Chapter 15. Event-Driven Architecture Style

The *event-driven* architecture (EDA) style is a popular distributed, asynchronous architecture style used to produce highly scalable and high-performance applications. It is also particularly adaptable and can be used for small applications as well as large, complex ones. Event-driven architecture is made up of decoupled event processing components that asynchronously trigger and respond to events. It can be used as a standalone architecture style or embedded within other architecture styles (such as an event-driven microservices architecture).

Many developers and software architects consider EDA more of an architectural pattern than an architectural style. We disagree. Your authors have developed entire systems that rely solely on EDA, which is why we maintain that it's primarily an architectural style. While EDA can be used in other architectural styles—such as microservices and space-based architecture—to form hybrid architectures, it remains at its core a way of designing complex systems.

Most applications follow what is called a *request-based* model, as illustrated in Figure 15-1. When a customer requests, for example, their order history for the past six months, this request is initially received by a *request orchestrator*. The request orchestrator is typically a user interface, but it can also be implemented through an API layer, orchestration services, event hubs, an event bus, or an integration hub. Its role is to direct the request to various *request processors*, deterministically and synchronously. They process the request by retrieving and updating the customer's information in a database. Retrieving order history information is a data-driven, deterministic request made to the system within a specific context, not an event happening that the system must react to, which is why this is a request-based model.

An *event-based* model, on the other hand, reacts to a particular event by taking action. For example, take submitting an online auction bid for a particular item. The bidder submitting the bid is not making a request to the system, but rather initiating an event that happens after the current asking price is announced. The system must respond to that event by comparing the bid to others received at the same time and determining the current highest bidder.

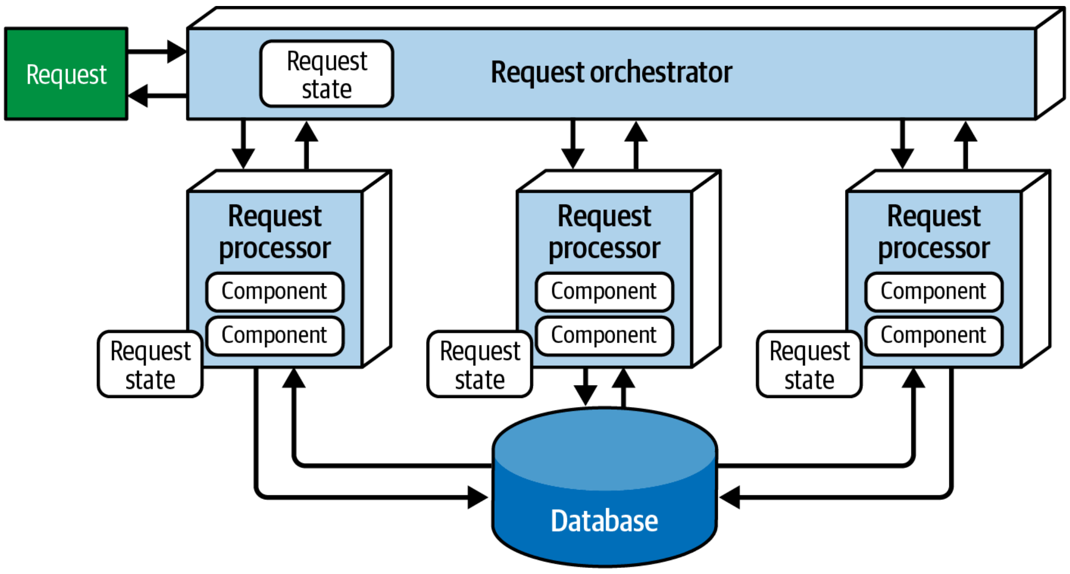


Figure 15-1. Request-based model

Topology

Event-driven architecture leverages asynchronous *fire-and-forget* communication, where services trigger events and other services respond to those events. The four primary architectural components of its topology are an *initiating event*, an *event broker*, an *event processor* (usually just called a *service*), and a *derived event*.

The *initiating event* is the event that starts the entire event flow. This could be a simple event, like placing a bid in an online auction, or more complex events, like making updates to a health-benefits system when an employee gets married. The initiating event is sent to an event channel in the *event broker* for processing. A single *event processor* accepts the initiating event from the event broker and begins processing that event.

The event processor that accepted the initiating event performs a specific task associated with processing that event (such as placing a bid for an auction item), then asynchronously advertises what it did to the rest of the system by triggering what is called a *derived event* to an *event broker*. Other event processors respond to the derived event, perform specific processing based on it, then advertise what they did through new derived events. This process continues until all event processors are idle and all derived events have been processed. Figure 15-2 illustrates this event-processing flow.

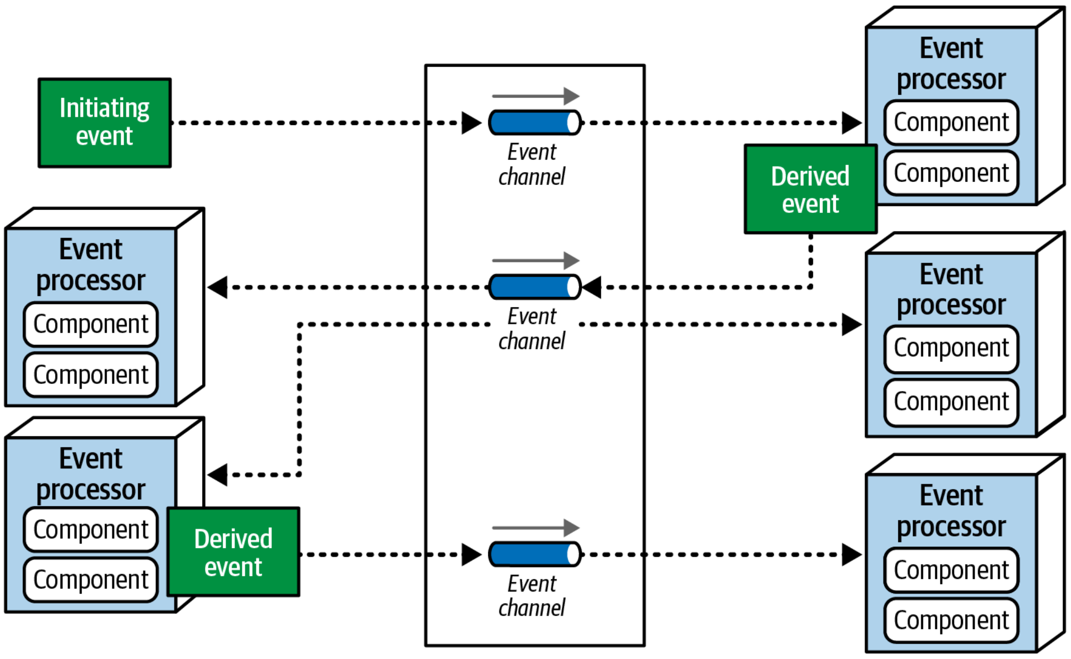


Figure 15-2. Basic topology of an event-driven architecture

The event-broker component is usually *federated* (meaning it has multiple domain-based clustered instances). Each federated broker contains all of the *event channels* (such as queues and topics) used within the *event flow* (the entire workflow for processing the event) for that particular domain. Because of the decoupled, asynchronous, fire-and-forget broadcast nature of this architectural style, the broker topology uses topics, topic exchanges (in the case of the Advanced Message Queuing Protocol (AMQP)), or streams with a publish-and-subscribe messaging model.

To illustrate how EDA processing works overall, consider the workflow of a typical retail-order entry system, as illustrated in Figure 15-3, where customers can place orders for items (say, a book like this one). In this example, the `Order Placement` event processor receives the initiating event (`place order`), inserts the order in a database table, and returns an order ID to the customer. It then advertises to the rest of the system that it created an order through an `order placed` derived event. Notice that three event processors are interested in that derived event: the `Notification` event processor, the `Payment` event processor, and the `Inventory` event processor, all of which perform their tasks in parallel.

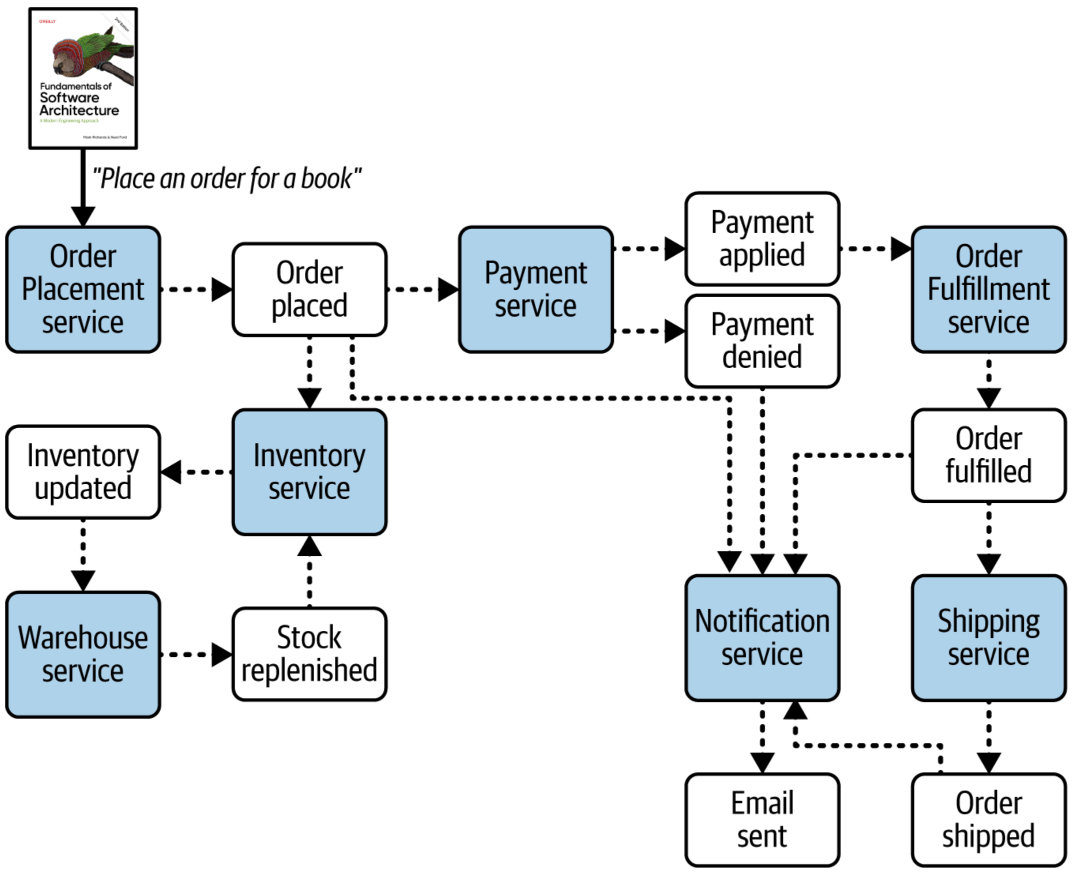


Figure 15-3. Example of the event-driven architecture topology

The `Notification` event processor receives the `order placed` derived event and emails the customer with the order details. Because the `Notification` event processor has performed an action, it then generates an `email sent` derived event. However, notice in Figure 15-3 that no other event processors are listening to that derived event. This is typical within EDA and illustrates this style's *architectural extensibility*—the ability for future event processors to respond to the derived event without any changes to the existing system (see "Triggering Extensible Events").

The `Inventory` event processor also listens for the `order placed` derived event and adjusts the corresponding inventory for that book. It then advertises this action by triggering an `inventory updated` derived event, which in turn triggers a response from the `Warehouse` event processor. This processor responds to and manages the corresponding inventory between warehouses, reordering items if supplies get too low. When stock is replenished, the `Warehouse` event processor triggers a `stock replenished` derived event, to which the `Inventory` event processor responds by adjusting the current inventory.

In this case, the `Inventory` event processor would not trigger a corresponding `inventory adjusted` event. If it did, that would lead to what is called a *poison event*—an event that keeps looping and looping forever.

Warning

A *poison event* occurs when a derived event keeps getting triggered and responded to in a continuous loop between services. These can happen frequently when using an event-driven architecture, so be careful to avoid them.

The `Payment` event processor also responds to the `order placed` derived event. It responds by charging the customer's credit card. Figure 15-3 shows that one of two possible derived events will be generated as a result of the `Payment` event processor's action: one to notify the rest of the system that the payment was applied (`payment applied`), and one to notify the system that the payment was denied (`payment denied`). The `Notification` event processor is interested in the `payment denied` derived event, because if this happens, it needs to email the customer informing them that they must update their credit card information or choose a different payment method.

The `Order Fulfillment` event processor listens for the `payment applied` derived event and performs various automated functions within order picking and packing, such as instructing the worker where to find the item and what size box is needed for the order. Once that's completed, it triggers an `order fulfilled` derived event telling the rest of the system that it has fulfilled the order. Both the `Notification` and the `Shipping` event processors listen for this derived event. Concurrently, the `Notification` event processor notifies the customer that the order has been fulfilled and is ready for shipment, and the `Shipping` event processor selects a shipping method, ships the order, and sends out an `order shipped` derived event. The `Notification` event processor also listens for the `order shipped` derived event and notifies the customer that the order is on its way.

All of the event processors are highly decoupled and independent of each other. One way to understand this asynchronous processing workflow is to think about it as a relay race. In a relay race, runners hold a baton (a wooden stick) and run for a certain distance (say, 1.5 kilometers), then hand off the baton to the next runner, who does the same, on down the chain until the last runner crosses the finish line. Once a runner hands off the baton, that runner is done with the race and can move on to other things. This is also true with event processors: once an event processor hands off an event, it is no longer involved with processing that specific event and is available to react to other initiating or derived events. In addition, each event processor can scale independently to handle varying load conditions or backups.

Style Specifics

The following sections describe EDA in more detail, including some considerations, patterns, and hybrids of this complex architectural style.

Events Versus Messages

Event-driven architecture leverages events to pass and process information. But is an *event* really that different from a *message*? Turns out, yes, it really is.

An *event* broadcasts to other event processors that something has already happened: "I just placed an order." A *message*, however, is more of a command or query, such as "apply the payment for this order" or "give the shipping options for this order." This is not a subtle difference. When we talk about *event processing*, we mean *reacting* to something that has already happened, whereas a message describes something that *needs to be done*. In our example, it's clear that "I just placed an order" is an event because it does not describe what processing needs to occur, as a message would. This illustrates EDA's decoupled nature.

The second main difference between an event and a message is that an event typically does not require a response from the receiver, whereas a message usually does. This reduces back-and-forth communication between event processors, further decoupling them from one another.

Another major difference between events and messages is that an event is typically broadcast to multiple event processors, whereas a message is almost always directed to one event processor. In our simple order-processing example, several event processors are interested in and respond to the `order created` event, whereas only one event processor responds to the message `apply payment`. Events typically use a *publish and subscribe* (one-to-many) form of communication, whereas messages typically use a *point-to-point* (one-to-one) form of communication.

A final difference between events and messages is the physical artifact that represents the communication channel. Events use a *topic*, *stream*, or notification service so that multiple event processors can subscribe to the channel and listen for the event. Messaging typically uses a *queue* or *messaging service* to guarantee that only one type of event processor will receive that message.

Event-driven architecture mostly uses *events* (hence its name), but it can use messages on occasion, such as requesting data from another event processor (see "Data Topologies" and "Request-Reply Processing"). Later in this chapter, we'll show you *mediated event-driven architecture* (see "Mediated Event-Driven Architecture"), which uses messages to control the processing order of events.

Can you pick out which of the following choices are events and which are messages?

- "Adventurous Air Flight 6557, turn left, heading 230 degrees."
- "In other news, a cold front has moved into the area."
- "OK, class, turn to page 145 in your workbooks."
- "Hi, everyone! Sorry I'm late for the meeting."

Let's look at these one by one:

"Adventurous Air Flight 6557, turn left, heading 230 degrees."

> This is a *message* because it's a command (something that needs to be done) and because it's directed toward one target, the pilot, even though several other pilots may hear the message.

"In other news, a cold front has moved into the area."

> This is an *event*: it's being broadcast to multiple people, it describes something that has already happened, and the news broadcaster is not expecting a reply. (There are, however, some messages that don't require a reply either.)

"OK, class, turn to page 145 in your workbooks."

> This one is a little tricky. Turns out this is a *message*, even though it's being broadcast to multiple students. It's a *command* to do something, not something that has already happened (which would make it an event). This illustrates an important point about the difference between an event and a message: broadcasting a command (such as turning to page 145 in the workbooks) through a publish-and-subscribe channel doesn't turn it into an *event*.

"Hi, everyone! Sorry I'm late for the meeting."

This is an *event*, because this person being late for the meeting has already happened. It's also being broadcast to multiple people, and no response is expected.

Derived Events

Derived events are a critical and necessary part of EDA. They are created and triggered by event processors *after* the initiating event is received. An event processor can trigger more than one derived event, based on its processing.

Consider the derived events triggered when the `Payment` event processor charges a customer's credit card for a purchase in Figure 15-3. As illustrated in Figure 15-4, charging a credit card involves checking for possible fraud (processed by the `Fraud Detection` event processor) and checking the credit card balance (processed by the `Credit Limit` event processor). EDA can leverage a single event (`creditcard charged`) to do both activities at the same time.

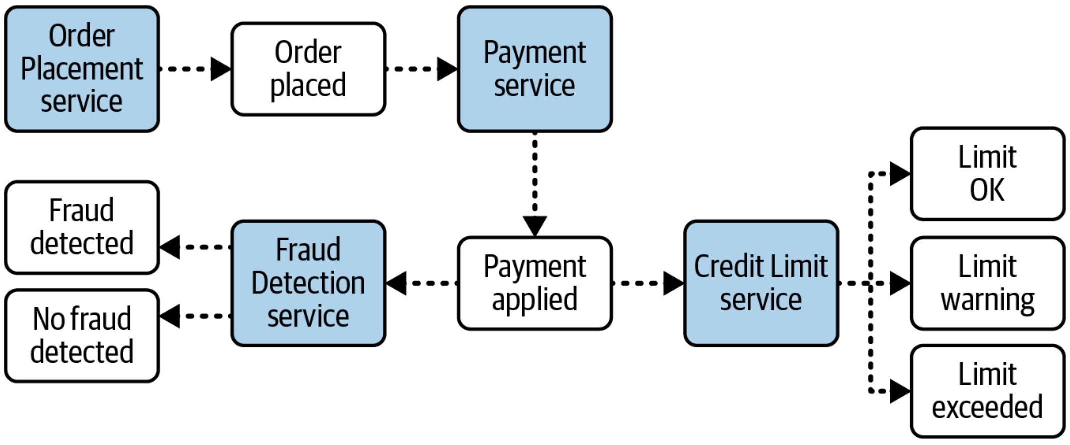


Figure 15-4. Derived events are generated in response to the initiating event

Notice how many derived events this single action generates. In the `Fraud Detection` event processor, it triggers two possible derived events: one if the processor detects fraud, another if no fraud is detected. Both of these derived events are necessary, because different event processors might take further actions depending on the outcome of the fraud detection.

Now look at the derived events from the `Credit Limit` event processor. First, a `limit okay` derived event indicates to the rest of the system there is no risk for this purchase and that the customer has plenty of credit available. As a matter of fact, this particular event could also store in its event payload the amount of credit the customer has left, which might be of interest to downstream event processors. Second, the `limit warning` derived event, warning that the card's balance is close to the credit limit, might be of interest to other downstream event processors—for instance, the `Notification` event processor, which could notify the customer that they are close to their credit limit. Last, the fatal `limit exceeded` derived event is of interest to several event processors, including `Notification`, `Decline Purchase`, and perhaps a marketing-based `Extend Credit Limit` event processor that automatically extends the customer's credit limit to allow the purchase.

This illustrates that more than one derived event can be triggered from an event processor. However, be careful not to get trapped in the *Swarm of Gnats* antipattern, where a processing unit sends out too many fine-grained events (see "The Swarm of Gnats Antipattern").

Triggering Extensible Events

In EDA, it's usually a good practice for each event processor to advertise what it has done to the rest of the system, regardless of whether or not any other event processor cares about what that action was.

When no event processors care about or respond to the event, we call it an *extensible derived event* because it nevertheless provides support for *architectural extensibility* by providing a built-in "hook" in case processing that event requires additional functionality. For example, suppose that, as part of a complex event process (as illustrated in Figure 15-5), the `Notification` event processor generates an email that it sends to a customer, notifying them of a particular action. It then advertises that it has sent the email to the rest of the system through a new derived event (`email sent`). Since no other event processors are currently listening or responding to that event, the message simply disappears (or is ignored, in the case of event streaming). That might seem like a waste of resources, but it's not. Suppose the business decides to analyze all emails sent to customers. The team can add a new `Email Analyzer` event processor to the overall system with minimal effort and with no changes to other event processors, because the email information is already available via the `email sent` derived event.

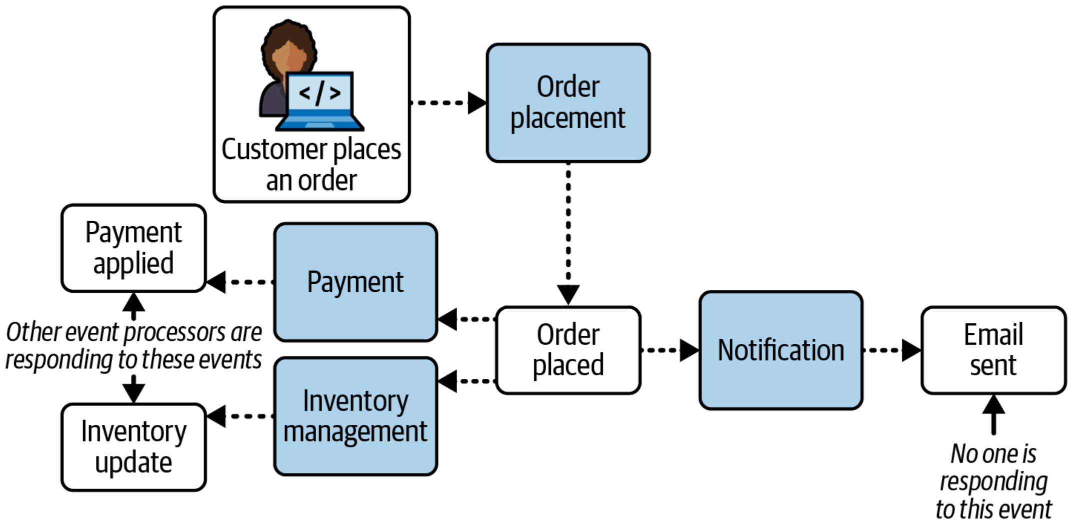


Figure 15-5. `Notification` event is sent, but ignored and not used

Asynchronous Capabilities

The event-driven architectural style has a unique characteristic in that it relies primarily on asynchronous communication for both *fire-and-forget* processing (no response required) and *request/reply* processing (when a response is required from the event consumer, see "Request-Reply Processing"). Asynchronous communication can be a powerful technique for increasing a system's overall responsiveness.

In Figure 15-6, a user is posting a product review on a website. The comment service, in this example. takes 3,000 milliseconds to validate and post that comment. The comment has to go through several parsing engines: a check for unacceptable words, a check to make sure that the comment is not indicating abusive text (such as "slow thinker" or "unable to think clearly"), and finally a context check to make sure the comment is about the product (and not, say, just a political rant).

The top path in Figure 15-6 posts the comment using a synchronous RESTful call. That means 50 ms in network latency for the service to receive the post, 3,000 ms to validate and post the comment, and 50 ms of latency to tell the user that the comment was posted. The total time to post a comment, from the user's point of view, is thus 3,100 milliseconds. Now look at the bottom path, which uses asynchronous messaging. Here, the user's total posting time is only 25 ms, as opposed to 3,100 ms. It still takes the system 25 ms to receive the comment and 3,000 ms to post it, for a total of 3,025 ms, but from the end user's perspective, only 25 ms go by before the system responds that it has accepted the comment (although it hasn't actually been posted yet).

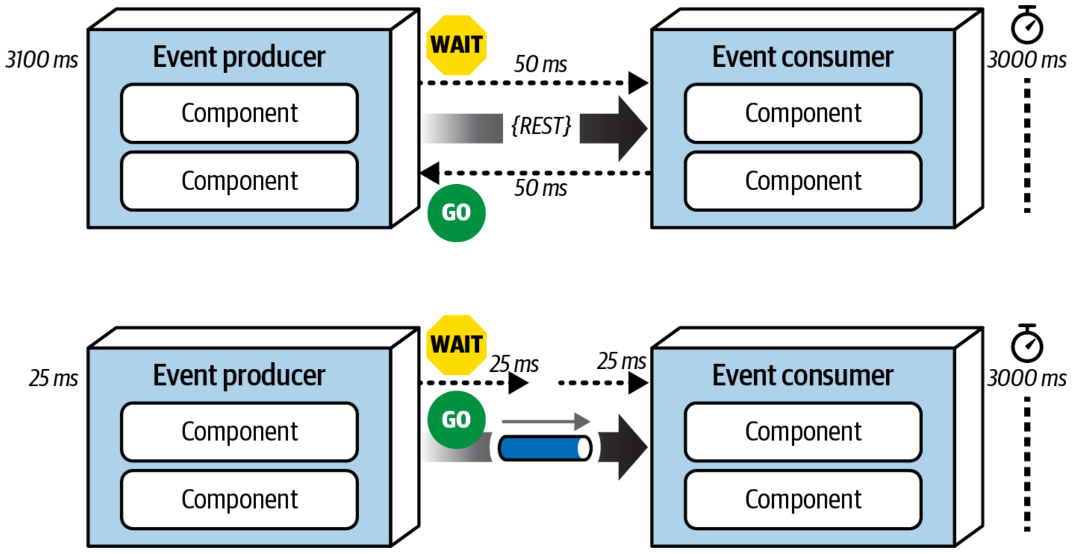


Figure 15-6. Synchronous versus asynchronous communication

The difference in response time between 3,100 ms and 25 ms is staggering. There is, however, one caveat: on the synchronous top path, the end user gets a *guarantee* that their comment has been posted. However, on the asynchronous bottom path, the post is only acknowledged, with a future promise that *eventually* it will be posted. What would happen if the user's comment includes profanity and the system rejects it? There is no way for it to notify the end user—or is there? If the user must register with the website to post a comment, the system could send them a message indicating a problem with the comment and suggesting how to repair it.

This example illustrates the difference between *responsiveness* (the time it takes to get information back to the user) and *performance* (the time it takes to insert the comment into the database). When the user doesn't need any information back (other than an acknowledgment or a thank-you message), why make them wait? Responsiveness is all about notifying the user that the action has been accepted and will be processed momentarily, whereas performance is about making the end-to-end process faster. On the asynchronous bottom path, the architect did nothing to optimize the way the comment service processes the comment (that's addressing *responsiveness*). If the architect took the time to optimize the comment service, perhaps by executing all of the text and grammar parsing engines in parallel, leveraging caching and other similar techniques but still using synchronous communication, they would be instead addressing overall *performance*.

That was a simple example. What about a more complicated one? This time, let's look at an online *stock trade*, where a user is purchasing some stock asynchronously. What if there is no way to notify the user of an error?

While asynchronous communication significantly improves responsiveness, error handling is a big issue. The difficulty of addressing error conditions adds to the complexity of this architecture style. "Error Handling" illustrates a pattern of reactive architecture for addressing error-handling challenges: the *Workflow Event* pattern.

In addition to its responsiveness, asynchronous communication also provides a good level of dynamic decoupling and avoids an antipattern, *Dynamic Quantum Entanglement*, which occurs when two architectural quanta communicate through synchronous communication. (You probably recall from Chapter 7 that an *architectural quantum* is a part of the system that can be deployed independently from the rest of the system and is bound through synchronous dynamic coupling, and that architectural characteristics live at the quantum level.) Because these two architectural quanta are now dependent on each other, they essentially become *entangled*. The dependency turns them into a single architectural quantum. Asynchronous communication can help detangle architectural quanta because it removes that dynamic dependency.

To illustrate this important point, consider the two systems in Figure 15-7. In this example, the `Portfolio Management` system creates a trade order to buy some stock. It synchronously sends this trade order to the `Trade Order` system, which will perform compliance checks and create the trade order. Because the communication between these two systems is synchronous, the `Portfolio Management` system must necessarily block and wait for a trade confirmation number from the `Trade Order` system. These two systems have become entangled and now form a single architectural quantum.

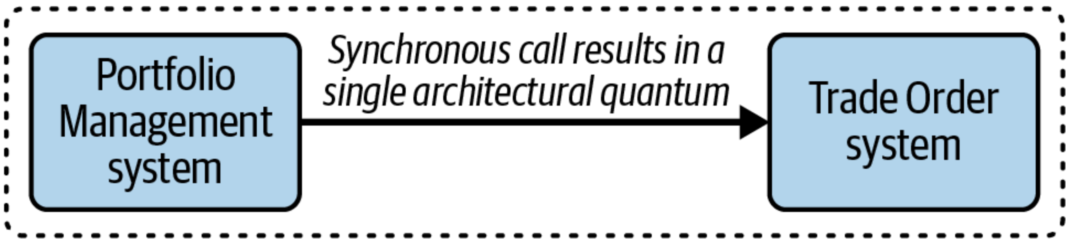

Portfolio Management system
Synchronous call results in a single architectural quantum
Trade Order system

Figure 15-7. These systems form a single architectural quantum due to synchronous dynamic coupling

The significance of this entanglement is that the architectural characteristics now live *between* these two systems. If the `Trade Order` system becomes unavailable or unresponsive, the `Portfolio Management` system cannot submit the trade order. This also degrades responsiveness: if the `Trade Order` system is slow, the `Portfolio Management` system will be slow too. Scalability also suffers because if the `Portfolio Management` system needs to scale, so does the `Trade Order` system. If it doesn't or can't, then the `Portfolio Management` system can't scale as needed.

An architect can detangle these architectural quanta by replacing the synchronous call with an asynchronous call between the two systems, as illustrated in Figure 15-8.

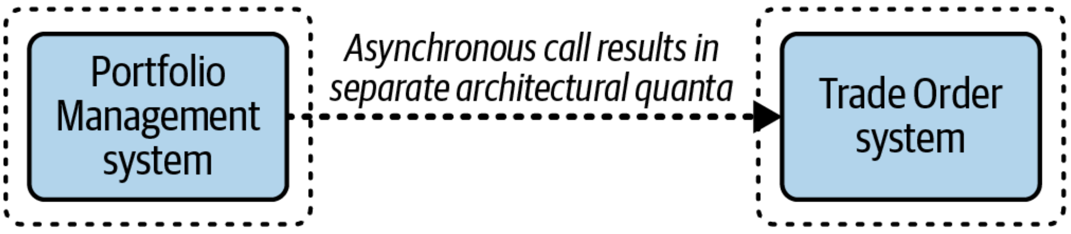

Portfolio Management system
Asynchronous call results in separate architectural quanta
Trade Order system

Figure 15-8. These systems form separate architectural quanta due to their asynchronous dynamic coupling

By using asynchronous communication, the `Portfolio Management` system can send the trade order through a queue or some other asynchronous means, so it doesn't have to wait for the `Trade Order` system to create the trade order. Once the `Trade Order` system performs its compliance checks and creates the trade order, it can send the confirmation number to the `Portfolio Management` system through a separate, asynchronous channel. Removing this dependency from the two systems' dynamic coupling allows them to form two separate architectural quanta. If the `Trade Order` system is unavailable or unresponsive, the `Portfolio Management` system can still issue trade orders, knowing that at some point they will be created and confirmation numbers sent back.

Broadcast Capabilities

Another of EDA's unique characteristics is its ability to broadcast events without knowing what other processing units (if any) are receiving those events or what processing they will perform in response. As Figure 15-9 shows, this dynamically decouples the event processors from each other.

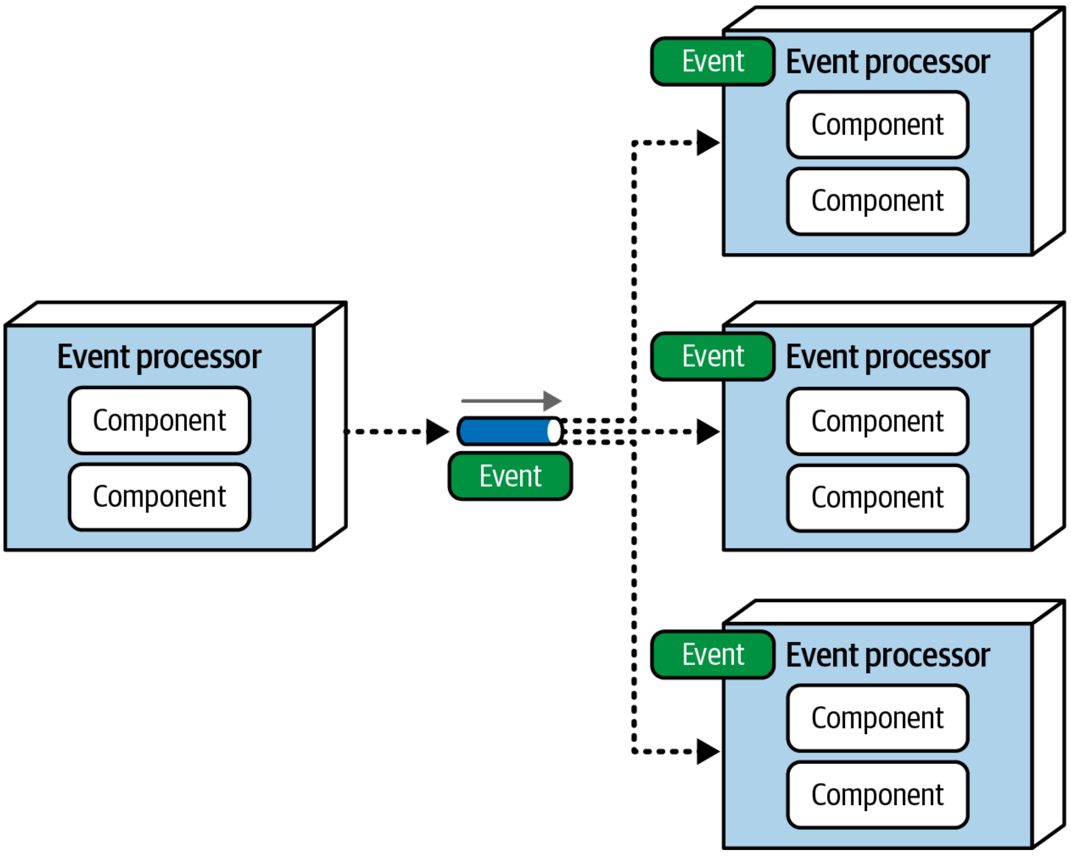


Figure 15-9. Broadcasting events to other event processors

Broadcast capabilities are an essential part of many patterns, including eventual consistency and complex event processing (CEP). For example, prices for instruments traded on the stock market change frequently. Every time a new ticker price (the current price of a particular stock) is published, many event processors might respond to the new price (such as trade analytics, or buying or selling the stock). However, the event processor that publishes the latest price simply broadcasts it, with no knowledge of how that information will be used. This is known as *semantic decoupling* in that one event processor has no knowledge of (or dependency on) the actions of other event processors.

Event Payload

The information contained in an event is known as its *payload*. Payloads can vary significantly: an event payload could be a simple key-value pair, or all the information necessary for downstream processing. The two basic types are *data-based* and a *key-based* event payloads. Architects must do careful trade-off analysis to determine which of these options suits each type of event triggered in the system. In this section, we describe these two payload types and their corresponding trade-offs.

Data-based event payloads

A *data-based event payload* is an event payload that sends all necessary information for processing. In the example illustrated in Figure 15-10, a customer places an order. First, the `Order Placement` event processor inserts the complete order into the database (the system of record). It then broadcasts an event called `order_placed`, which contains all of the order details (in this case, 45 attributes, totalling 500 KB of memory). The `Payment` event processor responds to this event by pulling information from the event payload—specifically the order ID, the customer's information, and the total cost of the order—and using it to apply the payment. Simultaneously, the `Inventory Management` event processor uses the item ID and item quantity from the event payload to adjust the current inventory for the item purchased.

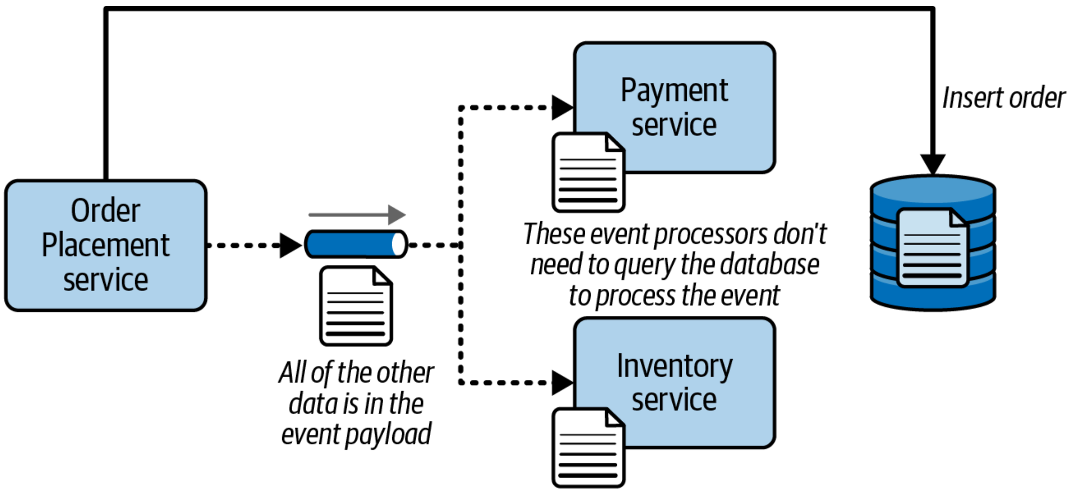

Order Placement service
Payment service
Inventory service
Insert order
These event processors don't need to query the database to process the event
All of the other data is in the event payload

Figure 15-10. Data-based event payloads contain all the data necessary for processing

The `Payment` and `Inventory Management` event processors didn't have to query the database to get the order information, because the data was already contained in the event payload. This is one of the biggest advantages of using data-based event payloads. The less an event processor queries the database, the better its performance, responsiveness, and scalability will be. Furthermore, given the highly dynamic decoupled nature of EDA, the `Order Placement` event processor might not know which other event processors are responding to the event or what data they might need for processing. Sending all of the information in the event payload guarantees that each responding event processor will have the information it needs to perform its processing. Event processors might not even have access to the database that contains the order information, particularly in data topologies with strict bounded contexts, domain based or database-per-service (see "Data Topologies").

While these advantages demonstrate ways of creating more responsive, scalable, and flexible systems, data-based payloads do have several disadvantages. The first is that it's harder to maintain data consistency and data integrity when you have *multiple systems of record*. Because all of the order information is contained in the database *and* in the events being triggered in the system, the order information can easily get out of sync—particularly if the order is updated during processing.

For example, suppose a customer places an order for a hundred items, but only meant to order one, and realizes their mistake immediately after submitting the order. Or perhaps the customer realizes just after ordering that they've used the incorrect shipping address (this has happened to your authors many times). In either of these situations, the customer immediately updates the order with the correct information. The database, which is the single system of record, contains the corrected values, but some of the events containing the older values might not be processed immediately. This means that any of the older, incorrect values still being processed will be used instead of the newer, correct ones. To further complicate this scenario, it is very difficult in EDA to control the timing of events, so it's possible that the newer values might be processed *before* the older values. This means that if other event processors use the older, incorrect values, those could be overlaid on the correct, newer values.

A second major disadvantage of the data-based event payload is about contract management and versioning. We know that an order in this system has 45 attributes. Because all of that information is contained in the event payload, the event needs some sort of *contract*—a way of structuring the data being sent. The architect is now faced with myriad decisions: should the payload type be a JSON object? An XML object? Should the contract be strict or loose? (A *strict* contract is one that uses some sort of schema or object definition, such as a JSON schema, GraphQL spec, or class definition; whereas a *loose* contract might use simple JSON name-value pairs.) Each of these decisions carries with it many trade-offs, and each forms a tight static coupling between event processors.

And then there's versioning. For strict contracts, an architect or developer might use a vendor MIME type in the event headers to specify the version number. This helps make the system more agile and provides backward compatibility (to avoid breaking other event processors). However, all event processors must leverage the same versioning logic, which requires strong governance. If an event processor ignores a contract version when responding to a strict contract's payload, changing that schema is likely to cause that event processor to fail. In addition, it's very difficult to implement version communication and deprecation strategies in a highly decoupled, asynchronous architecture like EDA. All this makes data-based event payloads somewhat fragile.

Data-based event payloads can also suffer from *stamp coupling*: a form of static coupling where several modules (in this case, event processors) all share a common data structure, but only use parts of it (and in many cases different parts of it). When this situation occurs, changing the common data structure can require changing other event processors, even ones that don't care about the data.

Figure 15-11 illustrates how stamp coupling works and its negative impact on the architecture. In this example, the `Order Placement` event processor sends an `order_placed` event consisting of 45 attributes, containing all the information about the order, at a size of 500 KB. The `Inventory` event processor responds to the `order_created` event, but only requires two attributes, the `item_id` and the `quantity`, totaling only 30 bytes. In this example, changing to the payload, for example by removing an address-line attribute, would affect the `Inventory` event processor even though it doesn't care about that field.

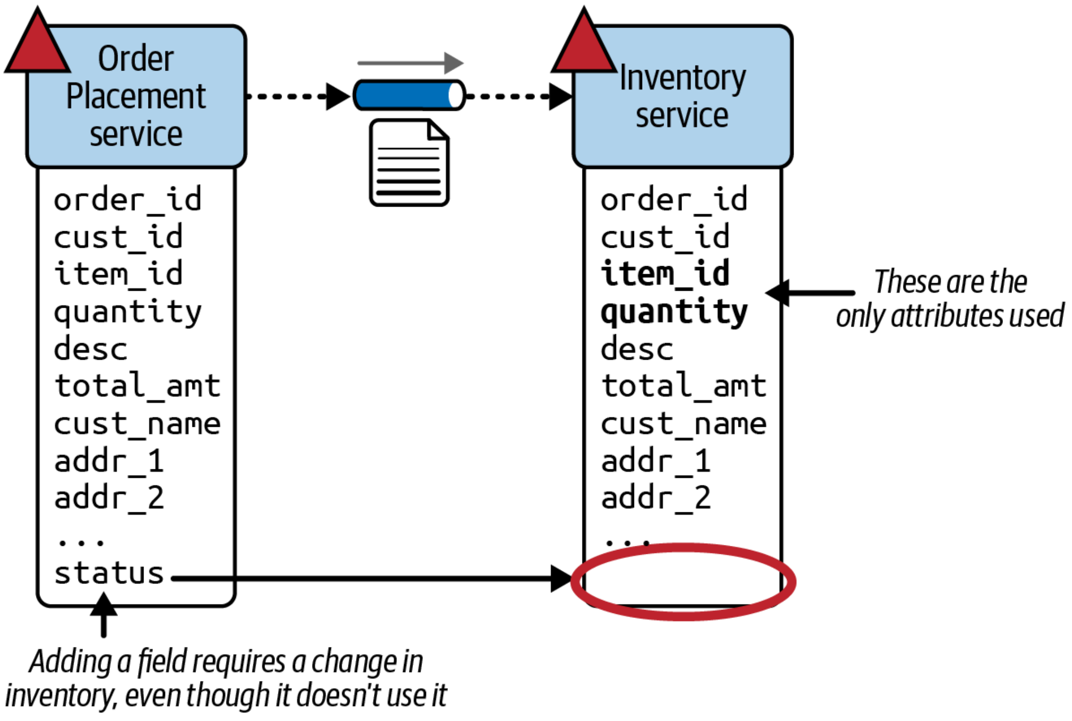


Figure 15-11. An example of stamp coupling, where another service only needs part of the data sent

In this example, using contract versioning with strict contracts helps mitigate the risk of the `Inventory` event processor breaking, but at some point—when the contract version is deprecated or a breaking contract change occurs—a developer will have to retest and redeploy it.

One commonly overlooked problem with stamp coupling is *bandwidth utilization*. The third fallacy of distributed computing is that "bandwidth is infinite." It's not, of course. As a matter of fact, in most cloud-based environments, bandwidth is what costs so much. Going back to our example in Figure 15-11, with a data-based payload, if customers are placing 500 orders per second, sending a single 500-KB event to the `Inventory` event processor will utilize 250,000 KB per second of bandwidth. However, if the system sends only the 30 bytes of data actually *needed*, the event will only utilize 15 KB per second of bandwidth. This is a staggering difference, one worth investigating when using data-based event payloads.

One reason architects sometimes limit stamp coupling is to leverage consumer-driven contracts, where each consumer of a message has its own contract containing only the data it needs for processing. However, because of EDA's broadcast capabilities and because the system can't always know which event processors will respond to an event, it's difficult to leverage consumer-driven contracts with events in event-driven architecture. For this reason (and to address the other disadvantages of data-based event payloads), many architects turn to key-based event payloads.

Key-based event payload

A *key-based event payload* is an event payload that contains only a key identifying the context for the event (such as an order ID or customer ID). With key-based event payloads, the event processors responding to the event must query a database to retrieve the information they need to process the event.

When a customer places an order, the `Order Placement` event processor inserts the order into the database and triggers a key-based event called `order_placed`. This event contains a single key value containing the order ID in simple JSON:

```
{
 "order_id": "123"
}
```

One of the main disadvantages of key-based event payloads is that each event processor responding to the event must query the database to get the information it needs to process the order. For instance, when the `Payment` event processor responds to the event, it must query the database for the order information it needs to process payment. The `Inventory` event processor also responds simultaneously to the event, so it must also query the database to retrieve the item ID and quantity. This can detract from responsiveness, performance, and scalability, and can overwhelm a database, particularly in a highly parallel and asynchronous architecture such as event-driven architecture. (See "Data Topologies" for ways to mitigate this risk.) Key-based event payloads also present a challenge if the required data is not easily accessible (such as if it lies within the bounded context of another event processor). Figure 15-12 illustrates this technique.

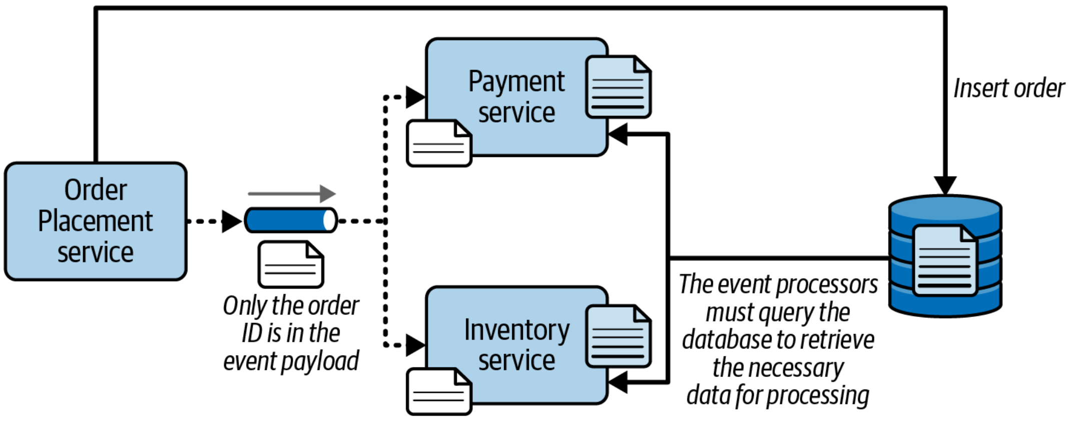

Payment service
Insert order
Order Placement service
Only the order ID is in the event payload
Inventory service
The event processors must query the database to retrieve the necessary data for processing

Figure 15-12. With key-based event payloads, only the context key is contained in the event

However, using key-based event payloads carries many advantages, some of which may outweigh the performance and scalability issues. The first main advantage is better overall data consistency and data integrity, thanks to having a *single system of record*. Because data about the event is located in only one place (the database), key-based event payloads can handle changes to the data during event processing much more easily than data-based event payloads.

The second main advantage is that because the contract in key-based event payloads is so simple and rarely changes, architects typically implement it through loose, schema-less JSON or XML. Therefore, key-based event payloads don't have the same issues with contract change management, versioning, and communication and deprecation strategies that data-based event payloads tend to have.

Another advantage of key-based event payloads: they don't have the same stamp coupling and bandwidth issues as data-based event payloads. Because there's no opaque data associated with the event, contracts are simple, small, and utilize minimal bandwidth. Thus, they tend to perform faster, from a network and message broker perspective, than data-based event payloads.

Trade-off summary

Choosing between a data-based event payload versus a key-based data payload requires careful trade-off analysis. Remember, it's not an all-or-nothing proposition: each type of event can use a different payload type. Table 15-1 summarizes the trade-offs associated with data-based and key-based event payloads.

Table 15-1. Data-based versus key-based event payloads

Criteria	Data-based payloads	Key-based payloads
Performance and scalability	Good	Bad
Contract management	Bad	Good
Stamp coupling	Bad	Good
Bandwidth utilization	Bad	Good
Restricted database access	Good	Bad
Overall system fragility	Bad	Good

Notice that the overall trade-off between these two options boils down to scalability and performance versus contract management and bandwidth utilization. Ask which one is more important for each particular event. Some event processing requires extreme levels of scale and performance, in which case a data-based event payload would be a better choice; some event processing data will undergo frequent change, in which case a key-based event payload might be more appropriate.

As with most things in software architecture, architects' choices lie on a spectrum, not a simple binary. That's why it's important to be careful to avoid triggering what are known as anemic events.

Anemic events

An *anemic event* is a derived event with a payload that doesn't contain enough information to help the event processor make decisions, and lacks the necessary context for further downstream processing.

Figure 15-13 illustrates an anemic derived event. In this example, a customer has updated some information in their user profile. Once that information is updated in the database, the `Customer Profile` event processor triggers a `profile_updated` event, using a key-based event payload that passes only the customer ID as its key-based data.

The three services responding to this event receive only the customer ID and the context that the customer's profile was changed. The first service (`Service 1`) has no idea what data was changed in the profile: name, address, some other critical information? Unfortunately, querying the database cannot answer this question, so `Service 1` has no idea how to respond or what action to take. `Service 2` responds to the `profile_updated` event, but going only by the key, does not know if it needs to perform any additional processing. Finally, `Service 3` responds to the same event, but has no idea what the prior values were and therefore cannot perform its processing. All three of these event processors need to respond in some way to the customer updating their profile but can't, due to a lack of information. These are anemic events: events that do not include additional information to further process the event.

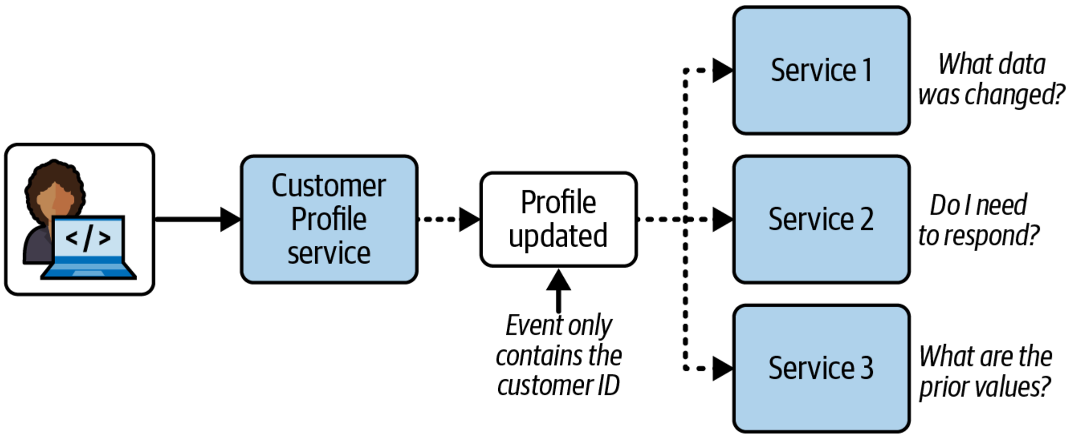


Figure 15-13. An anemic event lacks enough context to process the event

To avoid anemic events such as this one, include the updated customer information *as well as the prior values*, since most databases typically don't reflect that information.

This is an example of the *spectrum* of event payload granularity. On the extreme left side of the spectrum sit key-based event payloads, where only the key is contained within the event. While this has merit when creating or deleting an order, it doesn't work well when a customer updates an order. On the far right side of the spectrum is the data-based event payload, where *all* the data is included, whether it's needed or not. This is where stamp coupling rears its ugly head. The customer-profile-update scenario fits somewhere between these extremes because it provides the right level of information, thus avoiding the anemic derived event problem.

The Swarm of Gnats Antipattern

Related to anemic events is an antipattern known as the *Swarm of Gnats*. You probably know gnats as very small, annoying flying insects that buzz around your head, bothering you enough to send you back indoors on a beautiful sunny day. Whereas anemic events are concerned about the granularity of an event *payload*, the Swarm of Gnats antipattern is concerned about the granularity of the triggered events *themselves* and with how many derived events are triggered from an event processor. If an architect triggers too many derived events from a single event processor, they risk getting caught up in the Swarm of Gnats antipattern.

Consider the credit card payment example shown in Figure 15-14, where a customer places an order and their credit card is charged to pay for it. When the credit card is charged, the `Payment` event processor triggers a `payment applied` event, and (fortunately) the `Fraud Detection` event processor listens for it. This event processor analyzes each charge to determine whether it's legitimate or fraudulent. Whatever the outcome, the `Fraud Detection` event processor triggers a `fraud_checked` derived event with the outcome of the fraud check contained in its event payload.

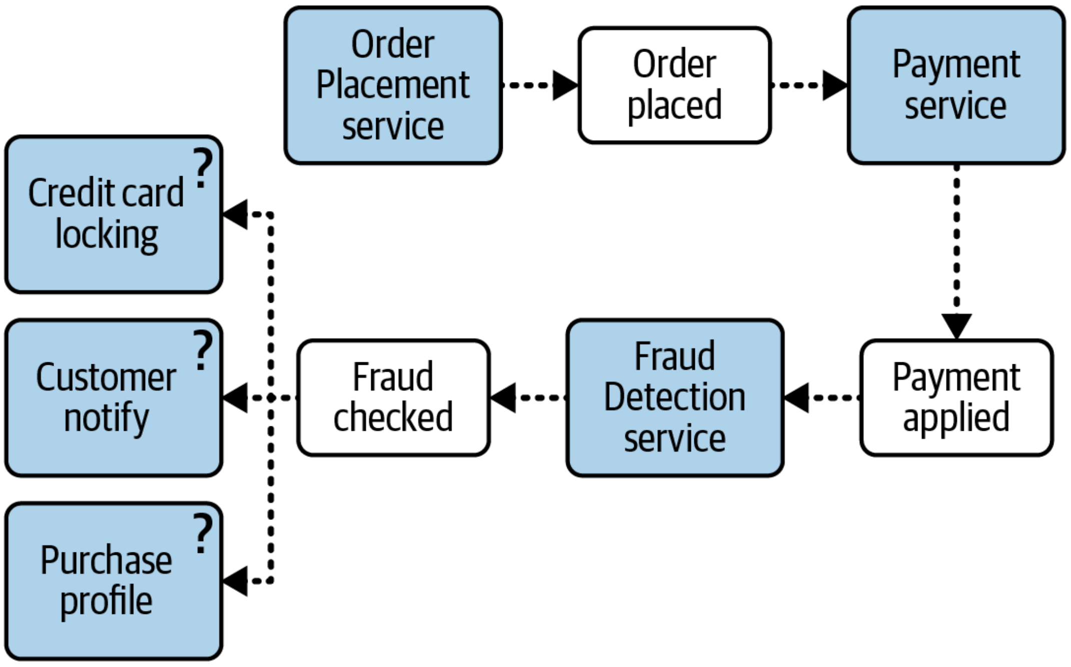


Figure 15-14. An example of an event that is too coarse-grained

Three event processors are interested in the outcome of the credit card fraud check:

- If fraud is detected, the `Credit Card Locking` event processor locks the customer's credit card to prevent further charges.
- The `Customer Notify` event processor notifies the customer of possible fraud.
- If fraud is *not* detected, the `Purchase Profile` event processor updates its algorithms.

Unfortunately, when a single `fraud_checked` derived event is triggered, *all* of these event processors must respond to the event, check the payload for the outcome, and decide whether to take action. Because this derived event is too coarse-grained, all of the event processors must perform additional processing: analyzing the payload of the single derived event to decide whether to take action. If no fraud has been detected, this is a waste of bandwidth and processing power, since only the `Purchase Profile` event processor needed to take any action.

A much more efficient approach would be to trigger *two* separate derived events (`fraud_detected` and `no_fraud_detected`), as shown in Figure 15-15. Here, the derived events triggered by the `Fraud Detection` event processor provide context *outside* of the event payload, allowing each event processor to decide whether to respond without having to analyze the event's internal payload.

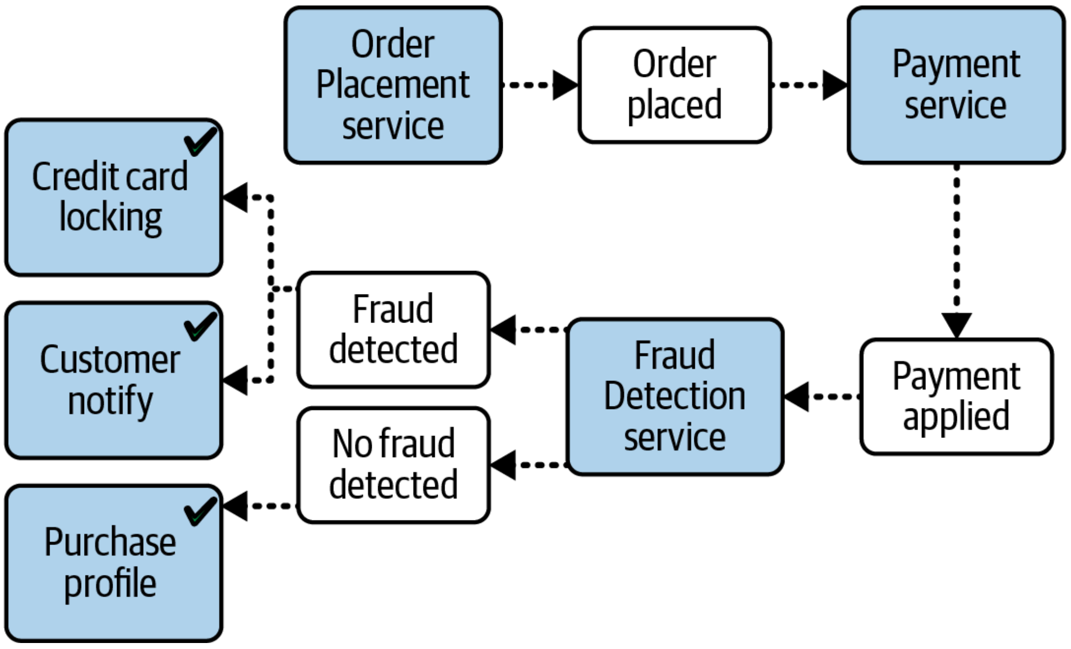


Figure 15-15. Triggering multiple events allows for more efficient processing and decision making

In this example, triggering multiple derived events for each outcome allows for better event flow, less churn, and more efficient processing. However, triggering *too many* derived events results in the *Swarm of Gnats* antipattern.

The scenario shown in Figure 15-16 illustrates how this antipattern can occur. A customer has recently moved and needs to change their user profile on the website to update their credit card's bill-to address, ship-to address (where orders will be shipped), and phone number from their old landline to their cell phone. When the customer hits the Submit button for these profile changes, the `Customer Profile` event processor receives the update request, updates the database, and triggers a separate event for each update that contains the necessary information to do any further processing.

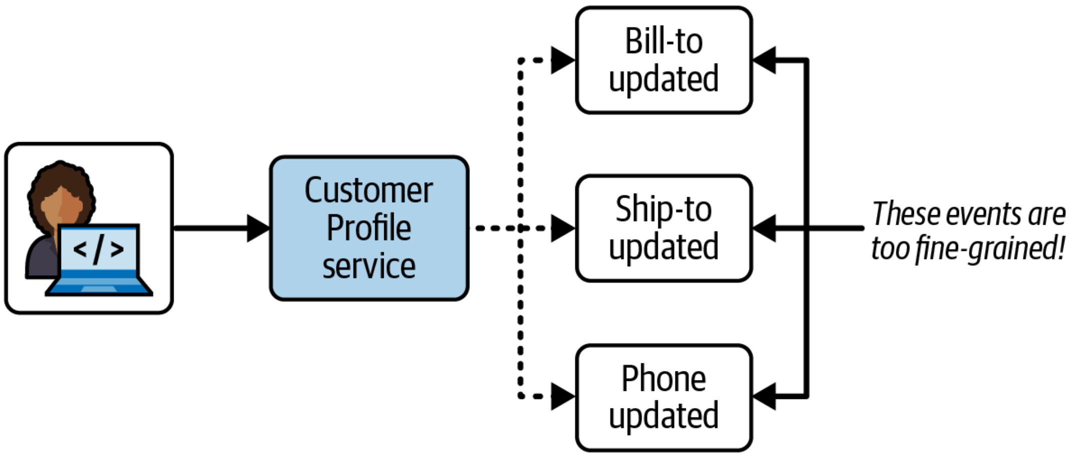


Figure 15-16. Triggering too many fine-grained derived events is known as the Swarm of Gnats antipattern

The problem with triggering too many fine-grained, detailed events is that it can saturate and overwhelm the system with derived events all related to the same thing: the customer updated their user profile. This antipattern also tends to proliferate numerous small derived events from other event processors, eventually making it hard for anyone to understand the system's overall event flows.

To avoid this antipattern, the architect could bundle each individual profile update into a single `profile_updated` derived event for the complete action, containing the before and after data of all updated fields. This more efficient approach is illustrated in Figure 15-17.

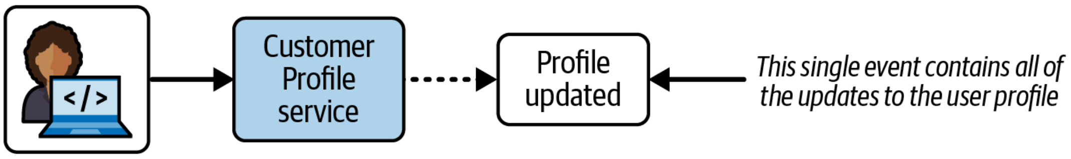

Customer Profile service
Profile updated
This single event contains all of the updates to the user profile

Figure 15-17. Combining individual state changes into a single derived event avoids the Swarm of Gnats antipattern

Determining the right level of granularity for derived events can be quite challenging. We recommend focusing on the *outcome* of the processing or state change to avoid the Swarm of Gnats antipattern and help simplify event flows.

Error Handling

The Workflow Event pattern of reactive architecture is one way of addressing error handling in an asynchronous workflow. This pattern addresses both resiliency and responsiveness, since it allows the system to handle asynchronous errors without affecting its responsiveness.

The Workflow Event pattern leverages delegation, containment, and repair by using a *workflow delegate*, as illustrated in Figure 15-18. In this pattern, an event processor asynchronously passes data through a message channel to the event consumer. If the event consumer experiences an error while processing the data, it immediately delegates that error to the `Workflow Processor` service and moves on to the next message in the event queue. This way, the next message is immediately processed, so overall responsiveness remains the same. If the event consumer were to spend time trying to figure out the error, then it would not be processing the next message in the queue—delaying not only the next message, but all other messages waiting in the processing queue.

When the `Workflow Processor` service receives an error, it tries to figure out what's wrong with the message. Perhaps there's a static, deterministic error? It could analyze the message using some machine-learning or AI algorithms to look for some anomaly in the data. Either way, the workflow processor *programmatically* (that is, without human intervention) makes changes to the original data to try to repair it, then sends it back to the originating queue. The event consumer sees this updated message as a new one and tries to process it again, hopefully this time with more success.

Of course, the workflow processor can't always determine what's wrong with the message. In these cases, it sends the message off to another queue, which is then received by a dashboard on the desktop of a knowledgeable person. This person looks at the message, applies manual fixes, and then resubmits it to the original queue (usually through a reply-to message header variable).

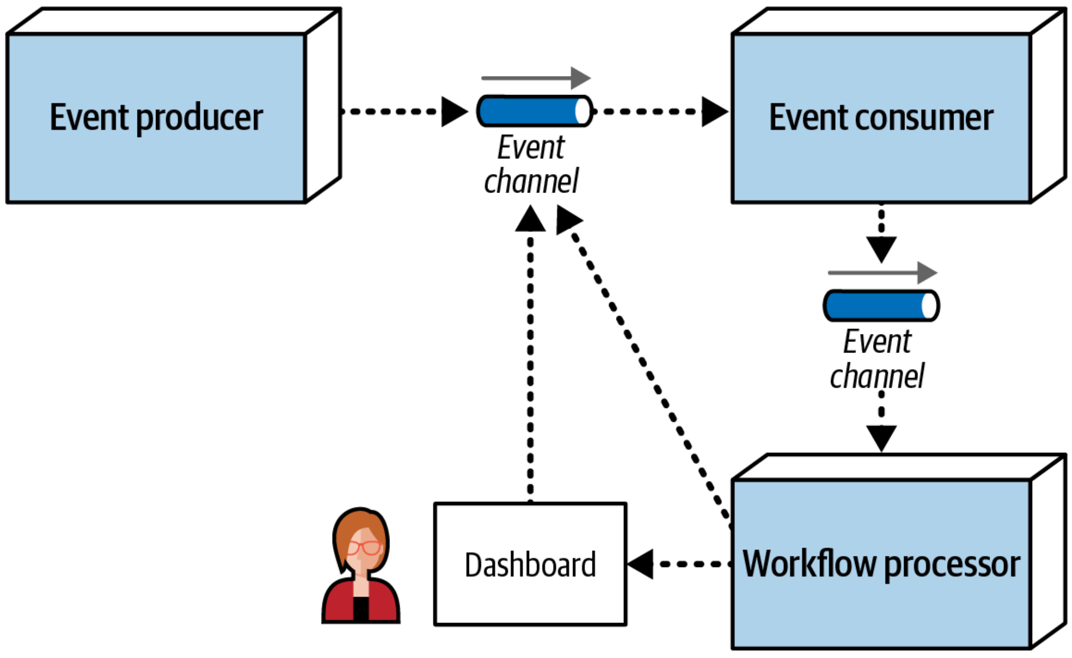


Figure 15-18. The Workflow Event pattern of reactive architecture

Suppose a trading advisor in one part of the country accepts *trade orders* (instructions on what stock to buy and for how many shares) on behalf of a large trading firm in another part of the country. The advisor batches up the trade orders in what is usually called a *basket* and asynchronously sends them to a broker across the country, who then purchases the stock. To simplify the example, suppose the contract for the trade instructions must adhere to the following:

```
ACCOUNT(String),SIDE(String),SYMBOL(String),SHARES(Long)
```

Suppose the large trading firm receives the following basket of Apple (AAPL) trade orders from the trading advisor:

```
12654A87FR4,BUY,AAPL,1254
87R54E3068U,BUY,AAPL,3122
6R4NB7609JJ,BUY,AAPL,5433
2WE35HF6DHF,BUY,AAPL,8756 SHARES
764980974R2,BUY,AAPL,1211
1533G658HD8,BUY,AAPL,2654
```

The fourth trade instruction (`2WE35HF6DHF,BUY,AAPL,8756 SHARES`) has the word `SHARES` after the number of shares for the trade. When the primary firm processes these asynchronous trade orders without any error handling capabilities, the following error occurs within the `TradePlacement` service:

```
Exception in thread "main" java.lang.NumberFormatException:
        For input string: "8756 SHARES"
        at java.lang.NumberFormatException.forInputString
        (NumberFormatException.java:65)
        at java.lang.Long.parseLong(Long.java:589)
        at java.lang.Long.<init>(Long.java:965)
        at trading.TradePlacement.execute(TradePlacement.java:23)
        at trading.TradePlacement.main(TradePlacement.java:29)
```

When this exception occurs, because this was an asynchronous request, there is no user to synchronously respond to and fix the error. The `TradePlacement` service can't do anything, except possibly log the error condition.

Applying the Workflow Event pattern can fix this error programmatically. Because the primary firm has no control over the trading advisor or the trade-order data it sends, it must react to fix the error itself (see Figure 15-19). When the same error occurs (`2WE35HF6DHF,BUY,AAPL,8756 SHARES`), the `TradePlacement` service immediately delegates the error via asynchronous messaging to the `Trade Placement Error` service for error handling, passing it along with the error information about the exception:

```
Trade Placed: 12654A87FR4,BUY,AAPL,1254
Trade Placed: 87R54E3068U,BUY,AAPL,3122
Trade Placed: 6R4NB7609JJ,BUY,AAPL,5433
Error Placing Trade: "2WE35HF6DHF,BUY,AAPL,8756 SHARES"
Sending to trade error processor <-- delegate the error fixing and move on
Trade Placed: 764980974R2,BUY,AAPL,1211
...
```

The `Trade Placement Error` service, acting as the workflow delegate, receives the error and inspects the exception. Seeing that the issue is with the word `SHARES` in the Number of Shares field, the `Trade Placement Error` service strips off the word `SHARES` and resubmits the trade for reprocessing:

```
Received Trade Order Error: 2WE35HF6DHF,BUY,AAPL,8756 SHARES
Trade fixed: 2WE35HF6DHF,BUY,AAPL,8756
Resubmitting Trade For Re-Processing
```

The `TradePlacement` service can now process the fixed trade successfully:

```
...
trade placed: 1533G658HD8,BUY,AAPL,2654
trade placed: 2WE35HF6DHF,BUY,AAPL,8756 <-- this was the original trade in error
```

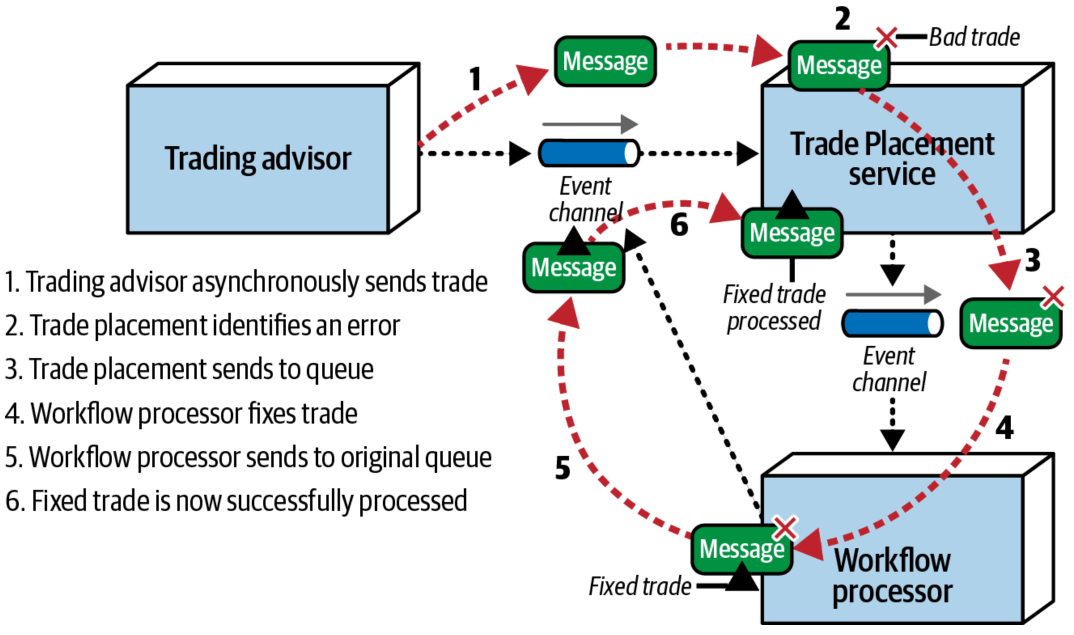

Trading advisor
Message
1
2
Bad trade
Message
Trade Placement service
Event channel
6
Message
Message
Fixed trade processed
3
Message
Event channel
4
5
Workflow processor
Message
Fixed trade
1. Trading advisor asynchronously sends trade
2. Trade placement identifies an error
3. Trade placement sends to queue
4. Workflow processor fixes trade
5. Workflow processor sends to original queue
6. Fixed trade is now successfully processed

Figure 15-19. Error handling with the Workflow Event pattern

One consequence of using the Workflow Event pattern is that messages that are sent to a workflow processor and then resubmitted are processed out of sequence. In our trading example, the order of the messages matters, because all trades within a given account must be processed in order (for example, a `SELL` for IBM must occur before a `BUY` for AAPL within the same brokerage account). It would be complex, although not impossible, to maintain the message order within a given context (in this case, the brokerage account number). One way to address this is for the `TradePlacement` service to queue and store the brokerage account number of the erroneous trade. Any trade with that same brokerage account number would be stored in a temporary queue for later processing (in first-in, first-out, or FIFO, order). Once the erroneous trade is fixed and processed, the

`TradePlacement` service de-queues the remaining trades for that same account and processes them in order.

Preventing Data Loss

Architects dealing with asynchronous communications are always concerned with *data loss*: when an event or message gets dropped or never makes it to its final destination. Fortunately, there are basic out-of-the-box techniques to prevent data loss.

Architects can implement event channels in a variety of ways. Most event-driven architectures use the Advanced Message Queuing Protocol (AMQP) for triggering and responding to events. Examples of AMQP brokers include Amazon SNS (Simple Notification Service), RabbitMQ, Solace, and Azure Event Hubs. With AMQP, events are published to an exchange. The exchange uses the binding rules set by the consuming event processors to forward the event to a queue for each event processor that subscribes to that event. AMQP brokers can also leverage what is known as the *Event Forwarding* pattern to prevent data loss, a technique we describe in this section.

Another event channel implementation is the Jakarta Messaging API (formerly Java Message Service or JMS), which uses *topics* rather than the two-step forwarding process queues use. Nevertheless, Jakarta Messaging can still leverage the *Event Forwarding* pattern to prevent data loss, provided that the event processors responding to an event are configured as durable subscribers. A *durable subscriber* is one that is guaranteed to receive an event. If the event processor is down or otherwise unavailable, the JMS topic stores the event until the subscribing event processor becomes available.

Another possible event-channel implementation is event streaming using Kafka as an *event broker* (the software product containing the queues and topics). The techniques for preventing data loss within event streaming are very different from those used for the Event Forwarding pattern described in this section. Refer to the Kafka website for more information about preventing data loss when using this kind of streaming event broker.

Consider a typical scenario in which event processor `A` asynchronously publishes an event to a message broker, which eventually goes to an AMQP queue or JMS topic. Event processor `B` responds to the event and inserts the data from the payload into a database. As illustrated in Figure 15-20, there are three ways data loss can occur in this scenario:

1. While event processor `A` is publishing the event, it crashes before an acknowledgment from the event broker can be sent; alternately, the event broker sends an acknowledgment to event processor `A`, but then crashes before the event is accepted by another event processor.

2. Event processor `B` accepts the event from the queue but crashes before it can process the event.

3. Event processor `B` is unable to persist the message to the database due to a data error.

Each of these areas of data loss can be mitigated through the *Event Forwarding* pattern.

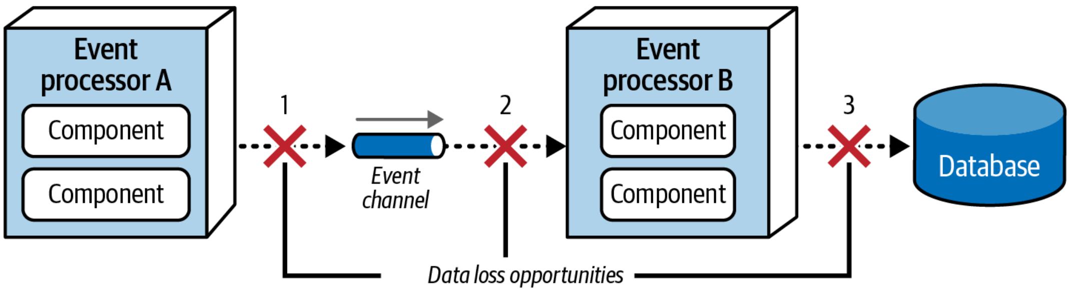


Figure 15-20. Places where data loss can happen within an event-driven architecture

With the first issue, the event never makes it to the queue or the broker fails before the event is read. To address this, use persistent message queues along with synchronous send. Persisted message queues support *guaranteed delivery*: when the event broker receives the event, not only does it store the event in memory for fast retrieval, it also persists the event in some sort of physical data store (such as a filesystem or database). If the event broker goes down, the event is physically stored on disk, so it will still be available for processing when the event broker comes back up. *Synchronous send* does a blocking wait in the event processor, stopping it from triggering the event until the broker acknowledges that it has persisted the event to the database. These two basic techniques prevent data loss between the event producer and the queue, because the event is either still with the event producer or persisted within the queue.

A basic messaging technique called *client acknowledge mode* can address the second issue, where event processor `B` de-queues the next available event and crashes before it can process the event. By default, when an event is read from a queue, it is immediately removed from that queue (this is called *auto acknowledge* mode). Client acknowledge mode keeps the event in the queue and attaches the client ID to it so that no other consumers can read or process the event. With this mode, if event processor `B` crashes, the event is still preserved in the queue, preventing message loss.

The third issue, in which event processor `B` is unable to persist the event to the database due to some data error, can be addressed with ACID transactions via a database commit. Once the event processor issues a database commit, the data is guaranteed to be persisted in the database. *Last participant support* (LPS) removes the event from the persisted queue by acknowledging that all processing has been completed and that the event has been persisted. This guarantees that the event has not been lost in transit from event processor `A` to the database. These techniques are illustrated in Figure 15-21.

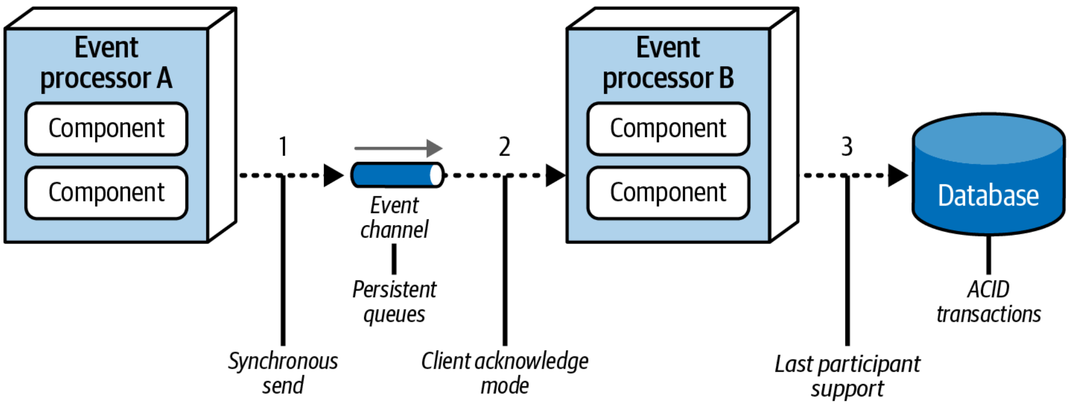


Figure 15-21. Preventing data loss within an event-driven architecture

Request-Reply Processing

So far in this chapter, we've dealt with asynchronous requests that don't need an immediate response from the event consumer. But what about event processors that need information back immediately from another event processor—for example, waiting for some sort of confirmation ID or acknowledgment before triggering an event? This scenario requires synchronous communication to complete the request.

In EDA, synchronous communication is typically accomplished through *request-reply* messaging (sometimes referred to as *pseudosynchronous communications*). Each event channel within request-reply messaging consists of two queues: a *request* queue and a *reply* queue. The message producer making the initial request for information asynchronously sends data to the request queue, and then returns control to the message producer. The message producer then does additional processing, and eventually waits on the reply queue for the response. The message consumer receives and processes the message, then sends the response to the reply queue. The event producer receives the message with the response data. This basic flow is illustrated in Figure 15-22.

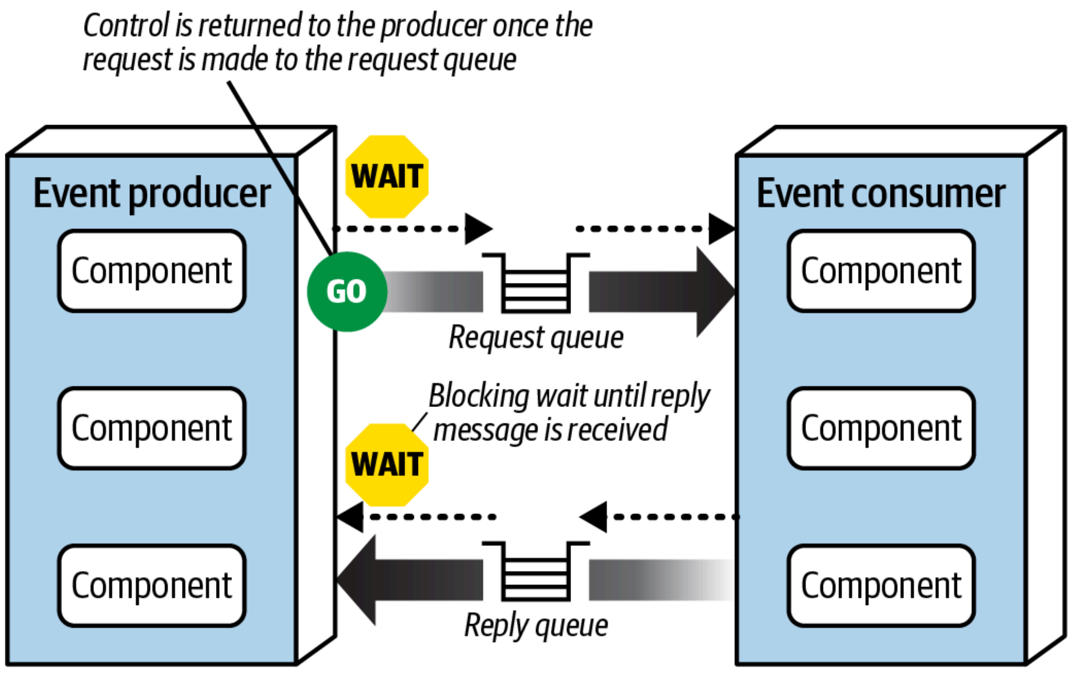

Control is returned to the producer once the request is made to the request queue
Event producer
Component
Component
Component
WAIT
GO
Request queue
Blocking wait until reply message is received
WAIT
Reply queue
Event consumer
Component
Component
Component

Figure 15-22. Request-reply message processing

There are two primary ways to implement request-reply messaging. The first (and most common) technique is to put a *correlation ID* (CID) field in the message header of the reply message, usually set to the message ID of the original request message (simply called ID in Figure 15-23). It works like this:

1. The event producer sends a message to the request queue and records the unique message ID (ID 124). Notice that the CID in this case is `null`.

2. The event producer does a blocking wait on the reply queue with a message filter (also called a *message selector*), where the CID in the message header equals the original message ID (124). There are two messages in the reply queue: ID 855 with CID 120, and ID 856 with CID 122. Neither of these messages will be picked up, because neither correlation ID matches what the event consumer is looking for (CID 124).

3. The event consumer receives the message (ID 124) and processes the request.

4. The event consumer creates the reply message containing the response and sets the CID in the message header to the original message ID (124).

5. The event consumer sends the new message ID (857) to the reply queue.

6. The event producer receives the message, because the CID (124) matches the message selector from step 2.

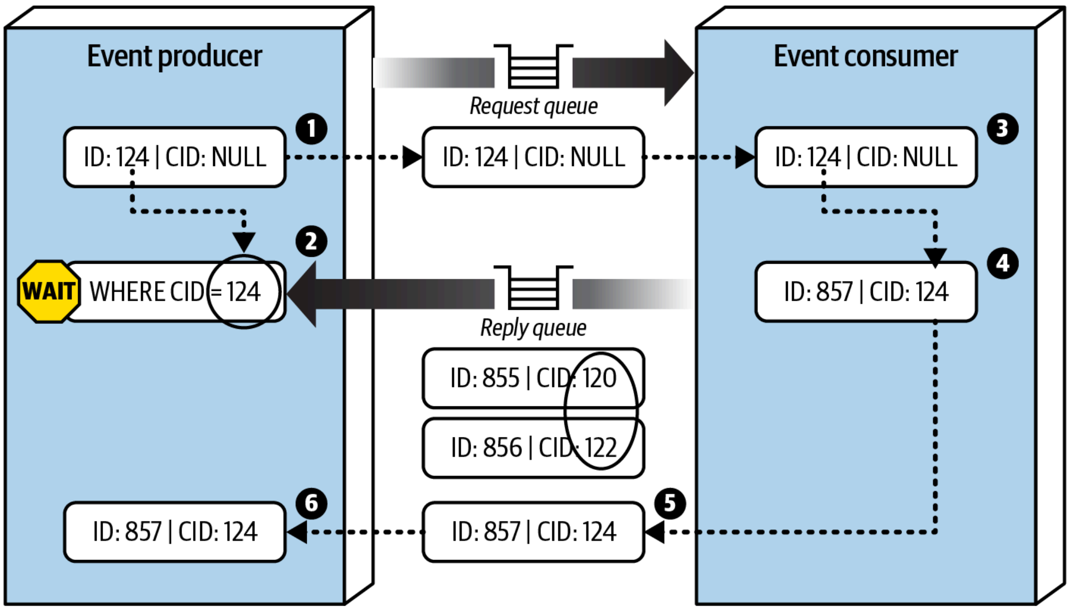


Figure 15-23. Request-reply message processing using a correlation ID

The other way to implement request-reply messaging is to use a *temporary queue* for the reply queue. A temporary queue is dedicated to a specific request, created when the request is made, and deleted when the request ends. This technique, as illustrated in Figure 15-24, does not require a correlation ID, because the temporary queue is a dedicated queue only known to the event producer for that specific request. The temporary queue technique works as follows:

1. The event producer creates a temporary queue (or one is automatically created, depending on the message broker) and sends a message to the request queue, passing the name of the temporary queue in the reply-to header (or some other agreed-upon custom attribute in the message header).

2. The event producer does a blocking wait on the temporary reply queue. No message selector is needed because any message sent to this queue belongs solely to the event producer that sent the original message.

3. The event consumer receives the message, processes the request, and sends a response message to the reply queue named in the reply-to header.

4. The event processor receives the message and deletes the temporary queue.

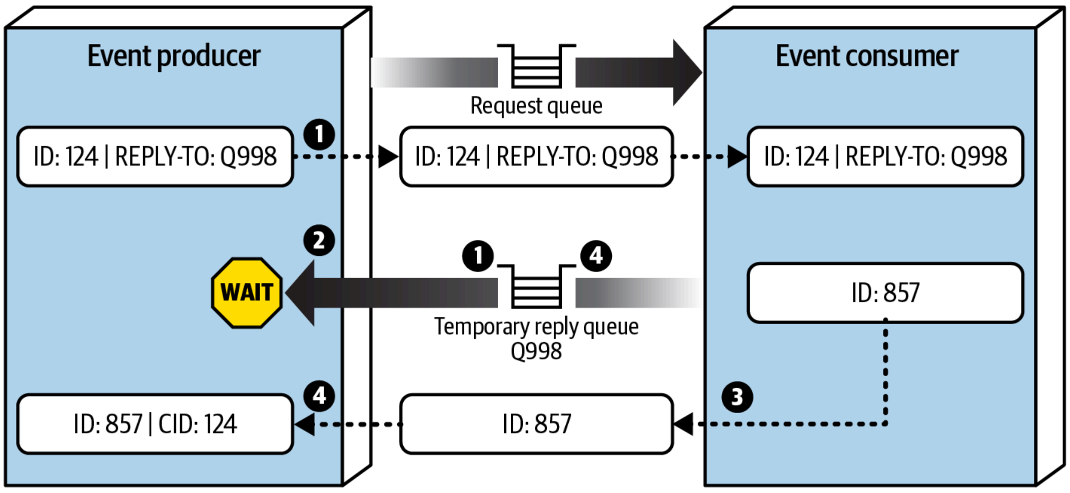


Figure 15-24. Request-reply message processing using a temporary queue

While the temporary queue technique is much simpler, the message broker must create a temporary queue for each request and then delete it immediately. This can significantly slow the broker down and can affect overall performance and responsiveness, particularly for large message volumes and high concurrency. For this reason, we usually recommend using the correlation ID technique.

Mediated Event-Driven Architecture

So far in this chapter we've focused on *choreographed* EDA, where event processors trigger events through broadcast capabilities, and multiple event processors respond to the event. However, there may be times when an architect wants more control over the processing of an event. In this case, the architect can use an *orchestrated* form of EDA known as the mediator topology.

The *mediator topology* addresses some of the shortcomings of the standard choreographed EDA topology we've described so far in this chapter. It's centered on an *event mediator*, which manages and controls the workflow for initiating events that require coordination between multiple event processors. The architecture components that make up the mediator topology are: an initiating event, an event queue, an event mediator, event channels, and event processors.

Importantly, the mediated topology typically uses *messages* rather than *events* (see "Events Versus Messages"). They are generally commands (such as `ship_order`) rather than events that have happened (such as `order_shipped`).

Like in the choreographed topology, the initiating event is what starts the whole process. However, in the mediator topology (Figure 15-25), an event mediator accepts the initiating event. It only knows the steps involved in processing the event, so it generates corresponding derived messages and sends them to dedicated message channels (usually queues) in a point-to-point fashion. Event processors then listen to the dedicated event channels, process messages, and (usually) respond to the mediator when they have completed their work. Event processors within the mediator topology do not advertise what they've done to the rest of the system through additional derived messages.

In most implementations of the mediator topology, there are multiple mediators, each usually associated with a particular domain or grouping of events. This avoids having a single point of failure, which can be an issue with this topology, and increases overall throughput and performance. For example, a customer mediator might handle all customer-related events (such as new customer registrations and profile updates), while an order mediator handles order-related activities (such as adding an item to a shopping cart and checking out).

How architects choose to implement the event mediator usually depends on the nature and complexity of the messages the event mediator is processing. For example, for events requiring simple error handling and orchestration, mediators such as Apache Camel, Mule ESB, or Spring Integration will usually suffice. Message flows and message routes within these types of mediators are typically custom written in programming code (such as Java or C#) to control the event-processing workflow.

However, if the event workflow requires lots of conditional processing and multiple dynamic paths with complex error handling directives, then a mediator such as Apache ODE or the Oracle BPEL Process Manager would be a more appropriate choice. These mediators are based on the Business Process Execution Language (BPEL), an XML-like structure that describes the steps involved in processing an event. BPEL artifacts also contain structured elements used for error handling, redirection, multicasting, and so on. BPEL is a powerful but relatively complex language to learn, so architects usually create mediators using the GUI tools in BPEL's engine suite.

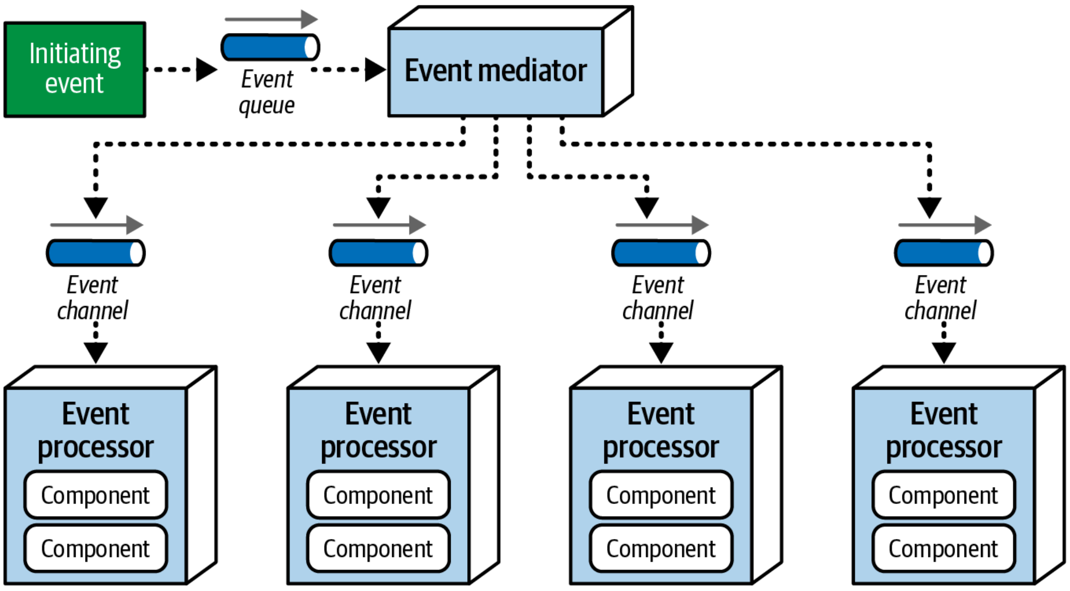


Figure 15-25. Mediator topology

BPEL is good for complex, dynamic workflows, but it does not work well for event workflows with long-running transactions that involve human intervention throughout the event process. For example, suppose a trade is being placed through a `place_trade` initiating event. The event mediator accepts this event, but during the processing, finds that manual approval is required because the trade is over a certain number of shares. The event mediator now has to stop the event processing, notify a senior trader to get the manual approval, and wait for that approval. In these cases, a Business Process Management (BPM) engine, such as jBPM, would be more appropriate than using an event mediator.

Before choosing what kind of event mediator to implement, it's important to know the types of events it will process. For complex, long-running events involving human interaction, Apache Camel would be extremely difficult to use and maintain. By the same token, using a BPM engine for simple event flows would waste months of effort on something Apache Camel could accomplish in a matter of days.

Of course, it's rare to have all events fit neatly into one class of complexity. We recommend classifying events as simple, hard, or complex, and sending every event through a simple mediator, such as Apache Camel or Mule. The simple mediator can handle the event itself or forward it to another, more complex event mediator based on that complexity classification. This mediator delegation model, illustrated in Figure 15-26, ensures that all types of events are handled by the type of mediator that will process them most effectively.

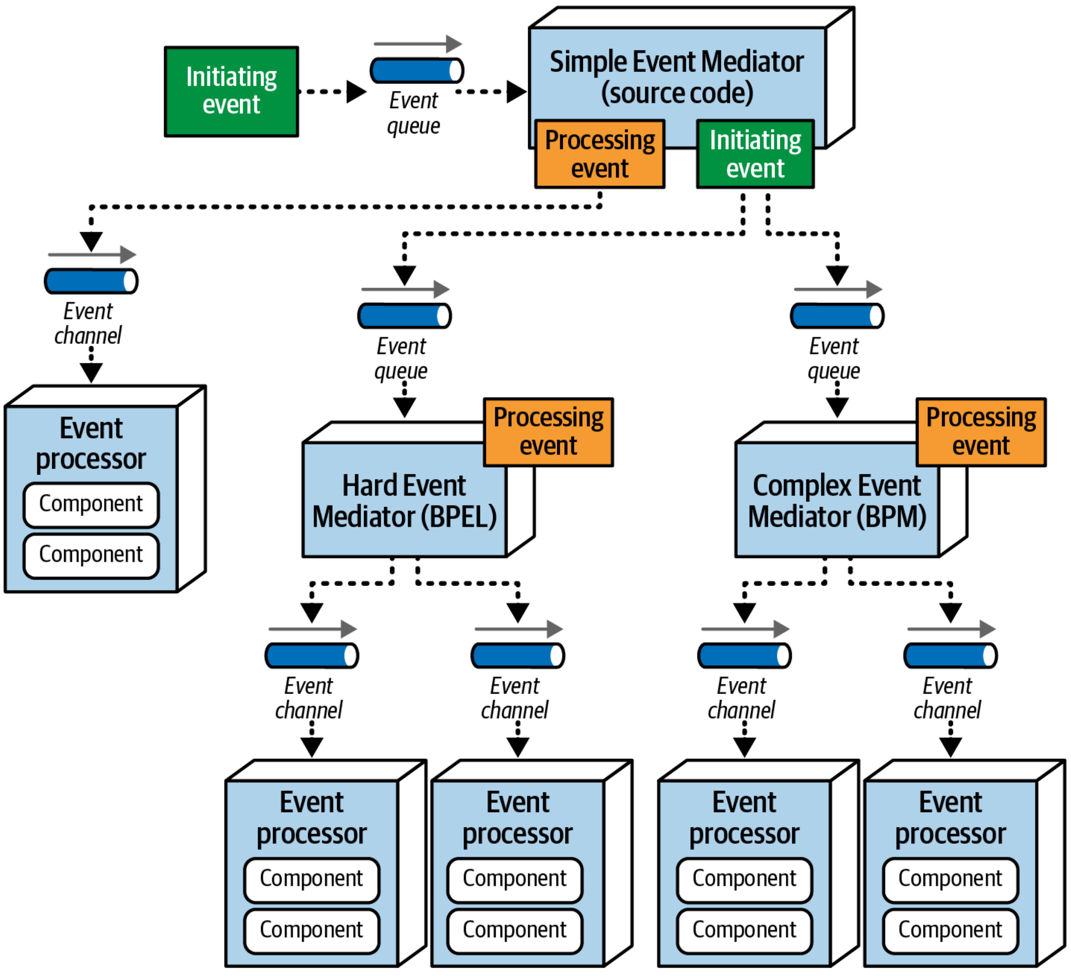


Figure 15-26. Delegating the event to the appropriate type of event mediator

Notice in Figure 15-26 that the `Simple Event Mediator` generates and sends a derived message when the event workflow is simple enough to be handled entirely by the simple mediator. However, when the initiating event is classified as hard or complex, the `Simple Event Mediator` forwards the original initiating event to the corresponding mediators (BPEL or BPM). The `Simple Event Mediator`, having intercepted the original event, may still be responsible for knowing when that event is complete, or it could simply delegate the entire workflow (including client notification) to the other mediators.

To examine how the mediator topology works, let's consider the same retail order entry system that we described in the section on choreographed topology, but this time using the mediator topology. The mediator knows the steps required to process this particular event. The mediator component's internal event flow is illustrated in Figure 15-27.

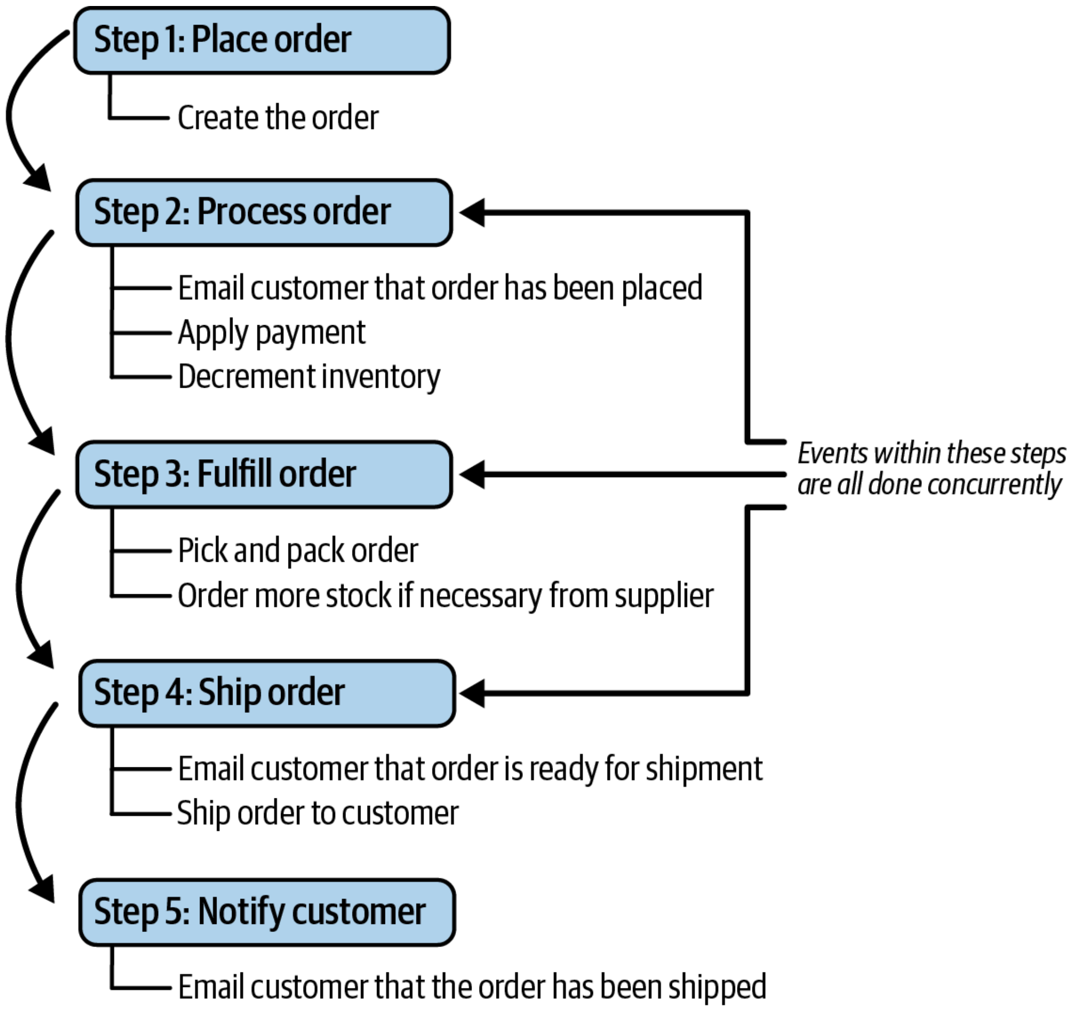

Step 1: Place order
Create the order
Step 2: Process order
Email customer that order has been placed
Apply payment
Decrement inventory
Step 3: Fulfill order
Pick and pack order
Order more stock if necessary from supplier
Step 4: Ship order
Email customer that order is ready for shipment
Ship order to customer
Step 5: Notify customer
Email customer that the order has been shipped
Events within these steps are all done concurrently

Figure 15-27. Mediator steps for placing an order

In keeping with the prior example, the same initiating event (`place order`) is sent to the event mediator via a dedicated queue for processing. The `Customer` mediator picks up this initiating event and begins generating derived messages, based on the flow in Figure 15-27. The events shown in steps 2, 3, and 4 are all done concurrently and serially, between steps. In other words, step 3 (fulfill order) must be completed and acknowledged before the customer can be notified that the order is ready to be shipped in step 4 (ship order).

Once it receives the initiating event, the `Customer` mediator generates a `create order` derived message, which it sends to the `order placement` queue (see Figure 15-28). The `Order Placement` event processor accepts the message and validates and creates the order, and sends the mediator an acknowledgment and the order ID in return. At this point, the mediator might send that order ID to the customer, indicating that the order was placed, or it might have to continue until all the steps are complete (this would depend on specific business rules about order placement).

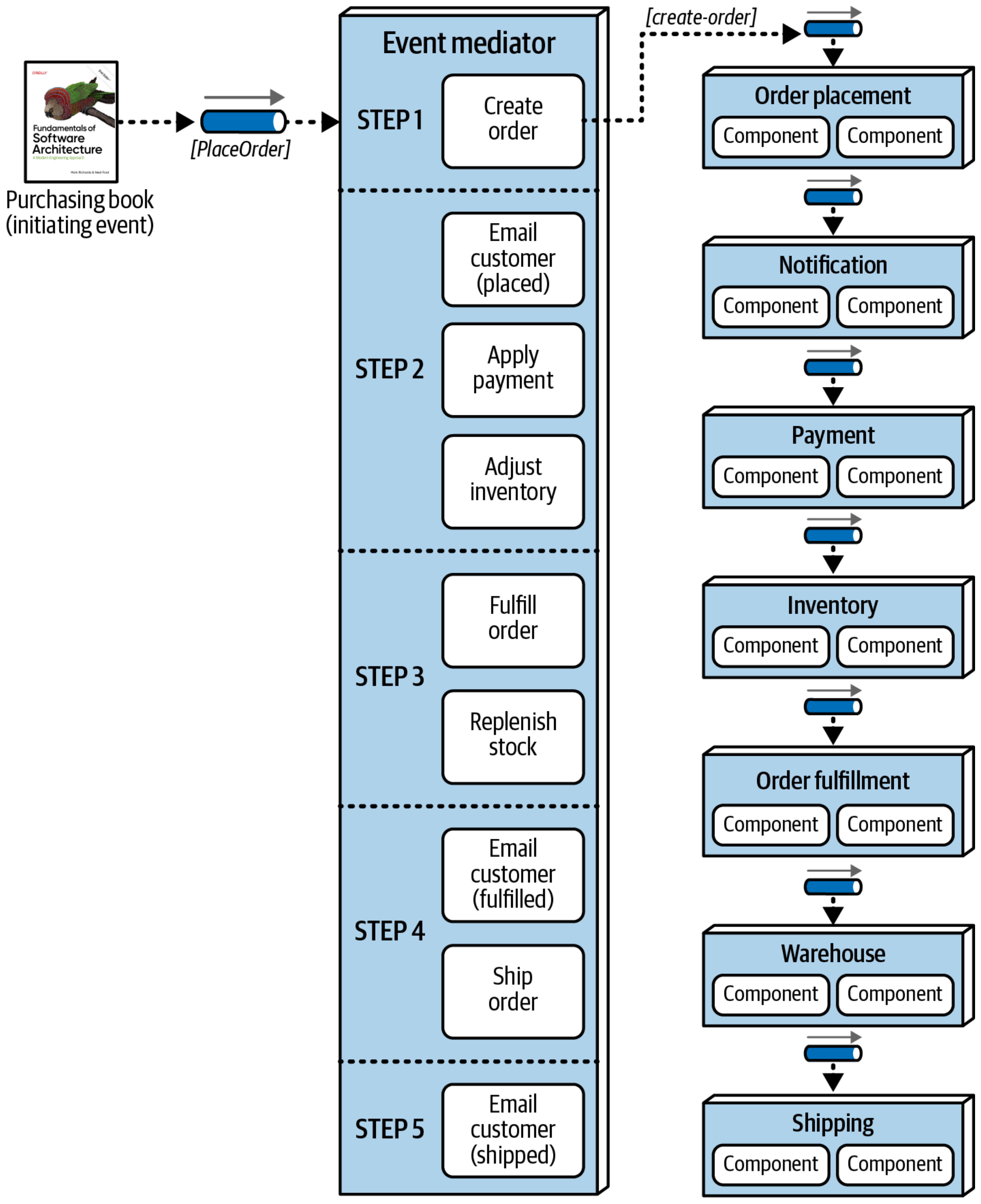


Figure 15-28. Step 1 of the mediator example

Now that step 1 is complete, the mediator moves to step 2 (see Figure 15-29) and generates three derived messages at the same time: `email customer`, `apply payment`, and `adjust inventory`. It sends all three to their respective queues. All three event processors receive these messages, perform their respective tasks, and notify the mediator that their processing is complete. The mediator must wait until it receives acknowledgment from all three parallel processes before moving on to step 3. If an error occurs in one of the parallel event processors, the mediator can take corrective action (more about that later in this section).

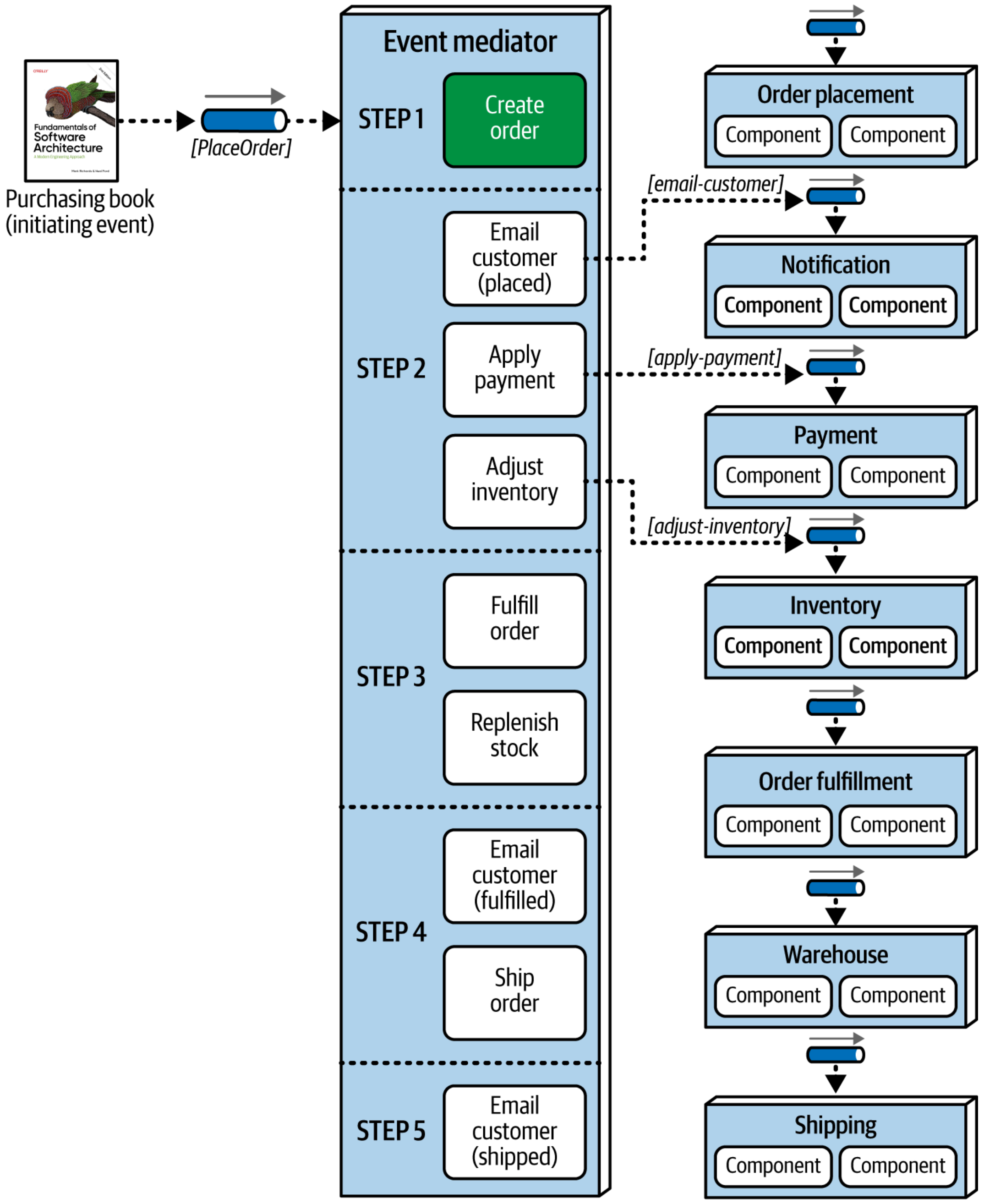


Figure 15-29. Step 2 of the mediator example

Once the mediator gets a successful acknowledgment from all of the event processors in step 2, it can move to step 3 to fulfill the order (see Figure 15-30). Once again, both of these messages (`fulfill order` and `order stock`) can occur simultaneously. The `Order Fulfillment` and `Warehouse` event processors accept the messages, perform their work, and return an acknowledgment to the mediator.

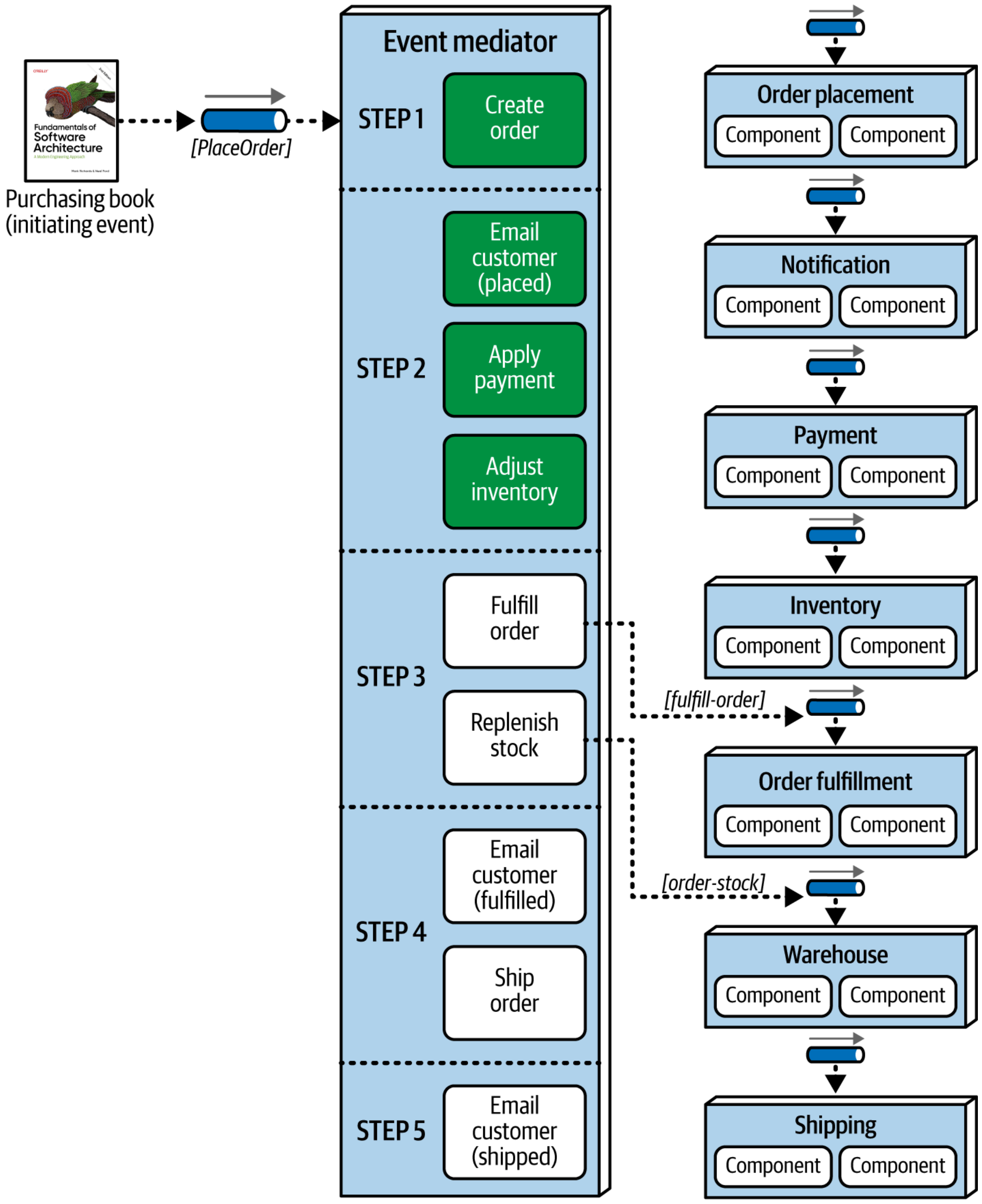


Figure 15-30. Step 3 of the mediator example

The mediator moves on to step 4 (see Figure 15-31) to ship the order. This step generates two derived messages: a `ship order` message, and another `email customer` message with specific information about what to do (notify the customer that the order is ready to be shipped).

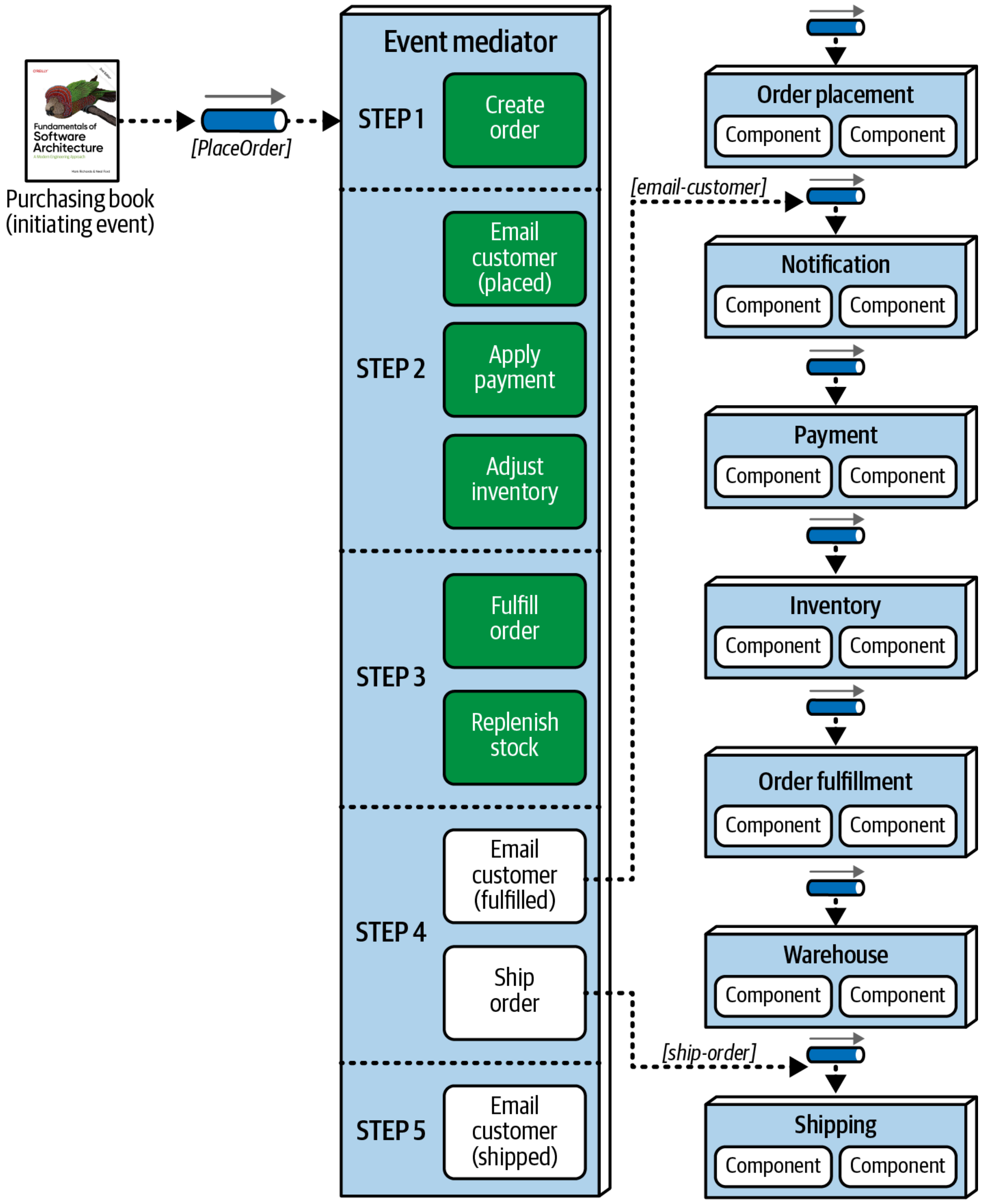


Figure 15-31. Step 4 of the mediator example

Finally, the mediator moves to step 5 (see Figure 15-32) and generates another contextual `email customer` message to notify the customer that the order has been shipped. This ends the workflow. The mediator marks the initiating event flow complete and removes all state associated with the initiating event.

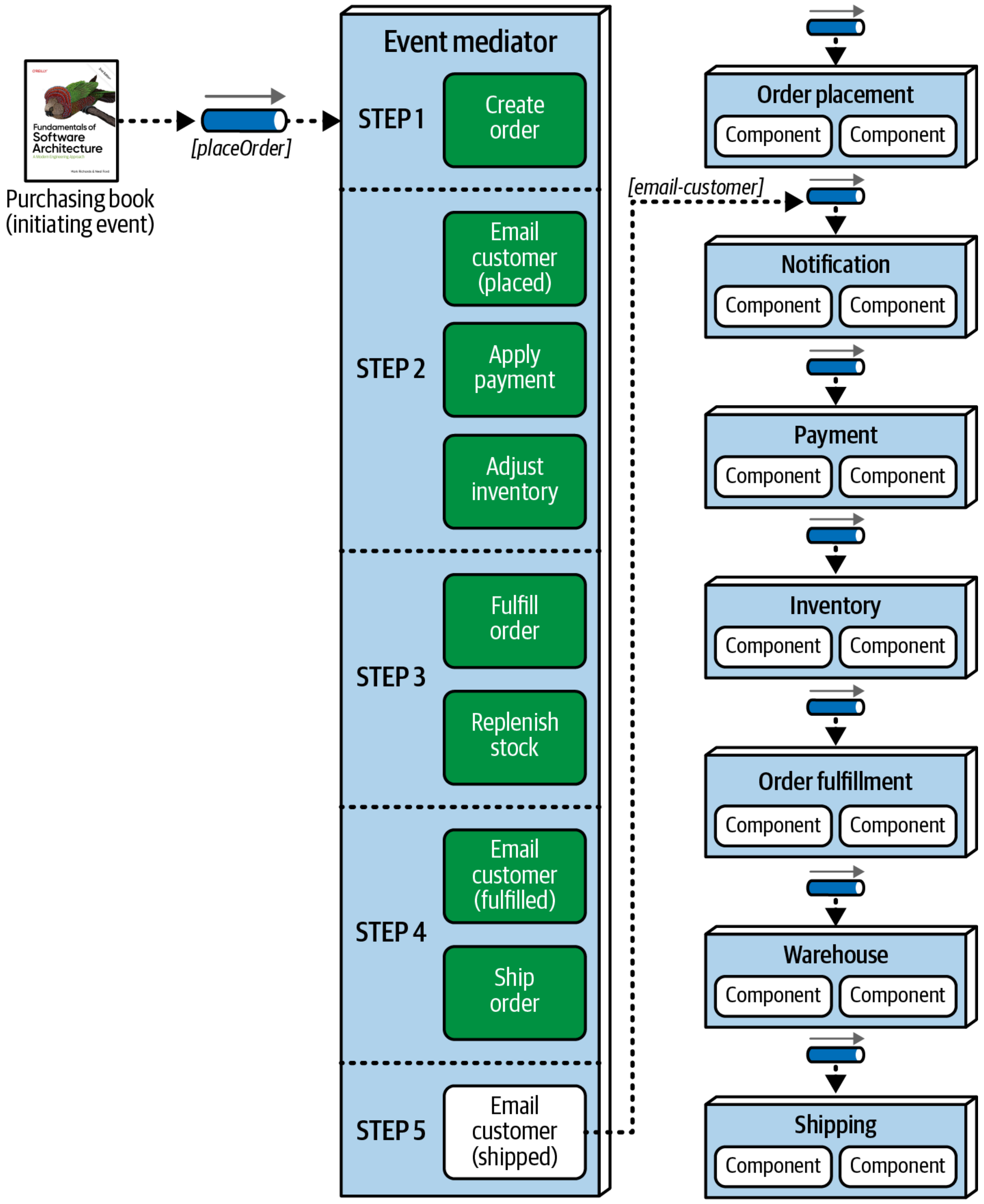


Figure 15-32. Step 5 of the mediator example

In this topology, unlike the choreographed topology, the mediator component has knowledge of and control over the workflow. It can maintain event state and manage error handling, recoverability, and restart capabilities. For example, suppose in our example that the payment was not applied because the credit card has expired. When the mediator receives this error condition, it knows that the order cannot be fulfilled (step 3) until payment is applied, so it stops the workflow and records the state of the request in its own persistent datastore. Once payment is eventually applied, the workflow can be restarted from where it left off (in this case, the beginning of step 3).

While the mediator topology addresses the issues associated with the choreographed topology, it has its own disadvantages. First of all, it is very difficult to declaratively model the dynamic processing that occurs within a complex event flow. As a result, many workflows within the mediator topology only handle general processing, but use a hybrid model that combines the mediator and choreographed topologies to address the dynamic nature of complex event processing, such as out-of-stock conditions or other nontypical errors. Furthermore, although the event processors can easily scale in the same manner as the choreographed topology, the mediator must scale as well—something that occasionally produces a bottleneck in the overall event-processing flow. Event processors are not as highly decoupled in the mediator topology as they are in the choreographed topology. Finally, performance is not as good in this topology, because the mediator controls event processing.

The trade-off between the choreographed and mediator topologies essentially comes down to weighing workflow control and error handling capability against high performance and scalability. Although performance and scalability are still good within the mediator topology, they are not as high as with the choreographed topology.

Data Topologies

With all this talk of events and event processing, it's easy to forget about the data side of EDA. Database topologies are a unique and interesting aspect of this architectural style. It offers many options, each with significant trade-offs that can have a big impact on the overall architecture. To describe the different database topologies within EDA, we'll use a simplified version of the example we showed in Figure 15-3 (see Figure 15-33).

When a customer places an order, the `Order Placement` event processor creates the order, then triggers an `order placed` event, to which both the `Payment` and `Inventory` event processors respond. Once the payment has been applied, the `Order Fulfillment` event processor helps the order packer prepare the order, then triggers an `order fulfilled` event. The `Shipping` event processor responds to the `order fulfilled` event by shipping the order to the customer, completing the processing and fulfilling the customer's request.

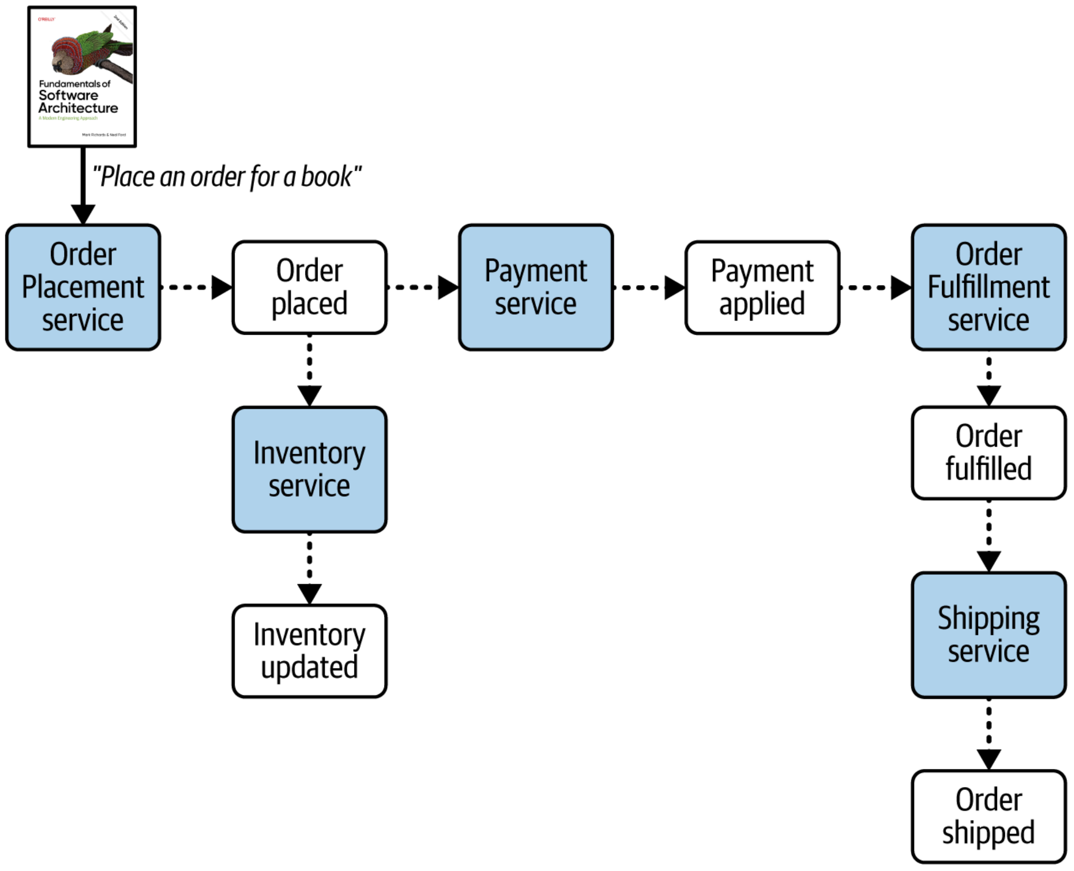


Figure 15-33. A simplified example of the example order-entry system using EDA

One complication of EDA is that the `Order Placement` event processor needs to know two pieces of information: how many items are currently in stock, and what shipping options are available based on the customer's location. How it gets this information will depend on the type of database topology the architecture uses. Let's look at each database topology option to see its significant trade-offs.

Monolithic Database Topology

The first, and perhaps most common, database topology used within EDA is the *single monolithic database* topology. With this topology, all data is available to all event processors through a central database.

The main benefit of the monolithic database topology is that any event processor can query the data it needs directly from the database, without having to synchronously communicate with any other event processors. This is a significant advantage, since event-driven architectures rely on highly decoupled event processors communicating through asynchronous communications. In Figure 15-34, the `Order Placement` event processor can simply query the central monolithic database to retrieve the number of items currently in stock and the customer's shipping options.

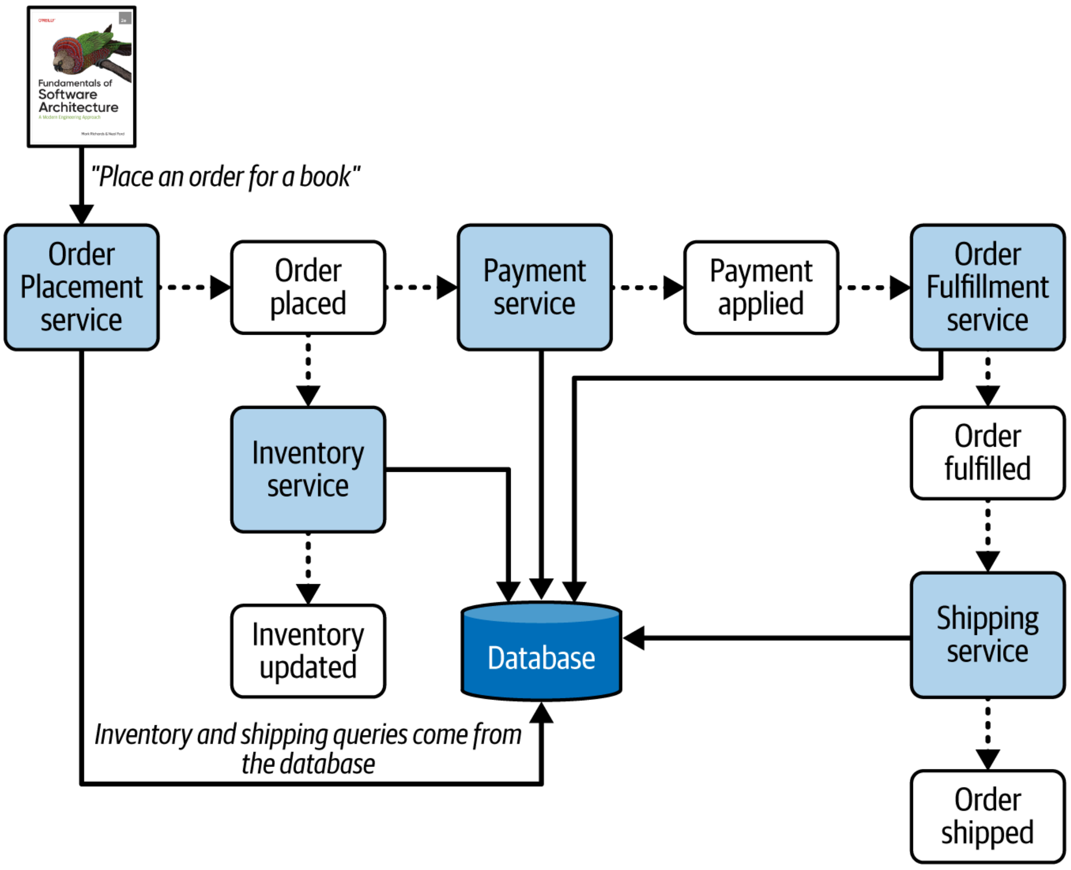


Figure 15-34. With the monolithic database topology, data is available directly from the database

While the monolithic database topology supports decoupling and limits communication between event processors, it carries with it some messy disadvantages, the first of which is fault tolerance. If the central monolithic database crashes or goes down for maintenance, all event processors become unavailable.

The second issue is scalability. Because event-driven architectures use asynchronous communication, each event processor can scale independently of the others. The event channel essentially acts as a backpressure point so that individual event processors can scale as necessary, regardless of whether other event processors scale as well. However, if all the event processors are simultaneously querying and writing to the same database, the *database* must scale to meet these demands. Many databases are unable to achieve this at high concurrency loads.

The third issue is change control. When the structure of the database changes (such as dropping a column or attribute), multiple event processors are affected and must coordinate, even for a single database change.

Finally, the monolithic database topology necessarily creates a single architectural quantum, due to the shared monolithic database.

Domain Database Topology

Another possible database topology within EDA is the *domain database topology*, which groups event processors into various domains, each owning its own database (Figure 15-35).

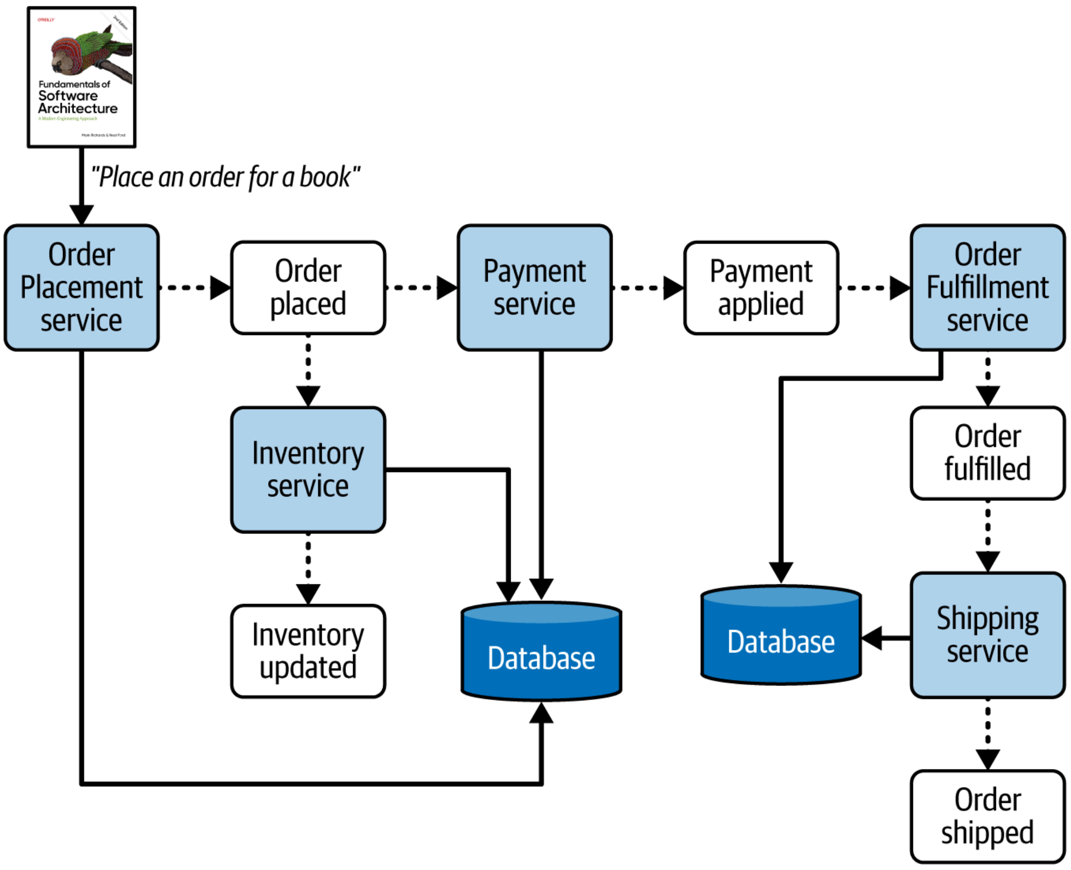


Figure 15-35. The domain database topology uses a separate database for each domain

The main advantages of the domain database topology over the monolithic database topology, which flow from being domain partitioned, are better fault tolerance, scalability, and change control. In the example shown in Figure 15-35, if the order-processing domain database (the one associated with the `Order Fulfillment` and `Order Shipping` event processors) crashes or becomes unavailable due to maintenance, the order placement domain is still fully operational and can continue to accept orders. The event channel containing the `payment applied` derived event acts as a backpressure point, queuing events until the order-processing database becomes available. The same is true with scalability and change control; each domain database only has to worry about scaling based on its domain's specific event processors, and only those domain-scoped event processors need to change if the database structure changes.

However, consider the two pieces of information needed by the `Order Placement` event processor: the number of books currently on hand and the shipping options. With the domain database topology, the `Order Placement` event processor can simply query its domain database to get the book inventory, similar to how it retrieved this information with the monolithic database topology. However, it must make a *synchronous* call to the `Order Shipping` event processor to retrieve the shipping options, thus synchronously coupling these services (see Figure 15-36).

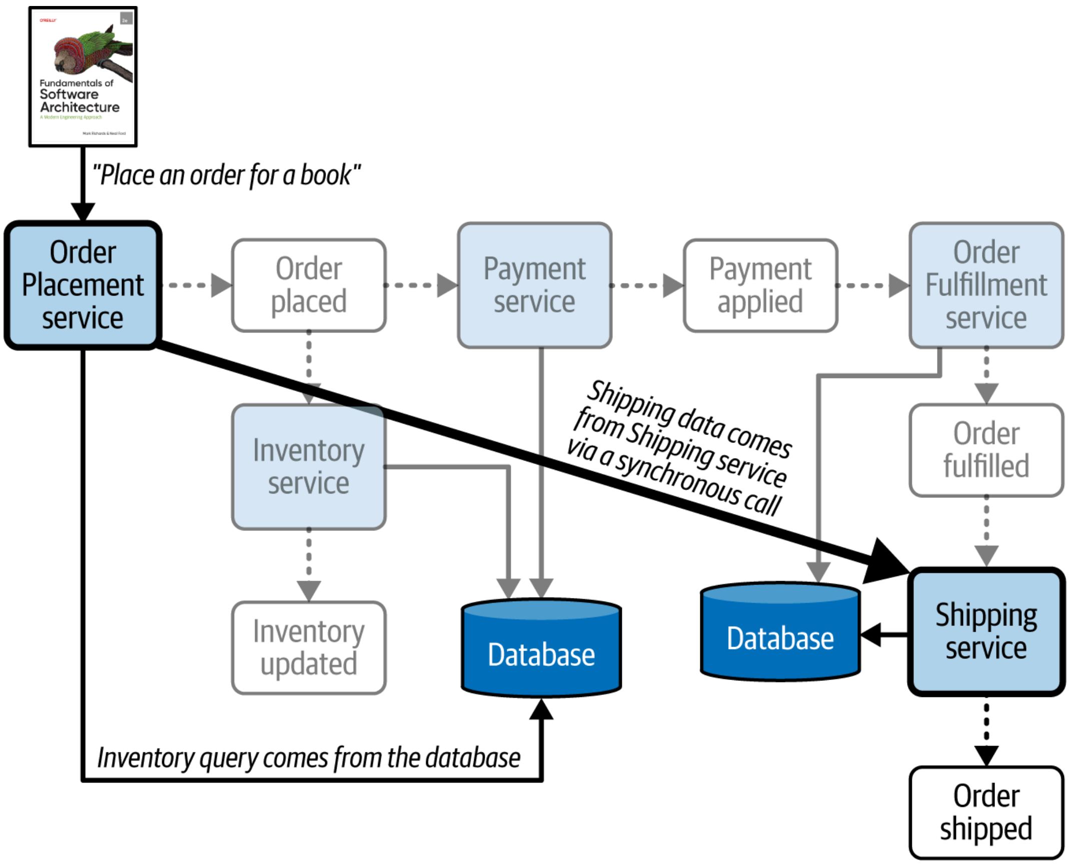

Fundamentals of Software Architecture
"Place an order for a book"
Order Placement service
Order placed
Payment service
Payment applied
Order Fulfillment service
Inventory service
Shipping data comes from Shipping service via a synchronous call
Order fulfilled
Inventory updated
Database
Database
Shipping service
Inventory query comes from the database
Order shipped

Figure 15-36. Data needed by the Order Placement event processor may require synchronous communication to an event processor in another domain

Architects should try to avoid synchronous coupling in a highly dynamic, decoupled architecture like EDA. Synchronous calls can take a toll on fault tolerance and scalability, negating many of the benefits of using this topology. Always check that the domains remain fairly independent from each other, and minimize synchronous interservice calls as much as possible. If too much synchronous communication between event processors is required, reevaluate the domain boundaries, combine the domains into a single one, or move to a monolithic database topology.

Dedicated Data Topology

Another viable option within EDA is the *dedicated database topology*, commonly referred to in the microservices world as the *database-per-service* pattern. With this database topology, each event processor owns its own dedicated database in a tightly formed bounded context, similar to microservices (see "Data Topologies" in Chapter 18). This topology is illustrated in Figure 15-37.

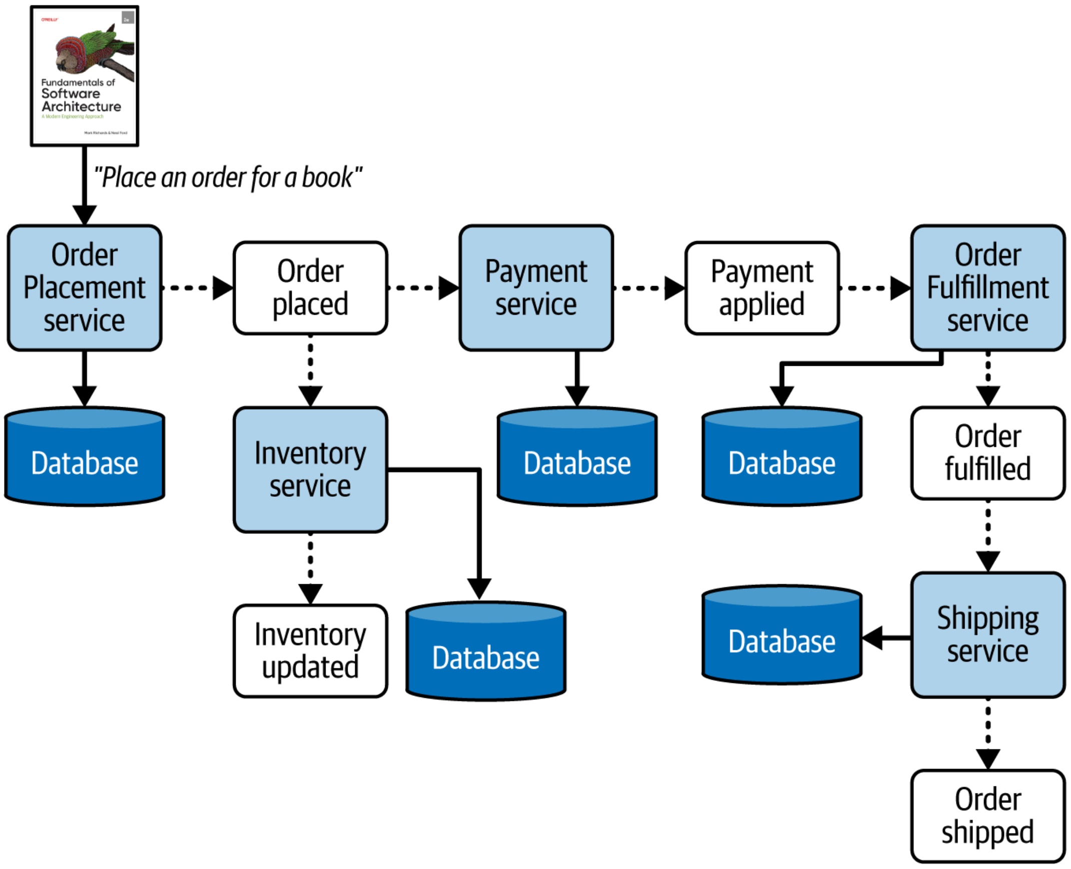


Figure 15-37. The dedicated database topology of EDA uses a separate database for each event processor

Not surprisingly, the dedicated database topology has the highest levels of fault tolerance, scalability, and change control out of all the available topologies. If an event processor or database goes down, the outage is isolated to just that event processor; all other event processors function normally. Databases only need to scale based on the single event processor within its bounded context, making this topology the most scalable of the three discussed in this section. Finally, changes to the database structure only impact the event processor attached to that database.

On the negative side, depending on the database technology stack, this can be a very expensive option. Perhaps the biggest disadvantage, however, is synchronous dynamic coupling between event processors. To illustrate this disadvantage, look again at the two pieces of information the `Order Placement` event processor needs for its processing: book inventory and shipping options. In this topology, the `Order Placement` event processor would need to make synchronous calls to *both* the `Inventory` event processor and the `Order Shipment` event processor to get the information it needs, forming tight synchronous coupling points throughout the architecture (as shown in Figure 15-38). As with the domain database topology, identify all data-related requirements in each event processor before selecting this database topology option.

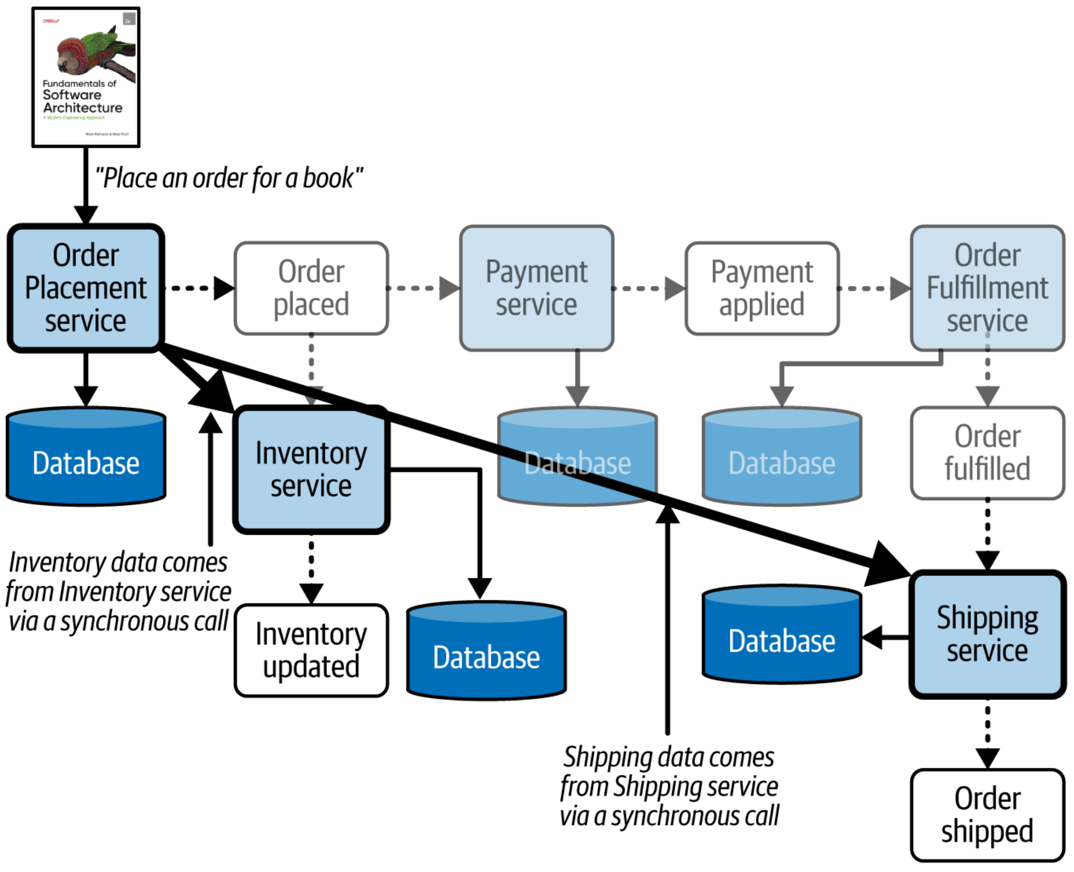

Fundamentals of Software Architecture
"Place an order for a book"
Order Placement service
Order placed
Payment service
Payment applied
Order Fulfillment service
Database
Inventory service
Database
Database
Order fulfilled
Inventory data comes from Inventory service via a synchronous call
Inventory updated
Database
Database
Shipping service
Shipping data comes from Shipping service via a synchronous call
Order shipped

Figure 15-38. Data needed by the `Order Placement` event processor may require synchronous communication to other event processors

The dedicated database topology is a good choice when event processors are mostly self-contained, only needing the data within its own bounded context and corresponding database. If event processors are communicating too much (see "Governance"), the architect should consider moving to the domain or even the monolithic database topology to improve overall performance and scalability. However, if the specific situation involves frequent structural changes to the database, that might warrant a trade-off between these operational characteristics to minimize the number of event processors affected.

Cloud Considerations

Event-driven architectures work well with cloud-based environments and implementations, primarily due to their highly decoupled nature. EDA can easily leverage the asynchronous services provided by cloud vendors, and the elastic nature of cloud infrastructure and cloud-based services matches its shape. Essentially, cloud-based environments are a good match for EDA.

Common Risks

One of the main risks associated with EDA is that its nondeterministic event processing can cause side effects: for instance, event processors unexpectedly triggering derived events or not responding to an event when they should. Event workflows can get very complex in an event-driven architecture, and many times it's difficult to know exactly what will happen when an event is triggered.

Another big risk is having too much static coupling (and hence brittleness) within the event-driven architecture. Although EDA is highly *dynamically* coupled, event payload contracts (see “Event Payload”) can also introduce tight *static* coupling. Changing a contract can be a daunting task, since architects don’t always know which event processors are responding to a particular event. When an event payload contract does change, it may negatively impact several other event processors, adding to the overall brittleness of this architectural style. Key-based event payloads help mitigate this risk, but the risks of this practice include issues with scalability and performance and the possibility of anemic events (see “Event Payload”).

Watch out for too much synchronous communication between event processors as well. Event-driven architecture gets its superpowers from its highly dynamically decoupled event processors. However, if event processors keep needing to communicate synchronously, it’s a good sign that EDA is not the most appropriate architectural style.

Finally, overall state management is both a risk and a challenge in EDA. It’s good to know when the initiating event has been completely processed, but that can be very hard to determine due to EDA’s nondeterministic, asynchronous parallel event processing. Occasionally an architect can identify the final processing endpoint and have the event processor that accepted the initiating event subscribe to that “ending” event, but in most cases this is hard to determine. As a result, it’s difficult to know when an initiating event has been fully processed, or even its current state.

Governance

A majority of the governance associated with EDA is nonstructural and requires observability in the form of logs as part of an overall governance mesh. Depending on the infrastructure and environment, some governance metrics associated with EDA may need to be manually collected.

The two main areas of governance in this style are static coupling through contract management, and dynamic coupling through synchronous calls. Both of these aspects are considered structural decay in EDA, so it’s important to keep an eye on them.

From a static coupling perspective, architects can set up governance around things like the rate of change of event payload contracts and overall stamp coupling. Changing contracts can be very risky in EDA because of its decoupled nature. Changing an event contract, particularly one with no associated schema, can break a downstream event processor. This is a particular risk in EDA because it’s so difficult to test nondeterministic end-to-end event flows.

Stamp coupling (see “Event Payload”) can be governed by continually recording and observing which fields in an event contract are not being used by event processors responding to the event. Observing these unused fields can help trim down contract size, reduce bandwidth, and help manage stamp coupling and corresponding unnecessary changes to event processors.

From a dynamic coupling standpoint, architects can write automated fitness functions to observe and track synchronous communication between event processors through logs and other observable means (such as source code annotations or use of standard synchronous custom-identifier libraries). *Any* synchronous communication in an event-driven architecture should be tracked and discussed to ensure that it is necessary, particularly if using a domain or dedicated database topology.

Team Topology Considerations

EDA is largely considered a technically partitioned architecture due to the many artifacts that make up each domain: multiple event processors, event channels, message brokers, and possibly databases, depending on the database topology. Nevertheless, it can work well when teams are aligned within domain areas (such as cross-functional teams with specialization). That said, specific types of team topologies (see "Team Topologies and Architecture") might find EDA challenging.

Here are some considerations regarding the alignment between these specific team topologies and EDA:

Stream-aligned teams

> Depending on the size of the system, stream-aligned teams might find themselves struggling to implement domain-based changes, due to the decoupled nature of event processors. Because domains and subdomains in EDA are typically implemented with multiple event processors and derived events, stream-aligned teams might find it a challenge to understand all of these moving parts. For example, adding a step to the order-placement workflow might require changing multiple event processors, as well as restructuring how (and when) the existing derived events are triggered. The larger and more complex the event-driven architecture, the less effective stream-aligned teams will be.

Enabling teams

> Because of the integration required between event processors based on derived events and their contracts, enabling teams don't work well in event-driven architectures. Experimentation and efficiency by enabling teams *within* a stream can disrupt a stream-aligned team's understanding and management of an overall event flow, and usually requires too much coordination between stream-aligned teams and enabling teams.

Complicated-subsystem teams

> Complicated-subsystem teams work well with EDA due to its decoupled, asynchronous nature. Complex processing can be easily isolated through separate event processors, and the complicated-subsystem team can focus on those, leaving less complicated processing to the stream-aligned teams. Because event processors are highly dynamically decoupled, stream-aligned teams only need to coordinate with complicated-subsystem teams for static event-payload contracts and derived events.

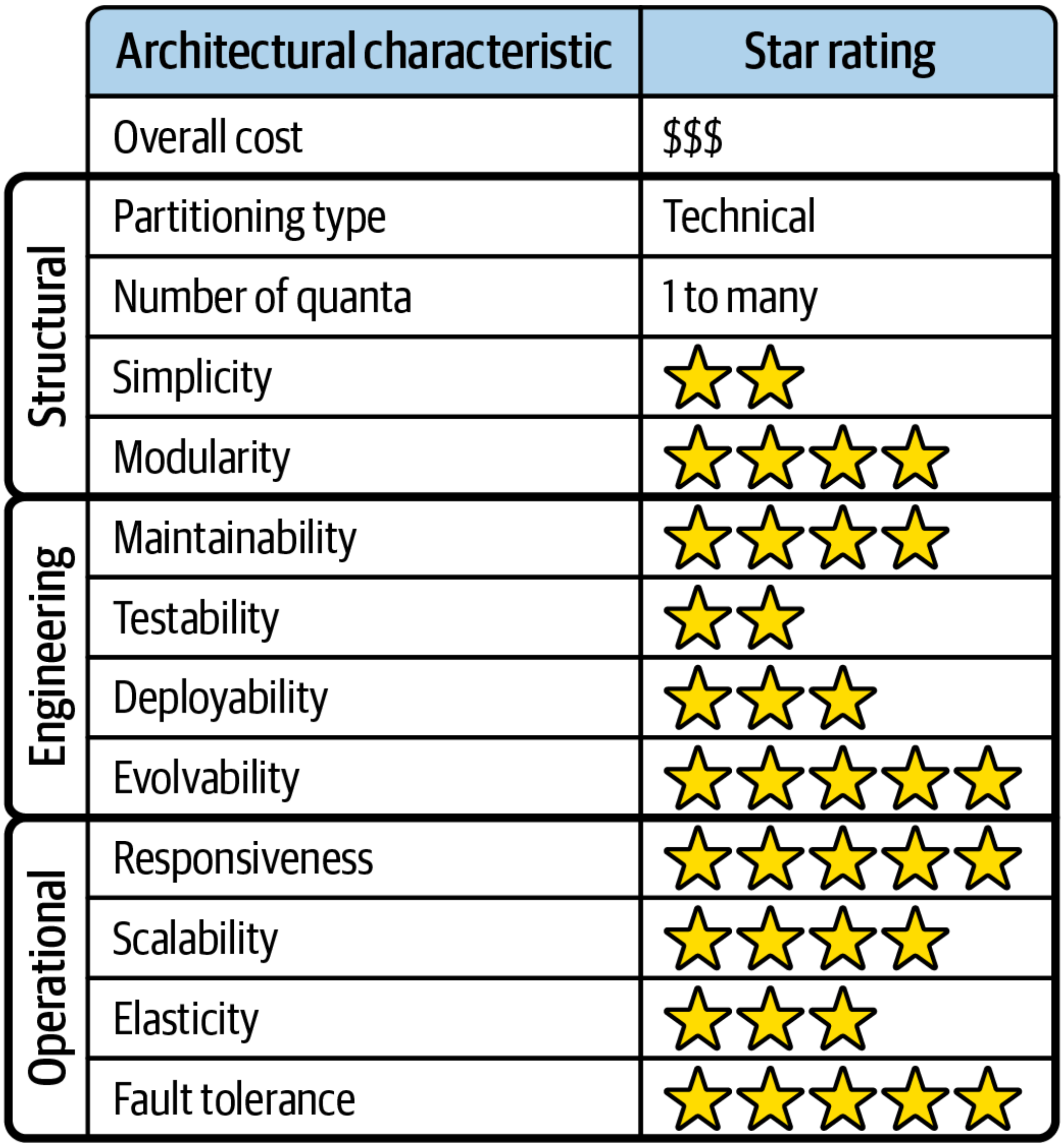
Platform teams

> In EDA, developers can leverage the benefits of the platform-teams topology by utilizing common tools, services, APIs, and tasks, primarily due to EDA's technical partitioning. This is especially true if teams treat the infrastructure-related parts of EDA (such as the message brokers, event hubs, event buses, and other event-channel artifacts) as platform related.

Style Characteristics

A one-star rating in the characteristics ratings table in Figure 15-39 means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means it's one of that style's strongest features. Definitions for each characteristic identified in the scorecard can be found in Chapter 4.

Event-driven architecture is primarily a technically partitioned architecture, in that any particular domain is spread across multiple event processors and tied together through brokers, contracts (event payload), and topics. Changes to a particular domain usually impact multiple event processors and other messaging artifacts, so EDA is not generally considered domain partitioned.



	Architectural characteristic	Star rating
	Overall cost	$$$
Structural	Partitioning type	Technical
	Number of quanta	1 to many
	Simplicity	★★
	Modularity	★★★★
Engineering	Maintainability	★★★★
	Testability	★★
	Deployability	★★★
	Evolvability	★★★★★
Operational	Responsiveness	★★★★★
	Scalability	★★★★
	Elasticity	★★★
	Fault tolerance	★★★★★

Figure 15-39. EDA characteristics ratings

The number of quanta within EDA can vary from one to many, depending on the database interactions within each event processor and whether the system uses request-reply processing. Even though communication in an EDA relies on asynchronous calls, if multiple event processors share a single database instance, they would all be contained within the same architectural quantum. The same is true for request-reply processing: even though the communication between the event processors is still asynchronous, if a response is needed right away from the event consumer, it ties those event processors together synchronously, forming a single quantum.

For example, imagine that one event processor sends a request to another event processor to place an order. The first event processor must wait for an order ID from the other event processor to continue. If the second event processor (the one that places the order and generates an order ID) is down, the first event processor cannot continue. This means they are part of the same architecture quantum and share the same architectural characteristics, even though they are both sending and receiving asynchronous messages.

Event-driven architecture rates very high (4 to 5 stars) for performance, scalability, and fault tolerance, its primary strengths. Its high performance is achieved by combining asynchronous communications with highly parallel processing. High scalability is realized through the programmatic load balancing of event processors (also called *competing consumers* and *consumer groups*). As the request load increases, additional event processors can be programmatically added to handle the additional requests. The reason we only gave it four stars (rather than five) is because of the database (see Chapter 16, on space-based architectures, for an example of a five-star rating for these characteristics). EDA achieves fault tolerance through its decoupled, asynchronous event processors, which provide eventual consistency and processing of event workflows. If other downstream processors are unavailable, as long as the user interface or an event processor making a request does not need an immediate response, the system can process the event at a later time.

EDA rates relatively low on overall *simplicity* and *testability*, mostly due to its nondeterministic and dynamic event flows. In request-based models, deterministic flows are relatively easy to test because their paths and outcomes are generally known. Such is not the case with the event-driven model. Sometimes architects just don’t know how event processors will react to dynamic events or what messages they might produce. These are known as *nondeterministic workflows*. These systems’ “event tree diagrams” can be extremely complex, generating hundreds or even thousands of scenarios, making it very difficult to govern and test.

Finally, EDAs are highly evolutionary, hence the five-star rating. Adding new features through existing or new event processors is relatively straightforward. By providing hooks via triggered derived events, the event and its corresponding data are already available for other processing, so no changes are required in the infrastructure or existing event processors to add that new functionality

EDA’s disadvantages include difficulty controlling the overall workflow associated with an initiating event. Event processing is very dynamic due to changing conditions, and it’s difficult to know when the business transaction based on the initiating event has been completed.

Error handling is also a big challenge with EDA. Because there is typically no mediator monitoring or controlling the business transaction (except with the mediator topology), if a failure occurs, other services will be unaware of the crash. The business process based on that initiating event gets stuck and is unable to move without some sort of automated or manual intervention, even as all other processes move along without regard for the error. For example, if the `Payment` event processor in our ordering system crashes and does not complete its assigned task, the `Inventory` event processor still adjusts the inventory, and all other further event processors react as though everything is fine.

Restarting a business transaction (recoverability) is very difficult to do in EDA. Because other actions have been taken asynchronously through the processing of the initiating event, many times it’s simply not feasible to resubmit the initiating event.

Choosing Between Request-Based and Event-Based

Models

The request-based and event-based models are both viable approaches for designing software systems. However, choosing the right model is essential to success. We recommend choosing the request-based model for well-structured, data-driven requests (such as retrieving customer profile data), when the priority is certainty and control over the workflow. We recommend choosing the event-based model for flexible, action-based events that require high levels of responsiveness and scale, with complex and dynamic user processing.

Understanding the event-based model's trade-offs also helps in determining the best fit. Table 15-2 lists the advantages and disadvantages of the event-based model of EDA.

Table 15-2. Trade-offs of the event-driven model

Advantages over request-based	Trade-offs
Better response to dynamic user content	Only supports eventual consistency
Better scalability and elasticity	Less control over processing flow
Better agility and change management	Less certainty over outcome of event flow
Better adaptability and extensibility	Difficult to test and debug
Better responsiveness and performance	
Better real-time decision making	
Better reaction to situational awareness	

Examples and Use Cases

Any business problem focused on responding to things happening in the system (either internal or external) is a good candidate for EDA. The order-entry system example we've used throughout this chapter is a good use case for EDA in that it allows for parallel, decoupled processing of an order. Systems that require high levels of responsiveness, performance, scalability, fault tolerance, and elasticity are also great candidates for EDA.

Another good use case to demonstrate the power and effectiveness of EDA is our ongoing Going, Going, Gone auction-system example, where users bid on items put up for auction until a final bid goes uncontested and the bidder wins that item. In these systems, the number of bidders is usually unknown, requiring both scalability and elasticity—particularly if the auction is timed and the bidding comes to a close. These systems also need high levels of responsiveness. Perhaps the best reason that online bidding systems and EDA are a good match, though, is that EDA regards placing a bid not as a *request made to the system*, but as an *event that has happened*.

As Figure 15-40 shows, lots of actions take place in the system when a bidder places a bid. These actions can all be asynchronous, done either at the same time or later, as backend processing (such as the `Bidder Tracker` event processor). When a bidder places a bid, the `Bid Capture` event processor receives that initiating event, determines if it's higher than the prior bid, and triggers a `bid placed` event. The `Auctioneer` event processor responds to that event and updates the website with the item's new bid price. Simultaneously, the `Bid Streamer` event processor responds to the same event, streaming the bid out to the website bid history or even to the individual bidders (depending on the UI). Finally, the `Bidder`

`Tracker` event processor responds to the event to persist the bidder and their bid for tracking and auditing purposes.

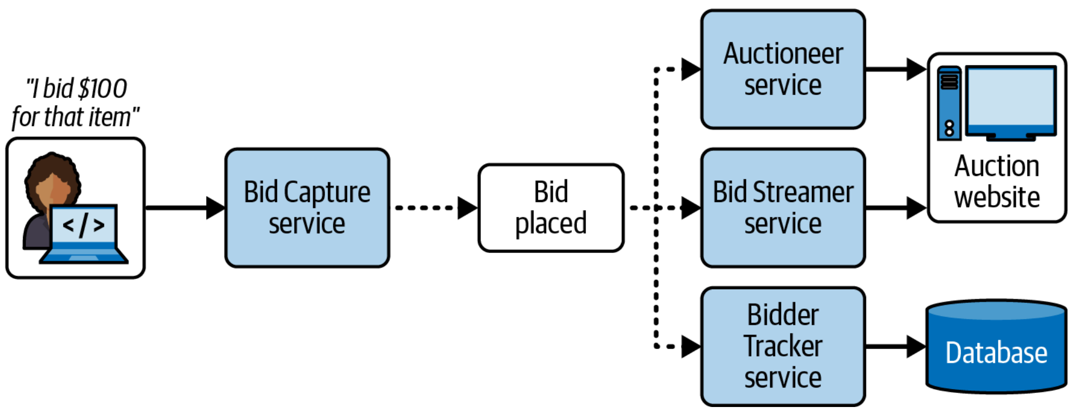


Figure 15-40. An online bidding system example using EDA

Many other event processors are involved in the workflow of this initiating event, and many other derived events are triggered, but this small part of the system demonstrates a good use of the event-driven architectural style, illustrating responsiveness, fault tolerance, scalability, and elasticity.

Event-driven architecture is a very complicated architectural style, but also a very powerful one. Closely analyze the workflows and processing needed for the business problem to determine if dealing with EDA's complexity is worthwhile given its superpowers. If a majority of the processing needed is request based, consider the microservices architectural style (discussed in Chapter 18) instead.