MICROSERVICES / CONTRIBUTED

Microservices vs. Monoliths: An Operational Comparison

7 Jul 2020 9:20am, by Alexander Kainz

Alexander Kainz

Alexander Kainz has been in IT for over 20 years. Java, Python, JavaScript and Smalltalk are his programming languages of choice. His focus has been on the enterprise especially in the financial industry. In his career he has worked in Germany, USA, Australia and finally Thailand. He is lucky to live in Bangkok, a city that he adores for its energy, its diversity and its people. He writes about his interests, that include Digital Transformation, Microservices, Rest APIs and DevOps.

Introduced as a counter to these so-called monolithic architectures, the microservice architecture splits the business process into multiple independent services.

For example, in the context of an air-ticket booking, the monolithic approach involves building a single software that has a process "book ticket."

"Book ticket" involves a number of individual processes. Maybe reserve a ticket with the airline, bill the customer's credit card and send out confirmations to the customer if the ticket is successfully booked.

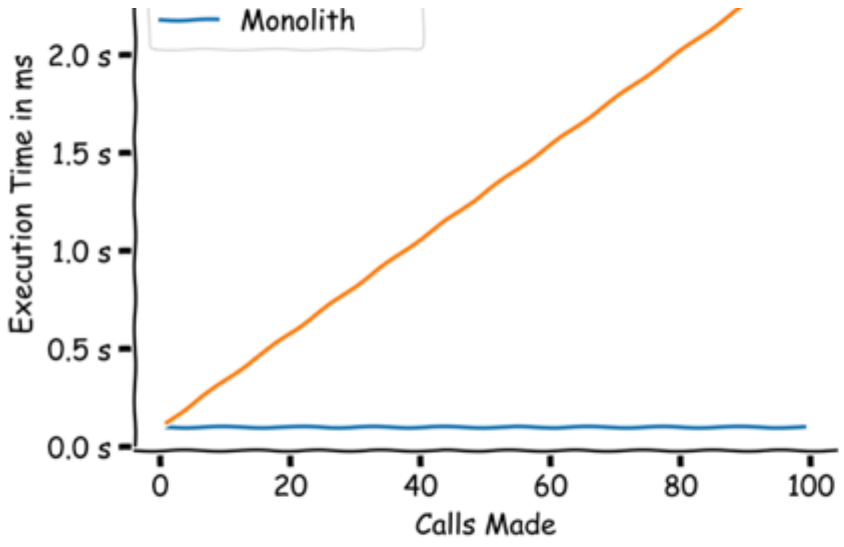
In a microservice architecture, the individual processes are broken out into independent services. In the example above, the services could be ticket booking, card payment and confirmation. Now the independent services communicate with each other through defined interfaces.

To say microservice architectures have become mainstream would be an understatement. In my news feed, I barely see monolithic architectures anymore — only in articles about going back from microservices to monolithic architectures. In this article, we'll pit these two against each other.

Two software architectural styles enter the ring, one will leave as a winner.

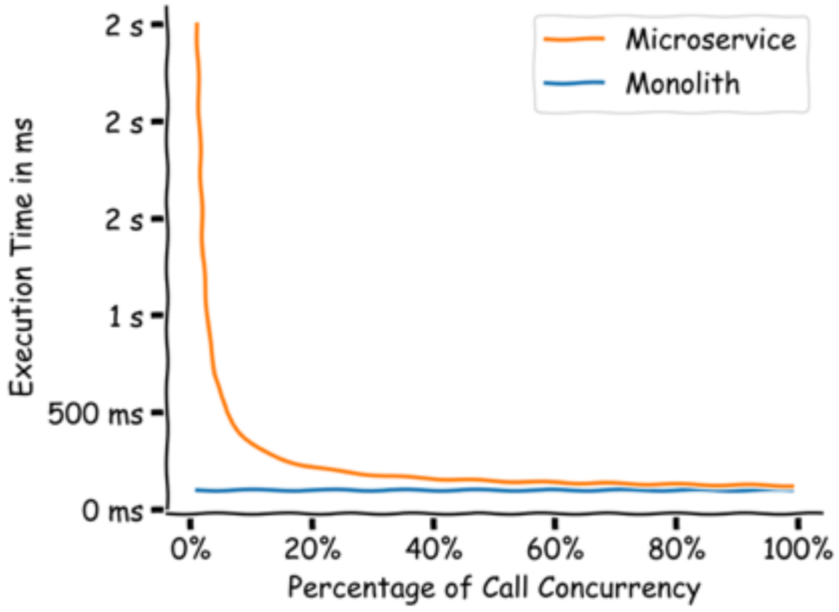
Round One: Latency

There is a fundamental law of physics at play when it comes to microservices. Whenever a microservice calls another service over the network, bytes sent over a network. This involves turning bytes into electrical signals, or pulsed light and then turning these signals back into bytes. According to this link, the latency of a microservices call is at least 24ms. If we assume that the actual processing takes about 100 ms then the total processing time looks as below:



Network Latency — Microservices vs. Monoliths

Ideally, all call execution can happen at the same time and don't depend on each other. This is called the fan-out pattern. Below is a diagram that shows, how the total time goes down as more and more calls execute at the same time.



Executing more calls concurrently means total execution time goes down

Executing all calls in parallel means the service will return to the consumer after the longest call finishes. A monolith has no network latency, as all calls are local. Even in a perfectly parallelizable world, the monoliths will still be faster. The way to fix this issue is reducing call chain length, using fan-out and keeping data as local as possible. Also using the fan-out pattern can significantly improve performance, as seen above. But in the end microservices can’t outrun physics when it comes to latency.

This is a clear win for monoliths.

Round Two: Complexity

There are a number of factors at play when considering complexity: The complexity of development, and the complexity of running the software. For the complexity of development the size of the codebase can quickly grow when building microservice-based software. Multiple source codes are involved, using different frameworks and even different languages. Since microservices need to be independent of one another there will often be code duplication. Also, different services may use different versions of libraries, as release schedules are not in sync. For the running and monitoring aspect, the number of affected services is highly relevant. A monolith only talks to itself. That means it has one potential partner in its processing flow. A single call in a microservice architecture can hit multiple services. These can be on different servers or even in different geographic locations.

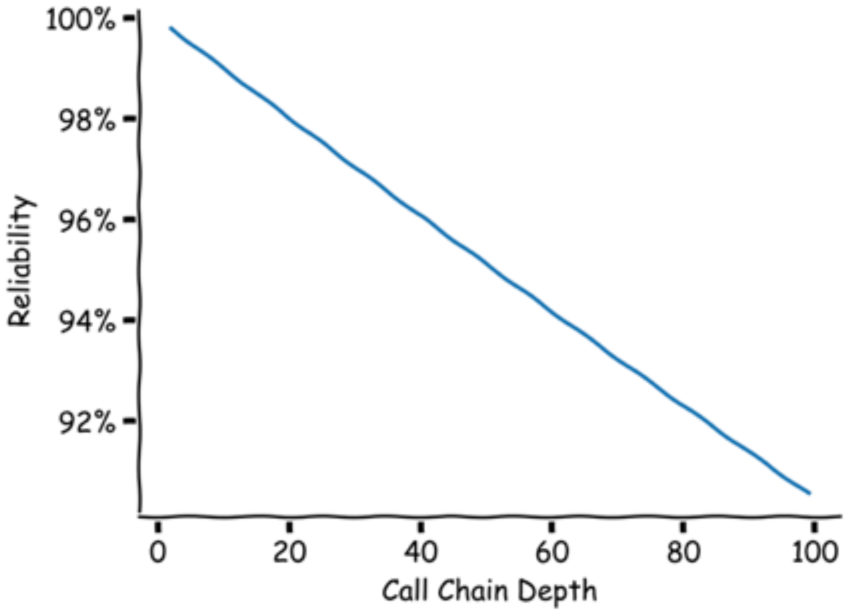
In a monolith, logging is as simple as viewing a single log-file. However, for microservices tracking an issue may involve checking multiple log files. Not only is it necessary to find all the relevant logs outputs, but also put them together in the correct order. Microservices use a unique id, or span, for each call. This allows tools such as Elasticsearch to find all the relevant log output across services. Tools such as Jaeger can trace and profile calls across multiple microservices.

When running microservices in a Kubernetes cluster, complexity further increases. While Kubernetes enable capabilities such as auto-scaling, it is not an easy system to administer. To deploy a monolith a simple copy operation may be enough. To start or stop the monolith often a simple command is enough. Kubernetes on the other hand is not for the faint of heart. Transactions also add complexity to running microservice architectures, compared to a monolith. Across service borders, it is difficult to guarantee that data is in sync. For example, a badly implemented call retry could execute a payment twice. Microservice architectures can manage this by using techniques such as a central coordinator. However, in a monolithic architecture, transactions are easy to handle or even transparent to the developer.

For complexity, another win goes to the monolith.

Round 3: Reliability

As all calls in the monolith are local, there is no chance of a network failure. Compare this to microservices. Assume a microservice calls another service over the network with a reliability of 99.9%. That means out of 1000 calls one will fail because of network issues. Now if that service calls another service we get a reliability of 99.8%. For a call chain that has a depth of 10 calls, we are down to 99% reliability — this means 1 in 100 calls fails.



Reliability goes down as call chains get longer

When designing microservice architectures, it is important to assume the network will break at some point. Microservices offer some solutions to address this problem. The open source Spring Cloud offers transparent load balancing and failure handling for Java. Service meshes such as Istio can achieve that for multiple languages. Network issues happen in microservice architectures more often, but since this is expected they are designed to handle these issues. On top of that, since network failures occur regularly, microservices have proven their ability to handle them. This ensures that there are no hidden bugs in regard to these issues.

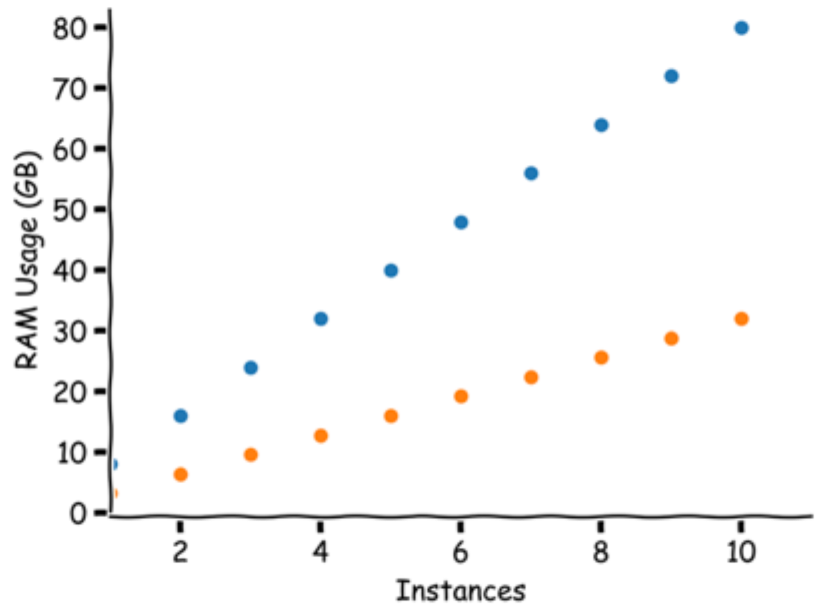
When a service fails in a microservice cluster, the cluster manager will just bring up a replacement. This makes microservice architectures highly resilient. Netflix created a tool called Chaos Monkey that randomly terminates virtual machines and containers. This way they can make sure that the system is able to handle outages in a production environment. Monoliths can of course also run in a cluster at scale. But due to their size, an issue hits them harder. It's hard to imagine randomly restarting monoliths to make sure they survive it. On the other hand, the most reliable software is a monolith, e.g. industrial controllers and aircraft flight controls. It definitely is possible to build highly reliable monoliths, but it becomes tough at scale and in the cloud.

In the end, microservices come out ahead with a win.

Round 4: Resource Usage

If a microservice call does the same work with the same algorithms it will always use more resources than a monolith. Docker and virtual machines add overhead. Another benchmark found about an 8% drop in the number of connections when running using in a docker container. Image orchestration will also consume resources, as will the log aggregation and monitoring.

However, microservices allow us to use resources smarter, a lot smarter. Since the cluster manager can allocate resources as needed, the actual resource usage can be a lot lower. Looking at a monolith that does 80% of the work in 20% of the code, we can show what happens if we can independently scale the “hot” part of the code. For example, if one instance of a monolith uses 8GB, two instances use 16GB, and so on. Let’s assume that 20% are executable in parallel to do the heavy lifting. We have one instance with 8GB and then microservices with 20% of the RAM usage which is 1.6GB. This means that for two instances we have a RAM usage of 9.6GB. The diagram below shows the difference in resource usage.



Monoliths require more resources than Microservices, as more instances are running

Monoliths can outperform microservices in edge cases. If, for instance, a call transfers large amounts of data.

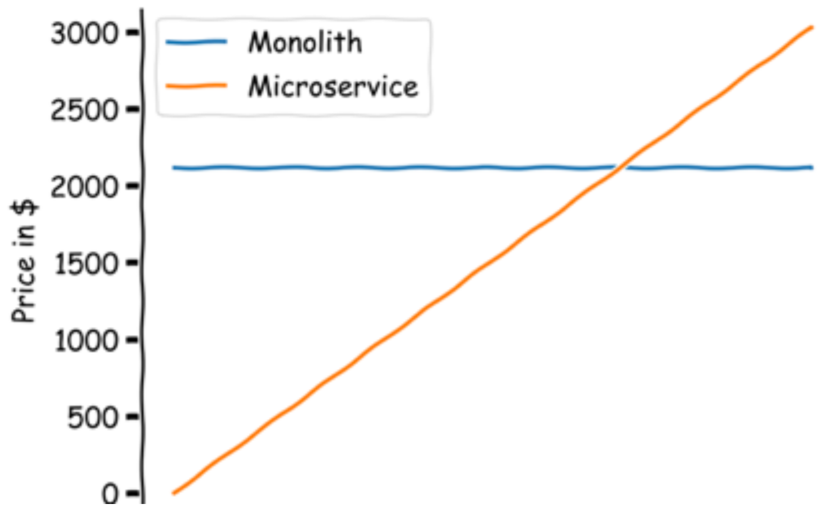
But in most scenarios, the resource usage is lower and this is a win for microservices.

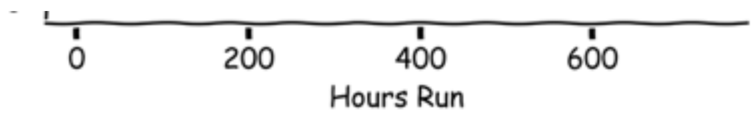
Round 5: Scalability

There are ways to scale a monolith. One can run multiple instances and route requests accordingly. Or one can run multiple threads or use non-blocking IO. For microservice architectures all three of these are true as well. But as examined in Resource Usage, they can be done with less resources. This means more connections can be handled per resource. For money spent on resources, microservices deliver more throughput. Also, more precise scaling is possible. If a monolith is using all the resources, the way to handle more connections is to bring up a second instance. If a single microservice uses all the resources, only this service will need more instances. As microservices are less resource-intensive this saves on resources. Since scaling is easy and precise, this means only the necessary amount of resources is used. Administrators can bring AWS or other cloud providers online or offline as needed.

For example, let's assume a monolith is running on the biggest Amazon Web Services instance at the moment. The **m5.24xlarge** instance offers a whopping 96 CPUs and 384 GB RAM. The cost at the moment is also whopping at $2,119.19 per month for instance that is running 24/7. For the same money 12 **c5.2xlarge** instances with 8 virtual CPUs and 16GB of RAM each running 24/7 can be bought. But most workloads don't demand 24/7 full resource usage. Instead, resource usage peaks at certain hours and is lower the rest of the time. Below is a graph that shows how much these 12 smaller instances cost if they are only run a certain amount of time instead of 24/7.

Since dedicated resources are cheaper than then one charged per hour, there is a point where the single instance becomes cheaper. The cross over in this example is 520 hours, or roughly if the instances are running 70% of the time.





Comparing the price of 12 on-demand microservice containers with a dedicated massive monolithic container

And since microservice architectures are more fine-grained, scaling individual services also is more fine-grained.

For the precise scaling and better resource usage, this is a clear win for microservices.

Round 6: Throughput

Let's look at one more performance metric. The absolute throughput. We have already explored the relationship between network latency and parallelization. The same relationship is at play for throughput. In workloads that cannot run concurrently across the network, monoliths may deliver better performance. Data needs to sent between services and also all the infrastructure induces a certain overhead. If the workload cannot be scaled to multiple instances, a monolith can deliver a higher throughput.

With highly localized workloads and no overhead due to containers, container orchestration or service meshes this is a point for monoliths.

Round 7: Time to Market

One of the reasons people cite for choosing microservice architectures is time to market. This is the time from making a business decision for a feature until the feature is publicly available. Because of their size and dependencies, monoliths are typically harder to deploy. It is, on the other hand, easier to deploy microservices often or continuously. One reason is that the effects of changes are highly localized. Microservices should only expose their interfaces and not their implementations. This means that the developer can change the implementation without modifying dependent services. Since the interface is also clear and can be versioned, any changes to it should have a clearly defined impact. For the monolith, object-oriented programming allows separation of interface of implementation. But it requires a highly disciplined team to not fall prey to the siren call of depending on the implementation.

Furthermore, microservices are easier to test. Since microservices cover only a limited set of features, the amount of dependencies is also less. With less dependencies, tests can be written and run faster. Thirdly microservices are less resource-intensive and built to scale. This allows microservices to have seamless rollouts. Microservices can be brought up on a portion of cluster nodes. Users can then be migrated to the new version successively. Other migration strategies involve running old and new versions concurrently. This allows a quick rollback to the old version if there is an issue. The fine-grained architecture of microservices allows faster and more robust rollouts. This reduces the time from idea to production deployments.

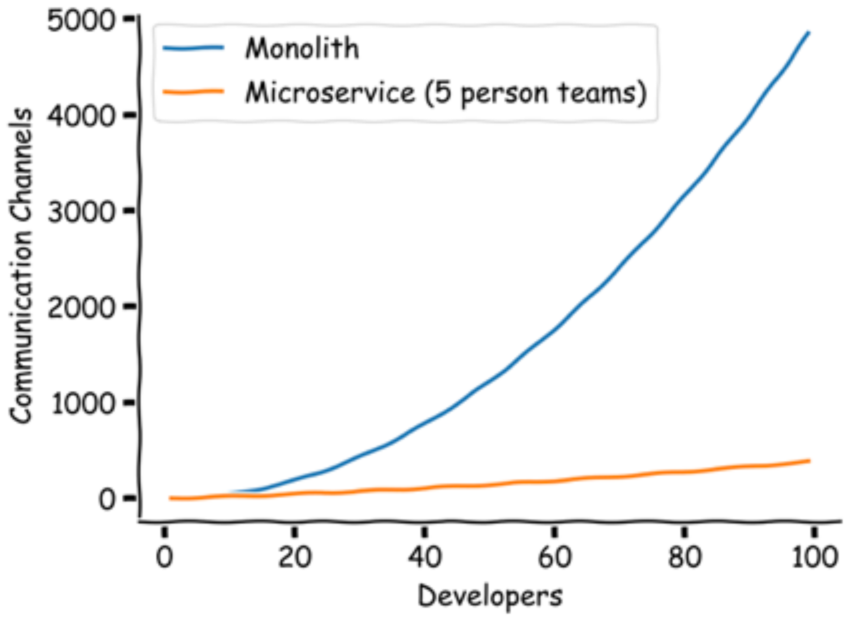
The win goes to microservices.

Round 8: Communication

Microservices are often defined in the size of the development team. Amazon for example has a "two-pizza rule," where a development team can be "fed with two pizzas" in the time it would take to create a microservice that corresponds to a team size of roughly 4-8 people. The idea behind that is, that the communication is limited to the team. Teams only need to communicate through service interfaces.

Long before the microservice idea was born, Fred Brooks wrote the seminal book "The Mythical Man Month". One of the takeaways of this book is that the number of communication channels increases with the number of team members. With a team of two people, there is one communication channel. With four people that could go up to six channels. Person one talks to person two, three and four. Person two talks to persons three and four and person three talks to person four. The formula for the number of communication channels is n(n - 1) / 2.

A team with 20 developers has 190 possible channels of communication. Splitting these developers into two-pizza teams reduces the number of communication channels significantly. If we take the example team of 20 developers and split it into four microservice teams with five people each, we have 10 communication channels per team. And the number of communication channels between the four teams is only six. The total number of communication channels is 46, which is roughly a quarter of the 20 people team. The plot below shows the number of communication channels for one big team versus individual microservice teams.

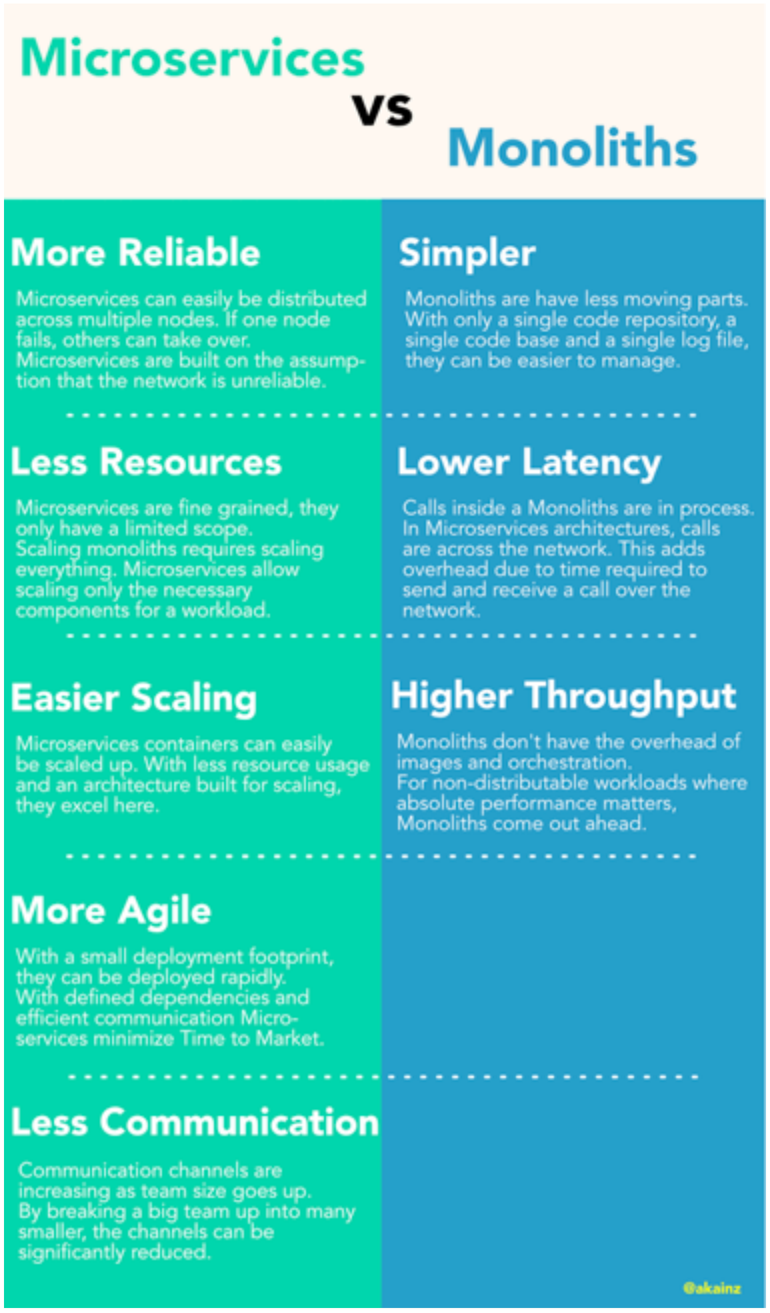


Monolithic vs. Microservices communication channels as total team size grows

At about 10 developers, the microservice model shows a clear advantage over the traditional model. At a team size of 50 developers, the number of communication channels is almost 10 times as high. Everyone that has worked in a team that size can attest to the fact that a lot of time spent on communication. A standup meeting alone, with 50 developers would be an exercise in inefficiency. Any development project over 10 developers is well served by breaking it up into smaller teams. Microservices are ideal for that, as they have well-defined interfaces along service borders.

Another clear win for microservices.

Who Is the Winner?



Microservices vs Monoliths

Microservices	Monoliths
More Reliable Microservices can easily be distributed across multiple nodes. If one node fails, others can take over. Microservices are built on the assumption that the network is unreliable.	**Simpler** Monoliths are have less moving parts. With only a single code repository, a single code base and a single log file, they can be easier to manage.
Less Resources Microservices are fine grained, they only have a limited scope. Scaling monoliths requires scaling everything. Microservices allow scaling only the necessary components for a workload.	**Lower Latency** Calls inside a Monoliths are in process. In Microservices architectures, calls are across the network. This adds overhead due to time required to send and receive a call over the network.
Easier Scaling Microservices containers can easily be scaled up. With less resource usage and an architecture built for scaling, they excel here.	**Higher Throughput** Monoliths don't have the overhead of images and orchestration. For non-distributable workloads where absolute performance matters, Monoliths come out ahead.
More Agile With a small deployment footprint, they can be deployed rapidly. With defined dependencies and efficient communication Micro-services minimize Time to Market.	
Less Communication Communication channels are increasing as team size goes up. By breaking a big team up into many smaller, the channels can be significantly reduced.	

@akainz

The results are in. Two wins for the monolith versus three wins for the microservices.

When looking at this chart, however, keep in mind it is relative. For example, the microservice only wins team communication for team sizes over 10 developers. A small startup team with five developers may be served well by a monolith. Monoliths are easier to manage with less moving parts. And a dedicated smaller team can also run a daily release schedule with a monolith. Scaling is only relevant over a certain amount of usage. If the products get a few hits per second, a monolith may be completely sufficient. Also if the calls move large amounts of data over the network, the performance hit may be significant and outweigh the other benefits. Microservices are not the cure-all for all development problems. Below are a few indications, that a microservice architecture might be a good fit:

- 24/7 reliability required
- Scale to beyond a few requests
- Peak and normal load are significantly different
- More than 10 developers
- The business domain can be divided into smaller domains
- Shorter lived operations
- Operations can be expressed as REST calls or queue events.
- No strict cross-service transactional requirements

In the end, Microservices are going to be a strong choice for many enterprise software projects.

Feature image by Jeff Jacobs from Pixabay.

CONTRIBUTED