Chapter 11. Pipeline Architecture Style

One of the fundamental styles in software architecture that appears again and again is the *pipeline* architecture (also known as the *pipes and filters* architecture). As soon as developers and architects decided to split functionality into discrete parts, this pattern followed. Most developers know this architecture as this underlying principle behind Unix terminal shell languages, such as Bash and Zsh.

Developers in many functional programming languages will see parallels between language constructs and elements of this architecture. In fact, many tools that utilize the MapReduce programming model follow this basic topology. While these examples show a low-level implementation of the pipeline architecture style, it can also be used for higher-level business applications.

Topology

The topology of the pipeline architecture consists of pipes and filters, illustrated in Figure 11-1.

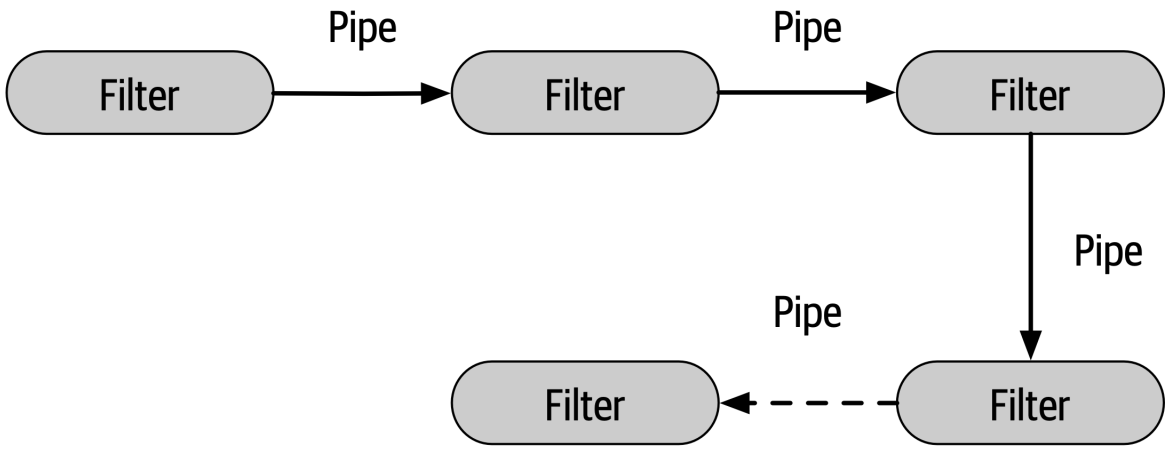

Figure 11-1. Basic topology for pipeline architecture

The pipes and filters coordinate in a specific fashion, with pipes forming one-way communication between filters, usually in a point-to-point fashion.

Pipes

Pipes in this architecture form the communication channel between filters. Each pipe is typically unidirectional and point-to-point (rather than broadcast) for performance reasons, accepting input from one source and always directing output to another. The payload carried on the pipes may be any data format, but architects favor smaller amounts of data to enable high performance.

Filters

Filters are self-contained, independent from other filters, and generally stateless. Filters should perform one task only. Composite tasks should be handled by a sequence of filters rather than a single one.

Four types of filters exist within this architecture style:

Producer

> The starting point of a process, outbound only, sometimes called the *source*.

Transformer

Accepts input, optionally performs a transformation on some or all of the data, then forwards it to the outbound pipe. Functional advocates will recognize this feature as *map*.

Tester

Accepts input, tests one or more criteria, then optionally produces output, based on the test. Functional programmers will recognize this as similar to *reduce*.

Consumer

The termination point for the pipeline flow. Consumers sometimes persist the final result of the pipeline process to a database, or they may display the final results on a user interface screen.

The unidirectional nature and simplicity of each of the pipes and filters encourages compositional reuse. Many developers have discovered this ability using shells. A famous story from the blog "More Shell, Less Egg" illustrates just how powerful these abstractions are. Donald Knuth was asked to write a program to solve this text handling problem: read a file of text, determine the *n* most frequently used words, and print out a sorted list of those words along with their frequencies. He wrote a program consisting of more than 10 pages of Pascal, designing (and documenting) a new algorithm along the way. Then, Doug McIlroy demonstrated a shell script that would easily fit within a Twitter post that solved the problem more simply, elegantly, and understandably (if you understand shell commands):

```
tr -cs A-Za-z '\n' |
tr A-Z a-z |
sort |
uniq -c |
sort -rn |
sed ${1}q
```

Even the designers of Unix shells are often surprised at the inventive uses developers have wrought with their simple but powerfully composite abstractions.

Example

The pipeline architecture pattern appears in a variety of applications, especially tasks that facilitate simple, one-way processing. For example, many Electronic Data Interchange (EDI) tools use this pattern, building transformations from one document type to another using pipes and filters. ETL tools (extract, transform, and load) leverage the pipeline architecture as well for the flow and modification of data from one database or data source to another. Orchestrators and mediators such as Apache Camel utilize the pipeline architecture to pass information from one step in a business process to another.

To illustrate how the pipeline architecture can be used, consider the following example, as illustrated in Figure 11-2, where various service telemetry information is sent from services via streaming to Apache Kafka.

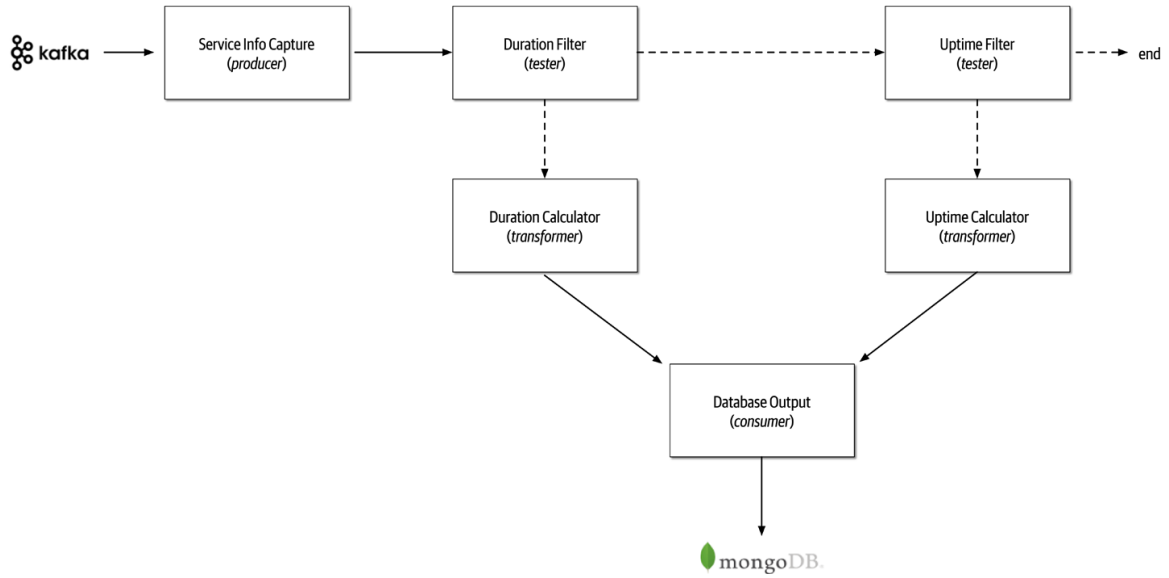

Figure 11-2. Pipeline architecture example

Notice in Figure 11-2 the use of the pipeline architecture style to process the different kinds of data streamed to Kafka. The `Service Info Capture` filter (producer filter) subscribes to the Kafka topic and receives service information. It then sends this captured data to a tester filter called `Duration Filter` to determine whether the data captured from Kafka is related to the duration (in milliseconds) of the service request. Notice the

separation of concerns between the filters; the `Service Metrics Capture` filter is only concerned about how to connect to a Kafka topic and receive streaming data, whereas the `Duration Filter` is only concerned about qualifying the data and optionally routing it to the next pipe. If the data is related to the duration (in milliseconds) of the service request, then the `Duration Filter` passes the data on to the `Duration Calculator` transformer filter. Otherwise, it passes it on to the `Uptime Filter` tester filter to check if the data is related to uptime metrics. If it is not, then the pipeline ends—the data is of no interest to this particular processing flow. Otherwise, if it is uptime metrics, it then passes the data along to the `Uptime Calculator` to calculate the uptime metrics for the service. These transformers then pass the modified data to the `Database Output` consumer, which then persists the data in a MongoDB database.

This example shows the extensibility properties of the pipeline architecture. For example, in Figure 11-2, a new tester filter could easily be added after the `Uptime Filter` to pass the data on to another newly gathered metric, such as the database connection wait time.

Architecture Characteristics Ratings

A one-star rating in the characteristics ratings table Figure 11-3 means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in Chapter 4.

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★★
Elasticity	★
Evolutionary	★★★
Fault tolerance	★
Modularity	★★★
Overall cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★★

Figure 11-3. Pipeline architecture characteristics ratings

The pipeline architecture style is a technically partitioned architecture due to the partitioning of application logic into filter types (producer, tester, transformer, and consumer). Also, because the pipeline architecture is usually implemented as a monolithic deployment, the architectural quantum is always one.

Overall cost and simplicity combined with modularity are the primary strengths of the pipeline architecture style. Being monolithic in nature,

pipeline architectures don’t have the complexities associated with distributed architecture styles, are simple and easy to understand, and are relatively low cost to build and maintain. Architectural modularity is achieved through the separation of concerns between the various filter types and transformers. Any of these filters can be modified or replaced without impacting the other filters. For instance, in the Kafka example illustrated in Figure 11-2, the `Duration Calculator` can be modified to change the duration calculation without impacting any other filter.

Deployability and testability, while only around average, rate slightly higher than the layered architecture due to the level of modularity achieved through filters. That said, this architecture style is still a monolith, and as such, ceremony, risk, frequency of deployment, and completion of testing still impact the pipeline architecture.

Like the layered architecture, overall reliability rates medium (three stars) in this architecture style, mostly due to the lack of network traffic, bandwidth, and latency found in most distributed architectures. We only gave it three stars for reliability because of the nature of the monolithic deployment of this architecture style in conjunction with testability and deployability issues (such as having to test the entire monolith and deploy the entire monolith for any given change).

Elasticity and scalability rate very low (one star) for the pipeline architecture, primarily due to monolithic deployments. Although it is possible to make certain functions within a monolith scale more than others, this effort usually requires very complex design techniques such as multithreading, internal messaging, and other parallel processing practices, techniques this architecture isn’t well suited for. However, because the pipeline architecture is always a single system quantum due to the monolithic user interface, backend processing, and monolithic database, applications can only scale to a certain point based on the single architecture quantum.

Pipeline architectures don’t support fault tolerance due to monolithic deployments and the lack of architectural modularity. If one small part of a

pipeline architecture causes an out-of-memory condition to occur, the entire application unit is impacted and crashes. Furthermore, overall availability is impacted due to the high mean time to recovery (MTTR) usually experienced by most monolithic applications, with startup times ranging anywhere from 2 minutes for smaller applications, up to 15 minutes or more for most large applications.

Chapter 12. Microkernel Architecture Style

The *microkernel* architecture style (also referred to as the *plug-in* architecture) was coined several decades ago and is still widely used today. This architecture style is a natural fit for product-based applications (packaged and made available for download and installation as a single, monolithic deployment, typically installed on the customer's site as a third-party product) but is widely used in many nonproduct custom business applications as well.

Topology

The microkernel architecture style is a relatively simple monolithic architecture consisting of two architecture components: a core system and plug-in components. Application logic is divided between independent plug-in components and the basic core system, providing extensibility, adaptability, and isolation of application features and custom processing logic. Figure 12-1 illustrates the basic topology of the microkernel architecture style.

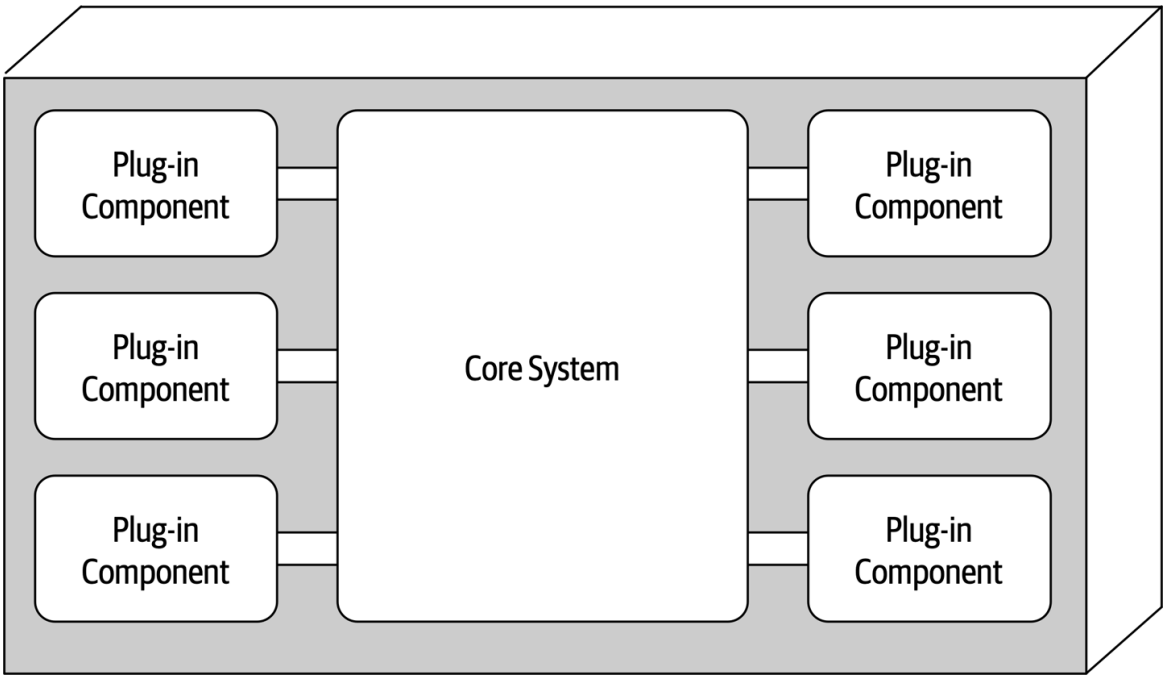

Figure 12-1. Basic components of the microkernel architecture style

Core System

The *core system* is formally defined as the minimal functionality required to run the system. The Eclipse IDE is a good example of this. The core system of Eclipse is just a basic text editor: open a file, change some text, and save the file. It's not until you add plug-ins that Eclipse starts becoming a usable product. However, another definition of the core system is the happy path (general processing flow) through the application, with little or no custom processing. Removing the cyclomatic complexity of the core system and placing it into separate plug-in components allows for better extensibility and maintainability, as well as increased testability. For example, suppose an electronic device recycling application must perform specific custom assessment rules for each electronic device received. The Java code for this sort of processing might look as follows:

```
public void assessDevice(String deviceID) {
   if (deviceID.equals("iPhone6s")) {
      assessiPhone6s();
   } else if (deviceID.equals("iPad1"))
      assessiPad1();
   } else if (deviceID.equals("Galaxy5"))
```

```
        assessGalaxy5();
    } else ...
        ...
    }
}
```

Rather than placing all this client-specific customization in the core system with lots of cyclomatic complexity, it is much better to create a separate plug-in component for each electronic device being assessed. Not only do specific client plug-in components isolate independent device logic from the rest of the processing flow, but they also allow for expandability. Adding a new electronic device to assess is simply a matter of adding a new plug-in component and updating the registry. With the microkernel architecture style, assessing an electronic device only requires the core system to locate and invoke the corresponding device plug-ins as illustrated in this revised source code:

```
public void assessDevice(String deviceID) {
        String plugin = pluginRegistry.get(deviceID);
        Class<?> theClass = Class.forName(plugin);
        Constructor<?> constructor = theClass.getConstructor();
        DevicePlugin devicePlugin =
                (DevicePlugin)constructor.newInstance();
        DevicePlugin.assess();
}
```

In this example all of the complex rules and instructions for assessing a particular electronic device are self-contained in a standalone, independent plug-in component that can be generically executed from the core system.

Depending on the size and complexity, the core system can be implemented as a layered architecture or a modular monolith (as illustrated in Figure 12-2). In some cases, the core system can be split into separately deployed domain services, with each domain service containing specific plug-in components specific to that domain. For example, suppose `Payment Processing` is the domain service representing the core system. Each payment method (credit card, PayPal, store credit, gift card, and purchase order) would be separate plug-in components specific to the payment

domain. In all of these cases, it is typical for the entire monolithic application to share a single database.

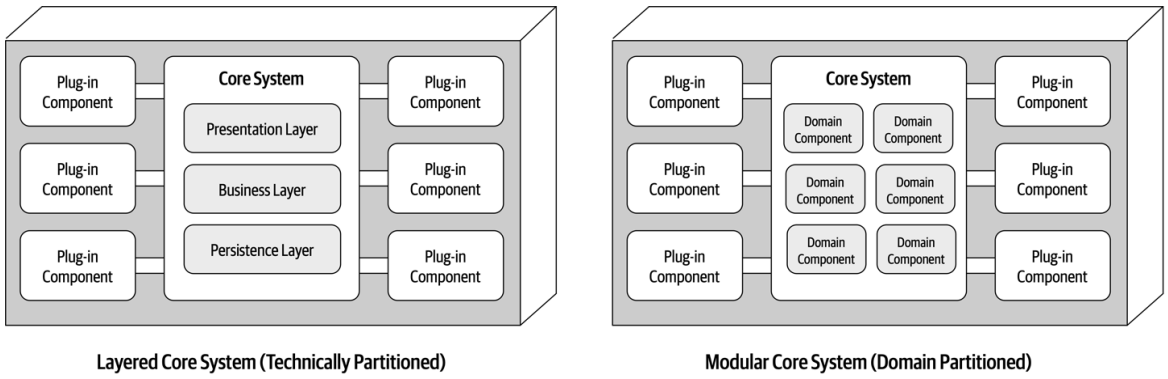

Figure 12-2. Variations of the microkernel architecture core system

The presentation layer of the core system can be embedded within the core system or implemented as a separate user interface, with the core system providing backend services. As a matter of fact, a separate user interface can also be implemented as a microkernel architecture style. Figure 12-3 illustrates these presentation layer variants in relation to the core system.

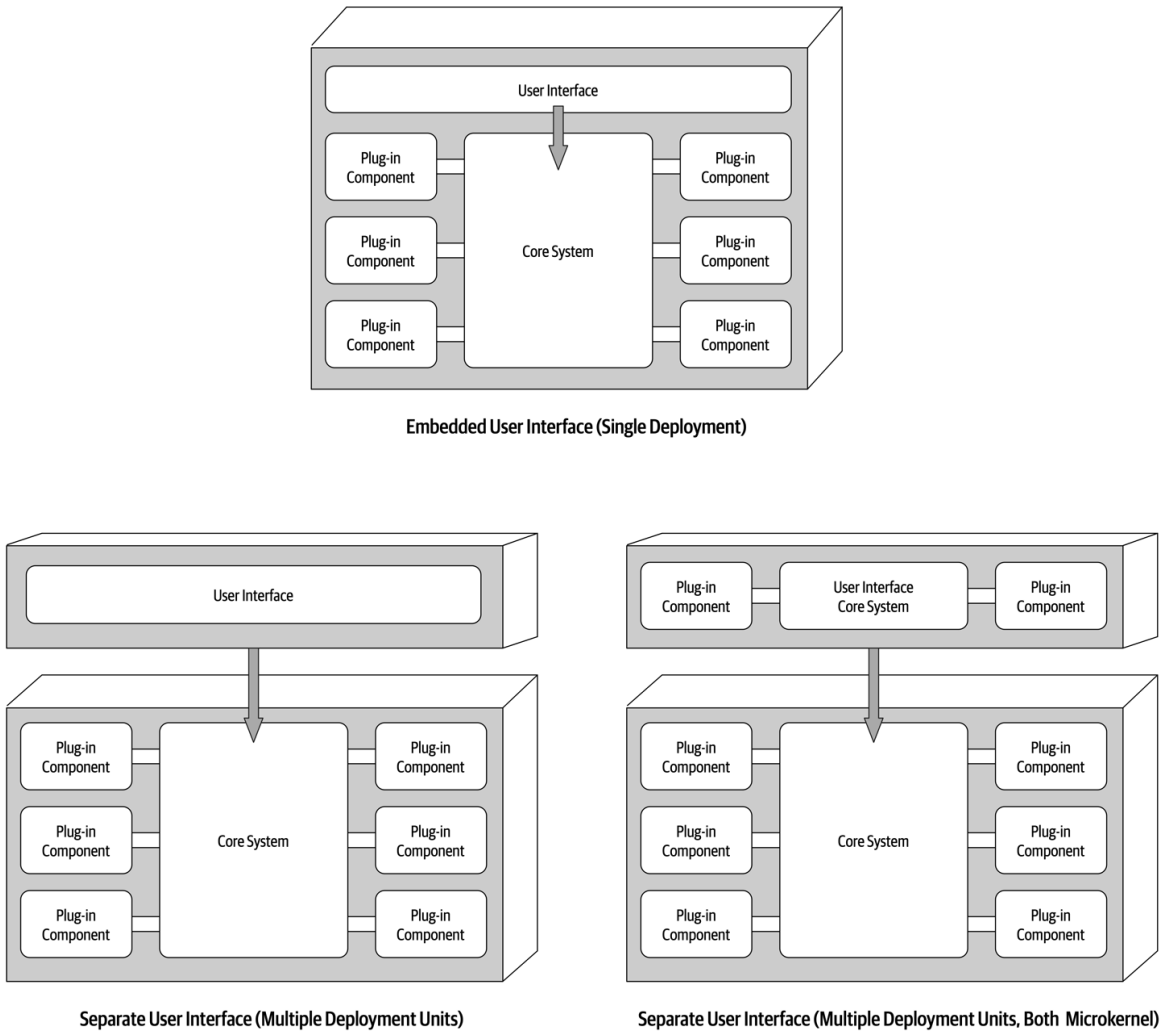

Figure 12-3. User interface variants

Plug-In Components

Plug-in components are standalone, independent components that contain specialized processing, additional features, and custom code meant to enhance or extend the core system. Additionally, they can be used to isolate highly volatile code, creating better maintainability and testability within the application. Ideally, plug-in components should be independent of each other and have no dependencies between them.

The communication between the plug-in components and the core system is generally point-to-point, meaning the “pipe” that connects the plug-in to the core system is usually a method invocation or function call to the entry-

point class of the plug-in component. In addition, the plug-in component can be either compile-based or runtime-based. Runtime plug-in components can be added or removed at runtime without having to redeploy the core system or other plug-ins, and they are usually managed through frameworks such as Open Service Gateway Initiative (OSGi) for Java, Penrose (Java), Jigsaw (Java), or Prism (.NET). Compile-based plug-in components are much simpler to manage but require the entire monolithic application to be redeployed when modified, added, or removed.

Point-to-point plug-in components can be implemented as shared libraries (such as a JAR, DLL, or Gem), package names in Java, or namespaces in C#. Continuing with the electronics recycling assessment application example, each electronic device plug-in can be written and implemented as a JAR, DLL, or Ruby Gem (or any other shared library), with the name of the device matching the name of the independent shared library, as illustrated in Figure 12-4.

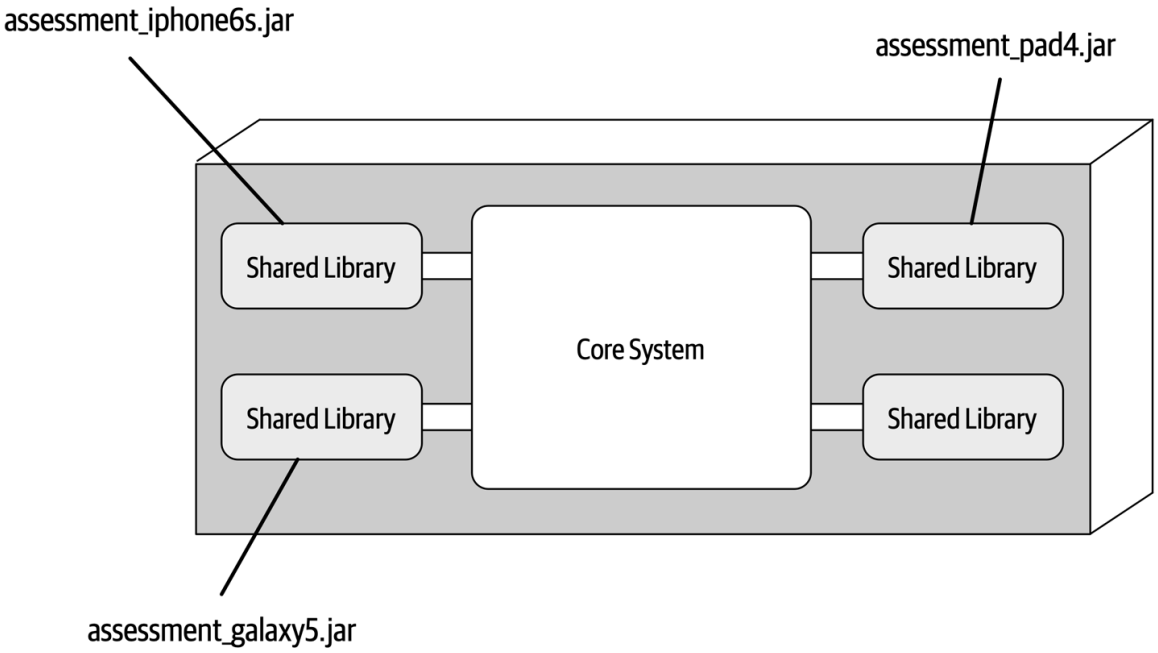

Figure 12-4. Shared library plug-in implementation

Alternatively, an easier approach shown in Figure 12-5 is to implement each plug-in component as a separate namespace or package name within the same code base or IDE project. When creating the namespace, we

recommend the following semantics: `app.plug-in.<domain>.<context>`. For example, consider the namespace `app.plugin.assessment.iphone6s`. The second node (`plugin`) makes it clear this component is a plug-in and therefore should strictly adhere to the basic rules regarding plug-in components (namely, that they are self-contained and separate from other plug-ins). The third node describes the domain (in this case, `assessment`), thereby allowing plug-in components to be organized and grouped by a common purpose. The fourth node (`iphone6s`) describes the specific context for the plug-in, making it easy to locate the specific device plug-in for modification or testing.

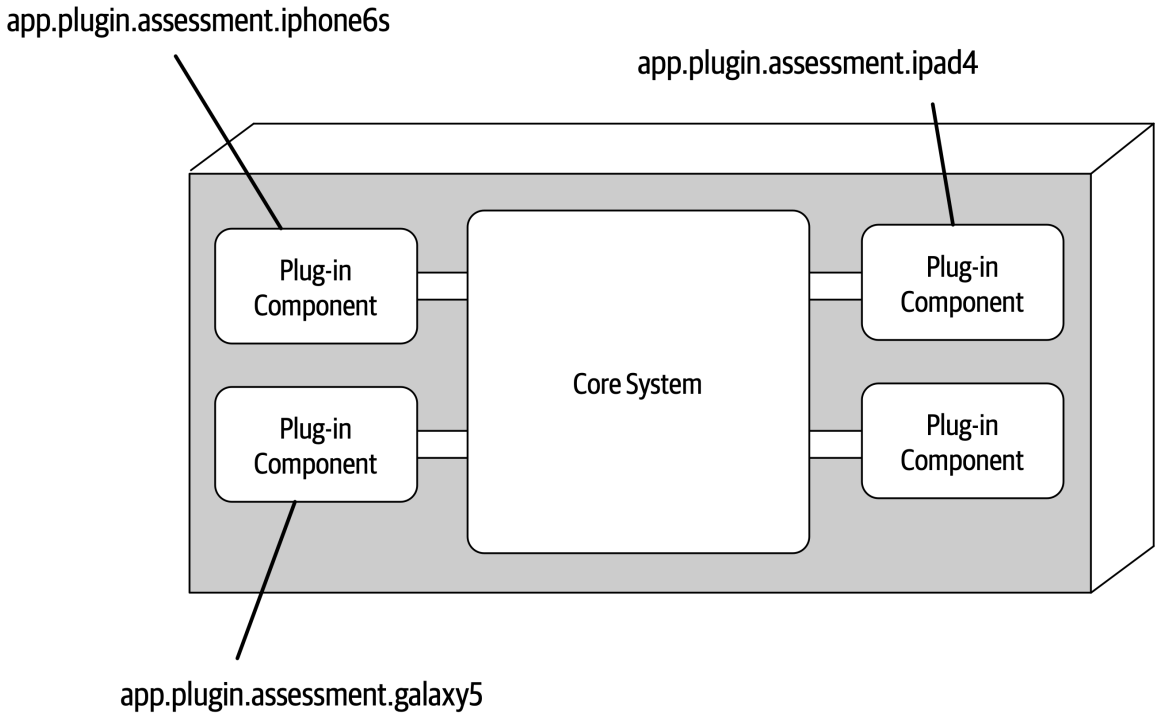

Figure 12-5. Package or namespace plug-in implementation

Plug-in components do not always have to be point-to-point communication with the core system. Other alternatives exist, including using REST or messaging as a means to invoke plug-in functionality, with each plug-in being a standalone service (or maybe even a microservice implemented using a container). Although this may sound like a good way to increase overall scalability, note that this topology (illustrated in Figure 12-6) is still

only a single architecture quantum due to the monolithic core system. Every request must first go through the core system to get to the plug-in service.

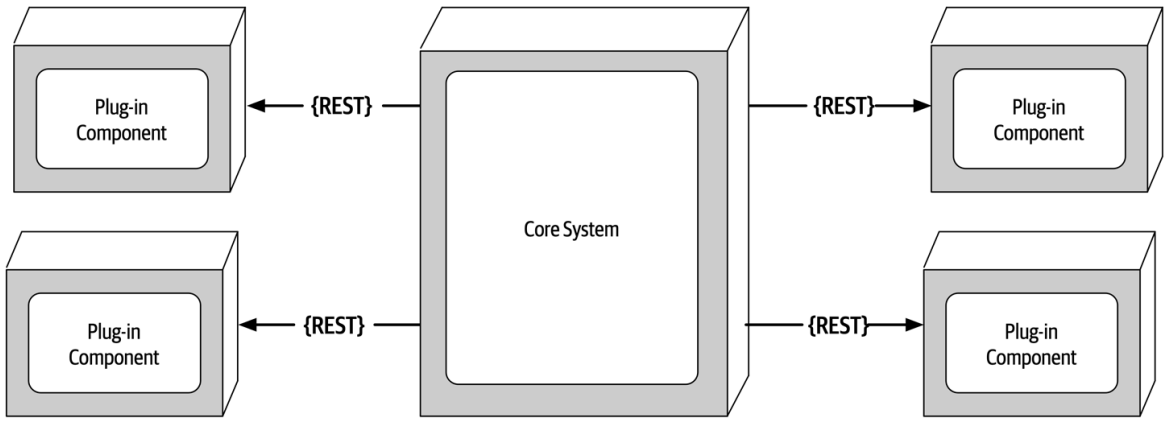

Figure 12-6. Remote plug-in access using REST

The benefits of the remote access approach to accessing plug-in components implemented as individual services is that it provides better overall component decoupling, allows for better scalability and throughput, and allows for runtime changes without any special frameworks like OSGi, Jigsaw, or Prism. It also allows for asynchronous communications to plug-ins, which, depending on the scenario, could significantly improve overall user responsiveness. Using the electronics recycling example, rather than having to wait for the electronic device assessment to run, the core system could make an asynchronous *request* to kick off an assessment for a particular device. When the assessment completes, the plug-in can notify the core system through another asynchronous messaging channel, which in turn would notify the user that the assessment is complete.

With these benefits comes trade-offs. Remote plug-in access turns the microkernel architecture into a distributed architecture rather than a monolithic one, making it difficult to implement and deploy for most third-party on-prem products. Furthermore, it creates more overall complexity and cost and complicates the overall deployment topology. If a plug-in becomes unresponsive or is not running, particularly when using REST, the request cannot be completed. This would not be the case with a monolithic deployment. The choice of whether to make the communication to plug-in components from the core system point-to-point or remote should be based

on specific requirements and thus requires a careful trade-off analysis of the benefits and drawbacks of such an approach.

It is not a common practice for plug-in components to connect directly to a centrally shared database. Rather, the core system takes on this responsibility, passing whatever data is needed into each plug-in. The primary reason for this practice is decoupling. Making a database change should only impact the core system, not the plug-in components. That said, plug-ins can have their own separate data stores only accessible to that plug-in. For example, each electronic device assessment plug-in in the electronic recycling system example can have its own simple database or rules engine containing all of the specific assessment rules for each product. The data store owned by the plug-in component can be external (as shown in Figure 12-7), or it could be embedded as part of the plug-in component or monolithic deployment (as in the case of an in-memory or embedded database).

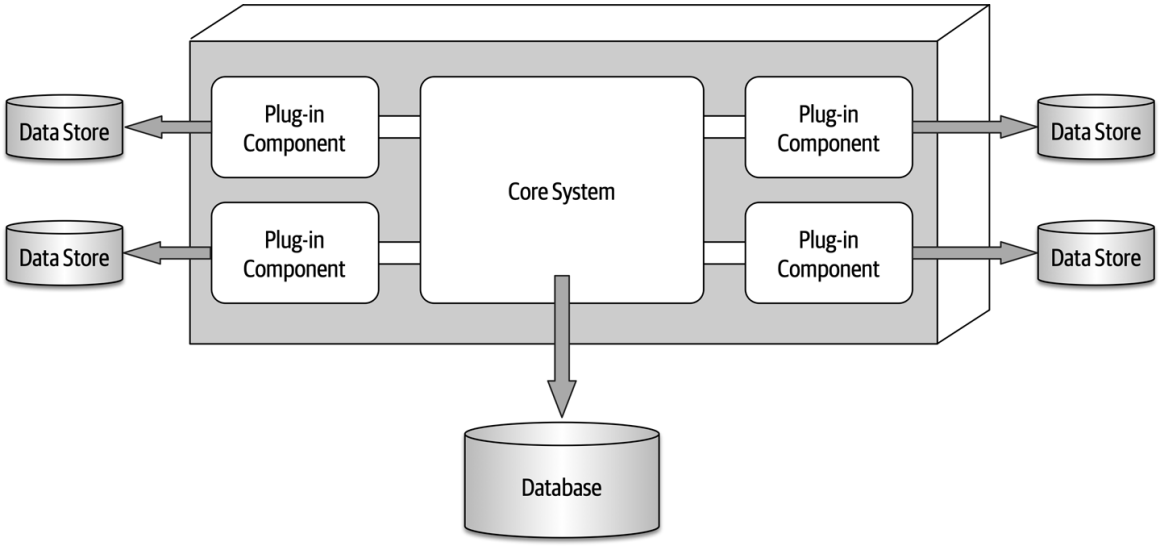

Figure 12-7. Plug-in components can own their own data store

Registry

The core system needs to know about which plug-in modules are available and how to get to them. One common way of implementing this is through a plug-in registry. This registry contains information about each plug-in

module, including things like its name, data contract, and remote access protocol details (depending on how the plug-in is connected to the core system). For example, a plug-in for tax software that flags high-risk tax audit items might have a registry entry that contains the name of the service (AuditChecker), the data contract (input data and output data), and the contract format (XML).

The registry can be as simple as an internal map structure owned by the core system containing a key and the plug-in component reference, or it can be as complex as a registry and discovery tool either embedded within the core system or deployed externally (such as Apache ZooKeeper or Consul). Using the electronics recycling example, the following Java code implements a simple registry within the core system, showing a point-to-point entry, a messaging entry, and a RESTful entry example for assessing an iPhone 6S device:

```
Map<String, String> registry = new HashMap<String, String>();
static {
  //point-to-point access example
  registry.put("iPhone6s", "Iphone6sPlugin");

  //messaging example
  registry.put("iPhone6s", "iphone6s.queue");

  //restful example
  registry.put("iPhone6s", "https://atlas:443/assess/iphone6s");
}
```

Contracts

The contracts between the plug-in components and the core system are usually standard across a domain of plug-in components and include behavior, input data, and output data returned from the plug-in component. Custom contracts are typically found in situations where plug-in components are developed by a third party where you have no control over the contract used by the plug-in. In such cases, it is common to create an

adapter between the plug-in contact and your standard contract so that the core system doesn't need specialized code for each plug-in.

Plug-in contracts can be implemented in XML, JSON, or even objects passed back and forth between the plug-in and the core system. In keeping with the electronics recycling application, the following contract (implemented as a standard Java interface named `AssessmentPlugin`) defines the overall behavior (`assess()`, `register()`, and `deregister()`), along with the corresponding output data expected from the plug-in component (`AssessmentOutput`):

```
public interface AssessmentPlugin {
        public AssessmentOutput assess();
        public String register();
        public String deregister();
}

public class AssessmentOutput {
        public String assessmentReport;
        public Boolean resell;
        public Double value;
        public Double resellPrice;
}
```

In this contract example, the device assessment plug-in is expected to return the assessment report as a formatted string; a resell flag (true or false) indicating whether this device can be resold on a third-party market or safely disposed of; and finally, if it can be resold (another form of recycling), what the calculated value is of the item and what the recommended resell price should be.

Notice the roles and responsibility model between the core system and the plug-in component in this example, specifically with the `assessmentReport` field. It is not the responsibility of the core system to format and understand the details of the assessment report, only to either print it out or display it to the user.

Examples and Use Cases

Most of the tools used for developing and releasing software are implemented using the microkernel architecture. Some examples include the Eclipse IDE, PMD, Jira, and Jenkins, to name a few). Internet web browsers such as Chrome and Firefox are another common product example using the microkernel architecture: viewers and other plug-ins add additional capabilities that are not otherwise found in the basic browser representing the core system. The examples are endless for product-based software, but what about large business applications? The microkernel architecture applies to these situations as well. To illustrate this point, consider an insurance company example involving insurance claims processing.

Claims processing is a very complicated process. Each jurisdiction has different rules and regulations for what is and isn't allowed in an insurance claim. For example, some jurisdictions (e.g., states) allow free windshield replacement if your windshield is damaged by a rock, whereas other states do not. This creates an almost infinite set of conditions for a standard claims process.

Most insurance claims applications leverage large and complex rules engines to handle much of this complexity. However, these rules engines can grow into a complex big ball of mud where changing one rule impacts other rules, or making a simple rule change requires an army of analysts, developers, and testers to make sure nothing is broken by a simple change. Using the microkernel architecture pattern can solve many of these issues.

The claims rules for each jurisdiction can be contained in separate standalone plug-in components (implemented as source code or a specific rules engine instance accessed by the plug-in component). This way, rules can be added, removed, or changed for a particular jurisdiction without impacting any other part of the system. Furthermore, new jurisdictions can be added and removed without impacting other parts of the system. The core system in this example would be the standard process for filing and processing a claim, something that doesn't change often.

Another example of a large and complex business application that can leverage the microkernel architecture is tax preparation software. For example, the United States has a basic two-page tax form called the 1040 form that contains a summary of all the information needed to calculate a person's tax liability. Each line in the 1040 tax form has a single number that requires many other forms and worksheets to arrive at that single number (such as gross income). Each of these additional forms and worksheets can be implemented as a plug-in component, with the 1040 summary tax form being the core system (the driver). This way, changes to tax law can be isolated to an independent plug-in component, making changes easier and less risky.

Architecture Characteristics Ratings

A one-star rating in the characteristics ratings in Figure 12-8 means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in Chapter 4.

Architecture characteristic	Star rating
Partitioning type	Domain and technical
Number of quanta	1
Deployability	★★★
Elasticity	★
Evolutionary	★★★
Fault tolerance	★
Modularity	★★★
Overall cost	★★★★★
Performance	★★★
Reliability	★★★
Scalability	★
Simplicity	★★★★
Testability	★★★

Figure 12-8. Microkernel architecture characteristics ratings

Similar to the layered architecture style, simplicity and overall cost are the main strengths of the microkernel architecture style, and scalability, fault tolerance, and extensibility its main weaknesses. These weaknesses are due to the typical monolithic deployments found with the microkernel architecture. Also, like the layered architecture style, the number of quanta is always singular (one) because all requests must go through the core system to get to independent plug-in components. That's where the similarities end.

The microkernel architecture style is unique in that it is the only architecture style that can be both domain partitioned *and* technically partitioned. While most microkernel architectures are technically partitioned, the domain partitioning aspect comes about mostly through a strong domain-to-architecture isomorphism. For example, problems that require different configurations for each location or client match extremely well with this architecture style. Another example is a product or application that places a strong emphasis on user customization and feature extensibility (such as Jira or an IDE like Eclipse).

Testability, deployability, and reliability rate a little above average (three stars), primarily because functionality can be isolated to independent plug-in components. If done right, this reduces the overall testing scope of changes and also reduces overall risk of deployment, particularly if plug-in components are deployed in a runtime fashion.

Modularity and extensibility also rate a little above average (three stars). With the microkernel architecture style, additional functionality can be added, removed, and changed through independent, self-contained plug-in components, thereby making it relatively easy to extend and enhance applications created using this architecture style and allowing teams to respond to changes much faster. Consider the tax preparation software example from the previous section. If the US tax law changes (which it does all the time), requiring a new tax form, that new tax form can be created as a plug-in component and added to the application without much effort. Similarly, if a tax form or worksheet is no longer needed, that plug-in can simply be removed from the application.

Performance is always an interesting characteristic to rate with the microkernel architecture style. We gave it three stars (a little above average) mostly because microkernel applications are generally small and don’t grow as big as most layered architectures. Also, they don’t suffer as much from the architecture sinkhole anti-pattern discussed in Chapter 10. Finally, microkernel architectures can be streamlined by unplugging unneeded functionality, therefore making the application run faster. A good example of this is Wildfly (previously the JBoss Application Server). By unplugging

unnecessary functionality like clustering, caching, and messaging, the application server performs much faster than with these features in place.

Chapter 13. Service-Based Architecture Style

Service-based architecture is a hybrid of the microservices architecture style and is considered one of the most pragmatic architecture styles, mostly due to its architectural flexibility. Although service-based architecture is a distributed architecture, it doesn't have the same level of complexity and cost as other distributed architectures, such as microservices or event-driven architecture, making it a very popular choice for many business-related applications.

Topology

The basic topology of service-based architecture follows a distributed macro layered structure consisting of a separately deployed user interface, separately deployed remote coarse-grained services, and a monolithic database. This basic topology is illustrated in Figure 13-1.

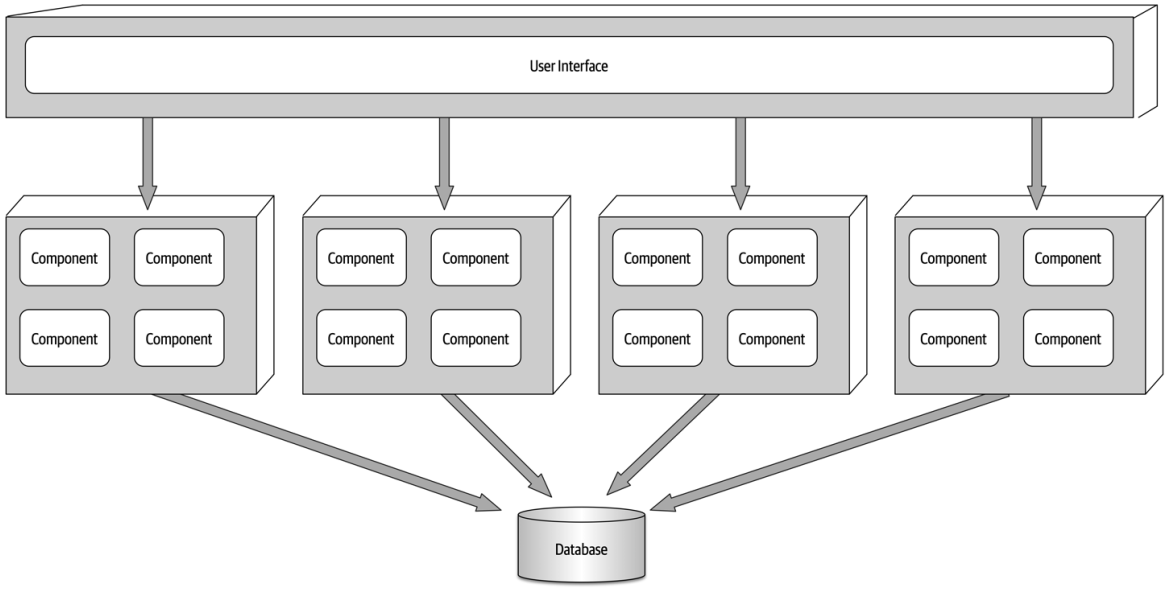

Figure 13-1. Basic topology of the service-based architecture style

Services within this architecture style are typically coarse-grained “portions of an application” (usually called *domain services*) that are independent and separately deployed. Services are typically deployed in the same manner as any monolithic application would be (such as an EAR file, WAR file, or assembly) and as such do not require containerization (although you could deploy a domain service in a container such as Docker). Because the services typically share a single monolithic database, the number of services within an application context generally range between 4 and 12 services, with the average being about 7 services.

In most cases there is only a single instance of each domain service within a service-based architecture. However, based on scalability, fault tolerance, and throughput needs, multiple instances of a domain service can certainly exist. Multiple instances of a service usually require some sort of load-balancing capability between the user interface and the domain service so that the user interface can be directed to a healthy and available service instance.

Services are accessed remotely from a user interface using a remote access protocol. While REST is typically used to access services from the user interface, messaging, remote procedure call (RPC), or even SOAP could be used as well. While an API layer consisting of a proxy or gateway can be used to access services from the user interface (or other external requests), in most cases the user interface accesses the services directly using a service locator pattern embedded within the user interface, API gateway, or proxy.

One important aspect of service-based architecture is that it typically uses a centrally shared database. This allows services to leverage SQL queries and joins in the same way a traditional monolithic layered architecture would. Because of the small number of services (4 to 12), database connections are not usually an issue in service-based architecture. Database changes, however, can be an issue. The section “Database Partitioning” describes techniques for addressing and managing database change within a service-based architecture.

Topology Variants

Many topology variants exist within the service-based architecture style, making this perhaps one of the most flexible architecture styles. For example, the single monolithic user interface, as illustrated in Figure 13-1, can be broken apart into user interface domains, even to a level matching each domain service. These user interface variants are illustrated in Figure 13-2.

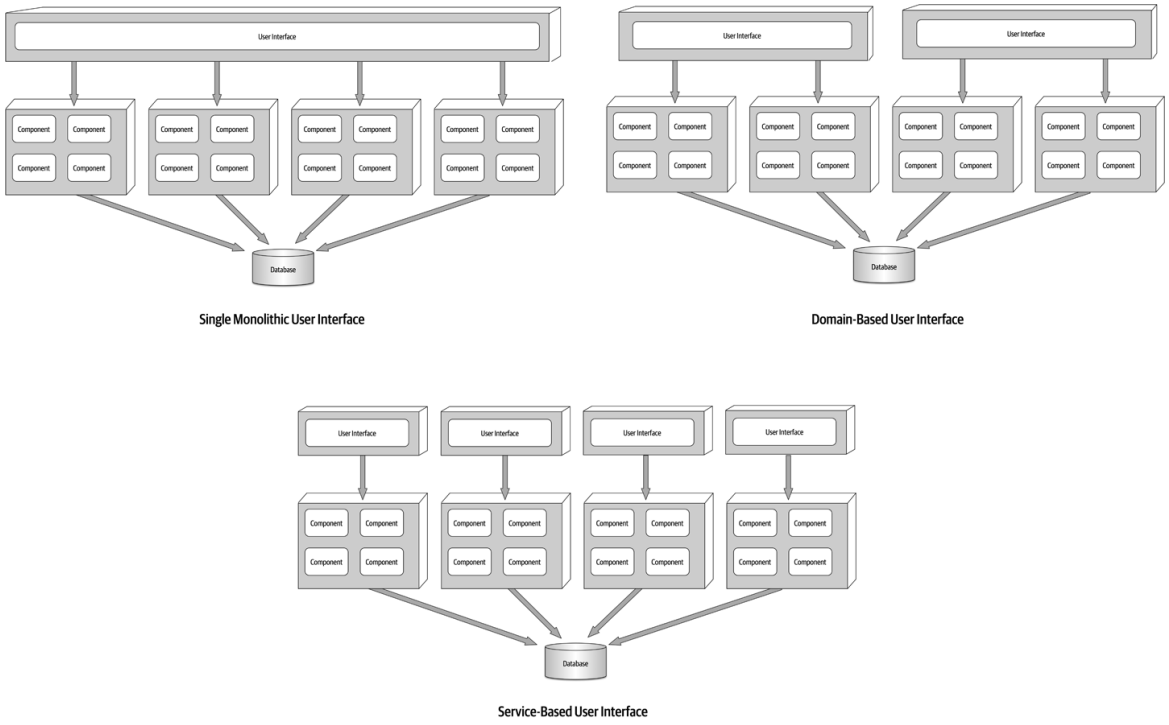

Figure 13-2. User interface variants

Similarly, opportunities may exist to break apart a single monolithic database into separate databases, even going as far as domain-scoped databases matching each domain service (similar to microservices). In these cases it is important to make sure the data in each separate database is not needed by another domain service. This avoids interservice communication between domain services (something to definitely avoid with service-based architecture) and also the duplication of data between databases. These database variants are illustrated in Figure 13-3.

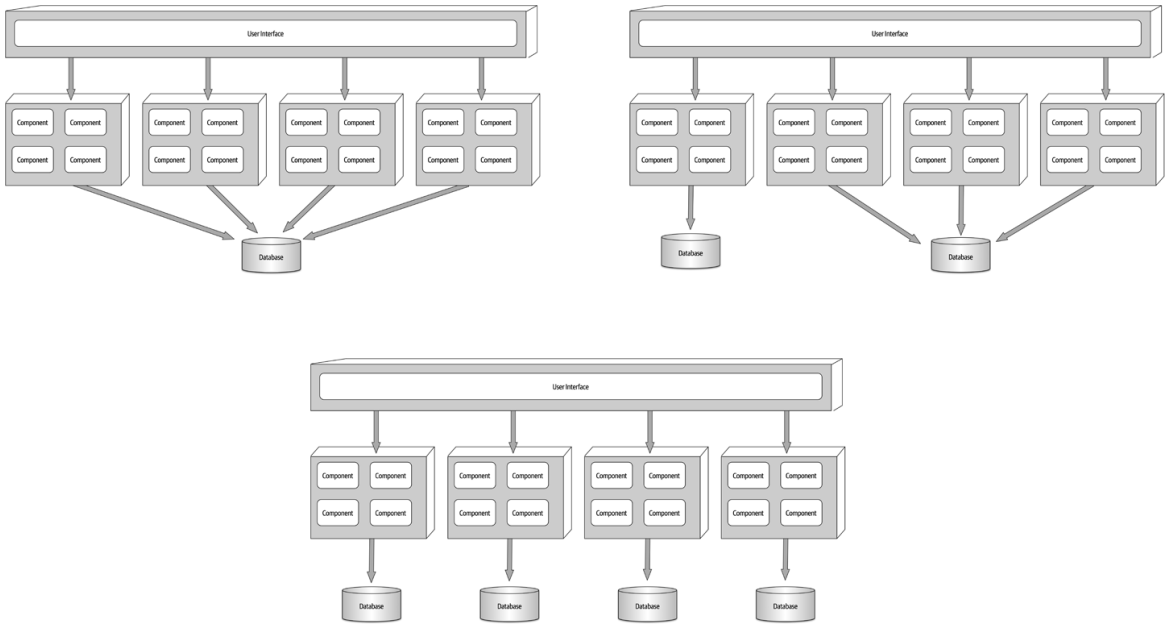

Figure 13-3. Database variants

Finally, it is also possible to add an API layer consisting of a reverse proxy or gateway between the user interface and services, as shown in Figure 13-4. This is a good practice when exposing domain service functionality to external systems or when consolidating shared cross-cutting concerns and moving them outside of the user interface (such as metrics, security, auditing requirements, and service discovery).

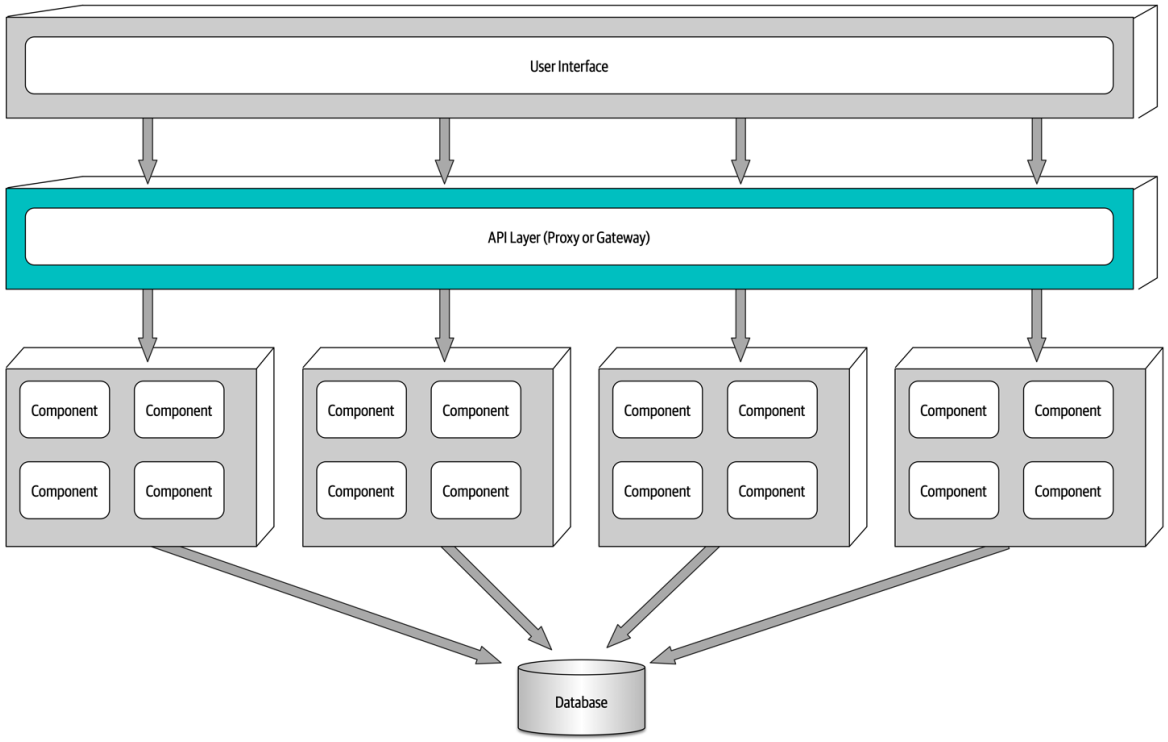

Figure 13-4. Adding an API layer between the user interface and domain services

Service Design and Granularity

Because domain services in a service-based architecture are generally coarse-grained, each domain service is typically designed using a layered architecture style consisting of an API facade layer, a business layer, and a persistence layer. Another popular design approach is to domain partition each domain service using sub-domains similar to the modular monolith architecture style. Each of these design approaches is illustrated in Figure 13-5.

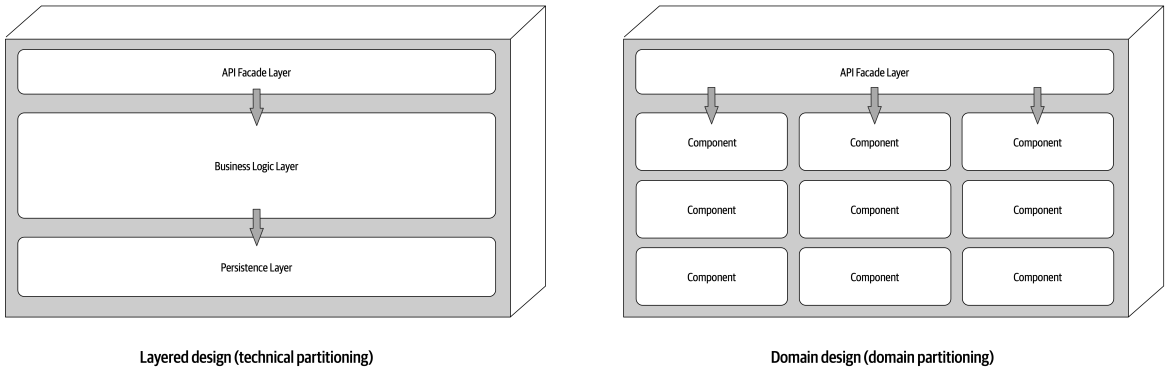

Figure 13-5. Domain service design variants

Regardless of the service design, a domain service must contain some sort of API access facade that the user interface interacts with to execute some sort of business functionality. The API access facade typically takes on the responsibility of orchestrating the business request from the user interface. For example, consider a business request from the user interface to place an order (also known as catalog checkout). This single request, received by the API access facade within the `OrderService` domain service, internally orchestrates the single business request: place the order, generate an order ID, apply the payment, and update the product inventory for each product ordered. In the microservices architecture style, this would likely involve the orchestration of many separately deployed remote single-purpose services to complete the request. This difference between internal class-level orchestration and external service orchestration points to one of the many significant differences between service-based architecture and microservices in terms of granularity.

Because domain services are coarse-grained, regular ACID (atomicity, consistency, isolation, durability) database transactions involving database commits and rollbacks are used to ensure database integrity within a single domain service. Highly distributed architectures like microservices, on the other hand, usually have fine-grained services and use a distributed transaction technique known as BASE transactions (basic availability, soft state, eventual consistency) transactions that rely on eventual consistency and hence do not support the same level of database integrity as ACID transactions in a service-based architecture.

To illustrate this point, consider the example of a catalog checkout process within a service-based architecture. Suppose the customer places an order and the credit card used for payment has expired. Since this is an atomic transaction within the same service, everything added to the database can be removed using a rollback and a notice sent to the customer stating that the payment cannot be applied. Now consider this same process in a microservices architecture with smaller fine-grained services. First, the `OrderPlacement` service would accept the request, create the order, generate an order ID, and insert the order into the order tables. Once this is done, the order service would then make a remote call to the `PaymentService`, which would try to apply the payment. If the payment cannot be applied due to an expired credit card, then the order cannot be placed and the data is in an inconsistent state (the order information has already been inserted but has not been approved). In this case, what about the inventory for that order? Should it be marked as ordered and decremented? What if the inventory is low and another customer wishes to purchase the item? Should that new customer be allowed to buy it, or should the reserved inventory be reserved for the customer trying to place the order with an expired credit card? These are just a few of the questions that would need to be addressed when orchestrating a business process with multiple finer-grained services.

Domain services, being coarse-grained, allow for better data integrity and consistency, but there is a trade-off. With service-based architecture, a change made to the order placement functionality in the `OrderService` would require testing the entire coarse-grained service (including payment processing), whereas with microservices the same change would only impact a small `OrderPlacement` service (requiring no change to the `PaymentService`). Furthermore, because more code is being deployed, there is more risk with service-based architecture that something might break (including payment processing), whereas with microservices each service has a single responsibility, hence less chance of breaking other functionality when being changed.

Database Partitioning

Although not required, services within a service-based architecture usually share a single, monolithic database due to the small number of services (4 to 12) within a given application context. This database coupling can present an issue with respect to database table schema changes. If not done properly, a table schema change can potentially impact every service, making database changes a very costly task in terms of effort and coordination.

Within a service-based architecture, the shared class files representing the database table schemas (usually referred to as *entity objects*) reside in a custom shared library used by all the domain services (such as a JAR file or DLL). Shared libraries might also contain SQL code. The practice of creating a single shared library of entity objects is the least effective way of implementing service-based architecture. Any change to the database table structures would also require a change to the single shared library containing all of the corresponding entity objects, thus requiring a change and redeployment to every service, regardless of whether or not the services actually access the changed table. Shared library versioning can help address this issue, but nevertheless, with a single shared library it is difficult to know which services are actually impacted by the table change without manual, detailed analysis. This single shared library scenario is illustrated in Figure 13-6.

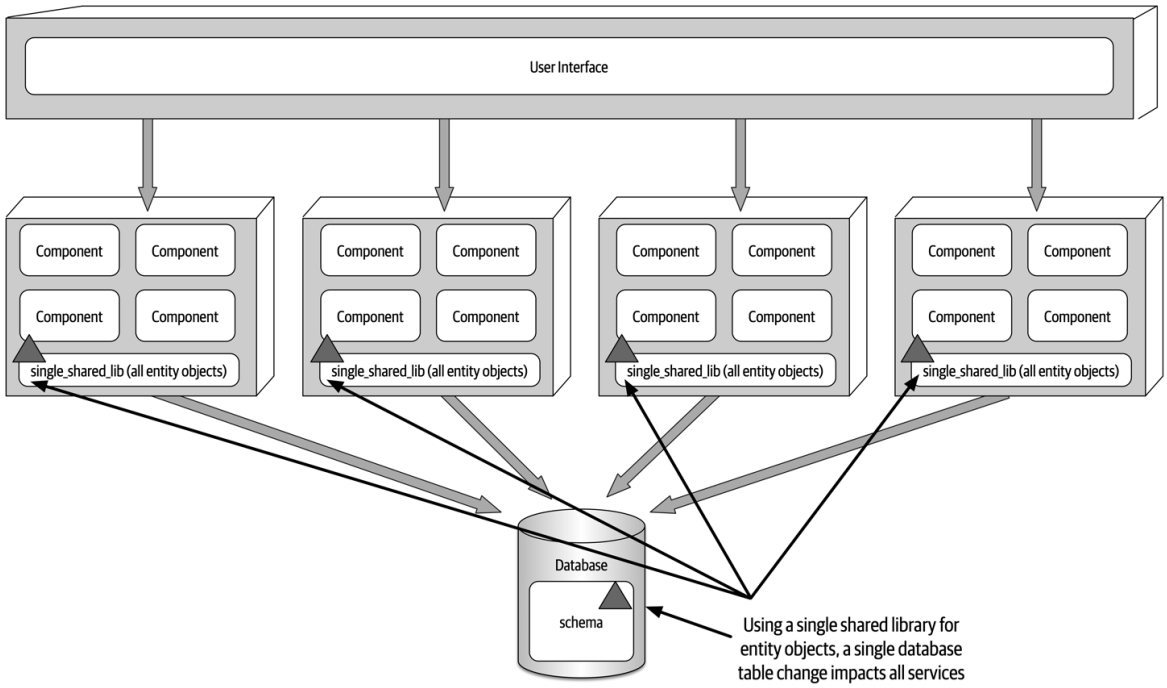

Figure 13-6. Using a single shared library for database entity objects

One way to mitigate the impact and risk of database changes is to logically partition the database and manifest the logical partitioning through federated shared libraries. Notice in Figure 13-7 that the database is logically partitioned into five separate domains (common, customer, invoicing, order, and tracking). Also notice that there are five corresponding shared libraries used by the domain services matching the logical partitions in the database. Using this technique, changes to a table within a particular logical domain (in this case, invoicing) match the corresponding shared library containing the entity objects (and possibly SQL as well), impacting only those services using that shared library, which in this case is the invoicing service. No other services are impacted by this change.

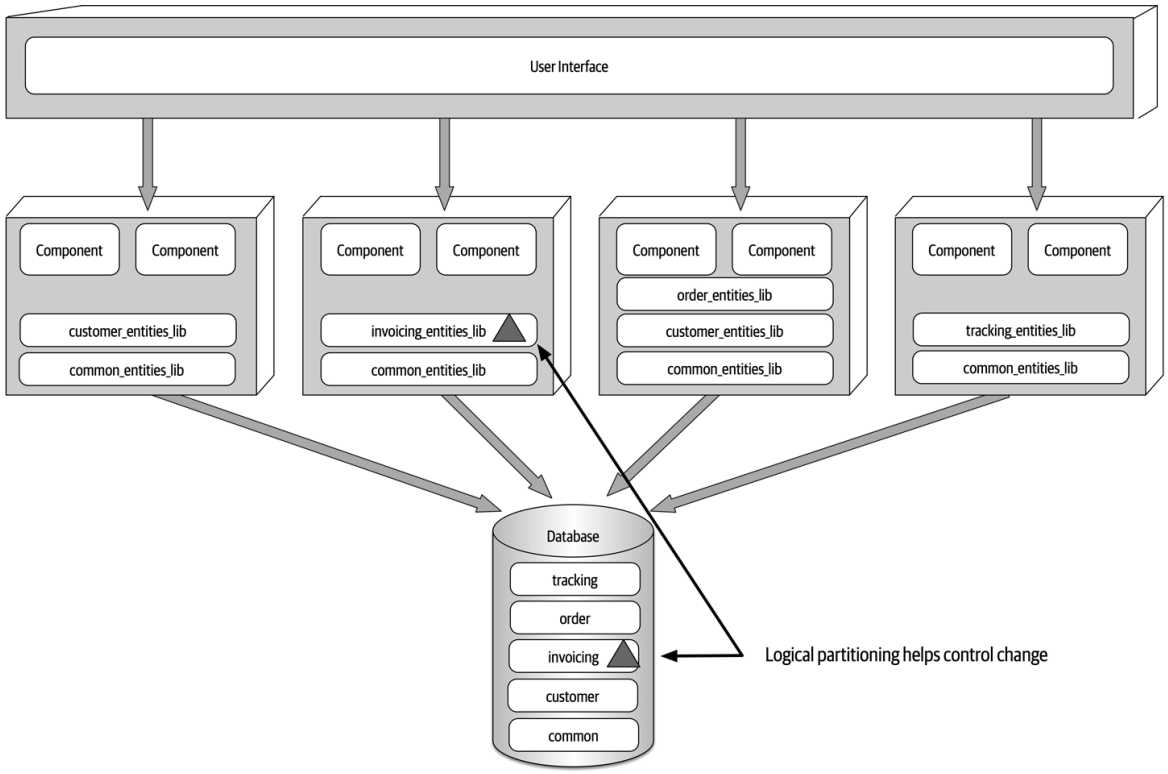

Figure 13-7. Using multiple shared libraries for database entity objects

Notice in Figure 13-7 the use of the *common* domain and the corresponding `common_entities_lib` shared library used by all services. This is a relatively common occurrence. These tables are common to all services, and as such, changes to these tables require coordination of all services accessing the shared database. One way to mitigate changes to these tables (and corresponding entity objects) is to lock the common entity objects in the version control system and restrict change access to only the database team. This helps control change and emphasizes the significance of changes to the common tables used by all services.

> **TIP**
>
> Make the logical partitioning in the database as fine-grained as possible while still maintaining well-defined data domains to better control database changes within a service-based architecture.

Example Architecture

To illustrate the flexibility and power of the service-based architecture style, consider the real-world example of an electronic recycling system used to recycle old electronic devices (such as an iPhone or Galaxy cell phone). The processing flow of recycling old electronic devices works as follows: first, the customer asks the company (via a website or kiosk) how much money they can get for the old electronic device (called *quoting*). If satisfied, the customer will send the electronic device to the recycling company, which in turn will receive the physical device (called *receiving*). Once received, the recycling company will then assess the device to determine if the device is in good working condition or not (called *assessment*). If the device is in good working condition, the company will send the customer the money promised for the device (called *accounting*). Through this process, the customer can go to the website at any time to check on the status of the item (called *item status*). Based on the assessment, the device is then recycled by either safely destroying it or reselling it (called *recycling*). Finally, the company periodically runs ad hoc and scheduled financial and operational reports based on recycling activity (called *reporting*).

Figure 13-8 illustrates this system using a service-based architecture. Notice how each domain area identified in the prior description is implemented as a separately deployed independent domain service. Scalability can be achieved by only scaling those services needing higher throughput (in this case, the customer-facing `Quoting` service and `ItemStatus` service). The other services do not need to scale, and as such only require a single service instance.

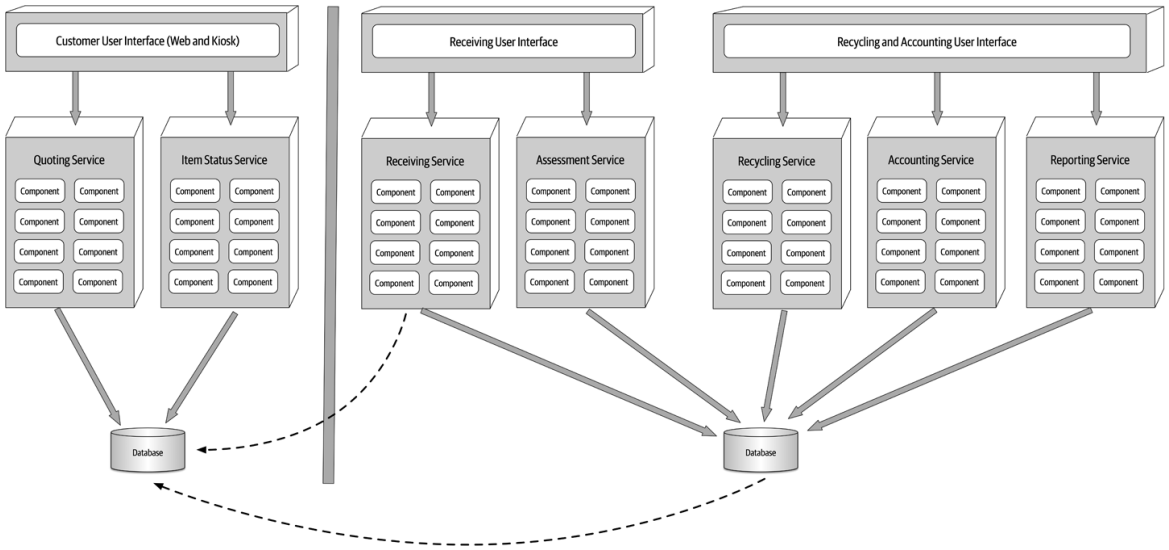

Figure 13-8. Electronics recycling example using service-based architecture

Also notice in Figure 13-8 how the user interface applications are federated into their respective domains: *Customer Facing*, *Receiving*, and *Recycling and Accounting*. This federation allows for fault tolerance of the user interface, scalability, and security (external customers have no network path to internal functionality). Finally, notice in this example that there are two separate physical databases: one for external customer-facing operations, and one for internal operations. This allows the internal data and operations to reside in a separate network zone from the external operations (denoted by the vertical line), providing much better security access restrictions and data protection. One-way access through the firewall allows internal services to access and update the customer-facing information, but not vice versa. Alternatively, depending on the database being used, internal table mirroring and table synchronization could also be used.

This example illustrates many of the benefits of the service-based architecture approach: scalability, fault tolerance, and security (data and functionality protection and access), in addition to agility, testability, and deployability. For example, the `Assessment` service is changed constantly to add assessment rules as new products are received. This frequent change is isolated to a single domain service, providing agility (the ability to respond quickly to change), as well as testability (the ease of and

completeness of testing) and deployability (the ease, frequency, and risk of deployment).

Architecture Characteristics Ratings

A one-star rating in the characteristics ratings table in Figure 13-9 means the specific architecture characteristic isn’t well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in Chapter 4.

Architecture characteristic	Star rating
Partitioning type	Domain
Number of quanta	1 to many
Deployability	★★★★
Elasticity	★★
Evolutionary	★★★
Fault tolerance	★★★★
Modularity	★★★★
Overall cost	★★★★
Performance	★★★
Reliability	★★★★
Scalability	★★★
Simplicity	★★★
Testability	★★★★

Figure 13-9. Service-based architecture characteristics ratings

Service-based architecture is a *domain-partitioned* architecture, meaning that the structure is driven by the domain rather than a technical consideration (such as presentation logic or persistence logic). Consider the prior example of the electronic recycling application. Each service, being a separately deployed unit of software, is scoped to a specific domain (such as item assessment). Changes made within this domain only impact the specific service, the corresponding user interface, and the corresponding

database. Nothing else needs to be modified to support a specific assessment change.

Being a distributed architecture, the number of quanta can be greater than or equal to one. Even though there may be anywhere from 4 to 12 separately deployed services, if those services all share the same database or user interface, that entire system would be only a single quantum. However, as illustrated in "Topology Variants", both the user interface and database can be federated, resulting in multiple quanta within the overall system. In the electronics recycling example, the system contains two quanta, as illustrated in Figure 13-10: one for the customer-facing portion of the application containing a separate customer user interface, database, and set of services (`Quoting` and `Item Status`); and one for the internal operations of receiving, assessing, and recycling the electronic device. Notice that even though the internal operations quantum contains separately deployed services and two separate user interfaces, they all share the same database, making the internal operations portion of the application a single quantum.

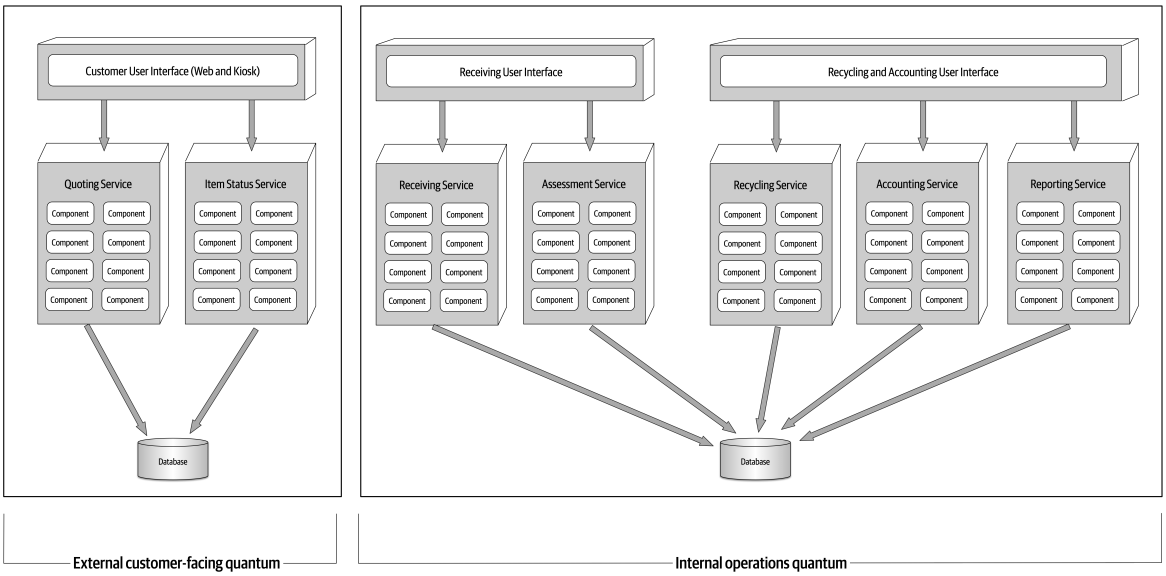

Figure 13-10. Separate quanta in a service-based architecture

Although service-based architecture doesn't contain any five-star ratings, it nevertheless rates high (four stars) in many important and vital areas. Breaking apart an application into separately deployed domain services using this architecture style allows for faster change (agility), better test

coverage due to the limited scope of the domain (testability), and the ability for more frequent deployments carrying less risk than a large monolith (deployability). These three characteristics lead to better time-to-market, allowing an organization to deliver new features and bug fixes at a relatively high rate.

Fault tolerance and overall application availability also rate high for service-based architecture. Even though domain services tend to be coarse-grained, the four-star rating comes from the fact that with this architecture style, services are usually self-contained and do not leverage interservice communication due to database sharing and code sharing. As a result, if one domain service goes down (e.g., the `Receiving` service in the electronic recycling application example), it doesn't impact any of the other six services.

Scalability only rates three stars due to the coarse-grained nature of the services, and correspondingly, elasticity only two stars. Although programmatic scalability and elasticity are certainly possible with this architecture style, more functionality is replicated than with finer-grained services (such as microservices) and as such is not as efficient in terms of machine resources and not as cost-effective. Typically there are only single service instances with service-based architecture unless there is a need for better throughput or failover. A good example of this is the electronics recycling application example—only the `Quoting` and `Item Status` services need to scale to support high customer volumes, but the other operational services only require single instances, making it easier to support such things as single in-memory caching and database connection pooling.

Simplicity and overall cost are two other drivers that differentiate this architecture style from other, more expensive and complex distributed architectures, such as microservices, event-driven architecture, or even space-based architecture. This makes service-based one of the easiest and cost-effective distributed architectures to implement. While this is an attractive proposition, there is a trade-off to this cost savings and simplicity

in all of the characteristics containing four-star ratings. The higher the cost and complexity, the better these ratings become.

Service-based architectures tend to be more reliable than other distributed architectures due to the coarse-grained nature of the domain services. Larger services mean less network traffic to and between services, fewer distributed transactions, and less bandwidth used, therefore increasing overall reliability with respect to the network.

When to Use This Architecture Style

The flexibility of this architecture style (see "Topology Variants") combined with the number of three-star and four-star architecture characteristics ratings make service-based architecture one of the most pragmatic architecture styles available. While there are certainly other distributed architecture styles that are much more powerful, some companies find that power comes at too steep of a price, while others find that they quite simply don't need that much power. It's like having the power, speed, and agility of a Ferrari used only for driving back and forth to work in rush-hour traffic at 50 kilometers per hour—sure it looks cool, but what a waste of resources and money!

Service-based architecture is also a natural fit when doing domain-driven design. Because services are coarse-grained and domain-scoped, each domain fits nicely into a separately deployed domain service. Each service in service-based architecture encompasses a particular domain (such as recycling in the electronic recycling application), therefore compartmentalizing that functionality into a single unit of software, making it easier to apply changes to that domain.

Maintaining and coordinating database transactions is always an issue with distributed architectures in that they typically rely on eventual consistency rather than traditional ACID (atomicity, consistency, isolation, and durability) transactions. However, service-based architecture preserves ACID transactions better than any other distributed architecture due to the coarse-grained nature of the domain services. There are cases where the

user interface or API gateway might orchestrate two or more domain services, and in these cases the transaction would need to rely on sagas and BASE transactions. However, in most cases the transaction is scoped to a particular domain service, allowing for the traditional commit and rollback transaction functionality found in most monolithic applications.

Lastly, service-based architecture is a good choice for achieving a good level of architectural modularity without having to get tangled up in the complexities and pitfalls of granularity. As services become more fine-grained, issues surrounding orchestration and choreography start to appear. Both orchestration and choreography are required when multiple services must be coordinated to complete a certain business transaction. Orchestration is the coordination of multiple services through the use of a separate mediator service that controls and manages the workflow of the transaction (like a conductor in an orchestra). Choreography, on the other hand, is the coordination of multiple services by which each service talks to one another without the use of a central mediator (like dancers in a dance). As services become more fine-grained, both orchestration and choreography are necessary to tie the services together to complete the business transaction. However, because services within a service-based architecture tend to be more coarse-grained, they don’t require coordination nearly as much as other distributed architectures.

Chapter 14. Event-Driven Architecture Style

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

Most applications follow what is called a *request-based* model (illustrated in Figure 14-1). In this model, requests made to the system to perform some sort of action are send to a *request orchestrator*. The request orchestrator is typically a user interface, but it can also be implemented through an API layer or enterprise service bus. The role of the request orchestrator is to deterministically and synchronously direct the request to various *request processors*. The request processors handle the request, either retrieving or updating information in a database.

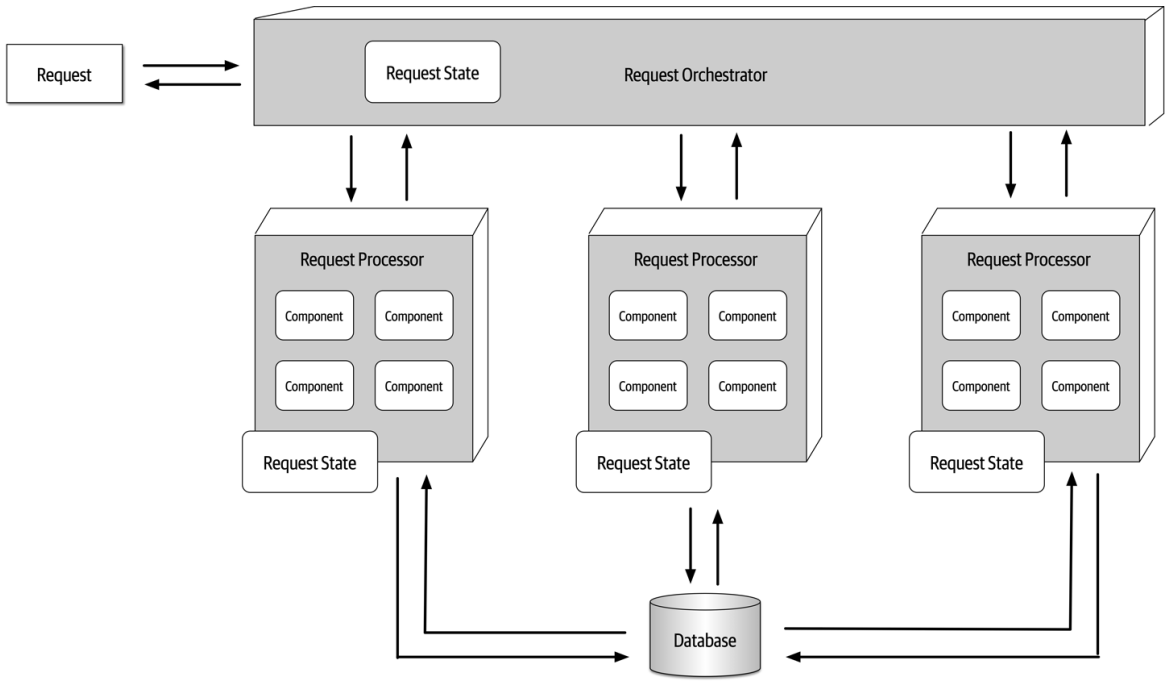

Figure 14-1. Request-based model

A good example of the request-based model is a request from a customer to retrieve their order history for the past six months. Retrieving order history information is a data-driven, deterministic request made to the system for data within a specific context, not an event happening that the system must react to.

An event-based model, on the other hand, reacts to a particular situation and takes action based on that event. An example of an event-based model is submitting a bid for a particular item within an online auction. Submitting the bid is not a request made to the system, but rather an event that happens after the current asking price is announced. The system must respond to this event by comparing the bid to others received at the same time to determine who is the current highest bidder.

Topology

There are two primary topologies within event-driven architecture: the *mediator topology* and the *broker topology*. The mediator topology is commonly used when you require control over the workflow of an event

process, whereas the broker topology is used when you require a high degree of responsiveness and dynamic control over the processing of an event. Because the architecture characteristics and implementation strategies differ between these two topologies, it is important to understand each one to know which is best suited for a particular situation.

Broker Topology

The broker topology differs from the mediator topology in that there is no central event mediator. Rather, the message flow is distributed across the event processor components in a chain-like broadcasting fashion through a lightweight message broker (such as RabbitMQ, ActiveMQ, HornetQ, and so on). This topology is useful when you have a relatively simple event processing flow and you do not need central event orchestration and coordination.

There are four primary architecture components within the broker topology: an initiating event, the event broker, an event processor, and a processing event. The *initiating event* is the initial event that starts the entire event flow, whether it be a simple event like placing a bid in an online auction or more complex events in a health benefits system like changing a job or getting married. The initiating event is sent to an event channel in the *event broker* for processing. Since there is no mediator component in the broker topology managing and controlling the event, a single *event processor* accepts the initiating event from the event broker and begins the processing of that event. The event processor that accepted the initiating event performs a specific task associated with the processing of that event, then asynchronously advertises what it did to the rest of the system by creating what is called a *processing event*. This processing event is then asynchronously sent to the event broker for further processing, if needed. Other event processors listen to the processing event, react to that event by doing something, then advertise through a new processing event what they did. This process continues until no one is interested in what a final event processor did. Figure 14-2 illustrates this event processing flow.

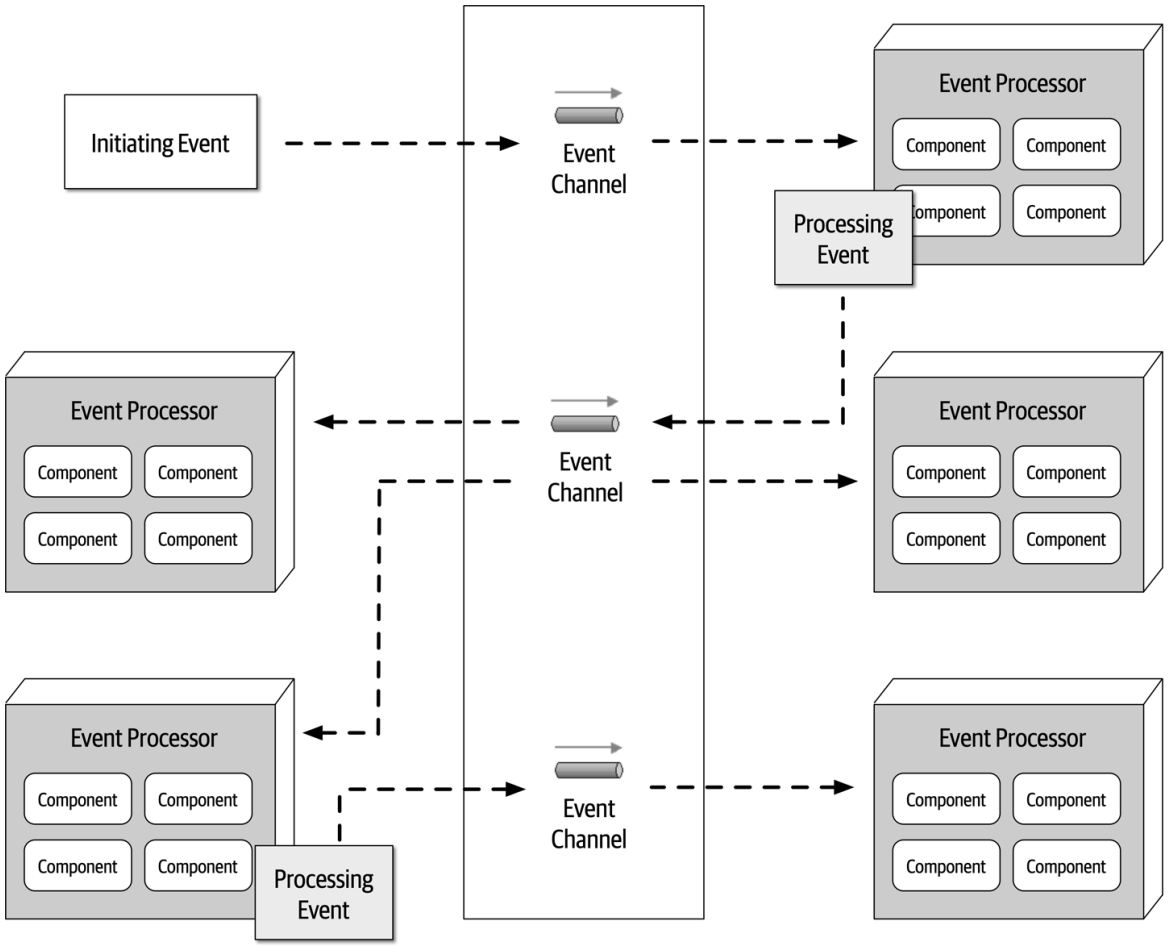

Figure 14-2. Broker topology

The event broker component is usually federated (meaning multiple domain-based clustered instances), where each federated broker contains all of the event channels used within the event flow for that particular domain. Because of the decoupled asynchronous fire-and-forget broadcasting nature of the broker topology, topics (or topic exchanges in the case of AMQP) are usually used in the broker topology using a publish-and-subscribe messaging model.

It is always a good practice within the broker topology for each event processor to advertise what it did to the rest of the system, regardless of whether or not any other event processor cares about what that action was. This practice provides architectural extensibility if additional functionality is required for the processing of that event. For example, suppose as part of a complex event process, as illustrated in Figure 14-3, an email is generated

and sent to a customer notifying them of a particular action taken. The `Notification` event processor would generate and send the email, then advertise that action to the rest of the system through a new processing event sent to a topic. However, in this case, no other event processors are listening for events on that topic, and as such the message simply goes away.

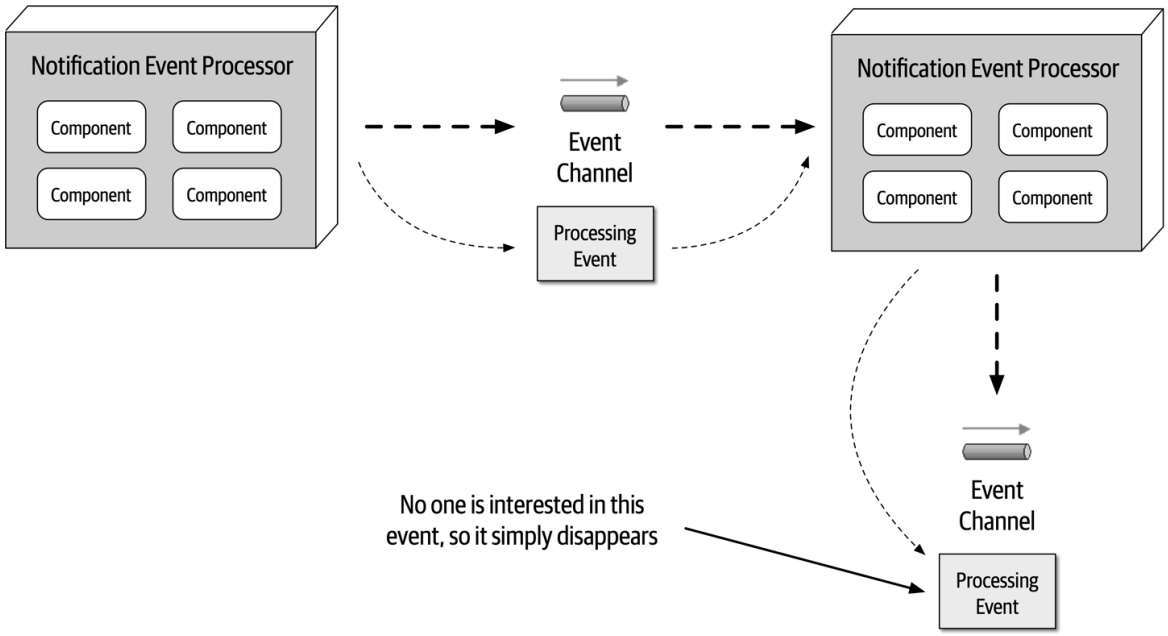

Figure 14-3. Notification event is sent but ignored

This is a good example of *architectural extensibility*. While it may seem like a waste of resources sending messages that are ignored, it is not. Suppose a new requirement comes along to analyze emails that have been sent to customers. This new event processor can be added to the overall system with minimal effort because the email information is available via the email topic to the new analyzer without having to add any additional infrastructure or apply any changes to other event processors.

To illustrate how the broker topology works, consider the processing flow in a typical retail order entry system, as illustrated in Figure 14-4, where an order is placed for an item (say, a book like this one). In this example, the `OrderPlacement` event processor receives the initiating event (`PlaceOrder`), 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-created` processing event. Notice that three event processors are interested in that event: the `Notification` event processor, the `Payment` event processor, and the `Inventory` event processor. All three of these event processors perform their tasks in parallel.

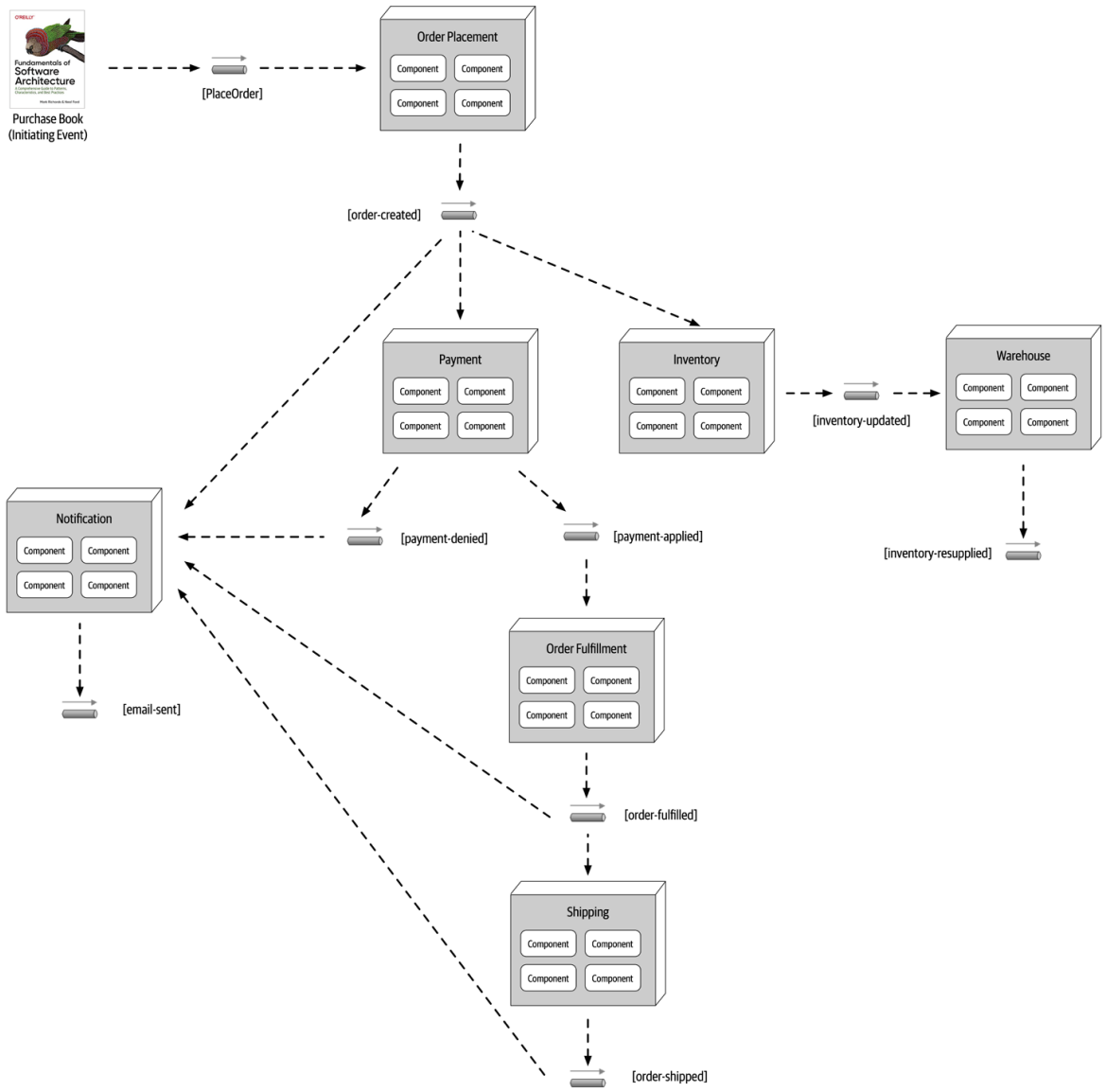

Figure 14-4. Example of the broker topology

The `Notification` event processor receives the `order-created` processing event and emails the customer. It then generates another processing event (`email-sent`). Notice that no other event processors are listening to that event. This is normal and illustrates the previous example

describing architectural extensibility—an in-place hook so that other event processors can eventually tap into that event feed, if needed.

The `Inventory` event processor also listens for the `order-created` processing event and decrements the corresponding inventory for that book. It then advertises this action through an `inventory-updated` processing event, which is in turn picked up by the `Warehouse` event processor to manage the corresponding inventory between warehouses, reordering items if supplies get too low.

The `Payment` event processor also receives the `order-created` processing event and charges the customer's credit card for the order that was just created. Notice in Figure 14-4 that two events are generated as a result of the actions taken by the `Payment` event processor: one to notify the rest of the system that the payment was applied (`payment-applied`) and one processing event to notify the rest of the system that the payment was denied (`payment-denied`). Notice that the `Notification` event processor is interested in the `payment-denied` processing event, because it must, in turn, send an email to the customer informing them that they must update their credit card information or choose a different payment method.

The `OrderFulfillment` event processor listens to the `payment-applied` processing event and does order picking and packing. Once completed, it then advertises to the rest of the system that it fulfilled the order via an `order-fulfilled` processing event. Notice that both the `Notification` processing unit and the `Shipping` processing unit listen to this processing event. Concurrently, the `Notification` event notifies the customer that the order has been fulfilled and is ready for shipment, and at the same time the `Shipping` event processor selects a shipping method. The `Shipping` event processor ships the order and sends out an `order-shipped` processing event, which the `Notification` event processor also listens for to notify the customer of the order status change.

In analyzing the prior example, notice that all of the event processors are highly decoupled and independent of each other. The best way to

understand the broker topology 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, and so on down the chain until the last runner crosses the finish line. In relay races, once a runner hands off the baton, that runner is done with the race and moves on to other things. This is also true with the broker topology. Once an event processor hands off the event, it is no longer involved with the processing of that specific event and is available to react to other initiating or processing events. In addition, each event processor can scale independently from one other to handle varying load conditions or backups in the processing within that event. The topics provide the back pressure point if an event processor comes down or slows down due to some environment issue.

While performance, responsiveness, and scalability are all great benefits of the broker topology, there are also some negatives about it. First of all, there is no control over the overall workflow associated with the initiating event (in this case, the `PlaceOrder` event). It is very dynamic based on various conditions, and no one in the system really knows when the business transaction of placing an order is actually complete. Error handling is also a big challenge with the broker topology. Because there is no mediator monitoring or controlling the business transaction, if a failure occurs (such as the `Payment` event processor crashing and not completing its assigned task), no one in the system is aware of that crash. The business process gets stuck and is unable to move without some sort of automated or manual intervention. Furthermore, all other processes are moving along without regard for the error. For example, the `Inventory` event processor still decrements the inventory, and all other event processors react as though everything is fine.

The ability to restart a business transaction (recoverability) is also something not supported with the broker topology. Because other actions have asynchronously been taken through the initial processing of the initiating event, it is not possible to resubmit the initiating event. No component in the broker topology is aware of the state or even owns the

state of the original business request, and therefore no one is responsible in this topology for restarting the business transaction (the initiating event) and knowing where it left off. The advantages and disadvantages of the broker topology are summarized in Table 14-1.

Table 14-1. Trade-offs of the broker topology

Advantages	Disadvantages
Highly decoupled event processors	Workflow control
High scalability	Error handling
High responsiveness	Recoverability
High performance	Restart capabilities
High fault tolerance	Data inconsistency

Mediator Topology

The mediator topology of event-driven architecture addresses some of the shortcomings of the broker topology described in the previous section. Central to this topology is an event mediator, which manages and controls the workflow for initiating events that require the coordination of 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.

Like in the broker topology, the initiating event is the event that starts the whole eventing process. Unlike the broker topology, the initiating event is sent to an initiating event queue, which is accepted by the event mediator. The event mediator only knows the steps involved in processing the event and therefore generates corresponding processing events that are sent to dedicated event channels (usually queues) in a point-to-point messaging fashion. Event processors then listen to dedicated event channels, process the event, and usually respond back to the mediator that they have

completed their work. Unlike the broker topology, event processors within the mediator topology do not advertise what they did to the rest of the system. The mediator topology is illustrated in Figure 14-5.

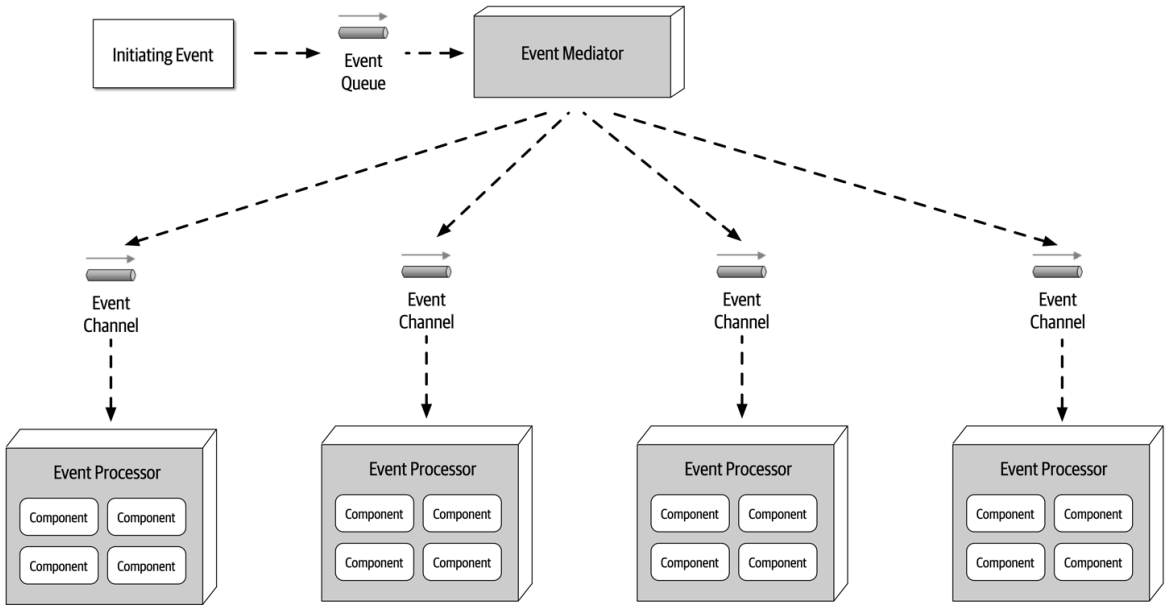

Figure 14-5. Mediator topology

In most implementations of the mediator topology, there are multiple mediators, usually associated with a particular domain or grouping of events. This reduces the single point of failure issue associated with this topology and also increases overall throughput and performance. For example, there might be a customer mediator that handles all customer-related events (such as new customer registration and profile update), and another mediator that handles order-related activities (such as adding an item to a shopping cart and checking out).

The event mediator can be implemented in a variety of ways, depending on the nature and complexity of the events it is processing. For example, for events requiring simple error handling and orchestration, a mediator 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 workflow of the event processing.

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 good choice. These mediators are based on 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, and as such is usually created using graphical interface tools provided in the product's BPEL engine suite.

BPEL is good for complex and dynamic workflows, but it does not work well for those event workflows requiring long-running transactions involving 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 a manual approval is required because the trade is over a certain amount of shares. In this case the event mediator would have to stop the event processing, send a notification to a senior trader for the manual approval, and wait for that approval to occur. In these cases a Business Process Management (BPM) engine such as jBPM would be required.

It is important to know the types of events that will be processed through the mediator in order to make the correct choice for the implementation of the event mediator. Choosing Apache Camel for complex and long-running events involving human interaction would be extremely difficult to write and maintain. By the same token, using a BPM engine for simple event flows would take months of wasted effort when the same thing could be accomplished in Apache Camel in a matter of days.

Given that it's rare to have all events of one class of complexity, we recommend classifying events as simple, hard, or complex and having every event always go through a simple mediator (such as Apache Camel or Mule). The simple mediator can then interrogate the classification of the event, and based on that classification, handle the event itself or forward it to another, more complex, event mediator. In this manner, all types of

events can be effectively processed by the type of mediator needed for that event. This mediator delegation model is illustrated in Figure 14-6.

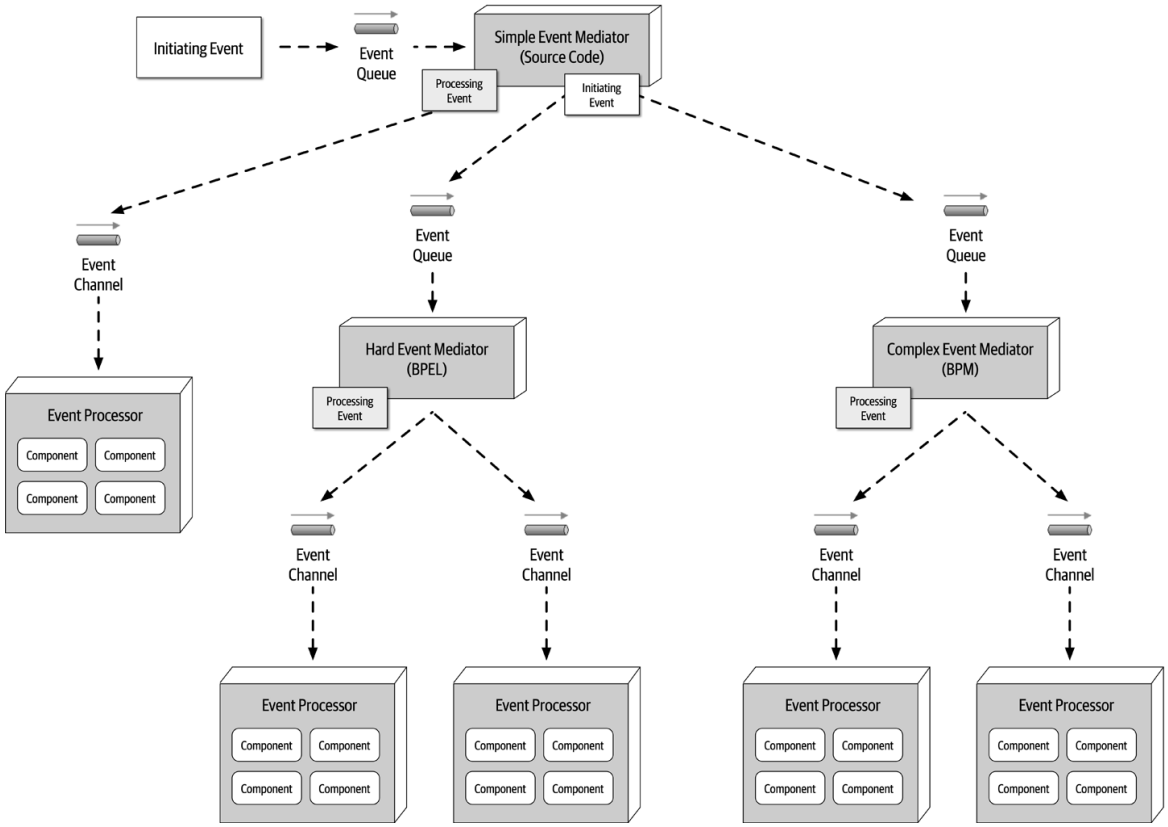

Figure 14-6. Delegating the event to the appropriate type of event mediator

Notice in Figure 14-6 that the `Simple Event Mediator` generates and sends a processing event when the event workflow is simple and can be handled by the simple mediator. However, notice that when the initiating event coming into the `Simple Event Mediator` is classified as either hard or complex, it forwards the original initiating event to the corresponding mediators (BPEL or BMP). The `Simple Event Mediator`, having intercepted the original event, may still be responsible for knowing when that event is complete, or it simply delegates the entire workflow (including client notification) to the other mediators.

To illustrate how the mediator topology works, consider the same retail order entry system example described in the prior broker topology section, but this time using the mediator topology. In this example, the mediator

knows the steps required to process this particular event. This event flow (internal to the mediator component) is illustrated in Figure 14-7.

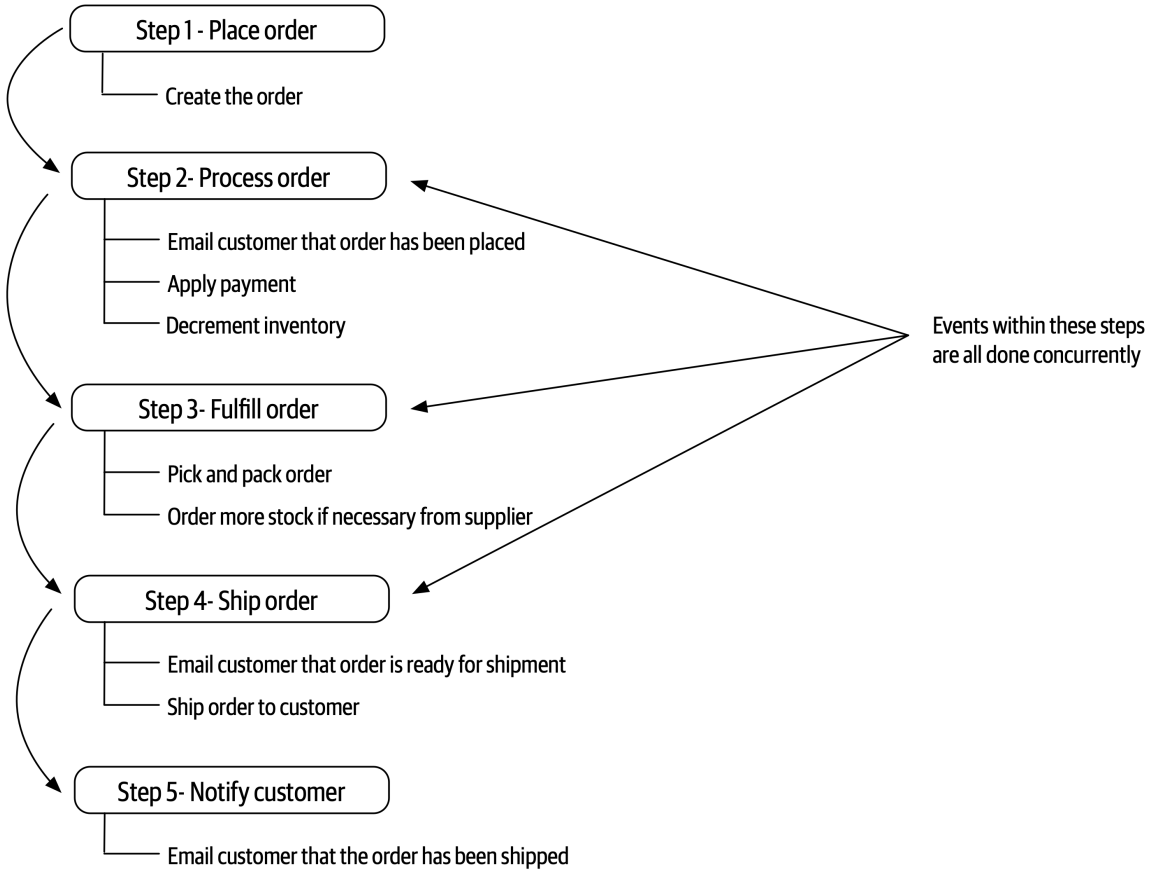

Figure 14-7. Mediator steps for placing an order

In keeping with the prior example, the same initiating event (`PlaceOrder`) is sent to the `customer-event-queue` for processing. The `Customer` mediator picks up this initiating event and begins generating processing events based on the flow in Figure 14-7. Notice that the multiple 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 the initiating event has been received, the `Customer` mediator generates a `create-order` processing event and sends this message to the `order-placement-queue` (see Figure 14-8). The `OrderPlacement` event processor accepts this event and validates and creates the order, returning to

the mediator an acknowledgement along with the order ID. At this point the mediator might send that order ID back to the customer, indicating that the order was placed, or it might have to continue until all the steps are complete (this would be based on specific business rules about order placement).

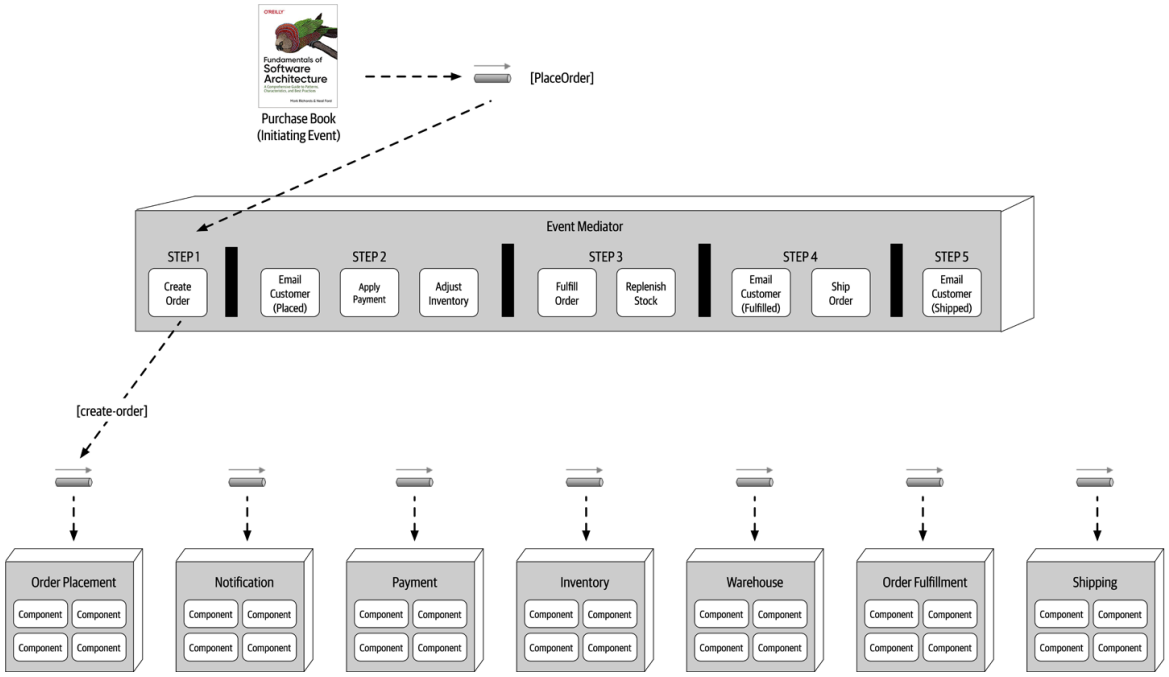

Figure 14-8. Step 1 of the mediator example

Now that step 1 is complete, the mediator now moves to step 2 (see Figure 14-9) and generates three messages at the same time: `email-customer`, `apply-payment`, and `adjust-inventory`. These processing events are all sent to their respective queues. All three event processors receive these messages, perform their respective tasks, and notify the mediator that the processing has been completed. Notice that the mediator must wait until it receives acknowledgement from all three parallel processes before moving on to step 3. At this point, if an error occurs in one of the parallel event processors, the mediator can take corrective action to fix the problem (this is discussed later in this section in more detail).

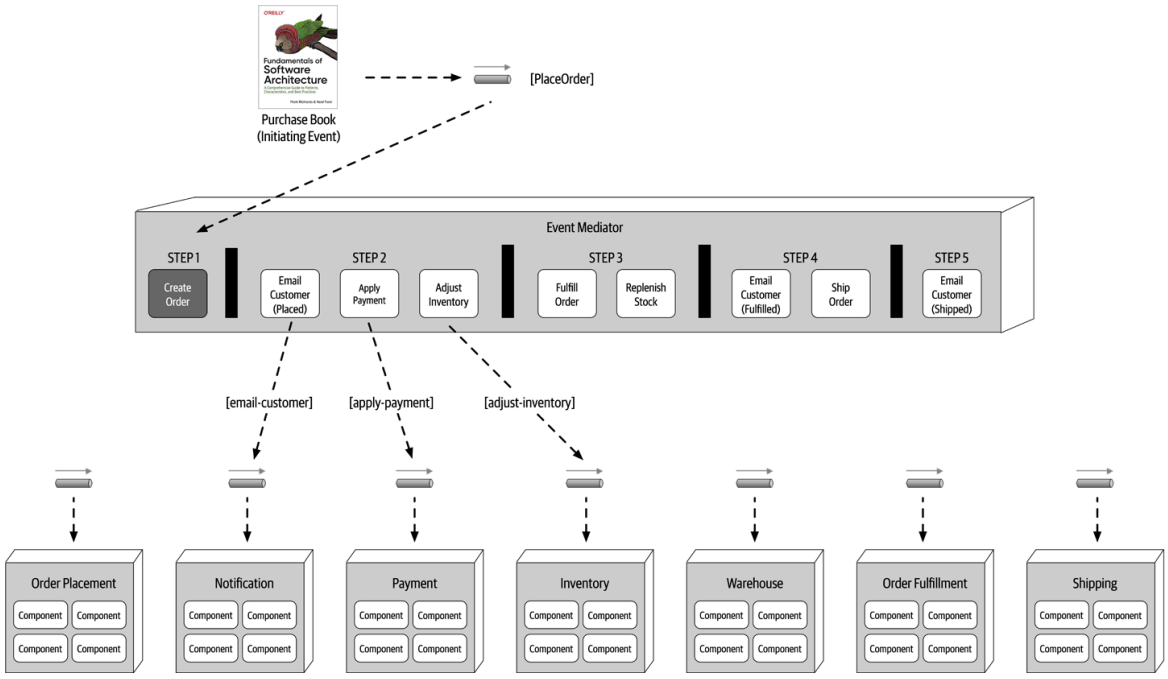

Figure 14-9. Step 2 of the mediator example

Once the mediator gets a successful acknowledgment from all of the event processors in step 2, it can move on to step 3 to fulfill the order (see Figure 14-10). Notice once again that both of these events (`fulfill-order` and `order-stock`) can occur simultaneously. The `OrderFulfillment` and `Warehouse` event processors accept these events, perform their work, and return an acknowledgement to the mediator.

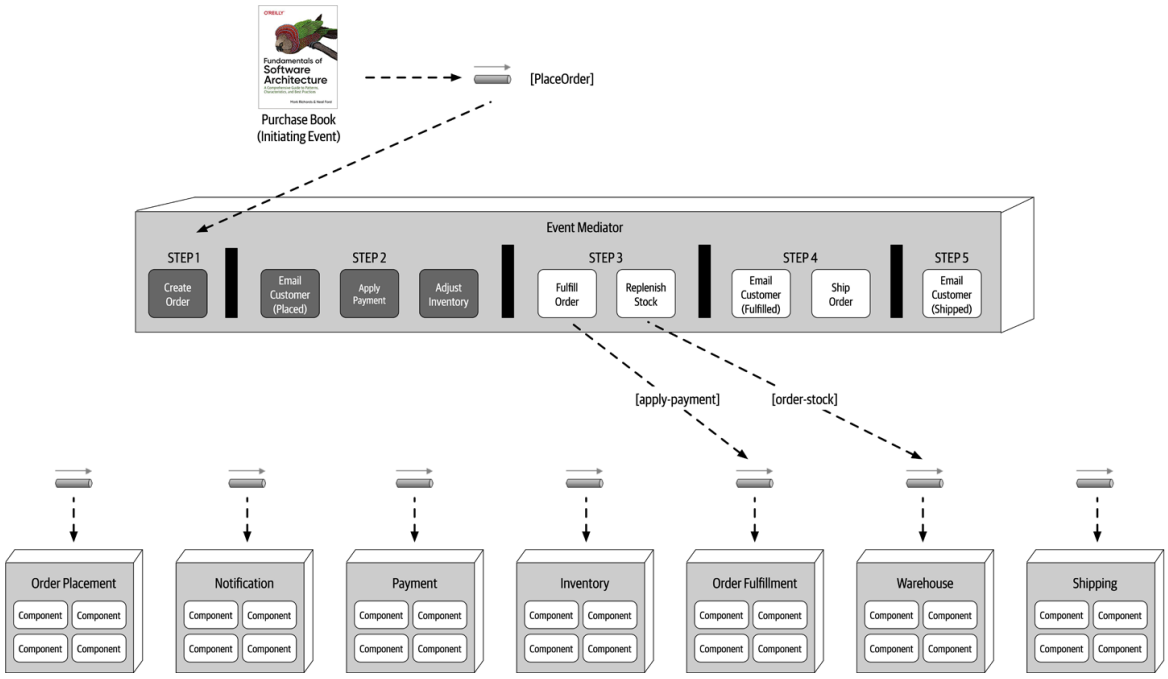

Figure 14-10. Step 3 of the mediator example

Once these events are complete, the mediator then moves on to step 4 (see Figure 14-11) to ship the order. This step generates another `email-customer` processing event with specific information about what to do (in this case, notify the customer that the order is ready to be shipped), as well as a `ship-order` event.

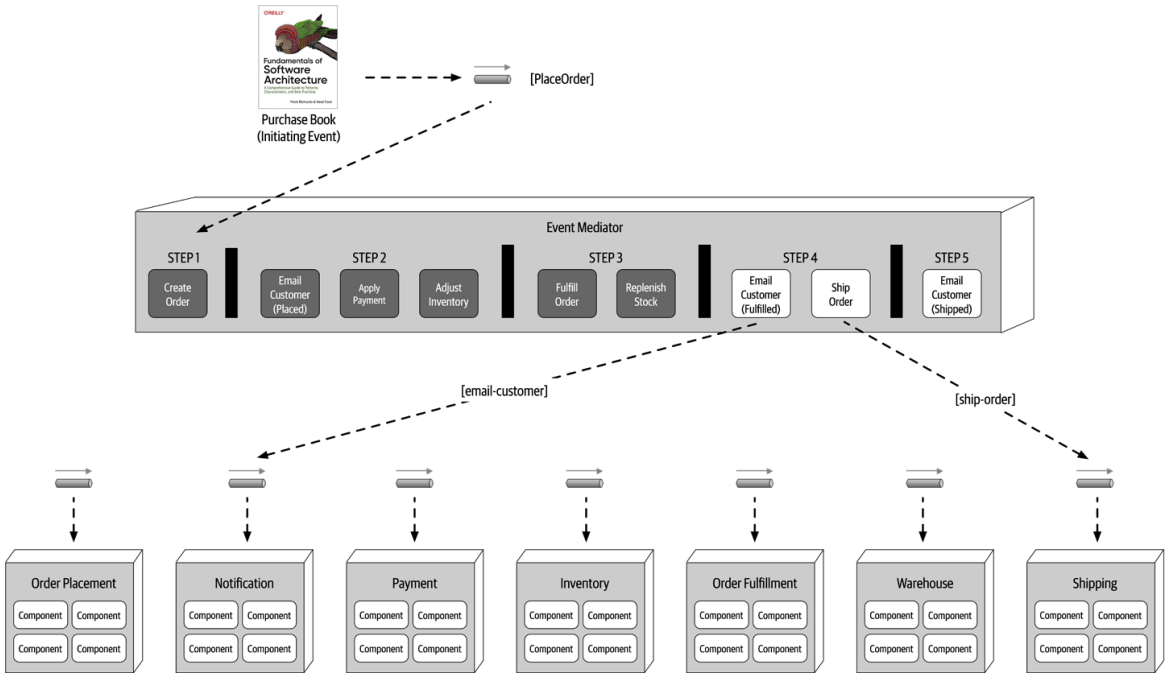

Figure 14-11. Step 4 of the mediator example

Finally, the mediator moves to step 5 (see Figure 14-12) and generates another contextual `email_customer` event to notify the customer that the order has been shipped. At this point the workflow is done, and the mediator marks the initiating event flow complete and removes all state associated with the initiating event.

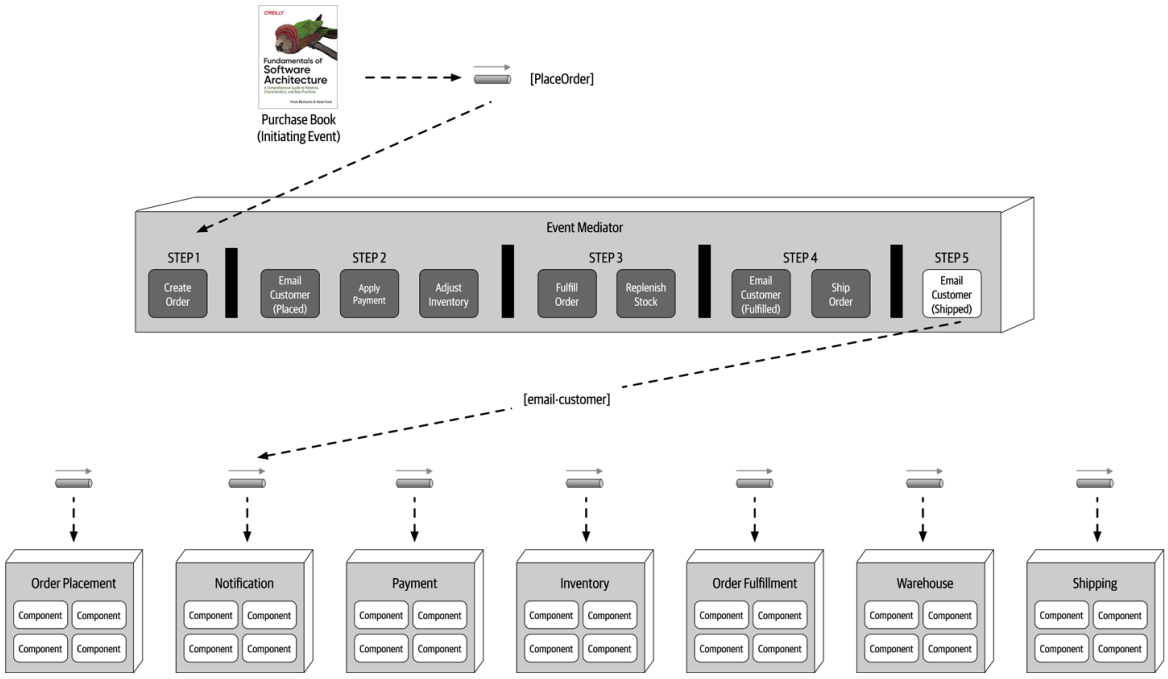

Figure 14-12. Step 5 of the mediator example

The mediator component has knowledge and control over the workflow, something the broker topology does not have. Because the mediator controls the workflow, it can maintain event state and manage error handling, recoverability, and restart capabilities. For example, suppose in the prior example the payment was not applied due to the credit card being expired. In this case the mediator receives this error condition, and knowing the order cannot be fulfilled (step 3) until payment is applied, 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).

Another inherent difference between the broker and mediator topology is how the processing events differ in terms of their meaning and how they are used. In the broker topology example in the previous section, the processing events were published as events that had occurred in the system (such as `order-created`, `payment-applied`, and `email-sent`). The event processors took some action, and other event processors react to that action. However, in the mediator topology, processing events such as `place-order`, `send-email`, and `fulfill-order` are *commands* (things that need

to happen) as opposed to *events* (things that have already happened). Also, in the mediator topology, a processing event must be processed (command), whereas it can be ignored in the broker topology (reaction).

While the mediator topology addresses the issues associated with the broker topology, there are some negatives associated with the mediator topology. 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 only handle the general processing, and a hybrid model combining both the mediator and broker topologies is used 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 broker topology, the mediator must scale as well, something that occasionally produces a bottleneck in the overall event processing flow. Finally, event processors are not as highly decoupled in the mediator topology as with the broker topology, and performance is not as good due to the mediator controlling the processing of the event. These trade-offs are summarized in Table 14-2.

Table 14-2. Trade-offs of the mediator topology

Advantages	Disadvantages
Workflow control	More coupling of event processors
Error handling	Lower scalability
Recoverability	Lower performance
Restart capabilities	Lower fault tolerance
Better data consistency	Modeling complex workflows

The choice between the broker and mediator topology essentially comes down to a trade-off between workflow control and error handling capability versus high performance and scalability. Although performance and

scalability are still good within the mediator topology, they are not as high as with the broker topology.

Asynchronous Capabilities

The event-driven architecture style offers a unique characteristic over other architecture styles in that it relies solely on asynchronous communication for both fire-and-forget processing (no response required) as well as request/reply processing (response required from the event consumer). Asynchronous communication can be a powerful technique for increasing the overall responsiveness of a system.

Consider the example illustrated in Figure 14-13 where a user is posting a comment on a website for a particular product review. Assume the comment service in this example takes 3,000 milliseconds to post the comment because it goes through several parsing engines: a bad word checker to check for unacceptable words, a grammar checker to make sure that the sentence structures are not saying something abusive, and finally a context checker to make sure the comment is about a particular product and not just a political rant. Notice in Figure 14-13 that the top path utilizes a synchronous RESTful call to post the comment: 50 milliseconds in latency for the service to receive the post, 3,000 milliseconds to post the comment, and 50 milliseconds in network latency to respond back to the user that the comment was posted. This creates a response time for the user of 3,100 milliseconds to post a comment. Now look at the bottom path and notice that with the use of asynchronous messaging, the response time from the end user's perspective for posting a comment on the website is only 25 milliseconds (as opposed to 3,100 milliseconds). It still takes 3,025 milliseconds to post the comment (25 milliseconds to receive the message and 3,000 milliseconds to post the comment), but from the end user's perspective it's already been done.

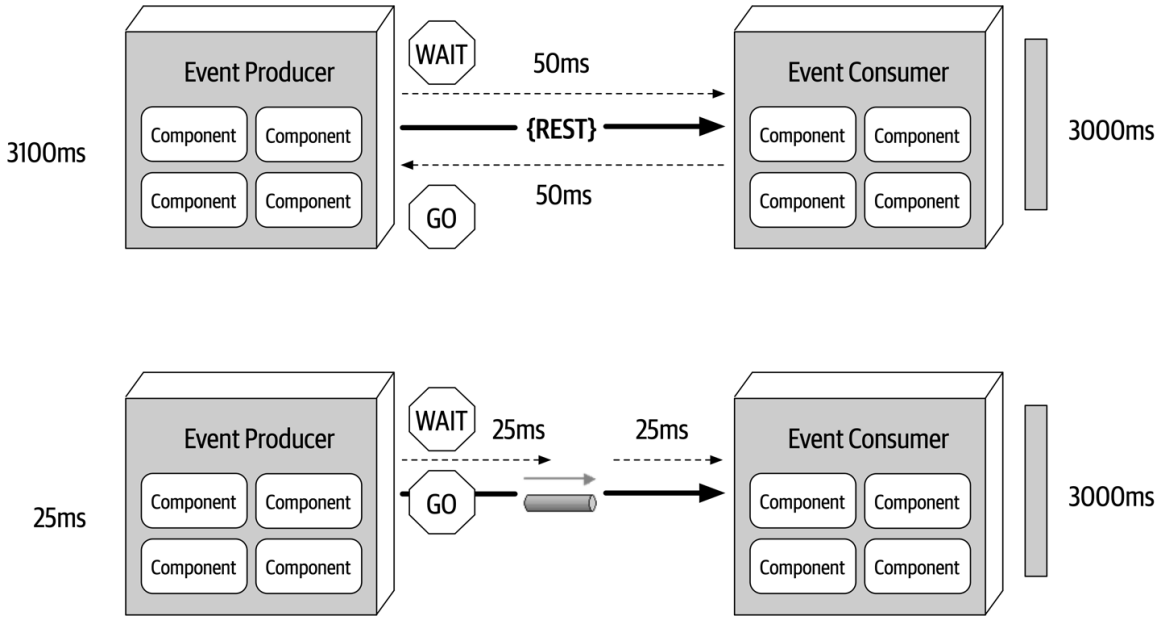

Figure 14-13. Synchronous versus asynchronous communication

This is a good example of the difference between *responsiveness* and *performance*. When the user does not need any information back (other than an acknowledgement or a thank you message), why make the user 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. Notice that nothing was done to optimize the way the comment service processes the text—in both cases it is still taking 3,000 milliseconds. Addressing *performance* would have been optimizing the comment service to run all of the text and grammar parsing engines in parallel with the use of caching and other similar techniques. The bottom example in Figure 14-13 addresses the overall responsiveness of the system but not the performance of the system.

The difference in response time between the two examples in Figure 14-13 from 3,100 milliseconds to 25 milliseconds is staggering. There is one caveat. On the synchronous path shown on the top of the diagram, the end user is guaranteed that the comment has been posted. However, on the bottom path there is only the acknowledgement of the post, with a future promise that eventually the comment will get posted. From the end user's perspective, the comment has been posted. But what happens if the user had typed a bad word in the comment? In this case the comment would be

rejected, but there is no way to get back to the end user. Or is there? In this example, assuming the user is registered with the website (which to post a comment they would have to be), a message could be sent to the user indicating a problem with the comment and some suggestions on how to repair it. This is a simple example. What about a more complicated example where the purchase of some stock is taking place asynchronously (called a stock trade) and there is no way to get back to the user?

The main issue with asynchronous communications is error handling. While responsiveness is significantly improved, it is difficult to address error conditions, adding to the complexity of the event-driven system. The next section addresses this issue with a pattern of reactive architecture called the *workflow event* pattern.

Error Handling

The workflow event pattern of reactive architecture is one way of addressing the issues associated with error handling in an asynchronous workflow. This pattern is a reactive architecture pattern that addresses both resiliency and responsiveness. In other words, the system can be resilient in terms of error handling without an impact to responsiveness.

The workflow event pattern leverages delegation, containment, and repair through the use of a *workflow delegate*, as illustrated in Figure 14-14. The event producer 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* and moves on to the next message in the event queue. In this way, overall responsiveness is not impacted because the next message is immediately processed. If the event consumer were to spend the time trying to figure out the error, then it is not reading the next message in the queue, therefore impacting the responsiveness not only of the next message, but all other messages waiting in the queue to be processed.

Once the workflow processor receives an error, it tries to figure out what is wrong with the message. This could be a static, deterministic error, or it

could leverage some machine learning algorithms to analyze the message to see some anomaly in the data. Either way, the workflow processor programmatically (without human intervention) makes changes to the original data to try and repair it, and then sends it back to the originating queue. The event consumer sees this message as a new one and tries to process it again, hopefully this time with some success. Of course, there are many times when the workflow processor cannot determine what is wrong with the message. In these cases the workflow processor sends the message off to another queue, which is then received in what is usually called a "dashboard," an application that looks similar to the Microsoft's Outlook or Apple's Mail. This dashboard usually resides on the desktop of a person of importance, who then looks at the message, applies manual fixes to it, and then resubmits it to the original queue (usually through a reply-to message header variable).

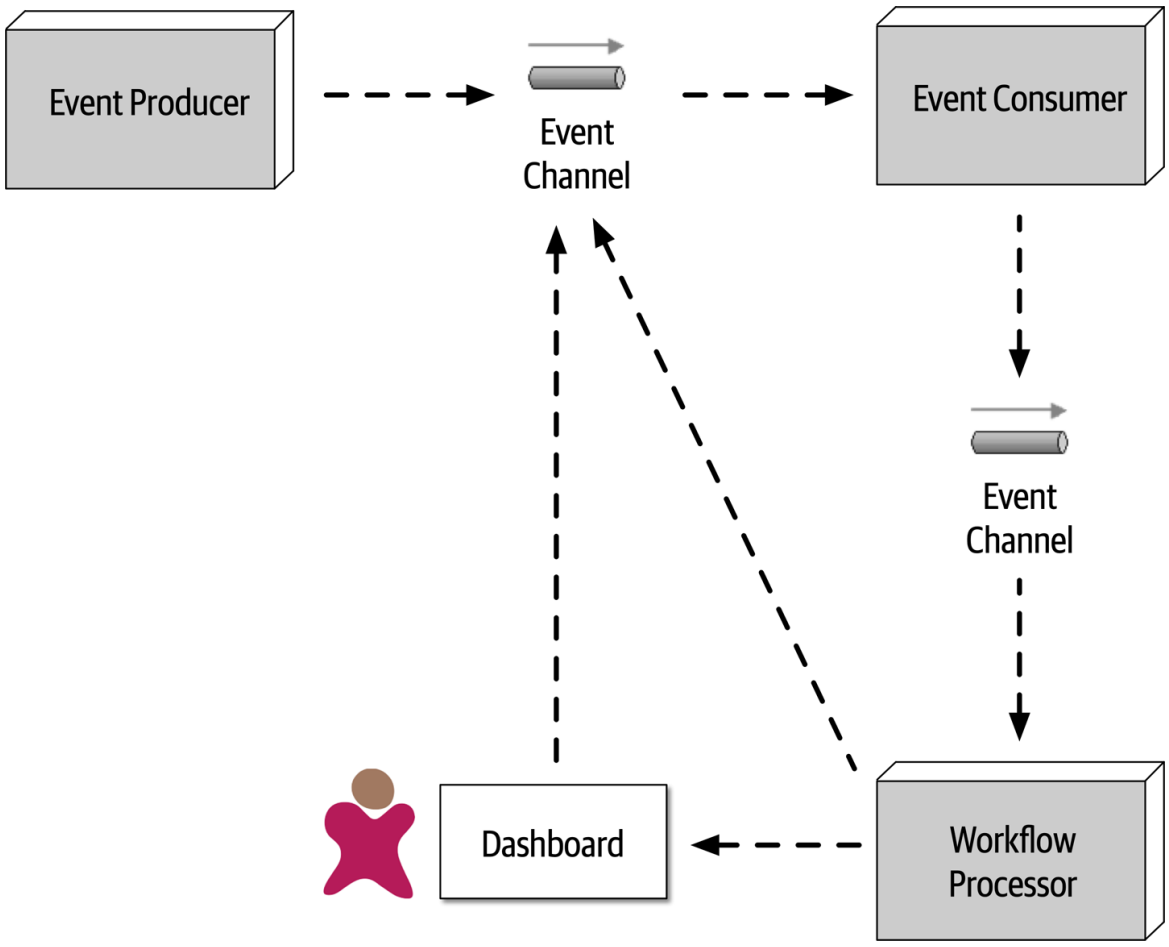

Figure 14-14. Workflow event pattern of reactive architecture

To illustrate the workflow event pattern, 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 (what is usually called a basket) and asynchronously sends those to the large trading firm to be placed with a broker so the stock can be purchased. 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
```

Notice the forth trade instruction (`2WE35HF6DHF,BUY,AAPL,8756 SHARES`) has the word `SHARES` after the number of shares for the trade. When these asynchronous trade orders are processed by the large trading firm without any error handling capabilities, the following error occurs within the trade placement 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, there is nothing that the trade placement service can do, because this was an asynchronous request, except to

possibly log the error condition. In other words, there is no user to synchronously respond to and fix the error.

Applying the workflow event pattern can programmatically fix this error. Because the large trading firm has no control over the trading advisor and the corresponding trade order data it sends, it must react to fix the error itself (as illustrated in Figure 14-15). When the same error occurs (`2WE35HF6DHF,BUY,AAPL,8756 SHARES`), the `Trade Placement` service immediately delegates the error via asynchronous messaging to the `Trade Placement Error` service for error handling, passing 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 it is an issue 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 fixed trade is then processed successfully by the trade placement service:

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

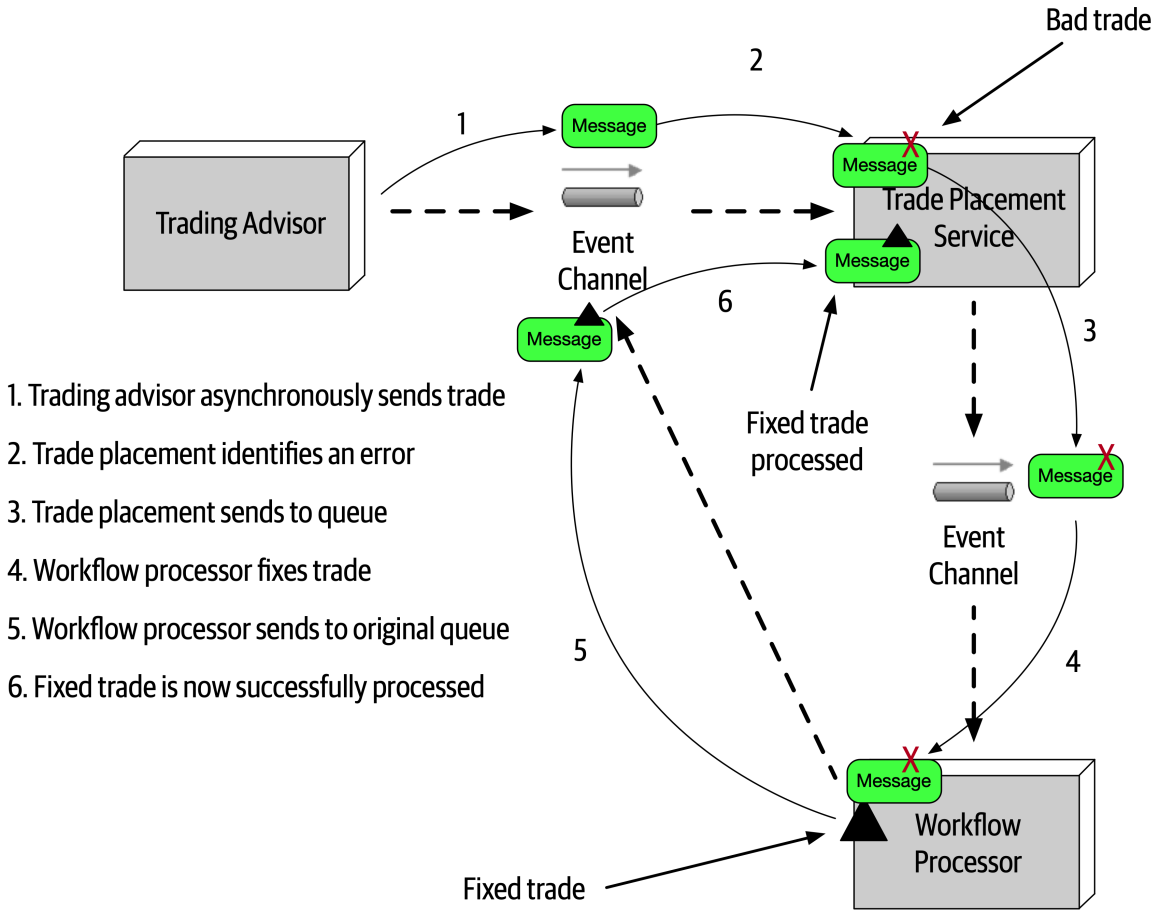

Figure 14-15. Error handling with the workflow event pattern

One of the consequences of the workflow event pattern is that messages in error are processed out of sequence when they are resubmitted. In our trading example, the order of 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). Although not impossible, it is a complex task to maintain message order within a given context (in this case the brokerage account number). One way this can be addressed is by the `Trade Placement` service queueing and storing the account number of the trade in error. Any trade with that same account number would be stored in a temporary queue for later processing (in FIFO order). Once the trade originally in error is fixed and processed, the `Trade Placement` service then de-queues the remaining trades for that same account and processes them in order.

Preventing Data Loss

Data loss is always a primary concern when dealing with asynchronous communications. Unfortunately, there are many places for data loss to occur within an event-driven architecture. By data loss we mean a message getting dropped or never making it to its final destination. Fortunately, there are basic out-of-the-box techniques that can be leveraged to prevent data loss when using asynchronous messaging.

To illustrate the issues associated with data loss within event-driven architecture, suppose `Event Processor A` asynchronously sends a message to a queue. `Event Processor B` accepts the message and inserts the data within the message into a database. As illustrated in Figure 14-16, three areas of data loss can occur within this typical scenario:

1. The message never makes it to the queue from `Event Processor A`; or even if it does, the broker goes down before the next event processor can retrieve the message.
2. `Event Processor B` de-queues the next available message and crashes before it can process the event.
3. `Event Processor B` is unable to persist the message to the database due to some data error.

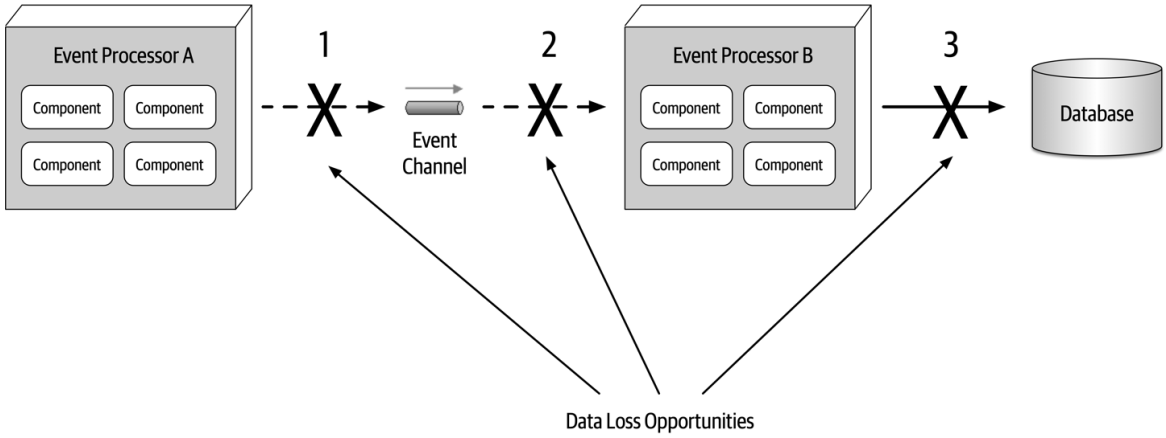

Figure 14-16. Where data loss can happen within an event-driven architecture

Each of these areas of data loss can be mitigated through basic messaging techniques. Issue 1 (the message never makes it to the queue) is easily solved by leveraging persisted message queues, along with something called *synchronous send*. Persisted message queues support what is known as guaranteed delivery. When the message broker receives the message, it not only stores it in memory for fast retrieval, but also persists the message in some sort of physical data store (such as a filesystem or database). If the message broker goes down, the message is physically stored on disk so that when the message broker comes back up, the message is available for processing. Synchronous send does a blocking wait in the message producer until the broker has acknowledged that the message has been persisted. With these two basic techniques there is no way to lose a message between the event producer and the queue because the message is either still with the message producer or persisted within the queue.

Issue 2 (`Event Processor B` de-queues the next available message and crashes before it can process the event) can also be solved using a basic technique of messaging called *client acknowledge mode*. By default, when a message is de-queued, it is immediately removed from the queue (something called *auto acknowledge* mode). Client acknowledge mode keeps the message in the queue and attaches the client ID to the message so that no other consumers can read the message. With this mode, if `Event Processor B` crashes, the message is still preserved in the queue, preventing message loss in this part of the message flow.

Issue 3 (`Event Processor B` is unable to persist the message to the database due to some data error) is addressed through leveraging ACID (atomicity, consistency, isolation, durability) transactions via a database commit. Once the database commit happens, the data is guaranteed to be persisted in the database. Leveraging something called *last participant support* (LPS) removes the message from the persisted queue by acknowledging that processing has been completed and that the message has been persisted. This guarantees the message is not lost during the transit from `Event Processor A` all the way to the database. These techniques are illustrated in Figure 14-17.

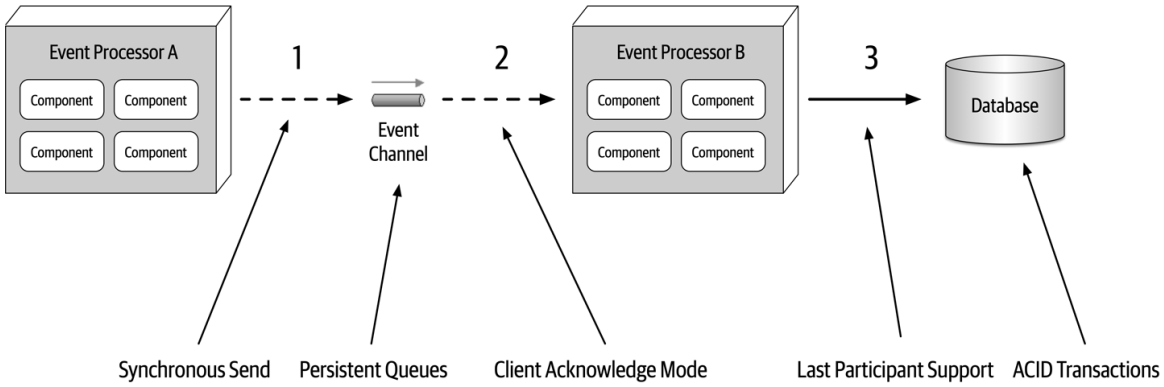

Figure 14-17. Preventing data loss within an event-driven architecture

Broadcast Capabilities

One of the other unique characteristics of event-driven architecture is the capability to broadcast events without knowledge of who (if anyone) is receiving the message and what they do with it. This technique, which is illustrated in Figure 14-18, shows that when a producer publishes a message, that same message is received by multiple subscribers.

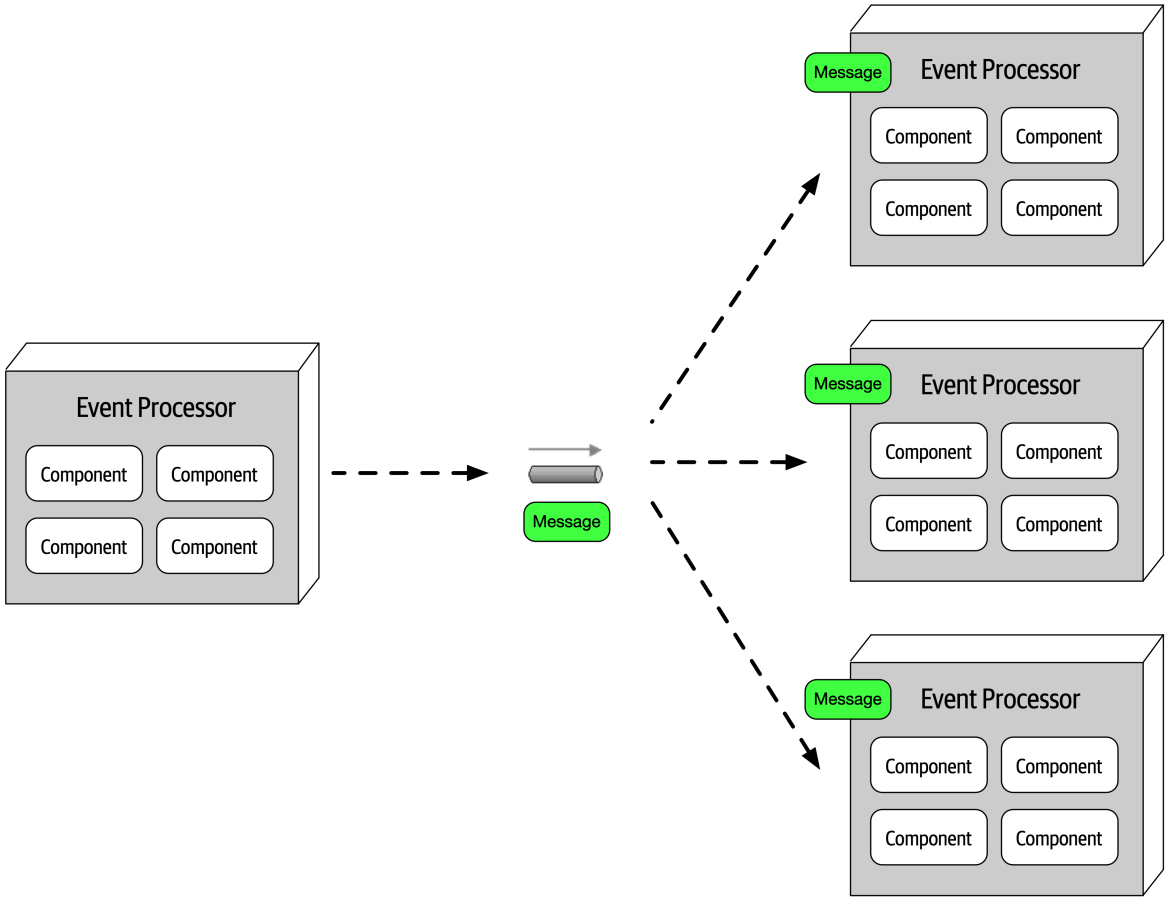

Figure 14-18. Broadcasting events to other event processors

Broadcasting is perhaps the highest level of decoupling between event processors because the producer of the broadcast message usually does not know which event processors will be receiving the broadcast message and more importantly, what they will do with the message. Broadcast capabilities are an essential part of patterns for eventual consistency, complex event processing (CEP), and a host of other situations. Consider frequent changes in stock prices for instruments traded on the stock market. Every ticker (the current price of a particular stock) might influence a number of things. However, the service publishing the latest price simply broadcasts it with no knowledge of how that information will be used.

Request-Reply

So far in this chapter we've dealt with asynchronous requests that don't need an immediate response from the event consumer. But what if an order ID is needed when ordering a book? What if a confirmation number is needed when booking a flight? These are examples of communication between services or event processors that require some sort of synchronous communication.

In event-driven architecture, synchronous communication is 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 initial request for information is asynchronously sent to the request queue, and then control is returned to the message producer. The message producer then does a blocking wait on the reply queue, waiting for the response. The message consumer receives and processes the message and then sends the response to the reply queue. The event producer then receives the message with the response data. This basic flow is illustrated in Figure 14-19.

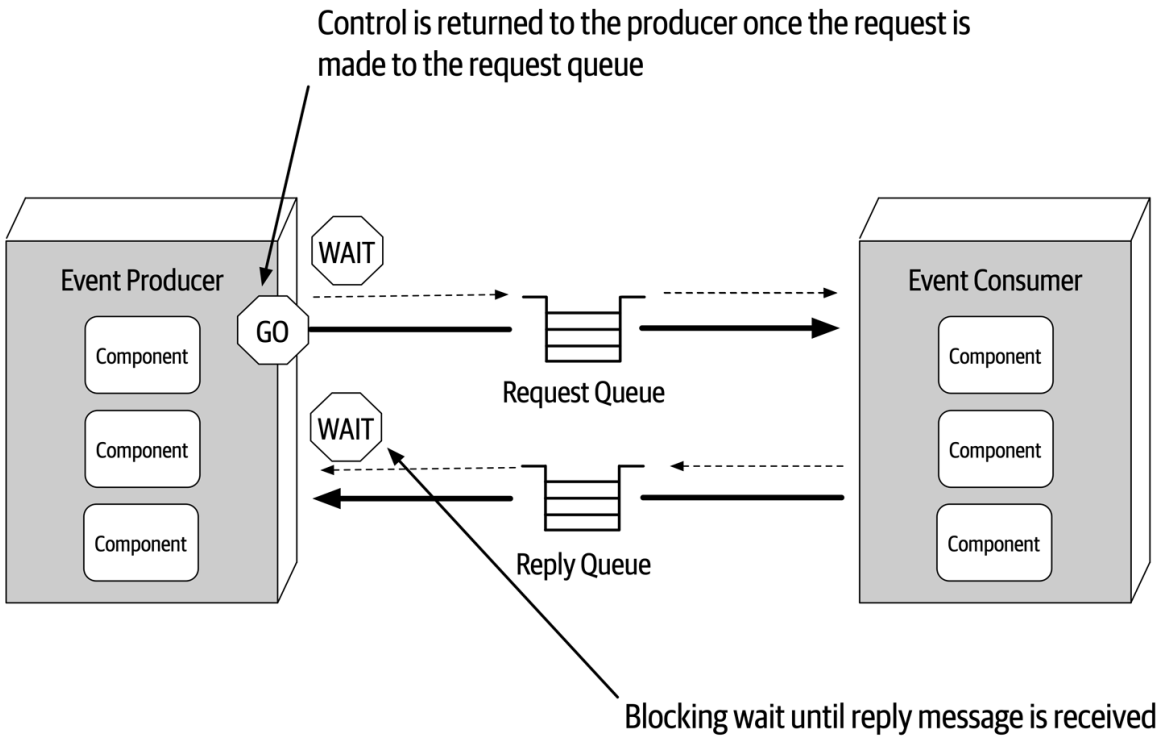

Figure 14-19. Request-reply message processing

There are two primary techniques for implementing request-reply messaging. The first (and most common) technique is to use a *correlation ID* contained in the message header. A correlation ID is a field in the reply message that is usually set to the message ID of the original request message. This technique, as illustrated in Figure 14-20, works as follows, with the message ID indicated with ID, and the correlation ID indicated with CID:

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

2. The event producer now does a blocking wait on the reply queue with a message filter (also called a message selector), where the correlation ID in the message header equals the original message ID (in this case 124). Notice there are two messages in the reply queue: message ID 855 with correlation ID 120, and message ID 856 with correlation ID 122. Neither of these messages will be

picked up because the correlation ID does not match 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 correlation ID (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 correlation ID (124) matches the message selector from step 2.

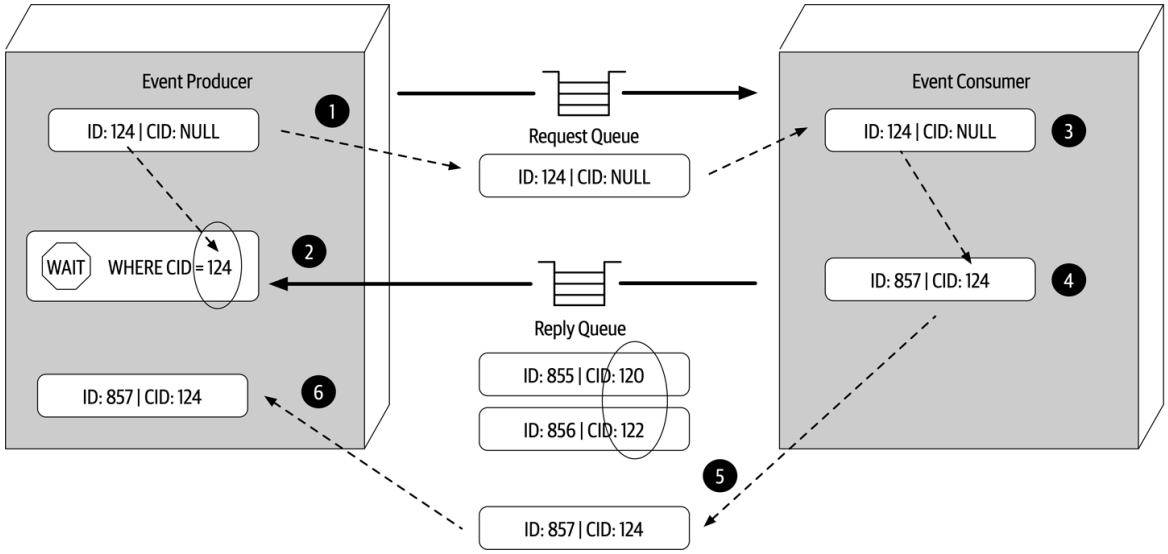

Figure 14-20. Request-reply message processing using a correlation ID

The other technique used to implement request-reply messaging is to use a *temporary queue* for the reply queue. A temporary queue is dedicated to the specific request, created when the request is made and deleted when the request ends. This technique, as illustrated in Figure 14-21, does not require a correlation ID because the temporary queue is a dedicated queue only known to the event producer for the 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 originally sent to the 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.

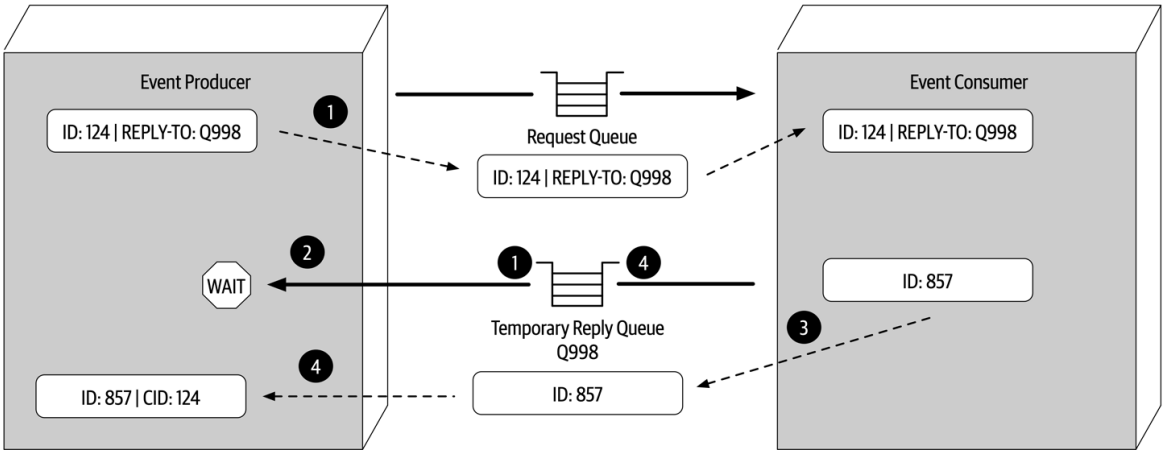

Figure 14-21. Request-reply message processing using a temporary queue

While the temporary queue technique is much simpler, the message broker must create a temporary queue for each request made and then delete it immediately afterward. Large messaging volumes can significantly slow down the message broker and impact overall performance and responsiveness. For this reason we usually recommend using the correlation ID technique.

Choosing Between Request-Based and Event-Based

The request-based model and event-based model are both viable approaches for designing software systems. However, choosing the right model is essential to the overall success of the system. We recommend choosing the request-based model for well-structured, data-driven requests (such as retrieving customer profile data) when certainty and control over the workflow is needed. 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 trade-offs with the event-based model also helps decide which one is the best fit. Table 14-3 lists the advantages and disadvantages of the event-based model of event-driven architecture.

Table 14-3. 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	

Hybrid Event-Driven Architectures

While many applications leverage the event-driven architecture style as the primary overarching architecture, in many cases event-driven architecture is

used in conjunction with other architecture styles, forming what is known as a hybrid architecture. Some common architecture styles that leverage event-driven architecture as part of another architecture style include microservices and space-based architecture. Other hybrids that are possible include an event-driven microkernel architecture and an event-driven pipeline architecture.

Adding event-driven architecture to any architecture style helps remove bottlenecks, provides a back pressure point in the event requests get backed up, and provides a level of user responsiveness not found in other architecture styles. Both microservices and space-based architecture leverage messaging for data pumps, asynchronously sending data to another processor that in turn updates data in a database. Both also leverage event-driven architecture to provide a level of programmatic scalability to services in a microservices architecture and processing units in a space-based architecture when using messaging for interservice communication.

Architecture Characteristics Ratings

A one-star rating in the characteristics ratings table in Figure 14-22 means the specific architecture characteristic isn’t well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in Chapter 4.

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1 to many
Deployability	★★★
Elasticity	★★★
Evolutionary	★★★★★
Fault tolerance	★★★★★
Modularity	★★★★
Overall cost	★★★
Performance	★★★★★
Reliability	★★★
Scalability	★★★★★
Simplicity	★
Testability	★★

Figure 14-22. Event-driven architecture characteristics ratings

Event-driven architecture is primarily a technically partitioned architecture in that any particular domain is spread across multiple event processors and tied together through mediators, queues, and topics. Changes to a particular domain usually impact many event processors, mediators, and other messaging artifacts, hence why event-driven architecture is not domain partitioned.

The number of quanta within event-driven architecture can vary from one to many quanta, which is usually based on the database interactions within each event processor and request-reply processing. Even though all communication in an event-driven architecture is asynchronous, 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 is still asynchronous between the event processors, if a request is needed right away from the event consumer, it ties those event processors together synchronously; hence they belong to the same quantum.

To illustrate this point, consider the example where 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 that places the order and generates an order ID is down, the first event processor cannot continue. Therefore, 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 gains five stars for performance, scalability, and fault tolerance, the primary strengths of this architecture style. High performance is achieved through asynchronous communications combined with highly parallel processing. High scalability is realized through the programmatic load balancing of event processors (also called *competing consumers*). As the request load increases, additional event processors can be programmatically added to handle the additional requests. Fault tolerance is achieved through highly decoupled and asynchronous event processors that provide eventual consistency and eventual processing of event workflows. Providing the user interface or an event processor making a request does not need an immediate response, promises and futures can be leveraged to process the event at a later time if other downstream processors are not available.

Overall *simplicity* and *testability* rate relatively low with event-driven architecture, mostly due to the nondeterministic and dynamic event flows

typically found within this architecture style. While deterministic flows within the request-based model are relatively easy to test because the paths and outcomes are generally known, such is not the case with the event-driven model. Sometimes it is not known how event processors will react to dynamic events, and what messages they might produce. These “event tree diagrams” can be extremely complex, generating hundreds to even thousands of scenarios, making it very difficult to govern and test.

Finally, event-driven architectures are highly evolutionary, hence the five-star rating. Adding new features through existing or new event processors is relatively straightforward, particularly in the broker topology. By providing hooks via published messages in the broker topology, the data is already made available, hence no changes are required in the infrastructure or existing event processors to add that new functionality.

Chapter 15. Space-Based Architecture Style

Most web-based business applications follow the same general request flow: a request from a browser hits the web server, then an application server, then finally the database server. While this pattern works great for a small set of users, bottlenecks start appearing as the user load increases, first at the web-server layer, then at the application-server layer, and finally at the database-server layer. The usual response to bottlenecks based on an increase in user load is to scale out the web servers. This is relatively easy and inexpensive, and it sometimes works to address the bottleneck issues. However, in most cases of high user load, scaling out the web-server layer just moves the bottleneck down to the application server. Scaling application servers can be more complex and expensive than web servers and usually just moves the bottleneck down to the database server, which is even more difficult and expensive to scale. Even if you can scale the database, what you eventually end up with is a triangle-shaped topology, with the widest part of the triangle being the web servers (easiest to scale) and the smallest part being the database (hardest to scale), as illustrated in Figure 15-1.

In any high-volume application with a large concurrent user load, the database will usually be the final limiting factor in how many transactions you can process concurrently. While various caching technologies and database scaling products help to address these issues, the fact remains that scaling out a normal application for extreme loads is a very difficult proposition.

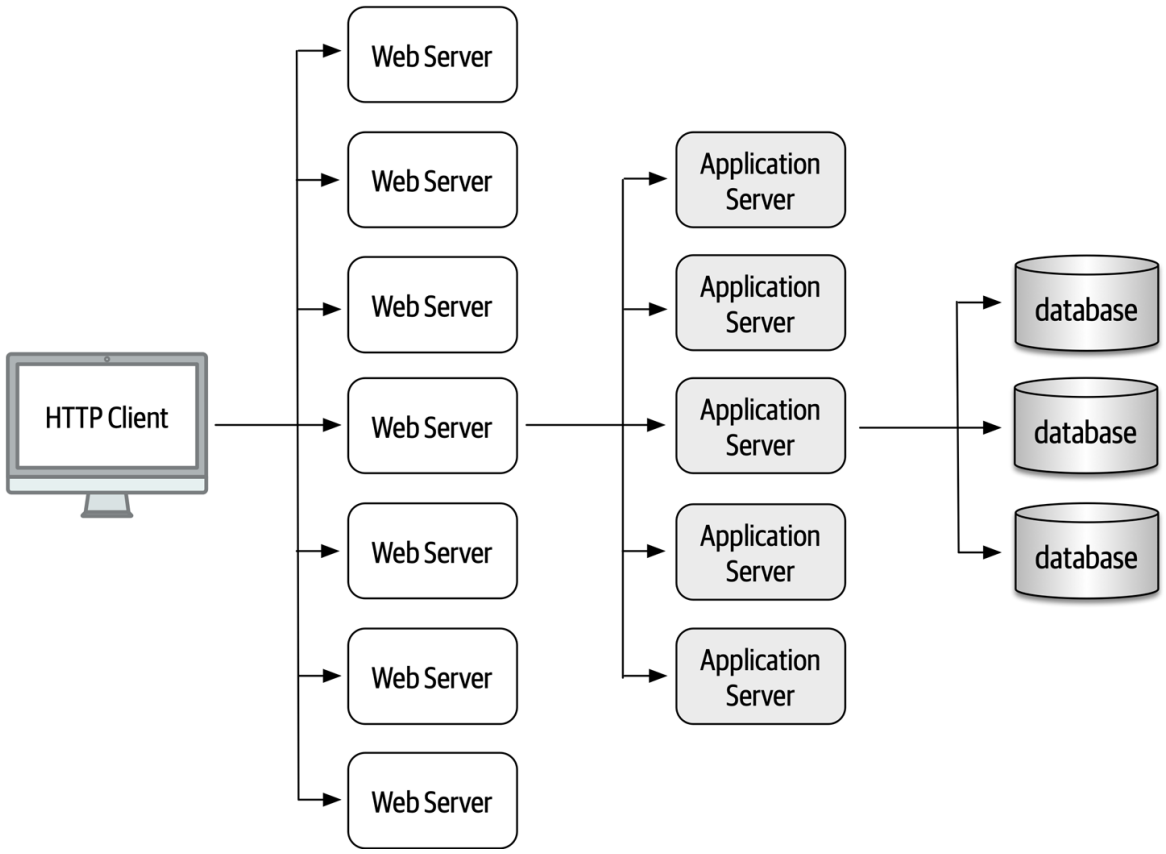

Figure 15-1. Scalability limits within a traditional web-based topology

The *space-based* architecture style is specifically designed to address problems involving high scalability, elasticity, and high concurrency issues. It is also a useful architecture style for applications that have variable and unpredictable concurrent user volumes. Solving the extreme and variable scalability issue architecturally is often a better approach than trying to scale out a database or retrofit caching technologies into a nonscalable architecture.

General Topology

Space-based architecture gets its name from the concept of *tuple space*, the technique of using multiple parallel processors communicating through shared memory. High scalability, high elasticity, and high performance are achieved by removing the central database as a synchronous constraint in the system and instead leveraging replicated in-memory data grids.

Application data is kept in-memory and replicated among all the active processing units. When a processing unit updates data, it asynchronously sends that data to the database, usually via messaging with persistent queues. Processing units start up and shut down dynamically as user load increases and decreases, thereby addressing variable scalability. Because there is no central database involved in the standard transactional processing of the application, the database bottleneck is removed, thus providing near-infinite scalability within the application.

There are several architecture components that make up a space-based architecture: a *processing unit* containing the application code, *virtualized middleware* used to manage and coordinate the processing units, *data pumps* to asynchronously send updated data to the database, *data writers* that perform the updates from the data pumps, and *data readers* that read database data and deliver it to processing units upon startup. Figure 15-2 illustrates these primary architecture components.

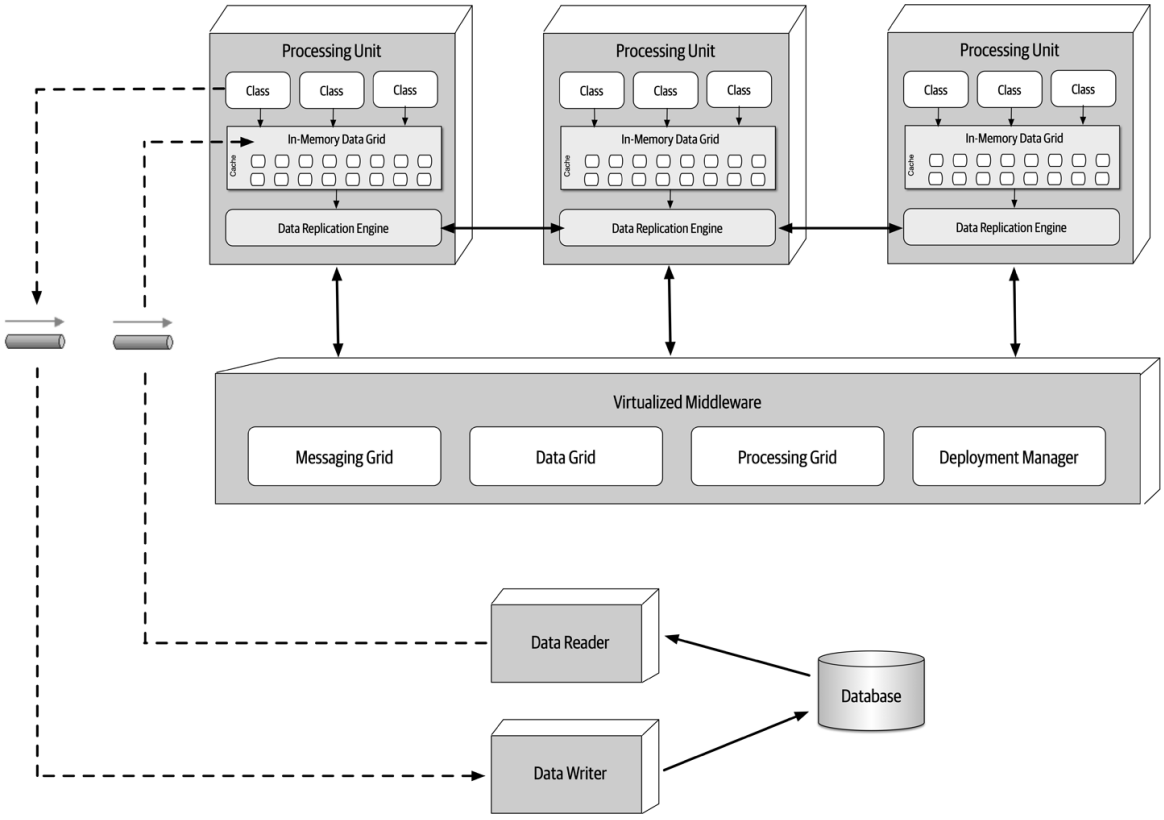

Figure 15-2. Space-based architecture basic topology

Processing Unit

The processing unit (illustrated in Figure 15-3) contains the application logic (or portions of the application logic). This usually includes web-based components as well as backend business logic. The contents of the processing unit vary based on the type of application. Smaller web-based applications would likely be deployed into a single processing unit, whereas larger applications may split the application functionality into multiple processing units based on the functional areas of the application. The processing unit can also contain small, single-purpose services (as with microservices). In addition to the application logic, the processing unit also contains an in-memory data grid and replication engine usually implemented through such products as Hazelcast, Apache Ignite, and Oracle Coherence.

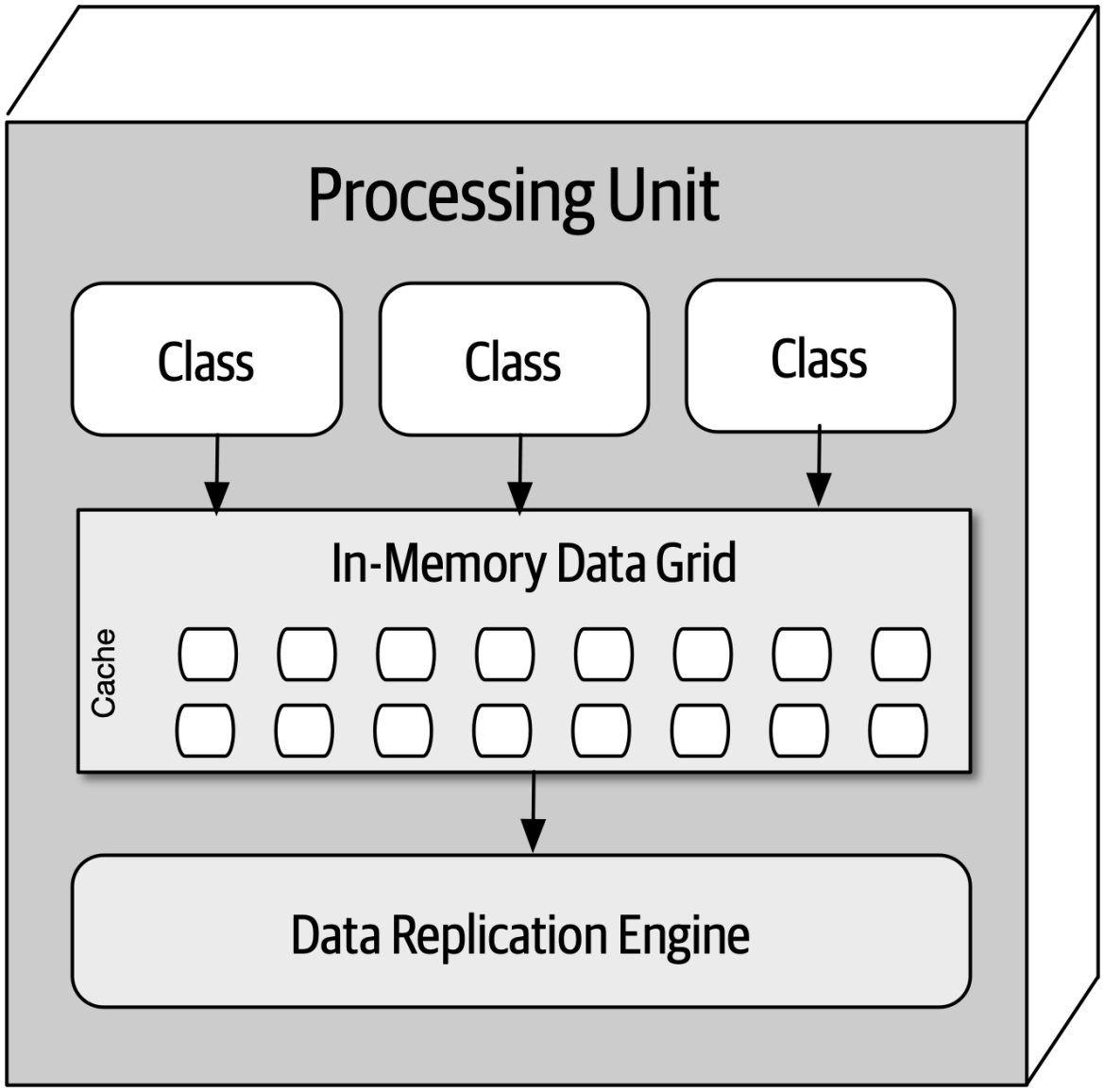

Figure 15-3. Processing unit

Virtualized Middleware

The virtualized middleware handles the infrastructure concerns within the architecture that control various aspects of data synchronization and request handling. The components that make up the virtualized middleware include a *messaging grid*, *data grid*, *processing grid*, and *deployment manager*. These components, which are described in detail in the next sections, can be custom written or purchased as third-party products.

Messaging grid

The messaging grid, shown in Figure 15-4, manages input request and session state. When a request comes into the virtualized middleware, the messaging grid component determines which active processing components are available to receive the request and forwards the request to one of those processing units. The complexity of the messaging grid can range from a simple round-robin algorithm to a more complex next-available algorithm that keeps track of which request is being processed by which processing unit. This component is usually implemented using a typical web server with load-balancing capabilities (such as HA Proxy and Nginx).

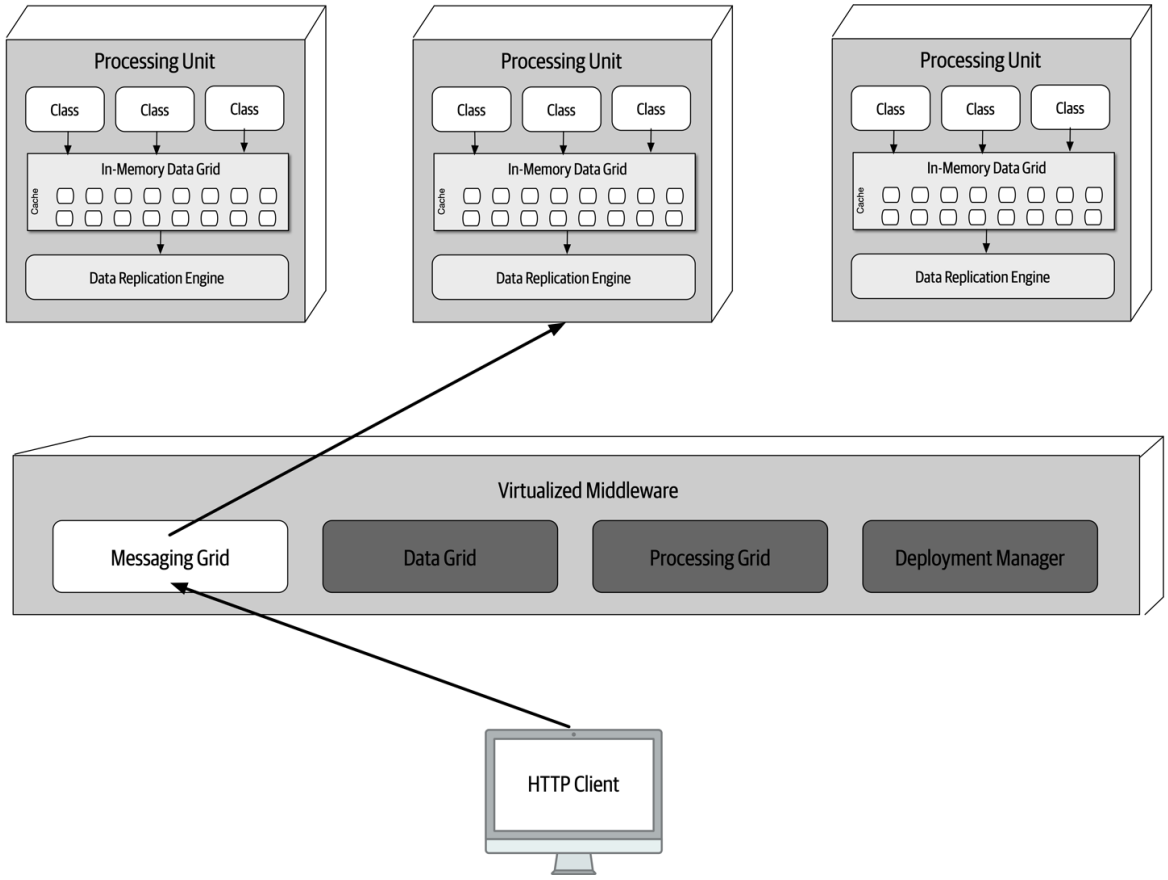

Figure 15-4. Messaging grid

Data grid

The data grid component is perhaps the most important and crucial component in this architecture style. In most modern implementations the

data grid is implemented solely within the processing units as a replicated cache. However, for those replicated caching implementations that require an external controller, or when using a distributed cache, this functionality would reside in both the processing units as well as in the data grid component within the virtualized middleware. Since the messaging grid can forward a request to any of the processing units available, it is essential that each processing unit contains exactly the same data in its in-memory data grid. Although Figure 15-5 shows a synchronous data replication between processing units, in reality this is done asynchronously and very quickly, usually completing the data synchronization in less than 100 milliseconds.

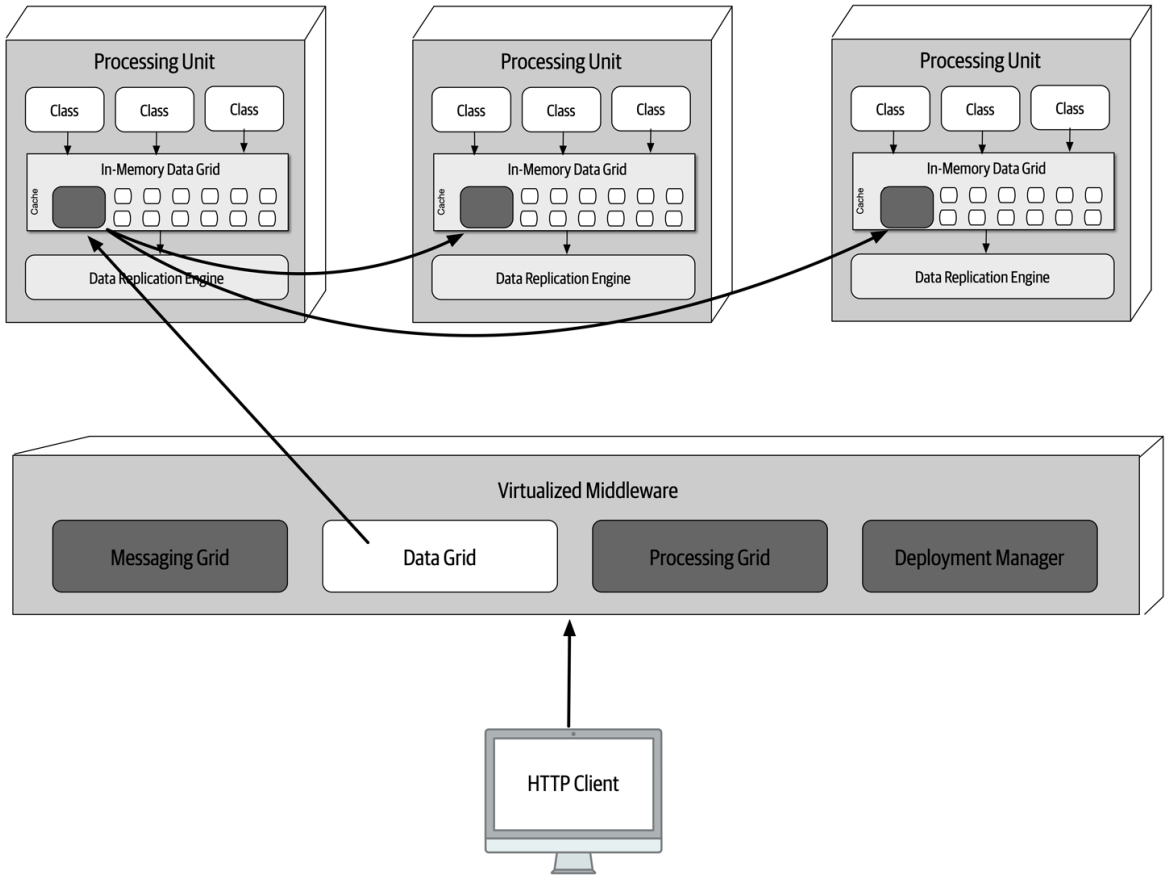

Figure 15-5. Data grid

Data is synchronized between processing units that contain the same named data grid. To illustrate this point, consider the following code in Java using Hazelcast that creates an internal replicated data grid for processing units containing customer profile information:

```
HazelcastInstance hz = Hazelcast.newHazelcastInstance();
Map<String, CustomerProfile> profileCache =
        hz.getReplicatedMap("CustomerProfile");
```

All processing units needing access to the customer profile information would contain this code. Changes made to the `CustomerProfile` named cache from any of the processing units would have that change replicated to all other processing units containing that same named cache. A processing unit can contain as many replicated caches as needed to complete its work. Alternatively, one processing unit can make a remote call to another processing unit to ask for data (choreography) or leverage the processing grid (described in the next section) to orchestrate the request.

Data replication within the processing units also allows service instances to come up and down without having to read data from the database, providing there is at least one instance containing the named replicated cache. When a processing unit instance comes up, it connects to the cache provider (such as Hazelcast) and makes a request to get the named cache. Once the connection is made to the other processing units, the cache will be loaded from one of the other instances.

Each processing unit knows about all other processing unit instances through the use of a *member list*. The member list contains the IP address and ports of all other processing units using that same named cache. For example, suppose there is a single processing instance containing code and replicated cached data for the customer profile. In this case there is only one instance, so the member list for that instance only contains itself, as illustrated in the following logging statements generated using Hazelcast:

```
Instance 1:
Members {size:1, ver:1} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
this
]
```

When another processing unit starts up with the same named cache, the member list of both services is updated to reflect the IP address and port of

each processing unit:

```
Instance 1:
Members {size:2, ver:2} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
this
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
]

Instance 2:
Members {size:2, ver:2} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
this
]
```

When a third processing unit starts up, the member list of instance 1 and instance 2 are both updated to reflect the new third instance:

```
Instance 1:
Members {size:3, ver:3} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
this
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
]

Instance 2:
Members {size:3, ver:3} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
this
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
]

Instance 3:
Members {size:3, ver:3} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.90]:5702 - ea9e4dd5-5cb3-4b27-8fe8-db5cc62c7316
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
this
]
```

Notice that all three instances know about each other (including themselves). Suppose instance 1 receives a request to update the customer profile information. When instance 1 updates the cache with a `cache.put()` or similar cache update method, the data grid (such as Hazelcast) will asynchronously update the other replicated caches with the same update, ensuring all three customer profile caches always remain in sync with one another.

When processing unit instances go down, all other processing units are automatically updated to reflect the lost member. For example, if instance 2 goes down, the member lists of instance 1 and 3 are updated as follows:

```
Instance 1:
Members {size:2, ver:4} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
this
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
]

Instance 3:
Members {size:2, ver:4} [
        Member [172.19.248.89]:5701 - 04a6f863-dfce-41e5-9d51-9f4e356ef268
        Member [172.19.248.91]:5703 - 1623eadf-9cfb-4b83-9983-d80520cef753
this
]
```

Processing grid

The processing grid, illustrated in Figure 15-6, is an optional component within the virtualized middleware that manages orchestrated request processing when there are multiple processing units involved in a single business request. If a request comes in that requires coordination between processing unit types (e.g., an order processing unit and a payment processing unit), it is the processing grid that mediates and orchestrates the request between those two processing units.

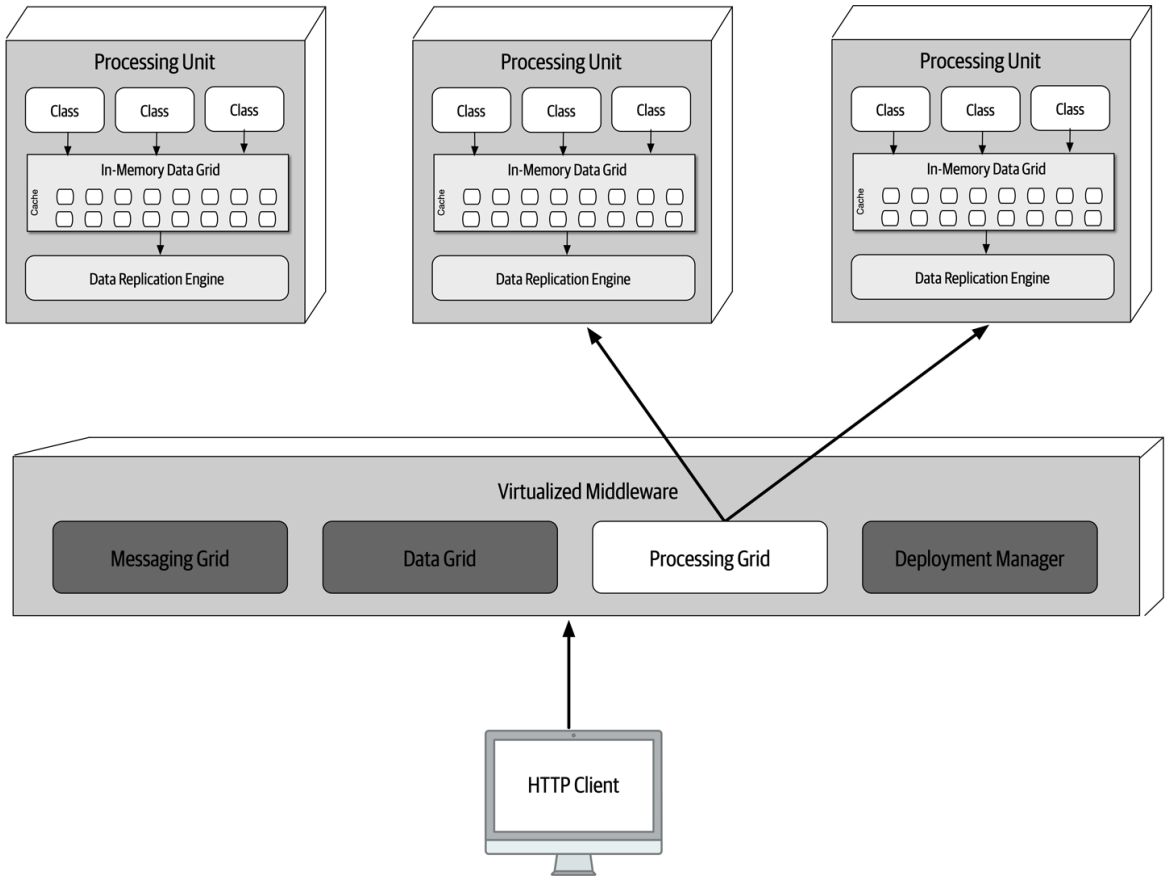

Figure 15-6. Processing grid

Deployment manager

The deployment manager component manages the dynamic startup and shutdown of processing unit instances based on load conditions. This component continually monitors response times and user loads, starts up new processing units when load increases, and shuts down processing units when the load decreases. It is a critical component to achieving variable scalability (elasticity) needs within an application.

Data Pumps

A *data pump* is a way of sending data to another processor which then updates data in a database. Data pumps are a necessary component within space-based architecture, as processing units do not directly read from and write to a database. Data pumps within a space-based architecture are always asynchronous, providing eventual consistency with the in-memory

cache and the database. When a processing unit instance receives a request and updates its cache, that processing unit becomes the owner of the update and is therefore responsible for sending that update through the data pump so that the database can be updated eventually.

Data pumps are usually implemented using messaging, as shown in Figure 15-7. Messaging is a good choice for data pumps when using a space-based architecture. Not only does messaging support asynchronous communication, but it also supports guaranteed delivery and preserving message order through first-in, first-out (FIFO) queueing. Furthermore, messaging provides a decoupling between the processing unit and the data writer so that if the data writer is not available, uninterrupted processing can still take place within the processing units.

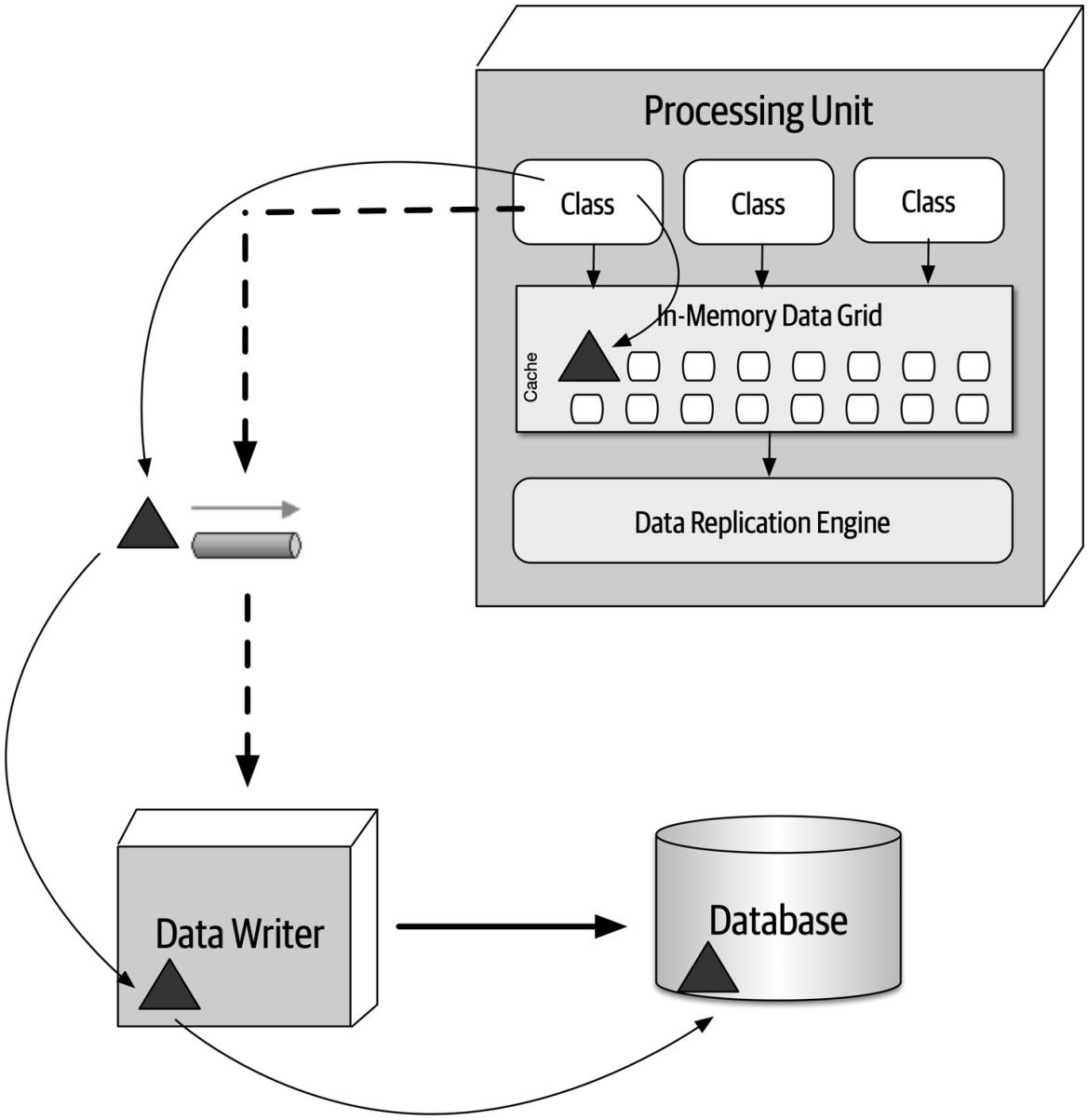

Figure 15-7. Data pump used to send data to a database

In most cases there are multiple data pumps, each one usually dedicated to a particular domain or subdomain (such as customer or inventory). Data pumps can be dedicated to each type of cache (such as `CustomerProfile`, `CustomerWishlist`, and so on), or they can be dedicated to a processing unit domain (such as `Customer`) containing a much larger and general cache.

Data pumps usually have associated contracts, including an action associated with the contract data (add, delete, or update). The contract can

be a JSON schema, XML schema, an object, or even a *value-driven message* (map message containing name-value pairs). For updates, the data contained in the message of the data pump usually only contains the new data values. For example, if a customer changes a phone number on their profile, only the new phone number would be sent, along with the customer ID and an action to update the data.

Data Writers

The data writer component accepts messages from a data pump and updates the database with the information contained in the message of the data pump (see Figure 15-7). Data writers can be implemented as services, applications, or data hubs (such as Ab Initio). The granularity of the data writers can vary based on the scope of the data pumps and processing units.

A domain-based data writer contains all of the necessary database logic to handle all the updates within a particular domain (such as customer), regardless of the number of data pumps it is accepting. Notice in Figure 15-8 that there are four different processing units and four different data pumps representing the customer domain (`Profile`, `WishList`, `Wallet`, and `Preferences`) but only one data writer. The single customer data writer listens to all four data pumps and contains the necessary database logic (such as SQL) to update the customer-related data in the database.

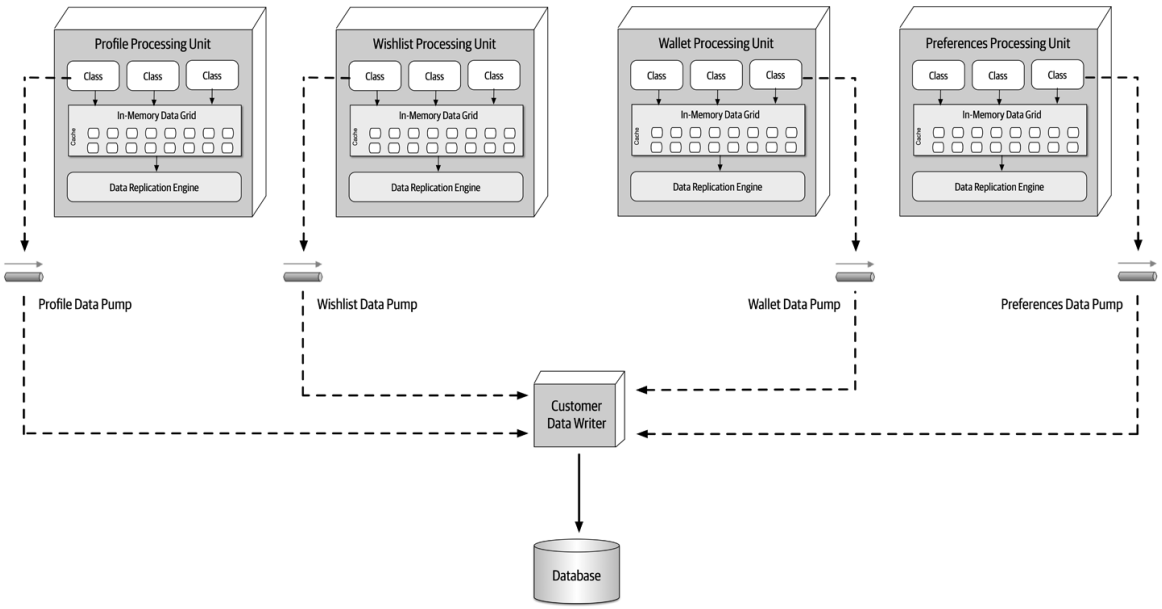

Figure 15-8. Domain-based data writer

Alternatively, each class of processing unit can have its own dedicated data writer component, as illustrated in Figure 15-9. In this model the data writer is dedicated to each corresponding data pump and contains only the database processing logic for that particular processing unit (such as `Wallet`). While this model tends to produce too many data writer components, it does provide better scalability and agility due to the alignment of processing unit, data pump, and data writer.

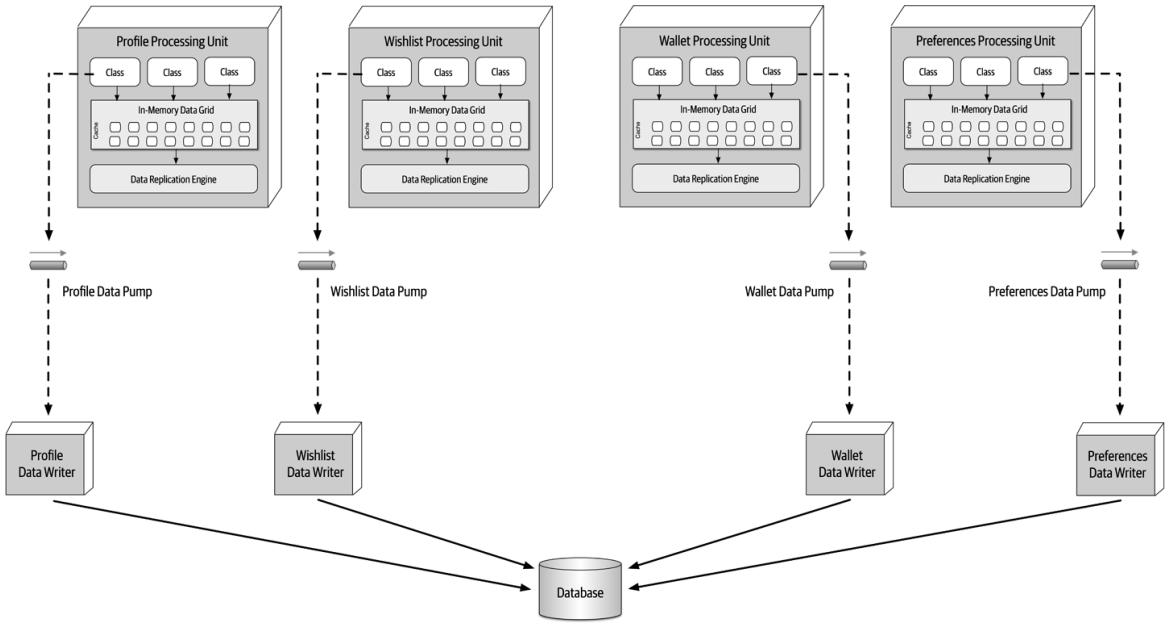

Figure 15-9. Dedicated data writers for each data pump

Data Readers

Whereas data writers take on the responsibility for updating the database, data readers take on the responsibility for reading data from the database and sending it to the processing units via a reverse data pump. In space-based architecture, data readers are only invoked under one of three situations: a crash of all processing unit instances of the same named cache, a redeployment of all processing units within the same named cache, or retrieving archive data not contained in the replicated cache.

In the event where all instances come down (due to a system-wide crash or redeployment of all instances), data must be read from the database (something that is generally avoided in space-based architecture). When instances of a class of processing unit start coming up, each one tries to grab a lock on the cache. The first one to get the lock becomes the temporary cache owner; the others go into a wait state until the lock is released (this might vary based on the type of cache implementation being used, but regardless, there is one primary owner of the cache in this scenario). To load the cache, the instance that gained temporary cache owner status sends a message to a queue requesting data. The data reader component accepts the read request and then performs the necessary

database query logic to retrieve the data needed by the processing unit. As the data reader queries data from the database, it sends that data to a different queue (called a reverse data pump). The temporary cache owner processing unit receives the data from the reverse data pump and loads the cache. Once all the data is loaded, the temporary owner releases the lock on the cache, all other instances are then synchronized, and processing can begin. This processing flow is illustrated in Figure 15-10.

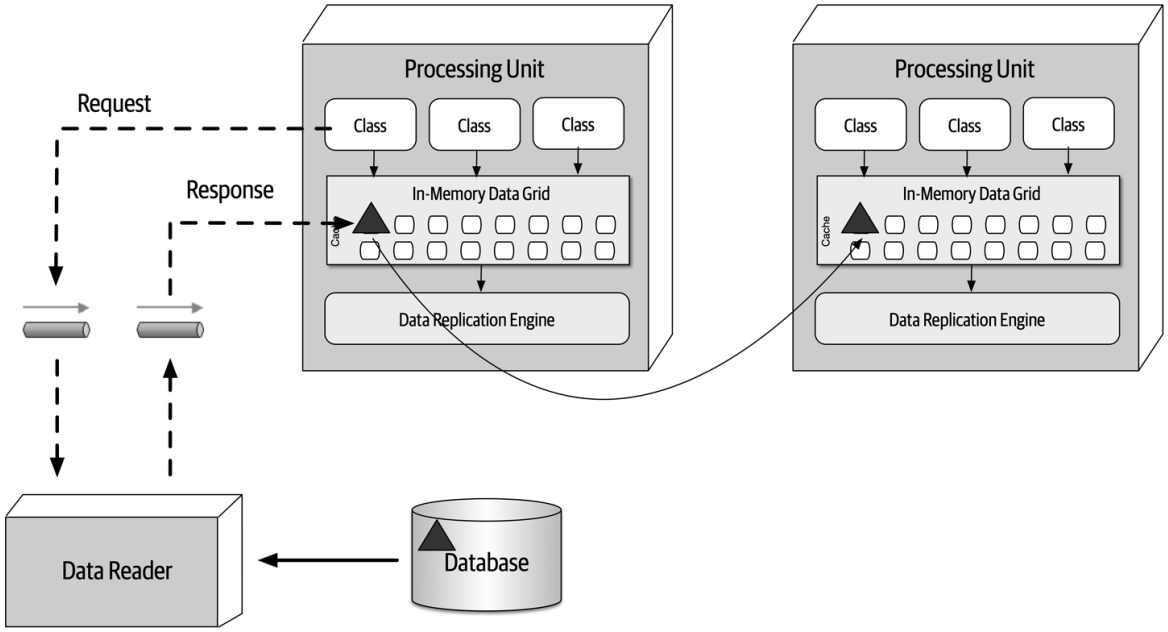

Figure 15-10. Data reader with reverse data pump

Like data writers, data readers can also be domain-based or dedicated to a specific class of processing unit (which is usually the case). The implementation is also the same as the data writers—either service, application, or data hub.

The data writers and data readers essentially form what is usually known as a *data abstraction layer* (or *data access layer* in some cases). The difference between the two is in the amount of detailed knowledge the processing units have with regard to the structure of the tables (or schema) in the database. A data access layer means that the processing units are coupled to the underlying data structures in the database, and only use the data readers and writers to indirectly access the database. A data abstraction layer, on the other hand, means that the processing unit is decoupled from

the underlying database table structures through separate contracts. Space-based architecture generally relies on a data abstraction layer model so that the replicated cache schema in each processing unit can be different than the underlying database table structures. This allows for incremental changes to the database without necessarily impacting the processing units. To facilitate this incremental change, the data writers and data readers contain transformation logic so that if a column type changes or a column or table is dropped, the data readers and data writers can buffer the database change until the necessary changes can be made to the processing unit caches.

Data Collisions

When using replicated caching in an active/active state where updates can occur to any service instance containing the same named cache, there is the possibility of a *data collision* due to replication latency. A data collision occurs when data is updated in one cache instance (cache A), and during replication to another cache instance (cache B), the same data is updated by that cache (cache B). In this scenario, the local update to cache B will be overridden through replication by the old data from cache A, and through replication the same data in cache A will be overridden by the update from cache B.

To illustrate this problem, assume there are two service instances (Service A and Service B) containing a replicated cache of product inventory. The following flow demonstrates the data collision problem:

- The current inventory count for blue widgets is 500 units
- Service A updates the inventory cache for blue widgets to 490 units (10 sold)
- During replication, Service B updates the inventory cache for blue widgets to 495 units (5 sold)

- The Service B cache gets updated to 490 units due to replication from Service A update
- The Service A cache gets updates to 495 units due to replication from Service B update
- Both caches in Service A and B are incorrect and out of sync (inventory should be 485 units)

There are several factors that influence how many data collisions might occur: the number of processing unit instances containing the same cache, the update rate of the cache, the cache size, and finally the replication latency of the caching product. The formula used to determine probabilistically how many potential data collisions might occur based on these factors is as follows:

$$CollisionRate = N * \frac{UR^2}{S} * RL$$

where *N* represents the number of service instances using the same named cache, *UR* represents the update rate in milliseconds (squared), *S* the cache size (in terms of number of rows), and *RL* the replication latency of the caching product.

This formula is useful for determining the percentage of data collisions that will likely occur and hence the feasibility of the use of replicated caching. For example, consider the following values for the factors involved in this calculation:

Update rate (UR):	20 updates/second
Number of instances (N):	5
Cache size (S):	50,000 rows
Replication latency (RL):	100 milliseconds
Updates:	72,000 per hour
Collision rate:	14.4 per hour
Percentage:	0.02%

Applying these factors to the formula yields 72,000 updates and hour, with a high probability that 14 updates to the same data may collide. Given the low percentage (0.02%), replication would be a viable option.

Varying the replication latency has a significant impact on the consistency of data. Replication latency depends on many factors, including the type of network and the physical distance between processing units. For this reason replication latency values are rarely published and must be calculated and derived from actual measurements in a production environment. The value used in the prior example (100 milliseconds) is a good planning number if the actual replication latency, a value we frequently use to determine the number of data collisions, is not available. For example, changing the replication latency from 100 milliseconds to 1 millisecond yields the same number of updates (72,000 per hour) but produces only the probability of 0.1 collisions per hour! This scenario is shown in the following table:

Update rate (UR):	20 updates/second
Number of instances (N):	5
Cache size (S):	50,000 rows
Replication latency (RL):	1 millisecond (changed from 100)
Updates:	72,000 per hour
Collision rate:	0.1 per hour
Percentage:	0.0002%

The number of processing units containing the same named cache (as represented through the *number of instances* factor) also has a direct proportional relationship to the number of data collisions possible. For example, reducing the number of processing units from 5 instances to 2 instances yields a data collision rate of only 6 per hour out of 72,000 updates per hour:

Update rate (UR):	20 updates/second
Number of instances (N):	2 (changed from 5)
Cache size (S):	50,000 rows
Replication latency (RL):	100 milliseconds
Updates:	72,000 per hour
Collision rate:	5.8 per hour
Percentage:	0.008%

The cache size is the only factor that is inversely proportional to the collision rate. As the cache size decreases, collision rates increase. In our example, reducing the cache size from 50,000 rows to 10,000 rows (and

keeping everything the same as in the first example) yields a collision rate of 72 per hour, significantly higher than with 50,000 rows:

Update rate (UR):	20 updates/second
Number of instances (N):	5
Cache size (S):	10,000 rows (changed from 50,000)
Replication latency (RL):	100 milliseconds
Updates:	72,000 per hour
Collision rate:	72.0 per hour
Percentage:	0.1%

Under normal circumstances, most systems do not have consistent update rates over such a long period of time. As such, when using this calculation it is helpful to understand the maximum update rate during peak usage and calculate minimum, normal, and peak collision rates.

Cloud Versus On-Premises Implementations

Space-based architecture offers some unique options when it comes to the environments in which it is deployed. The entire topology, including the processing units, virtualized middleware, data pumps, data readers and writers, and the database, can be deployed within cloud-based environments on-premises ("on-prem"). However, this architecture style can also be deployed between these environments, offering a unique feature not found in other architecture styles.

A powerful feature of this architecture style (as illustrated in Figure 15-11) is to deploy applications via processing units and virtualized middleware in managed cloud-based environments while keeping the physical databases and corresponding data on-prem. This topology supports very effective

cloud-based data synchronization due to the asynchronous data pumps and eventual consistency model of this architecture style. Transactional processing can occur on dynamic and elastic cloud-based environments while preserving physical data management, reporting, and data analytics within secure and local on-prem environments.

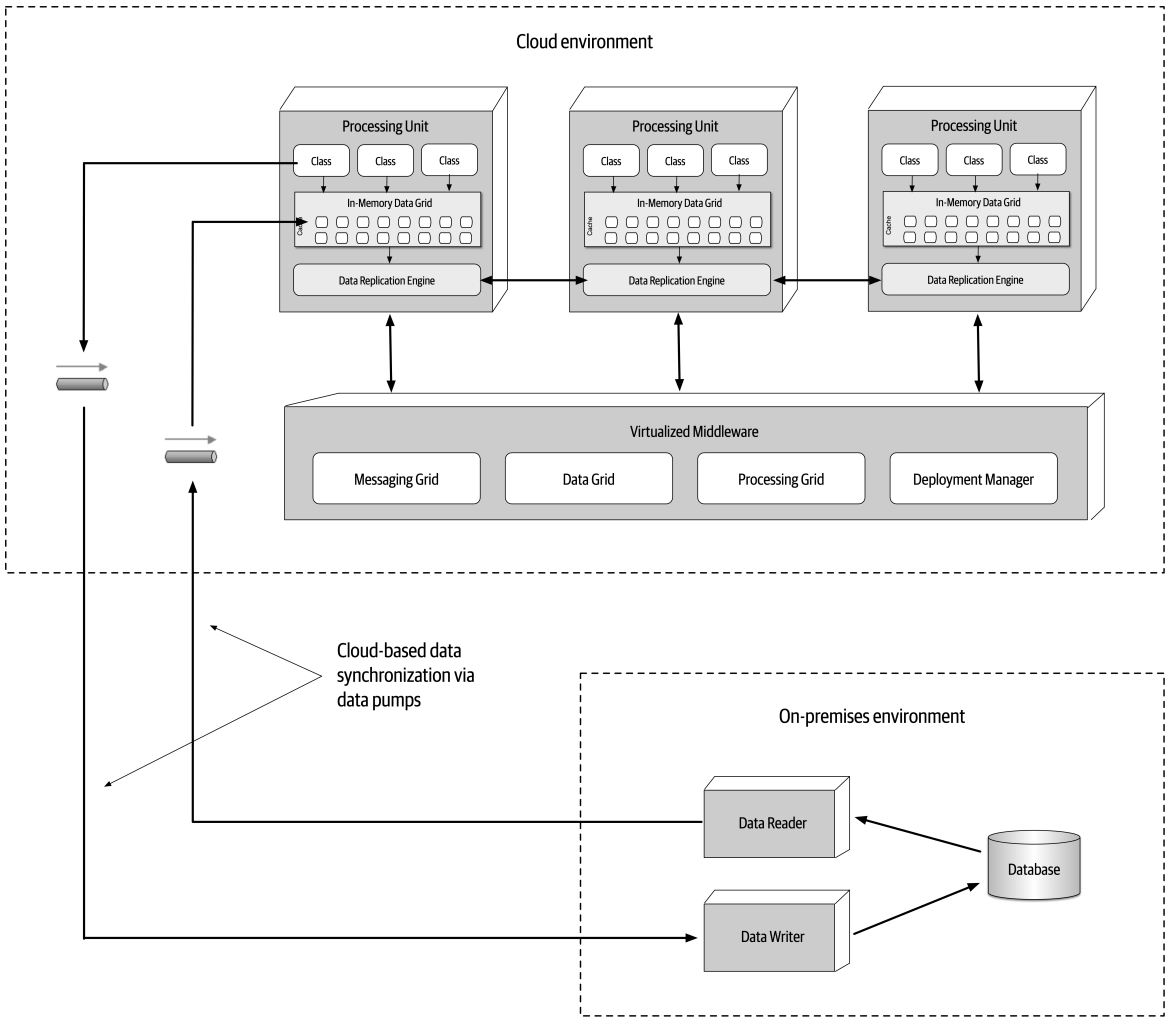

Figure 15-11. Hybrid cloud-based and on-prem topology

Replicated Versus Distributed Caching

Space-based architecture relies on caching for the transactional processing of an application. Removing the need for direct reads and writes to a database is how space-based architecture is able to support high scalability, high elasticity, and high performance. Space-based architecture mostly

relies on replicated caching, although distributed caching can be used as well.

With replicated caching, as illustrated in Figure 15-12, each processing unit contains its own in-memory data grid that is synchronized between all processing units using that same named cache. When an update occurs to a cache within any of the processing units, the other processing units are automatically updated with the new information.

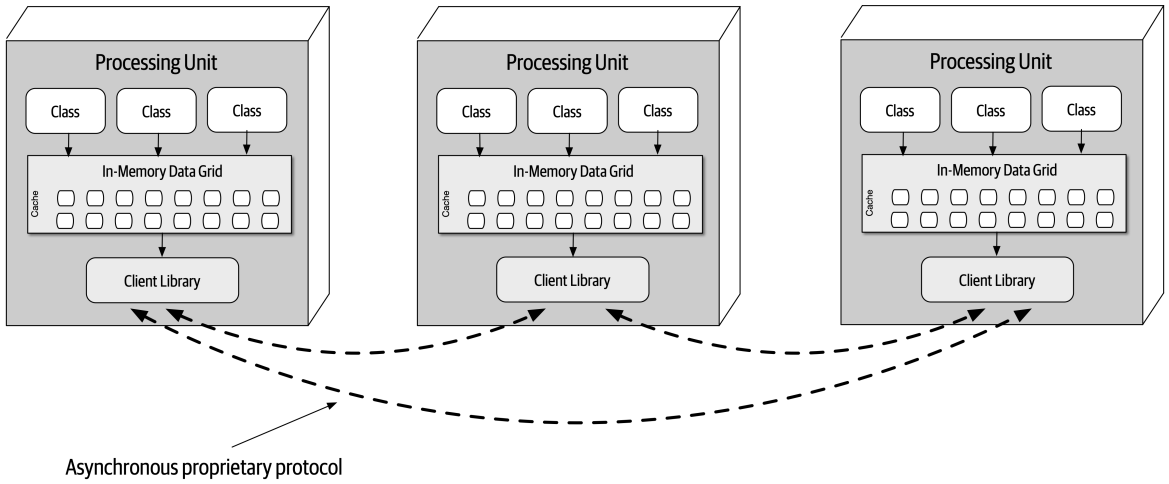

Figure 15-12. Replicated caching between processing units

Replicated caching is not only extremely fast, but it also supports high levels of fault tolerance. Since there is no central server holding the cache, replicated caching does not have a single point of failure. There may be exceptions to this rule, however, based on the implementation of the caching product used. Some caching products require the presence of an external controller to monitor and control the replication of data between processing units, but most product companies are moving away from this model.

While replicated caching is the standard caching model for space-based architecture, there are some cases where it is not possible to use replicated caching. These situations include high data volumes (size of the cache) and high update rates to the cache data. Internal memory caches in excess of 100 MB might start to cause issues with regard to elasticity and high scalability due to the amount of memory used by each processing unit.

Processing units are generally deployed within a virtual machine (or in some cases represent the virtual machine). Each virtual machine only has a certain amount of memory available for internal cache usage, limiting the number of processing unit instances that can be started to process high-throughput situations. Furthermore, as shown in "Data Collisions", if the update rate of the cache data is too high, the data grid might be unable to keep up with that high update rate to ensure data consistency across all processing unit instances. When these situations occur, distributed caching can be used.

Distributed caching, as illustrated in Figure 15-13, requires an external server or service dedicated to holding a centralized cache. In this model the processing units do not store data in internal memory, but rather use a proprietary protocol to access the data from the central cache server. Distributed caching supports high levels of data consistency because the data is all in one place and does not need to be replicated. However, this model has less performance than replicated caching because the cache data must be accessed remotely, adding to the overall latency of the system. Fault tolerance is also an issue with distributed caching. If the cache server containing the data goes down, no data can be accessed or updated from any of the processing units, rendering them nonoperational. Fault tolerance can be mitigated by mirroring the distributed cache, but this could present consistency issues if the primary cache server goes down unexpectedly and the data does not make it to the mirrored cache server.

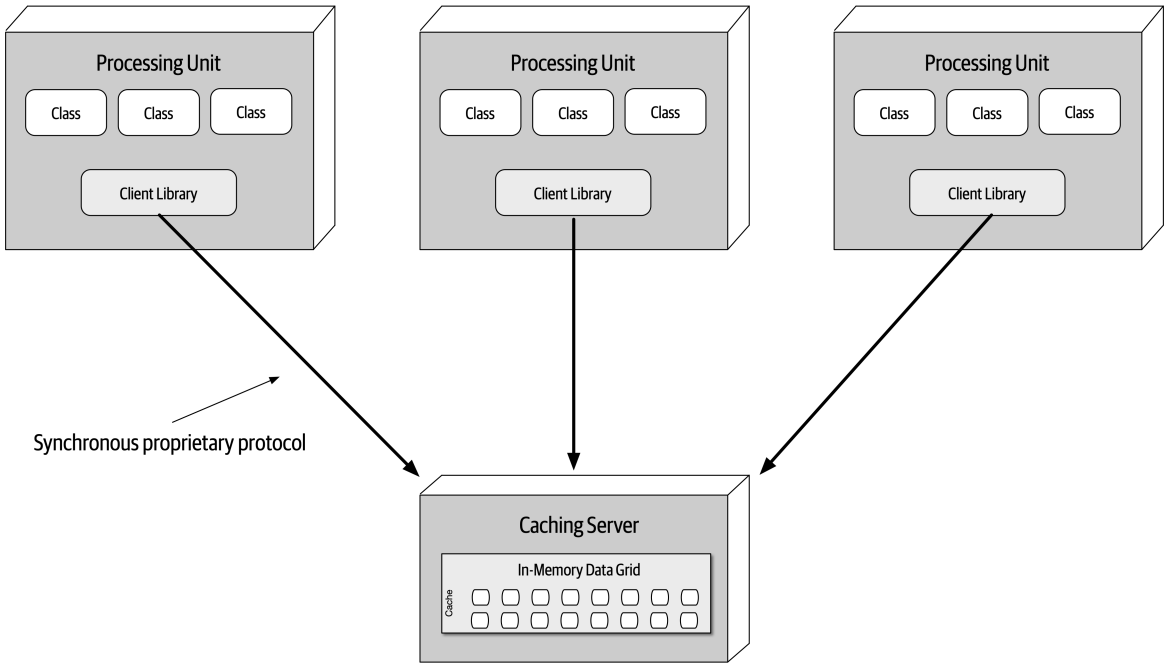

Figure 15-13. Distributed caching between processing units

When the size of the cache is relatively small (under 100 MB) and the update rate of the cache is low enough that the replication engine of the caching product can keep up with the cache updates, the decision between using a replicated cache and a distributed cache becomes one of data consistency versus performance and fault tolerance. A distributed cache will always offer better data consistency over a replicated cache because the cache of data is in a single place (as opposed to being spread across multiple processing units). However, performance and fault tolerance will always be better when using a replicated cache. Many times this decision comes down to the type of data being cached in the processing units. The need for highly consistent data (such as inventory counts of the available products) usually warrants a distributed cache, whereas data that does not change often (such as reference data like name/value pairs, product codes, and product descriptions) usually warrants a replicated cache for quick lookup. Some of the selection criteria that can be used as a guide for choosing when to use a distributed cache versus a replicated cache are listed in Table 15-1.

Table 15-1. Distributed versus replicated caching

Decision criteria	Replicated cache	Distributed cache
Optimization	Performance	Consistency
Cache size	Small (<100 MB)	Large (>500 MB)
Type of data	Relatively static	Highly dynamic
Update frequency	Relatively low	High update rate
Fault tolerance	High	Low

When choosing the type of caching model to use with space-based architecture, remember that in most cases *both* models will be applicable within any given application context. In other words, neither replicated caching nor distributed caching solve every problem. Rather than trying to seek compromises through a single consistent caching model across the application, leverage each for its strengths. For example, for a processing unit that maintains the current inventory, choose a distributed caching model for data consistency; for a processing unit that maintains the customer profile, choose a replicated cache for performance and fault tolerance.

Near-Cache Considerations

A *near-cache* is a type of caching hybrid model bridging in-memory data grids with a distributed cache. In this model (illustrated in Figure 15-14) the distributed cache is referred to as the *full backing cache*, and each in-memory data grid contained within each processing unit is referred to as the *front cache*. The front cache always contains a smaller subset of the full backing cache, and it leverages an *eviction policy* to remove older items so that newer ones can be added. The front cache can be what is known as a most recently used (MRU) cache containing the most recently used items or a most frequently used (MFU) cache containing the most frequently used

items. Alternatively, a *random replacement* eviction policy can be used in the front cache so that items are removed in a random manner when space is needed to add a new item. Random replacement (RR) is a good eviction policy when there is no clear analysis of the data with regard to keeping either the latest used versus the most frequently used.

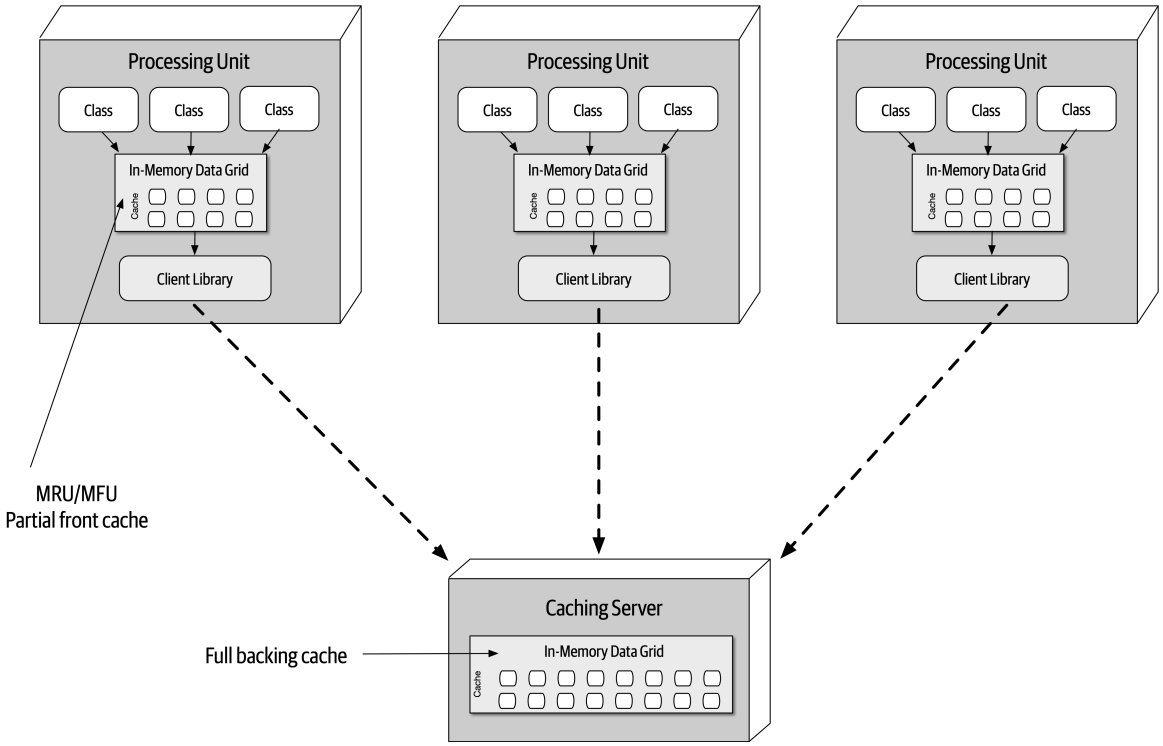

Figure 15-14. Near-cache topology

While the front caches are always kept in sync with the full backing cache, the front caches contained within each processing unit are not synchronized between other processing units sharing the same data. This means that multiple processing units sharing the same data context (such as a customer profile) will likely all have different data in their front cache. This creates inconsistencies in performance and responsiveness between processing units because each processing unit contains different data in the front cache. For this reason we do not recommended using a near-cache model for space-based architecture.

Implementation Examples

Space-based architecture is well suited for applications that experience high spikes in user or request volume and applications that have throughput in excess of 10,000 concurrent users. Examples of space-based architecture include applications like online concert ticketing systems and online auction systems. Both of these examples require high performance, high scalability, and high levels of elasticity.

Concert Ticketing System

Concert ticketing systems have a unique problem domain in that concurrent user volume is relatively low until a popular concert is announced. Once concert tickets go on sale, user volumes usually spike from several hundred concurrent users to several thousand (possibly in the tens of thousands, depending on the concert), all trying to acquire a ticket for the concert (hopefully, good seats!). Tickets usually sell out in a matter of minutes, requiring the kind of architecture characteristics supported by space-based architecture.

There are many challenges associated with this sort of system. First, there are only a certain number of tickets available, regardless of the seating preferences. Seating availability must continually be updated and made available as fast as possible given the high number of concurrent requests. Also, assuming assigned seats are an option, seating availability must also be updated as fast as possible. Continually accessing a central database synchronously for this sort of system would likely not work—it would be very difficult for a typical database to handle tens of thousands of concurrent requests through standard database transactions at this level of scale and update frequency.

Space-based architecture would be a good fit for a concert ticketing system due to the high elasticity requirements required of this type of application. An instantaneous increase in the number of concurrent users wanting to purchase concert tickets would be immediately recognized by the *deployment manager*, which in turn would start up a large number of

processing units to handle the large volume of requests. Optimally, the deployment manager would be configured to start up the necessary number of processing units shortly *before* the tickets went on sale, therefore having those instances on standby right before the significant increase in user load.

Online Auction System

Online auction systems (bidding on items within an auction) share the same sort of characteristics as the online concert ticketing systems described previously—both require high levels of performance and elasticity, and both have unpredictable spikes in user and request load. When an auction starts, there is no way of determining how many people will be joining the auction, and of those people, how many concurrent bids will occur for each asking price.

Space-based architecture is well suited for this type of problem domain in that multiple processing units can be started as the load increases; and as the auction winds down, unused processing units could be destroyed. Individual processing units can be devoted to each auction, ensuring consistency with bidding data. Also, due to the asynchronous nature of the data pumps, bidding data can be sent to other processing (such as bid history, bid analytics, and auditing) without much latency, therefore increasing the overall performance of the bidding process.

Architecture Characteristics Ratings

A one-star rating in the characteristics ratings table in Figure 15-15 means the specific architecture characteristic isn’t well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in Chapter 4.

Architecture characteristic	Star rating
Partitioning type	Domain and technical
Number of quanta	1 to many
Deployability	★★★
Elasticity	★★★★★
Evolutionary	★★★
Fault tolerance	★★★
Modularity	★★★
Overall cost	★★
Performance	★★★★★
Reliability	★★★★
Scalability	★★★★★
Simplicity	★
Testability	★

Figure 15-15. Space-based architecture characteristics ratings

Notice that space-based architecture maximizes elasticity, scalability, and performance (all five-star ratings). These are the driving attributes and main advantages of this architecture style. High levels of all three of these architecture characteristics are achieved by leveraging in-memory data caching and removing the database as a constraint. As a result, processing millions of concurrent users is possible using this architecture style.

While high levels of elasticity, scalability, and performance are advantages in this architecture style, there is a trade-off for this advantage, specifically with regard to overall simplicity and testability. Space-based architecture is a very complicated architecture style due to the use of caching and eventual consistency of the primary data store, which is the ultimate system of record. Care must be taken to ensure no data is lost in the event of a crash in any of the numerous moving parts of this architecture style (see “Preventing Data Loss” in Chapter 14).

Testing gets a one-star rating due to the complexity involved with simulating the high levels of scalability and elasticity supported in this architecture style. Testing hundreds of thousands of concurrent users at peak load is a very complicated and expensive task, and as a result most high-volume testing occurs within production environments with actual extreme load. This produces significant risk for normal operations within a production environment.

Cost is another factor when choosing this architecture style. Space-based architecture is relatively expensive, mostly due to licensing fees for caching products and high resource utilization within cloud and on-prem systems due to high scalability and elasticity.

It is difficult to identify the partitioning type of space-based architecture, and as a result we have identified it as both domain partitioned as well as technically partitioned. Space-based architecture is domain partitioned not only because it aligns itself with a specific type of domain (highly elastic and scalable systems), but also because of the flexibility of the processing units. Processing units can act as domain services in the same way services are defined in a service-based architecture or microservices architecture. At the same time, space-based architecture is technically partitioned in the way it separates the concerns about transactional processing using caching from the actual storage of the data in the database via data pumps. The processing units, data pumps, data readers and writers, and the database all form a technical layering in terms of how requests are processed, very similar with regard to how a monolithic n-tiered layered architecture is structured.

The number of quanta within space-based architecture can vary based on how the user interface is designed and how communication happens between processing units. Because the processing units do not communicate synchronously with the database, the database itself is not part of the quantum equation. As a result, quanta within a space-based architecture are typically delineated through the association between the various user interfaces and the processing units. Processing units that synchronously communicate with each other (or synchronously through the processing grid for orchestration) would all be part of the same architectural quantum.

Chapter 16. Orchestration-Driven Service-Oriented Architecture

Architecture styles, like art movements, must be understood in the context of the era in which they evolved, and this architecture exemplifies this rule more than any other. The combination of external forces that often influence architecture decisions, combined with a logical but ultimately disastrous organizational philosophy, doomed this architecture to irrelevance. However, it provides a great example of how a particular organizational idea can make logical sense yet hinder most important parts of the development process.

History and Philosophy

This style of service-oriented architecture appeared just as companies were becoming enterprises in the late 1990s: merging with smaller companies, growing at a break-neck pace, and requiring more sophisticated IT to accommodate this growth. However, computing resources were scarce, precious, and commercial. Distributed computing had just become possible and necessary, and many companies needed the variable scalability and other beneficial characteristics.

Many external drivers forced architects in this era toward distributed architectures with significant constraints. Before open source operating systems were thought reliable enough for serious work, operating systems were expensive and licensed per machine. Similarly, commercial database servers came with Byzantine licensing schemes, which caused application server vendors (which offered database connection pooling) to battle with database vendors. Thus, architects were expected to reuse as much as

possible. In fact, *reuse* in all forms became the dominant philosophy in this architecture, the side effects of which we cover in "Reuse…and Coupling".

This style of architecture also exemplifies how far architects can push the idea of technical partitioning, which had good motivations but bad consequences.

Topology

The topology of this type of service-oriented architecture is shown in Figure 16-1.

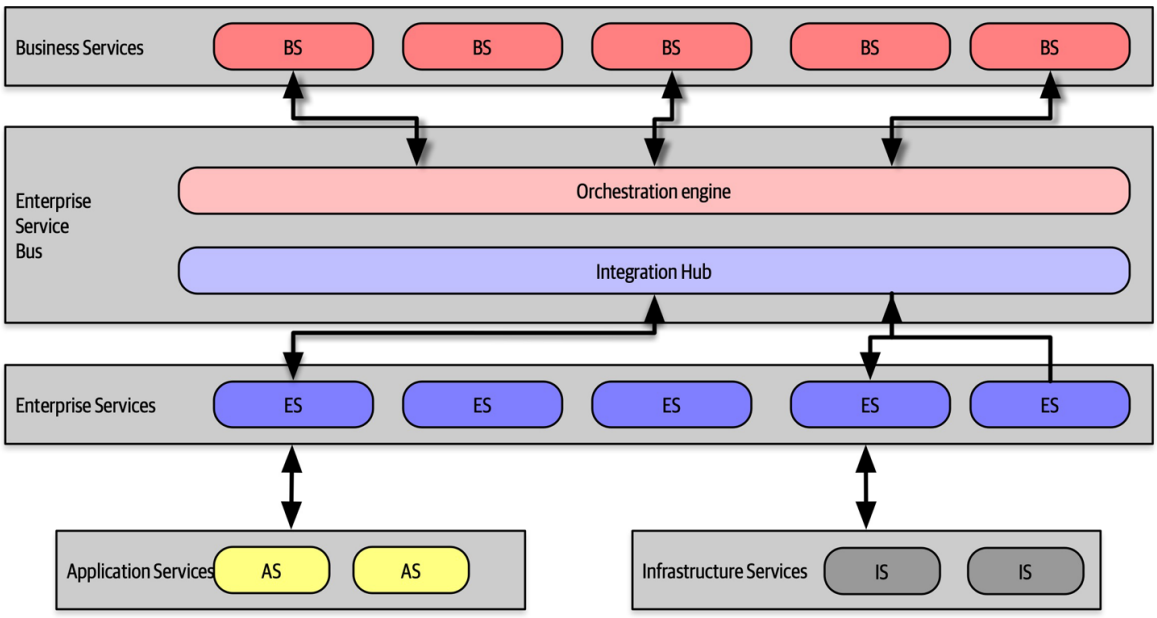

Figure 16-1. Topology of orchestration-driven service-oriented architecture

Not all examples of this style of architecture had the exact layers illustrated in Figure 16-1, but they all followed the same idea of establishing a taxonomy of services within the architecture, each layer with a specific responsibility.

Service-oriented architecture is a distributed architecture; the exact demarcation of boundaries isn't shown in Figure 16-1 because it varied based on organization.

Taxonomy

The architect's driving philosophy in this architecture centered around enterprise-level reuse. Many large companies were annoyed at how much they had to continue to rewrite software, and they struck on a strategy to gradually solve that problem. Each layer of the taxonomy supported this goal.

Business Services

Business services sit at the top of this architecture and provide the entry point. For example, services like `ExecuteTrade` or `PlaceOrder` represent domain behavior. One litmus test common at the time—could an architect answer affirmatively to the question "Are we in the business of…" for each of these services?

These service definitions contained no code—just input, output, and sometimes schema information. They were usually defined by business users, hence the name business services.

Enterprise Services

The *enterprise services* contain fine-grained, shared implementations. Typically, a team of developers is tasked with building atomic behavior around particular business domains: `CreateCustomer`, `CalculateQuote`, and so on. These services are the building blocks that make up the coarse-grained business services, tied together via the orchestration engine.

This separation of responsibility flows from the reuse goal in this architecture. If developers can build fine-grained enterprise services at just the correct level of granularity, the business won't have to rewrite that part of the business workflow again. Gradually, the business will build up a collection of reusable assets in the form of reusable enterprise services.

Unfortunately, the dynamic nature of reality defies these attempts. Business components aren't like construction materials, where solutions last decades.

Markets, technology changes, engineering practices, and a host of other factors confound attempts to impose stability on the software world.

Application Services

Not all services in the architecture require the same level of granularity or reuse as the enterprise services. *Application services* are one-off, single-implementation services. For example, perhaps one application needs geo-location, but the organization doesn't want to take the time or effort to make that a reusable service. An application service, typically owned by a single application team, solves these problems.

Infrastructure Services

Infrastructure services supply the operational concerns, such as monitoring, logging, authentication, and authorization. These services tend to be concrete implementations, owned by a shared infrastructure team that works closely with operations.

Orchestration Engine

The *orchestration engine* forms the heart of this distributed architecture, stitching together the business service implementations using orchestration, including features like transactional coordination and message transformation. This architecture is typically tied to a single relational database, or a few, rather than a database per service as in microservices architectures. Thus, transactional behavior is handled declaratively in the orchestration engine rather than in the database.

The orchestration engine defines the relationship between the business and enterprise services, how they map together, and where transaction boundaries lie. It also acts as an integration hub, allowing architects to integrate custom code with package and legacy software systems.

Because this mechanism forms the heart of the architecture, Conway's law (see "Conway's Law") correctly predicts that the team of integration

architects responsible for this engine become a political force within an organization, and eventually a bureaucratic bottleneck.

While this approach might sound appealing, in practice it was mostly a disaster. Off-loading transaction behavior to an orchestration tool sounded good, but finding the correct level of granularity of transactions became more and more difficult. While building a few services wrapped in a distributed transaction is possible, the architecture becomes increasingly complex as developers must figure out where the appropriate transaction boundaries lie between services.

Message Flow

All requests go through the orchestration engine—it is the location within this architecture where logic resides. Thus, message flow goes through the engine even for internal calls, as shown in Figure 16-2.

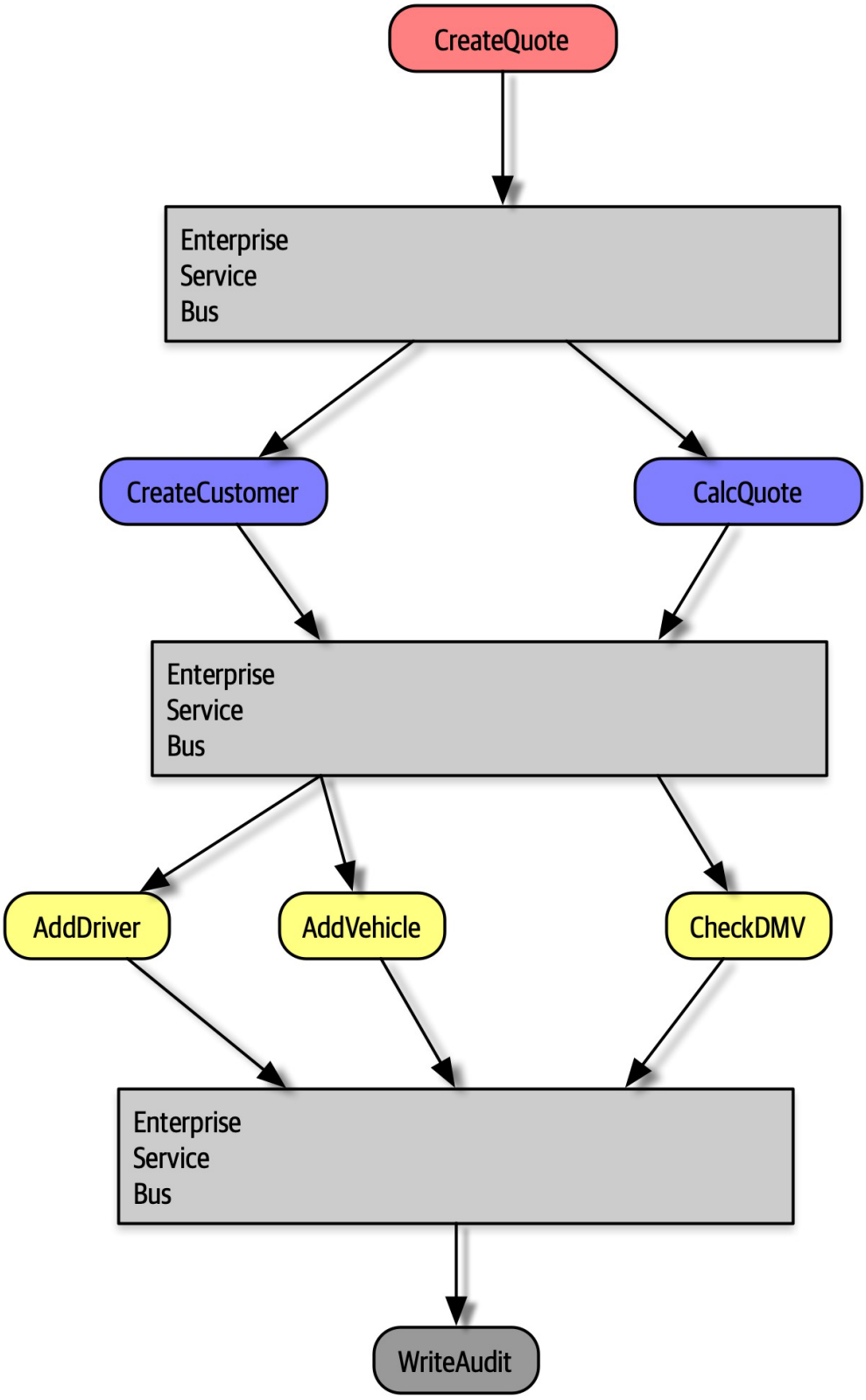

Figure 16-2. Message flow with service-oriented architecture

In Figure 16-2, the `CreateQuote` business-level service calls the service bus, which defines the workflow that consists of calls to `CreateCustomer` and `CalculateQuote`, each of which also has calls to application services. The service bus acts as the intermediary for all calls within this architecture, serving as both an integration hub and orchestration engine.

Reuse…and Coupling

A major goal of this architecture is reuse at the service level—the ability to gradually build business behavior that can be incrementally reused over time. Architects in this architecture were instructed to find reuse opportunities as aggressively as possible. For example, consider the situation illustrated in Figure 16-3.

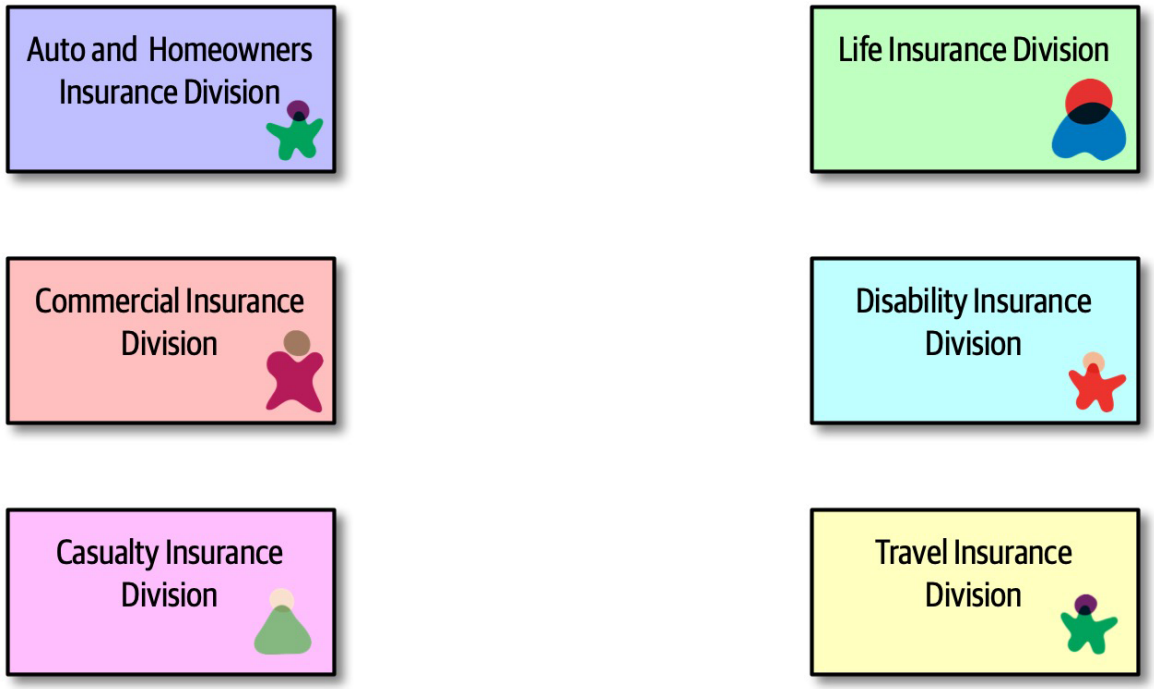

Figure 16-3. Seeking reuse opportunities in service-oriented architecture

In Figure 16-3, an architect realizes that each of these divisions within an insurance company all contain a notion of `Customer`. Therefore, the proper strategy for service-oriented architecture entails extracting the customer

parts into a reusable service and allowing the original services to reference the canonical `Customer` service, shown in Figure 16-4.

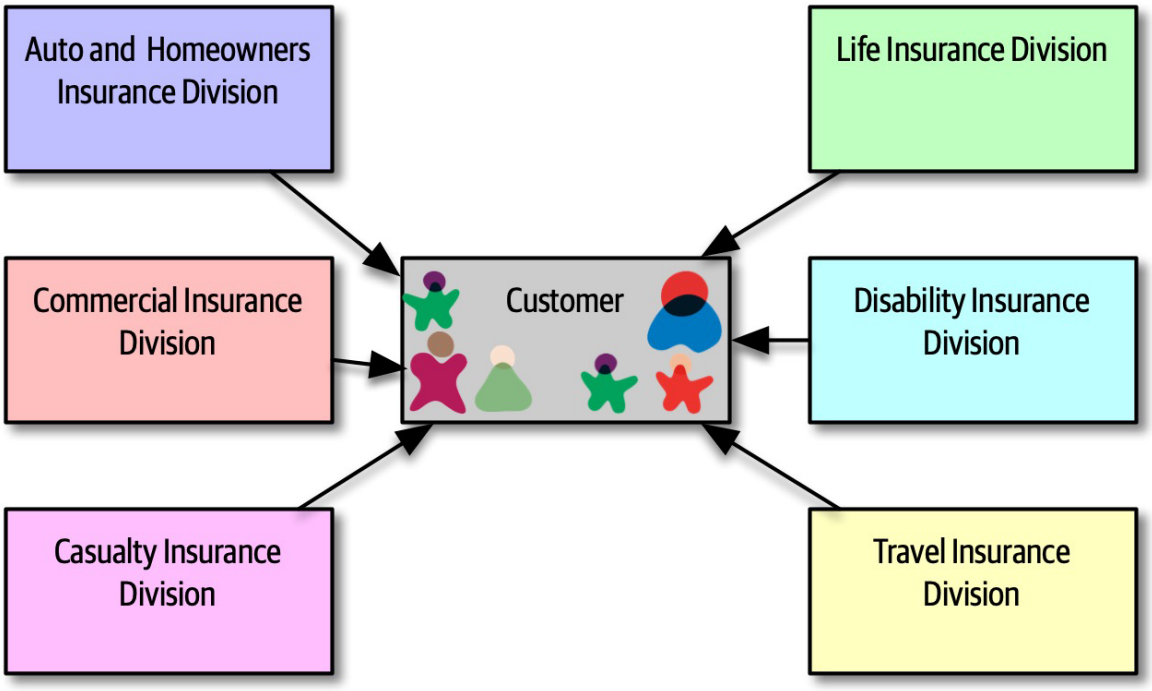

Figure 16-4. Building canonical representations in service-oriented architecture

In Figure 16-4, the architect has isolated all customer behavior into a single `Customer` service, achieving obvious reuse goals.

However, architects only slowly realized the negative trade-offs of this design. First, when a team builds a system primarily around reuse, they also incur a huge amount of coupling between components. For example, in Figure 16-4, a change to the `Customer` service ripples out to all the other services, making change risky. Thus, in service-oriented architecture, architects struggled with making incremental change—each change had a potential huge ripple effect. That in turn led to the need for coordinated deployments, holistic testing, and other drags on engineering efficiency.

Another negative side effect of consolidating behavior into a single place: consider the case of auto and disability insurance in Figure 16-4. To support a single `Customer` service, it must include all the details the organization knows about customers. Auto insurance requires a driver's license, which is a property of the person, not the vehicle. Therefore, the `Customer` service

will have to include details about driver's licenses that the *disability insurance division* cares nothing about. Yet, the team that deals with disability must deal with the extra complexity of a single customer definition.

Perhaps the most damaging revelation from this architecture came with the realization of the impractically of building an architecture so focused on technical partitioning. While it makes sense from a separation and reuse philosophy standpoint, it was a practical nightmare. Domain concepts like `CatalogCheckout` were spread so thinly throughout this architecture that they were virtually ground to dust. Developers commonly work on tasks like "add a new address line to `CatalogCheckout`." In a service-oriented architecture, that could entail dozens of services in several different tiers, plus changes to a single database schema. And, if the current enterprise services aren't defined at the correct transactional granularity, the developers will either have to change their design or build a new, near-identical service to change transactional behavior. So much for reuse.

Architecture Characteristics Ratings

Many of the modern criteria we use to evaluate architecture now were not priorities when this architecture was popular. In fact, the Agile software movement had just started and had not penetrated into the size of organizations likely to use this architecture.

A one-star rating in the characteristics ratings table in Figure 16-5 means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in Chapter 4.

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★★★
Evolutionary	★
Fault tolerance	★★★
Modularity	★★★
Overall cost	★
Performance	★★
Reliability	★★
Scalability	★★★★
Simplicity	★
Testability	★

Figure 16-5. Ratings for service-oriented architecture

Service-oriented architecture is perhaps the most technically partitioned general-purpose architecture ever attempted! In fact, the backlash against the disadvantages of this structure lead to more modern architectures such as microservices. It has a single quantum even though it is a distributed architecture for two reasons. First, it generally uses a single database or just a few databases, creating coupling points within the architecture across many different concerns. Second, and more importantly, the orchestration engine acts as a giant coupling point—no part of the architecture can have

different architecture characteristics than the mediator that orchestrates all behavior. Thus, this architecture manages to find the disadvantages of both monolithic *and* distributed architectures.

Modern engineering goals such as deployability and testability score disastrously in this architecture, both because they were poorly supported and because those were not important (or even aspirational) goals during that era.

This architecture did support some goals such as elasticity and scalability, despite the difficulties in implementing those behaviors, because tool vendors poured enormous effort into making these systems scalable by building session replication across application servers and other techniques. However, being a distributed architecture, performance was never a highlight of this architecture style and was extremely poor because each business request was split across so much of the architecture.

Because of all these factors, simplicity and cost have the inverse relationship most architects would prefer. This architecture was an important milestone because it taught architects how difficult distributed transactions can be in the real world and the practical limits of technical partitioning.

Chapter 17. Microservices Architecture

Microservices is an extremely popular architecture style that has gained significant momentum in recent years. In this chapter, we provide an overview of the important characteristics that set this architecture apart, both topologically and philosophically.

History

Most architecture styles are named after the fact by architects who notice a particular pattern that keeps reappearing—there is no secret group of architects who decide what the next big movement will be. Rather, it turns out that many architects end up making common decisions as the software development ecosystem shifts and changes. The common best ways of dealing with and profiting from those shifts become architecture styles that others emulate.

Microservices differs in this regard—it was named fairly early in its usage and popularized by a famous blog entry by Martin Fowler and James Lewis entitled "Microservices," published in March 2014. They recognized many common characteristics in this relatively new architectural style and delineated them. Their blog post helped define the architecture for curious architects and helped them understand the underlying philosophy.

Microservices is heavily inspired by the ideas in domain-driven design (DDD), a logical design process for software projects. One concept in particular from DDD, *bounded context*, decidedly inspired microservices. The concept of bounded context represents a decoupling style. When a developer defines a domain, that domain includes many entities and behaviors, identified in artifacts such as code and database schemas. For example, an application might have a domain called `CatalogCheckout`,

which includes notions such as catalog items, customers, and payment. In a traditional monolithic architecture, developers would share many of these concepts, building reusable classes and linked databases. Within a bounded context, the internal parts, such as code and data schemas, are coupled together to produce work; but they are never coupled to anything outside the bounded context, such as a database or class definition from another bounded context. This allows each context to define only what it needs rather than accommodating other constituents.

While reuse is beneficial, remember the First Law of Software Architecture regarding trade-offs. The negative trade-off of reuse is coupling. When an architect designs a system that favors reuse, they also favor coupling to achieve that reuse, either by inheritance or composition.

However, if the architect's goal requires high degrees of decoupling, then they favor duplication over reuse. The primary goal of microservices is high decoupling, physically modeling the logical notion of bounded context.

Topology

The topology of microservices is shown in Figure 17-1.

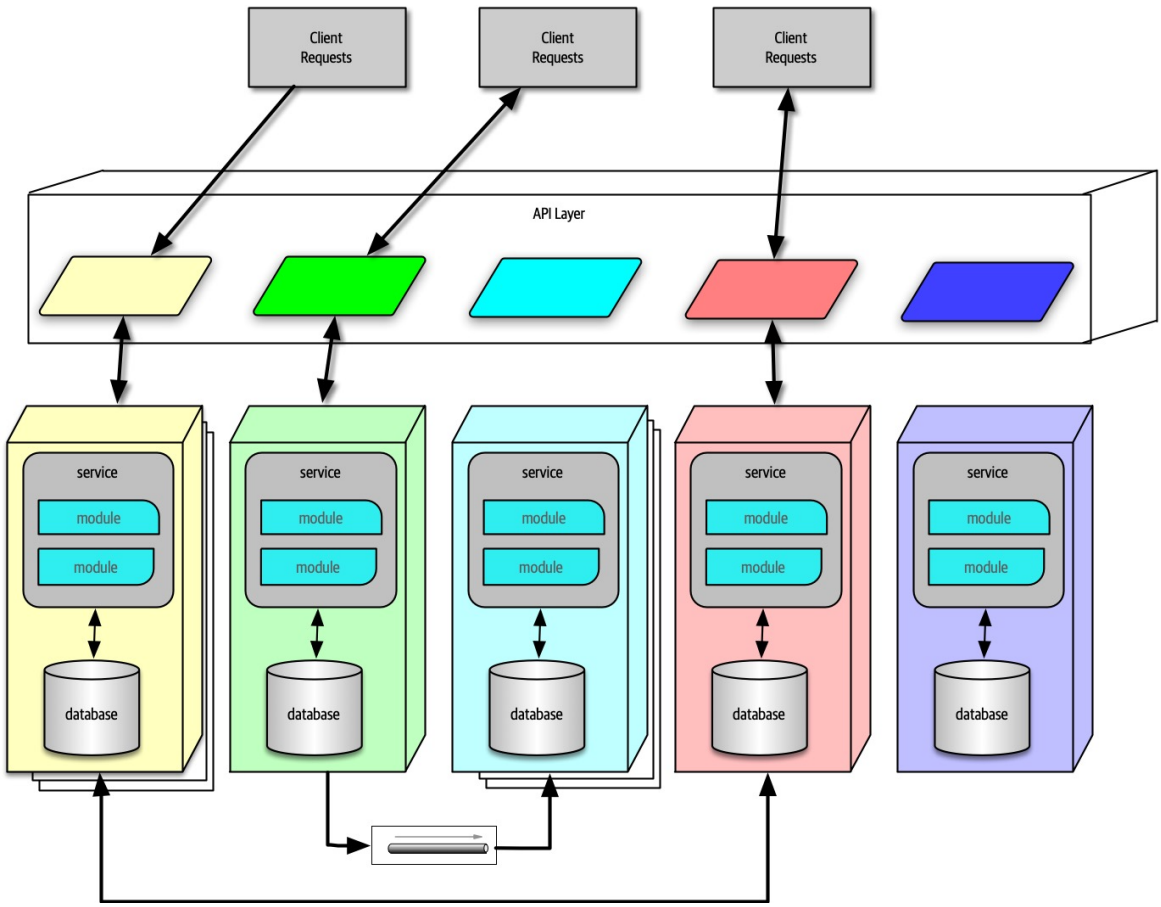

Figure 17-1. The topology of the microservices architecture style

As illustrated in Figure 17-1, due to its single-purpose nature, the service size in microservices is much smaller than other distributed architectures, such as the orchestration-driven service-oriented architecture. Architects expect each service to include all necessary parts to operate independently, including databases and other dependent components. The different characteristics appear in the following sections.

Distributed

Microservices form a *distributed architecture*: each service runs in its own process, which originally implied a physical computer but quickly evolved to virtual machines and containers. Decoupling the services to this degree allows for a simple solution to a common problem in architectures that heavily feature multitenant infrastructure for hosting applications. For

example, when using an application server to manage multiple running applications, it allows operational reuse of network bandwidth, memory, disk space, and a host of other benefits. However, if all the supported applications continue to grow, eventually some resource becomes constrained on the shared infrastructure. Another problem concerns improper isolation between shared applications.

Separating each service into its own process solves all the problems brought on by sharing. Before the evolutionary development of freely available open source operating systems, combined with automated machine provisioning, it was impractical for each domain to have its own infrastructure. Now, however, with cloud resources and container technology, teams can reap the benefits of extreme decoupling, both at the domain and operational level.

Performance is often the negative side effect of the distributed nature of microservices. Network calls take much longer than method calls, and security verification at every endpoint adds additional processing time, requiring architects to think carefully about the implications of granularity when designing the system.

Because microservices is a distributed architecture, experienced architects advise against the use of transactions across service boundaries, making determining the granularity of services the key to success in this architecture.

Bounded Context

The driving philosophy of microservices is the notion of *bounded context*: each service models a domain or workflow. Thus, each service includes everything necessary to operate within the application, including classes, other subcomponents, and database schemas. This philosophy drives many of the decisions architects make within this architecture. For example, in a monolith, it is common for developers to share common classes, such as `Address`, between disparate parts of the application. However,

microservices try to avoid coupling, and thus an architect building this architecture style prefers duplication to coupling.

Microservices take the concept of a domain-partitioned architecture to the extreme. Each service is meant to represent a domain or subdomain; in many ways, microservices is the physical embodiment of the logical concepts in domain-driven design.

Granularity

Architects struggle to find the correct granularity for services in microservices, and often make the mistake of making their services too small, which requires them to build communication links back between the services to do useful work.

> *The term "microservice" is a label, not a description.*
>
> —Martin Fowler

In other words, the originators of the term needed to call this new style *something*, and they chose "microservices" to contrast it with the dominant architecture style at the time, service-oriented architecture, which could have been called "gigantic services". However, many developers take the term "microservices" as a commandment, not a description, and create services that are too fine-grained.

The purpose of service boundaries in microservices is to capture a domain or workflow. In some applications, those natural boundaries might be large for some parts of the system—some business processes are more coupled than others. Here are some guidelines architects can use to help find the appropriate boundaries:

Purpose

The most obvious boundary relies on the inspiration for the architecture style, a domain. Ideally, each microservice should be extremely functionally cohesive, contributing one significant behavior on behalf of the overall application.

Transactions

> Bounded contexts are business workflows, and often the entities that need to cooperate in a transaction show architects a good service boundary. Because transactions cause issues in distributed architectures, if architects can design their system to avoid them, they generate better designs.

Choreography

> If an architect builds a set of services that offer excellent domain isolation yet require extensive communication to function, the architect may consider bundling these services back into a larger service to avoid the communication overhead.

Iteration is the only way to ensure good service design. Architects rarely discover the perfect granularity, data dependencies, and communication styles on their first pass. However, after iterating over the options, an architect has a good chance of refining their design.

Data Isolation

Another requirement of microservices, driven by the bounded context concept, is data isolation. Many other architecture styles use a single database for persistence. However, microservices tries to avoid all kinds of coupling, including shared schemas and databases used as integration points.

Data isolation is another factor an architect must consider when looking at service granularity. Architects must be wary of the entity trap (discussed in “Entity trap”) and not simply model their services to resemble single entities in a database.

Architects are accustomed to using relational databases to unify values within a system, creating a single source of truth, which is no longer an option when distributing data across the architecture. Thus, architects must decide how they want to handle this problem: either identifying one domain

as the source of truth for some fact and coordinating with it to retrieve values or using database replication or caching to distribute information.

While this level of data isolation creates headaches, it also provides opportunities. Now that teams aren't forced to unify around a single database, each service can choose the most appropriate tool, based on price, type of storage, or a host of other factors. Teams have the advantage in a highly decoupled system to change their mind and choose a more suitable database (or other dependency) without affecting other teams, which aren't allowed to couple to implementation details.

API Layer

Most pictures of microservices include an API layer sitting between the consumers of the system (either user interfaces or calls from other systems), but it is optional. It is common because it offers a good location within the architecture to perform useful tasks, either via indirection as a proxy or a tie into operational facilities, such as a naming service (covered in "Operational Reuse").

While an API layer may be used for variety of things, it should not be used as a mediator or orchestration tool if the architect wants to stay true to the underlying philosophy of this architecture: all interesting logic in this architecture should occur inside a bounded context, and putting orchestration or other logic in a mediator violates that rule. This also illustrates the difference between technical and domain partitioning in architecture: architects typically use mediators in technically partitioned architectures, whereas microservices is firmly domain partitioned.

Operational Reuse

Given that microservices prefers duplication to coupling, how do architects handle the parts of architecture that really do benefit from coupling, such as operational concerns like monitoring, logging, and circuit breakers? One of the philosophies in the traditional service-oriented architecture was to reuse

as much functionality as possible, domain and operational alike. In microservices, architects try to split these two concerns.

Once a team has built several microservices, they realize that each has common elements that benefit from similarity. For example, if an organization allows each service team to implement monitoring themselves, how can they ensure that each team does so? And how do they handle concerns like upgrades? Does it become the responsibility of each team to handle upgrading to the new version of the monitoring tool, and how long will that take?

The *sidecar* pattern offers a solution to this problem, illustrated in Figure 17-2.

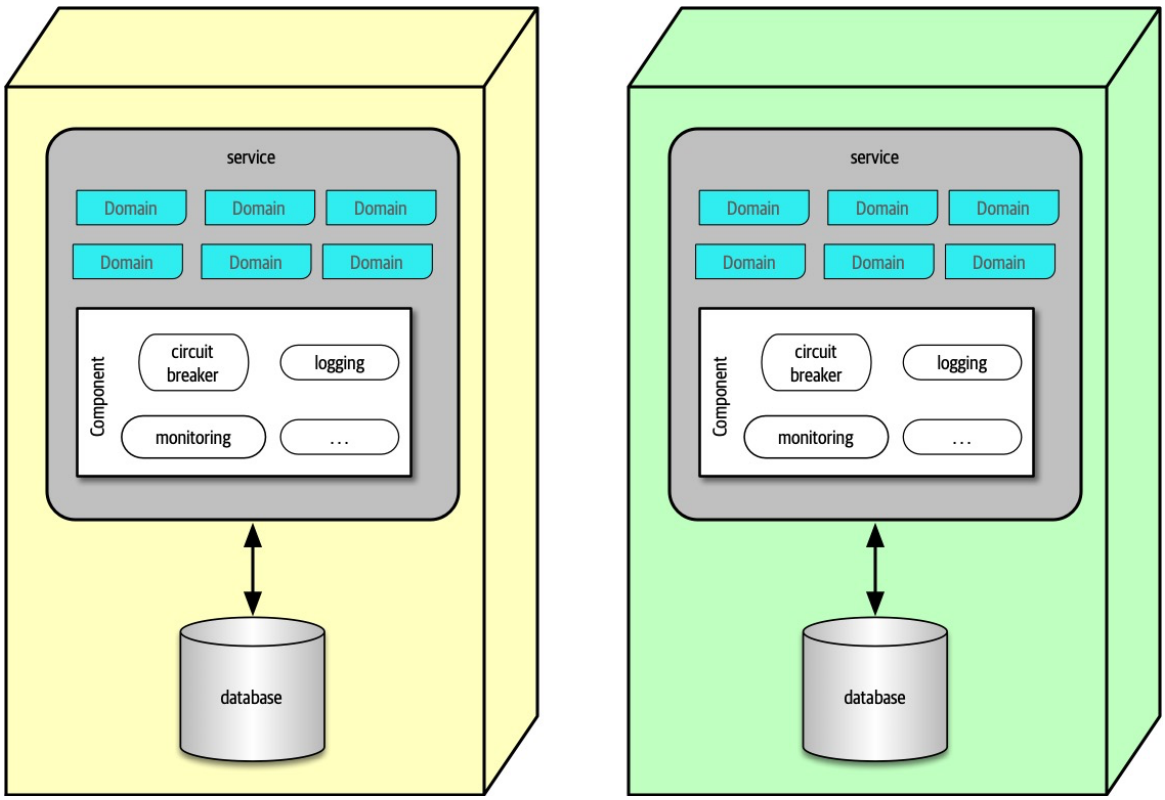

Figure 17-2. The sidecar pattern in microservices

In Figure 17-2, the common operational concerns appear within each service as a separate component, which can be owned by either individual teams or a shared infrastructure team. The sidecar component handles all the operational concerns that teams benefit from coupling together. Thus,

when it comes time to upgrade the monitoring tool, the shared infrastructure team can update the sidecar, and each microservices receives that new functionality.

Once teams know that each service includes a common sidecar, they can build a *service mesh*, allowing unified control across the architecture for concerns like logging and monitoring. The common sidecar components connect to form a consistent operational interface across all microservices, as shown in Figure 17-3.

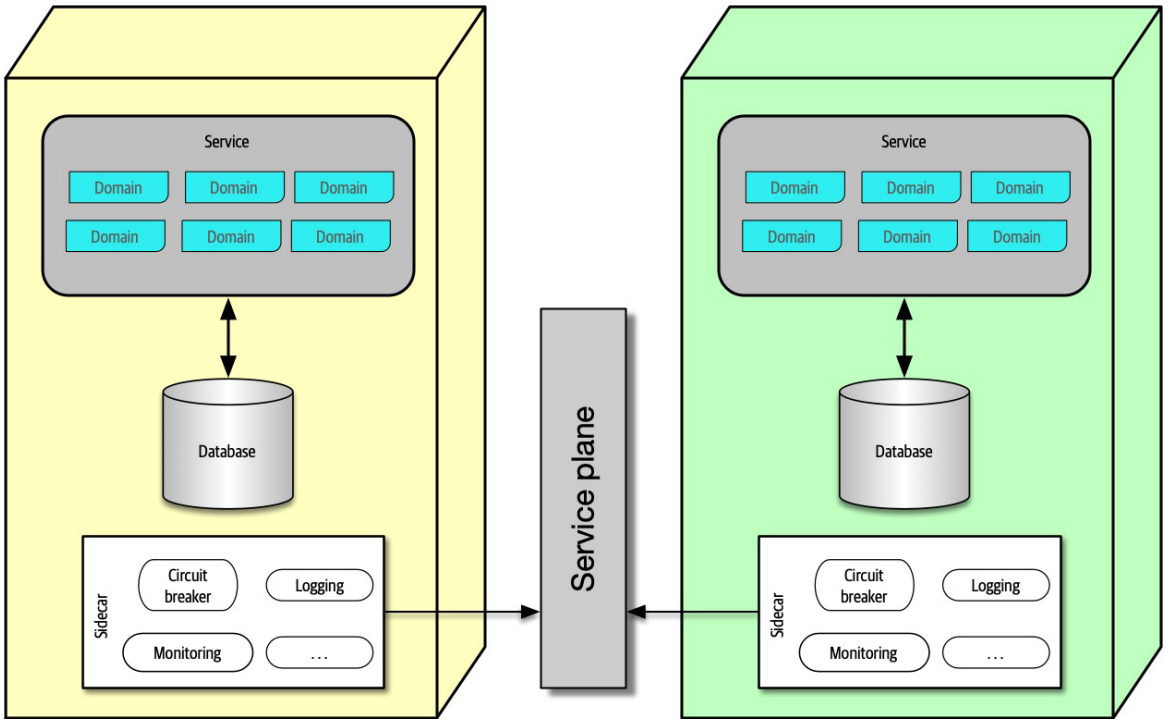

Figure 17-3. The service plane connects the sidecars in a service mesh

In Figure 17-3, each sidecar wires into the service plane, which forms the consistent interface to each service.

The service mesh itself forms a console that allows developers holistic access to services, which is shown in Figure 17-4.

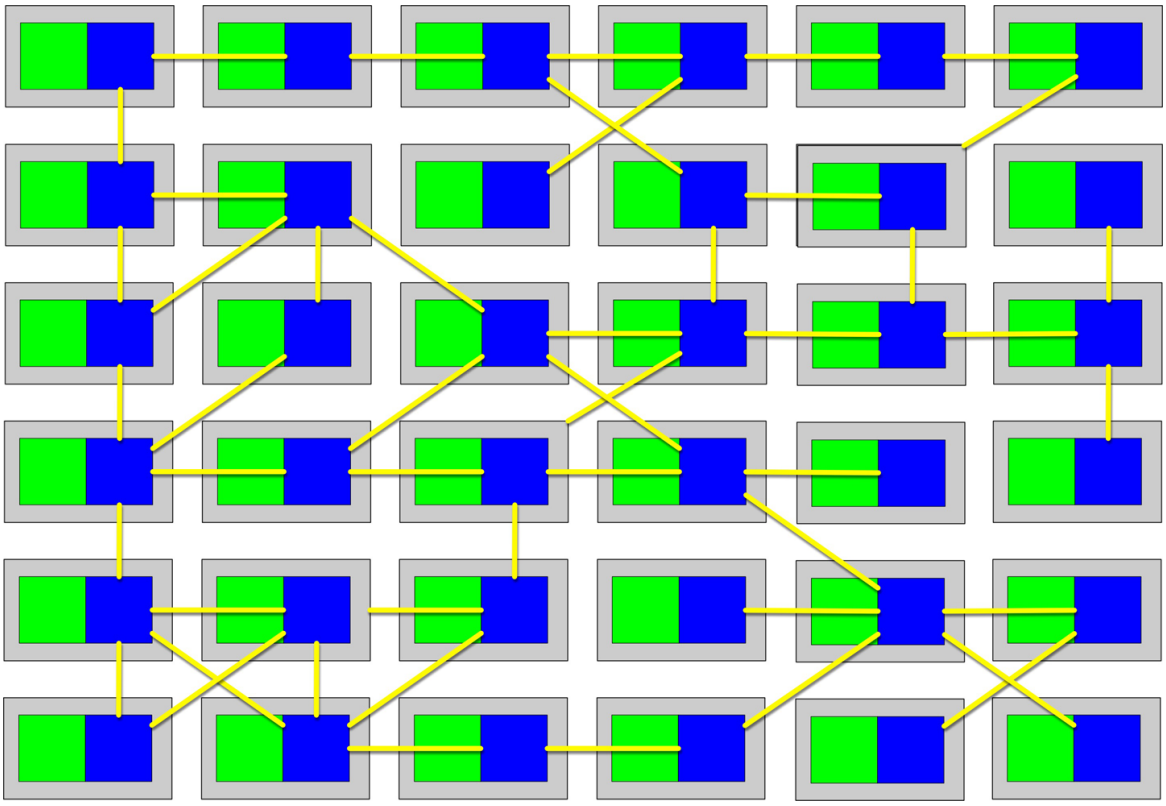

Figure 17-4. The service mesh forms a holistic view of the operational aspect of microservices

Each service forms a node in the overall mesh, as shown in Figure 17-4. The service mesh forms a console that allows teams to globally control operational coupling, such as monitoring levels, logging, and other cross-cutting operational concerns.

Architects use *service discovery* as a way to build elasticity into microservices architectures. Rather than invoke a single service, a request goes through a service discovery tool, which can monitor the number and frequency of requests, as well as spin up new instances of services to handle scale or elasticity concerns. Architects often include service discovery in the service mesh, making it part of every microservice. The API layer is often used to host service discovery, allowing a single place for user interfaces or other calling systems to find and create services in an elastic, consistent way.

Frontends

Microservices favors decoupling, which would ideally encompass the user interfaces as well as backend concerns. In fact, the original vision for microservices included the user interface as part of the bounded context, faithful to the principle in DDD. However, practicalities of the partitioning required by web applications and other external constraints make that goal difficult. Thus, two styles of user interfaces commonly appear for microservices architectures; the first appears in Figure 17-5.

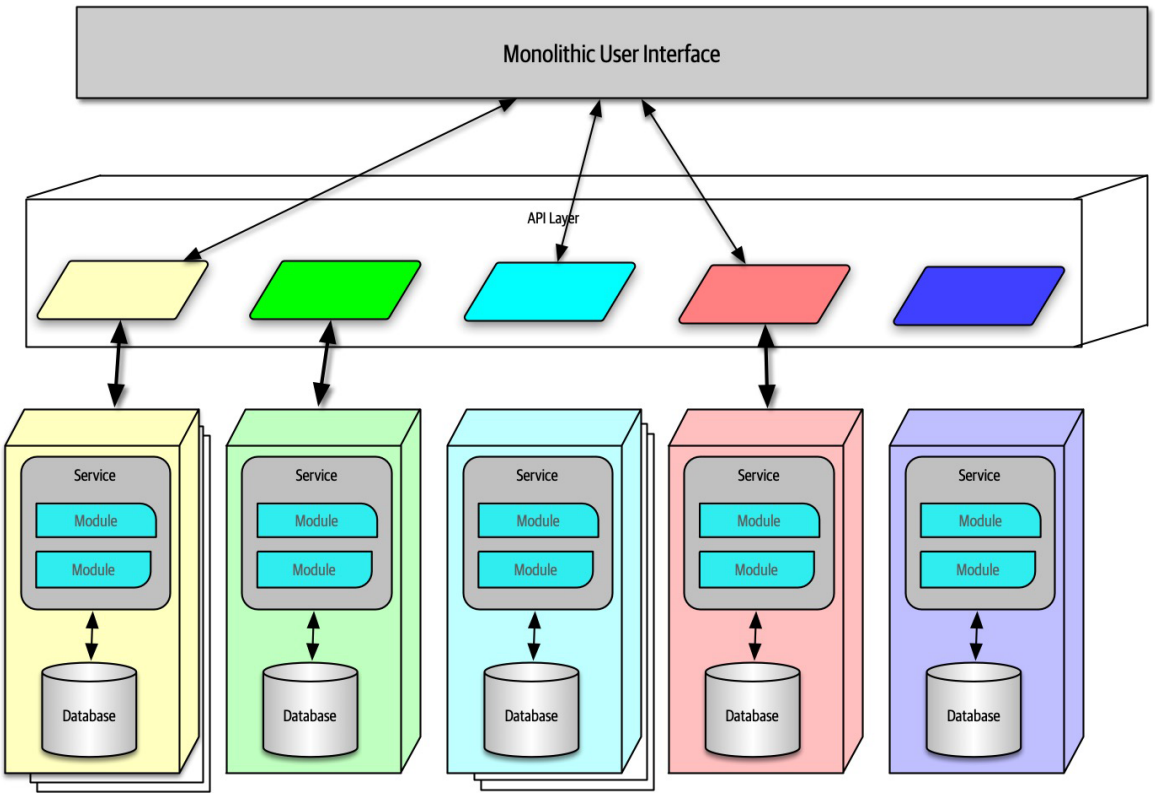

Figure 17-5. Microservices architecture with a monolithic user interface

In Figure 17-5, the monolithic frontend features a single user interface that calls through the API layer to satisfy user requests. The frontend could be a rich desktop, mobile, or web application. For example, many web applications now use a JavaScript web framework to build a single user interface.

The second option for user interfaces uses *microfrontends*, shown in Figure 17-6.

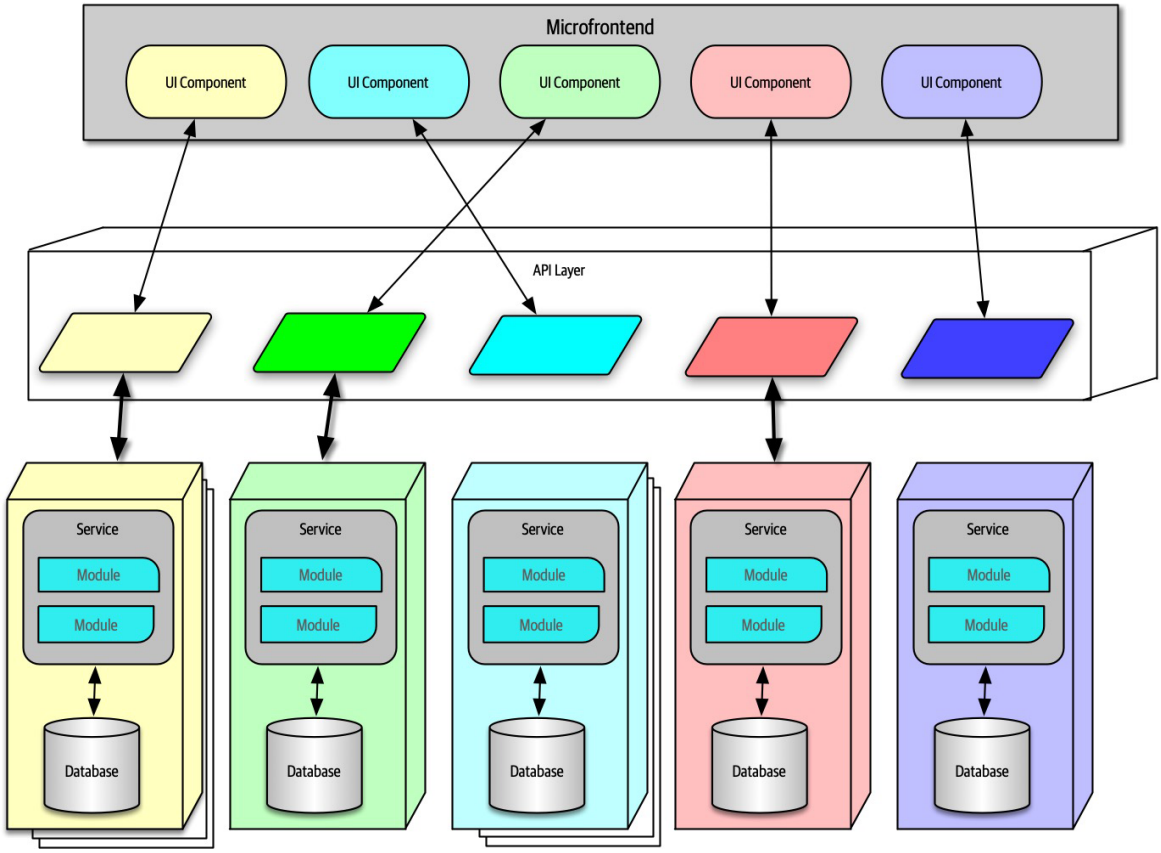

Figure 17-6. Microfrontend pattern in microservices

In Figure 17-6, this approach utilizes components at the user interface level to create a synchronous level of granularity and isolation in the user interface as the backend services. Each service emits the user interface for that service, which the frontend coordinates with the other emitted user interface components. Using this pattern, teams can isolate service boundaries from the user interface to the backend services, unifying the entire domain within a single team.

Developers can implement the microfrontend pattern in a variety of ways, either using a component-based web framework such as React or using one of several open source frameworks that support this pattern.

Communication

In microservices, architects and developers struggle with appropriate granularity, which affects both data isolation and communication. Finding

the correct communication style helps teams keep services decoupled yet still coordinated in useful ways.

Fundamentally, architects must decide on *synchronous* or *asynchronous* communication. Synchronous communication requires the caller to wait for a response from the callee. Microservices architectures typically utilize *protocol-aware heterogeneous interoperability*. We'll break down that term for you:

Protocol-aware

> Because microservices usually don't include a centralized integration hub to avoid operational coupling, each service should know how to call other services. Thus, architects commonly standardize on how particular services call each other: a certain level of REST, message queues, and so on. That means that services must know (or discover) which protocol to use to call other services.

Heterogeneous

> Because microservices is a distributed architecture, each service may be written in a different technology stack. *Heterogeneous* suggests that microservices fully supports polyglot environments, where different services use different platforms.

Interoperability

> Describes services calling one another. While architects in microservices try to discourage transactional method calls, services commonly call other services via the network to collaborate and send/receive information.

ENFORCED HETEROGENEITY

A well-known architect who was a pioneer in the microservices style was the chief architecture at a personal information manager startup for mobile devices. Because they had a fast-moving problem domain, the architect wanted to ensure that none of the development teams accidentally created coupling points between each other, hindering the teams' ability to move independently. It turned out that this architect had a wide mix of technical skills on the teams, thus mandating that each development team use a different technology stack. If one team was using Java and the other was using .NET, it was impossible to accidentally share classes!

This approach is the polar opposite of most enterprise governance policies, which insist on standardizing on a single technology stack. The goal in the microservices world isn't to create the most complex ecosystem possible, but rather to choose the correct scale technology for the narrow scope of the problem. Not every service needs an industrial-strength relational database, and forcing it on small teams slows them rather than benefitting them. This concept leverages the highly decoupled nature of microservices.

For asynchronous communication, architects often use events and messages, thus internally utilizing an event-driven architecture, covered in Chapter 14; the broker and mediator patterns manifest in microservices as *choreography* and *orchestration*.

Choreography and Orchestration

Choreography utilizes the same communication style as a broker event-driven architecture. In other words, no central coordinator exists in this architecture, respecting the bounded context philosophy. Thus, architects find it natural to implement decoupled events between services.

Domain/architecture isomorphism is one key characteristic that architects should look for when assessing how appropriate an architecture style is for a particular problem. This term describes how the shape of an architecture maps to a particular architecture style. For example, in Figure 8-7, the Silicon Sandwiches' technically partitioned architecture structurally supports customizability, and the microkernal architecture style offers the same general structure. Therefore, problems that require a high degree of customization become easier to implement in a microkernel.

Similarly, because the architect's goal in a microservices architecture favors decoupling, the shape of microservices resembles the broker EDA, making these two patterns symbiotic.

In choreography, each service calls other services as needed, without a central mediator. For example, consider the scenario shown in Figure 17-7.

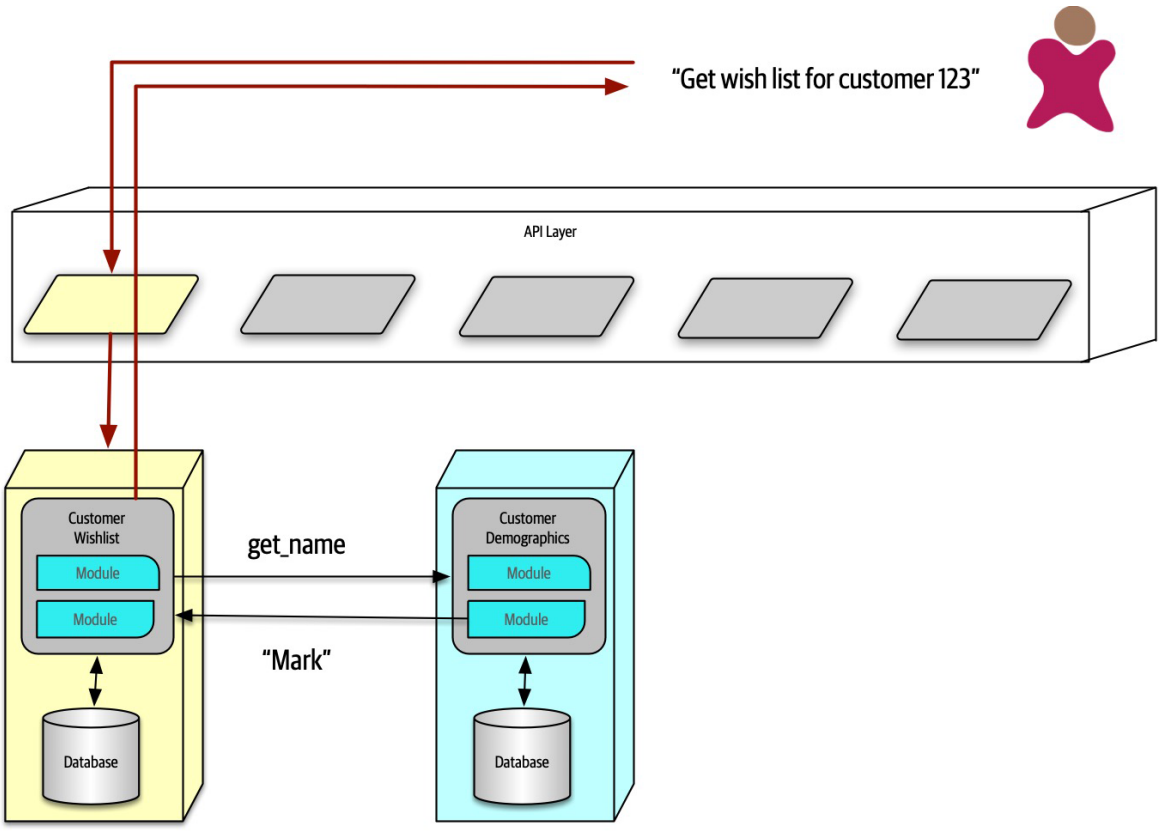

Figure 17-7. Using choreography in microservices to manage coordination

In Figure 17-7, the user requests details about a user's wish list. Because the `CustomerWishList` service doesn't contain all the necessary information, it makes a call to `CustomerDemographics` to retrieve the missing information, returning the result to the user.

Because microservices architectures don't include a global mediator like other service-oriented architectures, if an architect needs to coordinate across several services, they can create their own localized mediator, as shown in Figure 17-8.

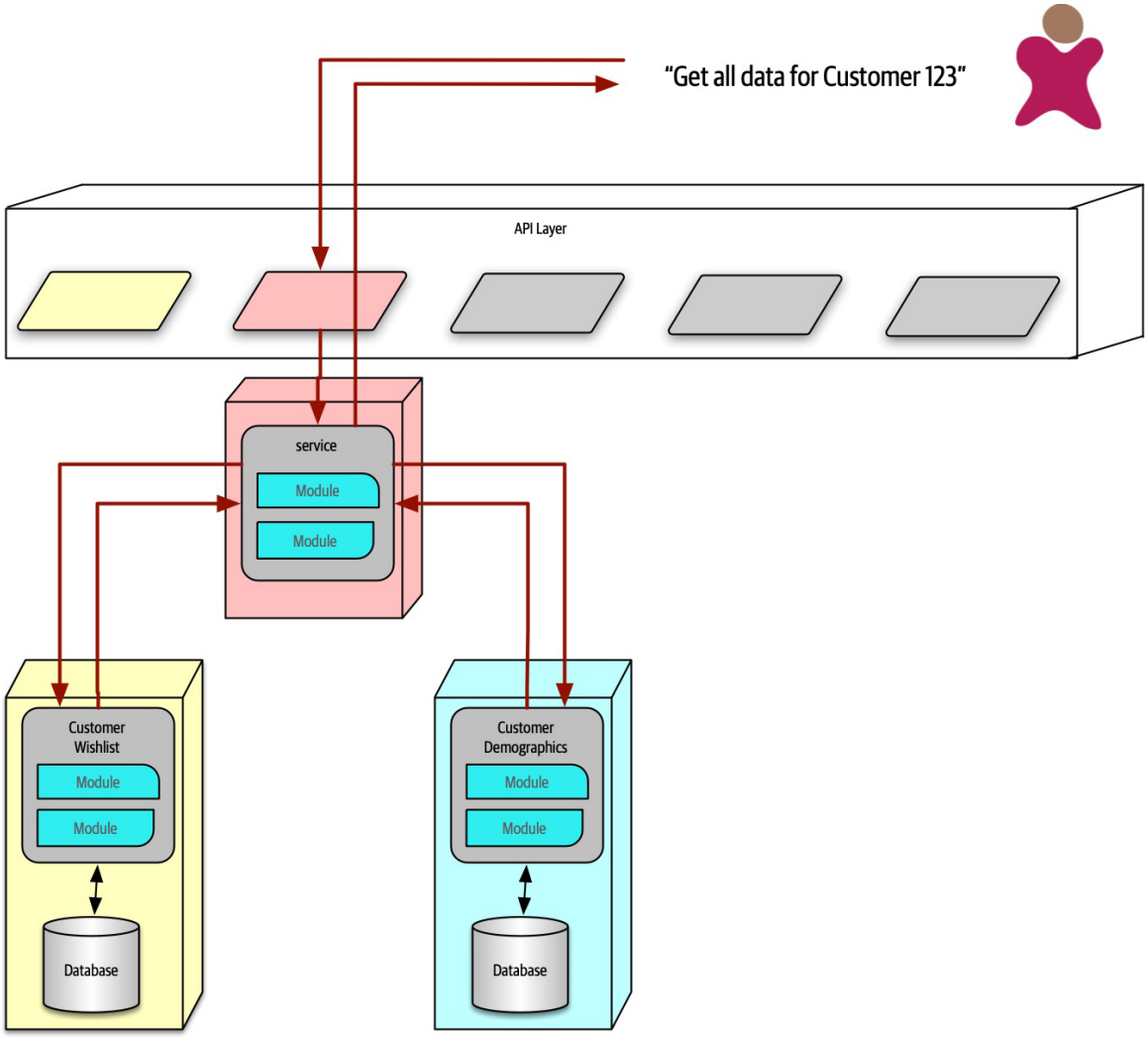

Figure 17-8. Using orchestration in microservices

In Figure 17-8, the developers create a service whose sole responsibility is coordinating the call to get all information for a particular customer. The

user calls the `ReportCustomerInformation` mediator, which calls the necessary other services.

The First Law of Software Architecture suggests that neither of these solutions is perfect—each has trade-offs. In choreography, the architect preserves the highly decoupled philosophy of the architecture style, thus reaping maximum benefits touted by the style. However, common problems like error handling and coordination become more complex in choreographed environments.

Consider an example with a more complex workflow, shown in Figure 17-9.

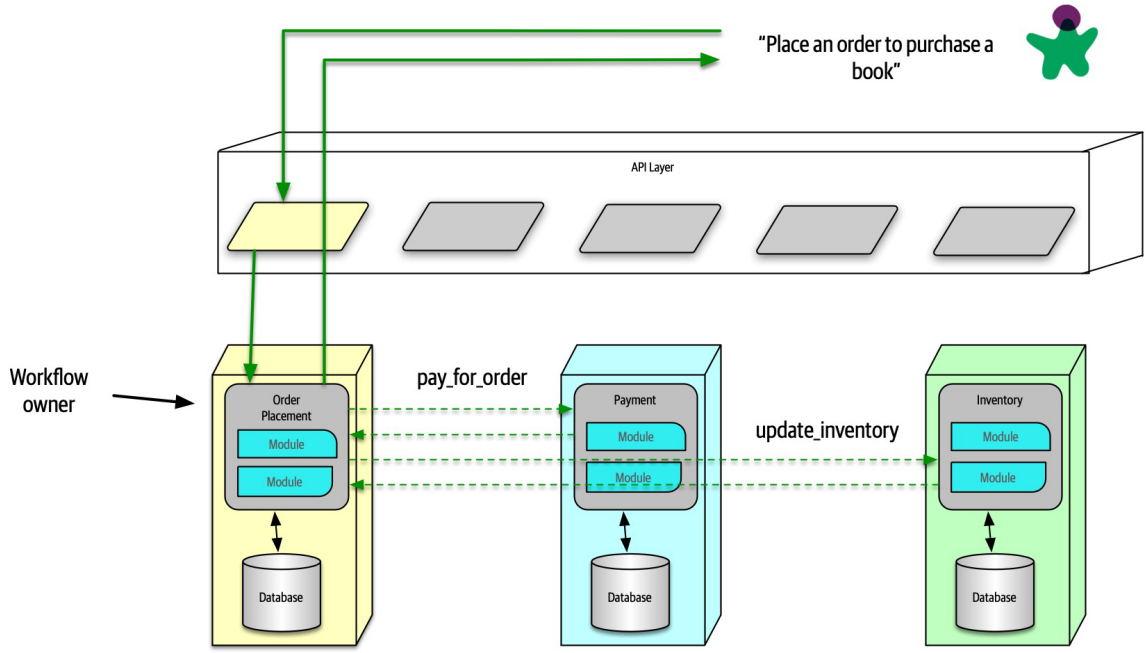

Figure 17-9. Using choreography for a complex business process

In Figure 17-9, the first service called must coordinate across a wide variety of other services, basically acting as a mediator in addition to its other domain responsibilities. This pattern is called the *front controller* pattern, where a nominally choreographed service becomes a more complex mediator for some problem. The downside to this pattern is added complexity in the service.

Alternatively, an architect may choose to use orchestration for complex business processes, illustrated in Figure 17-10.

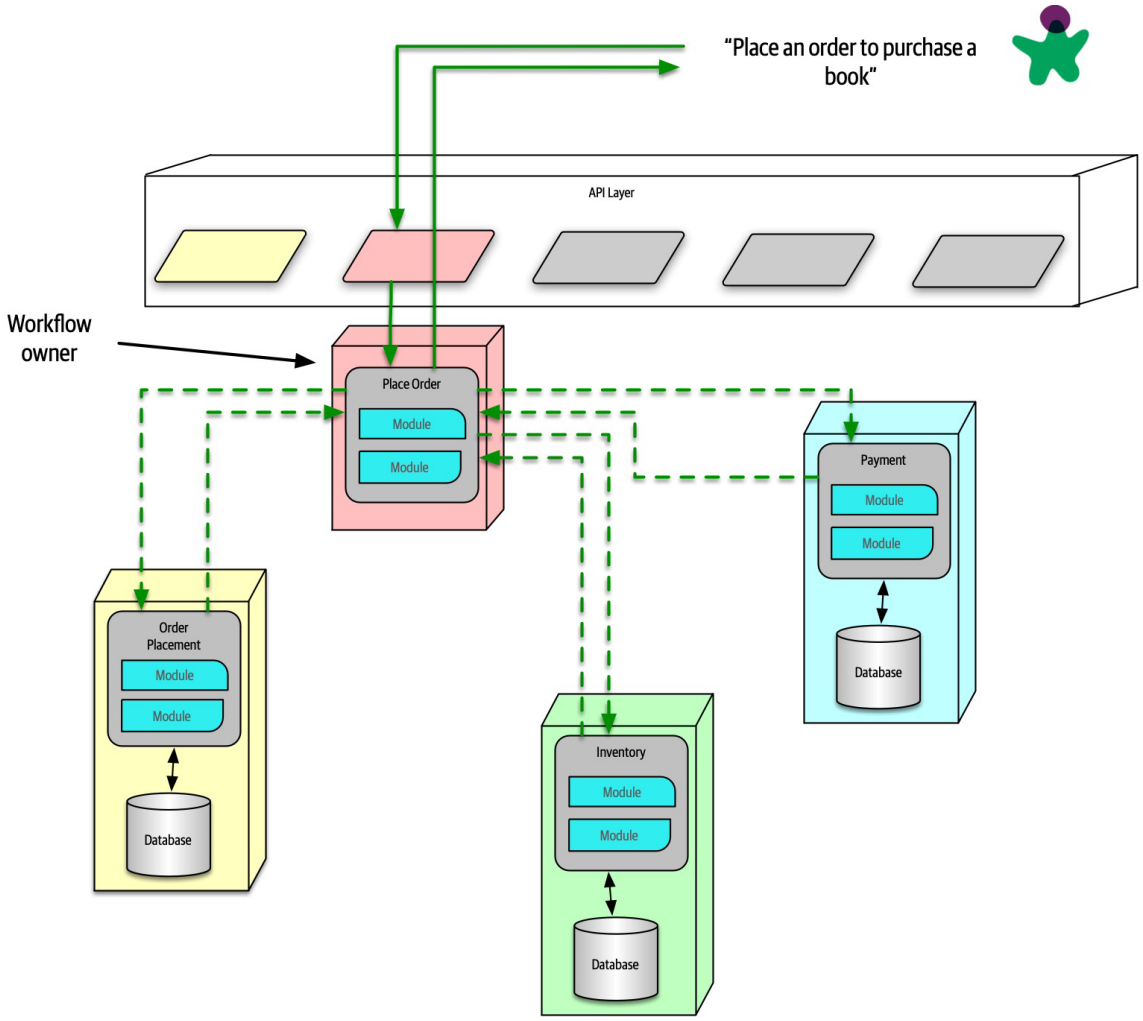

Figure 17-10. Using orchestration for a complex business process

In Figure 17-10, the architect builds a mediator to handle the complexity and coordination required for the business workflow. While this creates coupling between these services, it allows the architect to focus coordination into a single service, leaving the others less affected. Often, domain workflows are inherently coupled—the architect's job entails finding the best way to represent that coupling in ways that support both the domain and architectural goals.

Transactions and Sagas

Architects aspire to extreme decoupling in microservices, but then often encounter the problem of how to do transactional coordination across

services. Because the decoupling in the architecture encourages the same level for the databases, atomicity that was trivial in monolithic applications becomes a problem in distributed ones.

Building transactions across service boundaries violates the core decoupling principle of the microservices architecture (and also creates the worst kind of dynamic connascence, connascence of value). The best advice for architects who want to do transactions across services is: *don't!* Fix the granularity components instead. Often, architects who build microservices architectures who then find a need to wire them together with transactions have gone too granular in their design. Transaction boundaries is one of the common indicators of service granularity.

> **TIP**
>
> Don't do transactions in microservices—fix granularity instead!

Exceptions always exist. For example, a situation may arise where two different services need vastly different architecture characteristics, requiring distinct service boundaries, yet still need transactional coordination. In those situations, patterns exist to handle transaction orchestration, with serious trade-offs.

A popular distributed transactional pattern in microservices is the *saga* pattern, illustrated in Figure 17-11.

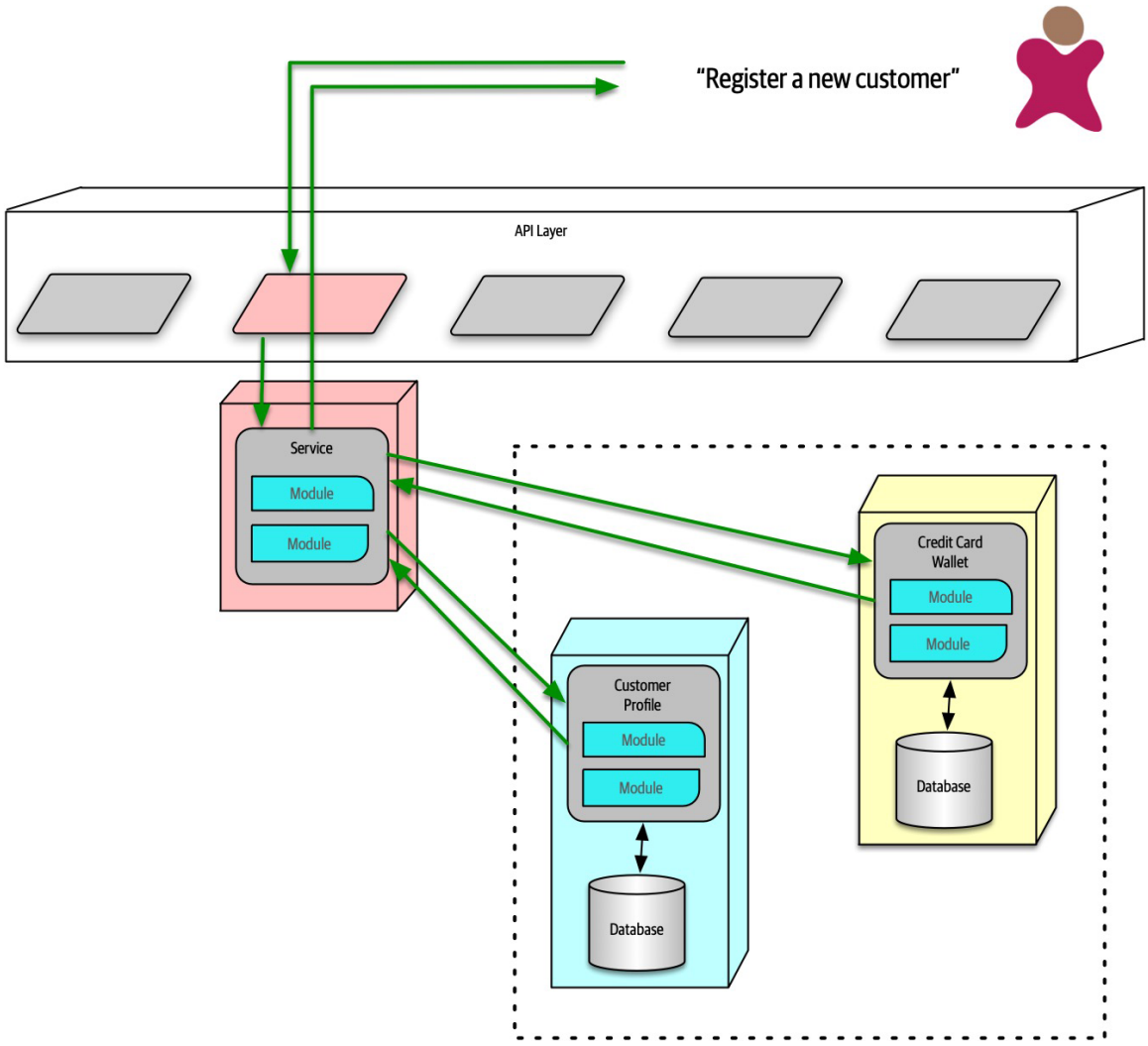

Figure 17-11. The saga pattern in microservices architecture

In Figure 17-11, a service acts a mediator across multiple service calls and coordinates the transaction. The mediator calls each part of the transaction, records success or failure, and coordinates results. If everything goes as planned, all the values in the services and their contained databases update synchronously.

In an error condition, the mediator must ensure that no part of the transaction succeeds if one part fails. Consider the situation shown in Figure 17-12.

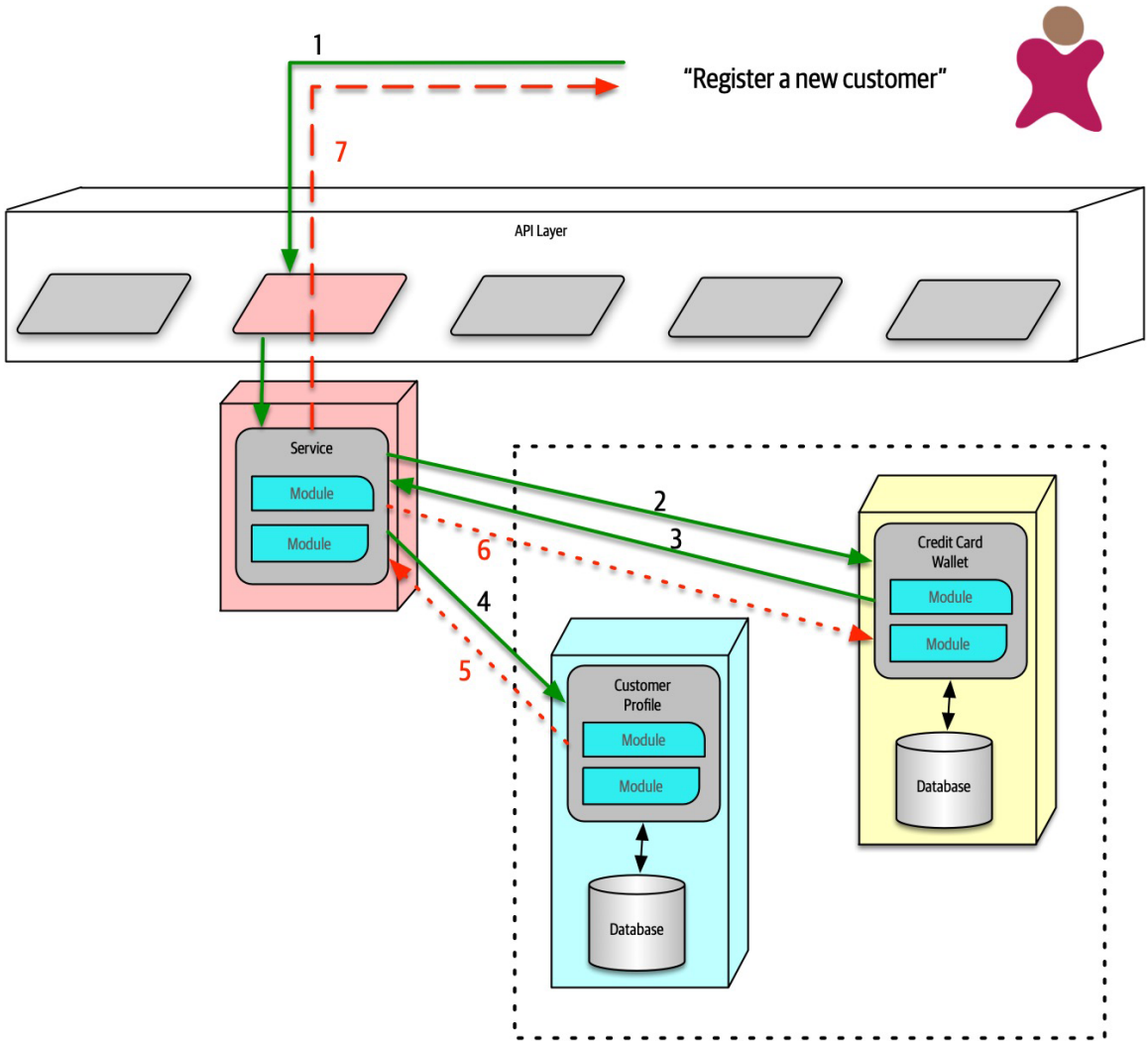

Figure 17-12. Saga pattern compensating transactions for error conditions

In Figure 17-12, if the first part of the transaction succeeds, yet the second part fails, the mediator must send a request to all the parts of the transaction that were successful and tell them to undo the previous request. This style of transactional coordination is called a *compensating transaction framework*. Developers implement this pattern by usually having each request from the mediator enter a `pending` state until the mediator indicates overall success. However, this design becomes complex if asynchronous requests must be juggled, especially if new requests appear that are contingent on pending transactional state. This also creates a lot of coordination traffic at the network level.

Another implementation of a compensating transaction framework has developers build *do* and *undo* for each potentially transactional operation. This allows less coordination during transactions, but the *undo* operations tend to be significantly more complex than the *do* operations, more than doubling the design, implementation, and debugging work.

While it is possible for architects to build transactional behavior across services, it goes against the reason for choosing the microservices pattern. Exceptions always exist, so the best advice for architects is to use the saga pattern sparingly.

> **TIP**
>
> A few transactions across services is sometimes necessary; if it's the dominant feature of the architecture, mistakes were made!

Architecture Characteristics Ratings

The microservices architecture style offers several extremes on our standard ratings scale, shown in Figure 17-13. A one-star rating means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in Chapter 4.

Architecture characteristic	Star rating
Partitioning type	Domain
Number of quanta	1 to many
Deployability	★★★★
Elasticity	★★★★★
Evolutionary	★★★★★
Fault tolerance	★★★★
Modularity	★★★★★
Overall cost	★
Performance	★★
Reliability	★★★★
Scalability	★★★★★
Simplicity	★
Testability	★★★★

Figure 17-13. Ratings for microservices

Notable in the ratings in Figure 17-13 is the high support for modern engineering practices such as automated deployment, testability, and others not listed. Microservices couldn't exist without the DevOps revolution and the relentless march toward automating operational concerns.

As microservices is a distributed architecture, it suffers from many of the deficiencies inherent in architectures made from pieces wired together at runtime. Thus, fault tolerance and reliability are impacted when too much

interservice communication is used. However, these ratings only point to tendencies in the architecture; developers fix many of these problems by redundancy and scaling via service discovery. Under normal circumstances, however, independent, single-purpose services generally lead to high fault tolerance, hence the high rating for this characteristic within a microservices architecture.

The high points of this architecture are scalability, elasticity, and evolutionary. Some of the most scalable systems yet written have utilized this style to great success. Similarly, because the architecture relies heavily on automation and intelligent integration with operations, developers can also build elasticity support into the architecture. Because the architecture favors high decoupling at an incremental level, it supports the modern business practice of evolutionary change, even at the architecture level. Modern business move fast, and software development has struggled to keep apace. By building an architecture that has extremely small deployment units that are highly decoupled, architects have a structure that can support a faster rate of change.

Performance is often an issue in microservices—distributed architectures must make many network calls to complete work, which has high performance overhead, and they must invoke security checks to verify identity and access for each endpoint. Many patterns exist in the microservices world to increase performance, including intelligent data caching and replication to prevent an excess of network calls. Performance is another reason that microservices often use choreography rather than orchestration, as less coupling allows for faster communication and fewer bottlenecks.

Microservices is decidedly a domain-centered architecture, where each service boundary should correspond to domains. It also has the most distinct quanta of any modern architecture—in many ways, it exemplifies what the quantum measure evaluates. The driving philosophy of extreme decoupling creates many headaches in this architecture but yields tremendous benefits when done well. As in any architecture, architects must understand the rules to break them intelligently.

Additional References

While our goal in this chapter was to touch on some of the significant aspects of this architecture style, many excellent resources exist to get further and more detailed about this architecture style. Additional and more detailed information can be found about microservices in the following references:

- *Building Microservices* by Sam Newman (O'Reilly)
- *Microservices vs. Service-Oriented Architecture* by Mark Richards (O'Reilly)
- *Microservices AntiPatterns and Pitfalls* by Mark Richards (O'Reilly)

Chapter 18. Choosing the Appropriate Architecture Style

It depends! With all the choices available (and new ones arriving almost daily), we would like to tell you which one to use—but we cannot. Nothing is more contextual to a number of factors within an organization and what software it builds. Choosing an architecture style represents the culmination of analysis and thought about trade-offs for architecture characteristics, domain considerations, strategic goals, and a host of other things.

However contextual the decision is, some general advice exists around choosing an appropriate architecture style.

Shifting "Fashion" in Architecture

Preferred architecture styles shift over time, driven by a number of factors:

Observations from the past

> New architecture styles generally arise from observations and pain points from past experiences. Architects have experience with systems in the past that influence their thoughts about future systems. Architects must rely on their past experience—it is that experience that allowed that person to become an architect in the first place. Often, new architecture designs reflect specific deficiencies from past architecture styles. For example, architects seriously rethought the implications of code reuse after building architectures that featured it and then realizing the negative trade-offs.

Changes in the ecosystem

> Constant change is a reliable feature of the software development ecosystem—everything changes all the time. The change in our

ecosystem is particularly chaotic, making even the type of change impossible to predict. For example, a few years ago, no one knew what *Kubernetes* was, and now there are multiple conferences around the world with thousands of developers. In a few more years, Kubernetes may be replaced with some other tool that hasn't been written yet.

New capabilities

When new capabilities arise, architecture may not merely replace one tool with another but rather shift to an entirely new paradigm. For example, few architects or developers anticipated the tectonic shift caused in the software development world by the advent of containers such as Docker. While it was an evolutionary step, the impact it had on architects, tools, engineering practices, and a host of other factors astounded most in the industry. The constant change in the ecosystem also delivers a new collection of tools and capabilities on a regular basis. Architects must keep a keen eye open to not only new tools but new paradigms. Something may just look like a new one-of-something-we-already-have, but it may include nuances or other changes that make it a game changer. New capabilities don't even have to rock the entire development world—the new features may be a minor change that aligns exactly with an architect's goals.

Acceleration

Not only does the ecosystem constantly change, but the rate of change also continues to rise. New tools create new engineering practices, which lead to new design and capabilities. Architects live in a constant state of flux because change is both pervasive and constant.

Domain changes

The domain that developers write software for constantly shifts and changes, either because the business continues to evolve or because of factors like mergers with other companies.

Technology changes

As technology continues to evolve, organizations try to keep up with at least some of these changes, especially those with obvious bottom-line benefits.

External factors

Many external factors only peripherally associated with software development may drive change within an organizations. For example, architects and developers might be perfectly happy with a particular tool, but the licensing cost has become prohibitive, forcing a migration to another option.

Regardless of where an organization stands in terms of current architecture fashion, an architect should understand current industry trends to make intelligent decisions about when to follow and when to make exceptions.

Decision Criteria

When choosing an architectural style, an architect must take into account all the various factors that contribute to the structure for the domain design. Fundamentally, an architect designs two things: whatever domain has been specified, and all the other structural elements required to make the system a success.

Architects should go into the design decision comfortable with the following things:

The domain

Architects should understand many important aspects of the domain, especially those that affect operational architecture characteristics. Architects don’t have to be subject matter experts, but they must have at least a good general understanding of the major aspects of the domain under design.

Architecture characteristics that impact structure

Architects must discover and elucidate the architecture characteristics needed to support the domain and other eternal factors.

Data architecture

Architects and DBAs must collaborate on database, schema, and other data-related concerns. We don't cover much about data architecture in this book; it is its own specialization. However, architects must understand the impact that data design might have on their design, particularly if the new system must interact with an older and/or in-use data architecture.

Organizational factors

Many external factors may influence design. For example, the cost of a particular cloud vendor may prevent the ideal design. Or perhaps the company plans to engage in mergers and acquisitions, which encourages an architect to gravitate toward open solutions and integration architectures.

Knowledge of process, teams, and operational concerns

Many specific project factors influence an architect's design, such as the software development process, interaction (or lack of) with operations, and the QA process. For example, if an organization lacks maturity in Agile engineering practices, architecture styles that rely on those practices for success will present difficulties.

Domain/architecture isomorphism

Some problem domains match the topology of the architecture. For example, the microkernel architecture style is perfectly suited to a system that requires customizability—the architect can design customizations as plug-ins. Another example might be genome analysis, which requires a large number of discrete operations, and space-based architecture, which offers a large number of discrete processors.

Similarly, some problem domains may be particularly ill-suited for some architecture styles. For example, highly scalable systems struggle

with large monolithic designs because architects find it difficult to support a large number of concurrent users in a highly coupled code base. A problem domain that includes a huge amount of semantic coupling matches poorly with a highly decoupled, distributed architecture. For instance, an insurance company application consisting of multipage forms, each of which is based on the context of previous pages, would be difficult to model in microservices. This is a highly coupled problem that will present architects with design challenges in a decoupled architecture; a less coupled architecture like service-based architecture would suit this problem better.

Taking all these things into account, the architect must make several determinations:

Monolith versus distributed

Using the quantum concepts discussed earlier, the architect must determine if a single set of architecture characteristics will suffice for the design, or do different parts of the system need differing architecture characteristics? A single set implies that a monolith is suitable (although other factors may drive an architect toward a distributed architecture), whereas different architecture characteristics imply a distributed architecture.

Where should data live?

If the architecture is monolithic, architects commonly assume a single relational databases or a few of them. In a distributed architecture, the architect must decide which services should persist data, which also implies thinking about how data must flow throughout the architecture to build workflows. Architects must consider both structure and behavior when designing architecture and not be fearful of iterating on the design to find better combinations.

What communication styles between services—synchronous or asynchronous?

Once the architect has determined data partitioning, their next design consideration is the communication between services—synchronous or asynchronous? Synchronous communication is more convenient in most cases, but it can lead to scalability, reliability, and other undesirable characteristics. Asynchronous communication can provide unique benefits in terms of performance and scale but can present a host of headaches: data synchronization, deadlocks, race conditions, debugging, and so on.

Because synchronous communication presents fewer design, implementation, and debugging challenges, architects should default to synchronous when possible and use asynchronous when necessary.

> **TIP**
>
> Use synchronous by default, asynchronous when necessary.

The output of this design process is architecture topology, taking into account what architecture style (and hybridizations) the architect chose, architecture decision records about the parts of the design which required the most effort by the architect, and architecture fitness functions to protect important principles and operational architecture characteristics.

Monolith Case Study: Silicon Sandwiches

In the Silicon Sandwiches architecture kata, after investigating the architecture characteristics, we determined that a single quantum was sufficient to implement this system. Plus, this is a simple application without a huge budget, so the simplicity of a monolith appeals.

However, we created two different component designs for Silicon Sandwiches: one domain partitioned and another technically partitioned. Given the simplicity of the solution, we'll create designs for each and cover trade-offs.

Modular Monolith

A modular monolith builds domain-centric components with a single database, deployed as a single quantum; the modular monolith design for Silicon Sandwiches appears in Figure 18-1.

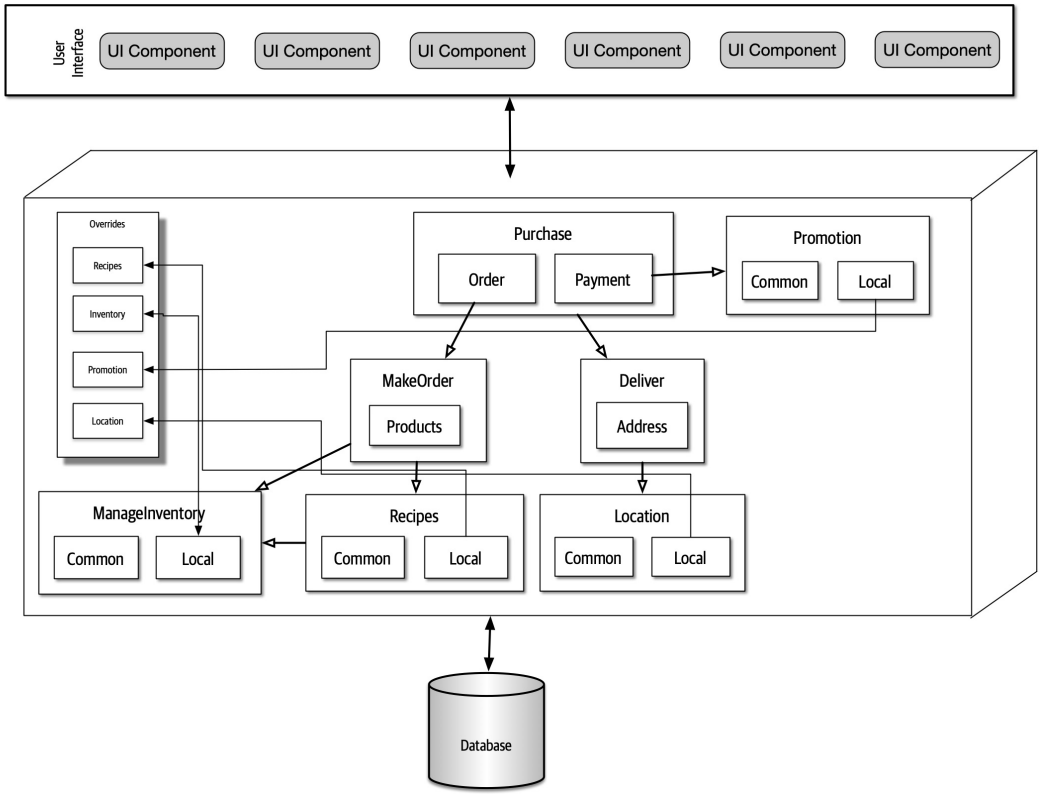

Figure 18-1. A modular monolith implementation of Silicon Sandwiches

In Figure 18-1, this is a monolith with a single relational database, implemented with a single web-based user interface (with careful design considerations for mobile devices) to keep overall cost down. Each of the domains the architect identified earlier appear as components. If time and resources are sufficient, the architect should consider creating the same separation of tables and other database assets as the domain components, allowing for this architecture to migrate to a distributed architecture more easily if future requirements warrant it.

Because the architecture style itself doesn't inherently handle customization, the architect must make sure that that feature becomes part of domain design. In this case, the architect designs an `Override` endpoint

where developers can upload individual customizations. Correspondingly, the architect must ensure that each of the domain components references the `Override` component for each customizable characteristic—this would make a perfect fitness function.

Microkernel

One of the architecture characteristics the architect identified in Silicon Sandwiches was customizability. Looking at domain/architecture isomorphism, an architect may choose to implement it using a microkernel, as illustrated in Figure 18-2.

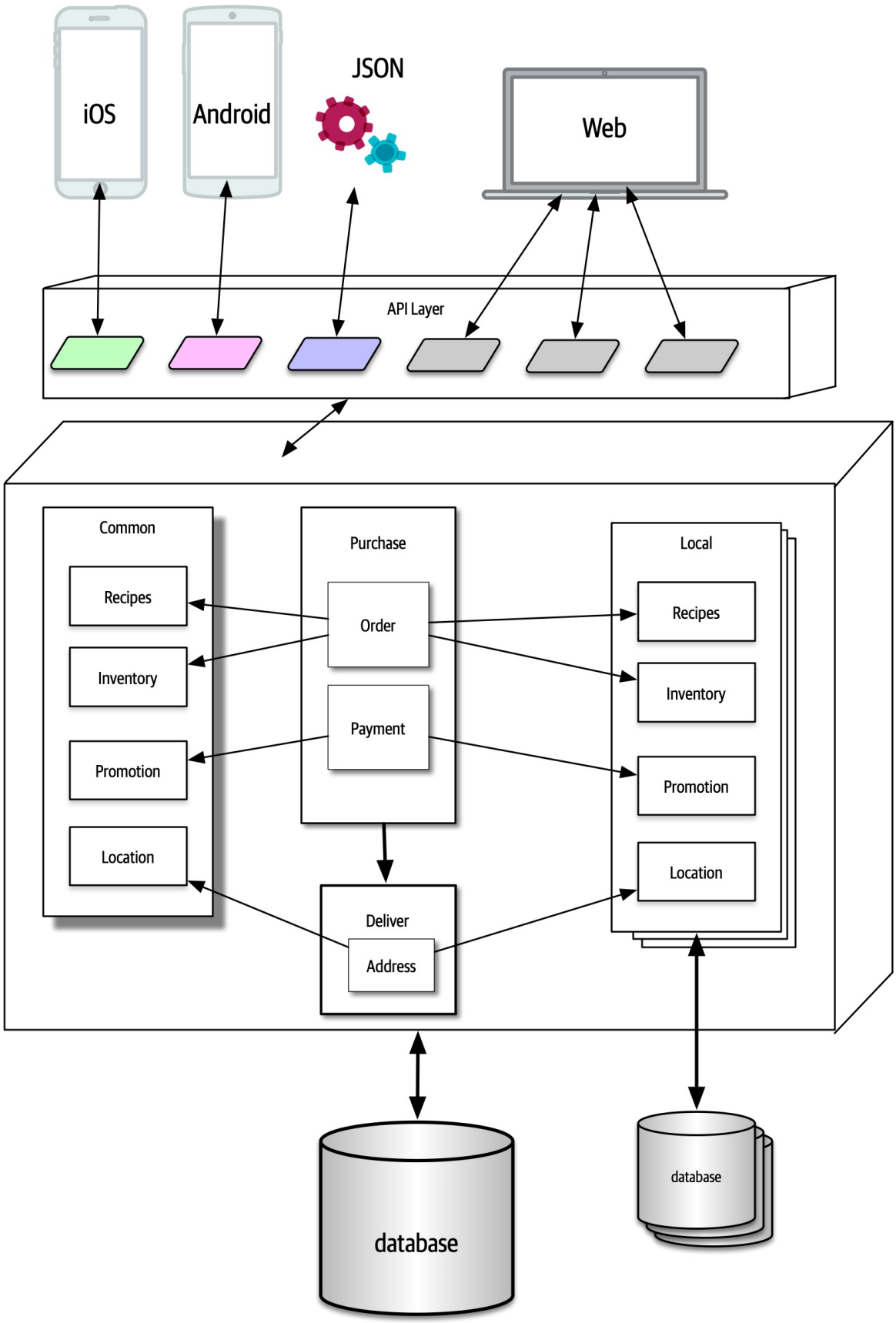
iOS
Android
JSON
Web
API Layer
Common
Recipes
Inventory
Promotion
Location
Purchase
Order
Payment
Deliver
Address
Local
Recipes
Inventory
Promotion
Location
database
database

Figure 18-2. A microkernel implementation of Silicon Sandwiches

In Figure 18-2, the core system consists of the domain components and a single relational database. As in the previous design, careful synchronization between domains and data design will allow future migration of the core to a distributed architecture. Each customization appears in a plug-in, the common ones in a single set of plug-ins (with a corresponding database), and a series of local ones, each with their own data. Because none of the plug-ins need to be coupled to the other plug-ins, they can each maintain their data, leaving the plug-ins decoupled.

The other unique design element here utilizes the Backends for Frontends (BFF) pattern, making the API layer a thin microkernel adaptor. It supplies general information from the backend, and the BFF adaptors translate the generic information into the suitable format for the frontend device. For example, the BFF for iOS will take the generic output from the backend and customize it for what the iOS native application expects: the data format, pagination, latency, and other factors. Building each BFF adaptor allows for the richest user interfaces and the ability to expand to support other devices in the future—one of the benefits of the microkernel style.

Communication within either Silicon Sandwich architecture can be synchronous—the architecture doesn't require extreme performance or elasticity requirements—and none of the operations will be lengthy.

Distributed Case Study: Going, Going, Gone

The Going, Going, Gone (GGG) kata presents more interesting architecture challenges. Based on the component analysis in "Case Study: Going, Going, Gone: Discovering Components", this architecture needs differing architecture characteristics for different parts of the architecture. For example, architecture characteristics like availability and scalability will differ between roles like auctioneer and bidder.

The requirements for GGG also explicitly state certain ambitious levels of scale, elasticity, performance, and a host of other tricky operational

architecture characteristics. The architect needs to choose a pattern that allows for a high degree of customization at a fine-grained level within the architecture. Of the candidate distributed architectures, either low-level event-driven or microservices match most of the architecture characteristics. Of the two, microservices better supports differing operational architecture characteristics—purely event-driven architectures typically don't separate pieces because of these operational architecture characteristics but are rather based on communication style, orchestrated versus choreographed.

Achieving the stated performance will provide a challenge in microservices, but architects can often address any weak point of an architecture by designing to accommodate it. For example, while microservices offers a high degrees of scalability naturally, architects commonly have to address specific performance issues caused by too much orchestration, too aggressive data separation, and so on.

An implementation of GGG using microservices is shown in Figure 18-3.

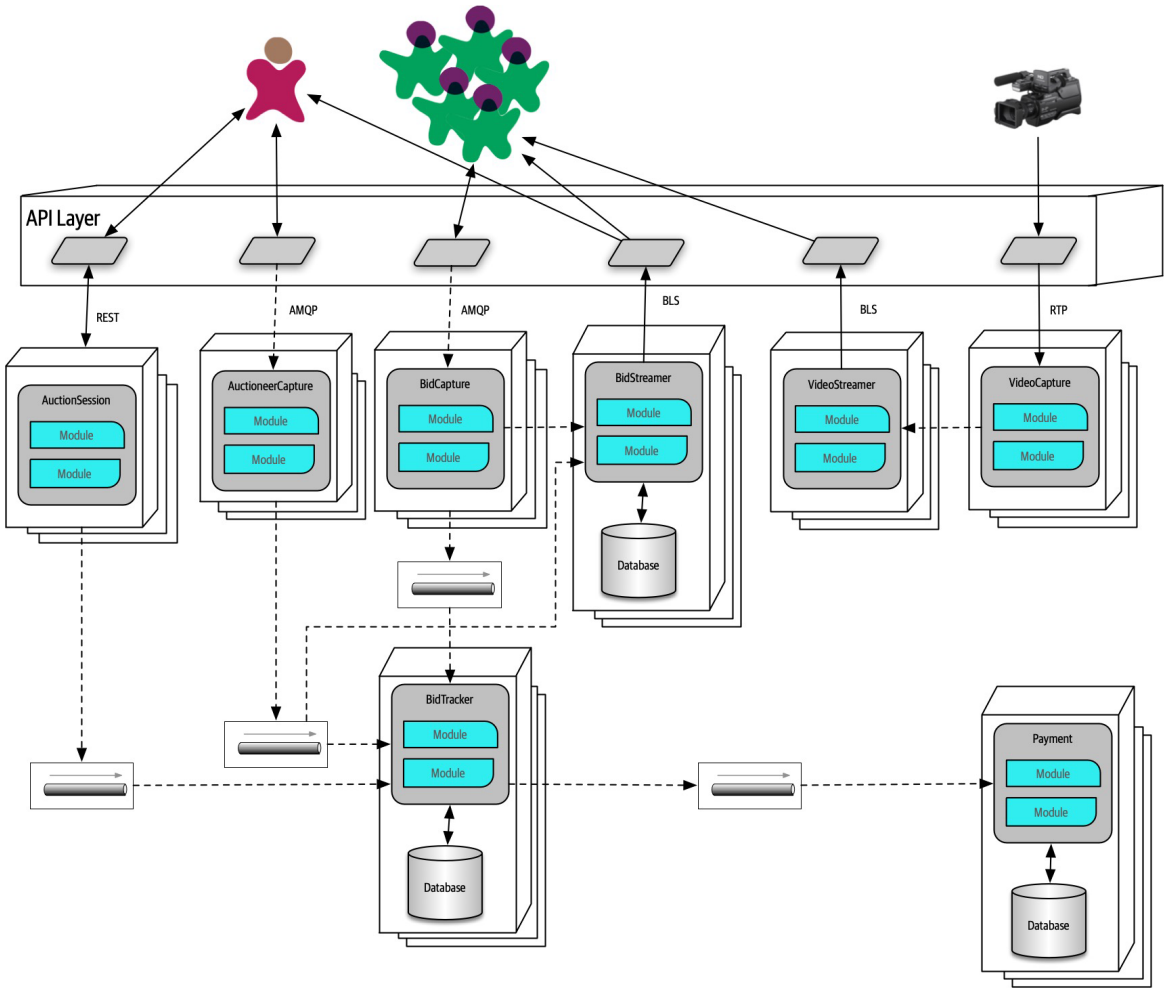

Figure 18-3. A microservices implementation of Going, Going, Gone

In Figure 18-3, each identified component became services in the architecture, matching component and service granularity. GGG has three distinct user interfaces:

Bidder

The numerous bidders for the online auction.

Auctioneer

One per auction.

Streamer

Service responsible for streaming video and bid stream to the bidders. Note that this is a read-only stream, allowing optimizations not

available if updates were necessary.

The following services appear in this design of the GGG architecture:

`BidCapture`

Captures online bidder entries and asynchronously sends them to `Bid Tracker`. This service needs no persistence because it acts as a conduit for the online bids.

`BidStreamer`

Streams the bids back to online participants in a high performance, read-only stream.

`BidTracker`

Tracks bids from both `Auctioneer Capture` and `Bid Capture`. This is the component that unifies the two different information streams, ordering the bids as close to real time as possible. Note that both inbound connections to this service are asynchronous, allowing the developers to use message queues as buffers to handle very different rates of message flow.

`Auctioneer Capture`

Captures bids for the auctioneer. The result of quanta analysis in “Case Study: Going, Going, Gone: Discovering Components” led the architect to separate `Bid Capture` and `Auctioneer Capture` because they have quite different architecture characteristics.

`Auction Session`

This manages the workflow of individual auctions.

`Payment`

Third-party payment provider that handles payment information after the `Auction Session` has completed the auction.

Video Capture

Captures the video stream of the live auction.

Video Streamer

Streams the auction video to online bidders.

The architect was careful to identify both synchronous and asynchronous communication styles in this architecture. Their choice for asynchronous communication is primarily driven by accommodating differing operational architecture characteristics between services. For example, if the `Payment` service can only process a new payment every 500 ms and a large number of auctions end at the same time, synchronous communication between the services would cause time outs and other reliability headaches. By using message queues, the architect can add reliability to a critical part of the architecture that exhibits fragility.

In the final analysis, this design resolved to five quanta, identified in Figure 18-4.

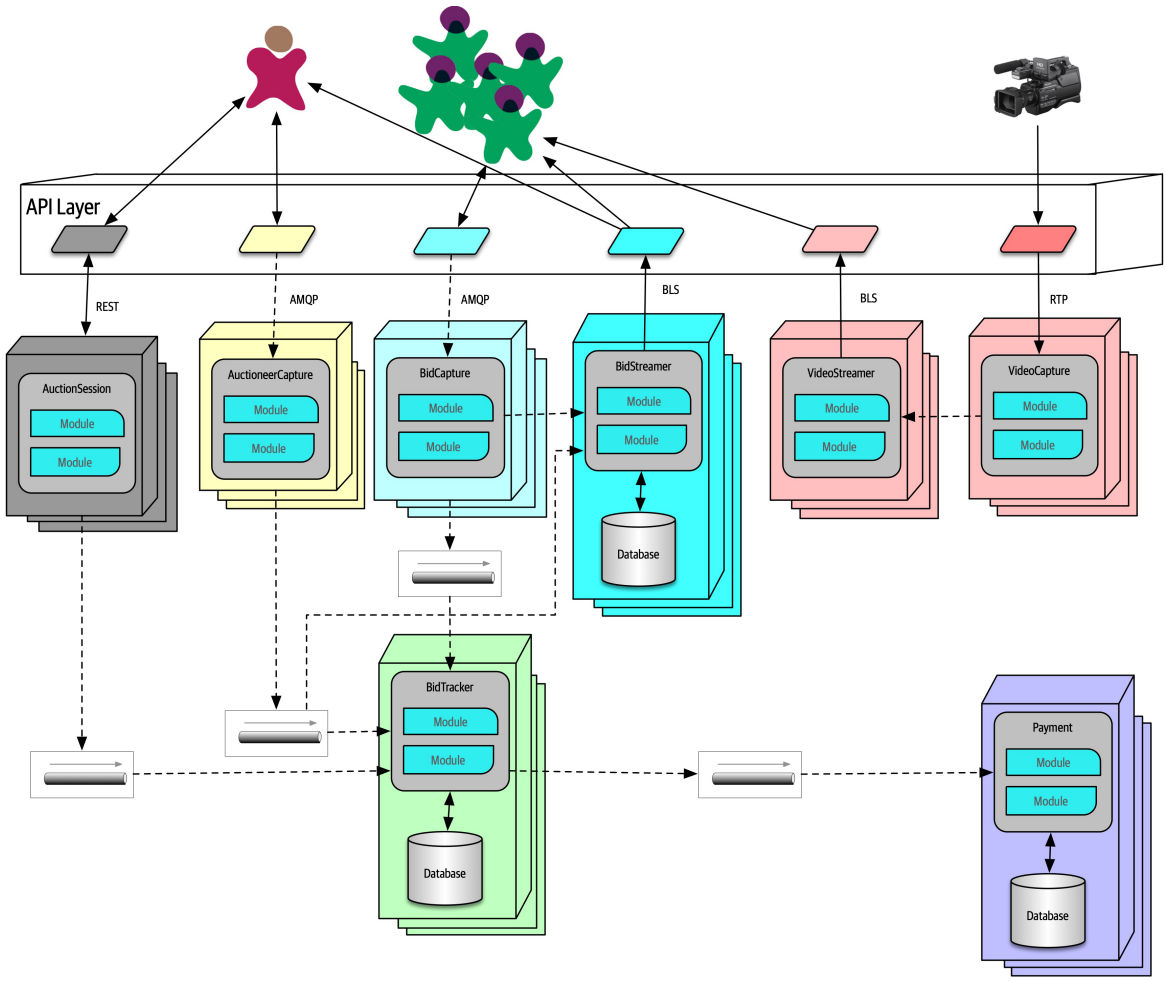

Figure 18-4. The quanta boundaries for GGG

In Figure 18-4, the design includes quanta for `Payment`, `Auctioneer`, `Bidder`, `Bidder Streams`, and `Bid Tracker`, roughly corresponding to the services. Multiple instances are indicated by stacks of containers in the diagram. Using quantum analysis at the component design stage allowed the architect to more easily identify service, data, and communication boundaries.

Note that this isn't the "correct" design for GGG, and it's certainly not the only one. We don't even suggest that it's the best possible design, but it seems to have the *least worst* set of trade-offs. Choosing microservices, then intelligently using events and messages, allows the architecture to leverage the most out of a generic architecture pattern while still building a foundation for future development and expansion.

Part III. Techniques and Soft Skills

An effective software architect must not only understand the technical aspects of software architecture, but also the primary techniques and soft skills necessary to think like an architect, guide development teams, and effectively communicate the architecture to various stakeholders. This section of the book addresses the key techniques and soft skills necessary to become an effective software architect.

Chapter 19. Architecture Decisions

One of the core expectations of an architect is to make architecture decisions. Architecture decisions usually involve the structure of the application or system, but they may involve technology decisions as well, particularly when those technology decisions impact architecture characteristics. Whatever the context, a good architecture decision is one that helps guide development teams in making the right technical choices. Making architecture decisions involves gathering enough relevant information, justifying the decision, documenting the decision, and effectively communicating that decision to the right stakeholders.

Architecture Decision Anti-Patterns

There is an art to making architecture decisions. Not surprisingly, several architecture anti-patterns emerge when making decisions as an architect. The programmer Andrew Koenig defines an anti-pattern as something that seems like a good idea when you begin, but leads you into trouble. Another definition of an anti-pattern is a repeatable process that produces negative results. The three major architecture anti-patterns that can (and usually do) emerge when making architecture decisions are the *Covering Your Assets* anti-pattern, the *Groundhog Day* anti-pattern, and the *Email-Driven Architecture* anti-pattern. These three anti-patterns usually follow a progressive flow: overcoming the Covering Your Assets anti-pattern leads to the Groundhog Day anti-pattern, and overcoming this anti-pattern leads to the Email-Driven Architecture anti-pattern. Making effective and accurate architecture decisions requires an architect to overcome all three of these anti-patterns.

Covering Your Assets Anti-Pattern

The first anti-pattern to emerge when trying to make architecture decisions is the Covering Your Assets anti-pattern. This anti-pattern occurs when an architect avoids or defers making an architecture decision out of fear of making the wrong choice.

There are two ways to overcome this anti-pattern. The first is to wait until the *last responsible moment* to make an important architecture decision. The last responsible moment means waiting until you have enough information to justify and validate your decision, but not waiting so long that you hold up development teams or fall into the *Analysis Paralysis* anti-pattern. The second way to avoid this anti-pattern is to continually collaborate with development teams to ensure that the decision you made can be implemented as expected. This is vitally important because it is not feasible as an architect to possibly know every single detail about a particular technology and all the associated issues. By closely collaborating with development teams, the architect can respond quickly to a change in the architecture decision if issues occur.

To illustrate this point, suppose an architect makes the decision that all product-related reference data (product description, weight, and dimensions) be cached in all service instances needing that information using a read-only replicated cache, with the primary replica owned by the catalog service. A replicated cache means that if there are any changes to product information (or a new product is added), the catalog service would update its cache, which would then be replicated to all other services requiring that data through a replicated (in-memory) cache product. A good justification for this decision is to reduce coupling between the services and to effectively share data without having to make an interservice call. However, the development teams implementing this architecture decision find that due to certain scalability requirements of some of the services, this decision would require more in-process memory than is available. By closely collaborating with the development teams, the architect can quickly become aware of the issue and adjust the architecture decision to accommodate these situations.

Groundhog Day Anti-Pattern

Once an architect overcomes the Covering Your Assets anti-pattern and starts making decisions, a second anti-pattern emerges: the Groundhog Day anti-pattern. The Groundhog Day anti-pattern occurs when people don't know why a decision was made, so it keeps getting discussed over and over and over. The Groundhog Day anti-pattern gets it name from the Bill Murray movie *Groundhog Day*, where it was February 2 over and over every day.

The Groundhog Day anti-pattern occurs because once an architect makes an architecture decision, they fail to provide a justification for the decision (or a complete justification). When justifying architecture decisions it is important to provide both technical and business justifications for your decision. For example, an architect may make the decision to break apart a monolithic application into separate services to decouple the functional aspects of the application so that each part of the application uses fewer virtual machine resources and can be maintained and deployed separately. While this is a good example of a technical justification, what is missing is the business justification—in other words, why should the business pay for this architectural refactoring? A good business justification for this decision might be to deliver new business functionality faster, therefore improving time to market. Another might be to reduce the costs associated with the development and release of new features.

Providing the business value when justifying decisions is vitally important for any architecture decision. It is also a good litmus test for determining whether the architecture decision should be made in the first place. If a particular architecture decision does not provide any business value, then perhaps it is not a good decision and should be reconsidered.

Four of the most common business justifications include cost, time to market, user satisfaction, and strategic positioning. When focusing on these common business justifications, it is important to take into consideration what is important to the business stakeholders. Justifying a particular decision based on cost savings alone might not be the right decision if the

business stakeholders are less concerned about cost and more concerned about time to market.

Email-Driven Architecture Anti-Pattern

Once an architect makes decisions and fully justifies those decisions, a third architecture anti-pattern emerges: *Email-Driven Architecture*. The Email-Driven Architecture anti-pattern is where people lose, forget, or don't even know an architecture decision has been made and therefore cannot possibly implement that architecture decision. This anti-pattern is all about effectively communicating your architecture decisions. Email is a great tool for communication, but it makes a poor document repository system.

There are many ways to increase the effectiveness of communicating architecture decisions, thereby avoiding the Email-Driven Architecture anti-pattern. The first rule of communicating architecture decisions is to not include the architecture decision in the body of an email. Including the architecture decision in the body of the email creates multiple systems of record for that decision. Many times important details (including the justification) are left out of the email, therefore creating the Groundhog Day anti-pattern all over again. Also, if that architecture decision is ever changed or superseded, how may people received the revised decision? A better approach is to mention only the nature and context of the decision in the body of the email and provide a link to the single system of record for the actual architecture decision and corresponding details (whether it be a link to a wiki page or a document in a filesystem).

The second rule of effectively communicating architecture decisions is to only notify those people who really care about the architecture decision. One effective technique is to write the body of the email as follows:

"Hi Sandra, I've made an important decision regarding communication between services that directly impacts you. Please see the decision using the following link…"

Notice the phrasing in the first sentence: "important decision regarding communication between services." Here, the context of the decision is

mentioned, but not the actual decision itself. The second part of the first sentence is even more important: “that directly impacts you.” If an architectural decision doesn’t directly impact the person, then why bother that person with your architecture decision? This is a great litmus test for determining which stakeholders (including developers) should be notified directly of an architecture decision. The second sentence provides a link to the location of the architecture decision so it is located in only one place, hence a single system of record for the decision.

Architecturally Significant

Many architects believe that if the architecture decision involves any specific technology, then it’s not an architecture decision, but rather a technical decision. This is not always true. If an architect makes a decision to use a particular technology because it directly supports a particular architecture characteristic (such as performance or scalability), then it’s an architecture decision.

Michael Nygard, a well-known software architect and author of *Release It!* (Pragmatic Bookshelf), addressed the problem of what decisions an architect should be responsible for (and hence what is an architecture decision) by coining the term *architecturally significant*. According to Michael, architecturally significant decisions are those decisions that affect the structure, nonfunctional characteristics, dependencies, interfaces, or construction techniques.

The *structure* refers to decisions that impact the patterns or styles of architecture being used. An example of this is the decision to share data between a set of microservices. This decision impacts the bounded context of the microservice, and as such affects the structure of the application.

The *nonfunctional characteristics* are the architecture characteristics (“-ilities”) that are important for the application or system being developed or maintained. If a choice of technology impacts performance, and performance is an important aspect of the application, then it becomes an architecture decision.

Dependencies refer to coupling points between components and/or services within the system, which in turn impact overall scalability, modularity, agility, testability, reliability, and so on.

Interfaces refer to how services and components are accessed and orchestrated, usually through a gateway, integration hub, service bus, or API proxy. Interfaces usually involve defining contracts, including the versioning and deprecation strategy of those contracts. Interfaces impact others using the system and hence are architecturally significant.

Finally, *construction techniques* refer to decisions about platforms, frameworks, tools, and even processes that, although technical in nature, might impact some aspect of the architecture.

Architecture Decision Records

One of the most effective ways of documenting architecture decisions is through *Architecture Decision Records* (ADRs). ADRs were first evangelized by Michael Nygard in a blog post and later marked as "adopt" in the ThoughtWorks Technology Radar. An ADR consists of a short text file (usually one to two pages long) describing a specific architecture decision. While ADRs can be written using plain text, they are usually written in some sort of text document format like AsciiDoc or Markdown. Alternatively, an ADR can also be written using a wiki page template.

Tooling is also available for managing ADRs. Nat Pryce, coauthor of *Growing Object-Oriented Software Guided by Tests* (Addison-Wesley), has written an open source tool for ADRs called ADR-tools. ADR-tools provides a command-line interface to manage ADRs, including the numbering schemes, locations, and superseded logic. Micha Kops, a software engineer from Germany, has written a blog post about using ADR-tools that provides some great examples on how they can be used to manage architecture decision records.

Basic Structure

The basic structure of an ADR consists of five main sections: *Title*, *Status*, *Context*, *Decision*, and *Consequences*. We usually add two additional sections as part of the basic structure: *Compliance* and *Notes*. This basic structure (as illustrated in Figure 19-1) can be extended to include any other section deemed needed, providing the template is kept both consistent and concise. A good example of this might be to add an *Alternatives* section if necessary to provide an analysis of all the other possible alternative solutions.

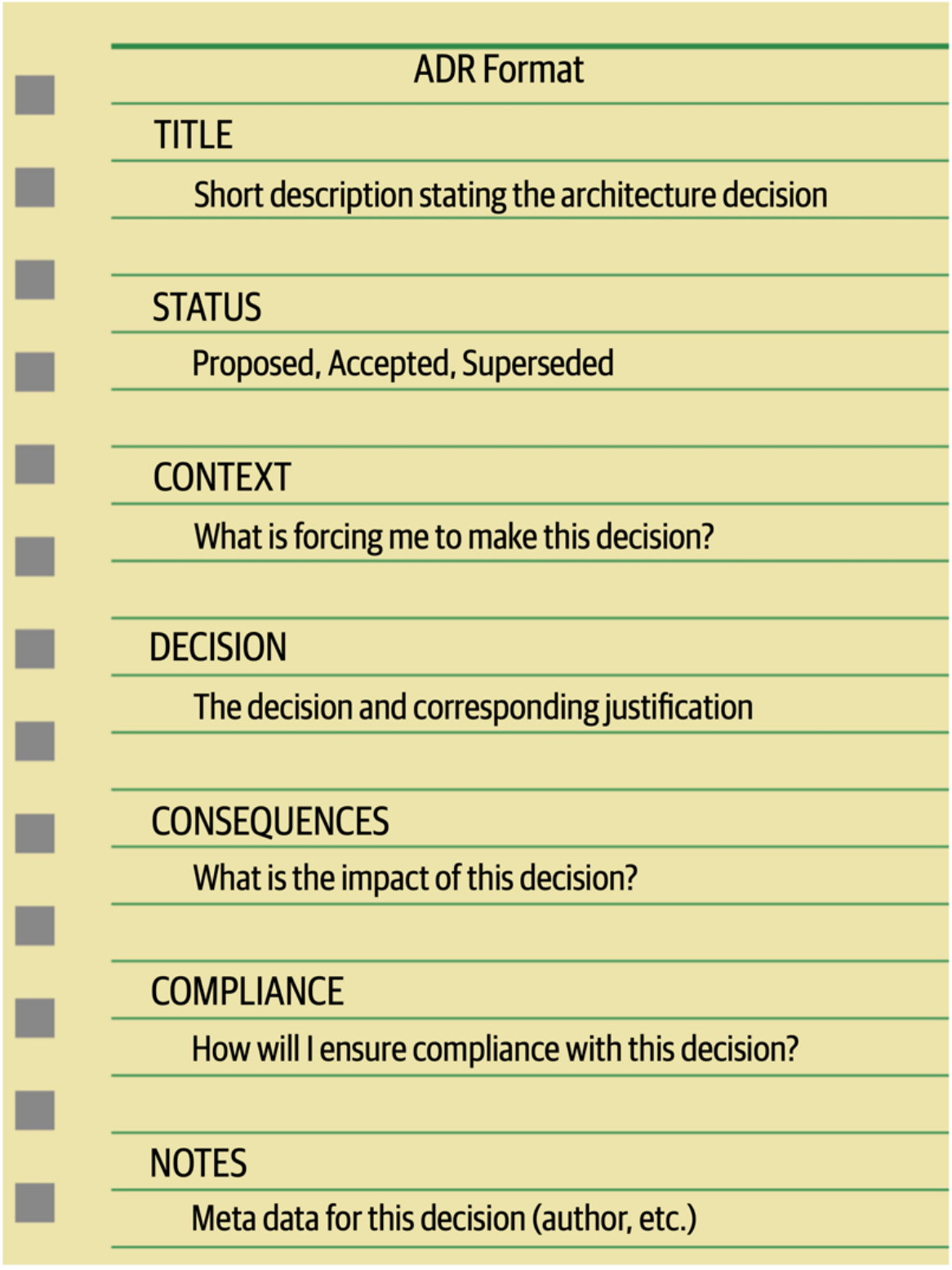

Figure 19-1. Basic ADR structure

Title

The title of an ADR is usually numbered sequentially and contains a short phase describing the architecture decisions. For example, the decision to use asynchronous messaging between the Order Service and the Payment Service might read: “42. Use of Asynchronous Messaging Between Order and Payment Services.” The title should be descriptive enough to remove any ambiguity about the nature and context of the decision but at the same time be short and concise.

Status

The status of an ADR can be marked as *Proposed*, *Accepted*, or *Superseded*. *Proposed* status means the decision must be approved by either a higher-level decision maker or some sort of architectural governance body (such as an architecture review board). *Accepted* status means the decision has been approved and is ready for implementation. A status of *Superseded* means the decision has been changed and superseded by another ADR. Superseded status always assumes the prior ADR status was accepted; in other words, a proposed ADR would never be superseded by another ADR, but rather continued to be modified until accepted.

The Superseded status is a powerful way of keeping a historical record of what decisions were made, why they were made at that time, and what the new decision is and why it was changed. Usually, when an ADR has been superseded, it is marked with the decision that superseded it. Similarly, the decision that supersedes another ADR is marked with the ADR it superseded. For example, assume ADR 42 (“Use of Asynchronous Messaging Between Order and Payment Services”) was previously approved, but due to later changes to the implementation and location of the Payment Service, REST must now be used between the two services (ADR 68). The status would look as follows:

ADR 42. Use of Asynchronous Messaging Between Order and Payment Services

Status: Superseded by 68

ADR 68. Use of REST Between Order and Payment Services

Status: Accepted, supersedes 42

The link and history trail between ADRs 42 and 68 avoid the inevitable "what about using messaging?" question regarding ADR 68.

ADRS AND REQUEST FOR COMMENTS (RFC)

If an architect wishes to send out a draft ADR for comments (which is sometimes a good idea when the architect wants to validate various assumptions and assertions with a larger audience of stakeholders), we recommend creating a new status named *Request for Comments* (or *RFC*) and specify a deadline date when that review would be complete. This practice avoids the inevitable Analysis Paralysis anti-pattern where the decision is forever discussed but never actually made. Once that date is reached, the architect can analyze all the comments made on the ADR, make any necessary adjustments to the decision, make the final decision, and set the status to Proposed (unless the architect is able to approve the decision themselves, in which case the status would then be set to Accepted). An example of an RFC status for an ADR would look as follows:

STATUS
Request For Comments, Deadline 09 JAN 2010

Another significant aspect of the Status section of an ADR is that it forces an architect to have necessary conversations with their boss or lead architect about the criteria with which they can approve an architecture decision on their own, or whether it must be approved through a higher-level architect, an architecture review board, or some other architecture governing body.

Three criteria that form a good start for these conversations are cost, cross-team impact, and security. Cost can include software purchase or licensing fees, additional hardware costs, as well as the overall level of effort to implement the architecture decision. Level of effort costs can be estimated by multiplying the estimated number of hours to implement the architecture decision by the company's standard *Full-Time Equivalency* (FTE) rate. The project owner or project manager usually has the FTE amount. If the cost of the architecture decision exceeds a certain amount, then it must be set to Proposed status and approved by someone else. If the architecture decision impacts other teams or systems or has any sort of security implication, then it cannot be self-approved by the architect and must be approved by a higher-level governing body or lead architect.

Once the criteria and corresponding limits have been established and agreed upon (such as "costs exceeding €5,000 must be approved by the architecture review board"), this criteria should be well documented so that all architects creating ADRs know when they can and cannot approve their own architecture decisions.

Context

The context section of an ADR specifies the forces at play. In other words, "what situation is forcing me to make this decision?" This section of the ADR allows the architect to describe the specific situation or issue and concisely elaborate on the possible alternatives. If an architect is required to document the analysis of each alternative in detail, then an additional Alternatives section can be added to the ADR rather than adding that analysis to the Context section.

The Context section also provides a way to document the architecture. By describing the context, the architect is also describing the architecture. This is an effective way of documenting a specific area of the architecture in a clear and concise manner. Continuing with the example from the prior section, the context might read as follows: "The order service must pass information to the payment service to pay for an order currently being placed. This could be done using REST or asynchronous messaging."

Notice that this concise statement not only specified the scenario, but also the alternatives.

Decision

The Decision section of the ADR contains the architecture decision, along with a full justification for the decision. Michael Nygard introduced a great way of stating an architecture decision by using a very affirmative, commanding voice rather than a passive one. For example, the decision to use asynchronous messaging between services would read "*we will use* asynchronous messaging between services." This is a much better way of stating a decision as opposed to "*I think* asynchronous messaging between services would be the best choice." Notice here it is not clear what the decision is or even if a decision has even been made—only the opinion of the architect is stated.

Perhaps one of the most powerful aspects of the Decision section of ADRs is that it allows an architect to place more emphasis on the *why* rather than the *how*. Understanding why a decision was made is far more important than understanding how something works. Most architects and developers can identify how things work by looking at context diagrams, but not why a decision was made. Knowing why a decision was made and the corresponding justification for the decision helps people better understand the context of the problem and avoids possible mistakes through refactoring to another solution that might produce issues.

To illustrate this point, consider an original architecture decision several years ago to use Google's Remote Procedure Call (gRPC) as a means to communicate between two services. Without understanding why that decision was made, another architect several years later makes the choice to override that decision and use messaging instead to better decouple the services. However, implementing this refactoring suddenly causes a significant increase in latency, which in turn ultimately causes time outs to occur in upstream systems. Understanding that the original use of gRPC was to significantly reduce latency (at the cost of tightly coupled services) would have prevented the refactoring from happening in the first place.

Consequences

The Consequences section of an ADR is another very powerful section. This section documents the overall impact of an architecture decision. Every architecture decision an architect makes has some sort of impact, both good and bad. Having to specify the impact of an architecture decision forces the architect to think about whether those impacts outweigh the benefits of the decision.

Another good use of this section is to document the trade-off analysis associated with the architecture decision. These trade-offs could be cost-based or trade-offs against other architecture characteristics ("-ilities"). For example, consider the decision to use asynchronous (fire-and-forget) messaging to post a review on a website. The justification for this decision is to significantly increase the responsiveness of the post review request from 3,100 milliseconds to 25 milliseconds because users would not need to wait for the actual review to be posted (only for the message to be sent to a queue). While this is a good justification, someone else might argue that this is a bad idea due to the complexity of the error handling associated with an asynchronous request ("what happens if someone posts a review with some bad words?"). Unknown to the person challenging this decision, that issue was already discussed with the business stakeholders and other architects, and it was decided from a trade-off perspective that it was more important to have the increase in responsiveness and deal with the complex error handling rather than have the wait time to synchronously provide feedback to the user that the review was successfully posted. By leveraging ADRs, that trade-off analysis can be included in the Consequences section, providing a complete picture of the context (and trade-offs) of the architecture decision and thus avoiding these situations.

Compliance

The compliance section of an ADR is not one of the standard sections in an ADR, but it's one we highly recommend adding. The Compliance section forces the architect to think about how the architecture decision will be measured and governed from a compliance perspective. The architect must

decide whether the compliance check for this decision must be manual or if it can be automated using a fitness function. If it can be automated using a fitness function, the architect can then specify in this section how that fitness function would be written and whether there are any other changes to the code base are needed to measure this architecture decision for compliance.

For example, consider the following architecture decision within a traditional n-tiered layered architecture as illustrated in Figure 19-2: “All shared objects used by business objects in the business layer will reside in the shared services layer to isolate and contain shared functionality.”

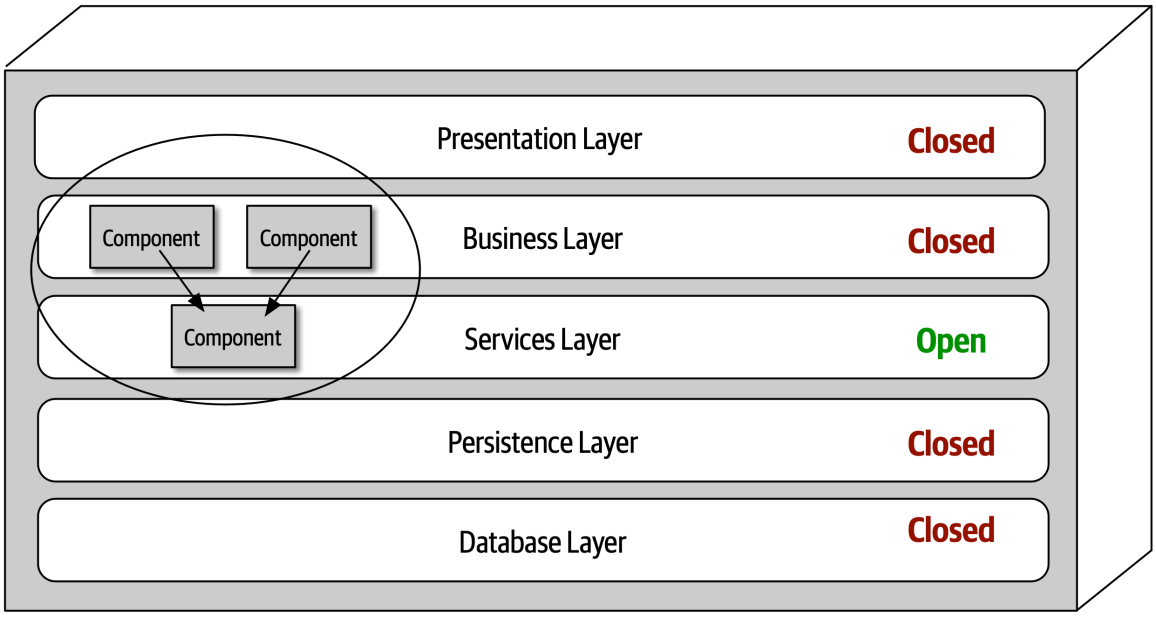

Figure 19-2. An example of an architecture decision

This architecture decision can be measured and governed automatically by using either ArchUnit in Java or NetArchTest in C#. For example, using ArchUnit in Java, the automated fitness function test might look as follows:

```
@Test
public void shared_services_should_reside_in_services_layer() {
    classes().that().areAnnotatedWith(SharedService.class)
        .should().resideInAPackage("..services..")
        .because("All shared services classes used by business " +
                 "objects in the business layer should reside in the services
" +
                 "layer to isolate and contain shared logic")
        .check(myClasses);
}
```

Notice that this automated fitness function would require new stories to be written to create a new Java annotation (`@SharedService`) and to then add this annotation to all shared classes. This section also specifies what the test is, where the test can be found, and how the test will be executed and when.

Notes

Another section that is not part of a standard ADR but that we highly recommend adding is the Notes section. This section includes various metadata about the ADR, such as the following:

- Original author
- Approval date
- Approved by
- Superseded date
- Last modified date
- Modified by
- Last modification

Even when storing ADRs in a version control system (such as Git), additional meta-information is useful beyond what the repository can support, so we recommend adding this section regardless of how and where ADRs are stored.

Storing ADRs

Once an architect creates an ADR, it must be stored somewhere. Regardless of where ADRs are stored, each architecture decision should have its own file or wiki page. Some architects like to keep ADRs in the Git repository with the source code. Keeping ADRs in a Git repository allows the ADR to be versioned and tracked as well. However, for larger organizations we caution against this practice for several reasons. First, everyone who needs to see the architecture decision may not have access to the Git repository. Second, this is not a good place to store ADRs that have a context outside of the application Git repository (such as integration architecture decisions, enterprise architecture decisions, or those decisions common to every application). For these reasons we recommend storing ADRs either in a wiki (using a wiki template) or in a shared directory on a shared file server that can be accessed easily by a wiki or other document rendering software. Figure 19-3 shows an example of what this directory structure (or wiki page navigation structure) might look like.

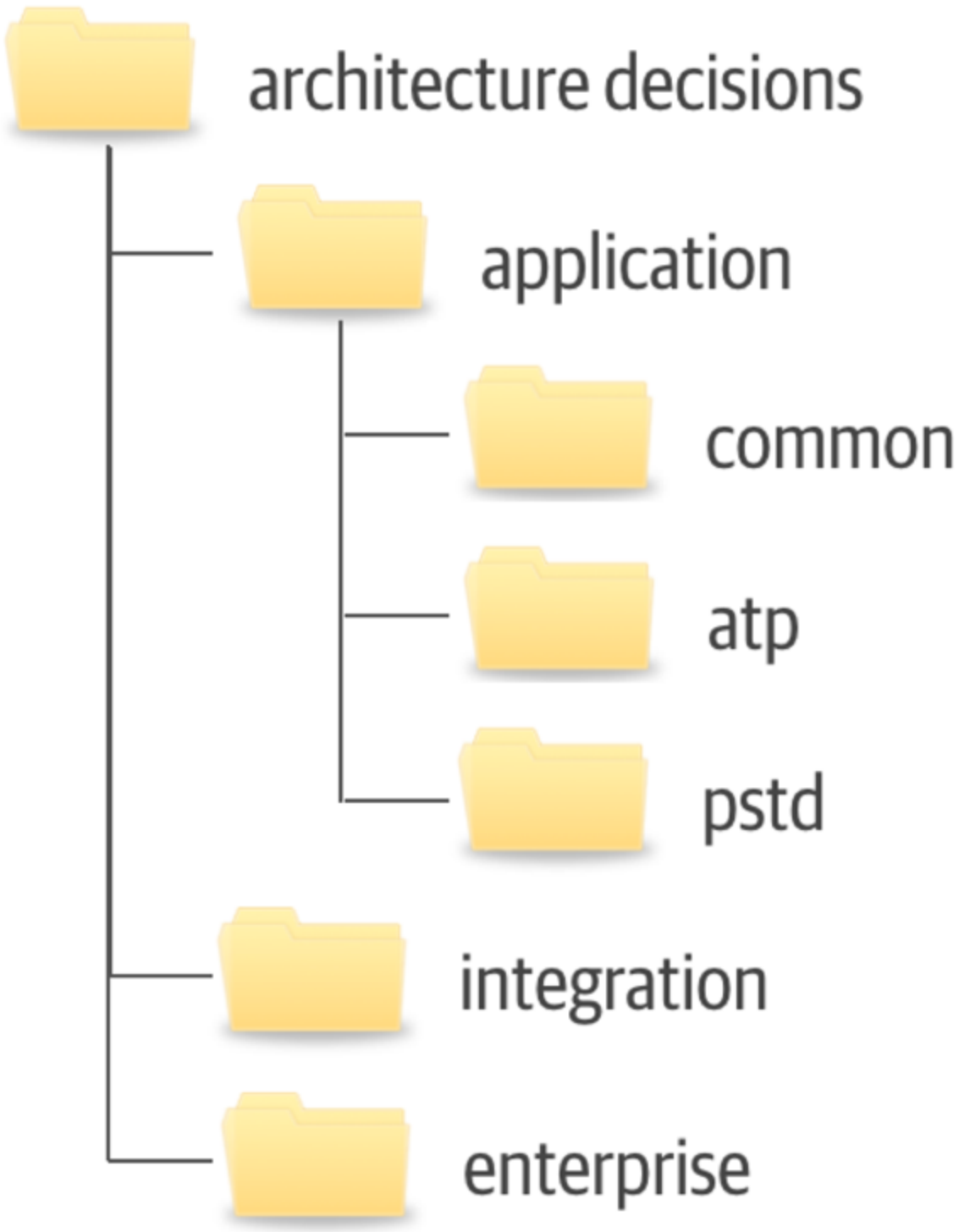

Figure 19-3. Example directory structure for storing ADRs

The *application* directory contains those architecture decisions that are specific to some sort of application context. This directory is subdivided into further directories. The *common* subdirectory is for architecture

decisions that apply to all applications, such as “All framework-related classes will contain an annotation (@Framework in Java) or attribute ([Framework] in C#) identifying the class as belonging to the underlying framework code.” Subdirectories under the *application* directory correspond to the specific application or system context and contain the architecture decisions specific to that application or system (in this example, the ATP and PSTD applications). The *integration* directory contains those ADRs that involve the communication between application, systems, or services. Enterprise architecture ADRs are contained within the *enterprise* directory, indicating that these are global architecture decisions impacting all systems and applications. An example of an enterprise architecture ADR would be “All access to a system database will only be from the owning system,” thus preventing the sharing of databases across multiple systems.

When storing ADRs in a wiki (our recommendation), the same structure previously described applies, with each directory structure representing a navigational landing page. Each ADR would be represented as a single wiki page within each navigational landing page (Application, Integration, or Enterprise).

The directory or landing page names indicated in this section are only a recommendation. Each company can choose whatever names fit their situation, as long as those names are consistent across teams.

ADRs as Documentation

Documenting software architecture has always been a difficult topic. While some standards are emerging for diagramming architecture (such as software architect Simon Brown’s C4 Model or The Open Group ArchiMate standard), no such standard exists for documenting software architecture. That’s where ADRs come in.

Architecture Decision Records can be used an an effective means to document a software architecture. The Context section of an ADR provides an excellent opportunity to describe the specific area of the system that

requires an architecture decision to be made. This section also provides an opportunity to describe the alternatives. Perhaps more important is that the Decision section describes the reasons why a particular decision is made, which is by far the best form of architecture documentation. The Consequences section adds the final piece to the architecture documentation by describing additional aspects of a particular decision, such as the trade-off analysis of choosing performance over scalability.

Using ADRs for Standards

Very few people like standards. Most times standards seem to be in place more for controlling people and the way they do things than anything useful. Using ADRs for standards can change this bad practice. For example, the Context section of an ADR describes the situation that is forcing the particular standard. The Decision section of an ADR can be used to not only indicate what the standard is, but more importantly why the standard needs to exist. This is a wonderful way of being able to qualify whether the particular standard should even exist in the first place. If an architect cannot justify the standard, then perhaps it is not a good standard to make and enforce. Furthermore, the more developers understand why a particular standard exists, the more likely they are to follow it (and correspondingly not challenge it). The Consequences section of an ADR is another great place an architect can qualify whether a standard is valid and should be made. In this section the architect must think about and document what the implications and consequences are of a particular standard they are making. By analyzing the consequences, the architect might decide that the standard should not be applied after all.

Example

Many architecture decisions exist within our ongoing “Case Study: Going, Going, Gone”. The use of event-driven microservices, the splitting up of the bidder and auctioneer user interfaces, the use of the Real-time Transport Protocol (RTP) for video capture, the use of a single API layer, and the use of publish-and-subscribe messaging are just a few of the dozens of

architecture decisions that are made for this auction system. Every architecture decision made in a system, no matter how obvious, should be documented and justified.

Figure 19-4 illustrates one of the architecture decisions within the Going, Going, Gone auction system, which is the use of publish-and-subscribe (pub/sub) messaging between the bid capture, bid streamer, and bid tracker services.

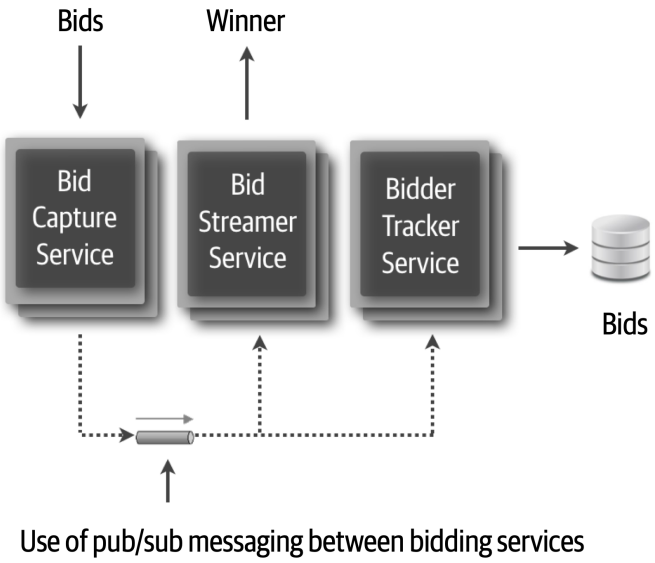

Figure 19-4. Use of pub/sub between services

The ADR for this architecture decision might look as follows:

ADR 76. Asynchronous Pub/Sub Messaging Between Bidding Services

STATUS
Accepted

CONTEXT
The Bid Capture Service, upon receiving a bid from an online bidder or from a live bidder via the auctioneer, must forward that bid onto the Bid Streamer Service and the Bidder Tracker Service. This could be done using asynchronous point-to-point (p2p) messaging, asynchronous publish-and-subscribe (pub/sub) messaging, or REST via the Online Auction API Layer.

DECISION
We will use asynchronous pub/sub messaging between the Bid Capture Service, Bid Streamer Service, and the Bidder Tracker Service.

The Bid Capture Service does not need any information back from the Bid Streamer Service or Bidder Tracker Service.

The Bid Streamer Service must receive bids in the exact order they were accepted by the Bid Capture Service. Using messaging and queues automatically guarantees the bid order for the stream.

Using async pub/sub messaging will increase the performance of the bidding process and allow for extensibility of bidding information.

CONSEQUENCES
We will require clustering and high availability of the message queues.

Internal bid events will be bypassing security checks done in the API layer.

UPDATE: Upon review at the April 14th, 2020 ARB meeting, the ARB decided that this was an acceptable trade-off and no additional security checks would be needed for bid events between these services.

COMPLIANCE
We will use periodic manual code and design reviews to ensure that asynchronous pub/sub messaging is being used between the Bid Capture Service, Bid Streamer Service, and the Bidder Tracker Service.

NOTES
Author: Subashini Nadella
Approved By: ARB Meeting Members, 14 APRIL 2020
Last Updated: 15 APRIL 2020 by Subashini Nadella

Figure 19-5. ADR 76. Asynchronous Pub/Sub Messaging Between Bidding Services