Bob is enrolled in a rewards program that gives points on purchases he makes at affiliated stores. The bank uses a different domain model to calculate and keep track of the points Bob has earned. Instead of tightly coupling the rewards service with the core transaction application, with the Event Sourcing pattern the bank can asynchronously build Bob’s points from the transactions stored in the commit logs. This not only allows us to build specific data views to support different access patterns to reduce load, but also helps consuming systems—such as a rewards service—to evolve and change at their own speed without interfering with the core banking system.

**Run temporal queries**

This pattern allows us to run temporal queries on the stored events. For example, let’s say the bank charges a $5 monthly fee if the account balance falls below $100. Bob had $140 in his account at the end of the month. But just after the initial $20 withdrawal, his account had only $90. Hence, he will be liable for the $5 monthly fee. Without having the events in the commit log, achieving such use cases will become difficult. This visibility into the event log demonstrates a fully lossless architecture, unlocking the ability to time-travel and build various views based on historical data.

**Replay events**

This pattern also allows us to correct mistakes. Let’s say the bank has recorded that Bob withdrew $50 from his account on April 15, even though he successfully canceled the withdrawal. Because of this withdrawal entry, the bank also charged the $5 monthly fee on April 30. Assume that the bank found out about this mistake on May 3. By using the Event Sourcing pattern, the bank can replay all the transactions of Bob’s account starting April 15, correctly reverting the withdrawal and refunding Bob the erroneous $5 monthly fee charged on April 30.

**Considerations**

With this pattern, we need to decide whether the application state or the event log is going to be our single source of truth. If we are using a database to keep track of the application state, the database can be the source of truth, as it is durable. We can then use the event log only for auditing purposes and to generate other domain models. But at the same time, if we are keeping the state in memory (such as in a data structure, in-memory database, or cache), then we have to use the event log as the single source of truth, as we can always regenerate the state by replaying the events from the logs.

When using an event log as the source of truth, the recovery of system failure can take a long time; we need to re-create the application state by replaying *all* the events in the event log. To improve recovery time, we can periodically take application state snapshots, as we’ll discuss in [Chapter 6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#stream_processing_patterns-id00204), and during recovery, we can load the latest snapshot and replay only the events logged after that.

Performing event playback and re-creating an application’s state can be tricky, especially when the application is interacting with external services. If we need to stop the application from calling external services (such as notifying Bob again about his $50 deposit), we have to either make the service intelligent enough to know that it is performing a replay and so shouldn’t send any external calls, or gate the external services with APIs and drop the service calls at the APIs. If we are replaying events on multiple services and they need to communicate with one another, we recommend adding some sort of reference point, such as date and time, to their request when calling other services. For example, rather than requesting the current account balance, we can request the account balance on 03/23 at 11:15 a.m. In this case, the responding application will be able to always respond with the correct balance at the given time, increasing the consistency of the application during event playback.

When defining the events for a commit log, they should be modeled as *change* events. In the banking use case, we should use events that reflect some kind of change, such as deposit of $50 and withdrawal of $20, and not use events such as set bank balance to $150, or set bank balance to $130. If we reverse or remove the events during event playback, we will be able to get the corrected final balance.

When designing the service, we can either store the application state as simple objects and let the application operate on them, or we can model the state within the domain model itself. We recommend building the state within the domain model, as this gives us flexibility, especially when the processing logic is complex. But we cannot use this approach if we need to reverse the events. In this case, we would need to store the state after each update, and then revert to the previous state when a reversal is necessary. This can complicate the application architecture. If event reversal is needed, we recommend storing events separate from the application logic.

When building different domain models from the same event source, we should keep in mind that those models are usually built asynchronously and so can only be eventually consistent. This is because there can be network and application processing delays to write the events to the logs, and then other services to read and populate their application state. Hence, we should not use this pattern for use cases that do not tolerate eventual consistency. We can use log-based event queries with Apache Kafka and NATS as the event logs, for example, when building this pattern as they provide the capability to store events in order, and allow us to replay past events when necessary.

While this pattern allows us to time-travel, it also enforces restrictions in the event schema. For example, we can add new attributes to the event, but we cannot remove or update existing attributes. When the system replays previous events, they will not be compatible with the running application state. If we try to handle multiple versions of the event schema in the application code, the system can soon become very complex and difficult to maintain.

**NOTE**

The Event Sourcing pattern is inherently complex to implement and maintain, especially with a changing event schema, and with multiple external services communicating. We recommend using this pattern only if rebuilding an application’s state, or different domain models, is essential.

**Related patterns**

The following are related to the Event Sourcing pattern:

*Periodic Snapshot State pattern*

Used to generate data store snapshots so that application state can be rebuilt much faster. This pattern is discussed in [Chapter 6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#stream_processing_patterns-id00204).

*CQRS pattern*

Used to store commands so that multiple applications can be built to serve queries. This pattern is discussed in [Chapter 4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch04.html#data_management_patterns).

*Materialized View pattern*

Used to store data so that it can generate materialized views based on the events. This pattern is discussed in [Chapter 4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch04.html#data_management_patterns).

**Summary of State Management Pattern**

This section outlined the Event Sourcing pattern and its use in managing cloud native application state. [Table 5-2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#event_sourcing_pattern) summarizes when we should and should not use this pattern, and its benefits.

| **Pattern** | **When to use** | **When not to use** | **Benefits** |
| --- | --- | --- | --- |
| Event Sourcing | Multiple applications use the same data and need different domain models. Application state should be rebuilt. Temporal queries need to be executed in historical data. The system needs to time-travel and change past event occurrences. We need to keep track of audit information. | The data model is simple, and the consuming application can query for the intended data. The event schema changes in a continuous manner. We need all consuming applications to have data in a consistent state at all times. | Allows consumers to build application state optimized for their domain models and access patterns. Replicates the data into multiple applications, thereby increasing availability. Supports system recovery with event replay. |
| Table 5-2. Event Sourcing pattern | | | |

**Orchestration Patterns**

In this section, we cover various *orchestration patterns* that help build an effective event-driven architecture. These patterns resemble the service composition patterns in [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern), which coordinate and orchestrate synchronous service calls. The orchestration patterns predominantly use asynchronous event-delivery patterns discussed previously to manage the movement of events across multiple applications.

Though building cloud native event-driven applications on a small scale is straightforward, the architecture can soon become complex and difficult to manage when many microservices are involved. This section covers the Mediator, Pipe and Filter, and Priority Queue patterns that can be used to streamline event flow and manage the complexity of the application. These patterns are also considered foundations of event-driven architecture.

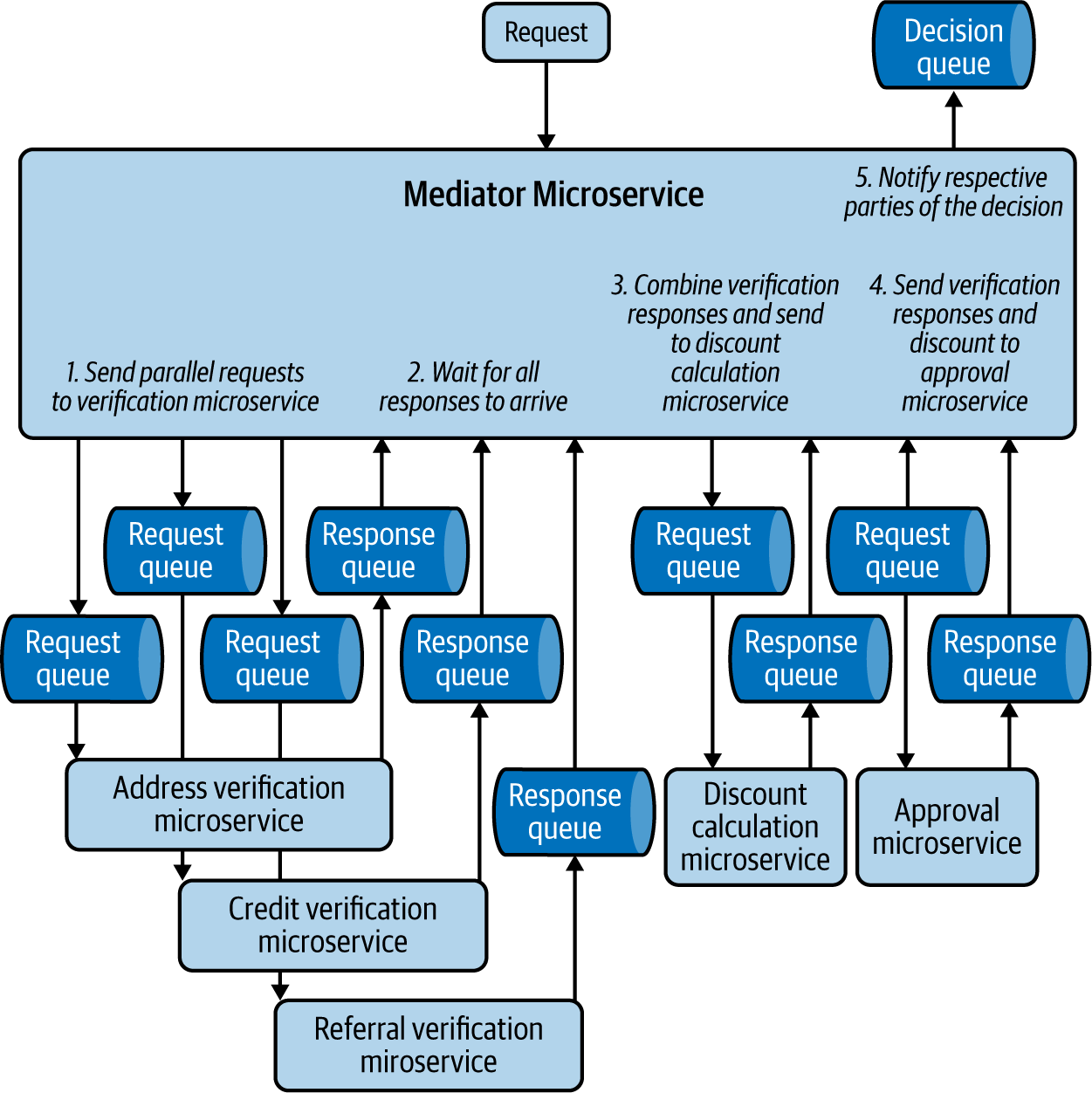
**Mediator Pattern**

The *Mediator pattern* provides centralized management of event orchestration. The mediator will not only understand and route events but also orchestrate events in sequential and parallel order across applications, while also handling failures. This pattern helps keep the coordination logic in a central location, allowing us to more simply change the behavior of the system.

**How it works**

The key element of this pattern is the mediator microservice runtime built as part of the cloud native application. It operates by interacting with all the microservices that integrate, via event queues, topics, and APIs. It connects on various protocols and transforms events for applications accordingly. These mediator microservices are usually stateless, and might need to perform only filtering, sorting, and event transformations. But when required, they can coordinate sequential and parallel tasks.

[Figure 5-11](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#mediator_microservice_orchestrating_a_n) depicts a use case of a new insurance request, which needs to perform address verification, credit verification, and referral verification in parallel. These are then followed by discount and final approval tasks. The mediator calls participating microservices by using the Asynchronous Request-Reply pattern from [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns). The mediator first passes events to the verification microservice in parallel, collects their responses, combines them, and then initiates the discount and approval tasks sequentially. Finally, it publishes the result to the decision queue so downstream systems can become aware of the decision.



**Figure 5-11. Mediator microservice orchestrating a new insurance request**

Mediators sometimes need to combine events from multiple systems, and so need to be stateful. In this example, each verification microservice sends its results at different times, and the mediator needs to persist the results until it receives all three, so it can create the combined event and send that to the discount microservice.

**How it’s used in practice**

We can use this pattern to sort and distribute events, split an event into multiple subevents, process those events in various tasks in both parallel and sequential order, and finally combine the results to generate an output.

**Sort and distribute events**

This pattern is used to sort events among multiple subprocesses. For example, as we are getting orders from a single queue, the mediator sorts the orders based on region. This also helps integrate decoupled producer and consumer applications, especially when they are external. In this case, mediators format the events and perform required protocol transformations.

**Split events into multiple subevents**

This pattern is also used to split one event into multiple events. For example, when we receive a new insurance application event, we need to split it into various subevents and send them to multiple systems to perform subtasks such as address and credit verification.

**Ensure task execution order**

The mediator pattern is used to perform some tasks sequentially while others are executed in parallel. This enables us to combine the results of multiple parent tasks before executing a dependent task. In our insurance scenario, we process verification tasks in parallel, and then the discount and approval tasks sequentially. The discount task is executed only when all the verification tasks have been completed and their results combined.

**Considerations**

Use this pattern instead of Pipe and Filter when the system is undergoing rapid changes. This pattern enables us to change the integration logic and operation flow by modifying the mediator. In the Pipe and Filter pattern, we need to update multiple applications and queues to perform the change.

As the mediator contains all the coordination logic, over time it can become complex and difficult to maintain. Split coordination logic among distinct separate mediator microservices so they are more manageable. Microservice integration with this pattern can also be done via configuration-based tools such as WSO2 Micro Integrator, Apache Camel, Siddhi, and BPMN frameworks.

Do not use this pattern when central control for orchestration is not required. When this pattern is overused, it will provide all the orchestration responsibility to a single team, which can constrain the autonomy of other teams, going against the principles of cloud native application development.

**Related patterns**

The following patterns, covered in this chapter, are related to the Mediator pattern:

*Pipe and Filter pattern*

Provides a decentralized approach to orchestrate events across applications.

*Event-delivery patterns*

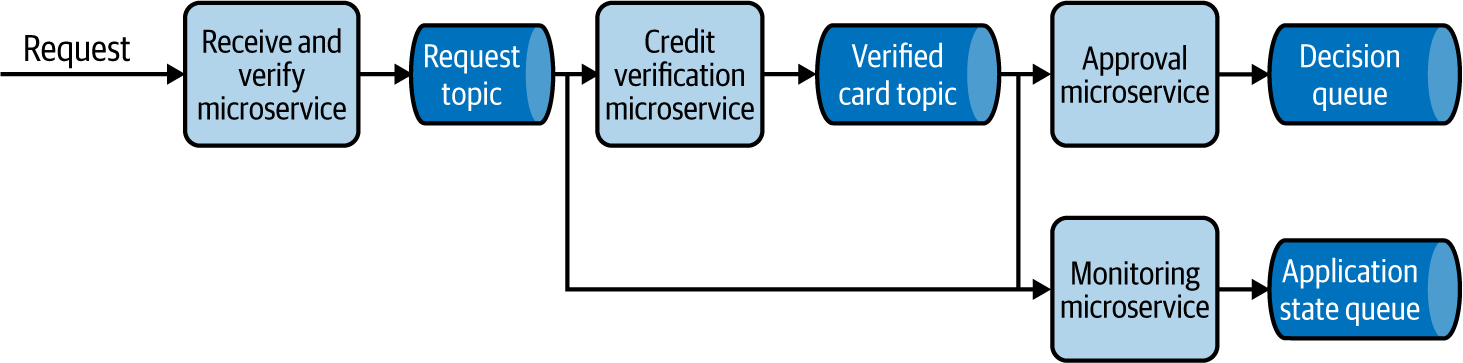
The Mediator pattern uses event-delivery patterns to communicate among applications.

**Pipe and Filter Pattern**

The *Pipe and Filter pattern* orchestrates events in a decentralized manner. It uses multiple event queues and topics to streamline the event flow across multiple microservices in a cloud native application.

**How it works**

This pattern uses event queues and topics to connect microservices. With this approach, we can build a very large graph of microservices via topics and queues to fulfill our business requirements. [Figure 5-12](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#a_credit_card_application_processing_fl) shows a credit card application processing flow using the Pipe and Filter pattern. Various microservices are integrated with one another via topics to asynchronously process and monitor the credit card application, and they publish the results to event queues for other microservices to consume.



**Figure 5-12. A credit card application processing flow using the Pipe and Filter pattern**

**How it’s used in practice**

Use this pattern when you need to build large-scale asynchronous systems that are managed by multiple teams.

**Build large-scale decoupled systems**

As the microservices are connected to each other via asynchronous protocols such as topics and queues, this pattern decouples services. We can continuously add new microservices to the topics and queues and extend the processing flow without impacting existing microservices.

In our example of the credit card application processing workflow, if we need to update users about their credit card application state, we introduce a notification microservice to consume from the application state queue and send updates to users. Similarly, we can attach a completely new workflow of printing the physical credit cards and shipping them to users by connecting it to the decision queue.

**Seamlessly add and remove functionality**

This pattern enables seamless addition and removal of microservices to the pipeline, and allows teams to add new business logic with minimal effect on other microservices. In the our credit card example, you can see how the status monitoring microservice is seamlessly integrated at each step without impacting the card processing event flow. If we need to add new functionality such as introducing income verification, we can simply add the income verification microservice between the Credit Verification and Approval microservices with minimal effect on the overall cloud native application.

**Provide segregation of duties**

As the events are distributed via multiple topics and queues, this pattern is ideal for providing a segregation of duties among teams. This allows each team to consume events, process them, and output corresponding events independently. If we are modeling a credit card application workflow, for example, then receiving and verifying the requests, performing credit checks, and taking approval decisions can be handled by various teams.

**Considerations**

Use this pattern instead of the Mediator pattern when building large-scale asynchronous systems with independent teams. This way, we can delegate the orchestration responsibilities to each team, as opposed to centralizing control of the event flow with the Mediator pattern. We recommend using the Mediator pattern when central control should be established across multiple services.

Because this pattern enables multiple teams to collaborate, it is vital to have well-defined event schemas. Also use a schema registry to store schemas to enable autonomous discovery and consumption of events.

At times we are interested in only specific events, such as credit card request events related to a specific region, so we can treat them differently according to the laws of that region. The microservices use filters in their subscription to consume only the events that they are interested in, or if they consume events from a topic, they can perform the filtering within the microservices.

Do not use this pattern if the flow of events changes frequently—for example, when you are rapidly experimenting or innovating on the event processing logic. More pipes and filters may need to be modified to accommodate the changes, which is more costly than performing changes via the Mediator pattern.

**Related patterns**

The following are related to the Pipe and Filter pattern:

*Mediator pattern*

Provides a more centralized approach for orchestrating event flow in event-driven architecture. This pattern was covered previously in this chapter.

*Event-delivery patterns*

The Pipe and Filter pattern uses event-delivery patterns, also covered in this chapter, to communicate among microservices.

*Saga pattern*

Uses this pattern to implement data processing pipelines that can support compensation transactions. [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern) describes this pattern.

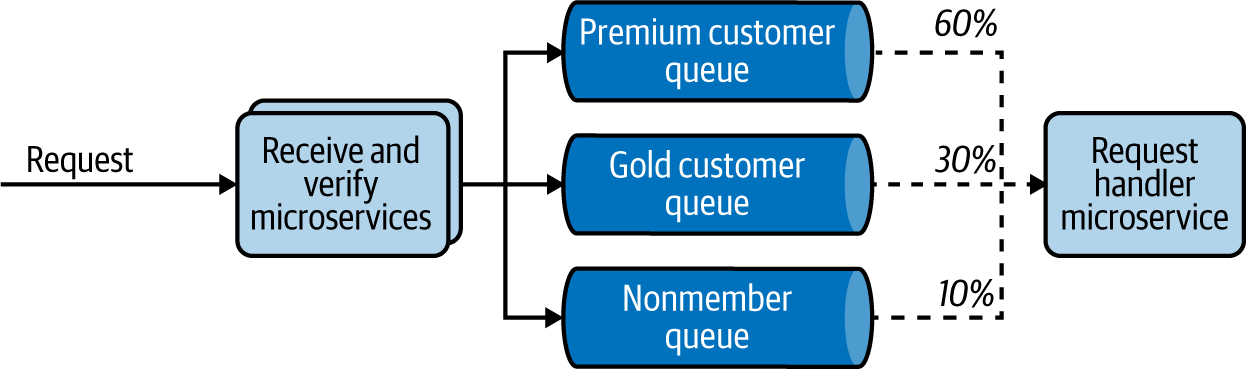
**Priority Queue Pattern**

The *Priority Queue pattern* handles events based on their priority so that high-priority events are handled first, while low-priority events are processed as capacity allows.

**How it works**

This pattern combines multiple queues, as in the Producer and Consumer pattern, to enable prioritized event processing. We achieve this by building a polling client that uses multiple event queues to process events based on priority.

[Figure 5-13](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#priority_based_customer_request_handlin) shows a system that handles customer requests. We need to give 60% priority to premium customer requests, 30% to gold customers, and 10% to nonmembers. We design this by making the request-handler application poll the premium queue 60% of the time, the gold queue 30% of the time, and the nonmember queue for the last 10%. You can also apply more logic to optimize what to be done with spare capacity.



**Figure 5-13. Priority-based customer-request handling**

**How it’s used in practice**

This pattern is needed when we want to preferentially handle some events over others or have insufficient capacity to process all events.

**Process some events quicker than others**

This pattern allows prioritization of important tasks. We presented a typical case previously: processing premium membership customer requests with higher priority over gold membership or nonmember customers.

**Optimize resource utilization**

Because of financial or other reasons, if we have constraints on the available processing nodes, we use this pattern to ensure that we process only the highest-value events. For example, if we are analyzing transaction fraud, we can categorize customer transactions by size and process them in that order. The application will always look to first analyze events from the largest transaction queue and then process events from the smaller queues only as capacity allows. This allows us to focus on the potentially higher-impact fraud investigations with our limited resources. Additionally, we should monitor queue depth and discard events over time that have become of little value.

**Considerations**

When applying this pattern, at times the client application will not have the capability to perform a polling operation. This may be because it is an external system outside our control. In this case, implement the polling client as an intermediary application that performs the prioritization and pushes events to other systems.

It is important to implement a cleaning task to discard old events based on queue depth; especially when the input event rate is higher than the processing rate, the low-priority queue can have stale events for a long time. Alternatively, you can design the application to promote events to a higher-priority queue when it has capacity and the events have stayed for a considerable time in the lower-priority queue. We recommend the prior approach when we do not have the mandate to process all events, and recommend the latter when we need to process all incoming events, and if we do not want the lower-priority events to starve when there is a steady flow of high-priority events.

You can also consider implementing this pattern by using a single topic and subscription filters, as we discussed in the [“Publisher-Subscriber Pattern”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#publisher_subscriber_pattern). In our customer-request processing example, we can deploy six microservices that subscribe with a membership==premium subscription filter, three microservices with membership==gold, and one with membership!=premium AND membership!=gold. This allows us to simulate the desired behavior. But this cannot calculate queue depth for each type of request to discard events, or promote events from a lower to a higher priority.

We recommend using this alternative only if there is enough capacity to process all incoming events. In addition, do not use this pattern unless priority-based processing is necessary for your use case, as it would introduce unnecessary complexity to the architecture.

**Related patterns**

The following patterns, both covered in this chapter, are related to the Priority Queue pattern:

*Publisher-Subscriber pattern*

With subscription filters, this can provide an alternative design to implement the Priority Queue pattern.

*Producer-Consumer pattern*

Can be used when prioritization is not necessary.

**Summary of Orchestration Patterns**

This section has outlined common orchestration patterns for building cloud native applications using an event-driven architecture. [Table 5-3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#orchestration_patterns-id00211) summarizes when we should and should not use these patterns, and their benefits.

| **Pattern** | **When to use** | **When not to use** | **Benefits** |
| --- | --- | --- | --- |
| Mediator | To simply sort the events among multiple subprocesses. To split an event into multiple events based on the use case. Tasks need to be performed in a sequential or parallel order. The system undergoes rapid changes, and using the Pipe and Filter pattern requires more overhead during each change. | Central control for orchestration is not necessary. | Provides the central system for orchestration, so change management is relatively easy. |
| Pipe and Filter | To provide segregation of duties among multiple teams. To add/remove business logic into multiple stages of processing without impacting other teams. | The overall flow of events is changed frequently. Central control for orchestration is necessary. | Allows seamless addition and removal of applications to the pipeline. Increases decoupling and reduces impact among multiple teams. |
| Priority Queue | To treat one type of an event with urgency compared to another. Resources are constrained, and we can process only a subset of events. | There is no strong need for priority-based processing. | Better utilization of resources based on priority of jobs. |
| Table 5-3. Orchestration patterns | | | |

**Technologies for Event-Driven Architecture**

Interactions among cloud native applications in event-driven architecture can be implemented using various message brokers, integration solutions such as ESB, and simple service calls.

Patterns like Fire and Forget and Polling use simple service calls to deliver events. We discussed service calls and related technologies such as REST, gRPC, and Thrift in Chapters [2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns) and [3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern).

When building orchestration patterns, we need applications that perform mediation, filtering, protocol switching, and data transformation, and that run business logic. We can implement these applications from scratch or use frameworks such as Spring Boot. When we require only standard functionalities, we can use configuration-based ESBs or integration systems such as WSO2 Micro Integrator and Apache Camel. (Refer to [Chapter 3](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch03.html#connectivity_and_composition_pattern) for details on ESBs or integration systems.) Also, when these applications need stateful data processing, event sourcing, or complex event processing, we use stream-processing applications for integration, such as Siddhi and Flink. We discuss these in detail in [Chapter 6](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch06.html#stream_processing_patterns-id00204).

Because event-driven architecture uses topics and queues, we also looked at commonly used technologies such as AMQP, Kafka, and NATS in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns). In this section, let’s dive deep into other message broker technologies that we can use to implement these patterns.

**Apache ActiveMQ**

*Apache ActiveMQ* is the one of the oldest and most widely used open source message brokers that implements AMQP. It provides reliable messaging among the client, broker, and consumer by using queues and event acknowledgments. It also supports OpenWire, STOMP over WebSocket, and MQTT for IoT devices. ActiveMQ natively supports Java Message Service (JMS) for integrating with Java-based applications. This is an ideal system that we can use to implement the Publisher-Consumer pattern, in which we require an exactly once event-processing guarantee and only one consumer should process an event.

ActiveMQ also supports delivering events to multiple subscribers via topics and the reliable delivery of events using durable subscriptions. It supports centralized and peer-based communication to achieve clustering and uses a relational database along with a high-performance journal to provide persistence for the events stored in the broker. But because it cannot scale to high numbers of durable subscriptions, it should not be used for building highly scalable event-driven systems with multiple subscribers.

**RabbitMQ**

*RabbitMQ* is an open source message broker that supports messaging protocols such as AMQP, STOMP, and MQTT. It also supports clustering and failover. RabbitMQ uses *exchanges,* which can be compared to post offices or mailboxes.

When exchanges get events, they distribute event copies to queues by using rules called *bindings*. This provides more flexibility on modeling event distribution. It can also support both push- and pull-based approaches: the broker either delivers messages to consumers via their subscribed queues or allows consumers to pull messages on demand. RabbitMQ is a popular choice for patterns like Priority Queue.

**Amazon SQS**

*Amazon Simple Queue Service* (*SQS*) is a fully managed event-queuing service provided by Amazon, enabling us to decouple and scale microservices and serverless applications. SQS offers two types of event queues. *Standard queues* offer maximum throughput, with best-effort ordering and at-least-once delivery. *SQS FIFO queues* offer event processing in the exact order that events are sent, with an exactly once processing guarantee, although it comes with a throughput constraint.

SQS doesn’t support standard messaging protocols like AMQP, STOMP, and MQTT. But it uses HTTPS to push and pull the events to and from the queue. Since SQS consumers need to poll the SQS queue to receive the events, they are not required to be online when the events are injected into the queue. SQS allows event processing to be retried on failure, but once the maximum number of attempts is reached, it moves the event to a dead-letter queue and removes it from the original SQS queue. This allows consumer services to not block on a corrupted message or waste resources, and enables operations to investigate messages from the dead-letter queue.

**Amazon SNS**

*Amazon Simple Notification Service* (*SNS*) is a fully managed messaging service that supports high-volume fan-out event delivery to all available consumers. It supports event delivery to applications such as Amazon SQS queues, AWS Lambda functions, HTTPS endpoints, and to persons via text messaging, mobile push, and email. It is often used to deliver high-volume real-time notifications following the Publisher-Subscriber pattern.

However, it doesn’t support standard messaging protocols like AMQP, MQTT, and STOMP (similar to Amazon SQS), and the messages are pushed to consumers via configured notification methods such as HTTP/HTTPS, email, and text. Because it provides only at-most-once delivery, SNS is often paired with SQS so consumers have a dedicated event queue for higher-reliability processing.

**Azure Event Grid**

*Azure Event Grid* is a fully managed service that enables event-driven, reactive programming for cloud native applications. It uses a publish-subscribe model by supporting webhooks. Event Grid has built-in support for events coming from Azure services, like storage blobs and resource groups. It allows you to call serverless functions, perform ops automation, build application integration, and integrate with third-party services.

Event Grid isn’t a data pipeline and doesn’t deliver the actual object that was updated, but rather notifies the event occurrence. When delivering the event, it can use filters and route specific events to different endpoints, multicast to multiple endpoints, and make sure the events are delivered with an at-least-once delivery guarantee by spreading its deployment across regions and availability zones.

**Azure Service Bus Queues**

Microsoft *Azure Service Bus* is a fully managed service provided by the Azure messaging infrastructure that can fan out events to multiple consumers through topics as well as support event queues. It can also provide ordered event delivery with FIFO, with at-least-once and at-most-once delivery guarantees.

This can be used to integrate applications or application components that span across multiple communication protocols, data contracts, trust domains, or network environments. It allows consumers to receive events without having to poll the queue, by using a long-polling operation to wait for events to become available.

**Google Cloud Pub/Sub**

*Google Cloud Pub/Sub* is a fully managed asynchronous messaging service provided by Google. It decouples services that produce events from services that process them. Pub/Sub can be used as messaging-oriented middleware or event ingestion and delivery for streaming analytics pipelines.

It requires clients to use HTTPS to send and consume messages, and supports webhooks to push messages to consuming services. It doesn’t support standard messaging protocols like AMQP, MQTT, and STOMP for event delivery. But it provides at-least-once delivery, with durable message storage and high availability. It can also provide around-the-world real-time message delivery with consistent performance at scale.

**Summary of Message Broker Technologies**

This section outlined commonly used message brokers for cloud native application development. [Table 5-4](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#message_broker_technologies) summarizes when we should and should not use these message brokers.

| **Message broker type** | **When to use** | **When not to use** |
| --- | --- | --- |
| Apache ActiveMQ | Need queues or topics. Exactly once processing guarantee. Small- to moderate-scale deployment. Need support for standard messaging protocols. | Highly scalable deployments. High number of durable subscriptions. Replay acknowledged messages. |
| RabbitMQ | Need queues, topics, or to perform event routing. Exactly once processing guarantee. Small- to moderate-scale deployment. Need support for standard messaging protocols. | Highly scalable deployments. High number of durable subscriptions. Replay acknowledged messages. |
| Apache Kafka | Need topics. Highly scalable deployments. High number of durable subscriptions. Need replay of events. Exactly once processing guarantee. Need to acknowledge all messages up to a certain offset. | Need support for standard messaging protocols. Need to selectively acknowledge messages. |
| NATS | Need for topics and queues. Highly scalable deployments. High number of durable subscriptions. Need replay of events. At-least-once delivery guarantee. | Need support for standard messaging protocols. |
| Amazon SQS | Need queues. At-least-once delivery guarantee. Highly scalable deployment. Managed infrastructure by Amazon. Need ordered delivery with FIFO. | Need support for standard messaging protocols. Replay acknowledged messages. Need to fan out the events to multiple consumers. Need a push model. |
| Amazon SNS | Need topics or webhooks. Highly scalable deployment. Managed infrastructure by Amazon. Need to send events to applications and people. No delivery guarantee is required. Call serverless functions. | Need support for standard messaging protocols. Guaranteed delivery. Replay acknowledged messages. |
| Azure Event Grid | Need topics, webhooks. At-least-once delivery guarantee. Managed infrastructure by Azure. Call serverless functions. | Need support for standard messaging protocols. Replay acknowledged messages. |
| Azure Service Bus queues | Need queues or topic At-least-once delivery guarantee. Managed infrastructure by Azure. Need ordered delivery with FIFO. Need to use AMQP 1.0 messaging protocol. Should store less than 80 GB of events. | Need support for MQTT or STOMP. Replay acknowledged messages. |
| Google Cloud Pub/Sub | Need topics, queries, or webhooks At-least-once delivery guarantee. Managed infrastructure by Google. Need consistent performance to delivery events around the world. | Need support for standard messaging protocols. Replay acknowledged messages. |
| Table 5-4. Message broker technologies | | |

**Testing**

In this section, we cover the most important aspects of testing in cloud native applications built on event-driven architecture. Event-driven applications need to follow conventional approaches of writing unit and integration tests. There is no difference in the way unit tests are written (such as the way business logic is isolated via well-defined interfaces and tested without need of dependent applications or network). But integration tests for cloud native event-driven applications require the following additional steps:

1. Ensure that event-transferring infrastructure is available, such as message brokers, and relevant topics or queues exist for processing.
2. Ensure that the tested application is connected with mock clients, such as test publishers to send events and mock consumers to consume the output events for validation.
3. Send input events.
4. Wait until applications finish processing the input.
5. Assert the final state, by calling the mock clients for the responses produced by the application and other systems such as databases, where the relevant updates are performed.

We recommend using a dedicated topic and queues for tests. When possible, start the message broker instance just for the test. Even when a shared messaging infrastructure is used, we recommend creating dedicated topics and queues. When running mock clients and message brokers, we recommend implementing them as containers. When possible, run the test in a namespaced environment. All these features will help isolate the cause of failures and accelerate troubleshooting. This also allows us to clean the test environment after the tests. This reduces interference between tests and other systems and increases the deterministic behavior of the test.

Though these steps seem straightforward, the complexity of integration testing comes with the asynchronous nature of the application. The tests need to wait an arbitrary amount of time for processing to finish for assertion, and during failures the system might not publish events to consumers. Hence, we need to implement the test to wait for a given time-out based on our previous experience before assuming that the test has failed. This time-out-based testing is not generally recommended because of its nondeterministic nature, as network delays and slower hardware can cause the test to fail intermittently. Unfortunately, this is something that we cannot eliminate in integration testing of event-driven applications, but we can try to reduce the impact by building the application to produce output for both success and failure cases, and by providing a way to query the application state for assertions.

In addition, because of the asynchronous nature of event processing, we cannot guarantee that the events are consumed, processed, and output in the same order that they are published by the client. For example, when we send events A, B, and C in order, the events can be outputted in B, C, A order after processing. Based on the use case we test, unless the events are commutative, writing test assertions expecting the output events to be consumed in A, B, and C order will fail the test case. In this case, use unique IDs for each event so we can uniquely identify the output and assert them.

Event-driven applications should also be tested for failure use cases, using chaos engineering. This can include simulating network failures, simulating slow producers and consumers, and bringing down consumers and message brokers for a brief period of time. This allows you to identify failures, improve the applications to have predictable behavior during failures, and enable smooth recovery.

**Security**

How can we enforce security for applications and systems in an event-driven architecture? Applications should enforce security by connecting to systems by using only secured protocols and encrypting data at rest and in transit.

Message brokers also support security by protecting queues and topics behind authentication and authorization mechanisms. But as the events are stored in the brokers, we need to ensure they are stored safely and not persisted longer than is necessary.

When events are transmitted through topics, we need to ensure that only authorized applications are able to consume the events. We use topic subscriptions for observability and monitoring purposes, but the same approach can also be used to eavesdrop on the events.

Not all types of message brokers and microservices used in cloud native applications can always provide the required level of security. We recommend using a bounded context that is fronted by an API or a secured message broker to consume the events from external systems and build the whole asynchronous architecture within that context. By always encrypting events before sending them to brokers, we can also make sure that the events are stored in their queues and topics encrypted, and applications eavesdropping on those events will not be able to decrypt them to access the data.

In addition to what we’ve discussed here, we recommend that you apply the general security best practices discussed in [Chapter 2](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch02.html#communication_patterns).

**Observability and Monitoring**

Observability and monitoring play a key role in the success of event-driven architectures. Event-driven applications with scale can soon become complex, making it difficult to even understand how an event flows through the components of the application. Without proper observability and monitoring, understanding this behavior and troubleshooting large-scale event-driven applications cannot be a reality. Furthermore, as event-driven cloud native applications can use serverless computing frameworks and process events asynchronously, clients are not usually notified of event-processing failures, so we can troubleshoot these systems only by using proper observability and monitoring tools.

Event-driven architecture forms chains of event-processing microservices that are connected via topics and queues. Failures can happen because of a bad event or a networking issue at any stage of the execution chain. This can result in the event getting dropped, and stopping it from propagating through the whole execution chain. For example, when a loan-processing request is initiated, and the event is dropped by an application error while performing a credit check, the customer or the bank cannot know the status of the request unless the organization has built proper observability around the application. This observability helps indicate that an error has occurred and where it has occurred. This helps build mechanisms to overcome the consequences of the error and recover the processing flow.

Distributed tracing applications such as Jaeger and Zipkin help us properly observe event-driven applications. First we assign a causation ID for the event; in this case, the loan request. Then as the request is processed by subsequent applications, and they produce further events, the causation ID is persisted. This causation ID propagates through the whole request flow. Having a causation ID allows us to pinpoint our current location in the flow. The distributed tracing systems enable us to visualize workflows by their causation ID, allowing us to quickly identify errors.

Tracing provides information only about where the error has happened; we need to use logging to find the root cause of the error. You should log events and errors with the causation ID at each participating microservice, and aggregate them by using log aggregation systems such as Fluentd, Logstash, Amazon CloudWatch, or Google Cloud Operations. This will help find the reasons behind those errors so you can mitigate them.

We should also continuously monitor the microservices and message brokers used in the cloud native application. This is critical to identify slow-performing microservices and to detect bottlenecks in the event-processing chain. As events are processed asynchronously, if the incoming rate of events remains higher than the consumption rate over a prolonged period, the events can excessively accumulate at the message brokers and cause the event-driven application to fail. This can be detected by observing high queue depth, and can be eliminated by scaling consumers, dropping excess events, or improving the performance of event consumers.

**DevOps**

In this chapter, we have discussed several event-driven architecture patterns that can be applied in cloud native applications. Because message brokers are a key component of event-driven architecture, here we focus on the DevOps process of message brokers.

The first step in deploying and managing message brokers is selecting the appropriate orchestration and delivery patterns required. This allows us to determine what type of event-delivery guarantees are needed. Based on these and the required scalability, we choose a message broker or proceed without one.

Often because of cost concerns, organizations choose a single message broker for all use cases. In such situations, we should evaluate how a delivery pattern is best implemented on the available message broker and whether that fits our requirements.

The next major step in deploying and managing message brokers is enforcing the security of the applications, protecting access to message brokers, and allowing only authorized applications to publish and consume events. See [“Security”](https://learning.oreilly.com/library/view/design-patterns-for/9781492090700/ch05.html#security-id00267) for further details.

Because failures in asynchronous applications are difficult to troubleshoot, robust observability and monitoring is critical. As discussed earlier in this chapter, we recommend that you implement distributed tracing, logging, and monitoring. As systems encounter network failures and application errors, unprocessed events end up in dead-letter queues when available. These queues should be monitored to allow for the correcting of failed events.

Autoscaling the event-driven application is critical, as without this, message brokers can become overloaded and degrade application performance or even cause failures. When deploying applications in Kubernetes, we can use the Kubernetes-based Event Driven Autoscaler ([KEDA](https://keda.sh/)) to monitor message broker queue depth, and autoscale the consumer microservices so they can process all incoming events without increasing the backlog. KEDA supports scaling based on various message brokers and stream processors such as Kafka, NATS, RabbitMQ, Azure Event Hubs, and Amazon SQS.

To achieve continuous delivery and smooth deployments, we recommend maintaining backward compatibility of the event schema. When major changes occur in the event schema and the reuse of topics and queues is not possible, we recommend migrating the applications to new topics or queues in stages by using canary or blue-green deployment strategies. Make sure that all events in the queues and durable topics of the previous application version are successfully processed before finishing the version upgrade. Finally, we also recommend using multiple deployment environments, such as development and staging/preproduction, to reduce the impact of the changes and to validate the event-driven application before moving it to production.

By following these steps, we can safely deploy and maintain cloud native applications and the respective message brokers used in the event-driven architecture while allowing rapid innovation and adoption to other systems.

**Summary**

In this chapter, we looked at delivery patterns applied to cloud native applications with event-driven architectures. We explored options for achieving asynchronous communications both with and without message brokers. We reviewed message broker types and the various event-delivery guarantees.

We then discussed how to use various patterns to deliver events from one application to another, how to manage application state by using event sourcing, and how events can be orchestrated for event processing. We discussed the complexity of building scalable cloud native event-driven applications, and how they can be managed. We also reviewed robust message broker technologies and discussed how event-driven applications can be secured, tested, continuously deployed, and observed and monitored. Next, we will explore the patterns related to cloud native stream-processing applications.