Chapter 19. 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 architecture style you should use—but we can't. 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 a whole process of analysis and thought about trade-offs for architecture characteristics, domain considerations, strategic goals, and a host of other things. That's why, as we said back in Chapter 2, it depends.

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

Shifting "Fashion" in Architecture

The software industry's preferences in architecture styles shift over time, driven by a number of factors. These include:

Observations from the past

> New architecture styles generally arise from observations about past experiences, especially observations about pain points. Architects' experiences working with systems—which are often what lead us to become architects in the first place—influence our thoughts about future systems. New architecture designs often reflect fixes for the specific deficiencies encountered in past architecture styles. For example, after building architectures that centered on code reuse, architects realized the negative trade-offs and seriously rethought the implications of reusing code.

Changes in the ecosystem

> In the software development ecosystem, everything changes all the time. This constant change is so chaotic that it's impossible to even predict what type of change will come next. Not too many years ago, no one knew what Kubernetes was, and now it is a daily feature of many developers' lives. In a few more years, Kubernetes may be replaced with some other tool that hasn't been written yet.

New capabilities

> Architects must keep a keen eye open not only for new tools but for new paradigms. When new capabilities arise, architecture can shift to an entirely new paradigm, rather than merely replacing one tool with another. For example, few anticipated the tectonic shift in the software development world caused by the advent of containers such as Docker. While that was an evolutionary step, it had an astounding impact on architects, tools, engineering practices, and so much more. Constant change in the ecosystem also regularly delivers new tools and capabilities. Even something that looks like a new one-of-something-we-already-have could include nuances

that make it a game changer. New features don’t even have to rock the entire development world: a minor change that aligns exactly with an architect’s goals can change everything.

Acceleration

Not only does the ecosystem constantly change, but change keeps getting faster and more pervasive. New tools create new engineering practices, which lead to new designs and capabilities, keeping software architects in a constant state of flux. The rise and influence of generative AI is an outstanding example of this constant evolution and corresponding unpredictability.

Domain changes

The domains for which we write software constantly shift and change, as businesses evolve or merge with other companies.

Technology changes

Organizations try to keep up with at least some technological changes, especially those with obvious bottom-line benefits.

External factors

Many external factors only peripherally associated with software development can drive change within an organization. For example, architects and developers might be perfectly happy with a particular tool, but if its licensing cost becomes prohibitive, the business might be forced to migrate to another option.

Architects should understand current industry trends so they can make intelligent decisions about which trends to follow and when to make exceptions, regardless of how closely their organization follows current architecture fashion.

Decision Criteria

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

Only approach choosing an architectural style when you have sufficient knowledge about the following factors:

The domain

Understand as many important aspects of the business domain as you can, especially those that affect operational architecture characteristics. Architects don’t have to be subject matter experts, but should at least have a good general understanding of the major aspects of the domain under design. Other specialists, such as business analysts, can help you fill any gaps in your domain knowledge.

Architecture characteristics that impact structural decisions

Identify and elucidate which architecture characteristics are needed to support the domain and other external factors by conducting an architectural

characteristics analysis, one of the core activities in choosing a style.

It's possible to implement any of the generic architecture styles in pretty much any problem domain—*generic* implies that they are general-purpose, after all. The exceptions are domains that require special operational architectural characteristics, like a highly scalable auction site. However, in most cases, the real differences between architectural styles concern not the domain, but how well each style supports various architectural characteristics.

You may have noticed that the star charts we've used in Part II of this book to compare each architecture style focus on architectural *characteristics*, not domains. This reflects the importance of understanding architectural characteristics when choosing a style.

Data architecture

Architects and data developers must collaborate on databases, schemas, and other data-related concerns. Data architecture is a specialization in its own right, and we don't cover it much in this book, outside of style-specific considerations. However, you need to understand the impact a given data design might have on your architectural design, particularly if the new system must interact with an older data architecture or one that's already in use.

Cloud deployments

Using the cloud as an architectural destination is the latest in a long line of fundamental shifts in where computation and data reside. The trade-offs involved in designing an application to run on-premises are quite different from those involved in designing one for the cloud. It's important to know how much data the application will need to store and how much data can move around (which can incur significant costs), among a host of other concerns.

The cloud is a great example of how sophisticated capabilities become commodities over time. A decade ago, building a highly elastic and scalable on-prem system required esoteric skills and was seen as almost magical. Now, architects can achieve the same results just by changing their cloud provider's configuration parameters.

Organizational factors

Many external factors influence design. For example, the cost of a particular cloud vendor may prevent a business from adopting what would otherwise be the ideal design. Likewise, knowing that the company plans to engage in mergers and acquisitions might encourage an architect to gravitate toward open solutions and integration architectures.

Knowledge of process, teams, and operational concerns

Many specific project factors influence architects' designs: the software development process, an architect's 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 (such as microservices) will present difficulties.

Domain/architecture isomorphism

> *Architecture isomorphism* is a fancy term for the generic "shape" of an architecture—in other words, the way its components depend on each other within the overall topology. The word *isomorphism* means "a map that preserves sets and relations among elements"; it derives from the Greek *isos*, meaning "equal," and *morph*, meaning "form" or "shape."

Architects think about the generic shape of the architecture when considering how suitable it is. For example, consider the differences between the two architecture isomorphism diagrams for the layered and modular monolith architectural styles shown in Figure 19-1. The internal shape of each architecture is apparent: separation by layers versus by domains.

Figure 19-1. Comparison between the isomorphic representations of layered and modular monoliths

The difference between monolithic and distributed is similarly clear from isomorphic drawings, as illustrated in Figure 19-2. Here, the distribution of core components makes the macro-level structure of the architecture clear.

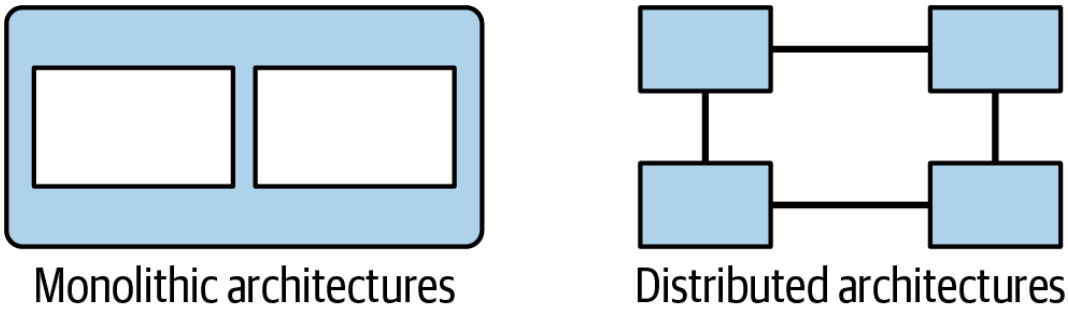

Figure 19-2. Comparing isomorphic representations of monolithic and distributed architecture styles

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

Similarly, some problem domains may be particularly ill-suited for certain architecture styles. Highly scalable systems struggle with large monolithic designs, because it's difficult for a highly coupled code base to support a large number of concurrent users. A problem domain that includes a huge amount of semantic coupling would match poorly with a highly decoupled, distributed architecture: for instance, an insurance application consisting of multipage forms, each based on the context of previous pages, is a highly coupled problem. It would be difficult to model in a decoupled architecture like microservices. An intentionally coupled architecture like service-based architecture would suit this problem better.

Taking all these things into account, you must make several determinations in choosing an architecture style:

Monolith versus distributed?

> Will a single set of architecture characteristics suffice for the design, or do different parts of the system need differing architecture characteristics? A single set implies that a monolith would be suitable (although other factors may suggest a distributed architecture); different sets of architecture characteristics imply a distributed architecture. The concept of architecture quantum (Chapter 7) is useful in making this determination.

Where should data live?

> If the architecture is monolithic, architects commonly assume it will use a single relational database, or perhaps a few of them. In a distributed architecture, you must decide which services should persist data, which also implies thinking about how data will flow through the architecture to build workflows. Consider both structure and behavior when designing architecture, and don't be afraid to iterate on your design to find better combinations.

Should services communicate synchronously or asynchronously?

> Once you've determined where data should live, the next design consideration is communication between services—should it be synchronous or asynchronous? Synchronous communication is often more convenient, but it can mean trading off on scalability, reliability, and other desirable characteristics. Asynchronous communication can provide unique benefits in terms of performance and scale, but also plenty of headaches around data synchronization, deadlocks, race conditions, debugging, and so on. (We cover many of these issues in Chapter 15.)

Because synchronous communication presents fewer design, implementation, and debugging challenges, we recommend defaulting to synchronous when possible, and using asynchronous communication only when necessary.

Tip

Use synchronous communication by default, asynchronous when necessary.

The output of this design process is an *architecture topology* that encompasses the chosen architecture style (and any hybridizations), Architectural Decision Records (ADRs) about the parts of the design that require the most effort, and architecture fitness functions to protect important principles and operational architecture characteristics.

Monolith Case Study: Silicon Sandwiches

In the Silicon Sandwiches architecture kata in Chapter 5, after our architecture characteristics analysis, we determined that a single quantum was sufficient to implement the system. Since we were looking at a simple application without a huge budget, the simplicity of a monolith was appealing.

However, we created two different component designs for Silicon Sandwiches: one domain partitioned, and another technically partitioned—you might want to flip back to Chapter 5 if you want a refresher. In this chapter, we return to those simple solutions to create designs for each option and discuss their trade-offs. We'll begin with a monolithic architecture.

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 19-3.

This is a monolith with a single relational database, implemented with a single web-based UI (with careful design considerations for mobile devices) to keep overall cost down. Each of the domains we identified appears as a component. If time and resources are sufficient, consider separating the tables and other database assets in the same way as the domain components, which would make it much easier to migrate this architecture to a distributed architecture if future requirements warrant it.

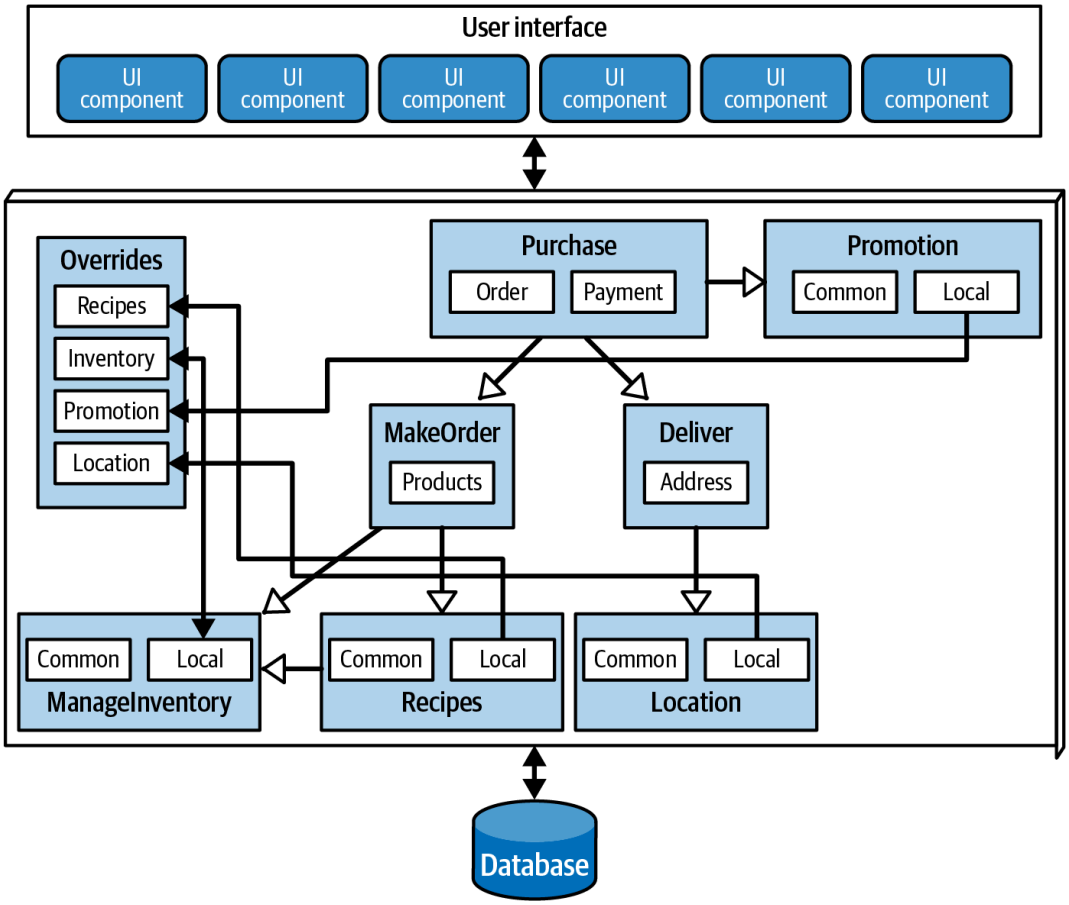

Figure 19-3. A modular monolith implementation of Silicon Sandwiches

Because the architecture style itself doesn't inherently handle customization, that feature should become part of the domain design. In this case, the architect designs an `Override` endpoint, where developers can upload individual customizations. Correspondingly, they must ensure that every domain component references the `Override` component for each customizable characteristic. (Checking this would be a perfect job for an architectural fitness function.)

Microkernel

One of the architecture characteristics we identified in Silicon Sandwiches was customizability. Figure 19-4 uses domain/architecture isomorphism to show how this could be implemented using a microkernel architecture.

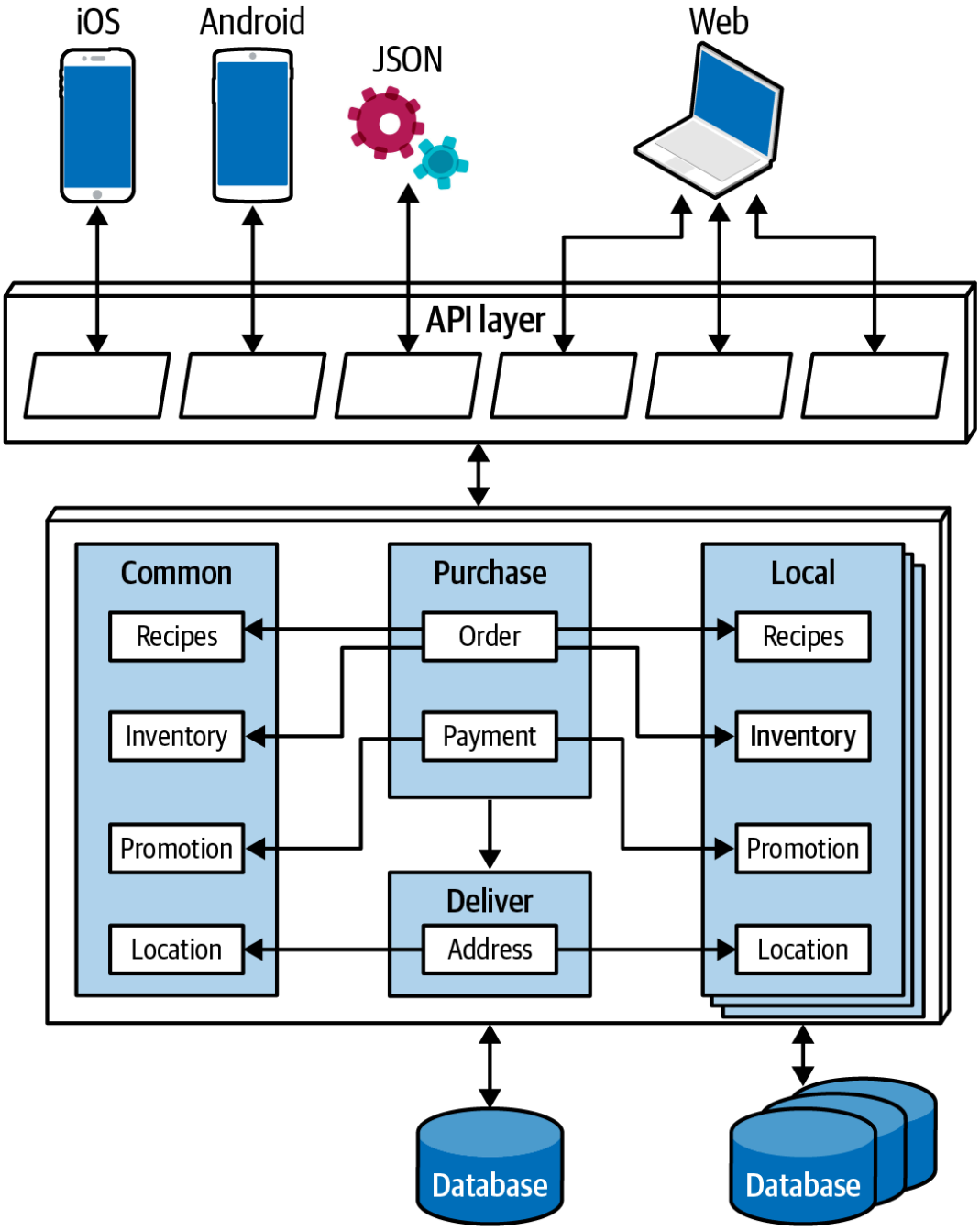

Figure 19-4. A microkernel implementation of Silicon Sandwiches

In Figure 19-4, the core system consists of the domain components and a single relational database. As in the modular monolith design, careful synchronization between the domains and the data design will facilitate migrating the core to a distributed architecture in the future. Each customization appears in a plug-in; the common ones appear in a single set of plug-ins (with a corresponding database) and a series of local ones, each with its own data. Because none of the plug-ins need to be coupled to the other plug-ins, they can each maintain their data and remain decoupled.

The other unique element of this design is that it utilizes the Backends for Frontends (BFF) pattern, making the API layer a thin microkernel adapter in

addition to the core architecture. The API layer supplies general information from the backend, which the BFF adapters translate into a suitable format for the frontend device. For example, the BFF for iOS takes the generic backend output and customizes the data format, pagination, latency, and other factors to fit what the iOS native application expects. Building each BFF adapter allows for the richest possible user interfaces and makes it possible to expand the architecture to support other devices in the future—one of the benefits of the microkernel style.

Communication within either of these two Silicon Sandwich architectures can be synchronous, since 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 from Chapter 8 presents some more interesting architectural challenges. Based on the component analysis in "Case Study: Going, Going, Gone—Discovering Components", we know that different parts of this architecture need different characteristics. For example, the need for availability and scalability will differ between roles like auctioneer and bidder.

The system requirements for GGG also explicitly state some ambitious expectations for scale, elasticity, performance, and other tricky operational architecture characteristics. The architecture pattern should allow for a high degree of customization at a fine-grained level. Of the candidate distributed architectures, low-level event-driven architecture and microservices are the two that best match most of the required architecture characteristics. Of the two, microservices is better at supporting variation among operational architecture characteristics. (Purely event-driven architectures typically separate pieces not by their architecture characteristics, but by whether they use orchestrated or choreographed communication.)

Achieving the stated performance goal would be a challenge in microservices, but the best way to address any weak point of an architecture is by designing to accommodate it. For example, while microservices by its nature offers a high degree of scalability, it often develops specific performance issues caused by too much orchestration or too aggressive data separation, among other issues.

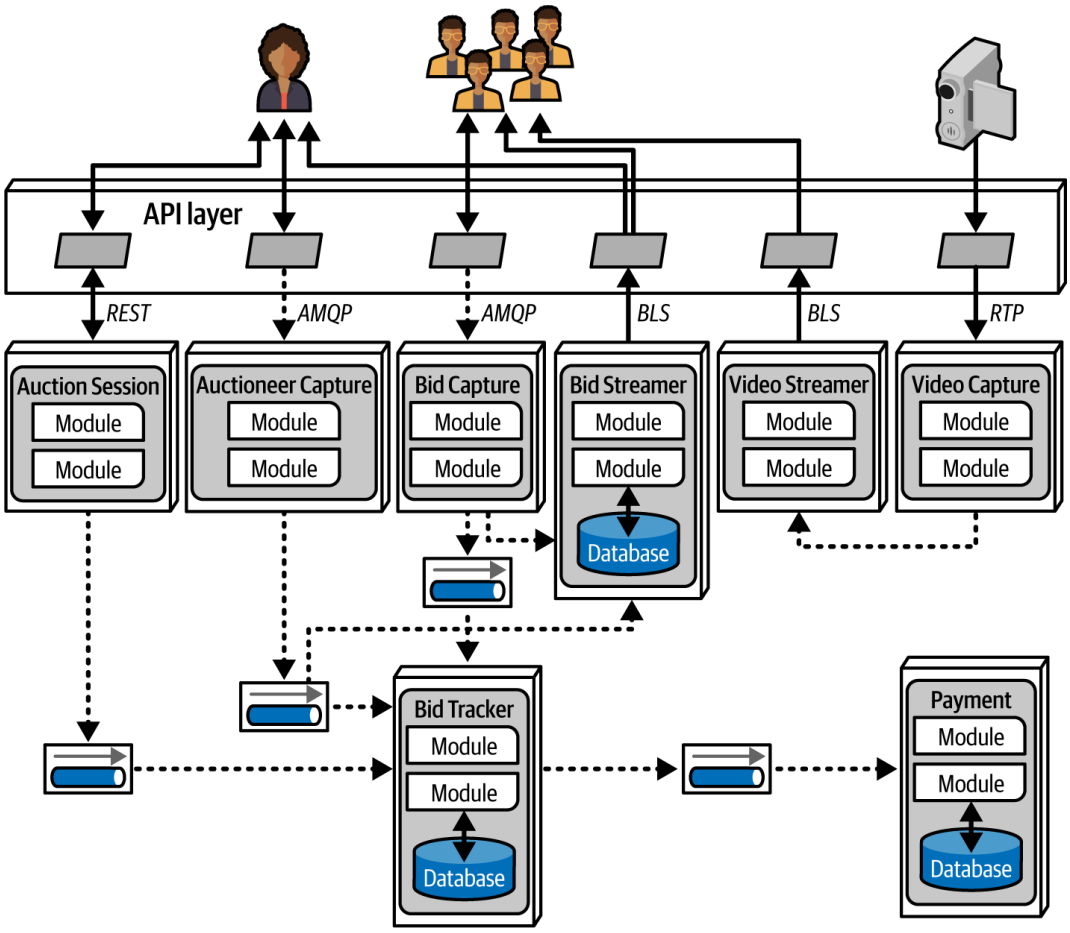

Figure 19-5. A microservices implementation of Going, Going, Gone

In Figure 19-5, each identified component becomes a service in the architecture, matching components with service granularity. GGG has three distinct user interfaces:

Bidder

The numerous bidders for the online auction.

Auctioneer

One per auction.

Streamer

A service responsible for streaming video and bids to the bidders. This is a read-only stream, which allows for optimizations that wouldn't be available if updates were necessary.

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

`Bid Capture`

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.

`Bid Streamer`

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

`Bid Tracker`

: Tracks bids from both `Auctioneer Capture` and `Bid Capture`, unifying these two information streams and ordering the bids in as close to real time as possible. Both inbound connections to this service are asynchronous, allowing the developers to use message queues as buffers to handle very different message flow rates.

`Auctioneer Capture`

: Captures bids for the auctioneer. The analysis in "Case Study: Going, Going, Gone—Discovering Components" showed that the architecture characteristics of `Bid Capture` and `Auctioneer Capture` are quite different, leading us to separate them.

`Auction Session`

: Manages the workflow of individual auctions.

`Payment`

: Third-party payment provider that handles payment information after `Auction Session` completes the auction.

`Video Capture`

: Captures the video stream of the live auction.

`Video Streamer`

: Streams the auction video to online bidders.

We were careful to identify both synchronous and asynchronous communication styles in this architecture. The choice of asynchronous communication was primarily to accommodate operational architecture characteristics varying 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, using synchronous communication between the services would cause timeouts and other reliability headaches. Message queues add reliability to a critical but fragile part of the architecture.

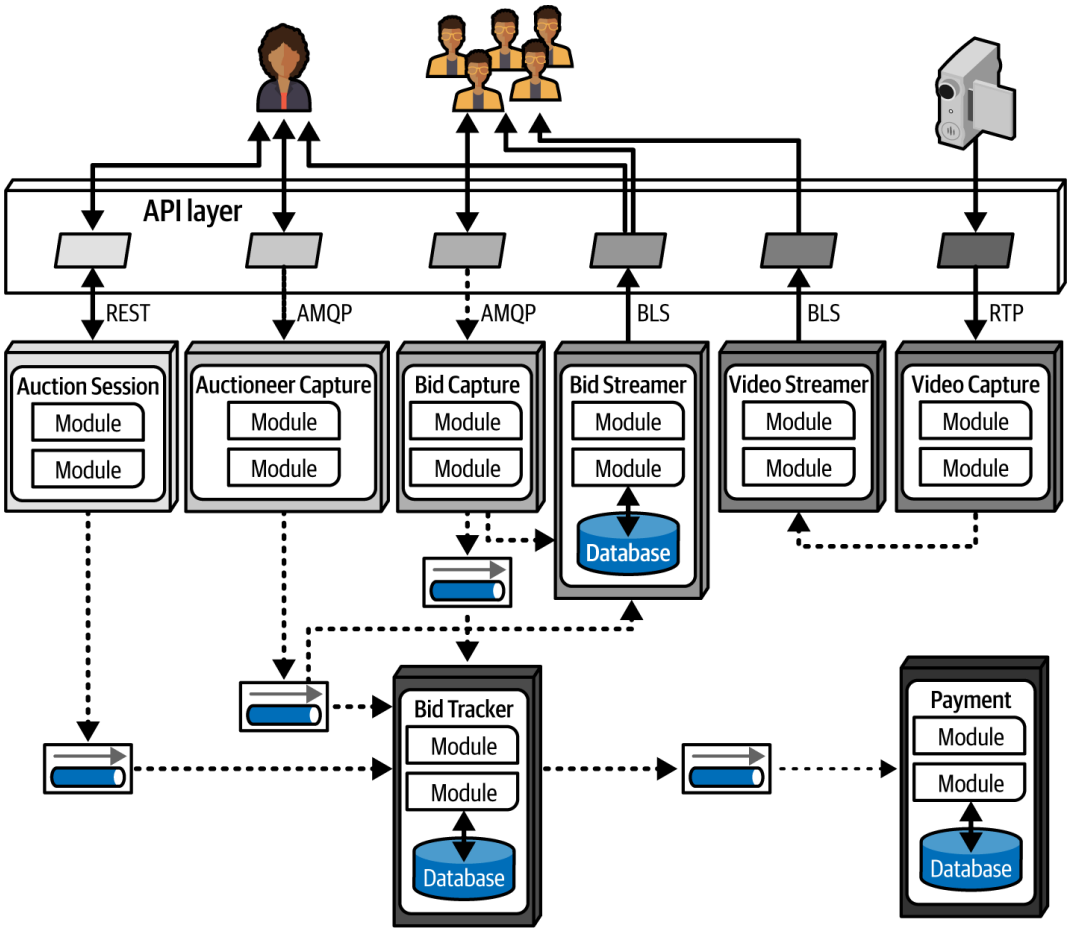

Figure 19-6. The quantum boundaries for GGG

In the final analysis, this design resolved to five quanta, identified in Figure 19-6, roughly corresponding to the services: `Payment`, `Auctioneer`, `Bidder`, `Bidder Streams`, and `Bid Tracker`. Multiple instances are indicated in the diagram by stacks of containers. Using quantum analysis at the component-design stage made it easier to 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 using events and messages intelligently, allows this architecture to get the most out of a generic architecture pattern while still building a foundation for future development and expansion.