EVOLUTION OF MICROSERVICES
SEVEN FEATURES TO LOOK FOR IN A MICROSERVICES STACK

TABLE OF CONTENTS

Trying to determine the direction and usefulness of microservices? Then it helps to know where this architectural style came from, how it relates to other technologies and the problems it can solve.

Rob Tiberio does just that in this white paper. As middleware technology expert and chief architect at Software AG, Rob demystifies microservices and explains the evolution from a Service-Oriented Architecture (SOA). Read this paper to learn recommended next steps for practical use of microservices in enabling digital initiatives and what to look for in a microservices stack. See how using an Enterprise Service Bus (ESB) can make a project built in microservices a resounding success.

Origin of microservices

The culmination of several trends

The basic concept of microservices has been around for quite some time; it's actually the culmination of several architectural styles.

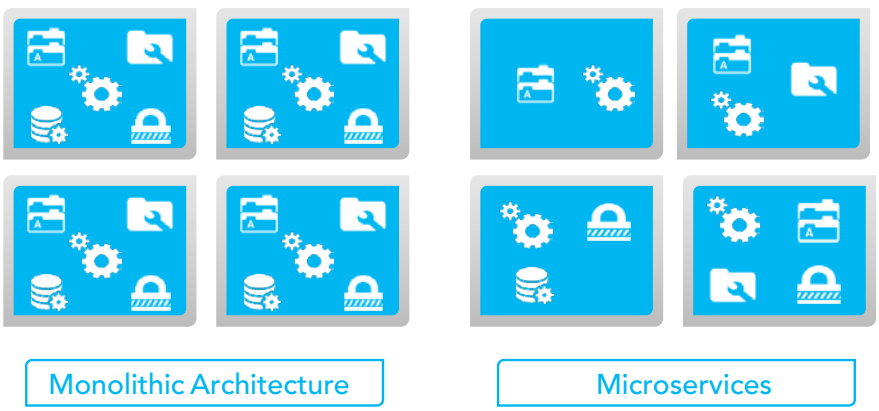
Several years ago, there were "shared nothing" architectures, also known simply SN architectures. The concept behind this architectural style was to build a large application with parts designed to function without any dependencies. Among the advantages was the ability to scale an instance of an application up and out independently. By reducing dependencies, the SN architecture also had the benefit of faster time-to-market. Removing dependencies in the application's design also

meant fewer dependencies between developers. This made project management easier—very valuable when delivering large-scale projects.

The notion of a microservice, at its very core, is that each piece of a complex system is discrete, and its business functionality can be broken down into much smaller pieces. Those smaller pieces can be engineered—they can be tested and deployed—independently of trying to cross-coordinate members of a large engineering team.

Conceptually, a microservice is a portion of a complex application that is a discrete amount of self-contained business functionality, meaning the run-time server, the configuration and the business logic are deployed as a single unit. These units can be developed, tested, deployed, upgraded, retired and so on without having to deploy the entire application at the same time. Additionally, microservices can scale up or out individually as needed.

There are additional benefits that microservices offer. The independence realized by breaking down business functionality into discrete deployment units lends itself very well to the idea of "use the right tool for the right job" and provides better agility and more flexibility. Developers can choose any programming language, storage technology and framework that solve the problem because each part of the overall system no longer must conform to standards imposed in a "monolithic architecture."



Monolithic architecture emerges

When enterprise IT was emerging from the mainframe-dominance era, developers intentionally built systems in a monolithic way. What gave rise to this approach 15 years ago was the great divide in the computing industry between Microsoft® .NET and Java® environments. There were certain kinds of applications and development activities that best suited the Microsoft .NET world, and Java was taking a stronghold on the back-end, high-end enterprise server arena. The programming models and practices from the past basically made it convenient to develop applications using standardized tooling offered by vendors in a way it was hard to avoid the monolithic situation. Also, IT operations had their own challenges and tended to lean towards standardizing on a small set of technologies they could fund and operate efficiently. It was not uncommon to label an enterprise as an Oracle®, IBM® or Microsoft® shop because of this effect. Developers also didn't have as many choices as they do today.

Depending on the platform they chose, developers would put all the pieces of an application together in a single deployment unit. That discrete unit became known as a monolithic application. All business functionality was contained in one particular

packaging format, and it complied with the rules of the platform's deployment model. This was the famous three-tiered architecture: presentation tier, server tier and storage tier. All of those tiers were implemented in a single implementation. Developers had to write all that functionality, put it inside something like a Java® Archive (JAR) file, a WAR file or a .NET assembly, and then deploy it as a single unit.

The problem with this monolithic approach? If any one piece of functionality needed to change, the developer had to make the change, enhancement or fix, and then redeploy the entire contents as a whole.

Microservices solved that problem by separating all those pieces of functionality to run independently. For example, instead of one deployment unit, with microservices there may be 30 deployment units, and each piece can run and be deployed independently. This makes it possible to change any piece without redeploying the whole application and leads to a basic rule of discrete functionality within microservices: Avoid dependencies between microservices.

Microservices are not necessarily just small services; they, in fact, can be rather large depending on the functionality they provide. Even so, they are discrete and self-contained.

Moving beyond monolithic

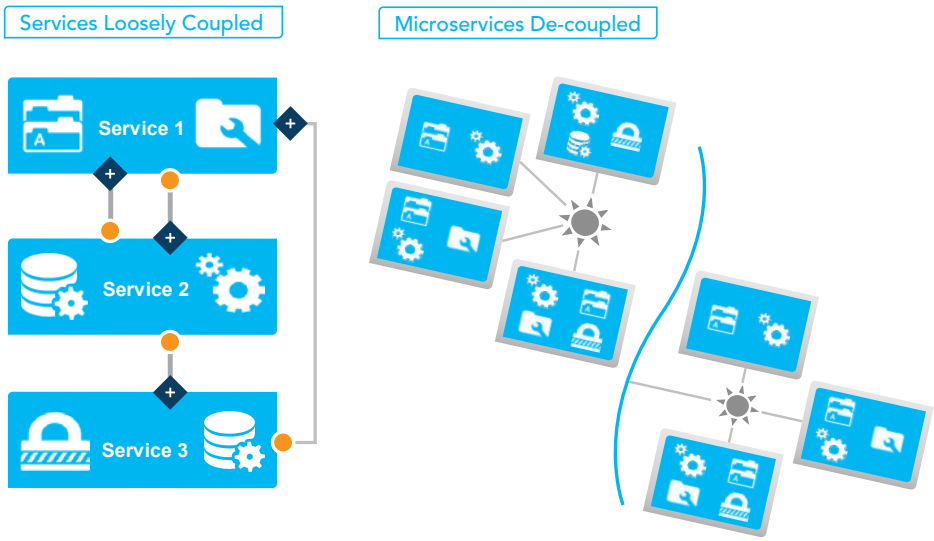
There's certainly nothing wrong with the standard three-tiered database style of architecture. As a matter of fact, over the years, the three-tier architecture evolved into a N-Tier architecture with more than three but still discrete tiers. The N-Tier architectural style ran into trouble because of the considerable pressure put on that single instance of the application running in a production environment. All clients, whether they were physical humans or systems, communicated with that one application server.

One of the things that N-Tier or three-tier application designers have to deal with today is how to scale the tiers of an application. At some point, there will be too many user sessions trying to hit the application server, and an attempt will have to be made to scale it—either vertically or horizontally. The problem comes in knowing when that limit will be reached. And because all the functionality is not separated, as in a microservices architecture, what tends to happen is one capability in the application might consume too many resources in that application server. This event causes a condition of starvation for all the other functionality residing within the server.

What has happened over time—and it is a fundamental change—is the substantial increase in the size of the applications organizations are attempting to deploy. Look at Netflix®, Facebook and LinkedIn®—these are massive applications. Even inside the enterprise, it is now not uncommon for a single massive application to be deployed for enterprise-wide or customer-facing use. There are hundreds of thousands of users today, as opposed to years ago when applications were mostly built in stovepipes and silos for departmental use. As a result, the complexity of trying to scale an N-Tier architecture built using a monolithic approach now outweighs the benefits of using that design. That design is very difficult to fan out. In contrast, with a microservices approach, fanning out the processing and load across multiple instances is much easier to manage operationally and lends itself to making use of newer elastic scaling capabilities.

This is especially true with the persistence layer. In a monolithic architecture, there might be a central location where all the data for every functional part of the business application resides in only a single database instance. Eventually the system will encounter a bottleneck where the database cannot scale large enough any longer, and an attempt must be made to fan out the database tier itself.

With a microservices-based architecture, things are different. Because it's oriented toward separating storage and functionality discretely, there can be many more instances of how things get stored, and many more copies of that implementation running in an environment. Because of this kind of distributed architecture, some of the largest applications today are able to scale to sizes unheard of before. Applications today are built not for thousands of people but for millions of connected users, and that's much of what is driving the move away from a monolithic approach and towards microservices.



How microservices matured

The difference between microservices and SOA

SOA and microservices share a common pattern. Both are based on the concept of a service, and both ideally separate the interface exposed to a caller from the implementation. However, microservices applies a practice that is neither defined nor enforced by SOA. In essence, both represent the construct of an Application Programming Interface (API) within the context of a larger system.

The main difference is that a microservice employs a practice that attempts to eliminate any dependencies on other microservices. SOA does not make this practice explicit as a requirement; it's left as an implementation detail.

Suppose a developer builds a Web service that does a credit check based on a customer's social security number and some other basic information. In order for that particular service to function in an SOA-style implementation, it may invoke a chain of dependent services. With the microservices style, you would try and avoid forming those dependencies.

One of the things a developer should do when creating a microservices architecture is to set some finite rules. One rule should be that each particular part of the system, the discrete functionality, is autonomous, meaning that it functions all by itself. It has its own persistence, its own interface and its own wire level protocols that it supports. The only way that one microservice should ever speak to another microservice is through a common network protocol, such as REST. SOA, by contrast, does not go so far as to dictate the implementation approach. In an SOA, it's left up to the system designer to decide whether those service chains should be called internally, in process or through a network protocol. Here again, microservices is a more specific approach to building an SOA.

Is microservices, as some have argued, simply a way of "finally getting SOA right?" No, microservices is a subset of SOA and applies some additional rules. It is the adherence to those rules that gives microservices its own unique "mission" in application development and enterprise architecture.

DevOps plus microservices equals disruption

Adopting a microservices style of architecture does have its challenges. Here's why. The traditional enterprise has created a separation between development and operations and for very good reasons. When developers are allowed to use "the right tool for the right job," the cost or complexity of operating a system composed of many technologies often isn't taken into consideration. Over time, the IT operations organization has standardized on the technologies it owns. This ensures that budgeting for capacity, daily operational tasks, planned upgrades, security and so on can be managed in the most efficient manner using the least amount of human capital. Developers, for example, need to consider which database, messaging technology, Web servers, app servers, and even operating systems and hardware are used in their production systems. Developers unfortunately have the daunting task of understanding business requirements and, at the same time, they are constrained by what technologies the operations teams agree they can support due to budgetary concerns.

For microservices to become truly an effective style of architecture, it also necessitates a disruption within the overall organization such that the developers who built the system play a much larger role in operating it as well. Interestingly this notion of developers operating the production system is not entirely new. If we think back to a few decades ago, this was typically the case but for a different reason. Years ago, the complexity of running a data center was something that only highly trained engineering scientists could actually do, and they were typically the same staff that built the applications.

The introduction of microservices in IT impacts deeply ingrained culture and breaks the barriers between Dev and Ops. This signifies a movement away from the traditional IT development and IT operations into DevOps. So, before any company goes too deep into microservices, it must step back and ask whether it's also ready to break the traditional barrier between IT operations and IT development.

A tipping point for DevOps and microservices

Once an organization is willing to move into DevOps along with microservices, it will find quickly that, as more microservice-based applications are built and deployed, the cost of owning them and operating them may outweigh the benefits of the speed of bringing them to market. The reason is that traditional IT might deal with hundreds of applications. Even with microservices, there is a tipping point that requires standardization.

Let's consider an example where a DevOps team is building an extremely large application for customer engagement that's exposed to all customers. This one application offers considerable business functionality and will require the deployment of hundreds of microservices using different technologies that are all interacting perhaps in a point-to-point manner.

The DevOps team, since it wrote the code, knows how to manage that environment; it knows exactly what each part of the system is doing. If something is not working correctly—say, an upgrade is needed or a fix applied—the team knows where to find everything. For this single large application, everything is fine. But what happens when the same DevOps team must deploy a second large application? Perhaps the team uses some of the same microservices, perhaps not. Maybe it creates additional microservices. Now what about the third application?

Eventually, over time, that DevOps team might realize that traditional IT operations created its list of "thou must …" deployment standards for a simple reason: Once an organization gets into the eleventh or twelfth application, it becomes difficult to keep track of where everything is. At that point, the DevOps team will likely accept that some things must be standardized. For example, they might say, "We realize we have 2,000 copies of a particular log4j jar file running around in the production environment, and we just got a phone call from our security auditor, who told us that there's a new vulnerability in log4j and that we must upgrade all the log4j jars deployed in our production environments. Now we must go find 2,000+ log4j files across all containers and we don't know where they are!" This makes a great case for why sharing common things is good and why standardization helps efficiency.

This is the tipping point that might be reached with DevOps and microservices: Building a monolithic application, in which everything is shared and there's a common library, has a significant cost in coordinating its development; however, it does make it simpler to operate and maintain those production environments. On the other hand, building a microservices-based application, where nothing is shared and each microservice has its own copy of a library, database and every other component, leads to many challenges in a production environment—but it provides great agility in development. The pendulum is likely to swing back and forth. In the end, it will likely be a good idea to share some things and a bad idea to share others. And similarly, the developers who are building the microservices today are going to start looking for pre-built frameworks and technologies that are best in class at certain functional areas, such as an ESB, so they don't have to "roll everything by hand."

Microservices as a usage pattern

It is commonly assumed the value of a microservices approach is that it represents a new architectural pattern and, therefore, leads to a superior application when compared with the monolithic approach. This is not necessarily true. The superiority of microservices is in its usage pattern, or the style in which it allows programmers to work. Programmers prefer to work by doing something small, testing it, throwing it into a production environment, receiving feedback from hundreds of people, and making improvements. That is the cycle they favor, and microservices allows them to work that way. The same is true for SOA and Event-Driven Architecture (EDA) styles. In addition, programmers prefer to work independently, where they don't have as many dependencies on other team members. They are much more agile that way. They can build things faster, incorporating immediate feedback into work that might be only partially completed. This usage pattern is the greatest benefit of microservices—not necessarily what is achieved but how the microservices are used. Decoupling large applications into smaller parts is a good idea; there are several styles of architecture that, in principle, offer the same advantage.

Imagine a developer deploys a microservice and someone tries to use it but realizes he can't because of some limitation. That person reaches out to the developer of the microservice and says, "I like what you built. However, I would like to use it in a slightly different way." The nice thing about microservices is that this immediate understanding of a limitation can turn into an immediate enhancement to alleviative that constraint. It is more of an Agile approach, as opposed to, "Hey, we're going to embark on long project; we're going to design everything on paper, we're going to code it up and we sure hope it works and if you want it enhanced we will repeat this process."

Microservices and applications

An application's construction has radically changed over the years as applications themselves have changed. First came the green-screen application, then the desktop application, and soon after the client/server application. Today, an application is a composition of orchestrated modular parts that are highly distributed throughout the enterprise architecture. Parts of an application exist in the SOA layer, the business rules server, the messaging provider, content management, storage tiers, caches in-memory and so on. The same application may have several different form factors. It may work on a tablet or smartphone as equally as it does on a Web browser. Applications might be process-enabled and drive business processes or interact with end-to-end processes. Trying to stuff all of this into one deployment unit, into a monolith, simply becomes "mission impossible" and does not lend itself for changing the application at the speed Digital Enterprises need today. The microservices architecture, therefore, is a welcome idea to help with the composition of a modern-day application.

Microservices in the cloud versus on-premises

Is there a difference in how a microservices-based solution is deployed in the cloud versus on-premises? The short answer is no. In fact, the cloud offers a distinct advantage. One of the things about microservices architecture is that very little capacity planning is needed. It is possible to start small, relocate and replicate as needed. The benefit of running as a cloud deployment is the speed at which you can provision additional hardware, which essentially matches the speed you can provision microservices.

Therefore, expect to see modern applications that are built in microservices deployed more frequently in the cloud than on-premises.

Docker® in the cloud benefits microservices

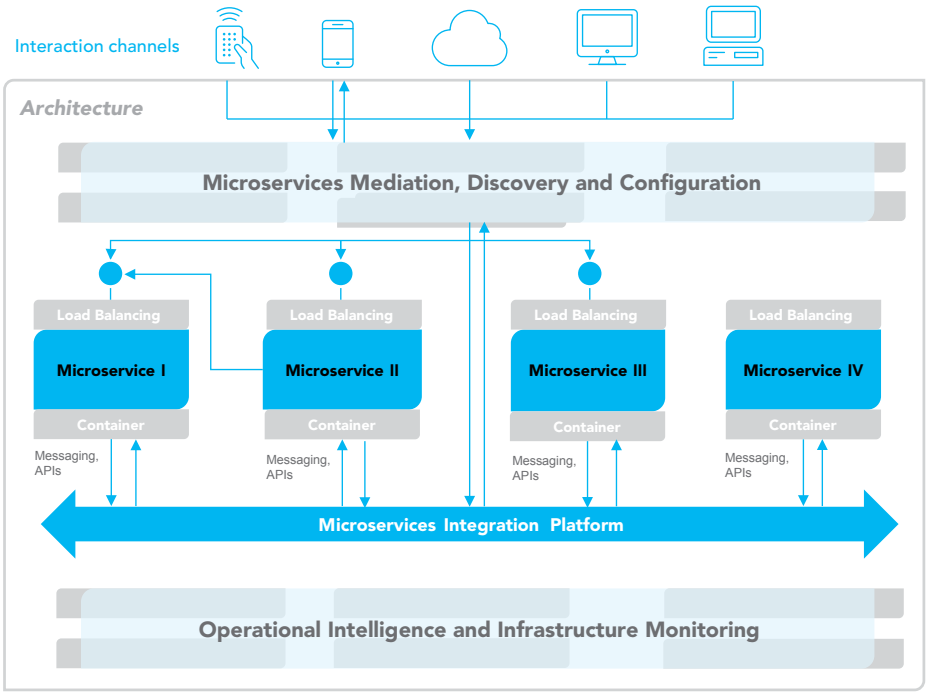
One popular way to deploy a microservices architecture is by means of using containers. A very popular container technology, Docker removes the need for IT operations professionals to be involved with software installation. Using Docker, a developer can prepare an image of any size in a container, and that container is immediately executable in an environment. Docker is highly effective in the on-premises data center as well as in cloud environments, but the cloud offers an endless supply of computing power on demand that is just a click away.

The ESB's changing role

The emergence, and re-emergence, of the ESB
The notion of an ESB evolved out of a need to take large monolithic applications and eliminate the point-to-point nature of how they interact with each other.

Fifteen years ago, enterprise applications were deployed in "stovepipes"—like the HR department, the finance department and the product development department. Eventually there was a need for these applications talk to each other. The concept of an ESB grew out of the search for a mechanism to take the applications that needed to talk to each other and provide a more common or efficient way to deliver interoperability and interconnectivity. The ESB further evolved into a way to extend these legacy applications with new capabilities. This was especially true for "shrink-wrapped" applications that a vendor might have to enhance for you.

Connectivity models have been around for a long time. There have been various "pre-ESB" forms of this enterprise interconnectivity model over the years. At the turn of the millennium, there were brokers like CORBA®, which was a hub-and-spokes type of architecture. All the interconnectivity resided in the center and spokes ran outward to connect various assets to that central place.



What emerged was a fleeting opportunity for the ESB to be part of a monolithic enterprise application. However, the school of thought prevailed that the true purpose of the ESB was always to remain a separate component and to wire together the large monolithic applications. It was the applications that took longer to realize the value of decoupling their functionality into smaller discrete and distributed modules. One reason was the cost of chopping up or refactoring the monolith that was not initially designed to be de-coupled. However, many new applications have been built using ESB technologies that provide a foundation for a much more agile architecture, and this trend will continue.

Today, applications have become increasingly large and have much higher scaling requirements. As a result, the very same concepts of an ESB—i.e., a governance-driven approach to integrating many different endpoints and creating many-to-many patterns—are now finding their way into application design. Due to that, the application itself and the ESB are becoming synonymous.

Instead of building an application that is monolithic (all self-contained) and trying to persuade the entire development team to coordinate on standards, programming practices and languages, the idea that the microservices architecture brings forward is to decompose everything and essentially apply the principles of what has been done with an ESB for integration to the application itself.

Does this evolution of the ESB concept mean the standalone ESB is unnecessary in a microservices-based architecture?

No, and here's why. As the large monolithic application gets decomposed into a microservices-based architecture or new applications are being developed, software designers and architects are slowly discovering they need a way to run and coordinate microservices by creating intelligent routing logic, data transformation, negotiating and mediating security profiles, for example. An ESB provides this gateway functionality. It is part of the plumbing, the abstraction and coordination of everything needed and part of the application itself. It is a serious part of the system's integrity. Reality is that by ignoring ESB, designers and architects actually start to build a proprietary version of an ESB out of microservices.

What is needed, in fact, is an intersection between the microservices approach and ESB-driven models—in other words, applying the concept of microservices in a loosely coupled, self-contained, fully distributed system with the capabilities found in the typical ESB. This makes microservices even more powerful with richer capabilities.

Where does the true cost reside in a microservices architecture?
The total cost of ownership is yet to be determined; time will reveal more insight. It is just as likely for a project to fail using microservices as any other style of architecture. Microservices, in practice, can lead unintentionally to the creation of a "Frankenstein" by misuse of too many variances in service interfaces, frameworks and redundant technologies. Without the right discipline, it is possible to create a new form of the "spaghetti" network and also end up with an overall footprint that might not be conducive to the business.

It is important to decompose a system into smaller services that are re-usable and self-contained, mediated and well managed. Similarly, microservices developers—or DevOps teams—will most likely realize that a fully decomposed design with too much granularity is not a panacea either, because it lacks the balance of a practical approach to deployment, monitoring, fault isolation, management and control.

Rather than simply re-inventing or replicating the ESB, developers are more likely to leverage the most useful aspects of an ESB inside the microservices architecture. The ESB will not vanish or be relegated to a strictly legacy support role within the enterprise, as some have suggested. Rather, it will take on a newly defined purpose.

Seven things to look for in microservices stack

Once the decision has been made to pursue a microservices approach, it is essential to find the right ingredients in a microservices-enabled technology stack or platform. Software AG recommends you look for:

1. Lightweight containers
Does the platform provide a lightweight, dynamic container? This is probably the top requirement. The typical application server in J2EE® style is too cumbersome and too heavy and offers too little value for this new category of applications.

2. Polyglot programming environment
Can the platform support a polyglot style environment—in other words, more than one programming model? The platform itself should never force particular programming model, style or language. Instead it should be "agnostic" and deliver multi-channel, multi-container capability wherein a single instance can host native code, .NET, Java and Spring™ Blueprint, etc.

3. Out-of-the-box capabilities
What out-of-the-box capabilities does this platform have to reduce implementation overhead? For example, how quickly can a messaging infrastructure be implemented? How costly is it to operate? How well integrated are the framework's capabilities? Is the platform capable of late binding between the interface and the service implementation?

4. Interoperability on security
How much standardization for interoperability on security is provided? Are there capabilities for certificate management? Are different styles of credentials supported within a given implementation and—very importantly—is that capability abstracted out of the service implementation? A big mistake is to select a technology in which security, interconnectivity and interoperability are baked into the service implementation itself. They must be separate.

5. Mediation and intelligent routing
Does the technology provide a mediation and intelligent routing layer? All microservices-based architectures end up having multiple instances of the same kind of functionality to support scaling. Therefore, one of the challenges is how to effectively route the right request to the right instance of a microservice, and how to mitigate and mediate that traffic. Mediation and intelligent routing address this problem. This kind of gateway functionality is very critical.

6. Hot "swap-ability"
Does the platform support hot-swapping of microservices? Getting a container provisioned at run-time is just one aspect. If microservices are replaced, deprecated or new services are added, how dynamic the changes are is another aspect to consider.

7. Monitoring and manageability
Considering all the benefits of microservices, one of the challenges is that microservices can be harder to manage, control and isolate faults due to the highly distributed nature of the architectural style. How will IT operations figure out where any problem resides? How will they monitor the system to discover what's failing, what's working well and what's not working well? How will they perform upgrades and corrections? Any technology platform, therefore, should include the ability to monitor an end-to-end implementation based on microservices in a way that doesn't require any instrumentation to be put into the microservice itself.

Five reasons to use the webMethods ESB

The right integration platform for microservices, especially when selected upfront, can help IT reduce long-term maintenance costs and also make the whole initiative a resounding success. Here are five key reasons to use the webMethods Integration Platform in a microservices environment:

1. The webMethods ESB is a lightweight container and will scale down, up or out depending on your needs. It is equally capable as a container in Point of Sales (POS) deployments or in massively horizontally scalable implementations.

2. The webMethods ESB has always been programming-model-agnostic and multi-container capable. From its very inception, it has been designed for the polyglot programming paradigm. This means the developer is free to use different programming languages as needed in a single solution. As a container, the webMethods ESB ships with hundreds of pre-built useful APIs and services, frameworks, and EDA and in-memory data grid support. The ESB supports the open standards, security standards and wire protocols needed by modern-day applications.

3. Services hosted in the ESB are "hot swappable" and can be changed safely while the container is still running. This feature makes it very relevant to the microservices architecture; as microservices change, the corresponding services on the ESB layer can be changed without needing to bring the server down.

4. The webMethods ESB may be deployed in a DMZ acting as a gateway, deployed inside the firewall acting as a gateway, or both. There is never a need to open inbound ports on the firewall, and there are no proxies to configure. The ESB provides routing and mediation services that are a key part of a microservices style architecture.

5. Microservices-style architectures require connectivity to a variety of data sources, storage technologies and applications. The webMethods ESB has a full range of connectors to support the most modern standards, such as OData, REST, and the full complement of SOAP-based Web-service support.

About the Author
Originally from Boston, Rob Tiberio has more than 25 years of experience in large-scale enterprise architecture, networks and communications, mainframes and open systems technologies. Early in his career, Rob helped develop technology for Arpanet, IBM® SNA, DECnet, Banyan, Novell and TCP/IP. At Digital Equipment Corporation, Raytheon and Computer Network Technology, he contributed to the founding of the Internet and developed fault-tolerant networking device drivers for Stratus Computer symmetrical parallel processing systems running FTX.

Rob joined webMethods (now Software AG) in 2001 as Director of R&D, BPMS Architecture. In 2009, he was named Senior Vice President and Chief Architect for Software AG. He is currently responsible for fostering a paradigm shift in Software AG's architecture and thought leadership on fully distributed computing, hybrid cloud architecture, EDA, big data and microservices, among other industry technologies and trends.

Next steps

As you look to build easy-to-adapt and maintain applications for your Digital Enterprise, consider the value of the webMethods ESB. It can be the ideal foundation for an agile microservices architecture, helping you reducing costs and more easily manage change. For more information, talk to your Software AG representative or visit www.softwareag.com/integration today.

wp_integration_api_evolution-of-microservices_en

software AG