Chapter 7. The Scope of Architectural Characteristics

Software architects' thinking must evolve with our ecosystem, and nowhere is that more apparent than in scoping architectural characteristics. Many of the outdated frameworks for determining architectural characteristics had a fatal flaw: assuming one set of architectural characteristics for the entire system. While that is sometimes still true, many modern architectures, like microservices, contain different architectural characteristics at the service and system levels.

The scope of architectural characteristics is a useful measure for architects, especially in determining the most appropriate architecture style to use as a starting point for implementation. When we were writing our book *Building Evolutionary Architectures*, we needed a technique to measure the structural evolvability of particular architecture styles. None of the existing measures offered the correct level of detail. The section "Structural Measures" discusses a variety of code-level metrics that allow architects to analyze structural aspects of an architecture—but none of these metrics reflects scope. They reveal low-level details about the code, but can't evaluate dependent components outside the code base (such as databases) that affect many architectural characteristics, especially operational ones. No matter how much effort an architect puts into designing a code base to be performant or elastic, if the system's database doesn't match those characteristics, their effort won't be successful.

Failing to find a good measure of scope, we developed one. We call it *architecture quantum*.

Architectural Quanta and Granularity

Component-level coupling isn't the only thing that binds software together. Many business concepts bind parts of the system together semantically, creating *functional cohesion*. To design, analyze, and evolve software successfully, architects and developers must consider all the coupling points that could break.

Plurals for Latin-Derived Technical Terms

Quantum originates in Latin, which means that the plural ends with an *a*. If you have more than one quantum, you have *quanta*—like the more common *data*, also derived from Latin. Architects rarely discuss a single datum. The *Chicago Manual of Style* usage guide notes: "Though originally this word was a plural of *datum*, it is now commonly treated as a mass noun and coupled with a singular verb" (18th edition, 2024, p. 336).

Many science-literate architects know of the concept of a quantum from physics, where it refers to the smallest possible amount of something—usually energy. The word *quantum* derives from Latin, meaning "how great" or "how much." In

general usage, it tends to mean “small, unbreakable thing,” which is how we define an *architecture quantum*. Another common informal definition of *architecture quantum* is “the smallest part of the system that runs independently.” For example, microservices often form architecture quanta, which exemplifies this definition: a service can run independently within the architecture, including its own data and other dependencies.

An *architecture quantum* establishes the scope for a set of architectural characteristics. It features:

- Independent deployment from other parts of the architecture
- High functional cohesion
- Low external implementation static coupling
- Synchronous communication with other quanta

This definition contains several parts. Let’s break them down:

Establishes the scope for a set of architectural characteristics

> Architects use the architecture quantum as a boundary delineating a set of architectural characteristics, particularly operational ones. Because it is independently deployable and has high functional cohesion (discussed next), the architecture quantum provides a useful measure of architecture modularity.

Independently deployable

> An architecture quantum includes all the necessary components to function independently from other parts of the architecture. For example, if an application uses a database, that database is part of the quantum, because the system won’t function without it. This requirement means that virtually all legacy systems that are deployed using a single database, by definition, form a quantum of one. However, in the microservices architecture style, each service includes its own database (part of the *bounded context* driving philosophy, described in detail in Chapter 18). This creates multiple quanta within that architecture, because each service has its own architectural characteristics scope.

High functional cohesion

> *Cohesion* in component design refers to how unified the contained code is in its purpose. For example, a `Customer` component, with properties and methods all pertaining to a *Customer* entity, exhibits high cohesion. A `Utility` component with a random collection of miscellaneous methods would not. High functional cohesion implies that an architecture quantum does something purposeful. This distinction matters little in traditional monolithic applications with a single database, where the cohesion is essentially the entire system. However, in distributed architectures like event-driven or microservices, architects are more likely to design each service to match a single workflow (a *bounded context*, as described in “Domain-Driven Design’s Bounded Context”), so that service would exhibit high functional cohesion.

Domain-Driven Design's Bounded Context

Eric Evans's book *Domain-Driven Design* (Addison-Wesley Professional, 2003) has deeply influenced modern architectural thinking. *Domain-driven design (DDD)* is a modeling technique that allows architects to decompose complex problem domains in an organized way. DDD defines a *bounded context*, where everything related to a portion of the domain is visible internally but opaque to other bounded contexts.

Before DDD, architects sought to reuse code holistically across common entities within the organization. But they found that creating common shared artifacts causes a host of problems, such as tight coupling, more difficult coordination, and increased complexity. The bounded-context concept recognizes that each entity works best within a localized context. Thus, instead of creating a unified `Customer` class across the entire organization, each problem domain can create its own `Customer` class and reconcile the differences at communication points with other domains.

To clarify the next parts of the definition, we need to make some finer distinctions about the types of coupling:

Semantic coupling

> *Semantic coupling* describes the natural coupling of the problem for which an architect is building a solution. For example, an order-processing application's inherent coupling includes things like inventory, catalogs, shopping carts, customers, and sales. The nature of the problem that motivates building a software solution defines this coupling. Architects have few techniques that prevent changes to the domain from rippling out through the system: a change to the domain (and therefore the semantics) means we're changing the requirements for the system. While architects can adapt to that, no magical architecture pattern prevents changing the core problem from affecting the architecture.

Implementation coupling

> *Implementation coupling* describes how an architect and team decide to implement particular dependencies. In an order-processing application, the team must consider a variety of constraints in setting domain boundaries. For example, should all the data reside in a single database, or should some of it be split apart for better scalability or availability? Should we build a monolith or a distributed architecture? The answers to these questions have little effect on the system's semantic coupling, but greatly affect architectural decisions.

Static coupling

> *Static coupling* refers to the "wiring" of an architecture—how services depend upon one another. Two services are part of the same architecture quantum if they both depend on the same coupling point. For example, let's say that two microservices, `Catalog` and `Shipping`, need to share address information, so they both create a dependency to a shared component. Because both services are coupled to that dependency, they are part of the same architecture quantum.

> Here's an easy way to think about coupling in software architecture: two things are coupled if changing one might break the other. Static coupling defines the scope dependencies in an architecture. For example, if several services use the same relational database, they are part of the same quantum.

Dynamic coupling

> *Dynamic coupling* describes the forces involved when architecture quanta must communicate with each other. For example, when two services are running, they must communicate to form workflows and perform tasks within the system. Architects must consider the trade-offs when communicating between services in a distributed architecture, which is discussed in detail in Chapter 15.

With these definitions in hand, we can complete our *architecture quantum* definition:

Low external implementation static coupling

> The level of implementation coupling between architecture quanta should be low. This characteristic is also derived from the DDD philosophy of low coupling between bounded contexts. Quanta form the operational building blocks of architecture that sit one level of abstraction higher than components. They often overlap with service boundaries. This goal reflects architects' general preference for loose coupling between different parts of the architecture.

In general, tight coupling is desirable when high cohesion is required, such as within a service or subsystem. We call an architecture "brittle" when a single implementation change can cause unexpected rippling side effects that break many other (ostensibly unrelated) things. The broader the scope of the system's coupling, the more loose coupling helps to make the architecture less brittle. For example, an architect might rename a field in a service call from `State` to `StateCode`, thinking it will only affect one caller, only to discover that many other dependencies have just broken unexpectedly.

Tip

Higher coupling is allowed for narrower scopes; the broader the scope, the looser the coupling should be.

Synchronous Communication

Communication refers to dynamic coupling—when architecture quanta call each other, which is common in distributed architectures. This particularly concerns the family of operational architectural characteristics described in "Operational Architectural Characteristics", because they often determine important timing and blocking in distributed architectures.

For example, consider a microservices architecture with a `Payment` service and an `Auction` service. When an auction ends, the `Auction` service sends payment information *synchronously* to the `Payment` service. However, let's say that the `Payment` service can only handle one payment every 500 ms. What happens when a large number of auctions end at once? The two services have different operational architectural characteristics. The first call will work, but then calls will start

failing because the `Payment` service can't handle the number of requests—it isn't as scalable as the `Auction` service.

We call out synchronous communication here because asynchronous communication has less potential impact. For example, if the `Auction` service calls the `Payment` service asynchronously, using a message queue, the queue can act as a buffer to allow the two systems to operate. Of course, if the `Auction` service continually sends more messages than `Payment` can handle, it will eventually overflow the message queue. However, when the flood of messages comes in bursts, the queue can hold on to pending messages until the receiver is ready. Synchronous communication is unforgiving in distributed architectures, especially if parts of the architecture have different architectural characteristics. Chapter 15 thoroughly covers the types of communication in event-driven architectures.

The concept of architecture quantum provides a new way to think about scope. In modern systems, architects define architectural characteristics at the quantum level rather than the system level. This provides important information when analyzing a new problem domain.

The Impact of Scoping

Architects can use the scope of architectural characteristics to help determine appropriate service boundaries illustrated overall in the decision tree in Figure 7-1 and further detailed in the steps.

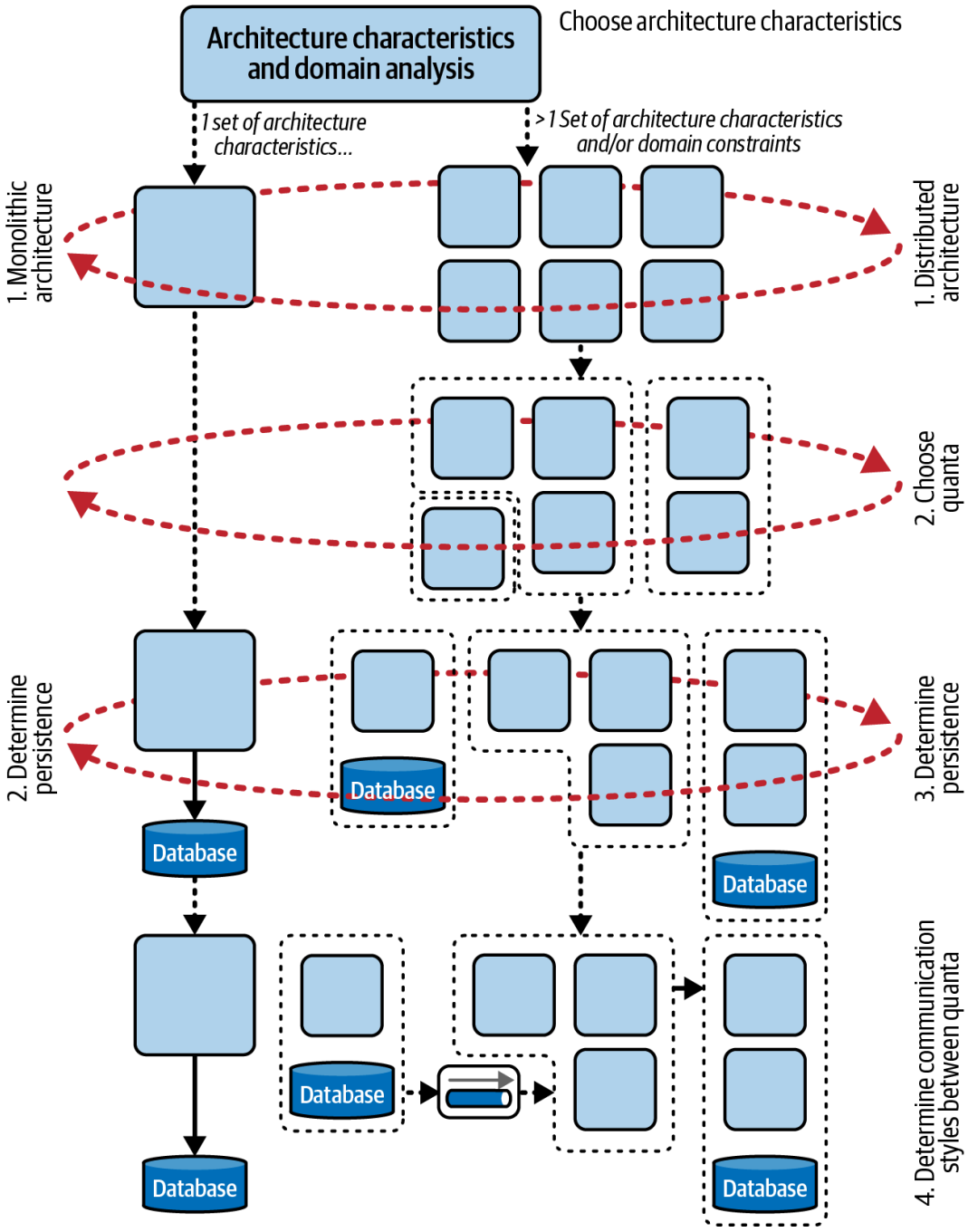

Figure 7-1. A decision tree that uses architectural characteristics scope to help determine architectural style

Scoping and Architectural Style

Determining the quantum boundaries of the problem domain helps in choosing an architectural style: is a monolithic architecture or distributed architecture most suitable? Consider the first part of the decision chart shown in Figure 7-2.

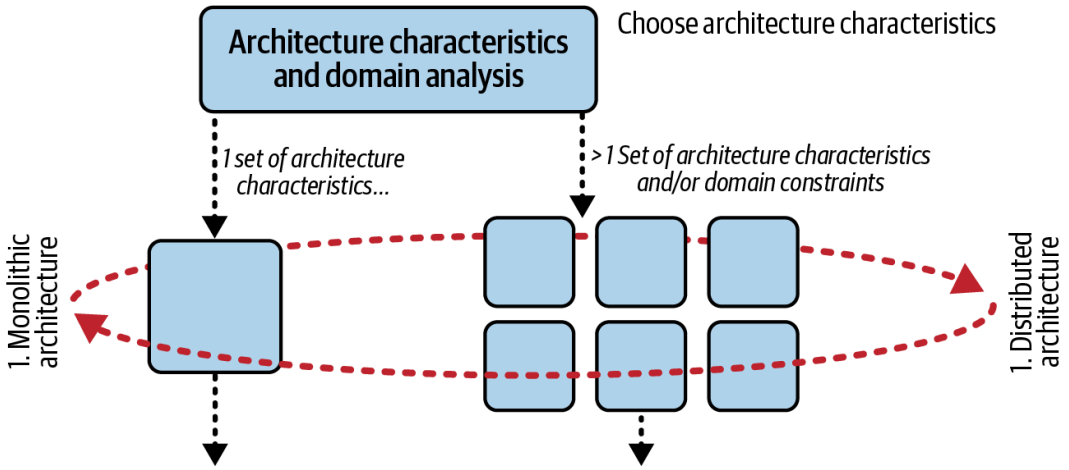

Figure 7-2. Choosing an appropriate architectural style based on architectural characteristics

As we discussed in Chapter 4, architects must analyze architectural characteristics and the domain to determine the most appropriate architectural style. Part of this analysis concerns whether the solution needs multiple groups of architectural characteristics. In step one in Figure 7-2, the architect determines if the system can succeed with a single set of architectural characteristics, or if more than one group is required (see "Kata: Going Green" for an example).

If the architect determines that a single set of architectural characteristics will suffice, they can choose a monolithic architecture, reducing the number of subsequent choices. If they choose a distributed architecture, the next step is to determine its quantum boundaries, as shown in Figure 7-3.

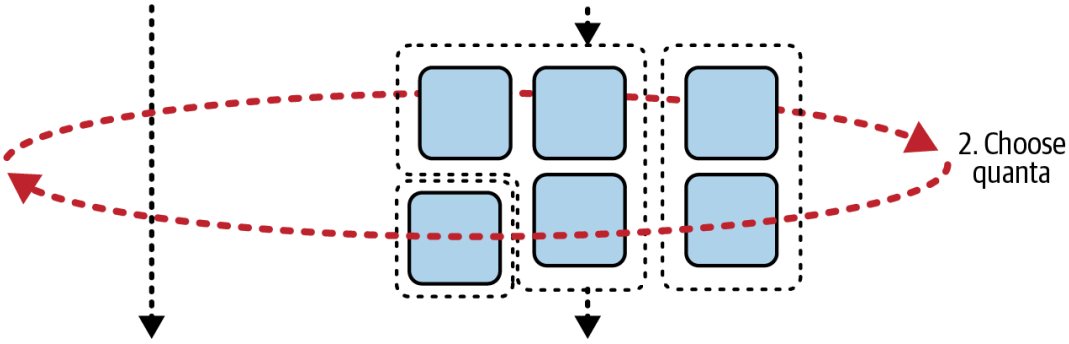

Figure 7-3. In distributed architectures, architects must choose the appropriate quantum boundaries

We provide some guidelines for determining granularity in Chapter 18. The next step is choosing a persistence mechanism, which concerns both architecture families, shown in Figure 7-4.

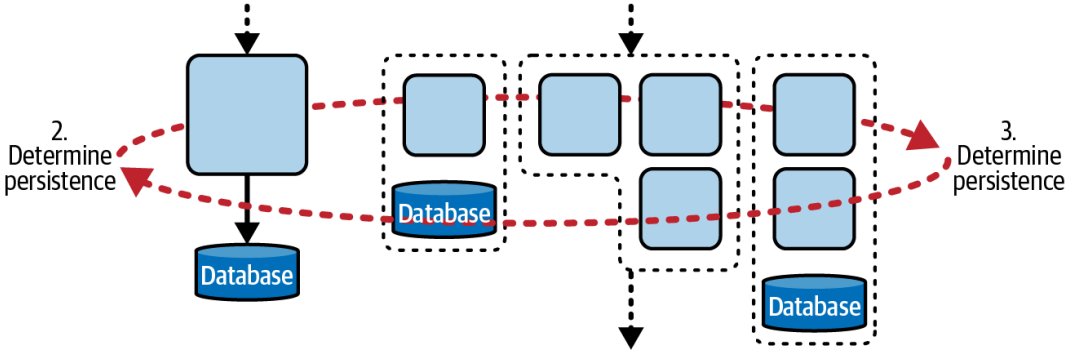

Figure 7-4. Both architecture styles generally require persistence of some kind

For monolithic architectures, a single monolithic database is generally suitable. The architecture and database are developed and deployed together, in lockstep. This completes the process—the architect can move on to choosing the most appropriate monolithic style.

Distributed architectures, on the other hand, can use a single database (common in event-driven architectures) or partition the data along the lines of the service granularity (as in microservices architectures). One further step remains: determining what type of communication to use between quanta, synchronous or asynchronous. (We discuss this question in detail in Chapter 15.) Remember that, in some systems, choosing synchronous communication can change the quantum boundaries established through static coupling; the two types of coupling interact frequently.

Kata: Going Green

Let's try analyzing a problem by using architecture quantum as a scope for architectural characteristics. The problem, called *Going Green*, is illustrated in Figure 7-5.

You are working with Going Green (GG), a business that recycles and resells old electronics, such as cell phones. Its system uses both public kiosks and a website, all running the same system. A user can upload their device's model number and condition, and GG will bid for it. If the user accepts the bid, they can deposit it in the kiosk or, if they go through the website, GG will send them a box to mail it in. Upon receiving the device, GG assesses it and sends the user their payment. GG then estimates the value of the device and either recycles or resells it. Its system also produces reports and other analytics.

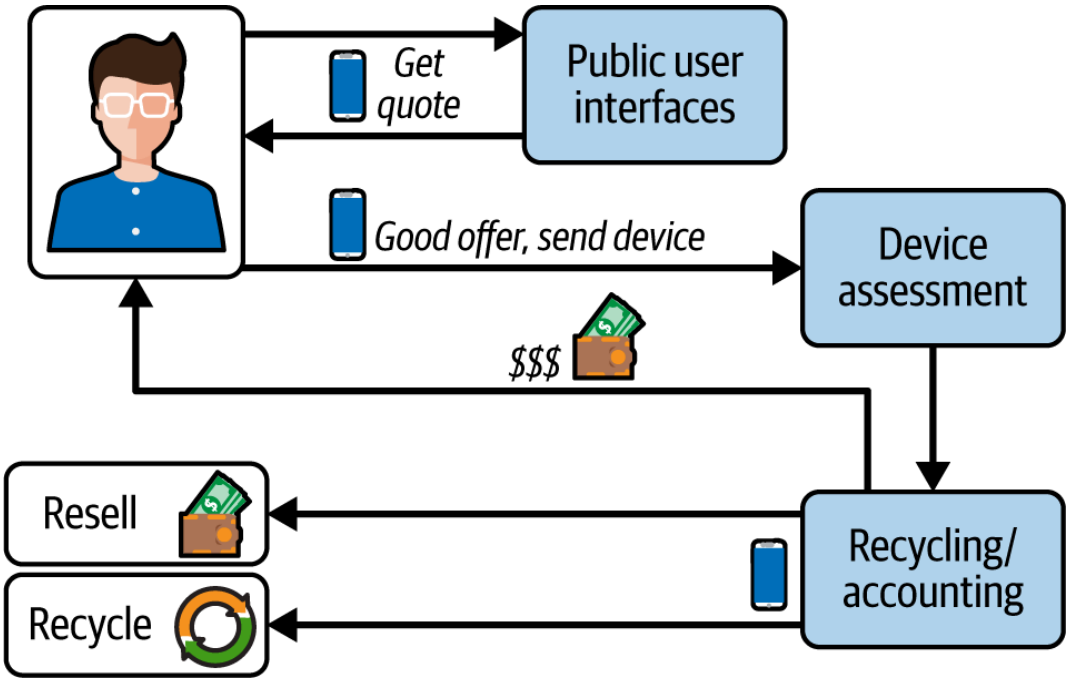

Figure 7-5. Requirements overview for Going Green

As you perform your architectural characteristics analysis, you notice three distinct clusters forming, as shown in Figure 7-6.

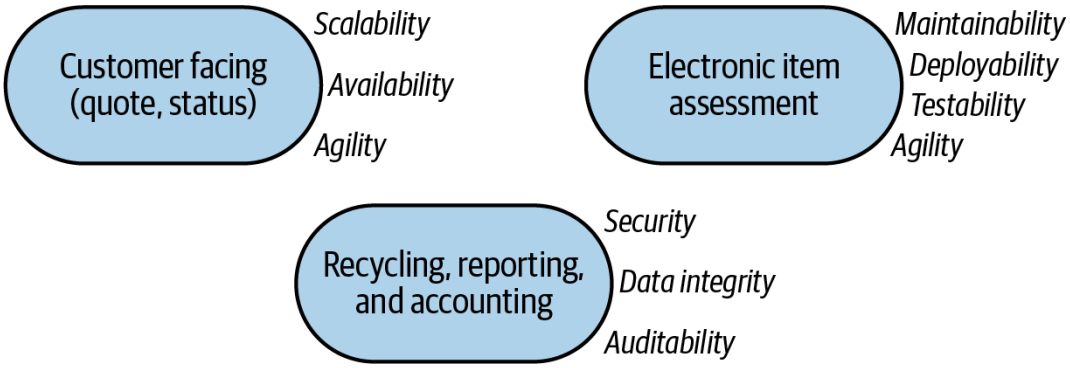

Figure 7-6. The GG architectural characteristics analysis yields three clusters of capabilities

The public-facing parts of the application need *scalability*, *availability*, and *agility*; the back-office functions need *security*, *data integrity*, and *auditability*; and the assessment part needs *maintainability*, *deployability*, and *testability* (which make up the composite architectural characteristic of *agility*). Why does the assessment part need a separate set of architectural characteristics? This is a good example of how business drivers intersect with architectural concerns. GG's business model relies on reselling the highest-value used electronics, and new models come out in a constant stream. The faster they can update their device assessments, the more they can get newer (and therefore more valuable) devices to resell.

Could you design a system that meets all these criteria: scalability, availability, security, data integrity, auditability, maintainability, deployability, and testability? It's possible…but it would be difficult. These architectural characteristics often counteract one another: for instance, achieving fast deployability is more difficult when you're also prioritizing back-office concerns like auditability. And the UI requires a vastly different level of scalability than other parts of the system.

Instead of trying to do it all, you can use the clusters of architectural characteristics as your guide to separating quanta. In Figure 7-7, the dotted lines illustrate each architectural quantum. Using characteristic scope as a guide for service granularity is a good first step in determining the most beneficial set of trade-offs. (We explore this topic further in Chapter 18.)

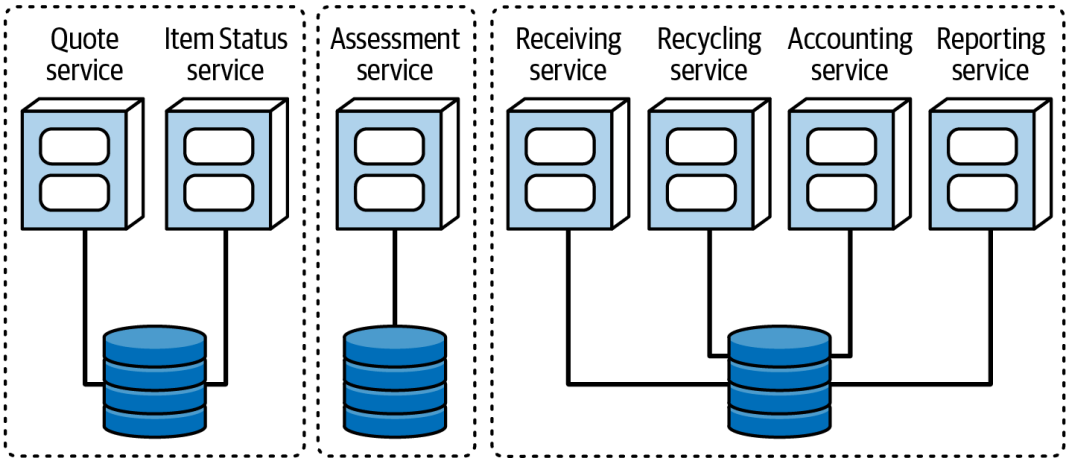

Figure 7-7. This GG architecture captures each set of architectural characteristics as an architectural boundary

Scoping and the Cloud

Cloud-based resources complicate the picture, because they encapsulate so many of the operational architectural characteristics of a system. Architects must

consider at least two scenarios when using cloud-based resources for all or part of an application, depending on the deployment model:

Using the cloud to host containers

> Many development teams use the cloud as an alternate operations center, running (and orchestrating) containers for servers. In such situations, architects must consider the architectural characteristics of the containers and the constraints introduced by the orchestration tool (for example, Kubernetes).

Using cloud-provider resources as system components

> Another flavor of a cloud-based system pieces applications together using the cloud provider's building blocks, such as triggered functions, databases, and so on. In this case, architects must look at the capabilities the provider advertises (and hopefully maintains) for insights on how to build suitable capabilities for that particular context.

Many of the capabilities we know today as cloud providers' configuration settings, such as elasticity, were hard-won by the previous generation of architects working on physical systems. Their major concerns have become easier—but we have our own modern set of trade-offs, such as provider availability and heightened security concerns. The details of software architecture change a lot, but the job of analyzing trade-offs remains constant.