As a training consultant, I often deal with very practical questions about microservices: What are they? What is so special about microservices? What are some of the best-justified and beneficial use cases for microservices?

Often, these questions are answered in quite a partial manner, with answers greatly depending on one’s past experiences and personal preferences. Answers range from “everything should be a microservice” to “one should avoid microservices like the plague,” with various degrees of cautionary approaches in between.

Despite the availability of multiple answers, I’ve found they generally lack scientific precision, representing points of view rather than hard facts. Indeed, many recommendations were essentially personal experience testimonies that describe the success or failure of specific cases of microservice implementations.

This article seeks to present something entirely different from such anecdotal evidence. I’ll explore some hard facts from an ocean of perspectives and points of view about the nature and applicability of microservices.

**Defining microservices**

Let’s start with a definition of what a microservice actually is. Oh, wait: There is no definition—at least there is no definition that is universally recognized. Instead, there are many competing definitions that appear to share a number of similarities. Instead, here are commonly recognized characteristics to define a microservice.

- Microservices are characterized as *micro* or small in size. This implies a small deployment footprint, making it easier to test, deploy, maintain, and scale a microservice application. Smaller application size is aimed at a shorter and thus cheaper production cycle and more flexible scalability.

- Microservices are described as *loosely coupled*, suggesting that each such application ought to be a self-contained unit of business logic that is not dependent on other applications. Loose
Defining monoliths

A microservice does not exist only to satisfy its own requirements but is a part of an extensive collection of services. Together these services meet the business requirements of an organization that owns these services.

Consider large enterprise-wide business applications, which are often described as monoliths. Unfortunately, much like a microservice, the term monolith is not strictly defined, so I’ll have to resort to describing the characteristics again.

- A monolith is characterized as a large application that implements many different business functions across the enterprise. A similar amount of business logic that many microservices provide can be implemented by a single monolith application, but the development of a larger application would take longer. The monolith will likely be harder to maintain than a group of microservices, and it would be less flexible when it comes to available scalability options.

- Components within a monolith application could be tightly coupled, suggesting that internally a monolith application would have many dependencies between its parts. Of course, this does not necessarily have to be the case, because the number of dependencies greatly relies on specific design and architecture choices. Still, it is certainly more likely that internal dependencies would exist, simply because creating such dependencies is less of a hurdle for an application developer when all code belongs to the same application anyway.

- Monolith development is a collective effort of many programmers and designers, making the development cycle longer, but it may promote a consistent design approach across many different business functions. Using a common technology stack across the enterprise can simplify maintenance and development. Unlike the polyglot approach promoted by microservice advocates, monolith development does not allow flexibility for design and architecture decisions that would be the best fit for a specific subset of business functions implementations. (See Figure 2.)
Common misconceptions

Here are some common misconceptions regarding microservices and monolith architectures.

**Service granularity.** The term service granularity describes the distribution of business functions and features across a number of services. For example, consider a business function that needs to create a description of a scientific experiment and to record measurements for this experiment. This business function can be implemented as a single service operation that handles a single large business object combining all the properties of an experiment and all associated measurements. Such an implementation approach is usually described as a coarse-grained service design.

An alternative approach is known as a fine-grained service design. The exact same business function can be implemented as a number of different service operations, separately handling smaller data units such as an experiment or a measurement. Notice that the difference is in the number of service operations it takes to represent a given amount of business functions. In other words, both approaches implement the same unit of logic but expose it as a different number of services.

The problem is that the granularity of the service is often confused with the concept of a microservice: Basically, a fine-grained service design is not necessarily implemented as a microservice, while a coarse-grained service is not necessarily synonymous with a monolith implementation.

The key to understanding why these are not synonymous concepts is linked to one of the most fundamental properties of a service, which is that a service invoker should not be able to tell anything about the service implementation. Therefore, it makes no difference to the service consumer exactly how a service is implemented behind the scenes. Whether it’s a monolith or not, service consumers should not be able to tell the difference anyway.

To resolve this confusion, I propose to use the phrase implementation granularity instead of service granularity, where implementation granularity could be described as either fine-grained or coarse-grained. This focuses on defining the actual complexity and size of the application behind a service interface. The idea of the implementation granularity should be helpful to clarify the confusion. You could essentially describe a microservices approach as based on fine-grained
implementation design, and which allows the developers to deliver services of any granularity, if that is convenient.

The issue has to do with the implication that microservices must be implemented as small-sized applications, which could be described as a fine-grained implementation design.

Remember that small size and loose coupling are important microservices characteristics that are considered to be beneficial because of the shorter production cycle, flexible scalability, independent versioning, and deployment. However, these benefits should not be considered automatically granted.

**Data fragmentation.** One unintended consequence of a fine-grained application implementation is data fragmentation. A loosely coupled design implies that each microservice application has its own data storage that contains information owned by this specific application.

Another implication of the loosely coupled design is that different applications should not use distributed transactions or a two-phase commit to synchronize their data in order to maintain a high degree of separation between microservice applications. This approach introduces the problem of data fragmentation and the potential lack of consistency.

Consider the case when a given microservice application needs information owned by another microservice application. What if the solution is simply to allow one application to invoke another to obtain or synchronize required pieces of information? This could work, but what if a given service experiences performance problems or an outage? This would inevitably have a cascading effect on any other dependent services, leading to larger outages and overall performance degradation. Such an approach may work for a small number of applications, but the larger the set of such applications, the greater the risks to their performance and availability.

Thus, consider another solution that addresses the data consistency and fragmentation implications. For example, what if each microservice application caches information that it needs from other applications?

Caching should provide some degree of autonomy for each application, contributing to its capability to be isolated and self-sustained. However, caching also means that applications would have to be designed, considering that the latest data state may not always be available. Different distributed caching and data-streaming solutions could be utilized to automate the handling of information replication. Finally, each application has to provide data state tracking and undo behaviors instead of the distributed transaction coordination.

Inevitably, these issues lead to design complications, making each microservice application not as simple as it appears at first glance.

Furthermore, each development team that works on a particular microservice cannot really remain in a state of perfect isolation but has to maintain data dependencies with other applications.

In other words, microservices architecture does not appear to actually deliver on the promise of completely solving the dependency issues experienced by monoliths. Instead, data consistency and integrity management are shifted from being an internal concern of a single
Versioning. Another problem arises from the promise of independent versioning capabilities for each microservice application. In more complex service interaction scenarios, functional dependencies had to be considered along with the data dependencies.

Imagine a service in which the internal implementation had been modified. Such modification may not necessarily cause any changes to the service interface or the shape and format of its data. Many developers would not consider such implementation modifications as having any consequences that would require the production of a new version of the service and thus would not notify the dependent application developers of these changes.

However, such modifications may affect the semantics of how the service interprets its data, leading to a discrepancy between microservice applications.

For example, consider the implications of a change in the interpretation of a specific value. Suppose an application that records measurements considers an inch to be a default unit of measure, and other applications may rely on this to be the default value. An internal change may lead to the centimeter being implied to be the default value instead of an inch. This could have a knock-on effect on any other microservice applications, which—all things considered—could even be dangerous. Yet, the chances are that developers of these other systems may not be any wiser about said change.

This shows that in any nontrivial application interaction scenario, microservices characteristics should not be automatically assumed as purely beneficial.

Monoliths and microservices face similar problems

Businesses face the exact same functional and data integration problems regardless of the choice of architecture. Because both monoliths and microservices must address them anyway, the question really is in understanding the benefits and drawbacks of each approach.

- A monolith offers data and functional consistency as an integral part of its centralized design and the unified development approach at the cost of scalability and flexibility.
- Microservices offer a significant degree of development autonomy yet shift the responsibility to resolve data and functional consistency problems to many different independent development teams, which could be a very precarious coordination task.

Perhaps a balanced approach aiming to embrace benefits and mitigate drawbacks of both microservices and monolith architectures could be the way forward. In my opinion, the most critical factor is the idea of the implementation granularity, as discussed earlier.

REST services are by far the most common form of representing microservices applications, and most of the use case examples for REST services focus on each such service representing a single business entity. This approach results in extremely fine-grained application implementations.
Consider the increase in the number of dependencies between such applications because of the need to maintain data cohesion across so many independently managed business entities.

However, strictly speaking, the microservices architecture does not require such a fine level of implementation granularity. In fact, microservices are usually described as focused on a single *business capability*, which is not necessarily the same as a single business entity, because a number of business entities can be used to support a single business capability.

Typically, such entities form data groups that exhibit very close ties and a significant number of dependencies. Using these data groupings as a guiding principle to decide on the implementation granularity of applications should result in a smaller number of microservice applications that are better isolated from each other. Each such application would not truly be micro compared to the one-application-per-entity structure, but the application would not be a single monolith that incorporates the entirety of the enterprise functions.

This approach should reduce the need to synchronize information across applications and, in fact, may have a positive effect on the overall system performance and reliability.

**Business capability.** What constitutes a single business capability? It obviously sounds like a set of commonly used business functions, but that is still a rather vague definition. It’s worth considering the way business functions use data as a *grouping principle*. For example, data could be grouped as a set of data objects produced in a context of a specific business process and having common ownership.

Common ownership implies that there is a specific business unit that is responsible for a number of business entities. Common ownership also implies that business decisions within this unit define the semantic context for these entities, changes that may affect their data structure and, most importantly, define a set of business functions that are responsible for creating, updating, and deleting this data.

Other business units may wish to read the same information, but they mostly act as consumers of this data rather than producers. Thus, each application would be responsible for its own subset of business entities and would be capable of performing all required transactions locally, without a need for distributed transaction coordination or two-phase commit operations.

Of course, data replication across applications would still be required for data caching purposes to improve the individual application autonomy. However, the overhead of maintaining a set of read-only data replicas is significantly smaller than the overhead of maintaining multidirectional data synchronization. Also, there would be a need to perform fewer data replications because of the overall reduction in the number of applications.

**Data ownership.** In large enterprises, the question of data ownership could be difficult to resolve and would require some investment into both data and business process analysis. Understanding a larger context of information helps to determine where data originates as well as the possible consumer of this data. This analysis picture has to represent a much broader landscape than that of an individual microservice application.
Practically speaking, in addition to a number of development teams dedicated to the production of specific applications, an extra group of designers and analysts has to be established to produce and maintain an integrated enterprise data model, assist in scoping individual applications, and reconcile any discrepancies between all other development teams.

As you can see, this approach proposes to borrow some monolith application design characteristics but use them differently, not aiming to produce a single enterprisewide application but rather support the integration of many applications, each focused on implementing their specific business capabilities.

Furthermore, this approach aims at ensuring that service application boundaries are well-defined and maintained, and it suggests criteria to establish such boundaries based on data ownership principles. There are even some interesting APIs, such as GraphQL, that can support these integration efforts.

Nomenclature. There is one more problem to resolve: What should we call this architecture? Should it still be called microservices, even though some business capabilities may own a relatively large number of entities, and thus some applications may turn out not to actually be that small?

I want to suggest calling such an approach a microlith architecture to indicate a hybrid nature of the strategy that attempts to combine the benefits of both microservices and monolith architectures. The actual word microlith means a small stone tool such as a prehistoric arrowhead or a needle made of stone. I like the sense of practicality projected by this term. (See Figure 3.)

![Microlith Application Architecture](image)

**Figure 3. Microlith application architecture**

I'm sure many would agree that the best design and architecture decisions are based on practical cost-benefit analysis rather than blindly following abstract principles.

**Dig deeper**

- Size still matters
- Building microservices with Micronaut
- Jakarta EE: Building microservices with Java EE's successor
- Book review: Testing Java Microservices
- Book review: Enterprise Java Microservices
Vasily Strelnikov

Vasily Strelnikov is a senior principal OCI solutions specialist at Oracle; previously, he was a senior principal training consultant. Strelnikov’s specialties are system design and integration using service oriented architecture (SOA), Service Component Architecture (SCA), and Java; he has created training courses for Java and Java EE. He is based in London.