Chapter 20. Architectural Patterns

We distinguish between architectural styles and architectural patterns in Chapter 9: to recap, *styles* are named topologies that architects distinguish by differences in topologies, physical architectures, deployments, communication styles, and data topologies. Architecture *patterns*, inspired by the often-mentioned book *Design Patterns*, are contextualized solutions to problems.

It's important to distinguish between architecture patterns and "best practices" (discussed in more depth in Chapter 21). Calling something a "best practice" implies that the architect has a clear duty, anytime a particular situation arises, to utilize that practice. Calling it a *better* practice would at least brook some argument, but no—we call it the *best* practice, allowing architects to shut off their brains and always follow the same solution.

It's also important to distinguish between patterns and *solutions*. Many tools, frameworks, libraries, and other software-development artifacts encapsulate one or more patterns—depending on how they are implemented, with differing degrees of fidelity and intermingling with other patterns. Focus on identifying the most appropriate pattern first, then choose the most appropriate implementation for it.

We provide a smattering of representative patterns in this chapter to contrast and provide context for the styles we've introduced in Part II of this book. Our first example, an architecture reuse pattern, clarifies the difference between patterns and implementations.

Reuse

The split between domain coupling and operational coupling is a common architectural concern in distributed architectures such as microservices.

Separating Domain and Operational Coupling

One of the design goals of microservices architectures is a high degree of decoupling, often manifested in the advice: "duplication is preferable to coupling."

Let's say that two services need to pass information about customer profiles back and forth, but the architecture's domain-driven bounded context insists that implementation details should remain private to each service. A common solution is to give each service its own internal representations of entities such as `Profile`, then passing that information in loosely coupled ways, such as name-value pairs in JSON. This lets each service change its internal representation—including the technology stack—at will, without breaking the integration. Developers generally frown on duplicating code, which can cause issues with synchronization, semantic drift, and more, but some things are worse than duplication…and in microservices, that includes coupling.

Architects who design microservices generally resign themselves to the reality that sometimes they have to duplicate implementations to preserve decoupling. But what about capabilities that *benefit* from high coupling? Each service should have certain common operational capabilities, such as monitoring, logging, authentication and authorization, and circuit breakers. However, allowing each team to manage these dependencies often descends into chaos.

For example, consider a company trying to choose one standard monitoring solution for all of its services to make it easier to operationalize them. The architect decides to make each team responsible for implementing monitoring for its service: the `Payment` service team, the `Inventory` service team, and so on. But how can the operations team be sure that each team actually did this? Also, what about issues such as unified upgrades? If the standardized monitoring tool needs an upgrade across the organization, how should the teams all coordinate that?

Hexagonal architecture

In the *Hexagonal* architecture pattern, shown in Figure 20-1, the domain logic resides in the center of the hexagon, which is surrounded by ports and adapters connected to other parts of the ecosystem (in fact, this pattern is alternately known as the *Ports and Adapters* pattern). A visual representation of this pattern appears in Figure 20-1.

Figure 20-1. The Hexagonal architecture pattern

Sharp-eyed readers may notice that only four of the hexagon's six sides are used. This pattern's creator, Alistair Cockburn, initially drew it as a hexagon and called it the Hexagonal architecture pattern. He almost instantly regretted it, since the name Ports and Adapters is more descriptive, but it was too late. Too many architects thought "Hexagonal" sounded cool, so the name stuck.

Mistaking the pattern for the implementation is a common hazard, and this is a good example. The Hexagonal architecture has a potentially serious flaw when used to describe microservices, but only for those who understand the pattern's original intent. Hexagonal architecture predates modern microservices, but there are many similarities between them. There's also one significant difference: data fidelity. Hexagonal architecture treats the database as just another adapter that can be plugged in. It didn't include the data schema as business logic, because of the misperception (which was common at the time that the Hexagonal pattern name was coined) that the database was an entirely separate piece of machinery. Eric Evans had the insight to correct this mistake in his book *Domain-Driven Design* by recognizing that, regardless of where they reside, database schemas must change to reflect the system's business logic.

This makes the Hexagonal pattern a constant source of confusion among architects. Whenever someone uses it, are they describing the separation of operational and domain concerns, or are they referring to the literal pattern, which would isolate the data and therefore violate a core microservices design principle? Using the pattern name as shorthand for "separation of domain and operational concerns" is fine, as long as it isn't misleading in context. However, architects today have no need for this implementation. We now have a more suitable mechanism to implement the Hexagonal pattern: the *Service Mesh* pattern.

Service Mesh

Back in "Operational Reuse", we described the common architectural approach to separation of technical concerns from domain concerns: using the Sidecar and Service Mesh patterns.

The Sidecar pattern isn't just a way to decouple operational capabilities from domains—it's an orthogonal reuse pattern to address a specific kind of coupling (see "Orthogonal Coupling"). Often, architectural solutions require several types of coupling, such as our current example of domain versus operational coupling. An *Orthogonal Reuse* pattern presents a way to reuse some aspect represented by one or more concerns in the architecture that doesn't fit within the preferred hierarchical organization. For example, microservices architectures are organized around domains, but operational coupling requires cutting across those domains. A sidecar allows an architect to isolate those concerns in a cross-cutting, but consistent, layer through the architecture.

Orthogonal Coupling

In mathematics, two lines are *orthogonal* if they intersect at right angles, which also implies independence. In software architecture, two parts of an architecture may be *orthogonally coupled*: they may have two distinct purposes that must still intersect to form a complete solution. The obvious example from this chapter is an operational concern such as monitoring, which is necessary but independent from domain behavior, like catalog checkout. Recognizing orthogonal coupling allows architects to find the intersection points that cause the least entanglement between concerns.

While the Sidecar pattern offers a nice abstraction, it has trade-offs like all other architectural approaches, as shown in Table 20-1.

Table 20-1. Trade-offs for the Sidecar and Service Mesh patterns

Advantages	Disadvantages
Offers a consistent way to create isolated coupling	Must implement a sidecar per platform
Allows consistent infrastructure coordination	Sidecar component may grow large/complex
Ownership per team, centralized, or some combination	Implementation "drift" between independent teams

Both the Hexagonal and Service Mesh patterns illustrate ways to implement the reuse pattern to separate domain from operational concerns. The Hexagonal implementation is general purpose, while the Service Mesh is well suited to microservices and other distributed architectures. The key for architects is to identify the pattern first—separation—and then decide how best to implement it in their architecture.

Communication

Many architecture patterns, including communication patterns, come from event-driven architecture and apply to any distributed architecture that communicates via messages and/or events, such as discussed in Chapters 15 and 18. In fact, you’ve seen examples of communication in each of those chapters—we just haven’t identified them as particular patterns because architects commonly implement patterns without realizing it. Patterns are, after all, solutions to common problems.

Orchestration Versus Choreography

Consider two communication patterns you’ve already seen, in both “Mediated Event-Driven Architecture” and “Choreography and Orchestration”: choreography and orchestration, as summarized in Figure 20-2.

Figure 20-2. Orchestration and choreography in a microservices architecture

In each of the isomorphic workflows shown in Figure 20-2, four domain services (Services A through D) need to collaborate to form a workflow. In the orchestration case, there's also a separate service that acts as a coordinator for the workflow: the *orchestrator*.

While we've described this communication as both *orchestration* and *mediation*, the pattern remains the same. Architects benefit from being able to recognize patterns lurking inside implementations because their trade-offs become more apparent.

Notice that when we described the trade-offs for mediation and orchestration, we touched on many of the same issues. We'll summarize the advantages here:

Centralized workflow

As complexity goes up, architects benefit from utilizing a unified component for state, behavior, and boundary conditions.

Error handling

Error handling, a major part of many domain workflows, is assisted by having a state owner for the workflow.

Recoverability

Because an orchestrator monitors the state of the workflow, if one or more domain services suffers from a short-term outage, the architect can add logic to retry.

State management

Having an orchestrator makes the state of the workflow queryable, providing a place for other workflows and other transient states.

General disadvantages of orchestration include:

Responsiveness

All communication must go through the orchestrator, which has the potential to create a throughput bottleneck that can harm responsiveness.

Fault tolerance

While orchestration enhances recoverability for domain services, it creates a potential single point of failure for the workflow. This can be addressed with redundancy, but that also adds more complexity.

Scalability

This communication style doesn't scale as well as choreography because the orchestrator adds more coordination points, which cuts down on potential parallelism.

Service coupling

Having a central orchestrator creates tighter coupling between it and the domain components, which is sometimes necessary but is frowned upon in microservices architectures.

Similarly, we discussed choreography in both microservices and event-driven architectures. The trade-offs for choreographed workflows include:

Responsiveness

This communication style has fewer single chokepoints, thus offering more opportunities for parallelism.

Scalability

The lack of coordination points like orchestrators allows more independent scaling.

Fault tolerance

The lack of a single orchestrator allows the architect to use multiple instances to enhance fault tolerance. They could, of course, create multiple orchestrators, but because all communication must go through them, multiple orchestrators are more sensitive to the workflow's overall level of fault tolerance.

Service decoupling

No orchestrator means less coupling.

Disadvantages of the choreography communication style include:

Distributed workflow

Having no workflow owner makes managing errors and other boundary conditions more difficult.

State management

Having no centralized state holder hinders ongoing state management.

Error handling

Error handling is more difficult without an orchestrator because the domain services must have more workflow knowledge.

Recoverability

Recoverability becomes more difficult without an orchestrator to attempt retries and other remediation efforts.

These two patterns nicely illustrate our distinction between styles and patterns. Any distributed architecture can use any of these communication patterns, and architects should understand how to evaluate their trade-offs. Remember our Second Law of Software Architecture: *you can't just do trade-off analysis once and be done with it*. They also illustrate that common patterns exist everywhere, which is why they are common.

CQRS

Another common communication pattern that appears in many distributed architectures (and a few monolithic ones) is *Command-Query-Responsibility-Segregation* (CQRS). This simple communication and data pattern splits a commonly monolithic communication with a database into two, illustrated in Figure 20-3.

Figure 20-3. Client/server versus CQRS

In Figure 20-3, the structure on the left illustrates a common *client/server* data interaction, where the application queries the database and performs transactional writes by utilizing the database as part of the application infrastructure. While this is a common pattern, some systems have stark differences between read and write volumes, or want to isolate reads from writes for the sake of security and other concerns. For those systems, CQRS, illustrated on the right in Figure 20-3, solves this problem.

CQRS isolates writes into one datastore (usually a database; sometimes another infrastructure, such as a durable message queue), which synchronizes the data to another database (usually asynchronously), which services read requests.

By separating reads and writes, architects can isolate different architectural characteristics, depending on the data. This also allows them to use different data models for each database if necessary.

CQRS is a good example of a data communication pattern that facilitates differing architectural characteristics for different types of data capabilities, security concerns, or other factors that benefit from physical separation.

Infrastructure

Software architecture has patterns in every place where teams have found useful contextualized solutions to common problems, and these patterns often intersect with other parts of the ecosystem (see Chapter 26 for a complete discussion).

Architects care about coupling: between components, data elements, APIs…and infrastructure, as exemplified by the *Broker-Domain pattern*.

Broker-Domain Pattern

In this section, we'll look at the same order-placement workflow, this time implemented with an event-driven architecture, as illustrated in Figure 20-4.

Figure 20-4. An order placement workflow implemented in EDA

As you've seen, EDA uses events to communicate between services, so event handlers need to subscribe to the proper services to build the workflow. Event handlers are implemented by *brokers*, which are part of the infrastructure of the architecture. In EDA, the topic or queue is typically owned by the sender: for example, `Payment` needs to know a topic's address to subscribe to it.

In Figure 20-5, `OrderPlacement` "owns" the broker to which other processors will subscribe; in other words, the infrastructure required to support this service includes the broker. If a system uses only one broker for all communication, all of its services depend on a single part of the infrastructure.

Figure 20-5. In event-driven choreography, the topic or queue is typically owned by the sender

The infrastructure for the workflow depicted in Figure 20-6 includes a single broker, meaning that each event processor "knows" where to go to subscribe to workflow collaborators. A single broker also allows a single place for logging, monitoring, and other governance.

Figure 20-6. Using one broker for the entire workflow

However, one of the goals of a distributed architecture is to improve fault tolerance. What happens if the single broker in Figure 20-6 goes down? The entire workflow stops working. Another potential issue is scale. If all messages must go through a single broker, there's a risk that the broker will become swamped as the volume of messages increases.

An alternative approach is the *Domain-Broker* pattern, which treats infrastructure in a similar manner to the granularity of domains. Consider Figure 20-7.

Figure 20-7. Utilizing the Domain-Broker pattern to assign ownership to infrastructure

In the alternative architecture shown in Figure 20-7, each group of related services shares a broker, reflecting the architecture's overall domain partitioning. This solution still allows good discovery while increasing fault tolerance, scalability, elasticity, and a host of other operational architectural characteristics. Note, however, that neither of these approaches is a "best practice." The Single-Broker pattern's trade-offs appear in Table 20-2.

Table 20-2. Single-Broker pattern trade-offs

Advantages	Disadvantages
Centralized discovery	Fault tolerance
Least possible infrastructure	Throughput limits

The Domain-Broker pattern also features trade-offs, shown in Table 20-3.

Table 20-3. Domain-Broker pattern trade-offs

Advantages	Disadvantages
Better isolation	More difficult discovery of queues/topics
Matches domain boundaries	More infrastructure = more expensive
More scalable	More moving parts to maintain

Architects must balance discovery with the need to isolate domains as they decide which of these infrastructure patterns make the most sense for their particular systems.