

Chapter 18. Microservices Architecture

Microservices is an extremely popular architecture style that has gained significant momentum in recent years. In this chapter, we provide an overview of the important characteristics that set this architecture apart, both topologically and philosophically.

Most architecture styles are named after they are created by architects who notice that a particular pattern keeps reappearing. There is no secret group of architects who decide what the next big movement will be—architects make decisions as the software development ecosystem shifts and changes, and of the most common ways of dealing with and profiting from those shifts, those that emerge as the best become architecture styles that others emulate.

Microservices differs in this regard—it was named fairly early in its usage. Martin Fowler and James Lewis popularized it in a famous 2014 [blog post](#) in which they recognized and delineated the characteristics of this relatively new architectural style. Their blog post shaped the definition of the architecture and helped curious architects understand the underlying philosophy.

Microservices is heavily inspired by the ideas in domain-driven design (DDD), a logical design process for software projects. One concept in particular from DDD, the *bounded context*, decidedly inspired microservices. The concept of a bounded context represents a decoupling style (as discussed previously in [“Domain-Driven Design’s Bounded Context”](#) in [Chapter 7](#)), which is why microservices is sometimes called a “share nothing” architecture.

When a developer defines a domain, that domain includes many entities and behaviors, identified in artifacts such as code and database schemas. For example, an application might have a domain called `CatalogCheckout` that includes notions such as catalog items, customers, and payment. In a traditional monolithic architecture, developers would share many of these concepts, building reusable classes and linked databases. Within a bounded context, internal parts such as code and data schemas can be coupled together to produce work, but they are *never* coupled to anything outside the bounded context, such as a database or class definition from another bounded context. This allows each context to define only what it needs rather than accommodating other constituents, limiting reuse between bounded contexts.

While reuse is generally beneficial, remember the First Law of Software Architecture? Everything’s a trade-off. The downside of reuse is that achieving it usually requires increasing the system’s coupling, either by inheritance or composition.

If the architect’s goal is a highly decoupled system—the primary goal of microservices—then they will favor duplication over reuse, physically modeling the logical notion of bounded context to include a service and its corresponding data.

Topology

The basic topology of microservices is shown in [Figure 18-1](#). Due to its single-purpose nature, services in this architecture style are much smaller than in other distributed architectures, such as orchestration-driven SOA ([Chapter 17](#)), event-driven architecture ([Chapter 15](#)), and service-based architecture ([Chapter 14](#)). Architects expect each service to include all parts necessary for it to operate independently, including databases and other dependent components.

Microservices is a *distributed* architecture style: each service runs in its own process, in either a virtual machine or container. Decoupling the services to this degree allows for a simple solution to a common problem in architectures that heavily feature multitenant infrastructure for hosting applications. For example, when the system is using an application server to manage multiple running applications, the application server allows operational reuse of network bandwidth, memory, and disk space, among other things. However, if all the supported applications continue to grow, eventually some resource on the shared infrastructure will become constrained.

Another problem concerns improper isolation between shared applications. Separating each service into its own process solves all the problems brought on by sharing. Before the evolution of freely available open source operating systems and automated machine provisioning, it was impractical for each domain to have its own infrastructure. Now, however, with cloud resources and container technology (see “[Cloud Considerations](#)”), teams can reap the benefits of extreme decoupling, both at the domain and operational level.

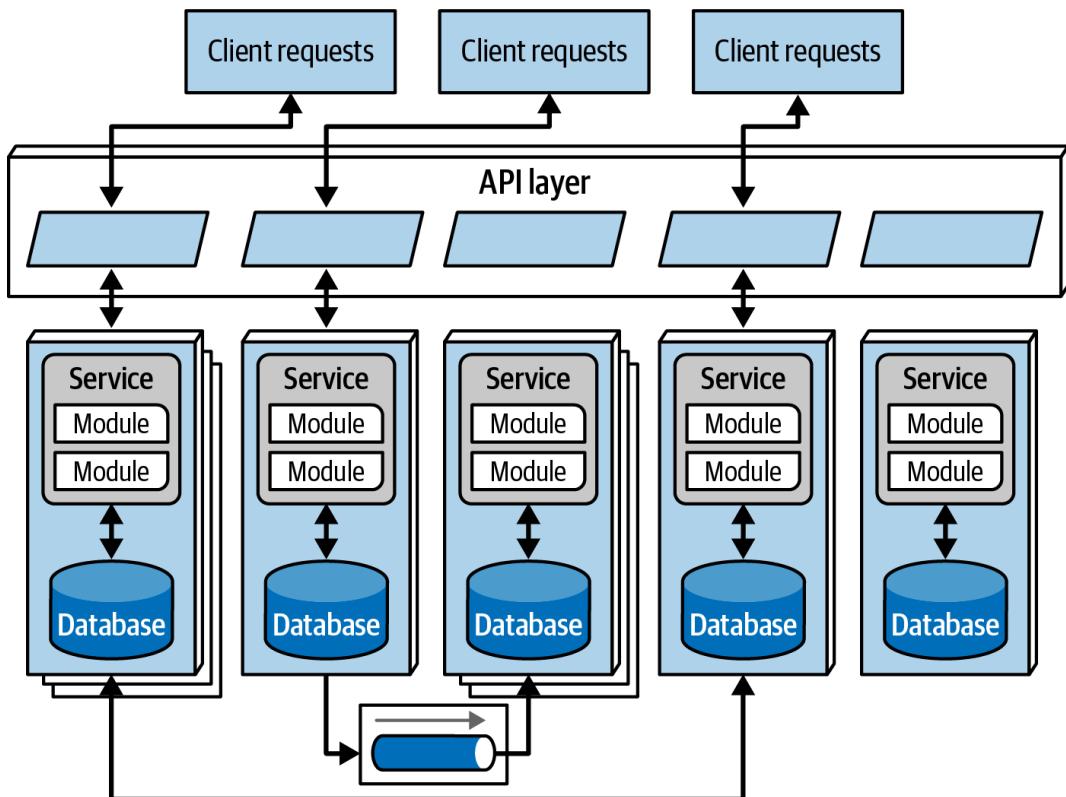


Figure 18-1. The topology of the microservices architecture style

Performance is often the downside of microservices’ distributed nature. Network calls take much longer than method calls, and security verifications at every endpoint add additional processing time, requiring architects to think carefully about the implications of granularity.

Because microservices is a distributed architecture, experienced architects advise against using transactions across service boundaries. Determining the granularity of services is really the key to success in this architecture.

Style Specifics

The following sections, while not exhaustive, describe some of the important aspects of the microservices topology, including some unique elements that are unique to microservices.

Bounded Context

Let's dive more deeply into the notion of *bounded context*, which we mentioned as the driving philosophy of microservices. Each service models a particular function, subdomain, or workflow. Each bounded context, then, includes everything necessary to operate within that function or subdomain—including services consisting of logical components and classes, database schemas, and the corresponding database the service requires to carry out its functions. Each service is meant to represent a particular subdomain or function.

This philosophy drives many of the decisions architects make within microservices. For example, in a monolith, it is common for developers to share common classes, such as `Address`, between disparate parts of the application. However, because microservices architectures try to avoid coupling, an architect building in this architecture style would use duplication rather than coupling to keep *all* code within the bounded context of that function or subdomain.

Microservices takes the concept of a domain-partitioned architecture to the extreme; in many ways, this architecture physically embodies the logical concepts in domain-driven design.

Granularity

Architects designing microservices often struggle to find the correct level of granularity, and end up making their services too small by taking the term *micro* literally. Then they have to build communication links back between the services to do useful work, which negates the point and results in a Big Ball of Distributed Mud.

The term *microservice* is a label, not a description.

Martin Fowler

In other words, the originators of the term needed to call this new style *something*, and they chose *microservices* to contrast it with the dominant architecture style at the time (circa 2007), the service-oriented architecture, which could have been called “gigantic services.” However, many developers take the term *microservices* as a commandment, not a description, and create services that are too fine-grained.

The purpose of service boundaries in microservices is to capture a domain or workflow. In some applications, those natural boundaries might be large for parts of the system, simply because some business processes are more highly coupled than others. Here are some guidelines architects can use to help find the appropriate boundaries:

Purpose

The most obvious boundary relies on the inspiration for the architecture style: the problem domain. Ideally, each microservice should be functionally cohesive, contributing one significant behavior on behalf of the overall application.

Transactions

Bounded contexts are business workflows, and often the entities that need to cooperate in a transaction suggest a good service boundary. Because transactions cause issues in distributed architectures, designing systems with the goal of avoiding them tends to result in better designs.

Choreography

A set of services offers excellent domain isolation but requires extensive communication to function. The architect may consider bundling these services back into a larger service to avoid the communication overhead.

Iteration is the only way to ensure good service design. Architects rarely discover the perfect granularity level, data dependencies, and communication styles on their first pass: they iterate over the options to refine their designs, particularly as they learn more about the system and its business functionality.

Data Isolation

Another requirement of microservices, also driven by the bounded-context concept, is *data isolation*. Many architecture styles use a single database for persistence. However, microservices tries to avoid *all* kinds of coupling, *including* using shared schemas and databases as integration points.

Data isolation is another factor to consider when looking at service granularity. Be wary of the Entity Trap (discussed in “[The Entity Trap](#)”)—don’t simply model services to resemble single entities in a database. Architects are accustomed to using relational databases to unify values within a system, creating a single source of truth, but that’s no longer an option when you’re distributing data across the architecture. So every architect must decide how they want to handle this problem: either identifying one domain as the source of truth for some fact and coordinating with it to retrieve values, or distributing information through database replication or caching.

While this level of data isolation creates headaches, it also provides opportunities. Now that they aren’t forced to unify around a single database, each team can choose the most appropriate database technology for their service’s budget, storage structure type, operational characteristics, process characteristics, and so on. Another advantage of a highly decoupled system is that any team can change course and choose a more suitable database (or other dependency) without affecting other teams, which aren’t allowed to couple to implementation details. (We cover data isolation and database considerations in more detail in [“Data Topologies”](#).)

API Layer

Most microservices architectures include an API layer (usually called an *API Gateway*) between the consumers of the system (either user interfaces or calls from other systems) and the microservices. The API layer can be implemented as a simple reverse-proxy or a more sophisticated gateway containing cross-cutting

concerns such as security, naming services, and so on (these are covered in more detail in “[Operational Reuse](#)”).

While API layers have many uses, to stay true to the underlying philosophy of this architecture, they should not be used as mediators or orchestrators. All interesting business logic in this architecture should reside inside a bounded context, and putting orchestration or other business logic into a mediator violates that rule. Architects typically use mediators in technically partitioned architectures, whereas microservices is firmly domain partitioned.

Tip

When using an API layer in a microservices architecture, only include request routing and cross-cutting concerns, such as security, monitoring, logging, and so on. Be careful to avoid putting any business-related logic within the API layer.

Operational Reuse

Given that microservices prefers duplication to coupling, how do architects handle the parts of this architecture that really do benefit from coupling, such as operational concerns like monitoring, logging, and circuit breakers? Traditional service-oriented architecture philosophy was to reuse as much functionality as possible, domain and operational alike. In microservices, however, architects try to split these two concerns.

Once a team has built several microservices, its members start to realize that each microservice has common elements that benefit from similarity. For example, if an organization allows each service team to implement monitoring independently, how can it ensure that each team does so? And how does the organization handle concerns like upgrades—does each team become responsible for upgrading to the new version of the monitoring tool, and how long will that take? The *Sidecar* pattern offers a solution to this problem ([Figure 18-2](#)).

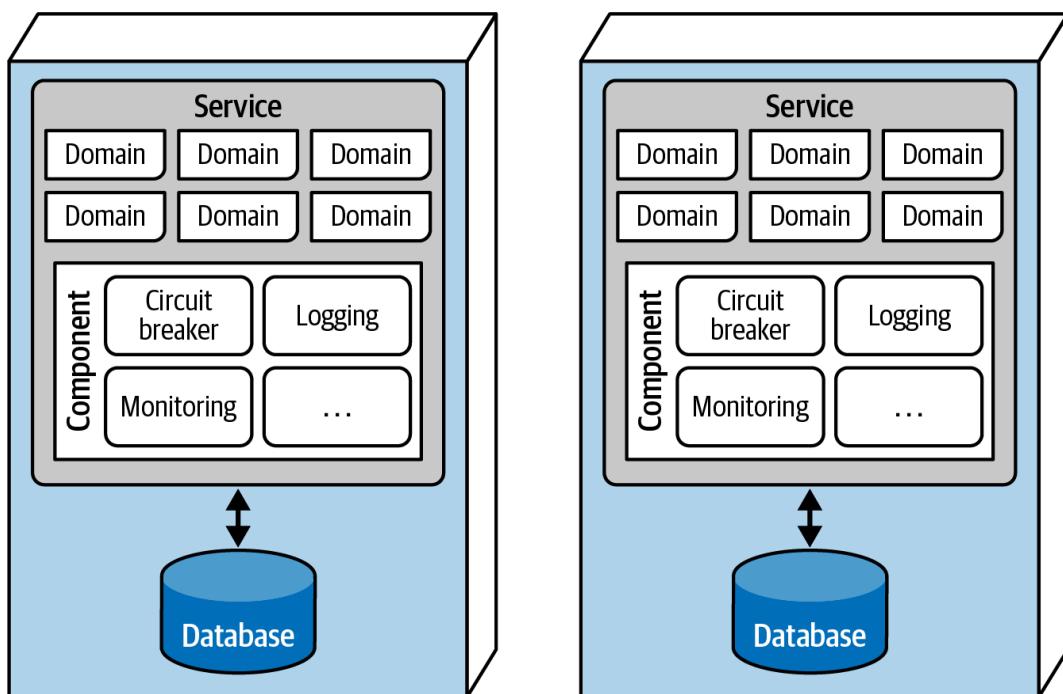


Figure 18-2. The Sidecar pattern in microservices

In [Figure 18-2](#), the common operational concerns (circuit breaker, logging, monitoring) appear within each service as separate components, each of which can be owned by individual teams or by a shared infrastructure team. The `Sidecar` component handles all the operational concerns that benefit from coupling, so when it comes time to upgrade the monitoring tool, the shared infrastructure team can update the sidecar, and each microservice will receive the new functionality (see [“Team Topology Considerations”](#)).

When each service includes a common `Sidecar` component, the architect can build a *service mesh* to allow teams unified control of these common concerns across the architecture. The sidecar components connect to form a consistent operational interface across all microservices, as shown in [Figure 18-3](#).

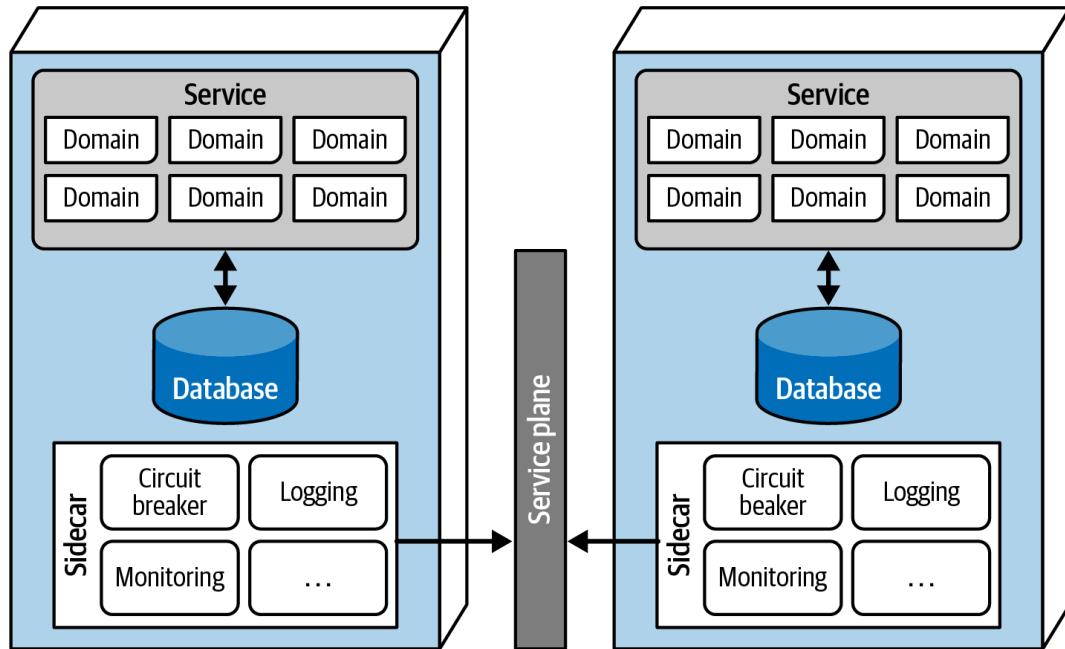


Figure 18-3. The service plane connects the sidecars in a service mesh

In [Figure 18-3](#), each sidecar wires into the *service plane*. A service plane is integration software (usually in the form of a product such as [Istio](#)) that connects each sidecar using a consistent interface, thus forming the service mesh.

Each service forms a node in the overall mesh, as shown in [Figure 18-4](#). The service mesh forms a console that allows teams to globally control operational coupling, such as monitoring levels, logging, and other cross-cutting operational concerns.

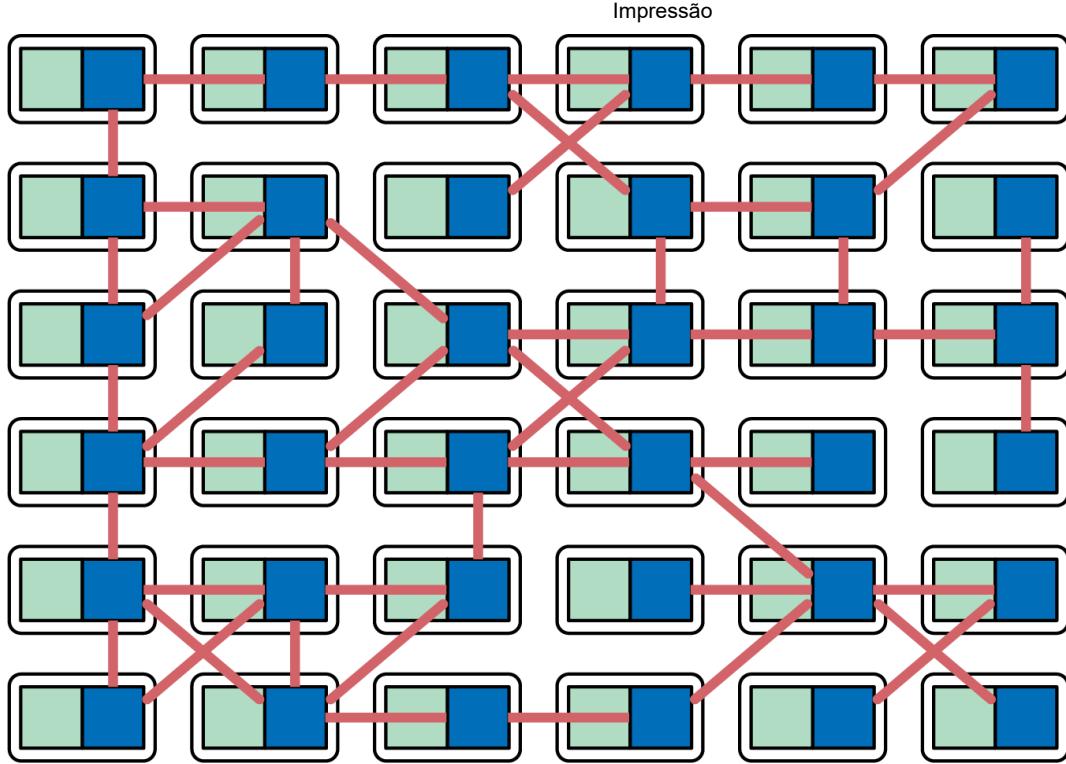


Figure 18-4. The service mesh forms a holistic view of the operational aspect of microservices

Architects use [service discovery](#) as a way to build elasticity into microservices architectures. *Service discovery* is a way of automatically detecting and locating services within a network. When a request comes in, rather than invoking a single service, it goes through a service discovery tool, which can monitor the number and frequency of requests and spin up new instances of services to handle scale or elasticity concerns. Architects often include service discovery in the service mesh, making it part of every microservice. The API layer is often used to host service discovery, allowing a single place for user interfaces or other calling systems to find and create services in an elastic, consistent way.

Frontends

Microservices favors decoupling, which would ideally encompass the user interfaces as well as backend concerns. In fact, the original vision for microservices included UI as part of the bounded context, faithful to the bounded-context principle in DDD. However, the practicalities of the partitioning required by web applications (and other external constraints) make that a difficult goal. That's why we commonly see two UI styles for microservices architectures.

The first style, shown in [Figure 18-5](#), is the *monolithic frontend*, which features a single UI that calls through the API layer to satisfy user requests. This frontend could be a rich desktop, mobile, or web application. For example, many web applications now use a JavaScript web framework to build a single UI.

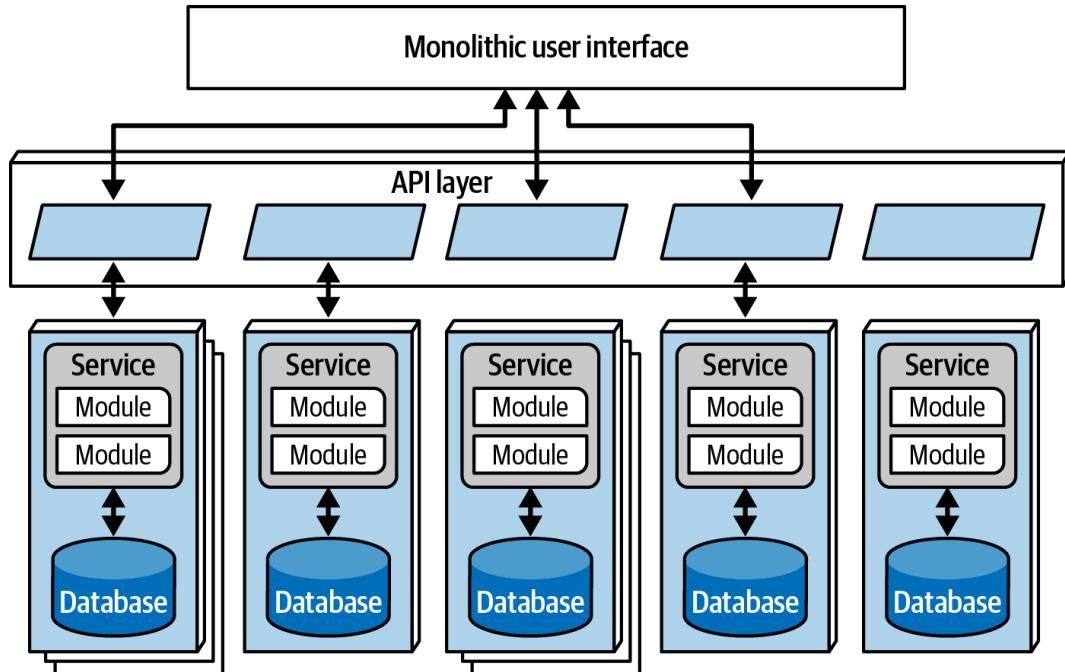


Figure 18-5. Microservices architecture with a monolithic user interface

The second UI option is *micro-frontends*, shown in [Figure 18-6](#).

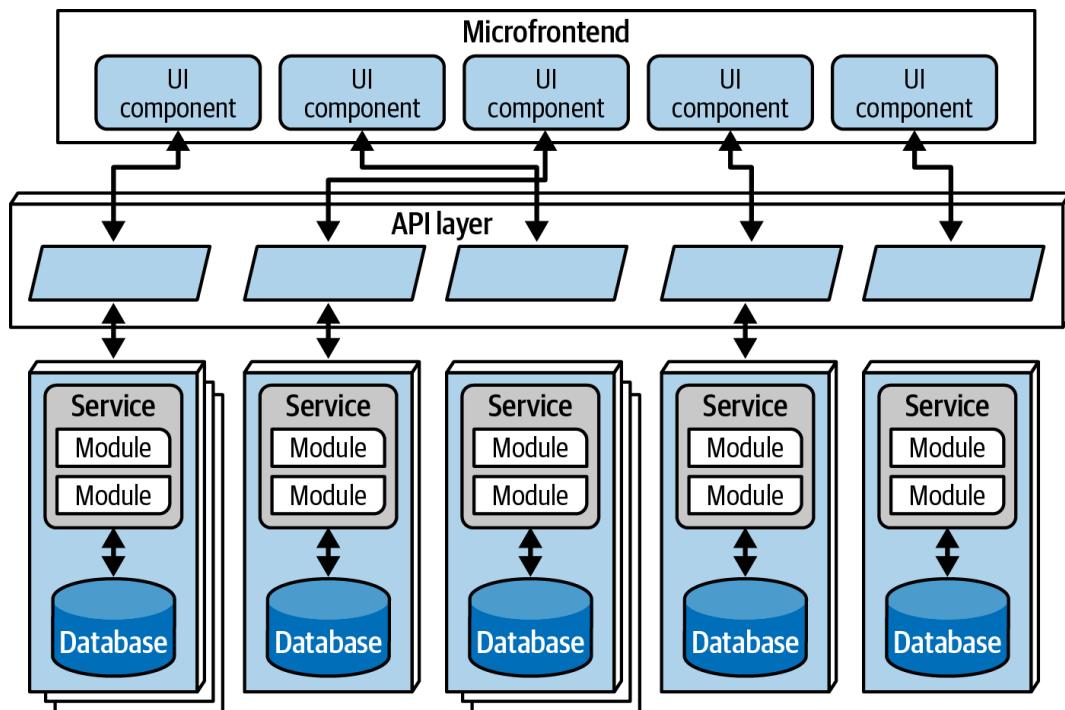


Figure 18-6. Micro-frontend pattern in microservices

The micro-frontend approach uses components at the UI level to create a synchronous level of granularity and isolation in the UI and the backend services, thus forming relationships between the UI components and corresponding backend services.

To learn more about micro-frontends, we highly recommend the book [*Building Micro-Frontends*](#), 2nd Edition, by Luca Mezzalira (O'Reilly, 2025).

Communication

In microservices, architects and developers struggle to find the appropriate service granularity, which affects both data isolation and communication. Finding the correct communication style helps teams keep services decoupled, yet still coordinate them in useful ways.

Fundamentally, architects must decide on *synchronous* or *asynchronous* communication. Synchronous communication requires the sender to wait for a response from the receiver. Microservices architectures typically utilize *protocol-aware heterogeneous interoperability* when communicating to and between services. Let's break that complicated term down into its parts to better understand what it means and why it's important:

Protocol-aware

Because microservices don't include a centralized integration hub, each service needs to know how to call other services. Thus, architects commonly standardize on *how* particular services call each other: a certain level of REST, message queues, and so on. That means that services must know (or discover) which protocol to use to call other services.

Heterogeneous

Because microservices is a distributed architecture, each service could be written in a different technology stack. *Heterogeneous* indicates that microservices fully supports polyglot environments, in which different services use different platforms.

Interoperability

Describes services calling one another. While architects in microservices try to discourage transactional method calls, services commonly call other services via the network to collaborate and exchange information.

Enforced Heterogeneity

A well-known architect who was a pioneer in the microservices style was the chief architect at a startup, building personal-information management software for mobile devices. Because mobile is such a fast-moving problem domain, the architect wanted to ensure that none of the development teams would accidentally create coupling points that could hinder their ability to move independently. It turned out that the teams had a wide mix of technical skills, so the architect mandated a new rule: each development team had to use a *different* technology stack. If one team was using Java and the other was using .NET, it would be impossible for them to share classes accidentally!

This approach is the polar opposite of most enterprise governance policies, which insist on standardizing on a single technology stack. In the microservices world, the goal isn't to create the most complex ecosystem possible, but to choose the correct scale of technology for the narrow scope of the problem. Not every service needs an industrial-strength relational database, and forcing one on a small team is more likely to slow them down than to benefit them. This concept leverages the highly decoupled nature of microservices.

For asynchronous communication, architects often use events and messages, similar to what we described in event-driven architecture in [Chapter 15](#).

Choreography and Orchestration

Choreography utilizes the same communication style as EDA. Choreographed architectures have no central coordinator, respecting the bounded context philosophy and making it natural to implement decoupled events between services.

In choreography, each service calls other services as needed, without a central mediator. For example, consider the scenario shown in [Figure 18-7](#). The user requests details about another user's wish list. Because the `CustomerWishList` service doesn't contain all the information it needs, it makes a call to `CustomerDemographics` to retrieve the missing information, then returns the result to the user.

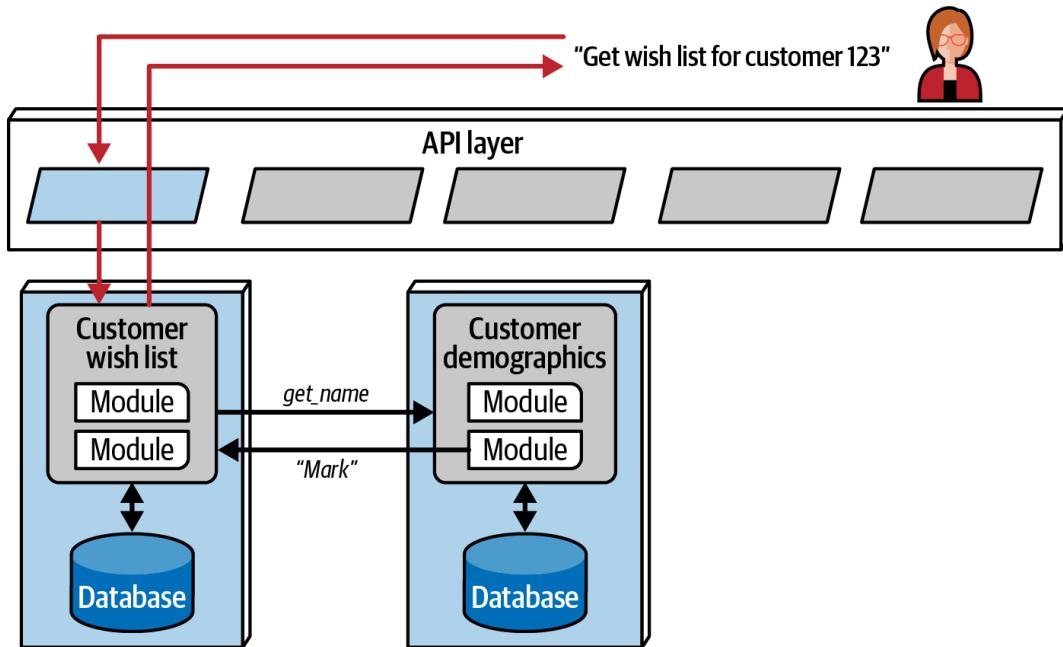


Figure 18-7. Using choreography in microservices to manage coordination

Because microservices architectures don't include a global mediator like other service-oriented architectures, if an architect needs to coordinate across several services, they can create their own localized mediator (usually called an *orchestration service*).

In [Figure 18-8](#), the developers create a service whose sole responsibility is coordinating calls. For instance, the user calls the `ReportCustomerInformation` mediator, which calls all necessary other services to get the requested information.

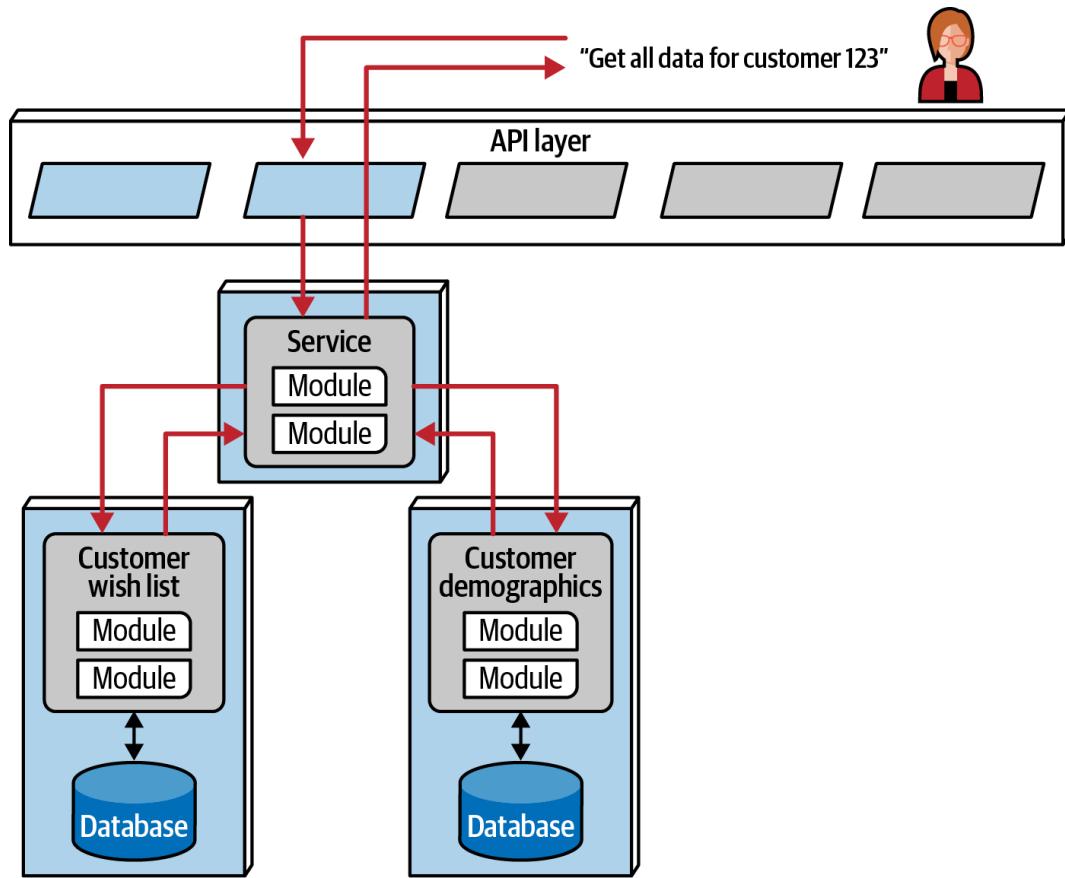


Figure 18-8. Using orchestration in microservices

The First Law of Software Architecture suggests that neither of these solutions is perfect—each has trade-offs. Choreography preserves the highly decoupled philosophy of microservices to reap its maximum benefits. However, it also makes common problems like error handling and coordination more complex.

Consider an example with a more complex workflow. In [Figure 18-9](#), the first service called must coordinate across a wide variety of other services, basically acting as a mediator in addition to its other domain responsibilities. This is called the *Front Controller* pattern, where a nominally choreographed service becomes a more complex mediator for some problem. The downside to this pattern is that the service taking on multiple roles adds complexity.

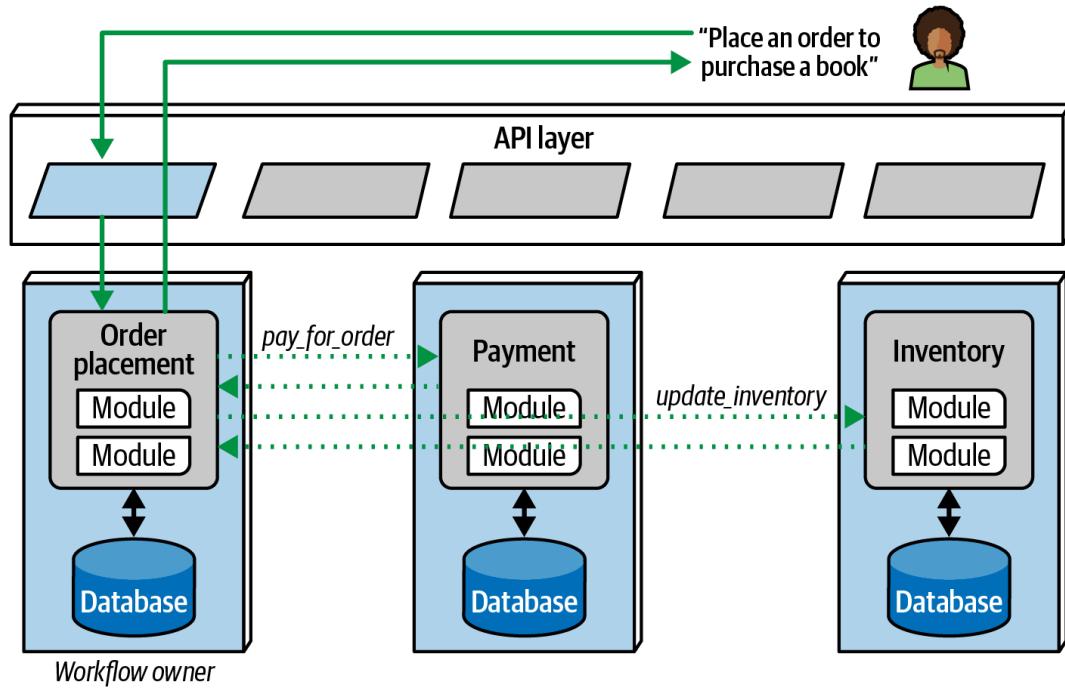


Figure 18-9. Using choreography for a complex business process

Alternatively, architects may choose to use orchestration for complex business processes, illustrated in [Figure 18-10](#). The architect builds a mediator service to coordinate the business workflow creating coupling between these services, but also allowing the architect to focus coordination into a single service, leaving the others less affected. Domain workflows are often inherently coupled, so the architect's job entails finding a way to represent that coupling that best supports the goals of both the domain and the architecture.

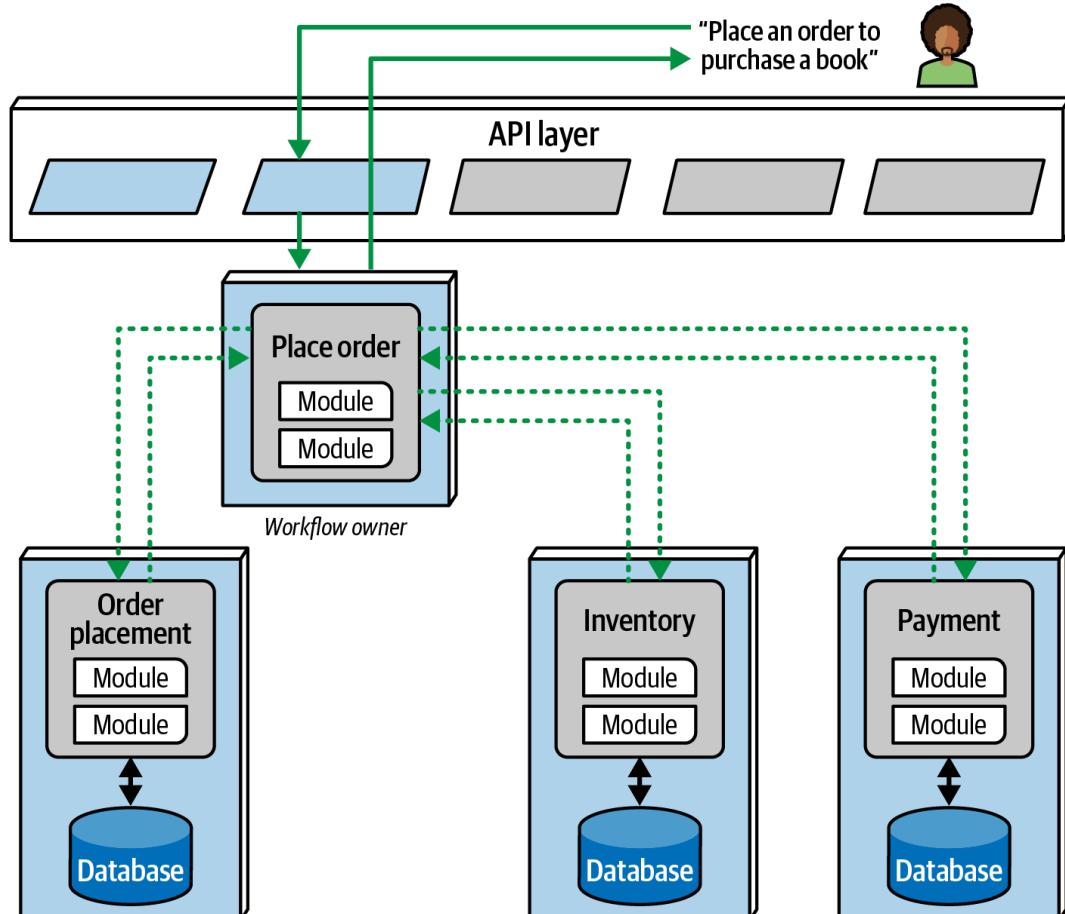


Figure 18-10. Using orchestration for a complex business process

Transactions and Sagas

Architects aspire to extreme decoupling in microservices, but often encounter the problem of how to coordinate transactions across services. Because microservices encourages the same level of decoupling in the databases as in the architecture, the atomicity that was trivial in monolithic applications becomes a problem in distributed ones.

Building transactions across service boundaries violates the core decoupling principle of microservices, and also creates the worst kind of dynamic connascence, *Connascence of Values* (see “[Connascence](#)”). The best advice we can offer architects who want to do transactions across services is: *don't!* Fix the service granularity instead. If you’re finding that you need to wire your microservices architecture together with transactions, that’s a sign that your design is too granular.

Tip

Try to avoid transactions that span multiple microservices—fix the service granularity instead!

In keeping with the “it depends” rule, there are always exceptions. For example, a situation could arise where two different services need vastly different architecture characteristics that require distinct service boundaries, yet still need transactional coordination. The architect in that case could, after considering the trade-offs carefully, leverage certain transactional patterns to orchestrate transactions.

Distributed transactions in microservices are usually handled by what is known as a *Saga* pattern. In literature, a *saga* is an epic story describing a long sequence of events that leads to some sort of heroic conclusion, hence the name of this transactional pattern.

In [Figure 18-11](#), a service acts as a mediator across multiple service calls to coordinate a transaction. The mediator calls each part of the transaction, records success or failure, and coordinates the results. If everything goes as planned, all the values in the services and their contained databases update synchronously. If there’s an error condition and one part of the transaction fails, the mediator must ensure that no part of the transaction succeeds. Consider the situation shown in [Figure 18-12](#).

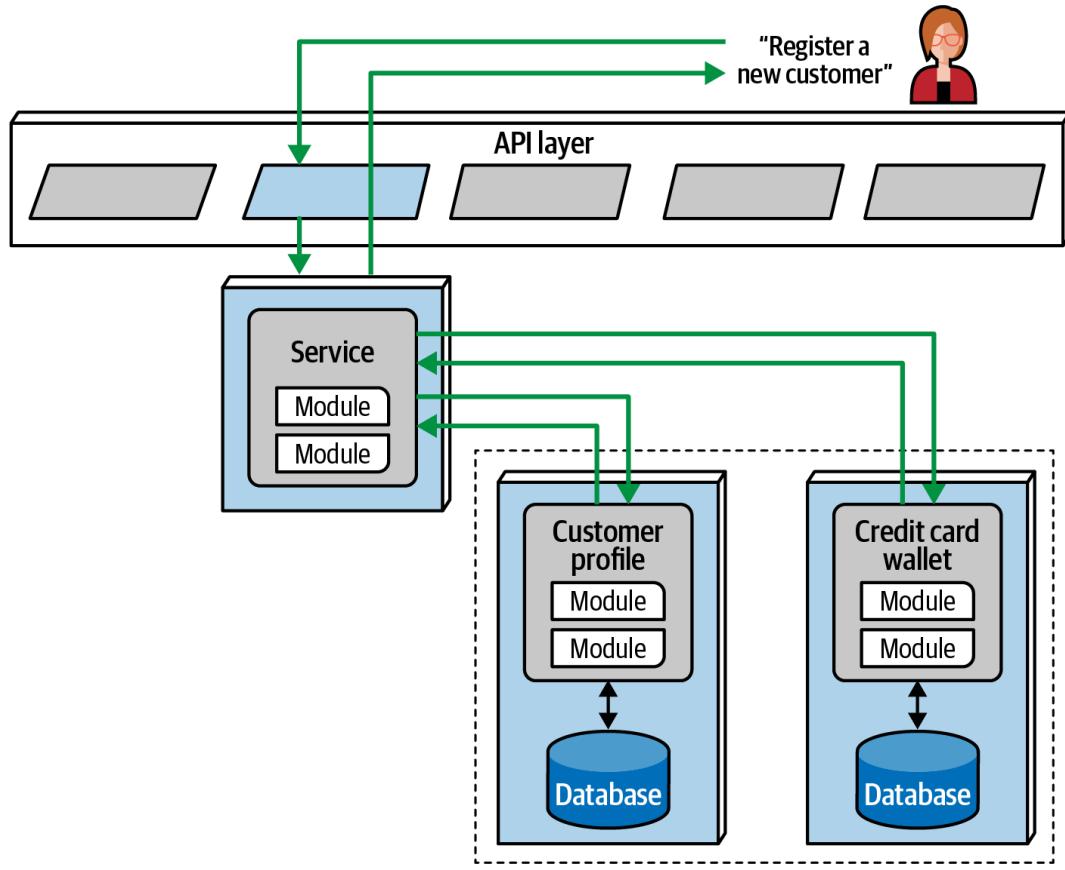


Figure 18-11. The Saga pattern in microservices architecture

If the first part of the transaction succeeds but the second part fails, the mediator must send a request to all other participating services of the transaction that were successful and tell them to undo the previous request. This style of transactional coordination is called a *compensating transaction framework*. Developers usually implement this pattern by having each request from the mediator enter a pending state until the mediator indicates overall success. However, juggling asynchronous requests can become complex, especially if new requests appear that are contingent on pending transactional state. Regardless of the protocol used, compensating transactions create a lot of coordination traffic at the network level.

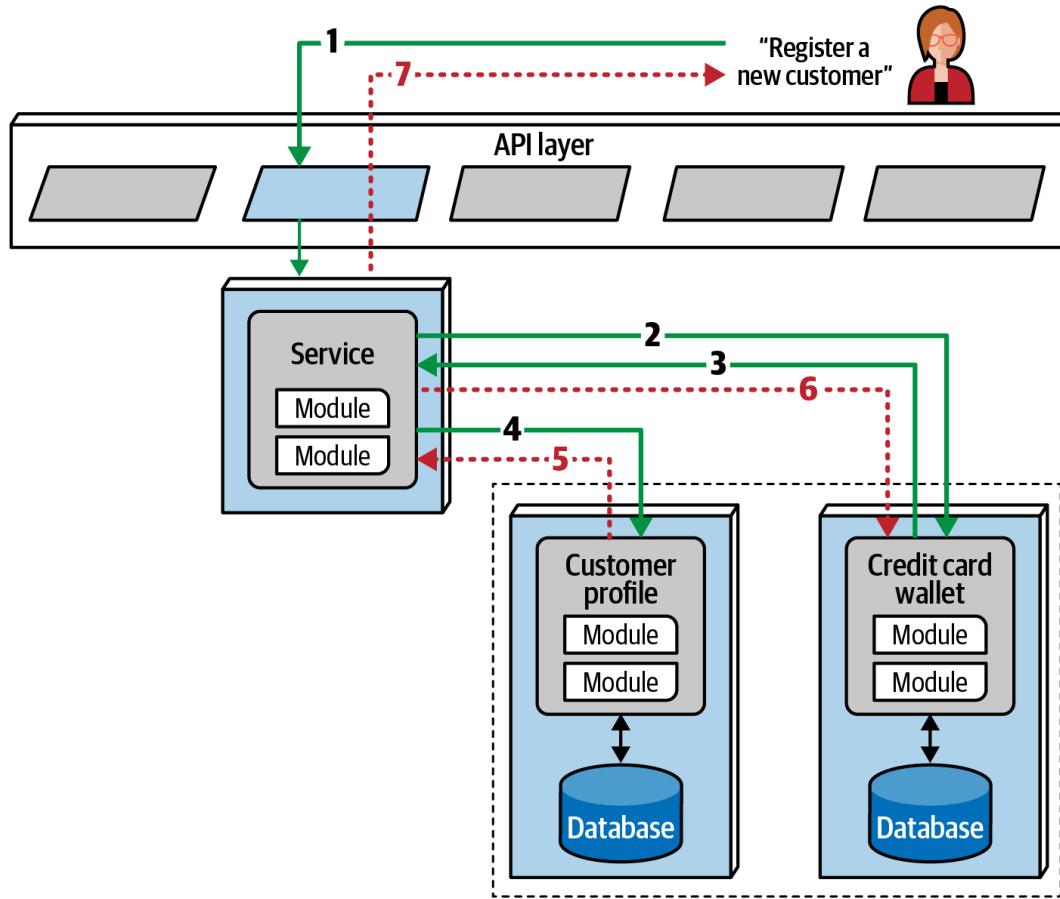


Figure 18-12. Saga pattern compensating transactions for error conditions

Tip

It's sometimes necessary for a few transactions to cross services; however, if it's the dominant feature of the architecture, then microservices is likely not the right choice!

Managing transactions in microservices is complicated, and you can dive much deeper. We've identified eight different transactional Saga patterns to solve a variety of scenarios, which you can find in Chapter 12 of our book [Software Architecture: The Hard Parts](#) (O'Reilly, 2021), coauthored with Pramod Sadalage and Zhamak Dehghani.

Data Topologies

As we've discussed throughout this chapter, data plays a critical role in microservices architectures. As a matter of fact, microservices is the only architectural style that *requires* architects to break apart data. While it's *possible* (if not always effective) to use a monolithic database in other distributed architectures, it's simply not an option in microservices. Neither are domain databases, as we discuss in "["Data Topologies"](#)" and also in "["Data Topologies"](#)". This is due to the fine-grained nature of microservices, bounded context, and the large number of services found in most microservices ecosystems.

To illustrate why a monolithic database isn't feasible in microservices, consider a scenario where 60 services share the same database. The first issue that arises will be about controlling change within the architecture. As [Figure 18-13](#) illustrates, changing the structure of that database (such as by changing a column name or dropping a table) would require corresponding changes to all 60 services using

that data. Imagine trying to coordinate the maintenance, testing, and release of five dozen separately deployed services while at the same time releasing the database changes! This task would be daunting, to say the least, and would likely end in disaster.

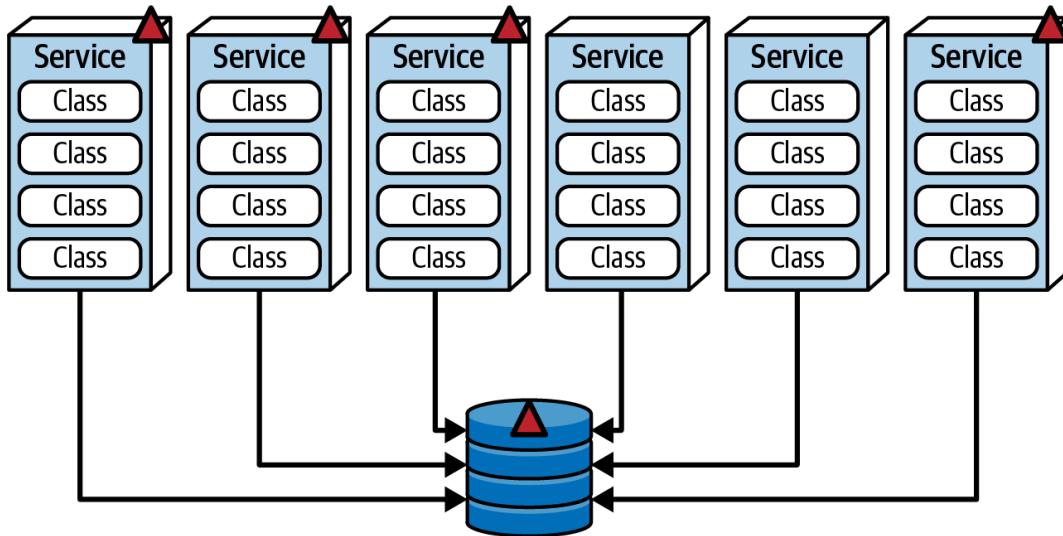


Figure 18-13. Controlling change with a monolithic database is a very challenging task

Perhaps the biggest issue with combining a monolithic data topology with microservices is that it breaks the entire notion of a physical bounded context. Recall that the bounded context includes *all* functionality required to execute a particular business function or subdomain, *including the database and corresponding data structures*. If every service shares the same database and data structures, the bounded context goes away.

Another issue with monolithic data is about scalability and elasticity. While many operational tools and products automatically monitor concurrent load and adjust the number of service instances to address increases, there aren't many databases that scale accordingly. This imbalance can cause overall responsiveness issues and request timeouts. Managing the database connections, which are typically located in each service *instance*, is another concern. As the number of services and service instances increase, the services can quickly run out of available database connections, resulting in further connection waits and request timeouts.

Last, if the database becomes unavailable due to a crash, planned maintenance, or backups, the entire microservices ecosystem goes down—as would any architectural style using a monolithic database. While not as extreme, a domain-based database topology can suffer from the same issues of scalability, database connection pool management, availability, and fault tolerance.

For these reasons, the standard database topology for microservices is the *Database-per-Service* pattern. With this database topology, each microservices owns its own data, which is contained as tables within a separate database or schema, as illustrated in [Figure 18-14](#).

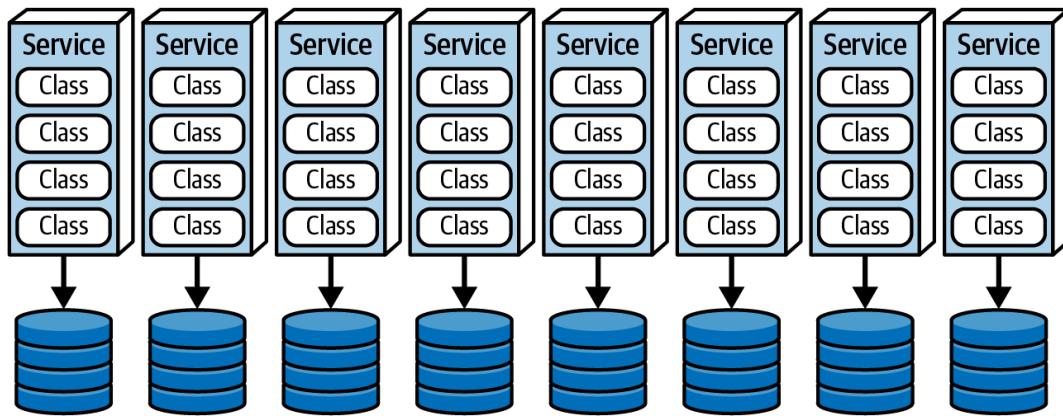


Figure 18-14. The Database-per-Service pattern is the typical database topology for microservices

This topology preserves the bounded context, making it easier for architects to control change. Because other services that need data must request it from the owning service through some sort of contract, they are decoupled from the data's internal structure. Changing the database structure only affects the owning service within the bounded context. This allows architects the freedom to change the database type (such as from a relational database to a document database) without affecting other services.

The Database-per-Service topology also provides excellent scalability, elasticity, availability, and fault tolerance, architectural characteristics that all suffer under the monolithic or domain-based database topologies. Furthermore, managing connections to the database within a bounded context is much easier than with a monolithic or domain-based database.

While the Database-per-Service topology is widely used within microservices, it does have its drawbacks. For example, what if two or more services write to the same database table? What if a service outside of the bounded context *must* query the database directly, for performance reasons? In these cases (which tend to be fairly common), it is possible for a couple of services to share a database, as shown in [Figure 18-15](#). We recommend that no more than five or six services share a single database (or schema). Any more than that and you'll start to experience the same issues with change control, scalability, elasticity, availability, fault tolerance, and so on.

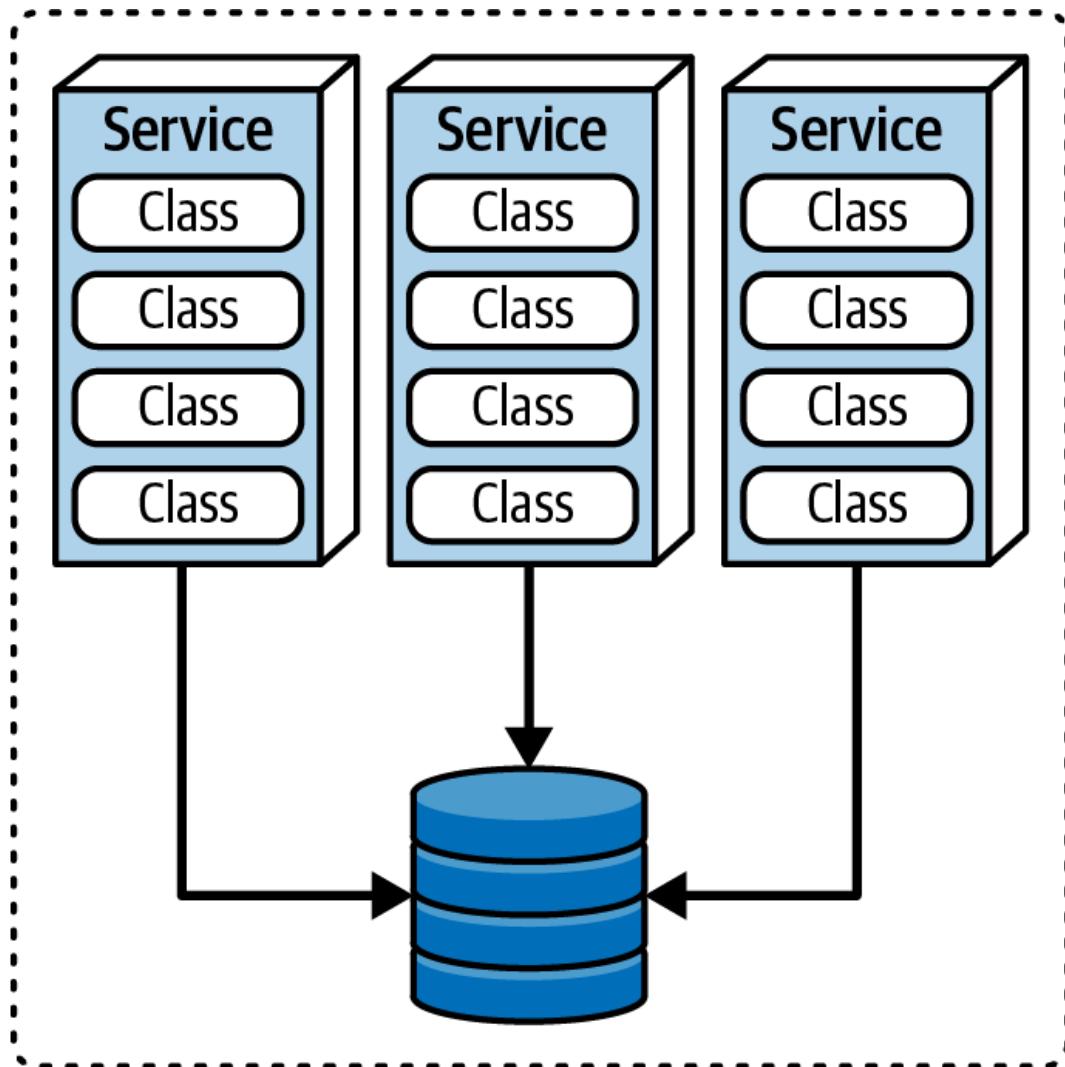


Figure 18-15. It's possible to share data between a few microservices

The box around the services and database in [Figure 18-15](#) denotes the bounded context. Just because services share a database doesn't mean there is no bounded context; it just means the architect has formed a *broader* bounded context. For example, they may have valid reasons for breaking payment processing apart into different payment types (such as credit card, gift card, PayPal, rewards points, and so on), but these individual services still need to update and access the same data. Similarly, an architect may break up a single shipping service into separately deployed services, one for each shipping method, but those services will all still require updates from and access to the same data.

The primary trade-off of sharing data between microservices within a broader bounded context is controlling database changes. When the database schema changes, the architect must now coordinate the change and deployment of multiple services, making database changes riskier and less agile. There might also be negative consequences for scalability, elasticity, and fault tolerance, depending on the business problem and the situation.

Cloud Considerations

While microservices can certainly be deployed in on-prem systems, particularly with the popularity of service orchestration platforms such as [Kubernetes](#) and [Cloud Foundry](#), this architectural style is very well suited for cloud-based deployments—so much so that microservices is sometimes referred to as a “cloud-native” architecture. On-demand provisioning of virtual machines, containers, and

databases, combined with the services-based approach found in cloud environments (such as AWS), fits well with the microservices architectural style.

The attentive reader might wonder why we didn't include [serverless](#) as an architecture style. *Serverless* refers to a cloud-computing model where functions are triggered upon request, allocating necessary machine resources on an on-demand basis. Serverless functions include such artifacts as [AWS Lambdas](#), [Google cloud functions](#), and [Azure cloud functions](#). We believe that serverless is not an architectural style, but a *deployment model* of the microservices architectural style.

Recall from earlier in the chapter that a *microservice* is defined as a single-purpose, separately deployed unit of software that does *one thing* really well (hence the prefix *micro*). As such, microservices tends to be relatively finer-grained than services in other architectural styles. This also essentially describes a *serverless function*, which is why we consider serverless part of microservices.

That said, microservices in cloud environments does not *have* to be deployed as serverless functions; it can be deployed as containerized services just as easily. Most cloud vendors have adopted Kubernetes (or some form of the Kubernetes platform), allowing developers to deploy containerized microservices as easily as serverless ones.

Common Risks

One of the biggest risks in microservices is making services *too* small. In 2016 one of your authors (Mark) coined the name *Grains of Sand* for the antipattern of making services too fine-grained, just like grains of sand on a beach. As we previously mentioned, the *micro* in microservices refers to *what* the service does, not how *big* it is. Service granularity is such an important aspect of microservices that we devote an entire chapter to it (Chapter 7) in our book [Software Architecture: The Hard Parts](#).

Another common risk in microservices is creating too much communication between services. The fine-grained nature of microservices, combined with tight bounded contexts, necessitates that services within a microservices ecosystem will invariably need to communicate with one another. This communication may be due to workflow processing (such as choreography or AWS step functions for serverless microservices) or needing another service's data (due to the bounded context within each service and its data). Regardless of the reason, take care to avoid too much dynamic coupling and interservice communication. Again, this is often the result of making services too fine-grained, and can be fixed by combining services together into coarser-grained microservices.

Another risk in microservices is going overboard with sharing data. While we've mentioned that it is possible (and sometimes necessary) to share data in microservices, too much data sharing creates risks to the system's in change control, scalability, fault tolerance, and overall agility—the things microservices does really well (see ["Style Characteristics"](#)). Know when it's necessary to share data and when to fix data sharing through service consolidation.

A last risk often overlooked in the microservices architectural style is that of reusing code and sharing functionality. Code reuse is a necessary part of software development. However, reusing code and functionality goes directly against the principles of microservices, hence the term “share nothing” architecture described earlier in this chapter. When an architect shares common functionality between services through custom libraries (such as JAR files or DLLs), a part of the

bounded context falls apart. In other words, reused code is spread across multiple bounded contexts, meaning that not *all* of the functionality for that particular function or subdomain is contained within its bounded context, and thus a change to shared code could break services in other bounded contexts. While versioning does help address this issue, sharing code nevertheless adds significant complication to a microservices ecosystem.

Governance

Many of the governance rules and techniques used in microservices address the common risks we described in the previous section. Governance within a microservices architecture is largely about avoiding structural decay.

First and foremost, architects should apply governance to monitor and control the amount of static and dynamic coupling between services. Recall from [Chapter 7](#) that static coupling occurs when microservices share common custom or third-party libraries, as well as in the form of contracts when services need to communicate. Contracts are particularly important: while architects can use asynchronous communication protocols to *dynamically* decouple services, they might nevertheless still be *statically* coupled by the contract used between them (regardless of communication type).

A software bill of materials, deployment scripts, and dependency-management tools can help architects to better understand and govern the number of artifacts shared between services. While we cannot prescribe exactly how much static coupling is *too much*, we recommend striving to minimize coupling between services.

Governing dynamic coupling is much more difficult than governing static coupling. Gathering proper metrics requires some creativity and consistency. One common governance technique is to use logs to identify calls to other services. As services make calls to internal or third-party services, those services log the interactions along with information about what service is being invoked, the protocol used, and so on. Architects can analyze this information through fitness functions to better understand dynamic coupling levels throughout the microservices ecosystem. This approach, however, requires keen governance to ensure each service is exposing this information through logging in a consistent manner. A custom library (such as a JAR file or DLL) is one way to provide a consistent API that all services can bind to at compile time and use to ensure consistent logging.

Another way to gather dynamic coupling metrics in microservices is through registry entries. Whenever the first instance of a service starts, that service registers its interservice calls through some sort of contract (such as JSON) to a custom configuration service or configuration server, such as [Apache ZooKeeper](#). The architect can then query the configuration server to get a map of all interservice calls throughout the microservices ecosystem, which they can use to govern and control the amount of communication between services.

Team Topology Considerations

Since microservices architectures are domain partitioned, they work best when teams are also aligned by domain area (such as cross-functional teams with specialization). When a domain-based requirement comes along, a domain-focused cross-functional team can work together on that feature within a specific

domain service, without interfering with other teams or services. Conversely, technically partitioned teams (such as UI teams, backend teams, database teams, and so on) do not work well with this architectural style because of its domain partitioning. Assigning domain-based requirements to a technically organized team requires a level of interteam communication and collaboration that proves difficult in most organizations.

Here are some considerations for architects who want to align a microservices architecture with the specific team topologies outlined in [“Team Topologies and Architecture”](#):

Stream-aligned teams

If the domain boundaries are properly aligned, stream-aligned teams work well with this architectural style, particularly if their streams are focused on a specific domain. However, microservices architecture becomes more challenging for teams whose streams cross multiple bounded contexts and services, outside of a particular subdomain or domain. In this case, we recommend analyzing the microservices’ bounded contexts and granularity and either realigning them to the streams or choosing a different architectural style.

Enabling teams

Enabling teams are most effective within a microservices architecture when they can use shared services for specialized or cross-cutting concerns. Due to the high degree of modularity found in microservices, enabling teams can work independently from stream-aligned teams to provide additional specialized and shared functionality without getting in the way. Working with platform teams, they can also assist with creating the sidecar components that make up a service mesh (see [“Operational Reuse”](#)).

Complicated-subsystem teams

Complicated-subsystem teams can leverage this architecture style’s service-level modularity to focus on complicated domain or subdomain processing, staying independent of other team members (and services).

Platform teams

The high degree of modularity found in microservices helps stream-aligned teams leverage the benefits of the platform-teams topology by utilizing common tools, services, APIs, and tasks. In many cases, platform teams (sometimes working with enabling teams) focus on creating and maintaining the cross-cutting operational functionality found in sidecars (see [“Operational Reuse”](#)) and the service mesh, freeing stream-aligned teams from these operational concerns.

Style Characteristics

The microservices architecture style offers several extremes on our standard ratings scale, shown in [Figure 18-16](#). A one-star rating means the specific architecture characteristic isn’t well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. Definitions for each characteristic identified in the scorecard can be found in [Chapter 4](#).

Microservices offers notably high support for modern engineering practices such as automated deployment and testability. Microservices couldn't exist without the DevOps revolution, with its relentless march toward automating operational concerns.

	Architectural characteristic	Star rating
Structural	Overall cost	\$\$\$\$
	Partitioning type	Domain
	Number of quanta	1 to many
	Simplicity	★
Engineering	Modularity	★★★★★
	Maintainability	★★★★★
	Testability	★★★★★
	Deployability	★★★★★
Operational	Evolvability	★★★★★
	Responsiveness	★★
	Scalability	★★★★★
	Elasticity	★★★★
	Fault tolerance	★★★★★

Figure 18-16. Microservices characteristics ratings

The independent, single-purpose, and hence fine-grained nature of services in this architectural style generally leads to high fault tolerance, hence the high rating for this characteristic.

The other high points of this architecture are scalability, elasticity, and evolvability. Some of the most scalable systems ever written have used microservices to great success. Similarly, because this style relies heavily on automation and intelligent integration with operations, architects can build in support for elasticity. Because this architecture favors high decoupling at an incremental level, it also supports the modern business practice of evolutionary change, even at the architecture level. Modern businesses move fast, and software development has struggled to keep pace. An architecture structured with extremely small, highly decoupled deployment units can support a faster rate of change.

Performance is often an issue in microservices. Distributed architectures must make many network calls to complete work, and that has a high performance overhead. They must also invoke security checks to verify identity and access for each endpoint, incurring more latency. Microservices also suffers from *data*

latency: when a request requires multiple services to coordinate, that means multiple database calls.

For this reason, the microservices world has many architecture patterns to increase performance, including intelligent data caching and replication to prevent an excess of network calls. Performance is another reason that microservices often use choreography rather than orchestration: less coupling allows for faster communication and fewer bottlenecks.

Microservices is decidedly a domain-partitioned architecture, where each service boundary should correspond to domains. Thanks to the bounded context, it also has the most distinct quanta of any modern architecture—in many ways, it exemplifies what the quantum measure evaluates. The driving philosophy of extreme decoupling creates headaches, but yields tremendous benefits when done well. As in any architecture, architects must understand the rules to break them intelligently.

Examples and Use Cases

Systems that have a high degree of functional and data modularity are good candidates for the microservices architecture. A good use case that exemplifies this style's power is a medical monitoring system that monitors a patient's vital signs: their heart rate, blood pressure, oxygen levels, and so on. Each vital sign the system monitors is a separate, independent function that manages its own data, which fits well with this architectural style and the bounded-context concept.

This patient-monitoring system reads inputs from patient-monitoring devices, records vital signs, analyzes those vital signs for any issues or discrepancies, and alerts a medical professional if it finds problems. Each vital sign can be represented by a separate microservice that is largely independent of other services and data. One exception to this independence, however, would be if a particular vital sign (such as heart rate) needed additional information from another vital sign (such as a sleep-monitor service) to analyze its vital sign for warnings or issues.

[Figure 18-17](#) illustrates how this system might be designed using a microservices architecture. Notice how each vital sign is implemented as a separate microservice, with each maintaining its own data store for vital-sign readings and historical data.

The alert functionality, which is common to all vital-signs services, is represented by a *shared service* called Alert Staff. It alerts a nurse or doctor if a particular service notices something wrong with a reading. Each service asynchronously sends its latest vital-signs reading to a monitor located in the patient's room, represented by the Display Vital Signs shared service.

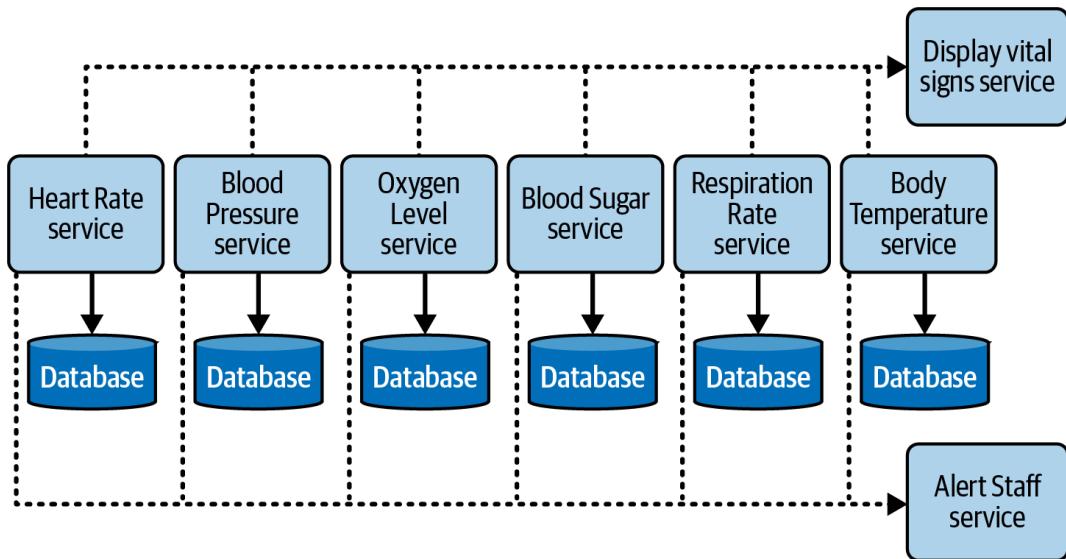


Figure 18-17. A patient medical-monitoring system implemented using a microservices architecture

This example clearly illustrates some of the strengths and advantages of the microservices architecture. Take *fault tolerance*: if one of the vital-sign services happens to crash or become unresponsive, all other vital-sign monitoring services remain fully operational. This is especially important in medical monitoring. Another superpower is *testability*. If a developer performs some maintenance on the blood-pressure monitoring service, the scope of testing is small enough that they can be assured that this particular vital sign is fully tested, and that the maintenance didn't impact other vital-sign services. Finally, a sign of *evolvability* (another microservices superpower) is that the architect could easily add another vital-sign monitor without affecting the other services.

We've touched on many of the significant aspects of microservices architecture in this chapter, and there are many excellent resources you can use to learn more. For a microservices deep dive, we recommend the following:

- [Building Microservices](#), 2nd Edition, by Sam Newman (O'Reilly, 2021)
- [Building Micro-Frontends](#), 2nd Edition, by Luca Mezzalira (O'Reilly, 2025)
- [Microservices vs. Service-Oriented Architecture](#) by Mark Richards (O'Reilly, 2016)
- [Microservices AntiPatterns and Pitfalls](#) by Mark Richards (O'Reilly, 2016)