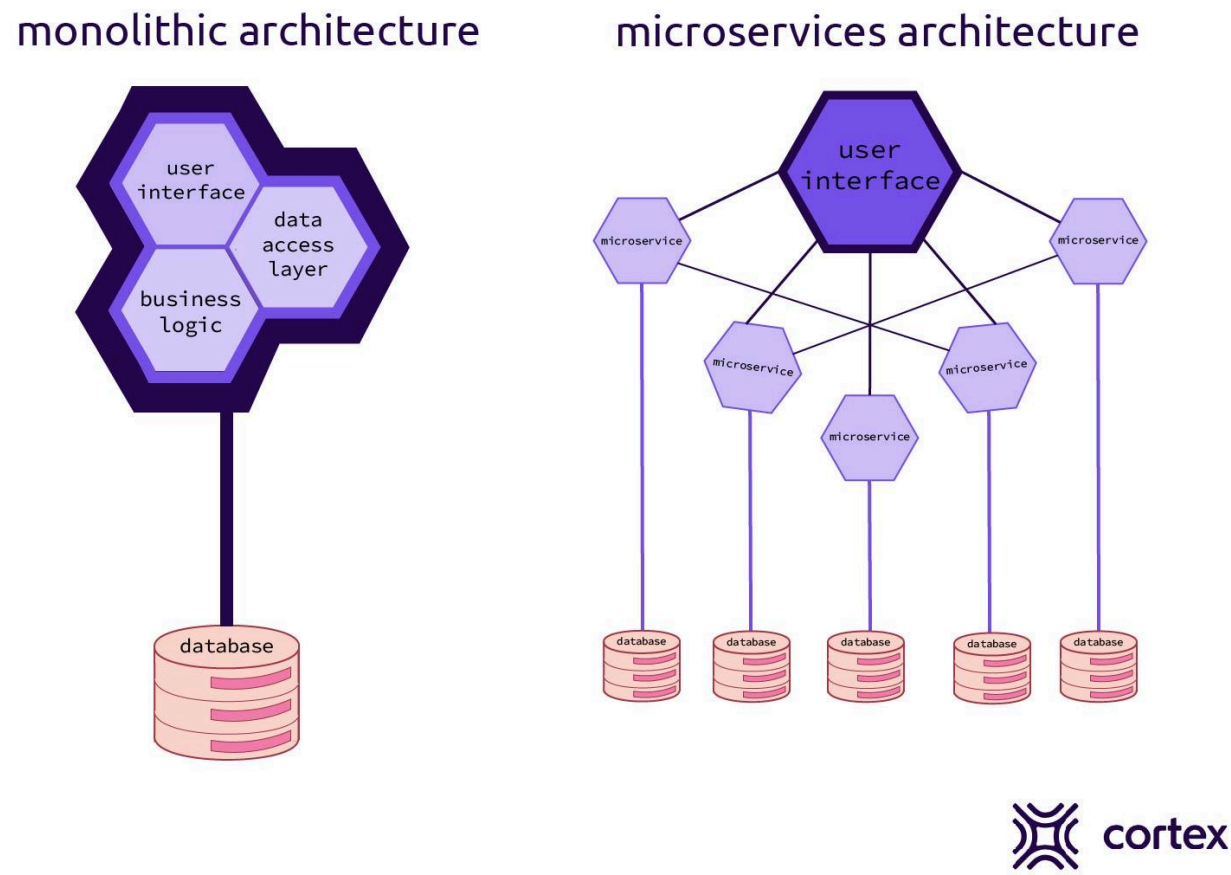
BLURB: Learn the differences between monolithic and microservice architectures, their pros and cons, and the complexities of scaling each.

Monoliths vs. Microservices: Differences, Pros & Cons, and Choosing the Right Architecture



Monoliths vs. microservices

Monolithic architectures operate as a single unit, where all of the application's functionality lives within the same codebase. Microservice architectures break components into individual services, each of which performs a specific function of the application.

As new technologies emerge, digital transformation — the act of adopting these new technologies — becomes more crucial for organizations of all sizes, from startups to decades-old enterprises. To stay competitive, you must consider how your company will adapt to emerging tech and how your applications will evolve, too.

The exact architectural style you adopt for your application will depend on your unique needs, but monoliths and microservices are predominant architecturals. To determine which is right for you, you should first understand the differences between monolithic and microservice applications, as well as their advantages and disadvantages.

What is a monolithic architecture?

In a monolithic architecture, all of the distinct components of an application, from business logic to user interface, exist within the same codebase. A monolith uses a single programming language, a single repository, and a single environment, so changes will impact the whole application. A monolithic architecture makes use of tight coupling, so components are highly dependent and interconnected.

Monoliths have been the standard architectural style for some time, and they remain a solid choice, especially for small organizations and small teams with a few engineers. Monoliths are also ideal for applications that won't require many updates over time.

Pros of monolithic architectures

While monoliths may sometimes be overlooked because of their legacy status, they have a number of benefits.

- **Simple development.** A monolith is the standard for a reason — all of the code lives in one place, so it's easier to build upon. New team members will be able to pick things up faster.
- **Simple debugging.** Because all of your code is in one place and your service has no dependencies, it's easier to identify the source of an issue. Developers will have an easier time recreating environments for testing.
- **Standardization and velocity.** Standardization has become increasingly important. Monolithic architectures establish that standard through a singular codebase, keeping data centralized.

Cons of monolithic architecture

Monoliths may have an edge when it comes to simplicity, but that doesn't make them the ideal architecture for every application. There are a few downsides to monoliths:

- **Scalability.** We'll dig into this more soon, but with a monolithic approach, you can't scale individual components. Even if you're adjusting a single component, you have to retest and deploy the whole application.
- **Slow development.** With an entire team working from the same codebase, developers have to tread carefully, which can slow them down. Testing becomes even more important — and more painstaking — since a single issue can impact the entire application.
- **Tech stack lock-in.** By using a single programming language and a single

repository, your developers are locked into a single way of doing things. With monoliths, you don't have the flexibility to adopt new technologies as they emerge.

What is a microservice?

In a microservice architecture, all of the components of an application are broken into independent, loosely coupled modules with distinct functions. Each module, or service, has its own repository, its own logic, and its own deployment process. Independent services interact with one another through interprocess communication mechanisms, often APIs. Although these services are autonomous, because of their limited scopes, a microservice application involves a number of dependencies, or services that rely on other services for data.

This modularity promises greater scalability and agility, which is why the microservice application has emerged as a popular alternative to the monolithic approach. Because services are autonomous, it's easier to update and replace individual services and spin up additional services when demand spikes. As scalability becomes more and more of a concern, it's no wonder that organizations like Netflix have adopted a microservices approach.

Pros of microservices architecture

Microservices can streamline your operations, especially if your organization has naturally divided into smaller teams with distinct domains. There are a few other significant benefits of microservices:

- **Fault isolation.** Services operate independently, so a bug is less likely to take down your entire application. Developers can experiment with new services without impacting existing service.
- **Independent deployments.** We'll discuss this more below, but the autonomous nature of microservices allows developers to deploy services independently, so they don't have to deploy the whole application for every small update.
- **Flexible tech stack.** Microservices are language agnostic, giving your devops team greater freedom to use the programming language that makes sense for each service. With a microservice architecture, it's easier to adopt new technologies with simple upgrades.

Cons of microservices architecture

There's a lot of buzz about microservices, and for good reason, but the complexity of this approach requires greater coordination across your team and comes with a few disadvantages.

- **Managing distributed services.** As your application grows, you'll develop dozens, if not hundreds, of services. Managing this many services and their dependencies can be overwhelming without the right tools. You need dedicated devops teams that can handle all aspects of a service, from programming to deployment.
- **Testing and troubleshooting.** Deployment may be faster, but because of the dynamic nature of microservices and dependencies, it can be challenging to recreate environments for testing.
- **Barriers to bulk changes.** If you want to make sweeping changes to your services, you'll need to update each one individually. It's entirely doable, but does require more development time.

Performance and scaling with microservices vs. monoliths

The distributed nature of a microservice architecture makes it highly scalable, both as demand grows and as your devops teams expand. Microservices allow you to scale up specific parts of the application, so you have comprehensive control over how the app is performing at any time. With tools like Kubernetes, you can not only gain insight into how each component is performing, but you can establish autoscaling protocols for seamless performance, even during peak demand.

In comparison, a monolithic application is difficult to scale because of its tightly coupled components. You can't isolate a specific component of a monolith — you have to scale the entire application, which can get expensive quickly. As features are added, scaling the unwieldy codebase only becomes more challenging.

While the scalability of a microservices approach is enticing, there are serious operational roadblocks to consider. Unless you have a catalog or other means of tracking your dozens of services, managing a distributed system can quickly become overwhelming for team members, leading to the neglect of certain services. Because this architectural style relies heavily on APIs, microservices are also more vulnerable to third-party outages or performance issues due to poorly designed APIs. This is where monoliths have an edge — in a monolith, all calls are local, so you don't have to worry about network latency or failure in the same way.

Deploying microservices vs. monoliths

The modularity of a microservices architecture allows for the independent deployments of individual services. Developers can upgrade the services that need attention without deploying the entire application, enabling them to move faster. The loose coupling of services allows for rolling deployments, so the application can be upgraded one component at a time without any downtime. With a microservice approach, teams can establish a robust CI/CD process and stay competitive.

Microservices have an edge in deployment, but only if the complexity doesn't become overwhelming to engineers. Monoliths, on the other hand, operate from a single codebase, so they tend to be easier to build and deploy, which means developers also enjoy a simpler workflow. However, altering one component impacts the whole application, which opens the door opens for bugs and unexpected behavior when deploying changes.

Security with monoliths vs. microservices

With a monolithic architecture, the whole application is at risk if it falls under attack. A traditional firewall will provide adequate protection for a small monolith, but large, complex applications will be more vulnerable.

With a microservices architecture, though, the threat is spread across all of the individual services. In most cases, the entire application will continue to function, even if one module comes under attack. There are more potential sites for attack, though, so software development teams should carefully consider the security of each service. Because third-party integrations and APIs are critical parts of microservices, authentication protocols and encryption are particularly important.

Cost of monoliths vs. microservices

When it comes to cost, it's ultimately a matter of when you'd like to pay. Monoliths have a low upfront cost, but are expensive to develop and scale. Microservices, on the other hand, come with a significant upfront cost, but their scalability makes this a cost-effective solution in the long run.

Ops and collaboration with microservices vs. monoliths

The distributed nature of a microservice application means that your teams will also be

distributed. Teams should take ownership over specific services and are expected to handle all aspects of programming, deploying, and maintaining their services. Even though teams operate independently of one another, just like services have to communicate with one another, communication between these teams is highly important. Leaders will need to make a concerted effort to encourage collaboration between teams to make sure standards are being met.

With a monolithic application, everyone is working from the same codebase, so collaboration may come more naturally. However, in many cases the structure of a monolith doesn't align with the structure of an organization, which can lead to operational challenges both within and between teams. Team members may also handle just one facet of development, rather than overseeing and entire service, which can lead to blind spots.

Microservices vs. monoliths: Use cases

Microservices certainly have their advantages, and it hasn't hurt their reputation that companies like Amazon and Uber have adopted this architectural style. The widespread adoption of microservices by leading engineering organizations has generated a lot of buzz, but that doesn't mean monolithic architectures have totally fallen by the wayside.

Netflix: A microservice use case

In 2008, a database corruption led to a three-day outage at Netflix and they were unable to ship out any DVDs. This incident caused them to completely reconsider their infrastructure. They began migrating to AWS Cloud and became one of the first major enterprises to adopt a microservice approach.

It took seven years for Netflix to complete their migration from a monolithic architecture. In that time, their user population grew by eight times, and individual users began consuming more content on average. According to Netflix, "Supporting such rapid growth would have been extremely difficult out of our own data centers; we simply could not have racked the servers fast enough."

Cloud-based microservices enabled Netflix to dynamically scale to meet demand. Not only could they spin up thousands of virtual servers within minutes, but they were able to expand their service to over 100 countries. In the process, they significantly reduced costs without even trying to.

It did take the organization nearly a decade to make the transition, which is a testament to the effort required to migrate to microservices. Netflix chose a cloud-native approach

to migration, which meant they essentially overhauled all of their technology and operations. This required serious coordination and centralized efforts, as well as the willingness of everyone to learn along the way.

Netflix attests that the migration was well worth the effort — not only were they able to expand their service, but they've created a more reliable service in the process.

Segment: A monolith use case

Segment, a customer data platform founded in 2011, adopted a microservices architecture early on, but later scrapped it for a monolith. Segment's devops teams were quickly overwhelmed by the complexity of microservices, and spent so much time managing the distributed services that their velocity took a nose-dive.

Segment's software "ingests hundreds of thousands of events [customer data] per second and forwards them to partner APIs." Initially, events were assembled in a queue, then dispatched to the appropriate destination. With a microservice architecture, this meant that there was a separate repo for every destination.

The team created shared libraries to manage their dozens of repos, which made it more difficult to test changes, since deployments impacted every destination. Eventually, destinations were using different versions of the shared libraries, so the team was left with the insurmountable task of managing over 100 distinct services with unique load patterns. Auto-scaling essentially became a manual task.

They decided to consolidate all of the services into a monorepo and merge the queues for each destination into a single service. For this organization, it made more sense to collect all of these destinations, rather than deploy over 100 individual services for a single change to a shared library. This dramatically increased the team's velocity, freeing developers up to make proactive improvements.

Making the choice that's right for you

Ultimately, it's not really a question of whether a microservices or monolithic approach is objectively better — it's a question of what's better for your organization and team members. Before making that decision, consider how your teams are naturally structured and the ways they already work together.

If you're encountering operational issues with large teams managing unrelated features within the same codebase, then it likely makes sense to break those features into microservices and split the team accordingly. This allows individuals to become experts

with their services, which can improve velocity.

On the other hand, if you have a small development team that works closely together, a monolith will likely be easier for them to manage than a sprawling distributed system of microservices and dependencies.

If the application you intend to build is relatively simple, then keep it simple: use a monolithic architecture. Simple monoliths also have a brief time to market, so if you're trying to develop an application quickly, this is the best way to get it off the ground.

However, if you anticipate performing a series of upgrades and updates, a monolith may quickly become untenable. A microservices approach will give you greater flexibility, and the long-term scalability may be worth more than entering the market quickly.

No matter how you're structuring your application, it's critical that you have a means of monitoring its performance. Cortex offers a suite of solutions that provide visibility into individual services, so it's easy for your teams to deliver high-quality software. With features that drive accountability and maintain centralized information, you can make informed decisions and improve performance. Book a demo of Cortex today to see how we can help you understand and improve your service architecture.

When to migrate from monoliths to microservices

Don't migrate to microservices just because your competitors are doing it — if your devops team is working in new systems and rarely interacts with your monolithic app, then there's likely no need to migrate. If, on the other hand, your legacy system is becoming difficult to maintain, or if your teams are already operating in distinct domains, then it's probably time to consider a migration.

Before embarking on a migration, keep in mind that it doesn't have to be all or nothing. You can experiment first by breaking your monolith into macroservices, which maintains the functionality of your application, but primes it to be broken out into smaller microservices later on.

Take an incremental approach and transition a few features into microservices, rather than trying to overhaul the whole application. Don't set a deadline for the migration — take as much time as you need to make sure everything works as intended before deployment.

An incremental approach allows your developers to adapt over time. They'll face responsibilities they may not have encountered with a monolith. For example,

microservice communications require a network, so team members need to be prepared for a serious uptick in traffic and logging. They also need to be ready to monitor and manage dozens of distributed services.

What about service-oriented architectures?

A service-oriented architecture (SOA) is very similar to a microservices approach — an SOA breaks an application into small components with specific responsibilities. The primary difference between the two is scope: SOAs extend beyond the application and apply to your entire enterprise.

Rather than communicate through APIs, services in SOAs communicate through an enterprise service bus (ESB), which requires greater coordination. It also enables services within SOAs to adopt broader responsibilities than microservices, which are highly specialized. An SOA will make more sense for large environments, while microservices are better for small environments, like mobile and web apps.

What about serverless architectures?

Serverless architectures allow your developers to code without considering infrastructure, handing off the responsibility of managing servers to a third-party cloud provider. This is not only cost-effective, but allows engineers to focus on innovating without worrying about managing a server.

At the same time, serverless architectures come with their own challenges, primarily when it comes to loss of control and security risks. For this reason, a serverless approach makes the most sense for short-term tasks, or when traffic is unpredictable.

Improve service with Cortex

There's quite a lot involved in choosing the right architectural style for your application and organization, and at Cortex, we understand the multitude of challenges that service-based architectures bring. You can trust us to help you transform the way you build your software — just book a demo today to see how we can help you streamline your operations and maximize the performance of your services.