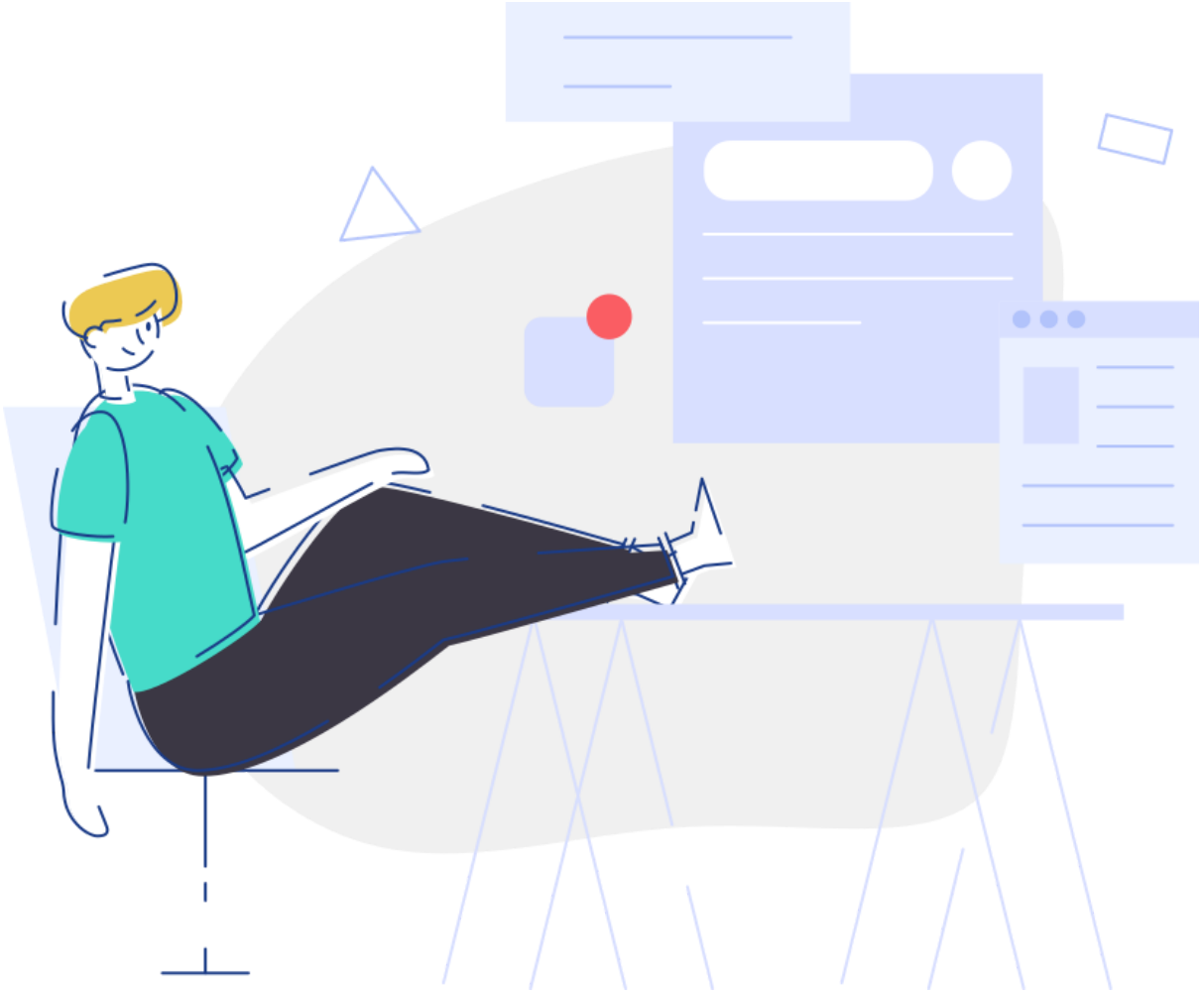
{ blocshop }

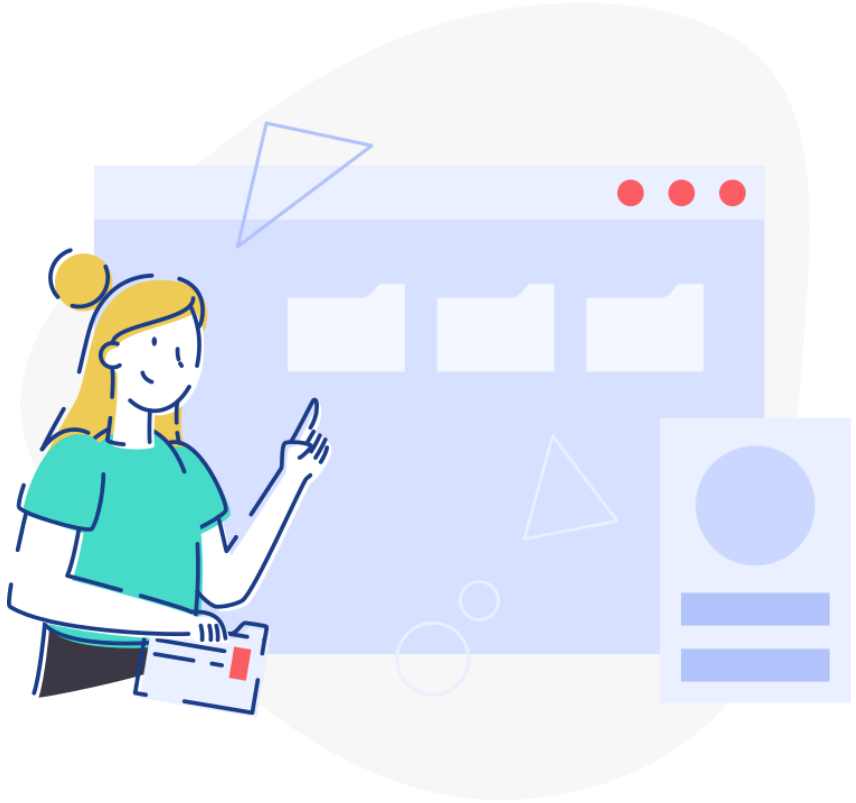
From monolith to microservices

How to break a monolith application into microservices



Written by Constantine Nalimov, Lead Architect at Blocshop

Content



1. Microservices or monolithic – which is better?

If you're about to begin a new project, you might be asking yourself whether you should go with a monolithic or microservices architecture. Or at least you should be asking yourself that question. We're going to give you some reasons to think seriously about going with microservices.

What is monolithic vs. microservices architecture?

Monolithic applications are built and deployed as a single unit. This is the traditional approach to creating applications: all modules are combined in one self-contained codebase.

All developers work on the same codebase and are committed to a single development stack, including languages, libraries, tools, and everything else

used to create the application. Changing any of these elements is a major challenge in a monolithic architecture. Any changes or fixes have the potential to cause problems for everyone involved.

Microservices take the opposite approach. With a microservices architecture, you divide an application into discrete components that can be developed and deployed completely separately. Any coding language can be used and the development teams can be small and autonomous. Each service represents a particular functionality in the application. If one service has a problem, it won't take down the entire application and fixes can be quickly rolled out by the relevant team.

Monolith vs. microservice: the showdown

Let's look at ten aspects of applications and see where monolithic and microservices architectures differ.

1. Size

A monolithic application is a single self-contained unit. Monolithic applications can get very big over time and your developers can end up dealing with a huge mass of code.
In contrast, microservices aim to be small independent services that focus on delivering a specific business objective.

2. Granularity

All the elements within a monolithic application are tightly coupled and interconnected. Changes made to any part of the application can have effects everywhere. A single bug can mean that the whole application fails.
Microservices are loosely coupled by design. If there's a fault, they can be isolated and fixed.

3. Ease of deployment

Monolithic applications are not easy to deploy. Making changes can be complicated and deployment is a big deal, involving the entire application. Microservices have small teams that can develop and deploy independently. They don't have to coordinate with everyone else involved in the project.

4. Speed of deployment

Not only are monolithic applications more difficult to deploy – they're also slower. It can take a lot of time to wait for all the changes to be ready and for large development teams to make sure that everything is ready to roll out. Those delays can be very costly, especially if your competitors have made the switch to microservices.
Again, microservices are designed to be the opposite. Leaner teams developing and deploying their own part of the application mean that microservices can indulge in rapid deployment. Some microservices can even go with continuous deployment and deliver really quick responses to feedback and feature requests.

5. Communications volume

Here's where monolithic applications have an advantage over microservices. Because everything in the application is self-contained, there's no need for remote calls between services.
Microservices can end up with significant communication overheads because of all the extra remote calls. While this isn't catastrophic, it does mean that communication between the microservices has to be designed and managed carefully to avoid bottlenecks because of overly chatty services.

6. Persistence of data

All components in a monolithic application share data storage. A single database sits at the lowest level of the architecture of the application and serves information to the entire application. That's a lot of work for one database and it can mean problems as it gets bigger over time.

One of the basic principles of microservices is that each service is free to choose its own data storage and to manage its own data. In fact, you should make sure that services don't share a data store so that you can avoid too much unwanted coupling between services. Like with communications, this can be a challenge. You especially need to rely on domain-driven design to make sure that no microservice can modify the databases of another. Keep each microservice focused on its own business objective.

7. Ease of onboarding

A monolithic application can become so complex that it becomes daunting for new hires to understand it all before they get started. It can be difficult for old hands to even explain how everything works!
With microservices, your new developers just need to understand the code used for their own service. Which means you get them onboard and coding faster and safer.

8. Polyglot programming

With a monolithic application, you're stuck with a single development stack for the whole thing. You can't decide to use a different programming language for one component or experiment with new approaches to storage solutions. Microservices give you the freedom to use a different technology stack whenever you want. You can experiment, optimize, and keep up with new ideas.

9. Communication method

Monolithic applications communicate with in-process calls and local method calls. As with everything else, it's all kept internal and dependent on the language used to develop the application.
Microservices make use of APIs and a modern RESTful approach to communicating with each other. The architecture is lightweight and technology-neutral, so again you end up with more freedom.

10. Scalability

So what about when your application needs to grow?
Monolithic applications can only scale horizontally. You can run multiple copies of the application. Additional servers, additional resources, and a lot of waste.
Microservices can be scaled individually as and when needed. Fewer resources get wasted and real business objectives determine where you add them.

At Blocshop, we've both converted monolithic applications to microservices and designed and built microservices from scratch. While we believe that the benefits of microservices are clear, we also know that there are horror stories out there about companies that underestimated the task of building or migrating. Contact us to find out how you can avoid making the same mistake. We'll guide you through how you can make the change, from quotation to deployment.

2. Top advantages of microservices

Starting a new project or dealing with an unmanageable behemoth of an application? It's time to decide whether microservices are right for you. To help you make the leap, here are our top ten reasons to choose a microservices future.

1. Easy and quick scalability

Need to scale up – or down? It's much easier when your services are separate. Just identify the services you really need to scale and leave the others untouched. Being able to scale dynamically means that you can grow when you have the demand, rather than trying to predict your needs at the outset.

2. Leaner teams

Separating services also means that you can use smaller teams that can work independently of each other. When you get to the point where you need dozens of developers, you don't want to have to force them all to collaborate and waste time in meetings. Small, autonomous teams can operate faster and more efficiently.

An extra advantage is that it's a lot faster to onboard new developers. That new hire no longer needs to understand absolutely everything before they can get coding. And you can try them out in less critical areas without worrying about their every keystroke.

3. Enhanced productivity

Those smaller teams will work a lot more efficiently, especially when your product really starts to grow. Not being forced to collaborate with absolutely everyone else lets your developers focus on their own codebase.

4. Flexibility to experiment and optimize

You can experiment when you know that you can add or remove features without risking the rest of your application. Want to try optimizing something? Go ahead. You can even decide to try out a new coding language or approach, safe in the knowledge that you aren't committing to reworking the entire codebase.

5. Freedom to try out new technology

This level of flexibility means that you can avoid getting locked into a particular technology or language. If something better comes along, you can try it out, confident that the team involved will be able to roll back the changes smoothly if it doesn't work out. As more modern technologies emerge, you can switch some of your services without replacing the whole application.

Microservices also enable you to use multiple languages at once. Each of your teams can choose what works best for their particular part of the project.

6. Faster deployment

When each of your services has its own discrete codebase, your teams can deploy faster. Each service can be deployed individually, without affecting other services or teams. Some teams might even choose to use a continuous deployment pipeline. Microservices give you the freedom to have a different deployment process for each part of your project.

7. Strength in isolation

If a service fails, it's not going to take down everything else. You'll still have to fix it, or use an alternative service, but at least you'll be dealing with a smaller problem than if you had to debug the whole application. Make use of this fault isolation and ensure that the system can survive failures in some services. You'll end up with a much more robust product.

8. Cleaner maintenance and faster response times

Maintaining lots of smaller, individual codebases means that you can keep them lean and clean. You can avoid the accumulation of code that can impact response times and consume resources and slow down your application. Optimizing communication between services can even reduce downtime and let your product get on with what you designed it to do.

9. Reduced costs

Even with all of the above advantages, some critics of microservices often argue that the approach is more expensive. Part of the problem is that microservices are relatively new, so they require a shift in thinking. The upfront costs of this switch can seem high or even impossible to quantify. As with the move to agile development practices, the investment in developer culture will pay dividends. From less expensive hardware to more stability, your costs will be lower, often in surprising ways.

10. Microservices – truly agile

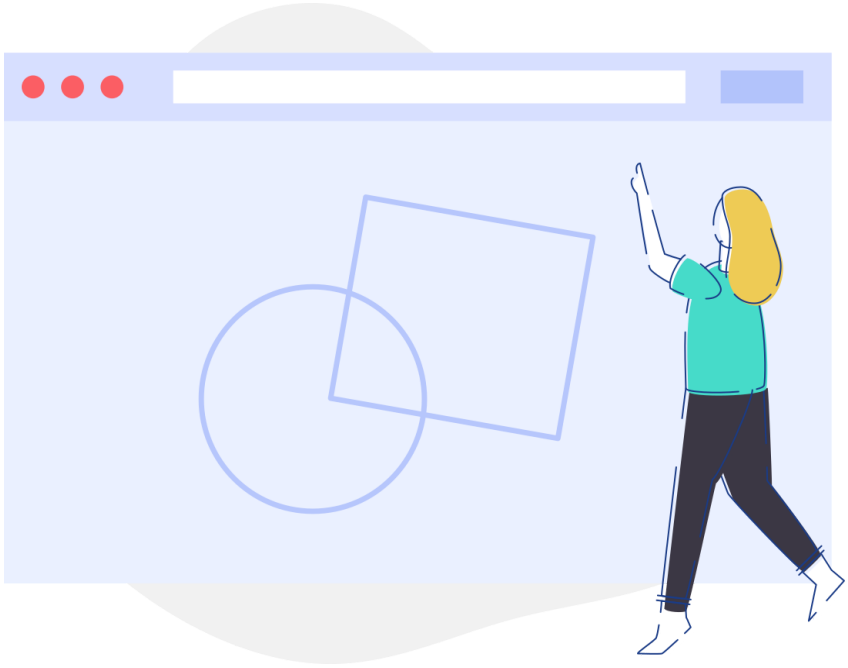
Leaner teams, rapid deployment, easy scalability, and incredible flexibility. Add these together and the combo bonus is that microservices closely align with an agile development approach. Over the last couple of decades, agile development has revolutionized software. Microservices can do the same for architecture, with the same kind of repayment in efficiency, speed, and quality.

So what's the catch?

Okay, we have to admit that there are some challenges you might face when using microservices. But in many cases, these are problems that would simply be shifted to a different level or stage if microservices were out of the picture.
An application built using microservices is complex because it is a distributed system. Communication between services must be handled carefully, including dealing with partial failure and data bottlenecks. Testing can also be more challenging with microservices, partly because the architecture gives you more options on how and where you can test. Areas that previously might have been closed off are now exposed.

Because of the inherent complexity of splitting up an application into multiple services, microservices can be more resource-intensive and take longer to develop, especially if you don't have enough experience in managing and orchestrating multiple teams. It is vital to have the right experts in place to ensure that you aren't overwhelmed by that complexity and can enjoy the long-term benefits of microservices.

We love microservices at Blocshop. Plus we have years of experience in implementing microservices solutions and migrating legacy systems to microservices architecture. So you should talk to us when you make the (incredibly great!) decision to switch to microservices.
Already thinking of changing from a monolithic monster to clean, lean microservices? Just want to discuss your use case and see whether microservices are the way forward for you? Talk to us today.



3. APIs and microservices

Are you confused about the differences between microservices and APIs? Read on for a quick and clear explanation of where they fit into modern approaches to web applications. And how you can use them both to supercharge your development process.

What's an API?

An Application Programming Interface (API) is a system of structured communication. It lets an application get information from an external service or product using a set of commands. These commands and the information they produce are predictable and usually exhaustively documented.
Here are some examples of APIs in the wild:

- Need to get a list of all restaurant addresses in your town from Google Maps? Use the Google Maps API.

- Want to gather account activity from Twitter or search for keywords in old Tweets? Use one of the many Twitter APIs.
- Want to add secure online payments from PayPal to your application? Use the PayPal API.

APIs include protocols for requests and responses, formats for the transfer of data, and conventions to be followed when interacting with their respective applications.

Ultimately, APIs allow applications to talk to each other and get what they need. Because the API is standardized, you can be sure that you'll get a particular response to your request. That makes it easy for you to feed that response into your application and use it for whatever you need.

What are microservices?

Until relatively recently, applications were always programmed as monolithic, self-contained entities. As some of these applications became more complex and unwieldy, it started to make sense to break them up into smaller components. Carve out some functionality, create a separate application, and enjoy the benefits of a smaller codebase, leaner team, and independent development pipelines.

These separate components are called microservices.

Microservices operate independently, so they can use different coding languages and even be replaced or improved without affecting the application as a whole. Microservices even use different databases. They really focus on their own area.

Large companies have been especially active in using microservices. Smaller teams and projects can be easier to manage when you're dealing with a vast ecosystem such as Netflix or Amazon.

How do APIs and microservices work together?

APIs enable external applications to communicate, but the same process can be used when you break up a single application into microservices. Microservices also use APIs to communicate.

What's the difference between microservices and API?

Microservices are an architectural approach to designing an application. They are part of a design decision whereby you use small, independent services for individual functionality.

In the microservices architecture, APIs are the communications framework that you use to enable each of the microservices to exchange information.

For example, an online store might be broken up into the following microservices:

- Product catalog – this contains all the items sold in the store.
- Orders – when someone orders a product, the order gets tracked here.
- Payments – the customer pays for their order and the payment is processed in this service.

You can probably already imagine several other microservices, but let's stick with these for now.

There are inherent advantages to keeping all of these areas separate, especially in a big online e-shop. If there's a change to the structure of the products offered, there's no need to mess with the way orders and payments services are coded. The same goes for if something new needs to be added to how payments are processed. There's no need to change *Products* or *Orders* just because you have to update *Payments*.

So our online store has three microservices. They need to talk to each other.

A customer browses a product and orders it. *Products* need to tell *Orders* what the item is so that it can be correctly pulled from the warehouse shelf and shipped.

When the customer finally decides to pay for the order, *Orders* need to interface with *Payments* so that the correct amount is billed.

None of these conversations need to be very complex. Most of the time, each service is just exchanging a small amount of data and updating its own separate database.

That's where the API comes in.

Each microservice almost certainly has its own API – its own set of rules for communicating with it. When *Products* talk to *Orders*, there are particular pieces of information that *Orders* need to receive. It probably isn't interested in user reviews, for example, but it does need to know the current price of the item.

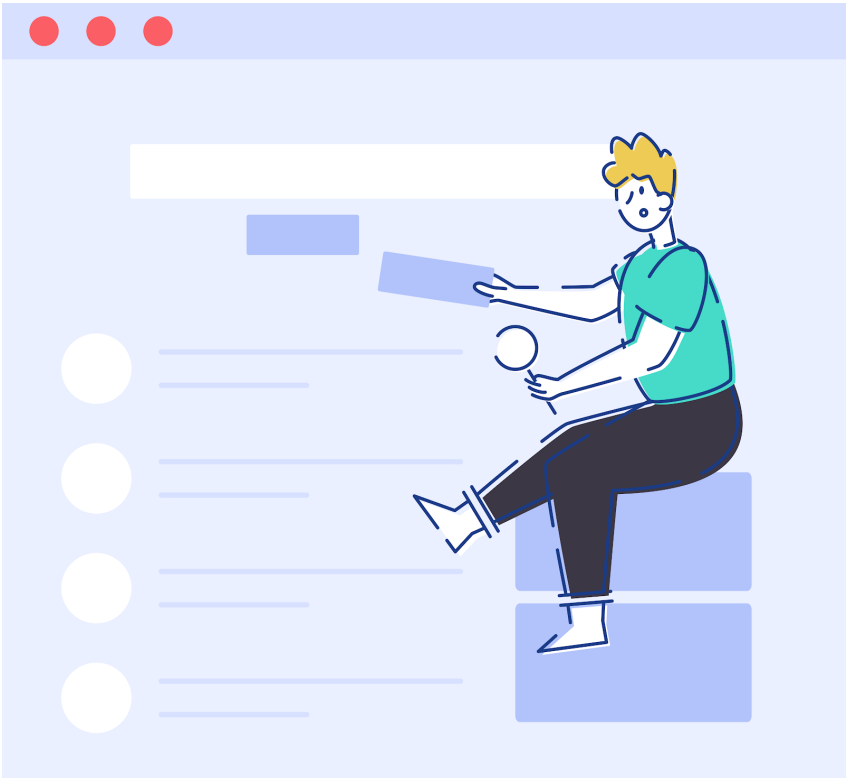
So a microservice needs to use APIs to talk to other microservices, but APIs are more neutral than that. As we saw above, they can be part of the microservice itself, but they can also be part of other external applications, such as Google Maps, Twitter, and PayPal.

Even better, because of the way that APIs can talk to services without worrying about the internal workings of those services or how they store data, some microservices within an application can be opened up to third parties.

In the same way that Google or PayPal allows communication with their services, our example online store could be given an API to enable another web application to serve data to online reseller or reviewing applications. This kind of integration can become incredibly powerful and lead to new, unexpected uses for data – and increased exposure for the company serving it. For some companies that provide really useful data, APIs can even be monetized directly.

Microservices and APIs are made for each other. The decentralized nature of microservices lends itself to a need for a framework of lightweight, specialized communications. When you get microservices and their APIs really working together, you might be surprised at just what you can achieve.

Blocshop has extensive experience in creating microservices from scratch and converting legacy monolithic systems to microservices. We also have the right know-how when it comes to creating the APIs you need to get everything talking and operating smoothly. Get in touch with us today to get a quote for your project, or even if you just want to have a chat about how microservices could help your business.



4. SOA versus microservices

Service-oriented architecture (SOA) debuted in the 1990s. Its goal was to address the problem of ever-growing codebases and complex systems. The idea was to break these up into components that would deal with specific business objectives. Focused on the enterprise level, SOA is based on the concept of reusing software and code to avoid waste.

Microservices architecture can, at first glance, seem very similar to SOA. It also has the goal of breaking up monolithic applications into components. A core difference is that microservices do not need to fulfill enterprise-wide requirements. Each microservice does what it does and does it well, without worrying about how it fits into the bigger picture.

What are the advantages of microservices architecture over SOA?

Microservices architecture in some ways evolved from SOA, or at least was able to learn from the mistakes made, and benefit from new practices and technology, including agile approaches to software development.

Microservices are faster, more robust and reliable, easier to deploy and scale, and just plain leaner than any SOA. Teams are small and independent and there is a refreshing lack of interdependencies that lives up to the initial dream of SOA and takes it to a new level.

Read on for six reasons why the microservices of today are nothing like the SOA of yesterday.

6 big differences between SOA and microservices

1. Communication

How SOA and microservices architecture communicate is one of the starkest illustrations of the differences between them. Each service in SOA must use the enterprise service bus (ESB) to communicate with other services. This should standardize and simplify communications with a single, centrally managed system. But it also means that the ESB acts as a bottleneck to development, with updates and integrations for components ending up queued. Microservices instead use an API layer, an approach that relies on simpler, and quicker, exchanges of information between separate components.

2. Granularity

The services in SOAs can be big or they can be small. They can even be massive and shared across the entire enterprise. SOAs were never designed to be as small as possible, so the services used can be considered coarse-grained in terms of granularity. Both the scope and size of the service can be vast, with shared databases, and multiple interdependencies. On the contrary, microservices are designed to be highly focused and fine-grained. Small teams, faster onboarding for new hires, fewer potential points of failure.

3. Coupling and cohesion

When you have small, specialized services, you can afford to have low coupling and high cohesion. Each microservice is based on the idea of "bounded context", in other words, it is self-contained, with as little sharing of components, data, or interdependencies as possible. This results in high cohesion, which means that the origin of problems can be rapidly identified as arising from a specific microservice. This component can then be isolated and fixed. In SOAs, despite the stated goal of loose coupling, the sharing ethos proves to be a vulnerability. Not only can there be an impact on speed, but faults can be harder to trace and fix.

4. Reuse

Reuse of components is built into the philosophy behind SOA, with shared services prioritized in importance over highly specialized implementations. Because of the enterprise-level focus of SOA, sharing is evaluated as reducing waste, saving time and money. Unfortunately, this misses out on a big part of the whole picture, where sharing also creates vulnerable interdependencies and slows down development. Microservices take into account that it is usually better to work with smaller components and keep everything separate. If you reuse something designed for another purpose, not only will you be dependent on its development schedule, your component will fail when it fails. If microservices need to reuse code, the preference is to duplicate it and disconnect its development from the original.

5. Data duplication and storage

Building on the concept of reuse, SOA usually prefers to share data. This extends to the sharing of databases between components. All services must access and change data that is effectively stored in a central source. This reduces the need to duplicate data, but it can result in speed problems and inevitably leads to interdependencies between services, as there must be agreed-upon protocols for how the data is stored and accessed. In contrast, microservices use whatever data storage is best suited to the task and aren't shy about duplicating data whenever necessary. While this can add complexity in

terms of data management and keeping everything up to date, the speed boosts of having local or exclusive access to data mean that microservices again win out over SOA.

6. Deployment

The focus on keeping microservices separate and small means that the teams can be smaller. These independent teams can work at differing paces and deploy their modules without worrying about the deployment schedules of other teams. They can even progress to continuous deployment, with the feature and bugfix pipelines reaching the end-user much faster than ever before. SOAs, on the other hand, suffer from the kind of deployment issues that plague monolithic systems. If you add or change a service, the entire application needs to be redeployed.

Are microservices a subset of SOA?

Both service-oriented architecture and microservices deal with the concept of converting monoliths into discrete modules. SOA came first, so it might seem natural to think of microservices as a subset of its predecessor. But SOA was designed to enable different applications of varying sizes to communicate, whereas microservices come together to form a network of small modules that can be considered an application. SOA is in some ways still stuck in the mindset of the monolithic era, with coarse-grained, big services that are designed to share a lot of functionality. Microservices are leaner, more independent, and fine-grained. One way to think of the relationship between the two is that microservices architecture learned from the shortcomings of SOA and managed to deliver the promise of truly breaking apart monolithic applications.

Blocshop has been breaking monoliths into microservices since the software architectural style got its name in 2012. We know our microservices and we can walk you through the process of deciding how to make them work for you. Contact us today for a project estimate or even if you just need something explained.

5. How to convert a monolith to microservices

Is microservices architecture better than monolithic architecture?

Microservices architecture has only been around for less than ten years, but we at Blocshop think – and sure, we're a little biased – that microservices completely knock this question out of the park. Monoliths had their time, but they're unquestionably unable to compete with microservices – and that's why companies like Netflix, Twitter, Uber, and other giants are turning to microservices. And why we're getting more and more clients coming to us and asking for a conversion of part or all of their aging monolithic systems.

What are the big differences between microservices and monolithic?

If you've got a monolith, you've got a big, lumbering beast of an application. Your developers will simultaneously work on the same codebase and need to coordinate all changes and deployment. Any bottlenecks or failures have the potential to bring the whole beast to its knees. Over time, it will become increasingly complex and difficult to maintain. New developers will struggle to understand how it all works. Your application will grow stale as it ages and new technology can't be integrated.

With microservices, you have a bustling hive of separated services communicating using simple commands. Each microservice focuses on doing a single task well and the application emerges from their cooperation. You have small, lean teams that can deploy independently, and you can keep up to date with new ideas and practices because each microservice can be redesigned as needed, without affecting the rest of the application.

Why go with microservices?

If you want your application to be faster and easier to maintain, you need to make the switch to microservices. If you want smaller, more focused teams that can operate independently, you need microservices. Here are just some of the ways in which microservices will transform your business:

Scalability: Grow as you need to. With microservices, you can scale up parts of your application without worrying about what happens elsewhere. Maybe part of your business hasn't developed as rapidly as another part. Scale up what you need to and don't touch the rest.

Flexible and efficient: Want to try a new technology or programming language? That's not a problem when your microservices don't depend on each other. Try it out and see what happens when you roll it out. You don't need to reengineer anything in the rest of the application.

Maintenance and debugging: Each microservice consists of a small codebase that can be cut off from the rest of the application in the event of failure. Debugging is faster and the code can be kept tight and tidy.

Deployment: Smaller teams that don't need to coordinate their development means that you get faster and more robust deployment. Features and fixes get to your users as soon as they're ready.

Blocshop's step-by-step guide to converting from monolith to microservices

So you've got a monolith, but you want microservices. Where should you start? Our quick-start guide will highlight the milestones you can expect on the path to modernizing your application.

1. Carve out the simple stuff

Even if your goal is to shrink your monolith until it finally disappears and its functionality is replaced by microservices, it makes sense to start small. Identify parts of the application that can be easily carved out and built as a microservice. Not only will this provide you with some of the immediate benefits of microservices in that module, but you'll also gain valuable experience in the planning and implementation of microservices. You'll be able to establish a workflow for turning other parts of the monolith into services.
The first services you build might be those which are already relatively well decoupled from the monolith and won't affect any front-end or user-facing elements. It could even be a good idea to avoid services that require a data store of any kind, so you don't need to deal with that side of the process – yet.

2. Reduce the dependencies of the new services

You can already begin to reap the rewards of your new microservices, but only if you try to minimize their dependency on the monolith. This is easier to do if you were careful about choosing modules that could be effectively carved out in the first step. Once you have microservices that no longer depend on the monolith,

they can enjoy their own release cycle, faster speeds, and keep focusing on the service they were designed to carry out.

3. Break it down vertically

At this point, you can start to look at the verticals of your monolith. You want to minimize dependencies between your microservices and one way of doing that is to have teams that completely own their microservice on every layer. Instead of team members working on a range of different services, they build and own the whole vertical, from front-end to data storage. Which brings us to step 4.

4. Break up data storage

Keeping a shared central database is going to undermine your move to microservices, so you need to break it up. Each microservice should have the freedom to determine its own approach to data storage and not have to worry about what other services are accessing and changing it. You might find that data replication is the best way to do this. Once you liberate the data, you should start to see real speed and efficiency boosts.

5. Decouple important and constantly changing elements

Even if your goal is to eliminate the monolith, there's a cost-benefit analysis to be made. Some parts of the monolith should be replaced earlier than others, especially if they already need to be updated more frequently, or if they're vital to the business. Identify what has changed the most and consider your business objectives when working out what capabilities need to be converted first.

6. Identify domain boundaries

Domain-driven design is at the heart of microservices and this is where it will save you from going too far with microservices. As you continue to carve out and decouple capabilities from the monolith, consider each step from the point of view of the domain that governs it. Remember that microservices that are too micro can have their own management challenges, so don't break up a service in a single domain until your monolith is already starting to fade into the background.

7. Continue to evolve

You may never completely see that dream of the monolith dissolving into a plethora of fast, efficient microservices, or at least it may take you a lot longer than you expect. As with much of development, the process is iterative and incremental. Your application will evolve towards its final state, with each microservice serving as a step in that evolutionary process and with your teams learning more and your application becoming more effective with every step.

How Blocshop can make microservices work for you

Blocshop has been breaking monoliths into microservices since 2012. We know our microservices and we can walk you through the process of deciding how to make them work for you. Contact us today for a project estimate or even if you just need something explained.

Constantine Nalimov has been the lead developer and architect on each of the projects that the Blocshop team has delivered over the years. Constantine is an evangelist for microservices and Domain Driven Design.