Chapter 27. The Laws of Software Architecture, Revisited

Way back in Chapter 1, we described the three laws of software architecture:

- Everything in software architecture is a trade-off.
- *Why* is more important than *how*.
- Most architecture decisions aren't binary but rather exist on a spectrum between extremes.

Just about every example in this book has illustrated these laws, which points to their origin story. When we wrote the first edition, we hoped to find numerous things that seemed universally true about software architecture and codify them as *laws*. To our surprise, we ended up identifying just two laws in the first edition, then uncovered one more while writing the second. True to our original intent, these three laws seem pretty universal and inform many important perspectives for working software architects.

In this brief chapter, we'll revisit these laws in light of the examples we've shown and point out some nuances about trade-off analysis.

First Law: Everything in Software Architecture Is a Trade-Off

Our first law is one of the defining characteristics of software architecture—everything is a trade-off. Many people think that the job of a software architect is to find silver bullet solutions to sticky problems and become a hero, but that rarely happens. (Architects rarely get credit for good decisions, but always get blamed for bad ones.) No, the real job of software architecture is analyzing trade-offs.

We believe, for a couple of reasons, that every architect should hone their reputation as an objective arbiter of trade-offs, rather than evangelizing for a particular approach.

First, evangelism in architecture is dangerous in the long term, because yesterday's best practice tends to become tomorrow's antipattern. Architects make trade-off decisions based on current factors and situations as best they can, with incomplete knowledge. However, even if a decision is sound when it's made, the software development ecosystem exists in a constant state of evolution and churn, which means that the circumstances slowly change and are likely to eventually weaken (or invalidate) the decision. If the architect has invested social capital in evangelizing for that solution, their reputation may suffer when that decision must change later. Always stay clear-eyed and objective about technology choices to avoid attaching your credibility to a decision that didn't age well.

Second, decision makers in organizations aren't really looking for enthusiastic advocacy so much as sober objectivity. An architect who develops a reputation as the go-to person for an objective trade-off analysis becomes a valuable asset to their organization. When the decision is critical, decision makers want someone whose judgment they can trust; that should be you.

Let's look at some examples of trade-off analysis for a couple of software architecture decisions, and then discuss a common gap in understanding.

Shared Library Versus Shared Service

A common conundrum for architects involves shared behavior in distributed architectures, such as microservices or EDA: should we use a shared library that is compiled into each service at build time, or use a shared service that other services call at runtime, as illustrated in Figure 27-1?

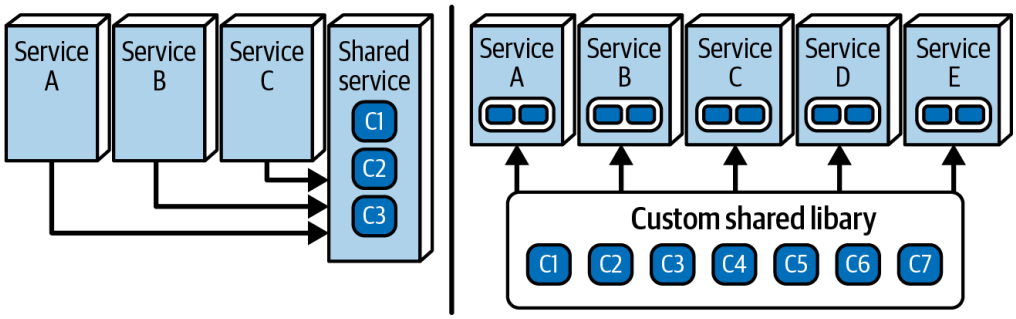

Figure 27-1. Should the solution use a shared library or shared service for common functionality?

In Figure 27-1, the service on the left encapsulates the shared behavior, and other services call to access it. On the right, we have a shared library, which is compiled into each service before deployment. Which is better?

By now, readers know the answer to all such questions: “It depends!” But you need to answer the inevitable follow-up question: “Depends on what?” Like many nontrivial decisions in software architecture, the answer to this question isn’t immediately obvious, so it’s time for you, as the architect, to undertake trade-off analysis.

First, you must determine all the contextualized trade-offs that make a difference for this solution. This list will be highly specific to both the organization and the solution, so you’ll need to rely on your knowledge of the organization, technology landscape, team capabilities, budgets, and everything else that informs your trade-off options. For this solution, we’ve created the following list of pertinent factors:

Heterogeneous code

If solutions are written in multiple platforms, then the service will be easier to work with because, regardless of the technology stack, callers access the service via the network, making the implementation platform irrelevant. For the library, the team will need a version of the code for each technology stack and will have to keep the versions in sync, greatly adding to the project’s overall complexity.

High code volatility

Recall that code volatility measures how fast the code changes (commonly known as *churn*). In a service, callers access new functionality as soon as they deploy the updated service, so if you need to change the library, you’ll have to recompile and deploy each service to take advantage of the new features.

Ability to version changes

Versioning is much easier in the library than in the service. When the library needs to update, the team can resolve the version differences at compile time and build just what they need. For the service, version information must be determined at runtime, which complicates the interaction with the service.

Overall change risk

Change risk is better for the library. Once you change the library code and successfully compile it into the service, you can have high confidence that it will work. The service, on the other hand, may

change without any compile-time verification, which raises the potential likelihood of a runtime fault at invocation time.

Performance

The library will clearly have better performance, as the calls to shared functionality are in-process calls, compared to the network calls required for the service. These calls will be much slower than in-process calls because of network latency and other factors.

Fault tolerance

As we discussed in Chapter 9, runtime access to services always suffers potential network issues, including the service in question here. The library provides better fault tolerance: once you compile, test, and deploy the service, you can have high confidence that it is stable.

Scalability

Just like performance, calls between services suffer from latency, which diminishes scalability. Thus, the library will offer better scalability, as access to the shared behavior is a very efficient in-process call.

Once you've determined your list of important factors, you can create a matrix to compare how well each does for each criterion, as shown in Table 27-1.

Table 27-1. Trade-offs between shared service and shared library

Factor	**Shared library**	**Shared service**
Heterogeneous code	-	+
High code volatility	-	+
Ability to version changes	+	-
Overall change risk	+	-
Performance	+	-
Fault tolerance	+	-
Scalability	+	-

The accumulation of positives for the shared library makes it the winner…at least, for these factors and in this context. It may or may not be the solution to

the problem at hand—you may need to apply additional weighting (see "Second Corollary: You Can't Do It Just Once" for more details)—but now you and your team have a good idea of the forces at play.

Synchronous Versus Asynchronous Messaging

Consider the following trade-off analysis: your team is building a distributed architecture to send trade information to both the `Notification` and `Analytics` services, and you're trying to decide whether to use a *queue* or *topic* to implement this behavior.

The first option is a queue, or a point-to-point communication protocol. As you learned in Chapter 2, the publisher knows who is receiving the message. To reach multiple consumers, the publisher needs to send a message to one queue for each consumer. If the `Trading` service wants to use queues to tell the `Analytics` service and the `Reporting` service about trades, the implementation will appear as in Figure 27-2.

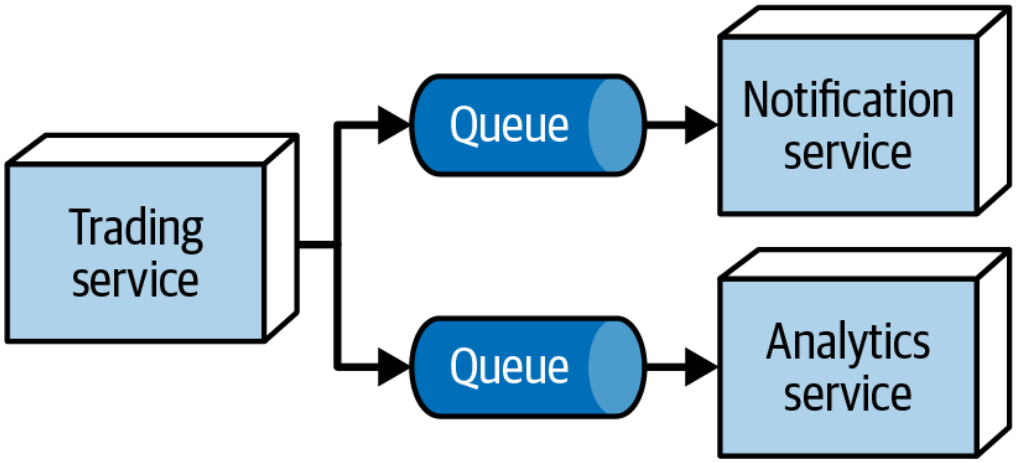

Figure 27-2. Using a queue to pass trade information to `Notification` and `Analytics`

In Figure 27-2, the `Trading` service issues a message to each consumer via a queue. Each consumer has its own queue, and the sender posts a message to each.

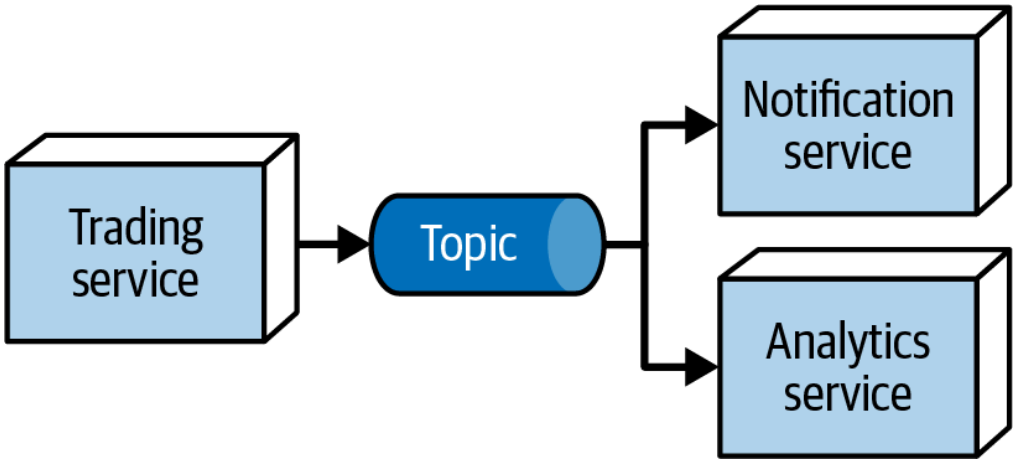

Figure 27-3. Using a topic to pass trade information to `Notification` and `Analytics`

The alternative topology uses a *topic* instead, which implements a broadcast rather than point-to-point communication. In Figure 27-3, the `Trading` service posts a single message to a topic. Each consumer subscribes to the topic, and they all receive a notification when a message appears. In this case, the publisher of the message doesn't know (or care) who the consumers are—in fact, the team can add new consumers anytime without changing the other consumers or the producer.

Both options are viable, so how should you decide? By performing a trade-off analysis, of course.

With queues, every service that the `Trading` service needs to notify will need a separate queue. This is nice if the `Notification` service and the `Analytics` service need different information, since you can send different messages to each queue. The `Trading` service is aware of every system to which it communicates, which makes it harder for another (potentially rogue) service to "listen in." That's especially beneficial if security is high on the priority list. Because each queue is independent, you can monitor them separately and even scale them independently if needed. The `Trading` service is tightly coupled to its consumers—it knows exactly how many there are. However, if you need to send messages to the `Compliance` service, you'll have to rework the `Trading` service to start sending messages to a third queue.

Table 27-2 summarizes the trade-offs of using queues for this project.

Table 27-2. Trade-offs when using a queue

Advantage	Disadvantage
Supports heterogeneous messages for different consumers	Higher degree of coupling
Allows independent monitoring of queue depth	Trading service must connect to multiple queues
More secure	Requires additional infrastructure
Less extensible (must add queues for more consumers)	Good support for extensibility and evolvability

Let's do the same exercise for topics. One advantage is clear: topics are extremely extensible. Any new consumer interested in, for example, compliance, can subscribe to that topic without touching any of the existing behavior. However, this advantage has downsides: each consumer must consume the same message from the topic, which can lead to *stamp coupling* (shown in Figure 9-9). And, because every consumer can read the entire message, there are security concerns: *should* everyone be able to read the entire message?

With this perspective in hand, you can make your trade-off table, shown in Table 27-3.

Table 27-3. Trade-offs when using a topic

Advantage	Disadvantage
Low coupling	Homogeneous message for each consumer
Trading service generates just one message	Can't monitor or scale individual consumers
More extensible/evolvable	Less secure
	Less scalability options

Now that you've done the trade-off analysis, you need to return to the organizational goals for this solution and see which option is the better fit. If security is more important, you should probably select queues. If the

organization is growing rapidly and has other services interested in trades, then extensibility might be a priority, which would lead you to use topics.

First Corollary: Missing Trade-Offs

Our first law has two corollaries. The first is:

> *If you think you've discovered something that isn't a trade-off, more likely you just haven't identified the trade-off…yet.*
>
> —Corollary 1

The essence of a software architecture decision is trade-off analysis, but what happens when you encounter a decision that seems to not have any trade-offs? Our advice: keep looking!

Consider code reuse. Surely this is a purely beneficial practice, right? The more code the organization can reuse, the less code it must write, saving time and duplication.

Two factors dictate how effective code reuse will be. Architects often discover the first but miss the second. The first factor is *abstraction*: if you can abstract this code and use it from multiple call points, it's a good candidate for reuse. The second factor is *low volatility*. When a team reuses a module of code that's always changing, this creates churn in the entire system. Every time the shared code changes, all callers of that code must coordinate around that change. Even if it isn't a breaking change, the team must still verify that the change hasn't broken anything. When architectures reuse code inappropriately, teams end up chasing breaking changes all over the architecture. This was one of the key lessons architects as a field learned from the orchestration-driven SOA architecture style (covered in Chapter 17), one of the underlying philosophies of which was to try to reuse as much code as possible. From a practical standpoint, teams working in those architectures were swimming through quicksand: every change had the potential to send unpredictable side effects rippling out through the system.

That's the hidden trade-off: reusing code effectively requires good abstraction *and* low volatility. This is why the most successful reuse targets in architecture are "plumbing": technology frameworks, libraries, platforms, and so on. The portion of most applications that changes the fastest is the domain (the

motivation for writing the software in the first place), so domain concepts are terrible candidates for reuse. (Notice that this underlies the DDD principle of bounded context—no bounded contexts can reuse any of the implementation details of another bounded context.)

WHY WE CAN'T HAVE NICE THINGS—TRADE-OFFS!

As professional consultants, our clients often make requests like the following: "We like the idea of microservices and distributed architectures that feature high degrees of decoupling because they allow high degrees of agility and fast deployment. However, we also want a high degree of institutional reuse so that teams aren't constantly rewriting code."

We have to be the bearers of bad news here: you cannot have both these things, because the way a system implements reuse is via coupling. No organization can have both decoupling *and* a high degree of reuse. The two things are fundamentally incompatible. This is a prime example of how organizations can misunderstand key trade-offs.

Second Corollary: You Can't Do It Just Once

It would be nice if an architect could just perform one trade-off analysis—just think *really hard* and decide once and for all to use choreography for all workflows. But there are two problems with trade-off analysis. First, there are often dozens or even hundreds of variables (technical and otherwise) that contribute to the decision: complexity, team experience, budget, team topology, schedule pressure…the list is endless. Subtle differences in those variables can push a particular analysis in one direction or another, making this an ongoing exercise. It's dangerous for architects to make sweeping, semipermanent decisions based on assumptions that might not be valid for future applications of this solution.

Think of this corollary as job security for architects. We have to keep doing trade-off analysis over and over, even for seemingly similar situations. This reinforces the idea that the real job is trade-off analysis, not making permanent, perfect decisions.

Second Law: Why Is More Important Than How

Our second law emphasizes the importance of *why* over *how*. As experienced architects, we can look at an existing system and tell someone *how* it works. However, there will be some decisions where it isn't clear *why* the previous architect chose this option over another, because they didn't record all of the decision criteria with their ultimate solution.

This is why we emphasize the importance of using both architecture diagrams *and* ADRs (covered in Chapter 21). Every trade-off analysis generates a tremendous amount of context that doesn't appear in the solution. It's critical to document that analysis (along with the known compromises and limitations of the solution) to prevent future architects (who might be you) from having to redo the analysis just to understand *why*.

Out of Context Antipattern

One antipattern that's common in trade-off analysis is the *Out of Context* antipattern. This antipattern occurs when the architect understands the trade-offs but not how to *weight* all of them based on the current context.

Consider the trade-off analysis we did in "Shared Library Versus Shared Service". Taken objectively, the shared library appears to be the preferred solution, since it achieved more positive than negative assessments. However, this trade-off analysis has a potential flaw: do all of its criteria have equal weight?

Imagine that a team has code in several platforms. They aren't overly concerned with performance or scale, but want a clean way to manage shared behavior. In this case, the first two trade-off criteria carry a much higher priority, which should lead them to choose the shared service. As a bonus, the team has already figured out what issues they will have to mitigate.

Architects rely on experience to build trade-off criteria, but must also weigh those criteria to find the correct fit for the solution. Generic trade-off analysis isn't very useful—it only becomes valuable when applied in a specific context.

The Spectrum Between Extremes

In our first edition, we identified only two laws. However, we have gradually realized that a third law exists:

> *Most architecture decisions aren't binary but rather exist on a spectrum between extremes.*
>
> —Third Law of Software Architecture

It would be nice to live in a world with nice, clean, binary decisions—but software architecture doesn't exist in that world. People often observe how difficult it is to come up with comprehensive definitions of important concepts in software architecture: architecture versus design, orchestration versus choreography, topics versus queues, and so on. The underlying reason is that the decision criteria aren't binary; they lie along a rather messy spectrum.

In "Architecture Versus Design", we discussed the spectrum between architecture and design and how to determine where a particular decision lies within that spectrum. In fact, this law provides a useful way to think about decisions that are closer to the software architecture side of the spectrum than the design side:

> *A software architecture decision is one where each of the options has significant trade-offs.*

If everything in software architecture is a trade-off, then an architectural decision must involve trade-offs for each option.

As an architect, don't try to reduce every decision to a binary—few of those exist in our world. That's one of the reasons that every answer in software architecture is "it depends"—it depends on where on the spectrum of possible solutions the criterion in question falls.

As architects, we make our decisions in a swamp of uncertainty. Accommodating this is annoying, but necessary. Not only do we sometimes have to base important decisions on incomplete information, but often the decision wouldn't be clear-cut, even if we had the full scoop, because it resides somewhere on a spectrum between two extremes.

Welcome to software architecture!

Parting Words of Advice

> *How do we get great designers? Great designers design, of course.*
>
> —Fred Brooks

> *So how are we supposed to get great architects, if they only get the chance to architect fewer than a half-dozen times in their career?*
>
> —Ted Neward

Practice is the proven way to build skills and become better at anything in life, including architecture. We encourage new and existing architects to keep honing their skills in the craft of designing architecture as well as widening their individual technology breadth. To that end, we've created some architecture katas modeled after the examples in this book, which you can find on the companion website. We encourage you to use them to practice and build your skills in architecture.

People who use our katas often ask: is there an answer guide somewhere? Unfortunately, no. To quote Neal:

> *There are not right or wrong answers in architecture—only trade-offs.*

When your authors started using architecture katas during our live training classes, we initially kept the drawings the students produced, with the goal of creating an answer repository. We quickly gave up, though, because we realized that these were incomplete artifacts. The teams had captured *how* they implemented their solutions with drawings of topologies. The *why* was much more interesting—but while they explained the trade-offs they considered in class, they didn't have the time to create architecture decision records. Keeping just the *how* meant we had only half of the story.

So, our parting words of advice: always learn, always practice, and *go do some architecture*!