Microservices:
Improve Your Application Performance, Scalability and Deployment Speed

- Introduction .................................................. 2
  Where we are today

- Comparative Anatomy ................................. 4
  Microservices vs Monolithic vs SOA

- A Strategic Approach .................................. 14
  Decoupling monolithic applications

- Hitting Microservices Out of the Park ............. 20
  The right organization and the right methodologies
Today’s software teams feel the pressure to be more agile, more flexible, and to continually increase the velocity of their development efforts. As a result, many businesses are moving away from monolithic application architectures that no longer meet the fast pace and competitive nature of the digital world.

To take on the challenge, many software teams are turning to microservices, which have become a trend in recent years as businesses strive to achieve the same success of large enterprises—such as Amazon, Netflix, Uber, and eBay. By decoupling monolithic applications and adopting microservices, these companies have greatly increased application scalability as well as deployment speed and frequency.

As an architectural style for application development that first emerged nearly 15 years ago, microservices began to be applied by software programmers to production applications in the past decade. Viewed as a way for applications to evolve beyond monolithic architectures and as an improvement over service-oriented architectures, microservices involves the building of independent modules to address specific tasks and business objectives.

While there is no consensus as to what the typical characteristics of a microservices architecture comprise, there are some necessary attributes to achieve the desired application benefits:

» Communicates over the network by utilizing technology-agnostic protocols.

» Provides the flexibility to deploy services independently.

» Maps to the business goals of the application.

» Leverages polyglot programming and persistence.

» Decentralizes governance and data management.

» Provides a high level of failure resistance and fault tolerance.

Applications built using a microservices architecture feature a modular structure. Each service is small in size, contains all the necessary components to independently execute its designated function, and exhibits low-coupling and
Microservices applications also offer enormous benefits from the standpoint of being fault tolerant and independently deployable. In addition, new and improved functionality can be frequently deployed and continuously delivered. The big payoff is that businesses gain the ability to transform and accelerate the way they approach software development.

In this eBook, we compare microservices to monolithic and service-oriented-architecture (SOA) applications and discuss how to know when microservices is the right choice for one of your applications. From there, we offer a few tips on getting started, a high-level overview of the process for converting a monolithic application to microservices, and how to build the right team and instill the right culture so your business can leverage microservices to the fullest extent.

The faster time-to-market for the new functionality you gain is well worth the effort—whether you’re providing improved services to your customers or a more efficient way for employees to do their jobs.
Comparative Anatomy

Comparing Microservices to Monolithic Applications

A monolithic application is the traditional model for designing and developing software. The architecture consists of a self-contained unit in which all code exists in a single codebase, and the modules are interconnected. At implementation time, the entire codebase is deployed, and scaling is achieved by adding additional nodes.

As a monolithic application expands, its lack of flexibility and modularity become its Achilles’ heel, and several issues emerge that can create application development and performance issues:

» **Utilization of a single development stack**—when monolithic applications grow, the data that is stored can be very heterogeneous with varied processing requirements. Because of the tight coupling, the applications can only utilize a single technology stack. This creates issues with flexibility as developers are not free to choose the programming language or data store with the functionality that best suits the need of a given process.

» **Difficult to understand**—the tight coupling and the many interconnections between modules in a monolithic application can also result in a codebase and interdependencies between modules that are difficult to comprehend. New developers will need to have a good understanding of how the entire application fits together, which can make the on-boarding of new talent a daunting task.

» **Resiliency, fault tolerance and isolation**—the tight coupling and low cohesion of a monolith further create issues because changes to the application present a high-risk situation. Since a monolith is a single unit with many inter-dependencies, a bug in one part of the application can cause performance problems in another part, or even bring the entire application down. Due to the possible chain of faults that can occur, it’s also difficult to isolate the source of an issue. As a result, developers need to test each module, and it may be impossible to predict the outcome of even minor changes to the codebase. Recovering from failures is also not an easy task. Once a bug is isolated and corrected, the entire application must be rebuilt and redeployed, costing you valuable time and resources.
» **Long deployment timelines**—large monolithic applications can take very long to develop and deploy, oftentimes in the range of months or even years. The slow deployment velocity can result in several problems. These include a long timeline for customers or end users to receive new features; products hitting the market too late; and falling behind competitors that have embraced a nimbler development approach such as microservices.

These are just some of the issues that organizations encounter with monolithic architectures. There are many more relating to poor scalability and the declining quality of code. There’s also the lack of flexibility that arises from large development teams and traditional development methodologies (such as the waterfall model) that are commonplace.

**How Microservices Architectures Solve Monolithic Issues**

Many of the problems that develop as monolithic applications increase in size and scope can be addressed by transforming applications into a microservices infrastructure. Here’s how:

» **Development Stack**—because microservices are loosely-coupled and independently-deployed, developers are free to choose a different technology stack for each service. This provides an enormous amount of flexibility, enabling the best programming language and data store to be selected for the functionality required by the service.

» **On-Boarding of New Talent**—microservices are much smaller in size and scope than an entire monolithic application—each service can be managed by a small cross-functional team whose sole focus is on that service. And because microservices removes interdependencies, the services are independently deployable. That means there is no need for individual developers to understand the entire scope of the application. As a result, the on-boarding of new talent happens quickly. The individual services are much easier to understand.

» **Resiliency, Fault Tolerance and Isolation**—microservices do not share the same issues with monolithic applications related to resiliency, fault tolerance, and fault isolation. Because microservices possess loose coupling, it is not likely that a bug in one service, will cause an issue elsewhere in the application.
This also provides the benefit of making it simple to isolate faults to a single service. Once the bug is fixed, developers are free to redeploy just the service where the issue occurred.

» **Accelerated Deployments**—microservices can be deployed very quickly, and small dedicated software teams can focus on making changes to a service and deploy it without having to worry about the changes affecting other parts of the application. Companies benefit further when they apply Agile, DevOps, and CI/CD principles to their microservices initiative and automate functions such as testing, monitoring and deployment.

Microservices may not align with every company’s culture, processes and applications. Nor are they a magical fix-all that can be adopted without careful planning and consideration. In some cases, a monolithic application is still a viable choice and may be the best choice—such as when applications are small or when you are still proving the applications viability. Adopting a distributed architecture like microservices for a small application can result in an unnecessary increase in complexity and overhead.

So if your company is building an application or running a proof-of-concept, it’s important to evaluate which architectural design style will best meet your needs. When small in size, monolithic architectures are simple to develop and deploy, and they are easy to understand. They are also straightforward to scale, test, and debug.

It is not until monoliths grow large, that the case for microservices is made. You will need to make an assessment in order to answer the question of whether or not transforming a monolith application into a microservices architecture makes sense for your business.

**Comparing Microservices to SOA**

When the Service Oriented Architecture (SOA) approach to application development burst onto the scene in the 1990s, it was heralded as a revolutionary. SOA was an architectural design style intended to break monolithic applications into smaller modules oriented towards business objectives.

SOA relied on messaging protocols such as AMQP and SOAP to communicate between services, and the service modules ranged in size from small application
services to large enterprise services. However, SOA was unable to fulfill its mission of addressing various issues associated with monolithic architectures.

Looking back years later, it’s now apparent that in many senses, an SOA is still a monolith. While composed of several services, architectures are relatively course-grained and possess a high level of interdependencies. They are thus prone to many of the same issues as monolithic architectures. These interdependencies require the entire application to be rebuilt and redeployed each time a change is made, and this limits the speed at which companies can deploy new features.

In addition, communication in an SOA passes through an Enterprise Service Bus (ESB), which essentially promotes a monolithic structure. An SOA is also characterized by slow communication speeds and oftentimes becomes a single point-of-failure. These issues, along with the various implementation challenges presented above, all led to SOA’s inability to succeed.

**SOA vs. Microservices**

| SOA: Larger services and central communications routing (ESB) | Microservices: More, smaller services with independent communications |

**Planning Mistakes Cause SOA to Fail**

When microservices first emerged, developers were skeptical and hard-pressed to distinguish the difference between microservices and a service-oriented architecture. Unfortunately, this relationship did not do microservices any favors since SOA failed in its mission to address the issues of monolithic architectures.

SOA thus became a scapegoat, and implementers were quick to blame the SOA style for their failures. However, the source of these failures actually came down
Looking back years later, it’s now apparent that in many senses, an SOA is still a monolith.

to people neglecting to implement SOA correctly. Here’s a quick rundown of the common major mistakes:

» Not aligning services to business objectives.

» Using the wrong organizational culture.

» Failing to get business leaders on board.

» Lacking the necessary skills—including architects, DevOps and automation.

» Poor planning practices.

» Not fully understanding the complexity and scope of a distributed application.

These are just a few reasons why SOA initiatives failed. However, the fundamental concept of SOA and loosely-coupled applications composed of multiple business-function services was, and still is, sound. The concept has since been refined over the years, and these refinements culminated in the emergence of microservices.

**Microservices Offer Enhanced Form of SOA**

Basically, microservices is a refinement of the SOA approach. Therefore, the two styles share many similarities. For example, both are a collection of services that focus on business goals and are much smaller in scope than an entire monolithic application. Each architecture also requires organizations to undergo a cultural transformation that embraces decentralization and empowerment of cross-functional development teams.

Furthermore, developers are free to choose a different programming language for each service in both SOA and microservices. This allows developers to leverage the programming language that possesses the best features for the specific goals of each service.

However, given the shortcomings of SOA discussed above, the similarities to microservices should not sway your thinking when determining if microservices are a good fit for your application. That’s because microservices are the spawn of every failed and successful SOA initiative. Developers and architects have learned from their SOA initiatives and succeeded with fine-tuned microservices that have become what SOA was supposed to be.
While microservices and SOA were each designed to solve the same issues associated with monolithic architectures, there are many key differences:

» **Data Storage**—an SOA is similar to a monolithic application in that it typically shares a single RDMS database. As an application grows, its data characteristics and processing requirements can be heterogeneous. At this point, a one-size-fits-all data solution is no longer ideal. In contrast, a microservices architecture enables each service to utilize its own data store, allowing developers to choose the storage type that best meets the storage and processing requirements of the data utilized by the service.

» **Size and Scope**—microservices focuses on achieving one function and performing that function very well. An individual microservice thus tends to be much smaller in size and scope when compared to the larger grouping or modules of services in an SOA application. Developers will work on a single microservice, but the services are independently deployable. To illustrate, a developer that makes changes to Service A doesn’t need to understand how Service B works, because the changes they make to Service A will not affect Service B. This creates a major benefit when on-boarding new talent. Services are easy to understand and operate independently—developers don’t need to understand the scope of the entire application. In contrast, SOA services can be composed of multiple functions with many interdependencies, a single database, and an ESB. This requires developers to understand not only the service, but also the application interdependencies in full.

» **Communication**—the way that microservices and SOA services communicate is also very different. Microservices communicate through language-agnostic protocols, typically over the network. While this increases the number of remote calls, and in turn increases overhead, it results in faster communication with a high degree of fault resistance. On the other hand, SOA communicates through an ESB. While this generates lower overhead, it also slows down communication and presents itself as a single point-of-failure—with the potential to bring down all communication throughout the application.

» **Coupling and Cohesion**—the differences in coupling and cohesion relate to the size, scope and communication variances presented above. Microservices feature extremely low coupling and high cohesion. They achieve this by focusing on a single business function. The messaging and data storage function along with their operating systems are all encapsulated in an application container. This results in services that can be independently
built, deployed and tested. Another major benefit of low coupling and independent design is that a failure in one service is unlikely to cause a failure elsewhere in the system. When errors occur, it’s simple to locate and isolate the source of the failure. These features become extremely helpful at deployment time—buggy updates can easily be rolled back, resolved, and redeployed. In contrast, SOA services are much larger in scope and have more interdependencies. And communication and data storage are handled outside the services. This requires the entire application to be rebuilt and redeployed, leading to slow deployment times and cascading failures.

A microservices migration still carries many of the same cultural and planning requirements as an SOA initiative. However, as shown above, microservices have come a long way in mitigating the issues that SOA experienced in its infancy.

**Snapshot View: Microservices vs. Monolithic and Service-Oriented Architectures**

<table>
<thead>
<tr>
<th></th>
<th>Microservices</th>
<th>Monolithic</th>
<th>Service Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Size</strong></td>
<td>Very small, function-oriented, independent services</td>
<td>Highly-interdependent services</td>
<td>Smaller, interdependent services</td>
</tr>
<tr>
<td><strong>Granularity</strong></td>
<td>Loosely coupled with high cohesion</td>
<td>Single, tightly-coupled unit with low cohesion</td>
<td>Moderately coupled with moderate cohesion</td>
</tr>
<tr>
<td><strong>Ease of Deployment</strong></td>
<td>Can independently build and deploy each service</td>
<td>Must build and redeploy entire application</td>
<td>Requires modifying the monolith</td>
</tr>
<tr>
<td><strong>Remote Call Overhead</strong></td>
<td>High communication overhead; many remote calls</td>
<td>Low/no communication overhead</td>
<td>Low communication overhead</td>
</tr>
<tr>
<td><strong>Speed of Deployment</strong></td>
<td>Rapid and continuous deployment</td>
<td>Very slow deployment speeds</td>
<td>Slow deployment speeds</td>
</tr>
<tr>
<td><strong>Persistence</strong></td>
<td>Each service is free to choose its own data storage</td>
<td>All services share data storage</td>
<td>All services share data storage</td>
</tr>
<tr>
<td><strong>Ease of On-Boarding</strong></td>
<td>Easy to onboard new developers</td>
<td>Difficult to onboard new developers</td>
<td>Semi-difficult to onboard new developers</td>
</tr>
<tr>
<td><strong>Polyglot Programming</strong></td>
<td>Can utilize various technology stacks per service</td>
<td>Utilizes a single technology stack</td>
<td>Can utilize different technology stacks per service</td>
</tr>
<tr>
<td><strong>Communication Method</strong></td>
<td>Communicates with lightweight protocols like REST</td>
<td>Many methods exist</td>
<td>Communicates through enterprise service bus</td>
</tr>
<tr>
<td><strong>Scalability</strong></td>
<td>Extremely scalable through use of containers</td>
<td>Can be very challenging to scale as applications become larger</td>
<td>Can be challenging to scale</td>
</tr>
</tbody>
</table>
Should Your Company Utilize a Microservices Architecture?

Now that you have an understanding of how microservices compares to monolithic and SOA applications, the next step is to determine if your company should utilize a microservices architecture. By doing so, you overcome the constraints of monolithic applications and avoid the pitfalls of SOA applications.

However, before launching your microservices initiative, there are several key factors to consider.

**Business Needs Should Drive the Initiative**

Microservices initiatives should not be undertaken just because your competitors are doing it, or because other businesses have had success with this application architecture approach. Look deep inside to identify the problems you are experiencing and ask the right questions to determine if microservices is the best solution:

» Does making small application changes require you to perform expensive and timely redeployments of the entire software suite?

» Could your internal and external users benefit from features being deployed in a timelier manner?

» Will faster deployments help you reach market faster and make you more competitive?

» Will decoupling your current application strengthen your system and result in increased reliability?

» Does your monolithic application host a large amount of unneeded functionality?

» Do you require your application to effectively utilize the scalability and flexibility of the cloud?

» Do certain services need to scale rapidly on-demand?

» Could you benefit from leveraging multiple programming languages for specific parts of your application?

» Are you dealing with large amounts of data that have heterogeneous characteristics—where individual services could benefit from utilizing their own data store?
If you answer “Yes” to the majority of these questions, chances are strong that you can benefit from transforming your application to a microservices infrastructure. It could also be that your monolithic, all-in-one software environment no longer fits the way your organization works. Over time, such legacy systems can become difficult and expensive to manage and may provide functionality that hardly anybody uses.

**Determine If Your Application Is a Good Fit**

Microservices are not the right fit for every application. For new applications, it is often best to start by building a monolithic application, and then decouple into microservices after the application concept has been proven and has grown substantially in size.

The reason for this is that monolithic applications are much simpler in terms of communication, security, and monitoring when compared to microservices. Monolithic applications also tend to come at a lower cost in terms of the overhead and the latency related to the increase in remote calls made by microservices.

For these reasons, start-ups should weigh the advantages and disadvantages of beginning with a microservices architecture...

**Advantages of a Microservices Architecture**—when a microservices architecture is utilized for the right reasons and implemented properly, businesses experience a massive reduction in unused functionality and gain enormous benefits compared to applications that utilize a monolithic architecture:

- Improved modularity results in applications that are easier to understand, develop and test.
- Streamlined new developer on-boarding—no need to understand the functionality of every service.
- Software teams can better leverage Agile, DevOps and CI/CD principles to automate testing and deployment.
- The ability to update a single service and roll back buggy updates reduces risk and improves the speed of deployment.
Each service can be written in a different programming language, allowing developers to choose a language that offers the best features for each service.

A small cross-functional team can manage each service, enabling team autonomy, distributed leadership and decentralization.

The organization is better able to leverage the scalability and flexibility of the cloud.

Resolving issues is much simpler as they can be isolated to a single service.

High degree of fault isolation: a change or failure in one service will not impact other services.

Each microservice can also leverage its own database and storage techniques, which is key since services have different needs when it comes to data storage; as applications become larger and the volume of data grows, the data sets can have varying characteristics and processing requirements. Software teams can also choose the best data storage implementation to address the specific need of the service in order to achieve the end goal.

**Disadvantages of a Microservices Architecture**—while there are several advantages to decoupling a monolithic application into microservices, as shown above, there are also some disadvantages to be aware of before making the switch:

- While each service on its own is easier to understand, develop and deploy, the bigger picture becomes more complicated to manage.

- More communication is required between services; this can result in higher overhead costs associated with network congestion and latency.

- Network security becomes increasingly complex; with a growing number of microservices, communication between each service needs to be secured.

- A strong understanding of DevOps is required to handle communications, automated deployments, testing, and monitoring.

Most of these challenges arise from the complexities introduced by adding independent services. But with the right tools, processes, preparation and expertise, all of the issues can be overcome.
A Strategic Approach to Decoupling Monolithic Applications

Once you decide to decouple a monolithic application, you then want to follow a process that helps you complete the decoupling as quickly as possible while making sure the application stays up and running for customers and/or internal end users. It’s best to prioritize the decoupling of the application’s modules based on three factors:

» Business value—how much does each module impact day-to-day operations?

» How frequently does each module change?

» What is the workload on each module?

In a nutshell, the modules with the greatest business value that change frequently and which are usually under a heavy workload should be decoupled first. It’s also best that these decisions be made by a combination of the company’s executives and the software team to make sure that both the business and the technical factors are considered.

By prioritizing modules for decoupling, the software development team can work on the services that mean the most to the business. You may discover, for example, that an application bogs down when managers run a specific report to generate KPIs that help run the company. The development team can then prioritize decoupling the module that contains that particular service.

Remember to take the perspective of decoupling capabilities—as opposed to decoupling code. You can rewrite the capabilities by extracting only the code that is actually in use. This will allow you to get the necessary features that need to improve up-and-running faster.
Analyze Current Environment to Identify Application Pain Points

The first step to decoupling a monolithic application is to review the current environment, including which technologies are in use and the libraries and frameworks that the technologies tap into. Then run basic monitoring tools to determine the current status of the application:

» How many queries and requests are made to specific compute resources?
» How long do queries take?
» What are the capacity and utilization of the CPUs and the memory?

If the application problems you are experiencing are caused by insufficient CPUs or memory, you may want to start by moving your monolithic application to a server with additional compute resources. If, after adding those resources, the application problems persist, then conduct in-depth log analysis and error tracing.

This allows you to collect additional metrics that provide details on the exceptions received from the application and to identify application bugs. By determining the application’s pain points, you will have a better idea of where to decouple first.

You may discover that you can re-use some of the existing code base. But more than likely, you will need to extract functionality from some of the services and move them to a new code base. The good news is that a microservices architecture gives you the flexibility to build each service using the programming code that works best.

If the coupling is really tight, you may not be able to decouple the entire monolith. In this case, you can decouple some of the services and leave the rest within the monolithic architecture. An application can use a combination of a monolithic and a microservices architecture by utilizing an API gateway to direct end users to the decoupled portion or the monolith depending on the services they request. You can also insert an API layer that allows the monolith and the decoupled services to exchange information.
Decoupling Best Practices

Next, set up the application containers in which you will build the microservices. It’s also helpful to utilize a container orchestration tool to automate the process of container creation. These tools will help you greatly in breaking up the monolithic application into small services and in managing the microservices post-production.

Begin the actual decoupling by taking a macro, then a micro, approach. First, break the monolithic application into modules based on how the different functions of the applications are grouped. Then divide the modules into subsystems that will consist of multiple microservices.

As you proceed through the decoupling process, here are some best practices to apply:

» Determine where to start and how it’s tangled by utilizing dependency trackers.

» Identify hidden coupling with code forensics.

» Uncover application failures and development practices that do not adhere to common standards by using static analysis.

» Determine if a service needs to cover any type of input validation or sanitization by implementing OWASP validators.

» Model subsystems and define the boundaries in the monolith for bounded context by leveraging domain-driven design.
» Map each service into separate domains to see where information flows and the dependencies among services.

» Strip the tables from the main database that are needed for each service (one-by-one), and turn each table into a separate database for its designated service; this improves query performance and makes it easier to diagnose query failures.

» Expose each of the new database tables as a service so that other services can access it via HTTP, messaging or queues.

» Decouple services vertically and release data early so each service can be managed in a decentralized manner.

Each microservice will need to ask for information from other microservices because they don’t share the same database and resources. So be sure to design a communication plan among the multiple microservices to get information from one service to another. You can set this up using a message queueing platform or event-driven design.

As you apply these best practices, it’s important to minimize the dependency that each microservice has on the part of the application that is still within the monolithic architecture. The lower the dependency, the faster you can roll out future updates to the service.

After testing the data flows and fault responses for each microservice, fix any bugs and check the logs to validate whether every resource/endpoint is exposed to each API. You’re now ready to deploy!

Developing a Plan for Microservices

Transitioning an application to a microservices architecture can be challenging—it’s a big step for most organizations, and the path to success is paved with risk and uncertainty. But by carefully evaluating the key considerations—including the decision as to whether microservices is the right fit—you can ensure your organization takes the right path at the right time. You will get your microservices initiative going in the right direction by making sound decisions and taking one well-planned step at a time—starting with the needed infrastructure and processes.
To begin the transition away from monolithic legacy software, you have the flexibility to implement microservices to a portion of an application to gain just the functionality you need, and then evolve the application further later on. That means conceptually and technically decoupling large application suites into distinct services, which can be created, tested, deployed and provisioned to users—one-at-a-time.

Once you identify your business needs and ensure the microservices model matches your culture and processes, then carefully plan your approach. Here’s a quick rundown of some of the key factors to consider and decisions to make:

» Decide which microservices patterns to use and what the best-fit choices are. If you decide on the pattern of one database per microservice—where changes in the database don’t impact other services—you also need to determine whether it’s to your advantage to use a composer pattern, which could be per domain, per product, or even per team.

» You don’t want to end up with too many composers and needless complications, but you also should not have fewer than you need, which would saddle you with a large code set that is difficult to maintain.

» If you need to maintain transactions across microservices, another tool decision needs to be made. Then, there is also the question of whether to use a discovery service, and if so, what kind: for the client side, the server side, or the registry service.

» Also determine how the clients of the microservices application will access the individual services. This can often be solved by using an API gateway (like Netflix did, in a well-known example) to provide a single entry point to the microservices for the front-end. You can choose among a variety of ways to set up such a gateway.

» The microservices architecture needs a solid recovery strategy, or else failure at any of the likely multiple points could be ruinous to your effort, especially if the application cascades.

» Consider the possible consequences of a microservice failing or taking too long to respond before you apply a remedy. Hystrix by Netflix is one possible solution that can help you ensure latency and fault tolerance, and it can make your distributed systems more resilient when failure is highly probable.
Another key factor to consider is whether to rely on multiple instances of the same microservice to enable quick failure-recovery. You can do this by using a client load-balancing tool. If the database crashes, a good remedy might be a cache from where you can retrieve data until the database is up once again. Of course, you can also combine the two approaches—multiple instances of one microservice together with a recovery tool—and benefit from better scalability and availability.

The decisions you make in relation to the factors discussed above will have a profound impact on the results you experience. Having a solid understanding of everything involved in a microservices migration and a strategic plan to guide you will be your best bet to achieving success.

What should failures look like?

Failover Caching
Hitting Microservices Out of the Park Requires Right Organization and Right Methodologies

“Hitting microservices out of the park” is not just about the coding and architecture effort that’s required to transform a monolithic application. Success also relies on a reorg—at least virtually—in the make-up of your IT teams.

Many businesses that buy into the benefits of microservices make the mistake of using their current IT-team structure and focusing only on building the microservices and writing new code. But they then run into issues when trying to move the microservices into production. They do not realize they also need to develop and automate new processes for data integration and interfaces, test that the new microservices are working properly and have the necessary security posture, and make sure they can be deployed rapidly.

Other keys to success include proficiency in Agile development, Lean principles, and CI/CD (continuous integration and continuous deployment). IT teams that already utilize these principles within a DevOps environment are well-suited for migrating monolithic applications to a microservices architecture.

In addition, new collaboration tools are needed to enable team members with vastly different job functions to communicate effectively and to automate the software development lifecycle. Only then is it possible to quickly roll out new functionality to end users—one of the key business value offerings of microservices.

Start with Business Requirements and Reorganize Your Teams

To successfully complete the microservices journey within the expected time frame set by the business, start with the requirements:

*Which application services need to be provisioned to end users from a microservices architecture?*

The answer to this question is usually dictated by how often services change, the end-user workload on the services, and the criticality of the services in the day-to-day functioning of the business. When the business needs are the driver, there’s
momentum behind the effort to transform to microservices across the entire organization.

Another key factor is your organizational structure. Traditionally, IT teams are broken into separate silos of sub-teams, responsible for individual functions:

» Development
» Application Interfaces
» Deployment

» Data Integration
» Testing
» Monitoring

» Business Logic
» Security
» End-User Support

Under historical organization models, each team does its own thing and then hands off the application to another team. When problems occur, this arrangement can lead to a lot of finger pointing as everyone tries to determine who is responsible and prove they are not the culprits.

Conversely, microservices utilizes a new organizational paradigm—where cross-functional teams consisting of resources from each of these functional areas come together and are matched to each microservice that the monolith has been divided into. It’s also important to add someone to the team from the business side—a manager and/or a process analyst who can work with the team to prioritize business needs.

Depending on the availability of internal resources and the number of microservices, you may decide to hire additional people. But you can also consider establishing virtual teams where some resources are part of multiple teams.

The small size and independence of microservices allow you to eliminate dependencies on the knowledge and influence of key individuals. You can also avoid the bottlenecks that result when key people leave or change roles—by
bringing enablement and intelligence to teams of people who collaborate in a goal-driven, shared-values process.

Evaluate your team-formulation strategy carefully. It’s not just a matter of development; you also want to consider how you will test, add security controls, and move into production. After you identify the roles you require and if you need additional resources, you will also want to train the staff on the tools to use in order to function within a DevOps environment.

**Entire Team Assumes Ownership and Responsibility**

The cross-functional structure of microservices teams emphasizes ownership and responsibility of each microservice—across an entire team that spans all IT roles. With teams operating autonomously and utilizing distributed leadership and decentralized decision-making, they can function more nimbly.

For example, if there’s a problem in production, it’s not the Ops team that tries to diagnose the issue, nor the Developer team determining if it made a coding mistake. Instead, the entire team assigned to that microservice will collaborate at once to solve the issue.

In the most successful microservices practices, team members are fully-dedicated to specific subsystems of related microservices. They are not contributing on multiple teams at the same time, which has often been tried and almost always proves to be very inefficient.

Several companies have also created full-stack domain teams instead of disparate front-end and back-end teams. Doing so requires some reorganization of resources but results in greater efficiencies as the software team delivers functionality and minimizes complicating dependencies.

Culturally, it helps to encourage proactive ownership and accountability on the software team. In the Agile spirit of continuous improvement, it’s also best to provide meaningful metrics along with constructive, regular performance reviews for individuals and teams.

Facilitating efficient communications among team contributors is important as well. Succeeding in these efforts may mean deploying chat or other tools that allow people to connect and communicate at any time from any location.
Tools Play Key Role

After creating teams and assigning them to part(s) of a monolithic application, create the processes to migrate to microservices incrementally—in iterative sprints, one service at a time—while also making sure each iteration delivers value. Start by picking a single business function that you can build as a small microservice, and then plug it back into the monolith via an API gateway. Quickly experiencing initial success will allow all teams to learn the best way of collaborating.

To facilitate collaboration, the teams need to be supported by tools that automate the hand-off of each microservice and streamline communication so that each person on the team can work self-sufficiently. The tools you choose may be driven by the cloud platform you are using, but there are also many tools that work across cloud platforms.

As you build your toolset, be sure to involve resources from all the IT functions and the business. This creates a holistic approach driven by business requirements as to choosing tools that enable teams to deliver the necessary functionality, speed of deployment, and the processes to enable continuous improvements. Tools that facilitate clear and smooth communication on the information that the team needs to share are also a must.

Microservices and Monolith Applications Can Still Play Well Together

Without the right organization, you may end up with a distributed monolith where the microservices can’t function fully as independent services, and supporting them becomes complex. To fully realize the benefits that microservices can offer—and to reduce the challenges associated with their increased complexity—the automation of testing, monitoring, and deployment is an absolute requirement. Success may also demand staggering resources to adapt monolithic applications to enable the rapid innovation and digital processes that your company wants to realize.

The processes you set up should also continuously evolve to support the high speed of change in today's digital world and to adapt quickly to new business conditions. Review your processes at least twice each year to determine if the current approach is working sufficiently and if any adjustments are needed to the
architecture or the processes. The changes may be driven by a reorganization of the company, new business requirements that drive additional application functionality, or if the company expands or acquires a new company.

If part of the monolith is working fine for how the business operates, you don’t need to convert the entire application—just the parts that are creating operational issues. In some cases, the company may decide to leave part of an application in monolithic mode and migrate other parts to microservices.

In these cases, the IT teams supporting the monolith will function differently than the team supporting the microservices, so it’s important that both teams understand how each of them manages the application lifecycle using different methodologies. For functions within the monolith portion that need to exchange data with the microservices, the members from the two teams that interact to handle the integration should set up a clear line of communication.

**Cultural/Organizational Checklist for Transforming to Microservices**

The checklist below will help guide in you considering the cultural and organizational changes that need to be made to adopt the microservices application architecture approach:

» Based on the functional requirements of the business, determine the microservices that the monolithic application will be broken into.

» Create teams responsible for each microservice; members should cover every function of the software development lifecycle—business logic, development, application interfaces, data integration, security, testing, deployment, monitoring, and support.

» Assign an appropriate business unit manager and/or analyst to every team based on which business unit each microservice pertains to.

» If not already in place, create the necessary processes and provide tools to enable each team to adopt Agile, CI/CD, and Lean principles and methodologies.

» Establish a DevOps environment in which each team can communicate efficiently and hand-off tasks to each other automatically.
Making the change to microservices will require your IT teams to adopt a new culture that incorporates Agile, CI/CD and DevOps principles.

Each time a team is tasked with creating or updating a microservice, clear goals should be established as to the new functionality that’s required, the roles of each team member, and the timeline of when the business must have the new functionality. As the team progresses through a microservice project, emphasize the importance of accountability, team autonomy, distributed leadership and decentralized decision-making.

The application design and technologies that are in play—and the processes to be followed to bring the change from design to production—should be transparent for the entire team, and progress updates should be communicated frequently. With the entire team assuming accountability for success, the team will be able to work together more effectively.

**Transforming Great Ideas Into Customer Value**

If you transact business with customers in the cloud or rely on cloud applications to drive business processes, application agility is key. You need to continually roll out new functionality that keeps your external and internal users fully engaged. To improve your application agility, microservices is the answer you’re looking for.

In contrast to monolithic applications, the microservices software development technique allows you to break large applications into loosely-coupled, independent modules that communicate with each other through lightweight-language agnostic protocols like RESTful APIs. As an evolution that goes beyond the SOA approach, the microservices architectural style has become widely-used among developers for web-based applications—thanks to its ability to manage APIs and execute highly-defined, discrete tasks.

As you consider the move to microservices, it’s important to first assess if each of your applications will benefit from the new architecture. In some cases, you may decide to maintain part of an application in monolithic mode and part in a microservices mode.

Making the change to microservices will require your IT teams to adopt a new culture that incorporates Agile, CI/CD and DevOps principles. IT will also need to reorganize into teams with resources who cover the entire software development lifecycle and are assigned to specific microservices. It will also be critical to adopt
microservices tools and best practices that enable each team to collaborate efficiently and keep each microservice running at peak performance.

The reward is well worth the effort. Microservices do not require IT teams to rewrite and deploy an entire application when they add new features. New functionality can be rolled out much faster. And since microservices applications produce a smaller codebase, ongoing software maintenance is much easier and takes less time.

The ultimate payoffs: Businesses that successfully adopt microservices significantly accelerate the time it takes to transform their great ideas into customer value. That value in turn generates more revenue and sustains long-term customer relationships.

For more information on transforming your monolithic applications into a microservices architecture contact Tiempo to learn more.

About Tiempo

Tiempo is widely recognized as one of the leading software engineering companies in the US. Using a combination of nearshore engineering resources, high-performance teams and relentless focus on client outcomes, Tiempo designs, builds and deploys software that makes lives better.

Tiempo is headquartered in Tempe, Arizona, with four world-class software development facilities in Mexico. Tiempo has been recognized annually by Inc. Magazine as one of the Fastest-Growing Private Companies in America.