martinfowler.com

Microservice Trade-Offs

Martin Fowler

22-28 minutes

Many development teams have found the microservices architectural style to be a superior approach to a monolithic architecture. But other teams have found them to be a productivity-sapping burden. Like any architectural style, microservices bring costs and benefits. To make a sensible choice you have to understand these and apply them to your specific context.

Microservices provide benefits…

- Strong Module Boundaries: Microservices reinforce modular structure, which is particularly important for larger teams.
- Independent Deployment: Simple services are easier to deploy, and since they are autonomous, are less likely to cause system failures when they go wrong.
- Technology Diversity: With microservices you can mix multiple languages, development frameworks and data-storage technologies.

…but come with costs

- Distribution: Distributed systems are harder to program, since remote calls are slow and are always at risk of failure.
- Eventual Consistency: Maintaining strong consistency is extremely difficult for a distributed system, which means everyone has to

manage eventual consistency.

- Operational Complexity: You need a mature operations team to manage lots of services, which are being redeployed regularly.

Summing Up

Strong Module Boundaries (Pro)

The first big benefit of microservices is strong module boundaries. This is an important benefit yet a strange one, because there is no reason, in theory, why a microservices should have stronger module boundaries than a monolith.

So what do I mean by a strong module boundary? I think most people would agree that it's good to divide up software into modules: chunks of software that are decoupled from each other. You want your modules to work so that if I need to change part of a system, most of the time I only need to understand a small part of that system to make the change, and I can find that small part pretty easily. Good modular structure is useful in any program, but becomes exponentially more important as the software grows in size. Perhaps more importantly, it grows more in importance as the team developing it grows in size.

Advocates of microservices are quick to introduce Conways Law, the notion that the structure of a software system mirrors the communication structure of the organization that built it. With larger teams, particularly if these teams are based in different locations, it's important to structure the software to recognize that inter-team communications will be less frequent and more formal than those within a team. Microservices allow each team to look after relatively independent units with that kind of communication pattern.

As I said earlier, there's no reason why a monolithic system shouldn't have a good modular structure. [1] But many people have observed that it seems rare, hence the Big Ball of Mud is most

common architectural pattern. Indeed this frustration with the common fate of monoliths is what's driven several teams to microservices. The decoupling with modules works because the module boundaries are a barrier to references between modules. The trouble is that, with a monolithic system, it's usually pretty easy to sneak around the barrier. Doing this can be a useful tactical shortcut to getting features built quickly, but done widely they undermine the modular structure and trash the team's productivity. Putting the modules into separate services makes the boundaries firmer, making it much harder to find these cancerous workarounds.

An important aspect of this coupling is persistent data. One of the key characteristics of microservices is Decentralized Data Management, which says that each service manages its own database and any other service must go through the service's API to get at it. This eliminates Integration Databases, which are a major source of nasty coupling in larger systems.

It's important to stress that it's perfectly possible to have firm module boundaries with a monolith, but it requires discipline. Similarly you can get a Big Ball of Microservice Mud, but it requires more effort to do the wrong thing. The way I look at, using microservices increases the probability that you'll get better modularity. If you're confident in your team's discipline, then that probably eliminates that advantage, but as a team grows it gets progressively harder to keep disciplined, just as it becomes more important to maintain module boundaries.

This advantage becomes a handicap if you don't get your boundaries right. This is one of the two main reasons for a Monolith First strategy, and why even those more inclined to run with microservices early stress that you can only do so with a well understood domain.

But I'm not done with caveats on this point yet. You can only really

tell how well a system has maintained modularity after time has passed. So we can only really assess whether microservices lead to better modularity once we see microservice systems that have been around for at least a few years. Furthermore, early adopters tend to be more talented, so there's a further delay before we can assess the modularity advantages of microservice systems written by average teams. Even then, we have to accept that average teams write average software, so rather than compare the results to top teams we have to compare the resulting software to what it would have been under a monolithic architecture - which is a tricky counter-factual to assess.

All I can go on for the moment is the early evidence I have hear from people I know who have been using this style. Their judgement is that it is significantly easier to maintain their modules.

One case study was particularly interesting. The team had made the wrong choice, using microservices on a system that wasn't complex enough to cover the Microservice Premium. The project got in trouble and needed to be rescued, so lots more people were thrown onto the project. At this point the microservice architecture became helpful, because the system was able to absorb the rapid influx of developers and the team was able to leverage the larger team numbers much more easily than is typical with a monolith. As a result the project accelerated to a productivity greater than would have been expected with a monolith, enabling the team to catch up. The result was still a net negative, in that the software cost more staff-hours than it would have done if they had gone with a monolith, but the microservices architecture did support ramp up.

Distribution (Con)

So microservices use a distributed system to improve modularity. But distributed software has a major disadvantage, the fact that it's distributed. As soon as you play the distribution card, you incur a

whole host of complexities. I don't think the microservice community is as naive about these costs as the distributed objects movement was, but the complexities still remain.

The first of these is performance. You have to be in a really unusual spot to see in-process function calls turn into a performance hot spot these days, but remote calls are slow. If your service calls half-a-dozen remote services, each which calls another half-a-dozen remote services, these response times add up to some horrible latency characteristics.

Of course you can do a great deal to mitigate this problem. Firstly you can increase the granularity of your calls, so you make fewer of them. This complicates your programming model, you now have to think of how to batch up your inter-service interactions. It will also only get you so far, as you are going to have to call each collaborating service at least once.

The second mitigation is to use asynchrony. If make six asynchronous calls in parallel you're now only as slow as the slowest call instead of the sum of their latencies. This can be a big performance gain, but comes at another cognitive cost. Asynchronous programming is hard: hard to get right, and much harder to debug. But most microservice stories I've heard need asynchrony in order to get acceptable performance.

Right after speed is reliability. You expect in-process function calls to work, but a remote call can fail at any time. With lots of microservices, there's even more potential failure points. Wise developers know this and design for failure. Happily the kinds of tactics you need for asynchronous collaboration also fit well with handling failure and the result can improve resiliency. That's not much compensation however, you still have the extra complexity of figuring out the consequences of failure for every remote call.

And that's just the top two Fallacies of Distributed Computing.

There are some caveats to this problem. Firstly many of these issues crop up with a monolith as it grows. Few monoliths are truly self-contained, usually there are other systems, often legacy systems, to work with. Interacting with them involves going over the network and running into these same problems. This is why many people are inclined to move more quickly to microservices to handle the interaction with remote systems. This issue is also one where experience helps, a more skillful team will be better able to deal with the problems of distribution.

But distribution is always a cost. I'm always reluctant to play the distribution card, and think too many people go distributed too quickly because they underestimate the problems.

Eventual Consistency (Con)

I'm sure you know websites that need a little patience. You make an update to something, it refreshes your screen and the update is missing. You wait a minute or two, hit refresh, and there it is.

This is a very irritating usability problem, and is almost certainly due to the perils of eventual consistency. Your update was received by the pink node, but your get request was handled by the green node. Until the green node gets its update from pink, you're stuck in an inconsistency window. Eventually it will be consistent, but until then you're wondering if something has gone wrong.

Inconsistencies like this are irritating enough, but they can be much more serious. Business logic can end up making decisions on inconsistent information, when this happens it can be extremely hard to diagnose what went wrong because any investigation will occur long after the inconsistency window has closed.

Microservices introduce eventual consistency issues because of their laudable insistence on decentralized data management. With a monolith, you can update a bunch of things together in a single

transaction. Microservices require multiple resources to update, and distributed transactions are frowned on (for good reason). So now, developers need to be aware of consistency issues, and figure out how to detect when things are out of sync before doing anything the code will regret.

The monolithic world isn't free from these problems. As systems grow, there's more of a need to use caching to improve performance, and cache invalidation is the other Hard Problem. Most applications need offline locks to avoid long-lived database transactions. External systems need updates that cannot be coordinated with a transaction manager. Business processes are often more tolerant of inconsistencies than you think, because businesses often prize availability more (business processes have long had an instinctive understanding of the CAP theorem).

So like with other distributed issues, monoliths don't entirely avoid inconsistency problems, but they do suffer from them much less, particularly when they are smaller.

Independent Deployment (Pro)

The trade-offs between modular boundaries and the complexities of distributed systems have been around for my whole career in this business. But one thing that's changed noticeably, just in the last decade, is the role of releasing to production. In the twentieth century production releases were almost universally a painful and rare event, with day/night weekend shifts to get some awkward piece of software to where it could do something useful. But these days, skillful teams release frequently to production, many organizations practicing Continuous Delivery, allowing them to do production releases many times a day.

> Microservices are the first post DevOps revolution architecture
>
> -- Neal Ford

This shift has had a profound effect on the software industry, and it is deeply intertwined with the microservice movement. Several microservice efforts were triggered by the difficulty of deploying large monoliths, where a small change in part of the monolith could cause the whole deployment to fail. A key principle of microservices is that services are components and thus are independently deployable. So now when you make a change, you only have to test and deploy a small service. If you mess it up, you won't bring down the entire system. After all, due the need to design for failure, even a complete failure of your component shouldn't stop other parts of the system from working, albeit with some form of graceful degradation.

This relationship is a two-way street. With many microservices needing to deploy frequently, it's essential you have your deployment act together. That's why rapid application deployment and rapid provisioning of infrastructure are Microservice Prerequisites. For anything beyond the basics, you need to be doing continuous delivery.

The great benefit of continuous delivery is the reduction in cycle-time between an idea and running software. Organizations that do this can respond quickly to market changes, and introduce new features faster than their competition.

Although many people cite continuous delivery as a reason to use microservices, it's essential to mention that even large monoliths can be delivered continuously too. Facebook and Etsy are the two best known cases. There are also plenty of cases where attempted microservices architectures fail at independent deployment, where multiple services need their releases to be carefully coordinated [2]. While I do hear plenty of people arguing that it's much easier to do continuous delivery with microservices, I'm less convinced of this than their practical importance for modularity - although naturally modularity does correlate strongly with delivery speed.

Operational Complexity (Con)

Being able to swiftly deploy small independent units is a great boon for development, but it puts additional strain on operations as half-a-dozen applications now turn into hundreds of little microservices. Many organizations will find the difficulty of handling such a swarm of rapidly changing tools to be prohibitive.

This reinforces the important role of continuous delivery. While continuous delivery is a valuable skill for monoliths, one that's almost always worth the effort to get, it becomes essential for a serious microservices setup. There's just no way to handle dozens of services without the automation and collaboration that continuous delivery fosters. Operational complexity is also increased due to the increased demands on managing these services and monitoring. Again a level of maturity that is useful for monolithic applications becomes necessary if microservices are in the mix.

Microservice proponents like to point out that since each service is smaller it's easier to understand. But the danger is that complexity isn't eliminated, it's merely shifted around to the interconnections between services. This can then surface as increased operational complexity, such as the difficulties in debugging behavior that spans services. Good choices of service boundaries will reduce this problem, but boundaries in the wrong place makes it much worse.

Handling this operational complexity requires a host of new skills and tools - with the greatest emphasis being on the skills. Tooling is still immature, but my instinct tells me that even with better tooling, the low bar for skill is higher in a microservice environment.

Yet this need for better skills and tooling isn't the hardest part of handling these operational complexities. To do all this effectively you also need to introduce a devops culture: greater collaboration between developers, operations, and everyone else involved in

software delivery. Cultural change is difficult, especially in larger and older organizations. If you don't make this up-skilling and cultural change, your monolithic applications will be hampered, but your microservice applications will be traumatized.

Technology Diversity (Pro)

Since each microservice is an independently deployable unit, you have considerable freedom in your technology choices within it. Microservices can be written in different languages, use different libraries, and use different data stores. This allows teams to choose an appropriate tool for the job, some languages and libraries are better suited for certain kinds of problems.

Discussion of technical diversity often centers on best tool for the job, but often the biggest benefit of microservices is the more prosaic issue of versioning. In a monolith you can only use a single version of a library, a situation that often leads to problematic upgrades. One part of the system may require an upgrade to use its new features but cannot because the upgrade breaks another part of the system. Dealing with library versioning issues is one of those problems that gets exponentially harder as the code base gets bigger.

There is a danger here that there is so much technology diversity that the development organization can get overwhelmed. Most organizations I know do encourage a limited set of technologies. This encouragement is supported by supplying common tools for such things as monitoring that make it easier for services to stick to a small portfolio of common environments.

Don't underestimate the value of supporting experimentation. With a monolithic system, early decisions on languages and frameworks are difficult to reverse. After a decade or so such decisions can lock teams into awkward technologies. Microservices allow teams to

experiment with new tools, and also to gradually migrate systems one service at a time should a superior technology become relevant.

Secondary Factors

I see the items as above as the primary trade-offs to think about. Here's a couple more things that come up that I think are less important.

Microservice proponents often say that services are easier to scale, since if one service gets a lot of load you can scale just it, rather than the entire application. However I'm struggling to recall a decent experience report that convinced me that it was actually more efficient to do this selective scaling compared to doing cookie-cutter scaling by copying the full application.

Microservices allow you to separate sensitive data and add more careful security to that data. Furthermore by ensuring all traffic between microservices is secured, a microservices approach could make it harder to exploit a break-in. As security issues grow in importance, this could migrate to becoming a major consideration for using microservices. Even without that, it's a not unusual for primarily monolithic systems to create separate services to handle sensitive data.

Critics of microservices talk about the greater difficulty in testing a microservices application than a monolith. While this is a true difficulty - part of the greater complexity of a distributed application - there are good approaches to testing with microservices. The most important thing here is to have the discipline to take testing seriously, compared to that the differences between testing monoliths and testing microservices are secondary.

Summing Up

Any general post on any architectural style suffers from the Limitations Of General Advice. So reading a post like this can't lay out the decision for you, but such articles can help ensure you consider the various factors that you should take into account. Each cost and benefit here will have a different weight for different systems, even swapping between cost and benefit (strong module boundaries are good in more complex systems, but a handicap to simple ones) Any decision you make depends on applying such criteria to your context, assessing which factors matter most for your system and how they impact your particular context. Furthermore, our experience of microservice architectures is relatively limited. You can usually only judge architectural decisions after a system has matured and you've learned what it's like to work with years after development began. We don't have many anecdotes yet about long-lived microservice architectures.

Monoliths and microservices are not a simple binary choice. Both are fuzzy definitions that mean many systems would lie in a blurred boundary area. There's also other systems that don't fit into either category. Most people, including myself, talk about microservices in contrast to monoliths because it makes sense to contrast them with the more common style, but we must remember that there are systems out there that don't fit comfortably into either category. I think of monoliths and microservcies as two regions in the space of architectures. They are worth naming because they have interesting characteristics that are useful to discuss, but no sensible architect treats them as a comprehensive partitioning of the architectural space.

That said, one general summary point that seems to be widely accepted is there is a Microservice Premium: microservices impose a cost on productivity that can only be made up for in more complex systems. So if you can manage your system's complexity with a monolithic architecture then you shouldn't be using microservices.

But the volume of the microservices conversation should not let us forget the more important issues that drive the success and failure of software projects. Soft factors such as the quality of people on the team, how well they collaborate with each other, and the degree of communication with domain experts, will have a bigger impact than whether to use microservices or not. On a purely technical level, it's more important to focus on things like clean code, good testing, and attention to evolutionary architecture.
