Chapter 9. Foundations

We know you are anxious to get to the details of architecture styles and patterns, but first we must cover some fundamental and definitional material to set the proper context for later chapters.

Styles Versus Patterns

We first need to distinguish between *architectural styles* and *patterns in architecture*, which are easily confused.

An architecture's style describes several different characteristics of that architecture, including its:

Component topology

> An architectural style defines how components and their dependencies are organized. For example, a layered architecture organizes component layers by their technical capabilities, whereas a modular monolith organizes its components around domains (more about this difference in "Architecture Partitioning").

Physical architecture

> Often, the style dictates the type of physical architecture: either monolithic or distributed. For example, a modular monolith is generally a monolithic architecture with a single database, whereas an event-driven architecture is always distributed.

Deployment

> A system's granularity and its deployment frequency are often associated with its architectural style. Teams generally deploy monolithic architectures as a single deployment along with a single relational database. Conversely, highly agile distributed architectures, such as microservices, feature automated integration, automated provisioning, and sometimes automated deployments; they are generally deployed in pieces, with a much faster cadence.

Communication style

> The architectural style also dictates how components communicate with each other. Monolithic architectures can make method calls within the monolith, whereas distributed architectures communicate via network protocols, like REST or message queues.

Data topology

> Just like component topology, a system's data topology is often dictated by its architectural style. Monolithic architectures tend to have a monolithic

database, whereas distributed architectures sometimes separate data—depending on the philosophy of the architecture style.

Naming a style provides a concise way to describe this complex set of factors. Each name captures a wealth of understood detail—that's one of the purposes of design patterns. However, whereas a pattern captures a contextualized solution, a style is more specific to architecture, describing the aspects listed above. An architecture style describes the architecture's topology and its assumed and default characteristics, both beneficial and detrimental. We cover a few common modern architecture patterns in Chapter 20. Architects should be familiar with several fundamental styles that are the common building blocks of systems.

Where Do Architectural Styles Come From?

Contrary to popular opinion, there's no official architectural cabal that meets in an ivory tower to decide what new architectural styles come next. Rather, new styles emerge from the constantly evolving ecosystems within which architects work.

For example, say a clever architect notices that a new capability that just appeared in the ecosystem solves a particular nagging problem. They decide to combine it with several other things, new and old. Other architects see this clever solution and copy it. It becomes common enough that giving it a name makes it easier to discuss.

The microservices architecture style is a great example of this phenomenon. The rise of new DevOps capabilities, reliable open source operating systems, and the domain-driven design philosophy allowed architects to build systems in new ways to solve issues like scalability. The name *microservices* came about as a reaction to the common architecture styles at the time, which featured large services and extensive orchestration. Microservices is a label, not a description. It is not a commandment for teams to build the smallest services they can but rather a way to refer to the architecture style.

Fundamental Patterns

Several fundamental patterns appear again and again throughout the history of software architecture, generally because they provide a useful perspective on organizing code, deployments, or other aspects of architecture. For example, the concept of layers separating different concerns based on functionality is as old as software itself. Yet layers (in both styles and patterns) continue to manifest in different guises, including the modern variants we discuss in Chapter 10.

Alas, there's another common antipattern that stems from the absence of architecture: the Big Ball of Mud.

Big Ball of Mud

Architects refer to the absence of any discernible architecture structure as a *Big Ball of Mud*, named after the antipattern Brian Foote and Joseph Yoder defined in a 1997 paper presented at the 1997 conference on Patterns Languages of Programs.

> A *Big Ball of Mud* is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated.
>
> The overall structure of the system may never have been well defined.
>
> If it was, it may have eroded beyond recognition. Programmers with a shred of architectural sensibility shun these quagmires. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.

Today, *Big Ball of Mud* might describe a simple scripting application with no real internal structure that has its event handlers wired directly to database calls. Many trivial applications start like this, then become unwieldy as they grow.

In general, avoid this type of architecture at all costs. Its lack of structure makes change increasingly difficult. Such architectures also suffer from problems with deployment, testability, scalability, and performance. The larger these systems become, the worse the pain caused by their lack of architecture becomes.

Unfortunately, this antipattern occurs quite often in the real world. Few architects intend to create a mess, but many projects inadvertently manage to, usually because of a lack of governance around code quality and structure. For example, Neal worked on one client project whose structure appears in Figure 9-1.

The client (whose name is withheld for obvious reasons) created a Java-based web application as quickly as possible over several years. The technical visualization in Figure 9-1 shows its coupling: each dot on the perimeter of the circle represents a class, and each line represents a connection between the classes, with bolder lines indicating stronger connections. In this code base, changing a class in any way makes it difficult to predict the effect on other classes, making change a terrifying affair.

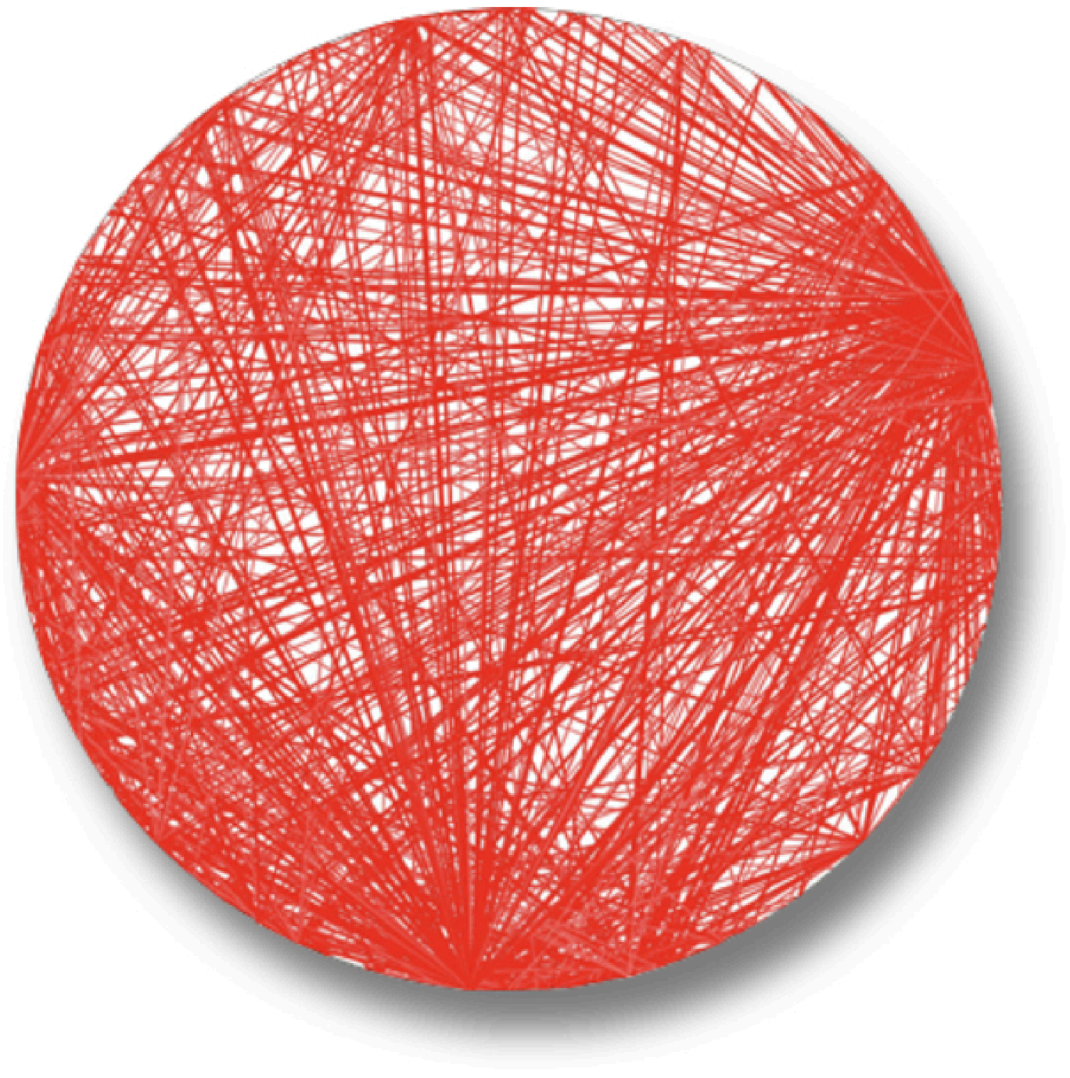

Figure 9-1. A Big Ball of Mud architecture visualized from a real code base. (Made with a now-retired tool called XRay, an Eclipse plug-in.)

The problem with Big Ball of Mud architectures isn't just their lack of structure. Because everything is coupled to everything else, changes tend to have hard-to-predict rippling side effects. This problem can reach a critical point where developers spend all their time chasing bugs and their side effects, rather than working on new features.

Unitary Architecture

In the beginning, there was only the computer, and software ran on it. The two started as a single entity, then split as they evolved and the need grew for more sophisticated capabilities. For example, mainframe computers started as singular systems, then gradually separated data into its own kind of system. Similarly, when personal computers (PCs) were first commercially developed, the focus was largely on single machines. As networking PCs became common, distributed systems (such as client/server) appeared.

Client/Server

Few unitary architectures exist outside embedded systems and other highly constrained environments. Generally, software systems tend to add functionality over time, and separating concerns becomes necessary to maintain their operational architecture characteristics, such as performance and scale.

Many architecture styles deal with how to separate parts of the system efficiently. One fundamental style in architecture separates technical functionality between frontend and backend: this is called a *two-tier*, or *client/server*, architecture. It comes in different flavors, depending on the era and the system's computing capabilities.

Desktop and database server

One early PC architecture encouraged developers to write rich desktop applications in user interfaces like Windows, separating data into a separate database server. This architecture coincided with the appearance of standalone database servers that could connect through standard network protocols. This allowed presentation logic to reside on the desktop, while the more computationally intense action (both in volume and complexity) occurred on more robust database servers.

Browser and web server

Once modern web development arrived, it became common to split architectures into a web browser connected to a web server (which, in turn, was connected to a database server). This separation of responsibilities was similar to the desktop variant but with even thinner clients as browsers, allowing a wider distribution both inside and outside firewalls. Even though the database is separate from the web server, many architects still consider this a two-tier architecture, because the web and database servers run on one class of machine within the operations center, while the UI runs on the user's browser.

Single-page JavaScript applications

As web responsiveness improved, so did JavaScript implementations in browsers. A family of client/server style applications emerged that resembles the original desktop variation, but with the rich client written in JavaScript in the browser, rather than as a desktop application.

As this section illustrates, there will always be layers to separate different parts of architectures, depending on the needs of the application and the capabilities of the platform.

Three-tier

Three-tier architecture, which provided even more layers of separation, became popular during the late 1990s. As tools like application servers became prominent in Java and .NET, companies started building even more layers into their topologies. One system might have a database tier using an industrial-strength database server; an application tier managed by an application server; and a frontend coded in generated HTML and, increasingly, JavaScript (as its capabilities expanded).

The three-tier architecture corresponded with network-level protocols such as Common Object Request Broker Architecture (CORBA) and Distributed Component Object Model (DCOM) to facilitate building distributed architectures.

Just as developers today don't worry about how network protocols like TCP/IP work (because they just work), most architects don't have to worry about this level of plumbing in distributed architectures. The capabilities offered by that era's tools exist today either as tools (like message queues) or as architecture patterns (such as event-driven architecture, covered in Chapter 15).

Three-Tier Architectures, Language Design, and Long-Term Implications

During the 1990s, as the Java language was designed, three-tier computing was all the rage. People assumed that, in the future, all systems would have three-tier architectures. One of the common headaches with the existing languages at the time, such as C++, was how cumbersome it was to move objects over the network in a consistent way between systems. So Java's designers decided to build this capability into its core, using a mechanism called *serialization*.

Every Java object implements an interface that requires it to support serialization. The designers figured that since three-tiered architecture would be around forever, baking it into the language would offer great convenience. Of course, that architectural style came and went—yet the leftovers appear in Java to this day. Even though virtually no one still uses serialization, new Java features must support it for backward compatibility, greatly frustrating language designers.

Understanding the long-term implications of design decisions has always eluded us, in software as in other engineering disciplines. The perpetual advice to favor simple designs is in many ways a future-proofing strategy.

Architecture Partitioning

The First Law of Software Architecture states that everything in software is a trade-off, and that includes how architects partition components in an architecture. Because components represent a general containment mechanism, architects can partition them any way they want. There are several common styles, with different sets of trade-offs. One particular type of component arrangement has an outsized impact: *top-level partitioning*.

Of the two architecture styles depicted in Figure 9-2, one will be familiar to many: the *layered monolith* (discussed in detail in Chapter 10). The other, called *modular monolith*, is an architecture style popularized by Simon Brown that consists of a single deployment unit, associated with a database and partitioned around domains rather than technical capabilities (discussed in Chapter 11). These two styles represent different methods of top-level partitioning. Note that in both variations, each top-level layer or component likely has other components embedded within it. Top-level partitioning is of particular interest to architects because it defines a fundamental architecture style and way of partitioning code.

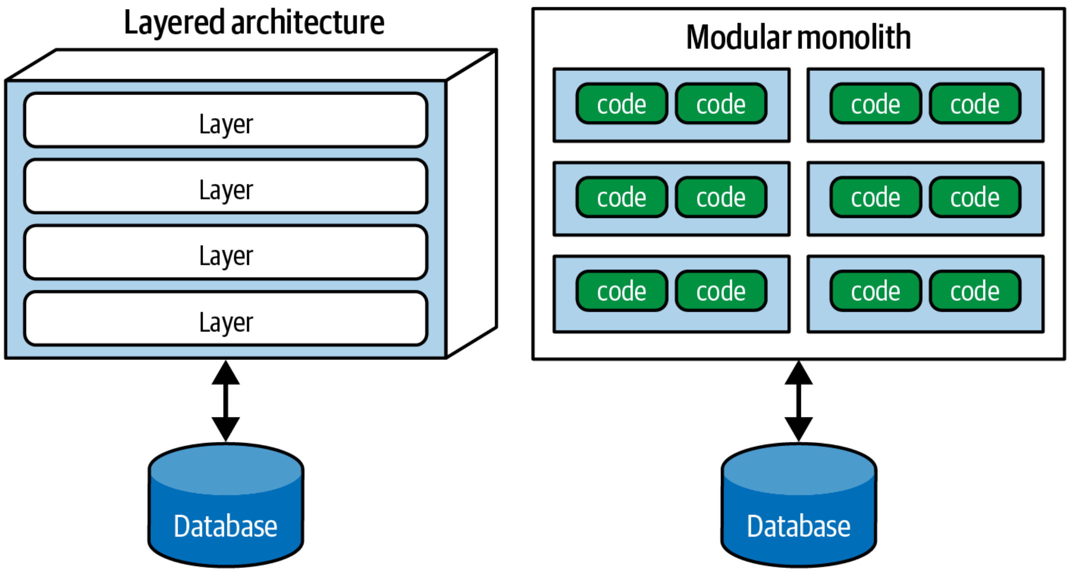

Figure 9-2. Two types of top-level partitioning: technical (such as the layered architecture) and domain (such as the modular monolith)

Organizing architecture based on its technical capabilities, like the layered monolith style does, represents *technical top-level partitioning*.

A common version of this appears in Figure 9-3, where the architect has partitioned the system's functionality into *technical* capabilities: presentation, business rules, services, persistence, and so on. This way of organizing a code base certainly makes sense. All the persistence code resides in one layer, making it easy for developers to find persistence-related code. Even though the basic concept of layered architecture predates it by decades, the Model-View-Controller design pattern (one of the fundamental patterns in *Head First Design Patterns* by Eric Freeman and Elisabeth Robson (O'Reilly, 2020)), matches with this architectural pattern, making it easy for developers to understand. Thus, in many organizations, it is the default architecture.

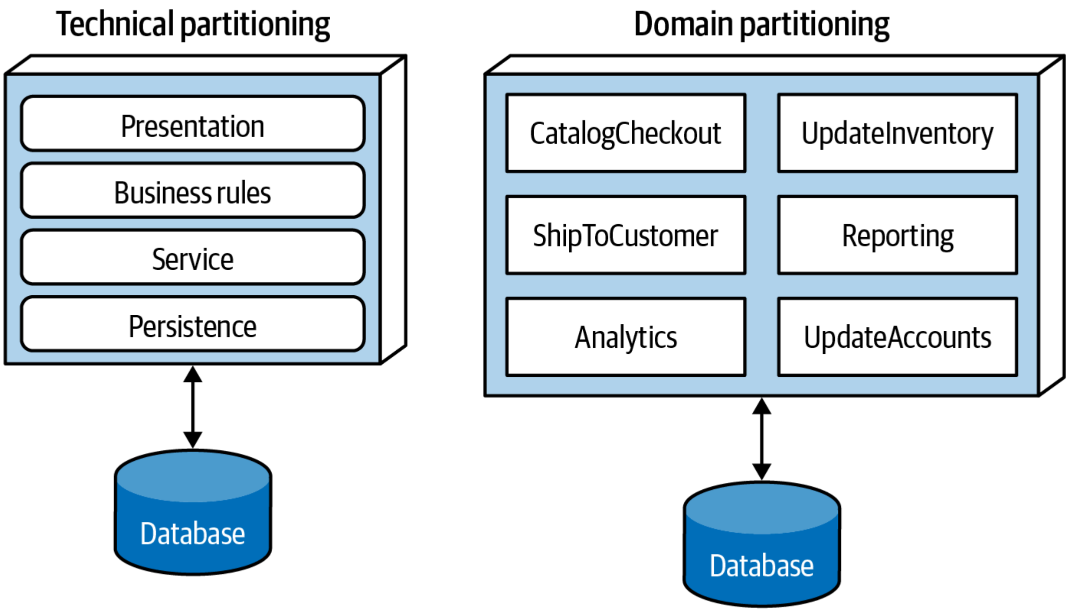

Figure 9-3. Two types of top-level partitioning in architecture

An interesting side effect of layered architecture's predominance relates to how companies often organized the seating in their physical offices according to different project roles. Because of Conway's Law, when using a layered architecture, it makes some sense to have all the backend developers sit together in one department, the DBAs in another, the presentation team in another, and so on.

The other architectural variation in Figure 9-3 is *domain partitioning*, a modeling technique for decomposing complex software systems that organizes components by domain rather than technical capabilities. Domain partitioning was inspired by Eric Evans's book *Domain-Driven Design*. In DDD, the architect identifies domains or workflows, independent and decoupled from each other. The microservices architecture style is based on this philosophy.

A modular monolith architecture is partitioned around domains or workflows, rather than technical capabilities. Because components often nest within one another, each component in the domain-partitioned architecture shown in Figure 9-3 (for example, `CatalogCheckout`) may use a persistence library and have a separate layer for business rules, but the top-level partitioning will still revolve around domains.

Conway's Law

Back in the late 1960s, Melvin Conway made an observation that has become known as *Conway's Law*:

> Organizations which design systems…are constrained to produce designs which are copies of the communication structures of these organizations.

Paraphrased, this law suggests that when a group of people designs some technical artifact, the structures of their design will replicate how those people communicate. People at all levels of organizations see this law in action, and they sometimes base decisions on it. For example, while it's common for organizations to partition workers based on technical capabilities, this is an artificial separation of common concerns that can hamper collaboration.

A related observation, coined by Jonny Leroy of Thoughtworks, is the *Inverse Conway Maneuver*, which suggests evolving the structures of teams and organizations together to promote the desired architecture. This consideration has since become universally known as *team topologies*.

Organizations have begun to realize that team topologies can have a significant impact on many important facets of their business, including software architecture. In the upcoming style-focused chapters, we discuss each architecture style's effect on these team types.

One of the fundamental distinctions between architecture patterns is what type of top-level partitioning each supports. In the style-specific chapters that follow, we cover this distinction for each individual pattern. Top-level partitioning also has a huge impact on whether the architect decides to initially identify components technically or by domain.

Architects using technical partitioning organize the components of the system by their technical capabilities: presentation, business rules, persistence, and so on. One of the organizing principles of the layered architecture is *separation of technical concerns*, which creates useful levels of decoupling. For example, if the Service layer is only connected to the Persistence layer below and the Business Rules layer above, then changes in persistence will potentially affect only those layers. This decoupling reduces the potential for rippling side effects on dependent components.

It's certainly logical to organize systems using technical partitioning, but, like all things in software architecture, there are some trade-offs. The separation enforced by technical partitioning enables developers to find certain categories of the code base quickly, since it is organized by capabilities, but most realistic software systems require workflows that cut across technical capabilities.

In the technically partitioned architecture shown in Figure 9-4, consider the common business workflow of `CatalogCheckout`. The code to handle `CatalogCheckout` in the technically partitioned architecture appears in all the layers. In other words, the domain is smeared across the technical layers.

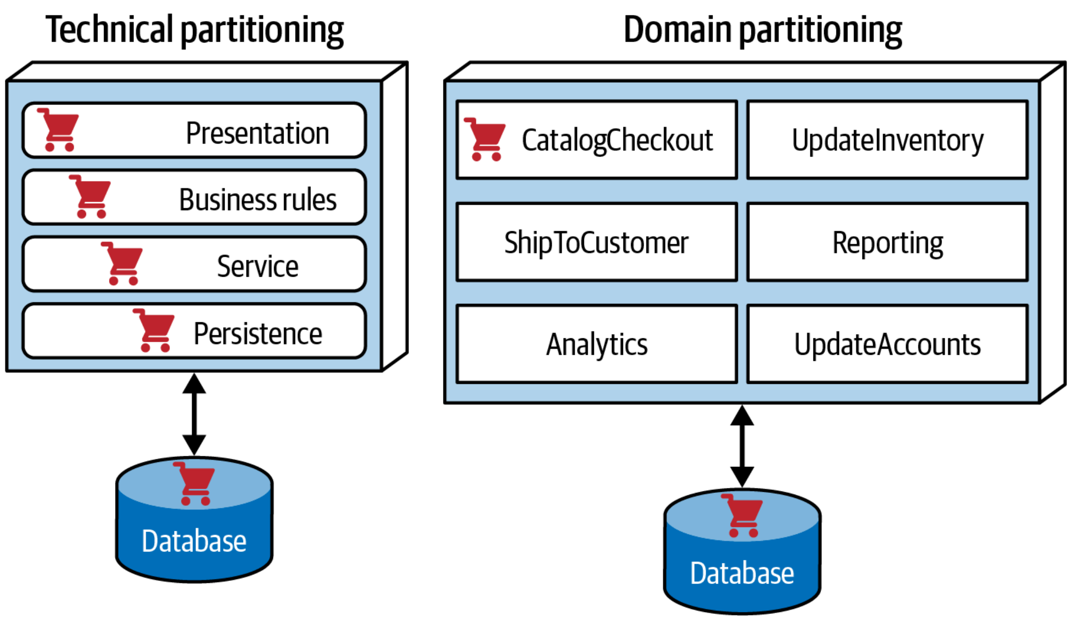

Figure 9-4. Where domains/workflows appear in technically partitioned and domain-partitioned architectures

Contrast this with the domain-partitioned architecture shown in Figure 9-4, in which the architects have built top-level components around workflows and domains. Each component may have subcomponents, including layers, but the top-level partitioning focuses on domains, which better reflects the kinds of changes that most often occur on projects.

Neither of these styles is more correct than the other. (Refer to the First Law of Software Architecture.) That said, we have observed a decided industry trend over the last few years toward domain partitioning for both monolithic and distributed (for example, microservices) architectures. As we've noted, this is one of the first decisions an architect must make.

Kata: Silicon Sandwiches—Partitioning

Consider the case of one of our example katas, "Kata: Silicon Sandwiches". Let's start by considering the first of two possibilities for Silicon Sandwiches: domain partitioning, shown in Figure 9-5.

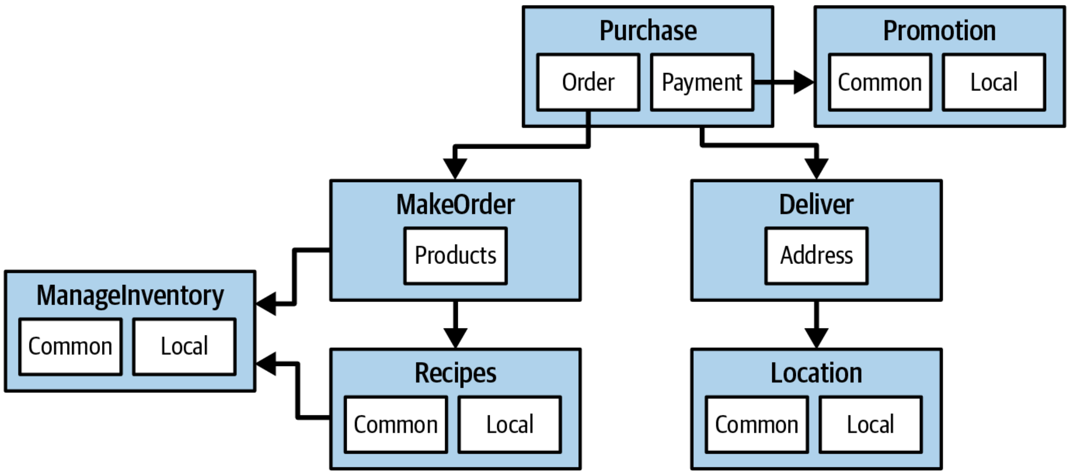

Figure 9-5. A domain-partitioned design for Silicon Sandwiches

In Figure 9-5, the architect has designed around domains (workflows), creating discrete components for `Purchase`, `Promotion`, `MakeOrder`, `ManageInventory`, `Recipes`, `Delivery`, and `Location`. Within many of these components reside subcomponents to handle both common and local variations types of customization.

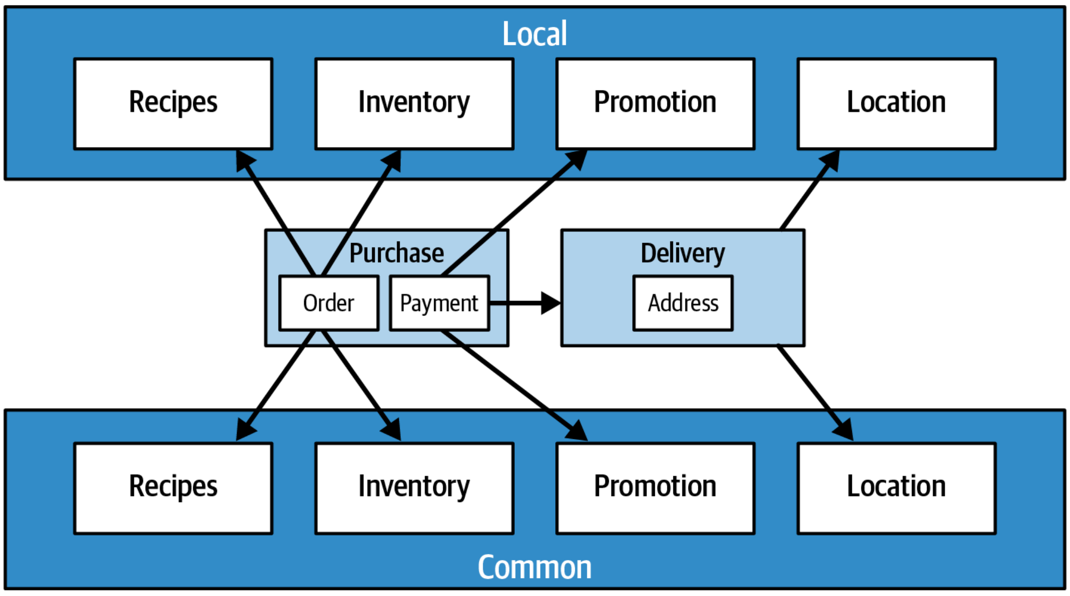

Figure 9-6. A technically partitioned design for Silicon Sandwiches

An alternative design isolates the common and local parts into their own partitions, as illustrated in Figure 9-6. `Common` and `Local` represent top-level components, while `Purchase` and `Delivery` remain to handle the workflow.

Which is better? It depends! Each kind of partitioning offers different advantages and drawbacks.

Domain partitioning

Domain-partitioned architectures separate top-level components by workflows and/or domains.

Advantages

- Modeled more closely on how the business functions rather than on an implementation detail

- Easier to build cross-functional teams around domains
- Aligns more closely to the modular monolith and microservices architecture styles
- Message flow matches the problem domain
- Easy to migrate data and components to a distributed architecture

Disadvantage

- Customization code appears in multiple places

Technical partitioning

Technically partitioned architectures separate top-level components based on technical capabilities rather than discrete workflows. This may manifest as layers inspired by Model-View-Controller separation or some other ad hoc technical partitioning. The architecture pictured in Figure 9-6 separates its components based on customization.

Advantages

- Clearly separates customization code.
- Aligns more closely to the layered architecture pattern.

Disadvantages

- Higher degree of global coupling. Changes to the `Common` or `Local` component will likely affect all the other components.
- Developers may have to duplicate domain concepts in both the Common and Local layers.
- Typically, higher coupling at the data level. In a system like this, the application architects and the data architects would likely collaborate to create a single database, including customization and domains. That in turn would create difficulties in untangling the data relationships later, if the architects eventually want to migrate this architecture to a distributed system. Many other factors contribute to choosing an architecture style, as we cover in Part II.

Monolithic Versus Distributed Architectures

As you learned in Part I, architecture styles can be classified into two main types: *monolithic* (single deployment unit of all code) and *distributed* (multiple deployment units connected through remote access protocols). While no classification scheme is perfect, distributed architectures all share a common set of challenges and issues not found in the monolithic architecture styles, making this classification scheme a good separation between the various architecture styles.

In Part II of this book, we describe the following architecture styles in detail:

Monolithic

- Layered architecture (Chapter 10)
- Pipeline architecture (Chapter 12)
- Microkernel architecture (Chapter 13)

Distributed

- Service-based architecture (Chapter 14)
- Event-driven architecture (Chapter 15)
- Space-based architecture (Chapter 16)
- Service-oriented architecture (Chapter 17)
- Microservices architecture (Chapter 18)

Distributed architecture styles, although much more powerful in terms of performance, scalability, and availability than monolithic architecture styles, have significant trade-offs. The first group of issues facing all distributed architectures are described in the "fallacies of distributed computing", first listed by L. Peter Deutsch and other colleagues from Sun Microsystems in 1994. A *fallacy* is something false that someone believes or assumes to be true. All eight of the fallacies of distributed computing apply to distributed architectures today. The following sections describe each fallacy.

Fallacy #1: The Network Is Reliable

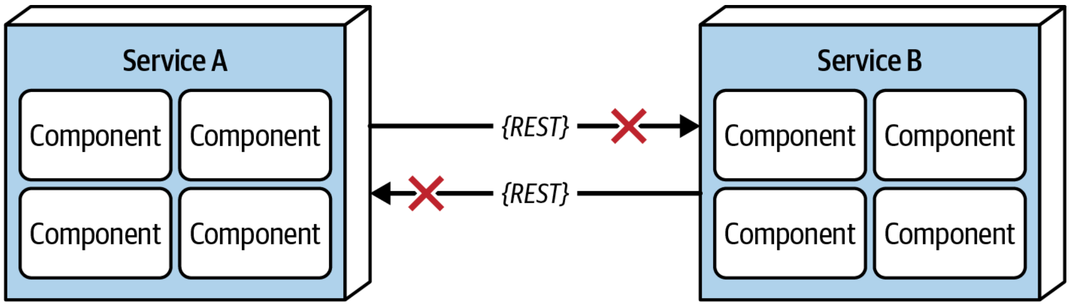

Figure 9-7. The network is not reliable

Developers and architects alike assume that the network is reliable, but it is not. While networks have become more reliable over time, the fact of the matter is that networks still remain generally unreliable. This is significant for all distributed architecture styles, because they rely on the network for communicating to, from, and between services. As illustrated in Figure 9-7, `Service B` may be totally healthy, but `Service A` cannot reach it due to a network problem. Even worse, `Service A` might request that `Service B` process some data, but receive no response because of a network issue. This is why things like timeouts and circuit breakers exist between services. The more a system relies on the network (as microservices architectures notably do), the more potential it has to become unreliable.

Fallacy #2: Latency Is Zero

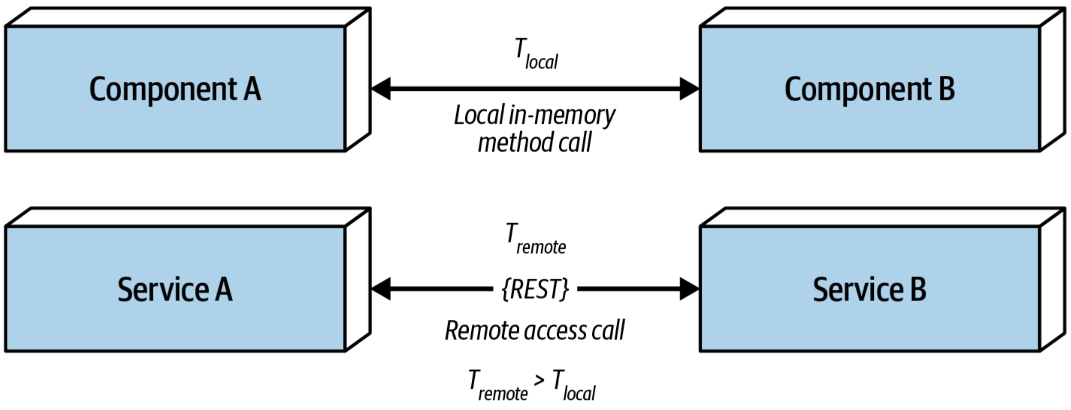

Figure 9-8. Latency is not zero

As Figure 9-8 shows, when a local call is made to another component via a method or function call, the time to access that component (`t_local`) is measured in nanoseconds or microseconds. However, when that same call is made through a remote access protocol (such as REST, messaging, or RPC), the time (`t_remote`) is measured in milliseconds and thus will always be greater than `t_local`. Latency, in any distributed architecture, is not zero—yet most architects ignore this fallacy, insisting that they have fast networks. Ask yourself: do you know what the average round-trip latency is for a RESTful call in your production environment? Is it 60 milliseconds? Is it 500 milliseconds?

When considering using any distributed architecture, particularly microservices, architects must know this latency average. It is the only way of determining whether a distributed architecture is feasible, due to the fine-grained nature of the services and the amount of communication between them.

For example, say we assume an average of 100 ms of latency per request. Chaining service calls together to perform a particular business function adds 1,000 ms to the request! Knowing the average latency is important, but knowing the 95th to 99th percentile latency is even more important. While the system's average latency might be only 60 ms (which is good), the 95th percentile might be 400 ms! It's usually this "long tail" latency that will kill performance in a distributed architecture. In most cases, a network administrator can provide latency values (see "Fallacy #6: There Is Only One Administrator").

Fallacy #3: Bandwidth Is Infinite

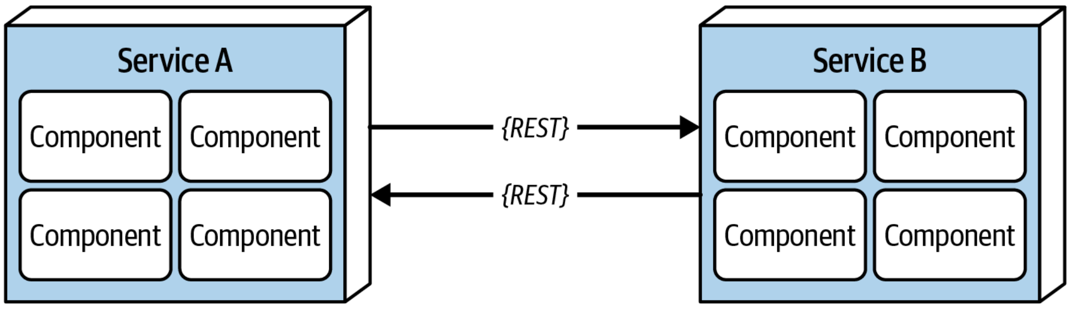

Figure 9-9. Bandwidth is not infinite

Bandwidth is usually not a concern in monolithic architectures, because once a business request goes into a monolith, little or no bandwidth is required to process it. However, as shown in Figure 9-9, once a system is broken apart into smaller deployment units (services) in a distributed architecture, such as microservices, communication to and between these services uses significant bandwidth. This slows the network, impacting latency (fallacy #2) and reliability (fallacy #1).

To illustrate the importance of this fallacy, consider the two services shown in Figure 9-9. Let's say `Service A` manages the wish list items for the website, and `Service B` manages the customer profiles. Whenever a request for a wish list comes into `Service A`, `Service A` needs the customer name for the response contract for the wish list. To get the name, it must make an interservice call to `Service B`. `Service B` returns 45 attributes, totaling 500 KB, to `Service A`, which only needs the name (200 bytes). This may not sound significant, but requests for the wish list items happen about 2,000 times a second. This means that `Service A` is calling `Service B` 2,000 times a second. At 500 KB for each request, *each* interservice call takes 1 GBps of bandwidth!

This form of coupling, called *stamp coupling*, consumes significant amounts of bandwidth in distributed architectures. If `Service B` were to pass back *only* the 200 bytes of data that `Service A` needs, it would use only 400 Kbps in total.

Stamp coupling can be resolved by:

- Creating private RESTful API endpoints
- Using field selectors in contracts
- Using GraphQL to decouple contracts
- Using value-driven contracts with consumer-driven contracts
- Using internal messaging endpoints

Regardless of the technique used, the best way to address this fallacy in a distributed architecture is to ensure that services or systems transmit only the necessary data.

Fallacy #4: The Network Is Secure

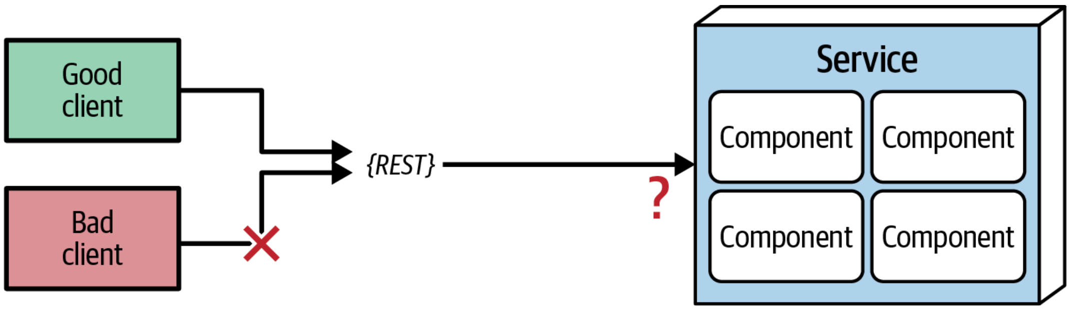

Figure 9-10. The network is not secure

Most architects and developers get so comfortable using virtual private networks (VPNs), trusted networks, and firewalls that they tend to forget about this fallacy of distributed computing—but *the network is not secure*. As shown in Figure 9-10, each and every endpoint to each distributed deployment unit must be secured against unknown or bad requests. The surface area for threats and attacks increases by magnitudes when moving from a monolithic to a distributed architecture, making security much more challenging. Securing every endpoint, even in interservice communication, is another reason performance tends to be slower in synchronous, highly distributed architecture styles, such as microservices and service-based architectures.

Fallacy #5: The Topology Never Changes

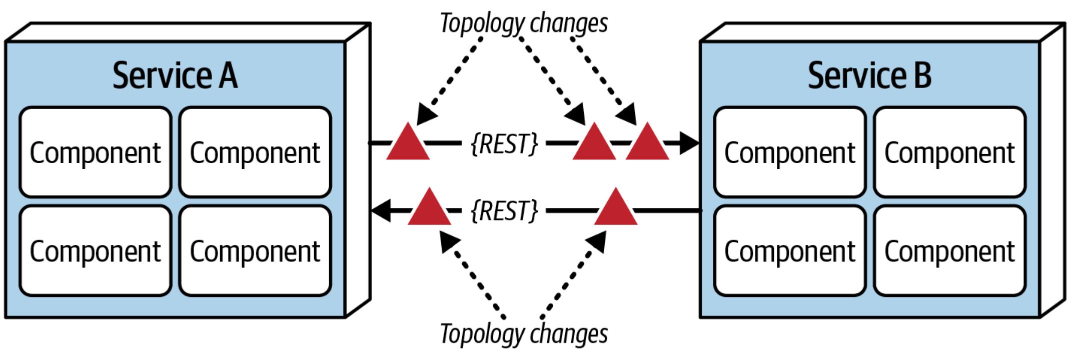

Figure 9-11. The network topology always changes

Fallacy #5, as shown in Figure 9-11, refers to the overall network topology, including all of its routers, hubs, switches, firewalls, networks, and appliances. Architects assume that the topology is fixed and never changes. *Of course it changes.* It changes all the time. What is the significance of this fallacy?

Suppose you come into work on a Monday morning and everyone is running around like crazy, because services keep timing out in production. You work with the teams, frantically trying to figure out why this is happening. No new services were deployed over the weekend. What could it be? After several hours, you discover that a supposedly “minor” network upgrade at two o’clock that morning invalidated all of the system’s latency assumptions, triggering timeouts and circuit breakers.

Architects must be in constant communication with operations and network administrators about what is changing and when so that they can make adjustments to avoid such surprises. This may seem obvious and easy, but it is neither. As a matter of fact, this fallacy leads directly to the next fallacy.

Fallacy #6: There Is Only One Administrator

Figure 9-12. There are many network administrators, not just one

Architects fall into this fallacy all the time: assuming they only need to collaborate and communicate with one administrator. As Figure 9-12 shows, there are dozens of network administrators in a typical large company. With whom should the architect talk about latency or topology changes? This fallacy points to the complexity of distributed architecture and the amount of coordination that must happen to get everything working correctly. A monolithic application, with its single deployment unit, doesn't require this level of communication and collaboration.

Fallacy #7: Transport Cost Is Zero

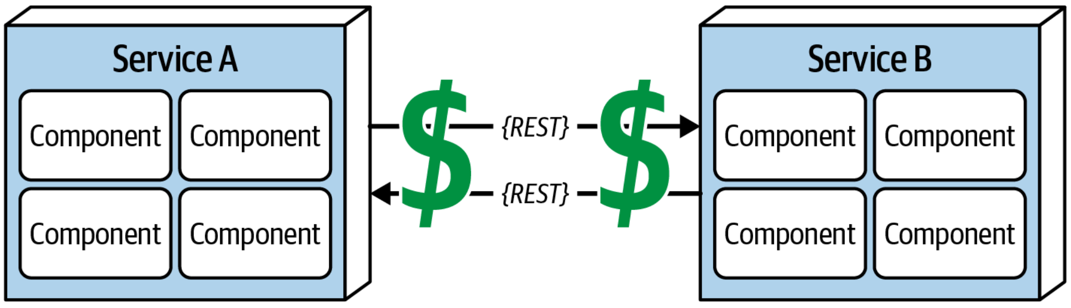

Figure 9-13. Remote access costs money

Many software architects confuse this fallacy, shown in Figure 9-13, with fallacy #2 (latency is zero). *Transport cost* here refers not to latency, but to the actual *monetary cost* of making a "simple RESTful call." Architects incorrectly assume that the necessary and sufficient infrastructure is already in place for making a simple RESTful call or breaking apart a monolithic application. *It is usually not.* Distributed architectures cost significantly more than monolithic architectures, primarily due to increased needs for hardware, servers, gateways, firewalls, new subnets, proxies, and so on.

We encourage architects embarking on a distributed architecture to analyze their current server and network topology with regard to capacity, bandwidth, latency, and security zones, to avoid getting surprised by this fallacy.

Fallacy #8: The Network Is Homogeneous

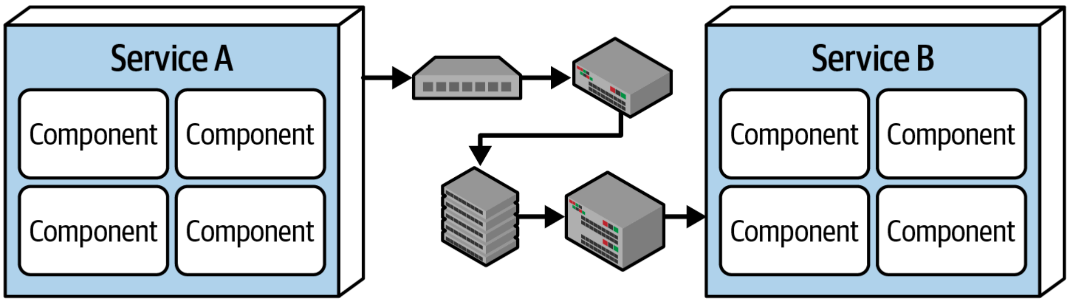

Figure 9-14. The network is not homogeneous

Most architects and developers assume that a network is homogeneous, as shown in Figure 9-14—that it's made up of network hardware from only one vendor. Nothing could be further from the truth. Most companies' infrastructures have multiple network-hardware vendors.

So what? The significance of this fallacy is that not all of those heterogeneous hardware vendors play together well. Does Juniper Networks hardware integrate seamlessly with Cisco Systems hardware? Most of it works, and networking standards have evolved over the years, making this less of an issue. However, not all situations, loads, and circumstances have been fully tested, so network packets do occasionally get lost. This in turn affects network reliability and assumptions and assertions about latency and bandwidth. In other words, this fallacy ties back into all of the other fallacies, forming an endless loop of confusion and frustration when dealing with networks (which is inescapable when using distributed architectures).

The Other Fallacies

The eight fallacies previously listed form a famous set of observations—every architect learns them either from Deutsch's list or the hard way, one by one, over the course of their career. The authors have also learned some painful, near-universal lessons that we offer as an extension to that famous list.

Fallacy #9. Versioning is easy

When two services need to communicate, they pass information in a *contract*, which includes information required for the communication. Often, a service's internal implementation evolves over time, changing fields that the service accepts and passes to other services. One way to solve this problem is to use versioning for the contract—that is, create different versions for the old and new contracts including different sets of information. However, this seemingly simple decision leads to a host of trade-offs:

- Should the team version at the individual service level or for the whole system?
- How far should the versioning reach? What portion of the architecture will need to support it?

- How many versions should the team support at any given time? (Some teams accidentally find themselves honoring dozens of different versions for different purposes.)

- Should the team deprecate older versions at the system level or service by service?

While versioning is a reasonable approach for evolving communication between services, it has a host of trade-offs that architects should anticipate.

Fallacy #10. Compensating updates always work

Compensating updates is an architectural pattern in which some mechanism (like an `Orchestrator` service) makes sure that several related services all update jointly. If they don't, the orchestrator reverses the update. The *compensating update* is from the orchestrator that issues a reversing operation to put the state back to what it was before.

This is a common pattern that most architects blithely assume always works… but it doesn't. What happens if the compensating update fails? When architects demonstrate how complex interactions in distributed architectures like microservices work, they must also show how compensating updates work. Thus, architects designing transactional workflows in microservices should accommodate the "normal" compensation workflow, but must also consider how to recover if the update and the compensating update (or a portion of it) *both* fail.

Fallacy #11. Observability is optional (for distributed architectures)

A common architectural characteristic for architects to prioritize in distributed architectures is *observability*: the ability to observe each service's interactions with other services and the ecosystem, as captured through monitors or logs. While logging is useful in monolithic architectures, it is *critical* in distributed architectures, which offer many communication failure modes that are hard to debug without comprehensive interaction logs.

Team Topologies and Architecture

Architects and teams have done lots of research on the intersection of architecture and team topologies, above and beyond the implications of architecture partitioning, which we previously discussed. The very influential book *Team Topologies* by Matthew Skelton and Manuel Pais (IT Revolution Press, 2019) defines several team types that intersect with software architecture:

Stream-aligned teams

> In team topologies terminology, a *stream* is a stream of work scoped to a particular business domain or capability. *Stream-aligned teams* focus narrowly on a single line of work, such as a product, service, or specific set of features.
>
> The goal of stream-aligned teams is to move as quickly as possible because they are delivering discrete value to the organization. Consequently, the other team types are designed to reduce any friction that could impede the stream-aligned teams.

Enabling teams

> An *enabling team* bridges a gap in some capability, providing a place for necessary research, learning, and other tasks that are important but not urgent. They supply knowledge and resources from specialized domains to support stream-aligned teams. Good enabling teams are highly collaborative and proactive.

Complicated-subsystem teams

> Many systems include highly specialized systems or parts that require similarly specialized skills. Members of *complicated-subsystem teams* fully understand a complex subsystem or domain and can help a stream-aligned team apply it. Their goal is to reduce other teams' cognitive load.

Platform teams

> A *platform team* provides internal services and building blocks for solutions. As defined by Evan Botcher, a platform is
>
> > a foundation of self-service APIs, tools, services, knowledge and support which are arranged as a compelling internal product. Autonomous delivery teams can make use of the platform to deliver product features at a higher pace, with reduced coordination.
>
> Platform teams support the other teams, attempting to remove needless friction while providing necessary governance around concerns such as quality and security.

On to Specific Styles

Architects need to understand a number of different architecture styles before they can perform trade-off analysis. Each style supports a different set of architectural characteristics—each one has a "sweet spot" that it handles best. By learning the different styles and their underlying philosophies, an architect better understands when each one works best (or least worst, anyway).